# The Standard Template Library

CHAPTER

15

## IN THIS CHAPTER

Most computer programs exist to process data. The data may represent a wide variety of real-world information: personnel records, inventories, text documents, the results of scientific experiments, and so on. Whatever it represents, data is stored in memory and manipulated in similar ways. University computer science programs typically include a course called "Data Structures and Algorithms." The term *data structures* refers to the ways data is stored in memory, and *algorithms* refers to how it is manipulated.

C++ classes provide an excellent mechanism for creating a library of data structures. In the past, compiler vendors and many third-party developers offered libraries of *container classes* to handle the storage and processing of data. Now, however, Standard C++ includes its own built-in container class library. It's called the Standard Template Library (STL), and was developed by Alexander Stepanov and Meng Lee of Hewlett Packard. The STL is part of the Standard C++ class library, and can be used as a standard approach to storing and processing data.

This chapter describes the STL and how to use it. The STL is large and complex, so we won't by any means describe everything about it; that would require a large book. (Many books are available on the STL; see Appendix H, "Bibliography.") We will introduce the STL and give examples of the more common algorithms and containers.

## Introduction to the STL

The STL contains several kinds of entities. The three most important are containers, algorithms, and iterators.

A *container* is a way that stored data is organized in memory. In earlier chapters we've explored two kinds of containers: stacks and linked lists. Another container, the array, is so common that it's built into C++ (and most other computer languages). However, there are many other kinds of containers, and the STL includes the most useful. The STL containers are implemented by template classes, so they can be easily customized to hold different kinds of data.

*Algorithms* in the STL are procedures that are applied to containers to process their data in various ways. For example, there are algorithms to sort, copy, search, and merge data. Algorithms are represented by template functions. These functions are not member functions of the container classes. Rather, they are standalone functions. Indeed, one of the striking characteristics of the STL is that its algorithms are so general. You can use them not only on STL containers, but on ordinary C++ arrays and on containers you create yourself. (Containers also include member functions for more specific tasks.)

*Iterators* are a generalization of the concept of pointers: they point to elements in a container. You can increment an iterator, as you can a pointer, so it points in turn to each element in a container. Iterators are a key part of the STL because they connect algorithms with containers.

Think of them as a software version of cables (like the cables that connect stereo components together or a computer to its peripherals).

Figure 15.1 shows these three main components of the STL. In this section we'll discuss containers, algorithms, and iterators in slightly more detail. In subsequent sections we'll explore these concepts further with program examples.

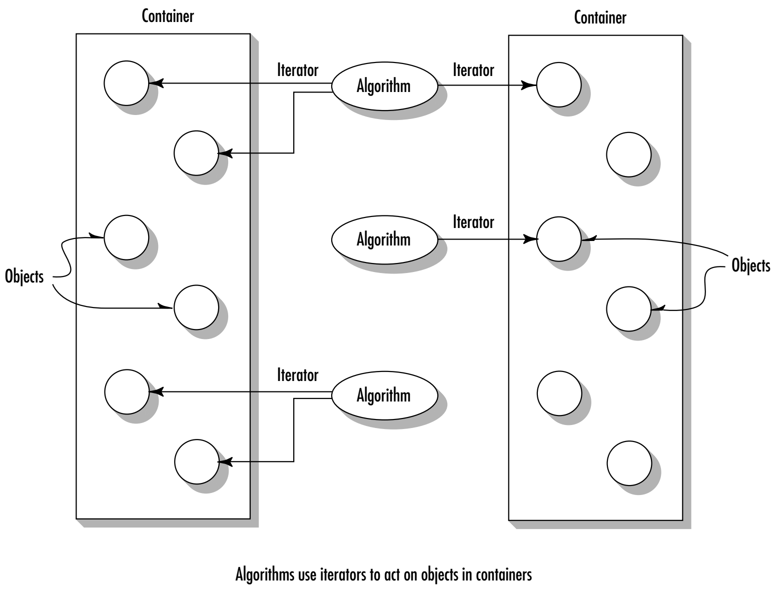

**FIGURE 15.1**
*Containers, algorithms, and iterators.*

## Containers

A container is a way to store data, whether the data consists of built-in types such as `int` and `float`, or of class objects. The STL makes seven basic kinds of containers available, as well as three more that are derived from the basic kinds. In addition, you can create your own containers based on the basic kinds. You may wonder why we need so many kinds of containers. Why not use C++ arrays in all data storage situations? The answer is efficiency. An array is awkward or slow in many situations.

Containers in the STL fall into two main categories: *sequence* and *associative*. The sequence containers are *vector*, *list*, and *deque*. The associative containers are *set*, *multiset*, *map*, and *multimap*. In addition, several specialized containers are derived from the sequence containers. These are *stack*, *queue*, and *priority queue*. We'll look at these categories in turn.

## Sequence Containers

A sequence container stores a set of elements in what you can visualize as a line, like houses on a street. Each element is related to the other elements by its position along the line. Each element (except at the ends) is preceded by one specific element and followed by another. An ordinary C++ array is an example of a sequence container.

One problem with a C++ array is that you must specify its size at compile time; that is, in the source code. Unfortunately, you usually don't know, when you write the program, how much data will be stored in the array. So you must specify an array large enough to hold what you guess is the maximum amount of data. When the program runs, you will either waste space in memory by not filling the array, or elicit an error message (or even blow up the program) by running out of space. The STL provides the *vector* container to avoid these difficulties.

Here's another problem with arrays. Say you're storing employee records, and you've arranged them in alphabetical order by the employee's last name. If you now want to insert a new employee whose name starts with L, you must move all the employees from M to Z to make room. This can be very time-consuming. The STL provides the *list* container, which is based on the idea of a linked list, to solve this problem. Recall from the LINKLIST example in Chapter 10, "Pointers," that it's easy to insert a new item in a linked list by rearranging several pointers.

The third sequence container is the *deque*, which can be thought of as a combination of a stack and a queue. A stack, as you may recall from previous examples, works on a last-in-first-out principle. Both input and output take place on the top of the stack. A queue, on the other hand, uses a first-in-first-out arrangement: data goes in at the front and comes out at the back, like a line of customers in a bank. A deque combines these approaches so you can insert or delete data from either end. The word deque is derived from Double-Ended QUEue. It's a versatile mechanism that's not only useful in its own right, but can be used as the basis for stacks and queues, as you'll see later.

Table 15.1 summarizes the characteristics of the STL sequence containers. It includes the ordinary C++ array for comparison.

**TABLE 15.1** Basic Sequence Containers

| *Container* | *Characteristic* | *Advantages and Disadvantages* |
|---|---|---|
| ordinary C++ array | Fixed size | Quick random access (by index number) |
| | | Slow to insert or erase in the middle |
| | | Size cannot be changed at runtime |
| vector | Relocating, expandable array | Quick random access (by index number) |
| | | Slow to insert or erase in the middle |
| | | Quick to insert or erase at end |

**TABLE 15.1** Continued

| | *Characteristic* | *Advantages and Disadvantages* |
|---|---|---|
| list | Doubly linked list | Quick to insert or delete at any location<br>Quick access to both ends<br>Slow random access |
| deque | Like vector, but can be accessed at either end | Quick random access (using index number)<br>Slow to insert or erase in the middle<br>Quick insert or erase (push and pop) at either the beginning or the end |

Instantiating an STL container object is easy. First you must include an appropriate header file. Then you use the template format with the kind of objects to be stored as the parameter. Examples might be

```
vector<int> aVect;  //create a vector of ints
```

or

```
list<airtime> departure_list;  //create a list of airtimes
```

Notice that there's no need to specify the size of STL containers. The containers themselves take care of all memory allocation.

## Associative Containers

An associative container is not sequential; instead it uses *keys* to access data. The keys, typically numbers or stings, are used automatically by the container to arrange the stored elements in a specific order. It's like an ordinary English dictionary, in which you access data by looking up words arranged in alphabetical order. You start with a key value (say the word *aardvark*, to use the dictionary example), and the container converts this key to the element's location in memory. If you know the key, you can access the associated value swiftly.

There are two kinds of associative containers in the STL: *sets* and *maps*. These both store data in a structure called a *tree*, which offers fast searching, insertion, and deletion. Sets and maps are thus very versatile general data structures suitable for a wide variety of applications. However, it is inefficient to sort them and perform other operations that require random access.

Sets are simpler and more commonly used than maps. A set stores a number of items which contain *keys*. The keys are the attributes used to order the items. For example, a set might store objects of the `person` class, which are ordered alphabetically using their `name` attributes as keys. In this situation, you can quickly locate a desired `person` object by searching for the object with a specified `name`. If a set stores values of a basic type such as `int`, the key is the entire item stored. Some writers refer to an entire object stored in a set as a *key*, but we'll call it the *key object* to emphasize that the attribute used to order it (the key) isn't necessarily the entire item.

A map stores pairs of objects: a key object and a value object. A map is often used as a container that's somewhat like an array, except that instead of accessing its elements with index numbers, you access them with indices that can be of an arbitrary type. That is, the key object serves as the index, and the value object is the value at that index.

The *map* and *set* containers allow only one key of a given value to be stored. This makes sense in, say, a list of employees arranged by unique employee numbers. On the other hand, the *multimap* and *multiset* containers allow multiple keys. In an English dictionary there might be several entries for the word "set," for example.

Table 15.2 summarizes the associative containers available in the STL.

**TABLE 15.2** Basic Associative Containers

| *Container* | *Characteristics* |
|---|---|
| set | Stores only the key objects<br>Only one key of each value allowed |
| multiset | Stores only the key objects<br>Multiple key values allowed |
| map | Associates key object with value object<br>Only one key of each value allowed |
| multimap | Associates key object with value object<br>Multiple key values allowed |

Creating associative containers is just like creating sequential ones:

```
set<int> intSet;  //create a set of ints
```

or

```
multiset<employee> machinists;  //create a multiset of employees
```

## Member Functions

Algorithms are the heavy hitters of the STL, carrying out complex operations like sorting and searching. However, containers also need member functions to perform simpler tasks that are specific to a particular type of container.

Table 15.3 shows some frequently-used member functions whose name and purpose (not the actual implementation) are common to most container classes.

**TABLE 15.3** Some Member Functions Common to All Containers

| *Name* | *Purpose* |
|---|---|
| `size()` | Returns the number of items in the container |
| `empty()` | Returns true if container is empty |
| `max_size()` | Returns size of the largest possible container |
| `begin()` | Returns an iterator to the start of the container, for iterating forwards through the container |
| `end()` | Returns an iterator to the past-the-end location in the container, used to end forward iteration |
| `rbegin()` | Returns a reverse iterator to the end of the container, for iterating backward through the container |
| `rend()` | Returns a reverse iterator to the beginning of the container; used to end backward iteration |

Many other member functions appear only in certain containers, or certain categories of containers. You'll learn more about these as we go along. Appendix F, "STL Algorithms and Member Functions," includes a table showing the STL member functions and which ones exist for which containers.

## Container Adapters

It's possible to create special-purpose containers from the normal containers mentioned previously using a construct called *container adapters*. These special-purpose containers have simpler interfaces than the more general containers. The specialized containers implemented with container adapters in the STL are *stacks*, *queues*, and *priority queues*. As we noted, a stack restricts access to pushing and popping a data item on and off the top of the stack. In a queue, you push items at one end and pop them off the other. In a priority queue, you push data in the front in random order, but when you pop the data off the other end, you always pop the *largest item* stored: the priority queue automatically sorts the data for you.

Stacks, queues, and priority queues can be created from different sequence containers, although the deque is often used. Table 15.4 shows the abstract data types and the sequence containers that can be used in their implementation.

**TABLE 15.4** Adapter-Based Containers

| *Container* | *Implementation* | *Characteristics* |
|---|---|---|
| stack | Can be implemented as vector, list, or deque | Insert (push) and remove (pop) at one end only |
| queue | Can be implemented as list or deque | Insert (push) at one end, remove (pop) at other |
| priority queue | Can be implemented as vector or deque | Insert (push) in random order at one end, remove (pop) in sorted order from other end |

You use a template within a template to instantiate these classes. For example, here's a stack object that holds type `int`, instantiated from the deque class:

```
stack< deque<int> > aStak;
```

A detail to note about this format is that you must insert a space between the two closing angle brackets. You can't write

```
stack<deque<int>> astak;  //syntax error
```

because the compiler will interpret the >> as an operator.

## Algorithms

An algorithm is a function that does something to the items in a container (or containers). As we noted, algorithms in the STL are not member functions or even friends of container classes, as they are in earlier container libraries, but are standalone template functions. You can use them with built-in C++ arrays, or with container classes you create yourself (provided the class includes certain basic functions).

Table 15.5 shows a few representative algorithms. We'll examine others as we go along. Appendix F contains a table listing most of the STL algorithms.

**TABLE 15.5** Some Typical STL Algorithms

| *Algorithm* | *Purpose* |
|---|---|
| `find` | Returns first element equivalent to a specified value |
| `count` | Counts the number of elements that have a specified value |
| `equal` | Compares the contents of two containers and returns true if all corresponding elements are equal |

**TABLE 15.5** Continued

| *Algorithm* | *Purpose* |
|---|---|
| `search` | Looks for a sequence of values in one container that corresponds with the same sequence in another container |
| `copy` | Copies a sequence of values from one container to another (or to a different location in the same container) |
| `swap` | Exchanges a value in one location with a value in another |
| `iter_swap` | Exchanges a sequence of values in one location with a sequence of values in another location |
| `fill` | Copies a value into a sequence of locations |
| `sort` | Sorts the values in a container according to a specified ordering |
| `merge` | Combines two sorted ranges of elements to make a larger sorted range |
| `accumulate` | Returns the sum of the elements in a given range |
| `for_each` | Executes a specified function for each element in the container |

Suppose you create an array of type `int`, with data in it:

```
int arr[8] = {42, 31, 7, 80, 2, 26, 19, 75};
```

You can then use the STL `sort()` algorithm to sort this array by saying

```
sort(arr, arr+8);
```

where `arr` is the address of the beginning of the array, and `arr+8` is the past-the-end address (one item past the end of the array).

