// Demonstrate the front_inserter, back_inserter, ostream_iterator, 
// and functors. 
// George F. Riley, Georgia Tech, Fall 2009

#include <iostream>
#include <algorithm>
#include <vector>
#include <set>
#include <deque>
#include <iterator>

using namespace std;

// Some of the STL algorithms, such as "copy" take three iterators 
// as arguments; the first two define the begin() and end() of 
// a container, and the third defines an iterator to use to 
// insert elements into a new container. A possible implementation 
// is below:

template <typename InputIterator, typename OutputIterator>
void Copy(InputIterator b, InputIterator e, OutputIterator d)
{ // b is begin of input sequence, e is end of input 
  // d is destination iterator 
  cout << "Hello from Copy" << endl;
  while(b != e) 
  {
    *d++ = *b++;
  } 
}

// Generic subroutine to print a container 
template <class ForwardIterator>
void Print(ForwardIterator b, ForwardIterator e, bool addEndl = true)
{
  while(b != e) 
  {
    cout << (*b++);
    if (addEndl) cout << endl;
    else cout << " ";
  } 
}

// Now we look at some special objects called "functors".
// A functor is simply a class defines the "()" operator.
template <typename T> class Greater
{
  public:
    bool operator()(const T & lhs, const T & rhs) const { return lhs > rhs; } 
};

int main()
{
  // Create a short vector 
  vector<int> v;
  v.push_back(1);
  v.push_back(2);

  Program special-iterators.cc
v.push_back(3);
// Create an empty vector
vector<int> v1;
// Now we want to copy the contents of the first vector to the second
// using Copy. The problem is, what do we use for the third argument
// to Copy. One guess would be "v1.end()", indicating that we want
// to insert at the end of the new vector.
// Copy(v.begin(), v.end(), v1.end()); // Like this
// While this is exactly what we want, the code fails miserably.
// Recall that "end()" is an
// iterator that points "one beyond the end" of the vector, and in fact
// does not point to a valid element. Thus in Copy when we say "*d++",
// we are dereferencing an invalid iterator, with unpredictable results.
// We could try passing v1.begin() as the third argument.
// Copy(v.begin(), v.end(), v1.begin()); // Like this
// If v1 already had v.size() elements, this would work. But if
// v1 has fewer elements (which it does in this example), the
// code again crashes, since Copy advances the output iterator
// v.size() times which extends beyond the end of v1.
// Luckily, the designers of the STL anticipated our need, and designed
// two special iterators that handle this situation. They are the
// "back_insert_iterator" and the "front_insert_iterator"
back_insert_iterator<vector<int> > bi(v1);
// Note the extra space between the two > characters in the declaration
// above. This is necessary, otherwise the compiler would parse
// a right shift operator ">>" which not valid in this context.
// The variable "bi" is a back insert iterator, which has semantics
// exactly like normal iterators, except that if dereferenced at
// the end of a container, it adds the element to the container.
Copy(v.begin(), v.end(), bi);
cout << "v is "; Print(v.begin(), v.end(), false); cout << endl;
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << endl;
// We can also dereference and increment the back_insert_iterator
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << endl; 
*bi++ = 4;
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << endl; 
*bi++ = 5;
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << endl; 
// And we can dereference without incrementing (same results)
*bi++;
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size (no assignment) " << endl; 
*bi = 6;
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << endl; 
*bi = 7;
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << endl; 
*bi = 8;
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << "v1.size " << v1.size() << endl; 
// But we can’t decrement it (doesn’t make sense and doesn’t compile)
// *bi--; = 6;
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << endl;
There is also a special shortcut for creating the back_insert_iterator, called "back_inserter". back_inserter is simply a global function that has a single argument of any container, and returns a back_insert_iterator for that container. So instead of defining "bi" separately, we could simply say:

```cpp
Copy(v.begin(), v.end(), back_inserter(v1));
```

```cpp
cout << "v1 is "; Print(v1.begin(), v1.end(), false); cout << endl;
```

Similarly, we can define a "front_insert_iterator" that will add elements to the front (beginning) of a container when dereferenced. However, we have to be careful, since the container being inserted must have a "push_front" operation defined, which vectors do not. If we tried to define and use a front_insert_iterator on a vector container, the compiler would complain. We can however use a front insert iterator on the "deque" container, which supports both push_back and "pop_front".

```cpp
deque<int> d1;
```

```cpp
front_insert_iterator<deque<int> > fi(d1);
Copy(v.begin(), v.end(), fi);
```

```cpp
cout << "v is "; Print(v.begin(), v.end(), false); cout << endl;
```

```cpp
cout << "d1 is "; Print(d1.begin(), d1.end(), false); cout << endl;
```

Similarly, we can use the "front_inserter" shortcut:

```cpp
Copy(v.begin(), v.end(), front_inserter(d1));
```

```cpp
cout << "d1 is "; Print(d1.begin(), d1.end(), false); cout << endl;
```

It is often the case that we use iterators for a sequence to call an ostream function (for example cout <<) for each element. There is a nice shortcut to do this called "ostream_iterator"

```cpp
ostream_iterator<int> osi(cout, ", ");
```

By passing this as the destination iterator for Copy, we cause the cout << operator to be called for each element.

```cpp
Copy(v1.begin(), v1.end(), osi);
```

```cpp
Copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, "\n"));
```

Now demonstrate the use of functors

```cpp
Greater<int> greater;// Class Greater has one member function, the operator()
if (greater(1,2)) cout << "Huh? 1 greater than 2???"); << endl;
else cout << "1 not greater than 2" << endl;
```

The above is not very interesting, as it’s not clear what value the functors are. Recall the STL "set" container that stores objects in sorted order. The default comparator for set containers is the "operator <" function.

```cpp
set<int> s;
s.insert(100);
s.insert(10);
s.insert(500);
```

This stores in normal ascending order

```cpp
Copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "))); cout << endl;
```

But we can create a set specifying a non-default comparator,

```cpp
by passing the Greater functor:
set<int, Greater<int> > s1;
s1.insert(100);
s1.insert(10);
```
s1.insert(500);

Copy(s1.begin(), s1.end(), ostream_iterator<int>(cout, " ")); cout << endl;
}

Program special-iterators.cc (continued)