## Iterators

Iterators are pointer-like entities that are used to access individual data items (which are usually called *elements*), in a container. Often they are used to move sequentially from element to element, a process called *iterating* through the container. You can increment iterators with the `++` operator so they point to the next element, and dereference them with the `*` operator to obtain the value of the element they point to. In the STL an iterator is represented by an object of an `iterator` class.

Different classes of iterators must be used with different types of container. There are three major classes of iterators: forward, bidirectional, and random access. A *forward iterator* can only move forward through the container, one item at a time. Its ++ operator accomplishes this. It can't move backward and it can't be set to an arbitrary location in the middle of the container. A *bidirectional iterator* can move backward as well as forward, so both its ++ and `--` operators are defined. A *random access iterator*, in addition to moving backward and forward, can jump to an arbitrary location. You can tell it to access location 27, for example.

There are also two specialized kinds of iterators. An *input iterator* can "point to" an input device (`cin` or a file) to read sequential data items into a container, and an *output iterator* can "point to" an output device (`cout` or a file) and write elements from a container to the device.

While the values of forward, bi-directional, and random access iterators can be stored (so they can be used later), the values of input and output iterators cannot be. This makes sense: the first three iterators point to memory locations, while input and output iterators point to I/O devices for which stored "pointer" values have no meaning. Table 15.6 shows the characteristics of these different kinds of iterators.

**TABLE 15.6** Iterator Characteristics

| *Iterator Type* | *Read/Write* | *Iterator Can Be Saved* | *Direction* | *Access* |
|---|---|---|---|---|
| Random access | Read and write | Yes | Forward and back | Random |
| Bidirectional | Read and write | Yes | Forward and back | Linear |
| Forward | Read and write | Yes | Forward only | Linear |
| Output | Write only | No | Forward only | Linear |
| Input | Read only | No | Forward only | Linear |

## Potential Problems with the STL

The sophistication of the STL's template classes places a strain on compilers, and not all of them respond well. Let's look at some potential problems.

First, it's sometimes hard to find errors because the compiler reports them as being deep in a header file when they're really in the class user's code. You may need to resort to brute force methods such as commenting out one line of your code at a time to find the culprit.

Precompilation of header files, which speeds up compilation dramatically on compilers that offer it, may cause problems with the STL. If things don't seem to be working, try turning off precompiled headers.

The STL may generate spurious compiler warnings. "Conversion may lose significant digits" is a favorite. These appear to be harmless, and can be ignored or turned off.

These minor complaints aside, the STL is a surprisingly robust and versatile system. Errors tend to be caught at compile time rather than at runtime. The different algorithms and containers present a very consistent interface; what works with one container or algorithm will usually work with another (assuming it's used appropriately).

This quick overview probably leaves you with more questions than answers. The balance of this chapter should provide enough specific details of STL operation to make things clearer.

## Algorithms

The STL algorithms perform operations on collections of data. These algorithms were designed to work with STL containers, but one of the nice things about them is that you can apply them to ordinary C++ arrays. This may save you considerable work when programming arrays. It also offers an easy way to learn about the algorithms, unencumbered with containers. In this section we'll examine how some representative algorithms are used. (Remember that the algorithms are listed in Appendix F.)

### The `find()` Algorithm

The `find()` algorithm looks for the first element in a container that has a specified value. The FIND example program shows how this looks when we're trying to find a value in an array of `int`s.

```
// find.cpp
// finds the first object with a specified value
#include <iostream>
#include <algorithm>                    //for find()
using namespace std;

int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };

int main()
   {
   int* ptr;
   ptr = find(arr, arr+8, 33);          //find first 33
   cout << "First object with value 33 found at offset "
        << (ptr-arr) << endl;
   return 0;
   }
```

The output from this program is

```
First object with value 33 found at offset 2.
```

As usual, the first element in the array is number 0, so the 33 is at offset 2, not 3.

#### Header Files

In this program we've included the header file ALGORITHM. Notice that, as with other header files in the Standard C++ library, there is no file extension (like .H). This file contains the declarations of the STL algorithms. Other header files are used for containers and for other purposes. If you're using an older version of the STL you may need to include a header file with a somewhat different name, like ALGO.H.

## Ranges

The first two parameters to `find()` specify the range of elements to be examined. These values are specified by iterators. In this example we use normal C++ pointer values, which are a special case of iterators.

The first parameter is the iterator of (or in this case the pointer to) the first value to be examined. The second parameter is the iterator of the location one past the last element to be examined. Since there are 8 elements, this value is the first value plus 8. This is called a *past-the-end* value; it points to the element just past the end of the range to be examined.

This syntax is reminiscent of the normal C++ idiom in a `for` loop:

```
for(int j=0; j<8; j++)   //from 0 to 7
   {
   if(arr[j] == 33)
      {
      cout << "First object with value 33 found at offset "
           << j << endl;
      break;
      }
   }
```

In the FIND example, the `find()` algorithm saves you the trouble of writing this `for` loop. In more complicated situations, algorithms may save you from writing far more complicated code.

## The `count()` Algorithm

Let's look at another algorithm, `count()`, which counts how many elements in a container have a specified value and returns this number. The COUNT example shows how this looks:

```
// count.cpp
// counts the number of objects with a specified value
#include <iostream>
#include <algorithm>              //for count()
using namespace std;

int arr[] = { 33, 22, 33, 44, 33, 55, 66, 77 };

int main()
   {
   int n = count(arr, arr+8, 33);  //count number of 33's
   cout << "There are " << n << " 33's in arr." << endl;
   return 0;
   }
```

The output is

```
There are 3 33's in arr.
```

## The sort() Algorithm

You can guess what the `sort()` algorithm does. Here's an example, called SORT, of this algorithm applied to an array:

```
// sort.cpp
// sorts an array of integers
#include <iostream>
#include <algorithm>
using namespace std;
                               //array of numbers
int arr[] = {45, 2, 22, -17, 0, -30, 25, 55};

int main()
   {
   sort(arr, arr+8);           //sort the numbers

   for(int j=0; j<8; j++)      //display sorted array
      cout << arr[j] << ' ';
   cout << endl;
   return 0;
   }
```

The output from the program is

```
-30, -17, 0, 2, 22, 25, 45, 55
```

We'll look at some variations of this algorithm later.

## The search() Algorithm

Some algorithms operate on two containers at once. For instance, while the `find()` algorithm looks for a specified value in a single container, the `search()` algorithm looks for a sequence of values, specified by one container, within another container. The SEARCH example shows how this looks.

```
// search.cpp
// searches one container for a sequence in another container
#include <iostream>
#include <algorithm>
using namespace std;

int source[] =  { 11, 44, 33, 11, 22, 33, 11, 22, 44 };
int pattern[] = { 11, 22, 33 };

int main()
   {
   int* ptr;
   ptr = search(source, source+9, pattern, pattern+3);
   if(ptr == source+9)                //if past-the-end
      cout << "No match found\n";
   else
      cout << "Match at " << (ptr - source) << endl;
   return 0;
}
```

The algorithm looks for the sequence 11, 22, 33, specified by the array `pattern`, within the array `source`. As you can see by inspection, this sequence is found in `source` starting at the fourth element (element 3). The output is

```
Match at 3
```

If the iterator value `ptr` ends up one past the end of the source, no match has been found.

The arguments to algorithms such as `search()` don't need to be the same type of container. The source could be in an STL vector, and the pattern in an array, for example. This kind of generality is a very powerful feature of the STL.

## The `merge()` Algorithm

Here's an algorithm that works with three containers, merging the elements from two source containers into a destination container. The MERGE example shows how it works.

```
// merge.cpp
// merges two containers into a third
#include <iostream>
#include <algorithm>          //for merge()
using namespace std;

int src1[] = { 2, 3, 4, 6, 8 };
int src2[] = { 1, 3, 5 };
int dest[8];

int main()
   {                          //merge src1 and src2 into dest
   merge(src1, src1+5, src2, src2+3, dest);
   for(int j=0; j<8; j++)     //display dest
      cout << dest[j] << ' ';
   cout << endl;
   return 0;
   }
}
```

The output, which displays the contents of the destination container, looks like this:

```
1 2 3 3 4 5 6 8
```

As you can see, merging preserves the ordering, interweaving the two sequences of source elements into the destination container.

## Function Objects

Some algorithms can take something called a *function object* as an argument. A function object looks, to the user, much like a template function. However, it's actually an object of a template class that has a single member function: the overloaded `()` operator. This sounds mysterious, but it's easy to use.

Suppose you want to sort an array of numbers into descending instead of ascending order. The SORTEMP program shows how to do it:

```
// sortemp.cpp
// sorts array of doubles in backward order,
// uses greater<>() function object
#include <iostream>
#include <algorithm>                    //for sort()
#include <functional>                   //for greater<>
using namespace std;
                                        //array of doubles
double fdata[] = { 19.2, 87.4, 33.6, 55.0, 11.5, 42.2 };

int main()
   {                                    //sort the doubles
   sort( fdata, fdata+6, greater<double>() );

   for(int j=0; j<6; j++)               //display sorted doubles
      cout << fdata[j] << ' ';
   cout << endl;
   return 0;
}
```

The `sort()` algorithm usually sorts in ascending order, but the use of the `greater<>()` function object, the third argument of `sort()`, reverses the sorting order. Here's the output:

```
87.4 55 42.2 33.6 19.2 11.5
```

Besides comparisons, there are function objects for arithmetical and logical operations. We'll look at function objects more closely in the last section of this chapter.

### User-Written Functions in Place of Function Objects

Function objects operate only on basic C++ types and on classes for which the appropriate operators (+, <, ==, and so on) are defined. If you're working with values for which this is not the case, you can substitute a user-written function for a function object. For example, the operator < is not defined for ordinary `char*` strings, but we can write a function to perform the comparison, and use this function's address (its name) in place of the function object. The SORTCOM example shows how to sort an array of `char*` strings:

```
// sortcom.cpp
// sorts array of strings with user-written comparison function
#include <iostream>
#include <string>                           //for strcmp()
#include <algorithm>
using namespace std;
                                            //array of strings
char* names[] = { "George", "Penny", "Estelle",
                  "Don", "Mike", "Bob" };

bool alpha_comp(char*, char*);              //declaration

int main()
   {
   sort(names, names+6, alpha_comp);        //sort the strings

   for(int j=0; j<6; j++)                   //display sorted strings
      cout << names[j] << endl;
   return 0;
   }

bool alpha_comp(char* s1, char* s2)         //returns true if s1<s2
   {
   return ( strcmp(s1, s2)<0 ) ? true : false;
   }
```

The third argument to the `sort()` algorithm is the address of the `alpha_comp()` function, which compares two `char*` strings and returns `true` or `false`, depending on whether the first is lexicographically (that is, alphabetically) less than the second. It uses the C library function `strcmp()`, which returns a value less than 0 if its first argument is less than its second. The output from this program is what you would expect:

```
Bob
Don
Estelle
George
Mike
Penny
```

Actually you don't need to write your own function objects to handle text. If you use the `string` class from the standard library, you can use built-in function objects such as `less<>()` and `greater<>()`.

## Adding `_if` to Algorithms

Some algorithms have versions that end in `_if`. These algorithms take an extra parameter called a *predicate*, which is a function object or a function. For example, the `find()` algorithm finds all elements equal to a specified value. We can also create a function that works with the `find_if()` algorithm to find elements with any arbitrary characteristic.

Our example uses `string` objects. The `find_if()` algorithm is supplied with a user-written `isDon()` function to find the first `string` in an array of `string` objects that has the value "Don". Here's the listing for FIND_IF:

```
// find_if.cpp
// searches array of strings for first name that matches "Don"
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
//---------------------------------------------------------------
bool isDon(string name)      //returns true if name=="Don"
   {
   return name == "Don";
   }
//---------------------------------------------------------------
string names[] = { "George", "Estelle", "Don", "Mike", "Bob" };

int main()
   {
   string* ptr;
   ptr = find_if( names, names+5, isDon );

   if(ptr==names+5)
      cout << "Don is not on the list.\n";
   else
      cout << "Don is element "
           << (ptr-names)
           << " on the list.\n";
   return 0;
   }
```

Since "Don" is indeed one of the names in the array, the output from the program is

```
Don is element 2 on the list.
```

The address of the function `isDon()` is the third argument to `find_if()`, while the first and second arguments are, as usual, the first and the past-the-end addresses of the array.

The `find_if()` algorithm applies the `isDon()` function to every element in the range. If `isDon()` returns true for any element, then `find_if()` returns the value of that element's pointer (iterator). Otherwise, it returns a pointer to the past-the-end address of the array.

Various other algorithms, such as `count()`, `replace()`, and `remove()`, have `_if` versions.

## The `for_each()` Algorithm

The `for_each()` algorithm allows you to do something to every item in a container. You write your own function to determine what that "something" is. Your function can't change the elements in the container, but it can use or display their values.

Here's an example in which `for_each()` is used to convert all the values of an array from inches to centimeters and display them. We write a function called `in_to_cm()` that multiplies a value by 2.54, and use this function's address as the third argument to `for_each()`. Here's the listing for FOR_EACH:

```
// for_each.cpp
// uses for_each() to output inches array elements as centimeters
#include <iostream>
#include <algorithm>
using namespace std;

void in_to_cm(double);     //declaration

int main()
   {                       //array of inches values
   double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
                           //output as centimeters
   for_each(inches, inches+5, in_to_cm);
   cout << endl;
   return 0;
   }

void in_to_cm(double in)   //convert and display as centimeters
   {
   cout << (in * 2.54) << ' ';
   }
```

The output looks like this:

```
8.89 15.748 2.54 32.385 10.9982
```

## The `transform()` Algorithm

The `transform()` algorithm does something to every item in a container, and places the resulting values in a different container (or the same one). Again, a user-written function determines what will be done to each item. The return type of this function must be the same as that of the destination container. Our example is similar to FOR_EACH, except that instead of displaying the converted values, our `in_to_cm()` function puts the centimeter values into a different array, `centi[]`. The main program then displays the contents of `centi[]`. Here's the listing for TRANSFO:

```
// transfo.cpp
// uses transform() to change array of inches values to cm
```

```
#include <iostream>
#include <algorithm>
using namespace std;

int main()
   {                           //array of inches values
   double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
   double centi[5];
   double in_to_cm(double);   //prototype
                              //transform into array centi[]
   transform(inches, inches+5, centi, in_to_cm);

   for(int j=0; j<5; j++)     //display array centi[]
      cout << centi[j] << ' ';
   cout << endl;
   return 0;
   }

double in_to_cm(double in)    //convert inches to centimeters
   {
   return (in * 2.54);        //return result
   }
```

The output is the same as that from the FOR_EACH program.

We've looked at just a few of the algorithms in the STL. There are many others, but what we've shown here should give you an idea of the kinds of algorithms that are available, and how to use them.

## Sequence Containers

As we noted earlier, there are two major categories of containers in the STL: sequence containers and associative containers. In this section we'll discuss the three sequence containers (vectors, lists, and deques), focusing on how these containers work and on their member functions. We haven't learned about iterators yet, so there will be some operations that we can't perform on these containers. We'll examine iterators in the next section.

Each program example in the following sections will introduce several member functions for the container being described. Remember, however, that different kinds of containers use member functions with the same names and characteristics, so what you learn about, say, `push_back()` for vectors will also be relevant to lists and queues.

### Vectors

You can think of vectors as smart arrays. They manage storage allocation for you, expanding and contracting the size of the vector as you insert or erase data. You can use vectors much like arrays, accessing elements with the `[]` operator. Such random access is very fast with vectors.

It’s also fast to add (or *push*) a new data item onto the end (the *back*) of the vector. When this happens, the vector’s size is automatically increased to hold the new item.

## Member Functions `push_back()`, `size()`, and `operator[]`

Our first example, VECTOR, shows the most common vector operations.

```
// vector.cpp
// demonstrates push_back(), operator[], size()
#include <iostream>
#include <vector>
using namespace std;

int main()
    {
    vector<int> v;                  //create a vector of ints

    v.push_back(10);                //put values at end of array
    v.push_back(11);
    v.push_back(12);
    v.push_back(13);

    v[0] = 20;                      //replace with new values
    v[3] = 23;

    for(int j=0; j<v.size(); j++)  //display vector contents
      cout << v[j] << ' ';          //20 11 12 23
   cout << endl;
   return 0;
    }
```

We use the vector’s default (no-argument) constructor to create a vector `v`. As with all STL containers, the template format is used to specify the type of variable the container will hold (in this case type `int`). We don’t specify the container’s size, so it starts off at 0.

The `push_back()` member function inserts the value of its argument at the back of the vector. (The back is where the element with the highest index number is.) The front of a vector (the element with index 0), unlike that of a list or queue, cannot be used for inserting new elements. Here we push the values 10, 11, 12, and 13, so that `v[0]` contains 10, `v[1]` contains 11, `v[2]` contains 12, and `v[3]` contains 13.

Once a vector has some data in it, this data can be accessed—both read and written to—using the overloaded `[]` operator, just as if it were in an array. We use this operator to change the first element from 10 to 20, and the last element from 13 to 23. Here’s the output from VECTOR:

```
20 11 12 23
```

The `size()` member function returns the number of elements currently in the container, which in VECTOR is 4. We use this value in the `for` loop to print out the values of the elements in the container.

Another member function, max_size() (which we don't demonstrate here), returns the maximum size to which a container can be expanded. This number depends on the type of data being stored in the container (the bigger the elements, the fewer of them you can store), the type of container, and the operating system. For example, on our system max_size() returns 1,073,741,823 for a vector type int.

## Member Functions swap(), empty(), back(), and pop_back()

The next example, VECTCON, shows some additional vector constructors and member functions.

```
// vectcon.cpp
// demonstrates constructors, swap(), empty(), back(), pop_back()
#include <iostream>
#include <vector>
using namespace std;

int main()
    {                                //an array of doubles
    double arr[] = { 1.1, 2.2, 3.3, 4.4 };

    vector<double> v1(arr, arr+4); //initialize vector to array
    vector<double> v2(4);          //empty vector of size 4

   v1.swap(v2);                    //swap contents of v1 and v2

    while( !v2.empty() )           //until vector is empty,
      {
      cout << v2.back() << ' ';    //display the last element
      v2.pop_back();               //remove the last element
      }                            //output: 4.4 3.3 2.2 1.1
   cout << endl;
   return 0;
    }
```

We've used two new vector constructors in this program. The first initializes the vector v1 with the values of a normal C++ array passed to it as an argument. The arguments to this constructor are pointers to the start of the array and to the element one past the end. The second constructor sets v2 to an initial size of 4, but does not supply any initial values. Both vectors hold type double.

The swap() member function exchanges all the data in one vector with all the data in another, keeping the elements in the same order. In this program there is only garbage data in v2, so it's swapped with the data in v1. We display v2 to show that it now contains the data that was in v1. The output is

```
4.4, 3.3, 2.2, 1.1
```

The back() member function returns the value of the last element in the vector. We display this value with cout. The pop_back() member function removes the last element in the vector.

Thus each time through the loop there is a different last element. (It's a little surprising that `pop_back()` does not simultaneously return the value of the last element and remove it from the vector, as we've seen `pop()` do in previous examples with stacks, but it doesn't, so `back()` must be used as well.)

Some member functions, such as `swap()`, also exist as algorithms. When this is the case, the member function version is usually provided because it's more efficient for that particular container than the algorithm version. Sometimes you can use the algorithm as well. For example, you can use it to swap elements in two different kinds of containers.

## Member Functions `insert()` and `erase()`

The `insert()` and `erase()` member functions insert or remove an element from an arbitrary location in a container. These functions aren't very efficient with vectors, since all the elements above the insertion or erasure must be moved to make space for the new element or close up the space where the erased item was. However, insertion and erasure may nevertheless be useful if speed is not a factor. The next example, VECTINS, shows how these member functions are used:

```
// vectins.cpp
// demonstrates insert(), erase()
#include <iostream>
#include <vector>
using namespace std;

int main()
    {
    int arr[] = { 100, 110, 120, 130 };  //an array of ints

    vector<int> v(arr, arr+4);       //initialize vector to array

   cout << "\nBefore insertion: ";
   for(int j=0; j<v.size(); j++)         //display all elements
      cout << v[j] << ' ';

   v.insert( v.begin()+2, 115);          //insert 115 at element 2

   cout << "\nAfter insertion:  ";
   for(j=0; j<v.size(); j++)             //display all elements
       cout << v[j] << ' ';

   v.erase( v.begin()+2 );               //erase element 2

   cout << "\nAfter erasure:    ";
   for(j=0; j<v.size(); j++)             //display all elements
       cout << v[j] << ' ';
   cout << endl;
   return 0;
    }
```

The `insert()` member function (at least this version of it) takes two arguments: the place where an element will be inserted in a container, and the value of the element. We add 2 to the `begin()` member function to specify element 2 (the third element) in the vector. The elements from the insertion point to the end of the container are moved upward to make room, and the size of the container is increased by 1.

The `erase()` member function removes the element at the specified location. The elements above the deletion point are moved downward, and the size of the container is decreased by 1. Here's the output from VECTINS:

```
Before insertion: 100 110 120 130
After insertion:  100 110 115 120 130
After erasure:    100 110 120 130
```

## Lists

An STL list container is a doubly linked list, in which each element contains a pointer not only to the next element but also to the preceding one. The container stores the address of both the front (first) and the back (last) elements, which makes for fast access to both ends of the list.

### Member Functions `push_front()`, `front()`, and `pop_front`

Our first example, LIST, shows how data can be pushed, read, and popped from both the front and the back.

```
//list.cpp
//demonstrates push_front(), front(), pop_front()
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   list<int> ilist;

   ilist.push_back(30);               //push items on back
   ilist.push_back(40);
   ilist.push_front(20);              //push items on front
   ilist.push_front(10);

   int size = ilist.size();           //number of items

   for(int j=0; j<size; j++)
      {
      cout << ilist.front() << ' ';   //read item from front
      ilist.pop_front();             //pop item off front
      }
   cout << endl;
   return 0;
   }
```

We push data on the back (the end) and front of the list in such a way that when we display and remove the data from the front it's in normal order:

```
10 20 30 40
```

The `push_front()`, `pop_front()`, and `front()` member functions are similar to `push_back()`, `pop_back()`, and `back()`, which we've already seen at work with vectors.

Note that you can't use random access for list elements, because such access is too slow. For this reason the `[]` operator is not defined for lists. If it were, this operator would need to traverse the list, counting elements as it went, until it reached the correct one, a time-consuming operation. If you need random access, you should use a vector or a deque.

Lists are appropriate when you will make frequent insertions and deletions in the middle of the list. This is not efficient for vectors and deques, because all the elements above the insertion or deletion point must be moved. However, it's quick for lists because only a few pointers need to be changed to insert or delete a new item. (However, it may still be time-consuming to find the correct insertion point.)

The `insert()` and `erase()` member functions are used for list insertion and deletion, but they require the use of iterators, so we'll postpone a discussion of these functions.

## Member Functions `reverse()`, `merge()`, and `unique()`

Some member functions exist only for lists; no such member functions are defined for other containers, although there are algorithms that do the same things. Our next example, LISTPLUS, shows some of these functions. It begins by filling two list-of-`int` objects with the contents of two arrays.

```
// listplus.cpp
// demonstrates reverse(), merge(), and unique()
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   int j;
   list<int> list1, list2;

   int arr1[] = { 40, 30, 20, 10 };
   int arr2[] = { 15, 20, 25, 30, 35 };
   for(j=0; j<4; j++)
      list1.push_back( arr1[j] );    //list1: 40, 30, 20, 10
   for(j=0; j<5; j++)
      list2.push_back( arr2[j] );    //list2: 15, 20, 25, 30, 35

   list1.reverse();                  //reverse list1: 10 20 30 40
   list1.merge(list2);               //merge list2 into list1
   list1.unique();                   //remove duplicate 20 and 30

   int size = list1.size();
   while( !list1.empty() )
      {
      cout << list1.front() << ' ';  //read item from front
      list1.pop_front();             //pop item off front
      }
   cout << endl;
   return 0;
   }
```

The first list is in backward order, so we return it to normal sorted order using the `reverse()` member function. (It's quick to reverse a list container because both ends are accessible.) This is necessary because the second member function, `merge()`, operates on two lists and requires both of them to be in sorted order. Following the reversal, the two lists are

```
10, 20, 30, 40
15, 20, 25, 30, 35
```

Now the `merge()` function merges `list2` into `list1`, keeping everything sorted and expanding `list1` to hold the new items. The resulting content of `list1` is

```
10, 15, 20, 20, 25, 30, 30, 35, 40
```

Finally we apply the `unique()` member function to `list1`. This function finds adjacent elements with the same value, and removes all but the first. The contents of `list1` are then displayed. The output of LISTPLUS is

```
10, 15, 20, 25, 30, 35, 40
```

To display the contents of the list we use the `front()` and `pop_front()` member functions in a `for` loop. Each element, from front to back, is displayed and then popped off the list. The result is that the process of displaying the list destroys it. This may not always be what you want, but for the moment it's the only way we have learned to access successive list elements. Iterators, described in the next section, will solve this problem.

## Deques

A deque is like a vector in some ways and like a linked list in others. Like a vector, it supports random access using the `[]` operator. However, like a list, a deque can be accessed at the front as well as the back. It's a sort of double-ended vector, supporting `push_front()`, `pop_front()`, and `front()`.

Memory is allocated differently for vectors and queues. A vector always occupies a contiguous region of memory. If a vector grows too large, it may need to be moved to a new location where it will fit. A deque, on the other hand, can be stored in several non-contiguous areas; it is segmented. A member function, `capacity()`, returns the largest number of elements a vector can store without being moved, but `capacity()` isn't defined for deques because they don't need to be moved.

```
// deque.cpp
// demonstrates push_back(), push_front(), front()
#include <iostream>
#include <deque>
using namespace std;

int main()
   {
   deque<int> deq;

   deq.push_back(30);              //push items on back
   deq.push_back(40);
   deq.push_back(50);
   deq.push_front(20);             //push items on front
   deq.push_front(10);

   deq[2] = 33;                    //change middle item

   for(int j=0; j<deq.size(); j++)
      cout << deq[j] << ' ';       //display items
   cout << endl;
   return 0;
   }
```

We've already seen examples of `push_back()`, `push_front()`, and `operator []`. They work the same for deques as for other containers. The output of this program is

```
10 20 33 40 50
```

Figure 15.2 shows some important member functions for the three sequential containers.

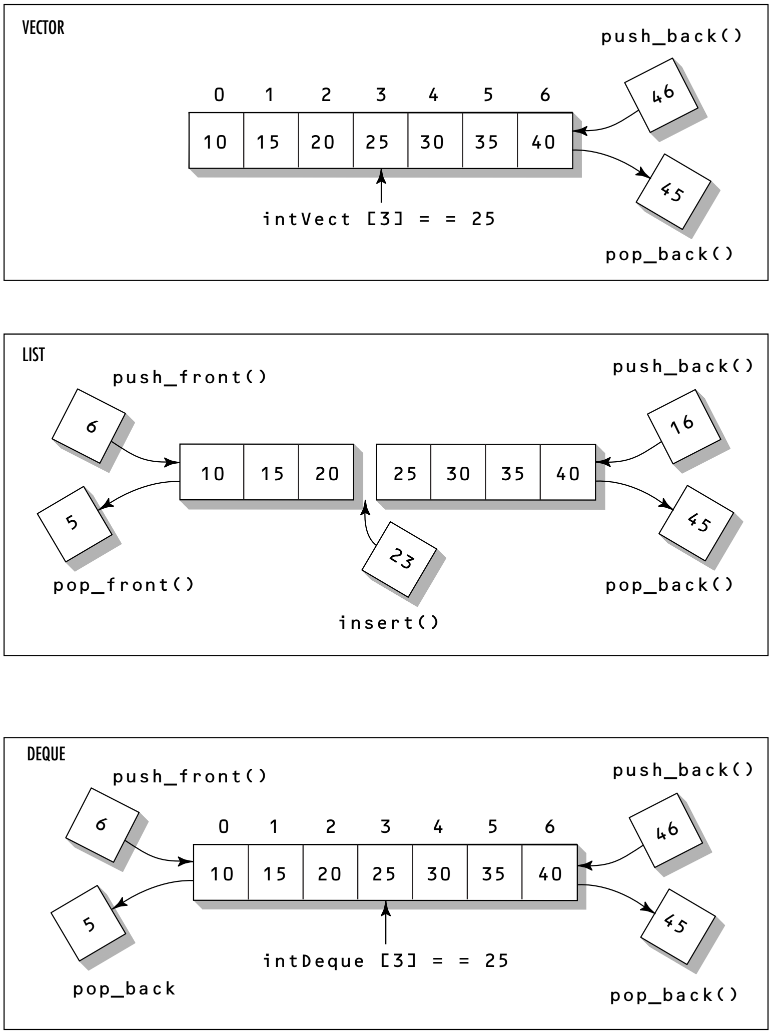

**FIGURE 15.2**
*Sequence containers.*

## Iterators

Iterators may seem a bit mysterious, yet they are central to the operation of the STL. In this section we'll first discuss the twin roles played by iterators: as smart pointers and as a connection between algorithms and containers. Then we'll show some examples of their use.

## Iterators as Smart Pointers

It's often necessary to perform an operation on all the elements in the container (or perhaps a range of elements). Displaying the value of each element in the container or adding its value to a total are examples. In an ordinary C++ array, such operations are carried out using a pointer (or the `[]` operator, which is the same underlying mechanism). For example, the following code iterates through a float array, displaying the value of each element:

```
float* ptr = start_address;
for(int j=0; j<SIZE; j++)
   cout << *ptr++;
```

We dereference the pointer `ptr` with the `*` operator to obtain the value of the item it points to, and increment it with the `++` operator so it points to the next item.

### Ordinary Pointers Underpowered

However, with more sophisticated containers, plain C++ pointers have disadvantages. For one thing, if the items stored in the container are not placed contiguously in memory, handling the pointer becomes much more complicated; we can't simply increment it to point to the next value. For example, in moving to the next item in a linked list we can't assume the item is adjacent to the previous one; we must follow the chain of pointers.

We may also want to store the address of some container element in a pointer variable so we can access the element at some future time. What happens to this stored pointer value if we insert or erase something from the middle of the container? It may not continue to be valid if the container's contents are rearranged. It would be nice if we didn't need to worry about revising all our stored pointer values when insertions and deletions take place.

One solution to these kinds of problems is to create a class of "smart pointers." An object of such a class basically wraps its member functions around an ordinary pointer. The `++` and `*` operators are overloaded so they know how to operate on the elements in their container, even if the elements are not contiguous in memory or change their locations. Here's how that might look, in skeleton form:

```
class SmartPointer
   {
   private:
      float* p;   //an ordinary pointer
   public:
      float operator*()
         {  }
      float operator++()
         {  }
   };
```

```
void main()
   {
   ...
   SmartPointer sptr = start_address;
   for(int j=0; j<SIZE; j++)
      cout << *sptr++;
   }
```

### Whose Responsibility?

Should the smart pointer class be embedded in a container, or should it be a separate class? The approach chosen by the STL is to make smart pointers, called *iterators*, into a completely separate class (actually a family of templetized classes). The class user creates iterators by defining them to be objects of such classes.

## Iterators as an Interface

Besides acting as smart pointers to items in containers, iterators serve another important purpose in the STL. They determine which algorithms can be used with which containers. Why is this necessary?

In some theoretical sense you should be able to apply every algorithm to every container. And, in fact, many algorithms will work with all the STL containers. However, it turns out that some algorithms are very inefficient (that is, slow) when used with some containers. The `sort()` algorithm, for example, needs random access to the container it's trying to sort; otherwise, it would need to iterate through the container to find each element before moving it, a time-consuming approach. Similarly, to be efficient, the `reverse()` algorithm needs to iterate backward as well as forward through a container.

Iterators provide a surprisingly elegant way to match appropriate algorithms with containers. As we noted, you can think of an iterator as a cable, like the cable used to connect a computer and printer. One end of the cable plugs into a container, and the other plugs into an algorithm. However, not all cables plug into all containers, and not all cables plug into all algorithms. If you try to use an algorithm that's too powerful for a given container type, you won't be able to find a cable (an iterator) to connect them. If you try it, you will receive a compiler error alerting you to the problem.

How many kinds of iterators (cables) do you need to make this scheme work? As it turns out, only five types are necessary. Figure 15.3 shows these five categories, arranged from bottom to top in order of increasing sophistication (input and output are equally unsophisticated).

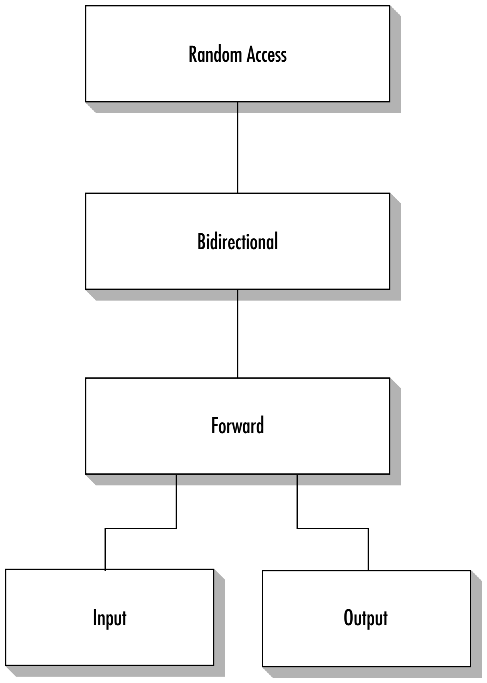

**FIGURE 15.3**
*Iterator categories.*

If an algorithm needs only to step forward through a container, reading (but not writing to) one item after another, it can use an *input* iterator to connect itself to the container. Actually, input iterators are typically used, not with containers, but when reading from files or `cin`.

If an algorithm steps through the container in a forward direction but writes to the container instead of reading from it, it can use an *output* iterator. Output iterators are typically used when writing to files or `cout`.

If an algorithm steps along forward and may either read from or write to a container, it must use a *forward* iterator.

If an algorithm must be able to step both forward and back through a container, it must use a *bidirectional* iterator.

Finally, if an algorithm must access any item in the container instantly, without stepping along to it, it must use a *random access* iterator. Random access iterators are like arrays, in that you can access any element. They are the only iterators that can be manipulated with arithmetic operations, as in

```
iter2 = iter1 + 7;
```

Table 15.7 shows which operations each iterator supports.

**TABLE 15.7** Capabilities of Different Iterator Categories

| *Iterator Type* | *Step Forward* `++` | *Read* `value=*i` | *Write* `*i=value` | *Step Back* `--` | *Random Access* *[n]* |
|---|---|---|---|---|---|
| Random access iterator | x | x | x | x | x |
| Bidirectional iterator | x | x | x | x | |
| Forward iterator | x | x | x | | |
| Output iterator | x | | x | | |
| Input iterator | x | x | | | |

As you can see, all the iterators support the `++` operator for stepping forward through the container. The input iterator can use the `*` operator on the right side of the equal sign (but not on the left):

```
 value = *iter;
```

The output iterator can use the `*` operator only on the right:

```
*iter = value;
```

The forward iterator handles both reading and writing, and the bidirectional iterator can be decremented as well as incremented. The random access iterator can use the `[]` operator (as well as simple arithmetic operators such as `+` and `-`) to access any element quickly.

An algorithm can always use an iterator with *more* capability than it needs. If it needs a forward iterator, for example, it's all right to plug it into a bidirectional iterator or a random access iterator.

## Matching Algorithms with Containers

We've used a cable as an analogy to an iterator, because an iterator connects an algorithm and a container. Let's focus on the two ends of this imaginary cable: the container end and the algorithm end.

## Plugging the Cable into a Container

If you confine yourself to the basic STL containers, you will be using only two kinds of iterators. As shown in Table 15.8, the vector and deque accept any kind of iterator, while the list, set, multiset, map, and multimap accept anything except the random iterator.

**TABLE 15.8** Iterator Types Accepted by Containers

| | *Vector* | *List* | *Deque* | *Set* | *Multiset* | *Map* | *Multimap* |
|---|---|---|---|---|---|---|---|
| Random Access | x | | x | | | | |
| Bidirectional | x | x | x | x | x | x | x |
| Forward | x | x | x | x | x | x | x |
| Input | x | x | x | x | x | x | x |
| Output | x | x | x | x | x | x | x |

How does the STL enforce the use of the correct iterator for a given container? When you define an iterator you must specify what kind of container it will be used for. For example, if you've defined a list holding elements of type `int`

```
list<int> iList;           //list of ints
```

then to define an iterator to this list you say

```
list<int>::iterator iter;  //iterator to list-of-ints
```

When you do this, the STL automatically makes this iterator a bidirectional iterator, because that's what a list requires. An iterator to a vector or a deque is automatically created as a random-access iterator.

This automatic selection process is implemented by causing an iterator class for a specific container to be derived (inherited) from a more general iterator class that's appropriate to a specific container. Thus the iterators to vectors and deques are derived from the `random_access_iterator` class, while iterators to lists are derived from the `bidirectional_iterator` class.

We now see how containers are matched to their end of our fanciful iterator cables. A cable doesn't actually plug into a container; it is (figuratively speaking) hardwired to it, like the cord on a toaster. Vectors and deques are always wired to random-access cables, while lists (and all the associative containers, which we'll encounter later in this chapter) are always wired to bidirectional cables.

## Plugging the Cable into the Algorithm

Now that we've seen how one end of an iterator cable is "wired" to the container, we're ready to look at the other end of the cable. How do iterators plug into algorithms? Every algorithm, depending on what it will do to the elements in a container, requires a certain kind of iterator. If the algorithm must access elements at arbitrary locations in the container, it requires a random-access iterator. If it will merely step forward through the iterator, it can use the less powerful forward iterator. Table 15.9 shows a sampling of algorithms and the iterators they require. (A complete version of this table is shown in Appendix F.)

**TABLE 15.9** Type of Iterator Required by Representative Algorithms

| *Algorithm* | *Input* | *Output* | *Forward* | *Bidirectional* | *Random Access* |
|---|---|---|---|---|---|
| `for_each` | x | | | | |
| `find` | x | | | | |
| `count` | x | | | | |
| `copy` | x | x | | | |
| `replace` | | | x | | |
| `unique` | | | x | | |
| `reverse` | | | | x | |
| `sort` | | | | | x |
| `nth_element` | | | | | x |
| `merge` | x | x | | | |
| `accumulate` | x | | | | |

Again, although each algorithm requires an iterator with a certain level of capability, a more powerful iterator will also work. The `replace()` algorithm requires a forward iterator, but it will work with a bidirectional or a random access iterator as well.

Now, imagine that algorithms have connectors with pins sticking out, like the cable connectors on your computer. This is shown in Figure 15.4. Those requiring random access iterators have 5 pins, those requiring bidirectional iterators have 4 pins, those requiring forward iterators have 3 pins, and so on.

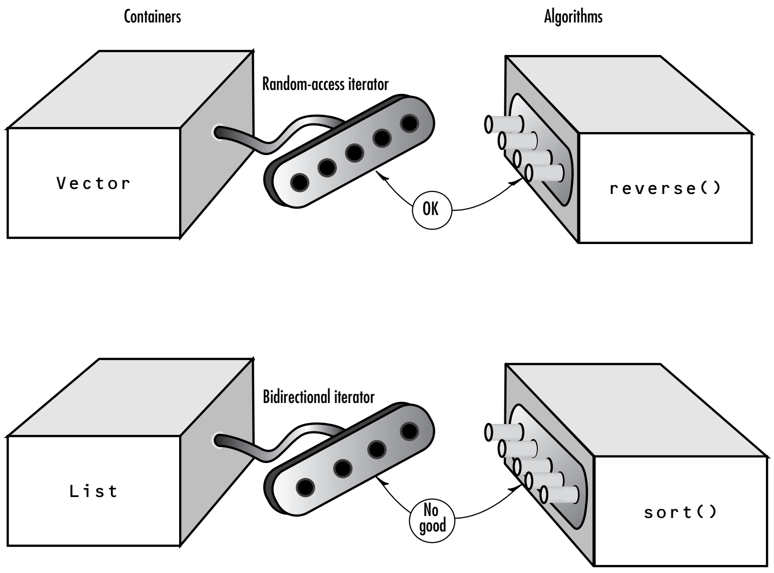

**FIGURE 15.4**
*Iterators connecting containers and algorithms.*

The algorithm end of an iterator (a cable) has a connector with a certain number of holes. You can plug a 5-hole iterator into a 5-pin algorithm, and you can also plug it into an algorithm with 4 or fewer pins. However, you can't plug a 4-hole (bidirectional) iterator into a 5-pin (random-access) algorithm. So vectors and deques, with random access iterators, can be plugged into any algorithm, while lists and associative containers, with only a 4-hole bidirectional iterator, can only be plugged into less powerful algorithms.

## The Tables Tell the Story

From Tables 15.8 and 15.9 you can figure out whether an algorithm will work with a given container. Table 15.9 shows that the `sort()` algorithm, for example, requires a random-access iterator. Table 15.8 indicates that the only containers that can handle random-access iterators are vectors and deques. There's no use trying to apply the `sort()` algorithm to lists, sets, maps, and so on.

Any algorithm that does *not* require a random-access iterator will work with any kind of STL container, because all these containers use bidirectional iterators, which is only one grade below random access. (If there were a singly-linked list in the STL it would use only a forward iterator, so it could not be used with the `reverse()` algorithm.)

As you can see, comparatively few algorithms require random-access iterators. Therefore most algorithms work with most containers.

## Overlapping Member Functions and Algorithms

Sometimes you must choose between using a member function or an algorithm with the same name. The `find()` algorithm, for example, requires only an input iterator, so it can be used with any container. However, sets and maps have their own `find()` member function (unlike sequential containers). Which version of `find()` should you use? Generally, if a member-function version exists, it's because, for that container, the algorithm version is not as efficient as it could be; so in these cases you should probably use the member-function version.

## Iterators at Work

Using iterators is considerably simpler than talking about them. We've already seen several examples of one of the more common uses, where iterator values are returned by a container's `begin()` and `end()` member functions. We've disguised the fact that these functions return iterator values by treating them as if they were pointers. Now let's see how actual iterators are used with these and other functions.

### Data Access

In containers that provide random access iterators (vector and queue) it's easy to iterate through the container using the `[]` operator. Containers such as lists, which don't support random access, require a different approach. In previous examples we've used a "destructive readout" to display the contents of a list by popping off the items one by one, as in the LIST and LISTPLUS examples. A more practical approach is to define an iterator for the container. The LISTOUT program shows how that might look:

```
// listout.cpp
// iterator and for loop for output
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
   {
   int arr[] = { 2, 4, 6, 8 };
   list<int> theList;

   for(int k=0; k<4; k++)      //fill list with array elements
      theList.push_back( arr[k] );

   list<int>::iterator iter;   //iterator to list-of-ints

   for(iter = theList.begin(); iter != theList.end(); iter++)
      cout << *iter << ' ';    //display the list
   cout << endl;
   return 0;
   }
```

The program simply displays the contents of the `theList` container. The output is

```
2 4 6 8
```

We define an iterator of type `list<int>` to match the container type. As with a pointer variable, we must give an iterator a value before using it. In the `for` loop we initialize it to `iList.begin()`, the start of the container. We can increment it with the `++` operator so that it steps through the elements in a container, and we can dereference it with the `*` operator to obtain the value of each element it points to. We can also compare it for equality using the `!=` operator, so we can exit the loop when it reaches the end of the container at `iList.end()`.

An equivalent approach, using a `while` loop instead of a `for` loop, might be

```
iter = iList.begin();
while( iter != iList.end() )
   cout << *iter++ << ' ';
```

The `*iter++` syntax is the same as it would be for a pointer.

## Data Insertion

We can use similar code to place data into existing elements in a container, as shown in LISTFILL:

```
// listfill.cpp
// uses iterator to fill list with data
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   list<int> iList(5);      //empty list holds 5 ints
   list<int>::iterator it;  //iterator
   int data = 0;
                            //fill list with data
   for(it = iList.begin(); it != iList.end(); it++)
      *it = data += 2;
                            //display list
   for(it = iList.begin(); it != iList.end(); it++)
      cout << *it << ' ';
   cout << endl;
   return 0;
   }
```

The first loop fills the container with the `int` values 2, 4, 6, 8, 10, showing that the overloaded `*` operator works on the left side of the equal sign as well as the right. The second loop displays these values.

## Algorithms and Iterators

Algorithms, as we've discussed, use iterators as arguments (and sometimes as return values). The ITERFIND example shows the `find()` algorithm applied to a list. (We know we can use the `find()` algorithm with lists, because it requires only an input iterator.)

```
// iterfind.cpp
// find() returns a list iterator
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   list<int> theList(5);      //empty list holds 5 ints
   list<int>::iterator iter;  //iterator
   int data = 0;
                              //fill list with data
   for(iter = theList.begin(); iter != theList.end(); iter++)
      *iter = data += 2;      //2, 4, 6, 8, 10
                              //look for number 8
   iter = find(theList.begin(), theList.end(), 8);
   if( iter != theList.end() )
      cout << "\nFound 8.\n";
   else
      cout << "\nDid not find 8.\n";
   return 0;
   }
```

As an algorithm, `find()` takes three arguments. The first two are iterator values specifying the range to be searched, and the third is the value to be found. Here we fill the container with the same 2, 4, 6, 8, 10 values as in the last example. Then we use the `find()` algorithm to look for the number 8. If `find()` returns `iList.end()`, we know it's reached the end of the container without finding a match. Otherwise, it must have located an item with the value 8. Here the output is

```
Found 8.
```

Can we use the value of the iterator to tell where in the container the 8 is located? You might think the offset of the matching item from the beginning of the container could be calculated from `(iter - iList.begin())`. However, this is not a legal operation on the iterators used for lists. A list iterator is only a bidirectional iterator, so you can't perform arithmetic with it. You can do arithmetic with random access iterators, such as those used with vectors and queues. Thus if you were searching a vector v rather than a list iList, you could rewrite the last part of ITERFIND like this:

```
iter = find(v.begin(), v.end(), 8);
if( iter != v.end() )
   cout << "\nFound 8 at location " << (iter-v.begin() );
else
   cout << "\nDid not find 8.";
```

The output would be

```
Found 8 at location 3
```

Here's another example in which an algorithm uses iterators as arguments. This one uses the copy() algorithm with a vector. The user specifies a range of locations to be copied from one vector to another, and the program copies them. Iterators specify this range.

```
// itercopy.cpp
// uses iterators for copy() algorithm
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
   {
   int beginRange, endRange;
   int arr[] = { 11, 13, 15, 17, 19, 21, 23, 25, 27, 29 };
   vector<int> v1(arr, arr+10);  //initialized vector
   vector<int> v2(10);           //uninitialized vector

   cout << "Enter range to be copied (example: 2 5): ";
   cin >> beginRange >> endRange;

   vector<int>::iterator iter1 = v1.begin() + beginRange;
   vector<int>::iterator iter2 = v1.begin() + endRange;
   vector<int>::iterator iter3;
                                 //copy range from v1 to v2
   iter3 = copy( iter1, iter2, v2.begin() );
                                 //(it3 -> last item copied)
   iter1 = v2.begin();           //iterate through range
   while(iter1 != iter3)         //in v2, displaying values
      cout << *iter1++ << ' ';
   cout << endl;
   return 0;
   }
```

Here's some interaction with this program:

```
Enter range to be copied (example: 2 5): 3 6
17 19 21
```

We don't display the entire contents of v2, only the range of items copied. Fortunately, copy() returns an iterator that points to the last item (actually one past the last item) that was copied to the destination container, v2 in this case. The program uses this value in the while loop to display only the items copied.

## Specialized Iterators

In this section we'll examine two specialized forms of iterators: iterator adapters, which can change the behavior of iterators in interesting ways, and stream iterators, which allow input and output streams to behave like iterators.

### Iterator Adapters

The STL provides three variations on the normal iterator. These are the *reverse iterator*, the *insert iterator*, and the *raw storage iterator*. The reverse iterator allows you to iterate backward through a container. The insert iterator changes the behavior of various algorithms, such as copy() and merge(), so they insert data into a container rather than overwriting existing data. The raw storage iterator allows output iterators to store data in uninitialized memory, but it's used in specialized situations and we'll ignore it here.

#### Reverse Iterators

Suppose you want to iterate backward through a container, from the end to the beginning. You might think you could say something like

```
list<int>::iterator iter;       //normal iterator
iter = iList.end();             //start at end
while( iter != iList.begin() )  //go to beginning
   cout << *iter-- << ' ';      //decrement iterator
```

but unfortunately this doesn't work. (For one thing, the range will be wrong (from n to 1, instead of from n–1 to 0).

To iterate backward you can use a *reverse iterator*. The ITEREV program shows an example where a reverse iterator is used to display the contents of a list in reverse order.

```
// iterev.cpp
// demonstrates reverse iterator
#include <iostream>
#include <list>
using namespace std;
```

```
int main()
   {
   int arr[] = { 2, 4, 6, 8, 10 };       //array of ints
   list<int> theList;

   for(int j=0; j<5; j++)                //transfer array
      theList.push_back( arr[j] );       //to list

   list<int>::reverse_iterator revit;    //reverse iterator

   revit = theList.rbegin();             //iterate backward
   while( revit != theList.rend() )      //through list,
      cout << *revit++ << ' ';           //displaying output
   cout << endl;
   return 0;
   }
```

The output of this program is

```
10 8 6 4 2
```

You must use the member functions `rbegin()` and `rend()` when you use a reverse iterator. (Don't try to use them with a normal forward iterator.) Confusingly, you're starting at the end of the container, but the member function is called `rbegin()`. Also, you must increment the iterator. Don't try to decrement a reverse iterator; `revit--` doesn't do what you want. With a `reverse_iterator`, always go from `rbegin()` to `rend()` using the increment operator.

## Insert Iterators

Some algorithms, such as `copy()`, overwrite the existing contents (if any) of the destination container. The COPYDEQ program, which copies from one deque to another, provides an example:

```
// copydeq.cpp
//demonstrates normal copy with queues
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

int main()
   {
   int arr1[] = { 1, 3, 5, 7, 9 };
   int arr2[] = { 2, 4, 6, 8, 10 };
   deque<int> d1;
   deque<int> d2;
   for(int j=0; j<5; j++)          //transfer arrays to deques
      {
      d1.push_back( arr1[j] );
      d2.push_back( arr2[j] );
      }                            //copy d1 to d2
   copy( d1.begin(), d1.end(), d2.begin() );

   for(int k=0; k<d2.size(); k++)  //display d2
      cout << d2[k] << ' ';
   cout << endl;
   return 0;
   }
```

The output of this program is

```
1 3 5 7 9
```

The contents of `d2` have been written over by the contents of `d1`, so when we display `d2` there's no trace of its former (even-numbered) contents. Usually this behavior is what you want. Sometimes, however, you'd rather that `copy()` inserted new elements into a container along with the old ones, instead of overwriting the old ones. You can cause this behavior by using an *insert iterator*. There are three flavors of this iterator:

- `back_inserter` inserts new items at the end
- `front_inserter` inserts new items at the beginning
- `inserter` inserts new items at a specified location

The DINSITER program shows how to use a back inserter.

```
//dinsiter.cpp
//demonstrates insert iterators with queues
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

int main()
   {
   int arr1[] = { 1, 3, 5, 7, 9 };  //initialize d1
   int arr2[] = {2, 4, 6};          //initialize d2
   deque<int> d1;
   deque<int> d2;

   for(int i=0; i<5; i++)           //transfer arrays to deques
      d1.push_back( arr1[i] );
   for(int j=0; j<3; j++)
      d2.push_back( arr2[j] );
```

```
                                    //copy d1 to back of d2
copy( d1.begin(), d1.end(), back_inserter(d2) );

cout << "\nd2: ";                  //display d2
for(int k=0; k<d2.size(); k++)
   cout << d2[k] << ' ';
cout << endl;
return 0;
}
```

The back inserter uses the container's `push_back()` member function to insert the new items from source container `d1` at the end of the target container `d2`, following the existing items. Container `d1` is unchanged. The output of the program, which displays the new contents of `d2`, is

```
d2: 2 4 6 1 3 5 7 9
```

If we specified a front inserter instead

```
copy( d1.begin(), d1.end(), front_inserter(d2) );
```

then the new items would be inserted into the front of the container. The underlying mechanism of the front inserter is the container's `push_front()` member function, which pushes the items into the front of the container, effectively reversing their order. The output would be

```
9 7 5 3 1 2 4 6
```

You can also insert the new items starting at any arbitrary element by using the *inserter* version of the insert iterator. For example, to insert the new items at the beginning of `d2`, we would say

```
copy( d1.begin(), d1.end(), inserter(d2, d2.begin() );
```

The first argument to `inserter` is the container to be copied into, and the second is an iterator pointing to the location where copying should begin. Because `inserter` uses the container's `insert()` member function, the order of the elements is not reversed. The output resulting from this statement would be

```
1 3 5 7 9 2 4 6
```

By changing the second argument to `inserter` we could cause the new data to be inserted anywhere in `d2`.

Note that a `front_inserter` can't be used with a vector, because vectors don't have a `push_front()` member function; they can only be accessed at the end.

## Stream Iterators

Stream iterators allow you to treat files and I/O devices (such as `cin` and `cout`) as if they were iterators. This makes it easy to use files and I/O devices as arguments to algorithms. (This is another demonstration of the versatility of using iterators to link algorithms and containers.)

The major purpose of the input and output iterator categories is to support these stream iterator classes. Input and output iterators make it possible for appropriate algorithms to be used directly on input and output streams.

Stream iterators are actually objects of classes that are templetized for different types of input or output. There are two stream iterators: `ostream_iterator` and `istream_iterator`. Let's look at them in turn.

### The `ostream_iterator` Class

An `ostream_iterator` object can be used as an argument to any algorithm that specifies an output iterator. In the OUTITER example we'll use it as an argument to `copy()`:

```
//outiter.cpp
//demonstrates ostream_iterator
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   int arr[] = { 10, 20, 30, 40, 50 };
   list<int> theList;

   for(int j=0; j<5; j++)               //transfer array to list
      theList.push_back( arr[j] );

   ostream_iterator<int> ositer(cout, ", ");  //ostream iterator

   cout << "\nContents of list: ";
   copy(theList.begin(), theList.end(), ositer);  //display list
   cout << endl;
   return 0;
   }
```

We define an `ostream` iterator for reading type `int` values. The two arguments to this constructor are the stream to which the `int` values will be written, and a string value that will be displayed following each value. The stream value is typically a filename or `cout`; here it's `cout`. When writing to `cout`, the delimiting string can consist of any characters you want; here we use a comma and a space.

The copy() algorithm copies the contents of the list to cout. The ostream iterator is used as the third argument to copy(); it's the destination.

The output of OUTITER is

```
Contents of list: 10, 20, 30, 40, 50,
```

Our next example, FOUTITER, shows how to use an ostream iterator to write to a file:

```
//foutiter.cpp
//demonstrates ostream_iterator with files
#include <fstream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   int arr[] = { 11, 21, 31, 41, 51 };
   list<int> theList;

   for(int j=0; j<5; j++)              //transfer array
      theList.push_back( arr[j] );     //   to list
   ofstream outfile("ITER.DAT");       //create file object

   ostream_iterator<int> ositer(outfile, " ");  //iterator
                                       //write list to file
   copy(theList.begin(), theList.end(), ositer);
   return 0;
   }
```

You must define an ofstream file object and associate it with a file, here called ITER.DAT. This object is the first argument to the ostream_iterator. When writing to a file, use a whitespace character in the string argument, not characters like "--". This makes it easier to read the data back from the file. Here we use a space (" ") character.

There's no displayable output from FOUTITER, but you can use a text editor (like the Notepad utility in Windows) to examine the file ITER.DAT, which was created by the ITER program. It should contain the data

```
11 21 31 41 51
```

## The istream_iterator Class

An istream_iterator object can be used as an argument to any algorithm that specifies an input iterator. Our example, INITER, shows such objects used as the first two arguments to copy(). This program reads floating-point numbers entered into cin (the keyboard) by the user, and stores them in a list.

```
// initer.cpp
// demonstrates istream_iterator
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
   {
   list<float> fList(5);            //uninitialized list

   cout << "\nEnter 5 floating-point numbers: ";
                                    //istream iterators
   istream_iterator<float> cin_iter(cin);   //cin
   istream_iterator<float> end_of_stream;   //eos
                                    //copy from cin to fList
   copy( cin_iter, end_of_stream, fList.begin() );

   cout << endl;                    //display fList
   ostream_iterator<float> ositer(cout, "--");
   copy(fList.begin(), fList.end(), ositer);
   cout << endl;
   return 0;
   }
```

Here's some interaction with INITER

```
Enter 5 floating-point numbers: 1.1  2.2  3.3  4.4  5.5
1.1--2.2--3.3--4.4--5.5--
```

Notice that for `copy()`, because the data coming from `cin` is the source and not the destination, we must specify both the beginning and the end of the range of data to be copied. The beginning is an `istream_iterator` connected to `cin`, which we define as `cin_iter` using the one-argument constructor. But what about the end of the range? The no-argument (default) constructor to `istream_iterator` plays a special role here. It always creates an `istream_iterator` object that represents the end of the stream.

How does the user generate this end-of-stream value when inputting data? By typing the Ctrl+Z key combination, which transmits the end-of-file character normally used for streams. Sometimes several presses of Ctrl+Z are necessary. Pressing Enter won't end the file, although it will delimit the numbers.

We use an `ostream_iterator` to display the contents of the list, although of course there are many other ways to do this.

You must perform any display output, such as the "Enter 5 floating-point numbers" prompt, not only before using the istream iterator, but even before defining it. As soon as this iterator is defined, it locks up the display, waiting for input.

Our next example, FINITER, uses a file instead of `cin` as input to the `copy()` algorithm.

```
// finiter.cpp
// demonstrates istream_iterator with files
#include <iostream>
#include <list>
#include <fstream>
#include <algorithm>
using namespace std;

int main()
   {
   list<int> iList;               //empty list
   ifstream infile("ITER.DAT");   //create input file object
                                  //(ITER.DAT must already exist)
                                  //istream iterators
   istream_iterator<int> file_iter(infile);  //file
   istream_iterator<int> end_of_stream;      //eos
                                  //copy from infile to iList
   copy( file_iter, end_of_stream, back_inserter(iList) );

   cout << endl;                  //display iList
   ostream_iterator<int> ositer(cout, "--");
   copy(iList.begin(), iList.end(), ositer);
   cout << endl;
   return 0;
   }
```

The output from FINITER is

```
11--21--31--31--41--51--
```

We define an `ifstream` object to represent the ITER.DAT file, which must already exist and contain data. (The FOUTITER program, if you ran it, will have generated this file.)

Instead of using `cout`, as in the `istream` iterator in the INITER example, we use the `ifstream` object named `infile`. The end-of-stream object is the same.

We've made another change in this program: it uses a `back_inserter` to insert data into `iList`. This makes it possible to define `iList` as an empty container instead of one with a specified size. This often makes sense when reading input, since you may not know how many items will be entered.

## Associative Containers

We've seen that the sequence containers (vector, list, and deque) store data items in a fixed linear sequence. Finding an item in such a container (unless its index number is known or it's located at an end of the container) will involve the slow process of stepping through the items in the container one by one.

In an associative container the items are not arranged in sequence. Instead they are arranged in a more complex way that makes it much faster to find a given item. This arrangement is typically a tree structure, although different approaches (such as hash tables) are possible. The speed of searching is the main advantage of associative containers.

Searching is done using a *key*, which is usually a single value like a number or string. This value is an attribute of the objects in the container, or it may be the entire object.

The two main categories of associative containers in the STL are sets and maps.

A set stores objects containing keys. A map stores pairs, where the first part of the pair is an object containing a key and the second part is an object containing a value.

In both a set and a map, only one example of each key can be stored. It's like a dictionary that forbids more than one entry for each word. However, the STL has alternative versions of set and map that relax this restriction. A *multiset* and a *multimap* are similar to a set and a map, but can include multiple instances of the same key.

Associative containers share many member functions with other containers. However, some algorithms, such as `lower_bound()` and `equal_range()`, exist only for associative containers. Also, some member functions that do exist for other containers, such as the push and pop family (`push_back()` and so on) have no versions for associative containers. It wouldn't make sense to use push and pop with associative containers, because elements must always be inserted in their ordered locations, not at the beginning or end of the container.

## Sets and Multisets

Sets are often used to hold objects of user-defined classes such as employees in a database. (You'll see examples of this later in this chapter.) However, sets can also hold simpler elements such as strings. Figure 15.5 shows how this looks. The objects are arranged in order, and the entire object is the key.

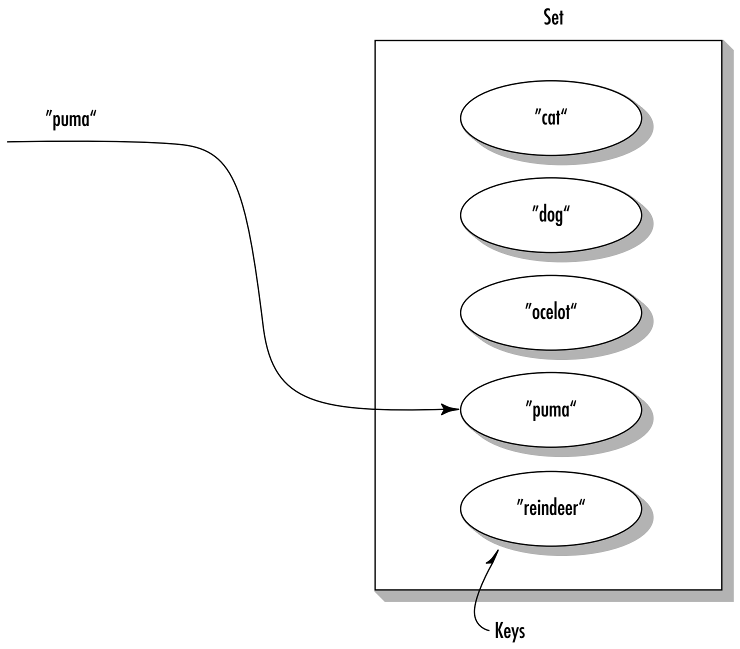

**FIGURE 15.5**
*A set of string objects.*

Our first example, SET, shows a set that stores objects of class `string`.

```
// set.cpp
// set stores string objects
#pragma warning (disable:4786)  //for set (Microsoft only)
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
   {                           //array of string objects
   string names[] = {"Juanita", "Robert",
                     "Mary", "Amanda", "Marie"};
                               //initialize set to array
   set<string, less<string> > nameSet(names, names+5);
                               //iterator to set
   set<string, less<string> >::iterator iter;

   nameSet.insert("Yvette");  //insert more names
   nameSet.insert("Larry");
   nameSet.insert("Robert");  //no effect; already in set
   nameSet.insert("Barry");
   nameSet.erase("Mary");     //erase a name
                              //display size of set
   cout << "\nSize=" << nameSet.size() << endl;
   iter = nameSet.begin();    //display members of set
   while( iter != nameSet.end() )
      cout << *iter++ << '\n';

   string searchName;         //get name from user
   cout << "\nEnter name to search for: ";
   cin >> searchName;
                              //find matching name in set
   iter = nameSet.find(searchName);
   if( iter == nameSet.end() )
      cout << "The name " << searchName << " is NOT in the set.";
   else
      cout << "The name " << *iter << " IS in the set.";
   cout << endl;
   return 0;
   }
```

The directive

```
#pragma warning (disable:4786)
```

may be necessary on the Microsoft compiler when you use the SET or MAP files. It disables warning 4786 ("identifier was truncated to 255 characters in the debug information"), whose appearance seems to be a bug. The pragma must preceed the `#includes` for all files, not just for SET and MAP, which cause the problem. A *pragma* is a compiler-specific directive that fine-tunes compiler operations.

To define a set we specify the type of objects to be stored (in this case class `string`) and also the function object that will be used to order the members of the set. Here we use `less<>()` applied to string objects.

As you can see, a set has an interface similar to other STL containers. We can initialize a set to an array, and insert new members into a set with the `insert()` member function. To display the set we can iterate through it.

To find a particular entry in the set we use the `find()` member function. (Sequential containers use `find()` in its algorithm version.) Here's some sample interaction with SET, where the user enters "George" as the name to be searched for:

```
Size = 7
Amanda
Barry
```

```
Juanita
Larry
Marie
Robert
Yvette

Enter name to search for: George
The name George is NOT in the set.
```

Of course the speed advantage of searching an associative container isn't apparent until you have many more entries than in this example.

Let's look at an important pair of member functions available only with associative containers. Our example, SETRANGE, shows the use of `lower_bound()` and `upper_bound()`:

```
// setrange.cpp
// tests ranges within a set
#pragma warning (disable:4786)  //for set (Microsoft only)
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
   {                           //set of string objects
   set<string, less<string> > organic;
                               //iterator to set
   set<string, less<string> >::iterator iter;

   organic.insert("Curine");  //insert organic compounds
   organic.insert("Xanthine");
   organic.insert("Curarine");
   organic.insert("Melamine");
   organic.insert("Cyanimide");
   organic.insert("Phenol");
   organic.insert("Aphrodine");
   organic.insert("Imidazole");
   organic.insert("Cinchonine");
   organic.insert("Palmitamide");
   organic.insert("Cyanimide");

   iter = organic.begin();    //display set
   while( iter != organic.end() )
      cout << *iter++ << '\n';

   string lower, upper;       //display entries in range
   cout << "\nEnter range (example C Czz): ";
   cin >> lower >> upper;
   iter = organic.lower_bound(lower);
   while( iter != organic.upper_bound(upper) )
      cout << *iter++ << '\n';
   return 0;
   }
```

The program first displays an entire set of organic compounds. The user is then prompted to type in a pair of key values, and the program displays those keys that lie within this range. Here's some sample interaction:

```
Aphrodine
Cinchonine
Curarine
Curine
Cyanimide
Imidazole
Melamine
Palmitamide
Phenol
Xanthine

Enter range (example C Czz): Aaa Curb
Aphrodine
Cinchonine
Curarine
```

The `lower_bound()` member function takes an argument that is a value of the same type as the key. It returns an iterator to the first entry that is not less than this argument (where the meaning of "less" is determined by the function object used in the set's definition). The `upper_bound()` function returns an iterator to the first entry that is greater than its argument. Together, these functions allow you to access a specified range of values.

## Maps and Multimaps

A *map* stores pairs. A pair consists of a *key object* and a *value object*. The key object contains a key that will be searched for. The value object contains additional data. As in a set, the key objects can be strings, numbers, or objects of more complex classes. The values are often strings or numbers, but they can also be objects or even containers.

For example, the key could be a word, and the value could be a number representing how many times that word appears in a document. Such a map constitutes a *frequency table*. Or the key could be a word and the value could be a list of page numbers. This arrangement could represent an index, like the one at the back of this book. Figure 15.6 shows a situation in which the keys are words and the values are definitions, as in an ordinary dictionary.

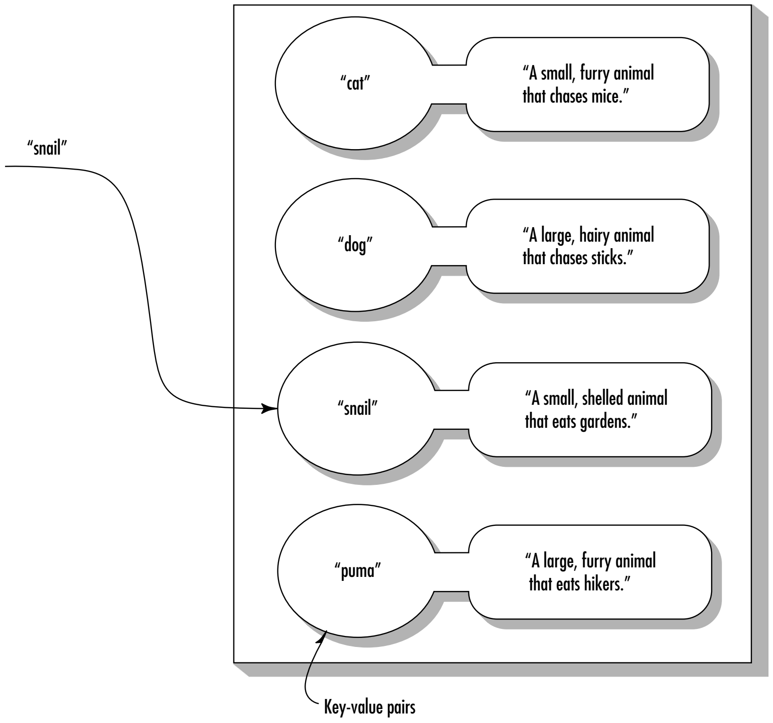
"snail"
"cat"
"A small, furry animal that chases mice."
"dog"
"A large, hairy animal that chases sticks."
"snail"
"A small, shelled animal that eats gardens."
"puma"
"A large, furry animal that eats hikers."
Key-value pairs

**FIGURE 15.6**
*A map of word-phrase pairs.*

One common way to use maps is as associative arrays. In an ordinary C++ array the array index, which is used to access a particular element, is an integer. Thus in the expression `anArray[3]`, the 3 is the array index. An associative array works in a similar way except that you can choose the data type of the array index. If you've defined the index to be a string, for example, you can say `anArray["jane"]`.

## An Associative Array

Let's look at a simple example of a map used as an associative array. The keys will be the names of states, and the values will be the populations of the states. Here's the listing for ASSO_ARR:

```
// asso_arr.cpp
// demonstrates map used as associative array
#pragma warning (disable:4786)  //for map (Microsoft only)
#include <iostream>
#include <string>
#include <map>
using namespace std;
```

```
int main()
   {
   string name;
   int pop;

   string states[] = { "Wyoming", "Colorado", "Nevada",
                       "Montana", "Arizona", "Idaho"};
   int pops[] = { 470, 2890, 800, 787, 2718, 944 };

   map<string, int, less<string> > mapStates;       //map
   map<string, int, less<string> >::iterator iter;  //iterator

   for(int j=0; j<6; j++)
      {
      name = states[j];                  //get data from arrays
      pop = pops[j];
      mapStates[name] = pop;             //put it in map
      }
   cout << "Enter state: ";              //get state from user
   cin >> name;
   pop = mapStates[name];                //find population
   cout << "Population: " << pop << ",000\n";

   cout << endl;                         //display entire map
   for(iter = mapStates.begin(); iter != mapStates.end(); iter++)
      cout << (*iter).first << ' ' << (*iter).second << ",000\n";
   return 0;
   }
```

When the program runs, the user is prompted to type the name of a state. The program then looks in the map, using the state name as an index, and returns the population of the state. Finally, it displays all the name-population pairs in the map. Here's some sample output:

```
Enter state: Wyoming
Population: 470,000

Arizona 2718,000
Colorado 2890,000
Idaho 944,000
Montana 787,000
Nevada 800,000
Wyoming 470,000
```

Search speed is where sets and maps excel. Here the program quickly finds the appropriate population when the user enters a state's name. (This would be more meaningful if there were millions of data items.) Iterating through the container, as is shown by the list of states and populations, isn't as fast as in a sequential container, but it's still fairly efficient. Notice that the states are ordered alphabetically, although the original data was not.

The definition of a map takes three template arguments:

```
map<string, int, less<string> > maStates;
```

The first is the type of the key. In this case it's `string`, representing the state name. The second is the type of the value; in this case it's `int`, which represents the population, in 1,000s. The third argument specifies the ordering that will be used for the keys. We choose to have it ordered alphabetically by the names of the states; that's what `less<string>` does. We also define an iterator to this map.

Our input data is in two separate arrays. (In a real program it would probably come from a file.) To put this data into the map we read it into the variables `name` and `pop`, and execute the statement

```
mapStates[name] = pop;
```

This is a particularly elegant construction, looking just like an insertion into an ordinary array. However, the array index `name` is a string, not an integer.

When the user types in a state name, the program finds the appropriate population with the statement

```
pop = mapStates[name];
```

Besides using the array-index syntax, we can also access the two parts of an entry in the map, the key, and the value, using an iterator. The key is obtained from `(*iter).first`, and the value from `(*iter).second`. Otherwise the iterator works as it does in other containers.

## Storing User-Defined Objects

Until now our example programs have stored objects of basic types. However, the big payoff with the STL is that you can use it to store and manipulate objects of classes that you write yourself (or that someone else has written). In this section we'll show how this is done.

### A Set of `person` Objects

We'll start with a `person` class that includes a person's last name, first name, and telephone number. We'll create some members of this class and insert them in a set, thus creating a phone book database. The user interacts with the program by entering a person's name. The program then searches the list and displays the data for that person, if it finds a match. We'll use a multiset so two or more `person` objects can have the same name. Here's the listing for SETPERS:

```
// setpers.cpp
// uses a multiset to hold person objects
#pragma warning (disable:4786)  //for set (Microsoft only)
#include <iostream>
```

```
#include <set>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:                    //default constructor
      person() : lastName("blank"),
                 firstName("blank"), phoneNumber(0)
         {  }
                              //3-arg constructor
      person(string lana, string fina, long pho) :
              lastName(lana), firstName(fina), phoneNumber(pho)
         {  }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);

      void display() const   //display person's data
         {
         cout << endl << lastName << ",\t" << firstName
              << "\t\tPhone: " << phoneNumber;
         }
   };
                             //operator < for person class
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
                             //operator == for person class
bool operator==(const person& p1, const person& p2)
   {
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName ) ? true : false;
   }
////////////////////////////////////////////////////////////////
int main()
   {                         //create person objects
   person pers1("Deauville", "William", 8435150);
   person pers2("McDonald", "Stacey", 3327563);
   person pers3("Bartoski", "Peter", 6946473);
   person pers4("KuangThu", "Bruce", 4157300);
   person pers5("Wellington", "John", 9207404);
   person pers6("McDonald", "Amanda", 8435150);
   person pers7("Fredericks", "Roger", 7049982);
   person pers8("McDonald", "Stacey", 7764987);
                             //multiset of persons
   multiset< person, less<person> > persSet;
                             //iterator to a multiset of persons
   multiset<person, less<person> >::iterator iter;

   persSet.insert(pers1);    //put persons in multiset
   persSet.insert(pers2);
   persSet.insert(pers3);
   persSet.insert(pers4);
   persSet.insert(pers5);
   persSet.insert(pers6);
   persSet.insert(pers7);
   persSet.insert(pers8);

   cout << "\nNumber of entries = " << persSet.size();

   iter = persSet.begin();   //display contents of multiset
   while( iter != persSet.end() )
      (*iter++).display();
                             //get last and first name
   string searchLastName, searchFirstName;
   cout << "\n\nEnter last name of person to search for: ";
   cin >> searchLastName;
   cout << "Enter first name: ";
   cin >> searchFirstName;
                             //create person with this name
   person searchPerson(searchLastName, searchFirstName, 0);

                             //get count of such persons
   int cntPersons = persSet.count(searchPerson);
   cout << "Number of persons with this name = " << cntPersons;

                             //display all matches
   iter = persSet.lower_bound(searchPerson);
   while( iter != persSet.upper_bound(searchPerson) )
      (*iter++).display();
   cout << endl;
   return 0;
   }  //end main()
```

## Necessary Member Functions

To work with STL containers, the `person` class needs a few common member functions. These are a default (no-argument) constructor (which is actually not necessary in this example but is usually essential), the overloaded < operator, and the overloaded == operator. These member functions are used by the list class and by various algorithms. You may need other member functions in other specific situations. (As in most classes, you should probably also provide overloaded assignment and copy constructors and a destructor, but we'll ignore these here to avoid complicating the listing.)

The overloaded < and == operators should use `const` arguments. Generally it's best to make them `friends`, but you can use member functions as well.

## Ordering

The overloaded < operator specifies the way the elements in the set will be ordered. In SETPERS we define this operator to order the last name of the person, and, if the last names are the same, to order the first names.

Here's some interaction with SETPERS. The program first displays the entire list. (Of course this would not be practical on a real database with a large number of elements.) Because they are stored in a multiset, the elements are ordered automatically. Then, at the prompt, the user enters the name "McDonald" followed by "Stacey" (last name first). There are two persons on the list with this particular name, so they are both displayed.

```
Number of entries = 8
Bartoski,       Peter           phone: 6946473
Deauville,      William         phone: 8435150
Fredericks,     Roger           phone: 7049982
KuangThu,       Bruce           phone: 4157300
McDonald,       Amanda          phone: 8435150
McDonald,       Stacey          phone: 3327563
McDonald,       Stacey          phone: 7764987
Wellington,     John            phone: 9207404

Enter last name of person to search for: McDonald
Enter first name: Stacey
Number of persons with this name = 2
McDonald,       Stacey          phone: 3327563
McDonald,       Stacey          phone: 7764987
```

## Just Like Basic Types

As you can see, once a class has been defined, objects of that class are handled by the container in the same way as variables of basic types.

We first use the `size()` member function to display the total number of entries. Then we iterate through the list, displaying all the entries.

Because we're using a multiset, the `lower_bound()` and `upper_bound()` member functions are available to display all elements that fall within a range. In the example output the lower and upper bound are the same, so all persons with the same name are displayed. Notice that we must create a "fictitious" person with the same name as the person (or persons) we want to find. The `lower_bound()` and `upper_bound()` functions then match this person against those on the list.

## A List of `person` Objects

It's very fast to search a set or multiset for a person with a given name, as in the SETPERS example. If, however, we're more concerned with being able to quickly insert or delete a `person` object, we might decide to use a list instead. The LISTPERS example shows how this looks.

```
// listpers.cpp
// uses a list to hold person objects
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:
      person() :              //no-arg constructor
           lastName("blank"), firstName("blank"), phoneNumber(0L)
         {  }
                              //3-arg constructor
      person(string lana, string fina, long pho) :
              lastName(lana), firstName(fina), phoneNumber(pho)
         {  }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);
      friend bool operator!=(const person&, const person&);
      friend bool operator>(const person&, const person&);

      void display() const    //display all data
         {
         cout << endl << lastName << ",\t" << firstName
              << "\t\tPhone: " << phoneNumber;
         }
      long get_phone() const //return phone number
         { return phoneNumber; }
   };
                              //overloaded == for person class
bool operator==(const person& p1, const person& p2)
   {
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName ) ? true : false;
   }
                              //overloaded < for person class
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
                              //overloaded != for person class
bool operator!=(const person& p1, const person& p2)
   { return !(p1==p2); }
                              //overloaded > for person class
bool operator>(const person& p1, const person& p2)
   { return !(p1<p2) && !(p1==p2); }
////////////////////////////////////////////////////////////////
int main()
   {
   list<person> persList;    //list of persons
                              //iterator to a list of persons
   list<person>::iterator iter1;
                              //put persons in list
   persList.push_back( person("Deauville", "William", 8435150) );
   persList.push_back( person("McDonald", "Stacey", 3327563) );
   persList.push_back( person("Bartoski", "Peter", 6946473) );
   persList.push_back( person("KuangThu", "Bruce", 4157300) );
   persList.push_back( person("Wellington", "John", 9207404) );
   persList.push_back( person("McDonald", "Amanda", 8435150) );
   persList.push_back( person("Fredericks", "Roger", 7049982) );
   persList.push_back( person("McDonald", "Stacey", 7764987) );

   cout << "\nNumber of entries = " << persList.size();

   iter1 = persList.begin();  //display contents of list
   while( iter1 != persList.end() )
      (*iter1++).display();

//find person or persons with specified name (last and first)
   string searchLastName, searchFirstName;
   cout << "\n\nEnter last name of person to search for: ";
   cin >> searchLastName;
   cout << "Enter first name: ";
   cin >> searchFirstName;
                               //make a person with that name
   person searchPerson(searchLastName, searchFirstName, 0L);
                               //search for first match of names
   iter1 = find(persList.begin(), persList.end(), searchPerson);
   if( iter1 != persList.end() )  //find additional matches
      {
      cout << "Person(s) with that name is(are)";
      do
         {
         (*iter1).display();  //display match
         ++iter1;             //search again, one past match
         iter1 = find(iter1, persList.end(), searchPerson);
         } while( iter1 != persList.end() );
      }
   else
      cout << "There is no person with that name.";

//find person or persons with specified phone number
   cout << "\n\nEnter phone number (format 1234567): ";
   long sNumber;               //get search number
   cin >> sNumber;
                               //iterate through list
   bool found_one = false;
   for(iter1=persList.begin(); iter1 != persList.end(); ++iter1)
      {
      if( sNumber == (*iter1).get_phone() )  //compare numbers
         {
         if( !found_one )
            {
            cout << "Person(s) with that phone number is(are)";
            found_one = true;
            }
         (*iter1).display();  //display the match
         }
      }  //end for
   if( !found_one )
      cout << "There is no person with that phone number";
   cout << endl;
   return 0;
   }  //end main()
```

## Finding All Persons with a Specified Name

We can't use the `lower_bound()/upper_bound()` member functions because we're dealing with a list, not a set or map. Instead we use the `find()` member function to find all the persons with a given name. If this function reports a hit, we must apply it again, starting one person past the original hit, to see whether there are other persons with the same name. This complicates the programming; we must use a loop and two calls to `find()`.

## Finding All Persons with a Specified Phone Number

It's harder to search for a person with a specified phone number than one with a specified name, because the class member functions such as `find()` are intended to be used to find the primary search characteristic. In this example we use the brute force approach to finding the phone number, iterating through the list and making a "manual" comparison of the number we're looking for and each member of the list:

```
if( sNumber == (*iter1).getphone() )
   ...
```

The program first displays all the entries, then asks the user for a name and finds the matching person or persons. It then asks for a phone number and again finds any matching persons. Here's some interaction with LISTPERS:

```
Number of entries = 8
Deauville,      William         phone: 8435150
McDonald,       Stacey          phone: 3327563
Bartoski,       Peter           phone: 6946473
KuangThu,       Bruce           phone: 4157300
Wellington,     John            phone: 9207404
McDonald,       Amanda          phone: 8435150
Fredericks,     Roger           phone: 7049982
McDonald,       Stacey          phone: 7764987

Enter last name of person to search for: Wellington
Enter first name: John
Person(s) with that name is(are)
Wellington,     John            phone: 9207404

Enter phone number (format 1234567): 8435150
Person(s) with that number is(are)
Deauville,      William         phone: 8435150
McDonald,       Amanda          phone: 8435150
```

Here the program has found one person with the specified name and two people with the specified phone number.

When using lists to store class objects we must declare four comparison operators for that class: ==, `!=`, <, and >. Depending on what algorithms you actually use, you may not need to define (provide function bodies for) all these operators. In this example we only need to define the == operator, although for completeness we define all four. If we used the `sort()` algorithm on the list, we would need to define the < operator as well.

## Function Objects

Function objects are used extensively in the STL. One important use for them is as arguments to certain algorithms. They allow you to customize the operation of these algorithms. We mentioned function objects earlier in this chapter, and used one in the SORTEMP program. There we showed an example of the predefined function object `greater<>()` used to sort data in reverse order. In this section we'll examine other predefined function objects, and also see how you can write your own so that you have even greater control over what the STL algorithms do.

Recall that a function object is a function that has been wrapped in a class so that it looks like an object. The class, however, has no data and only one member function, which is the overloaded `()` operator. The class is often templatized so it can work with different types.

### Predefined Function Objects

The predefined STL function objects, located in the FUNCTIONAL header file, are shown in Table 15.10. There are function objects corresponding to all the major C++ operators. In the table, the letter *T* indicates any class, either user-written or a basic type. The variables x and y represent objects of class T passed to the function object as arguments.

**TABLE 15.10** Predefined Function Objects

| *Function Object* | *Return Value* |
|---|---|
| `T = plus(T, T)` | `x+y` |
| `T = minus(T, T)` | `x-y` |
| `T = times(T, T)` | `x*y` |
| `T = divide(T, T)` | `x/y` |
| `T = modulus(T, T)` | `x%y` |
| `T = negate(T)` | `-x` |
| `bool = equal_to(T, T)` | `x == y` |
| `bool = not_equal_to(T, T)` | `x != y` |
| `bool = greater(T, T)` | `x > y` |
| `bool = less(T, T)` | `x < y` |
| `bool = greater_equal(T, T)` | `x >= y` |

**TABLE 15.10** Continued

| *Function Object* | *Return Value* |
|---|---|
| `bool = less_equal(T, T)` | `x <= y` |
| `bool = logical_and(T, T)` | `x && y` |
| `bool = logical_or(T, T)` | `x \|\| y` |
| `bool = logical_not(T)` | `!x` |

There are function objects for arithmetic operations, comparisons, and logical operations. Let's look at an example where an arithmetic function object might come in handy. Our example uses a class called `airtime`, which represents time values consisting of hours and minutes, but no seconds. This data type is appropriate for flight arrival and departure times in airports. The example shows how the `plus<>()` function object can be used to add all the `airtime` values in a container. Here's the listing for PLUSAIR:

```
//plusair.cpp
//uses accumulate() algorithm and plus() function object
#include <iostream>
#include <list>
#include <numeric>          //for accumulate()
using namespace std;
////////////////////////////////////////////////////////////////
class airtime
   {
   private:
      int hours;            //0 to 23
      int minutes;          //0 to 59
   public:
                            //default constructor
      airtime() : hours(0), minutes(0)
         {  }
                            //2-arg constructor
      airtime(int h, int m) : hours(h), minutes(m)
         {  }
      void display() const  //output to screen
         { cout << hours << ':' << minutes; }

      void get()            //input from user
         {
         char dummy;
         cout << "\nEnter airtime (format 12:59): ";
         cin >> hours >> dummy >> minutes;
         }
```

```
                           //overloaded + operator
      airtime operator + (const airtime right) const
         {                 //add members
         int temph = hours + right.hours;
         int tempm = minutes + right.minutes;
         if(tempm >= 60)    //check for carry
            { temph++; tempm -= 60; }
         return airtime(temph, tempm); //return sum
         }
                           //overloaded == operator
      bool operator == (const airtime& at2) const
         { return (hours == at2.hours) &&
                  (minutes == at2.minutes); }
                           //overloaded < operator
      bool operator < (const airtime& at2) const
         { return (hours < at2.hours) ||
                  (hours == at2.hours && minutes < at2.minutes); }
                           //overloaded != operator
      bool operator != (const airtime& at2) const
         { return !(*this==at2); }
                           //overloaded > operator
      bool operator > (const airtime& at2) const
         { return !(*this<at2) && !(*this==at2); }
   };  //end class airtime
////////////////////////////////////////////////////////////////
int main()
   {
   char answer;
   airtime temp, sum;
   list<airtime> airlist;   //list of airtimes

   do {                     //get airtimes from user
      temp.get();
      airlist.push_back(temp);
      cout << "Enter another (y/n)? ";
      cin >> answer;
      } while (answer != 'n');
                           //sum all the airtimes
   sum = accumulate( airlist.begin(), airlist.end(),
                     airtime(0, 0), plus<airtime>() );
   cout << "\nsum = ";
   sum.display();           //display sum
   cout << endl;
   return 0;
   }
```

This program features the `accumulate()` algorithm. There are two versions of this function. The three-argument version always sums (using the `+` operator) a range of values. In the four-argument version shown here, any of the arithmetic function objects shown in Table 15.10 can be used.

The four arguments to this version of `accumulate()` are the iterators of the first and last elements in the range, the initial value of the sum (often 0), and the operation to be applied to the elements. In this example we add them using `plus<>()`, but we could subtract them, multiply them, or perform other operations using different function objects. Here's some interaction with PLUSAIR:

```
Enter airtime (format 12:59) : 3:45
Enter another (y/n)? y

Enter airtime (format 12:59) : 5:10
Enter another (y/n)? y

Enter airtime (format 12:59) : 2:25
Enter another (y/n)? y

Enter airtime (format 12:59) : 0:55
Enter another (y/n)? n

sum = 12:15
```

The `accumulate()` algorithm is not only easier and clearer than iterating through the container yourself to add the elements, it's also (unless you put a lot of work into your code) more efficient.

The `plus<>()` function object requires that the `+` operator be overloaded for the `airtime` class. This operator should be a `const` function, since that's what the `plus<>()` function object expects.

The other arithmetic function objects work in a similar way. The logical function objects such as `logical_and<>()` can be used on objects of classes for which these operations make sense (for example, type `bool` variables).

## Writing Your Own Function Objects

If one of the standard function objects doesn't do what you want, you can write your own. Our next example shows two situations where this might be desirable, one involving the `sort()` algorithm and one involving `for_each()`.

It's easy to sort a group of elements based on the relationship specified in the class < operator. However, what happens if you want to sort a container that contains pointers to objects, rather than the objects themselves? Storing pointers is a good way to improve efficiency, especially for large objects, because it avoids the copying process that takes place whenever an object is placed in a container. However, if you try to sort the pointers, you'll find that the objects are arranged by pointer address, rather than by some attribute of the object.

To make the `sort()` algorithm work the way we want in a container of pointers, we must supply it with a function object that defines how we want the data ordered.

Our example program starts with a vector of pointers to `person` objects. These objects are placed in the vector, then sorted in the usual way, which leads to the pointers, not the persons, being sorted. This isn't what we want, and in this case causes no change in the ordering at all, because the items were inserted in order of increasing addresses. Next, the vector is sorted correctly, using the function object `comparePersons()`. This orders items using the *contents* of pointers, rather than the pointers themselves. The result is that the `person` objects are sorted alphabetically by name. Here's the listing for SORTPTRS:

```
// sortptrs.cpp
// sorts person objects stored by pointer
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:
      person() :              //default constructor
           lastName("blank"), firstName("blank"), phoneNumber(0L)
         {  }
                              //3-arg constructor
      person(string lana, string fina, long pho) :
              lastName(lana), firstName(fina), phoneNumber(pho)
         {  }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);

      void display() const    //display person's data
         {
         cout << endl << lastName << ",\t" << firstName
              << "\t\tPhone: " << phoneNumber;
         }
      long get_phone() const //return phone number
         { return phoneNumber; }
   }; //end class person
//--------------------------------------------------------------
//overloaded < for person class
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
//--------------------------------------------------------------
//overloaded == for person class
bool operator==(const person& p1, const person& p2)
   {
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName ) ? true : false;
   }
//--------------------------------------------------------------
//function object to compare persons using pointers
class comparePersons
   {
   public:
   bool operator() (const person* ptrP1,
                    const person* ptrP2) const
      { return *ptrP1 < *ptrP2; }
   };
//--------------------------------------------------------------
//function object to display a person, using a pointer
class displayPerson
   {
   public:
   void operator() (const person* ptrP) const
      { ptrP->display(); }
   };
////////////////////////////////////////////////////////////////
int main()
   {                          //a vector of ptrs to persons
   vector<person*> vectPtrsPers;
                              //make persons
   person* ptrP1 = new person("KuangThu", "Bruce", 4157300);
   person* ptrP2 = new person("Deauville", "William", 8435150);
   person* ptrP3 = new person("Wellington", "John", 9207404);
   person* ptrP4 = new person("Bartoski", "Peter", 6946473);
   person* ptrP5 = new person("Fredericks", "Roger", 7049982);
   person* ptrP6 = new person("McDonald", "Stacey", 7764987);
   vectPtrsPers.push_back(ptrP1);  //put persons in set
   vectPtrsPers.push_back(ptrP2);
   vectPtrsPers.push_back(ptrP3);
   vectPtrsPers.push_back(ptrP4);
   vectPtrsPers.push_back(ptrP5);
   vectPtrsPers.push_back(ptrP6);

   for_each(vectPtrsPers.begin(),               //display vector
            vectPtrsPers.end(), displayPerson() );
                                                //sort pointers
   sort( vectPtrsPers.begin(), vectPtrsPers.end() );
   cout << "\n\nSorted pointers";
   for_each(vectPtrsPers.begin(),               //display vector
            vectPtrsPers.end(), displayPerson() );

   sort( vectPtrsPers.begin(),                  //sort persons
         vectPtrsPers.end(), comparePersons() );
   cout << "\n\nSorted persons";
   for_each(vectPtrsPers.begin(),               //display vector
            vectPtrsPers.end(), displayPerson() );
   while( !vectPtrsPers.empty() )
      {
      delete vectPtrsPers.back();               //delete person
      vectPtrsPers.pop_back();                  //pop pointer
      }
   cout << endl;
   return 0;
   }  //end main()
```

Here's the output of SORTPTRS:

```
KuangThu,       Bruce           phone: 4157300
Deauville,      William         phone: 8435150
Wellington,     John            phone: 9207404
Bartoski,       Peter           phone: 6946473
Fredericks,     Roger           phone: 7049982
McDonald,       Stacey          phone: 7764987

Sorted pointers
KuangThu,       Bruce           phone: 4157300
Deauville,      William         phone: 8435150
Wellington,     John            phone: 9207404
Bartoski,       Peter           phone: 6946473
Fredericks,     Roger           phone: 7049982
McDonald,       Stacey          phone: 7764987

Sorted persons
Bartoski,       Peter           phone: 6946473
```

```
Deauville,      William         phone: 8435150
Fredericks,     Roger           phone: 7049982
KuangThu,       Bruce           phone: 4157300
McDonald,       Stacey          phone: 7764987
Wellington,     John            phone: 9207404
```

First the original order is shown, then the ordering sorted incorrectly by pointer, and finally the order sorted correctly by name.

## The `comparePersons()` Function Object

If we use the two-argument version of the `sort()` algorithm

```
sort( vectPtrsPers.begin(), vectPtrsPers.end() );
```

then only the pointers are sorted, by their addresses in memory. This is not usually what we want. To sort the `person` objects by name, we use the three-argument version of `sort()`, with the `comparePersons()` function object as the third argument:

```
sort( vectPtrsPers.begin(),
      bectPtrsPers.end(), comparePersons() );
```

The function object `comparePersons()` is defined like this in the SORTPTRS program:

```
//function object to compare persons using pointers
class comparePersons
   {
   public:
   bool operator() (const person* ptrP1,
                    const person* ptrP2) const
      { return *ptrP1 < *ptrP2; }
   };
```

The `operator()` takes two arguments that are pointers to persons and compares their contents, rather than the pointers themselves.

## The `displayPerson()` Function Object

We use a different approach to display the contents of a container than we have before. Instead of iterating through the container, we use the `for_each()` function, with a function object as its third argument.

```
for_each(vectPtrsPers.begin(),
         bectPtrsPers.end(), displayPeson() );
```

This causes the `displayPerson()` function object to be called once for each person in the vector. Here's how `displayPerson()` looks:

```
//function object to display a person, using a pointer
class displayPerson
```

```
{
public:
void operator() (const person* ptrP) const
   { ptrP->display(); }
};
```

With this arrangement a single function call displays all the `person` objects in the vector.

## Function Objects Used to Modify Container Behavior

In SORTPTRS we showed function objects used to modify the behavior of algorithms. Function objects can also modify the behavior of containers. For example, if you want a set of pointers to objects to sort itself automatically based on the objects instead of the pointers, you can use an appropriate function object when you define the container. No `sort()` algorithm need be used. We'll examine this approach in an exercise.

## Summary

This chapter has presented a quick and dirty introduction to the STL. However, we've touched on the major topics, and you should have acquired enough information to begin using the STL in a useful way. For a fuller understanding of the STL we recommend that readers avail themselves of a complete text on the topic.

You've learned that the STL consists of three main components: containers, algorithms, and iterators. Containers are divided into two groups: sequential and associative. Sequential containers are the vector, list, and deque. Associative containers are the set and map, and the closely-related multiset and multimap. Algorithms carry out operations on containers, such as sorting, copying, and searching. Iterators act like pointers to container elements and provide connections between algorithms and containers.

Not all algorithms are appropriate for all containers. Iterators are used to ensure that algorithms and containers are appropriately matched. Iterators are defined for specific kinds of containers, and used as arguments to algorithms. If the container's iterators don't match the algorithm, a compiler error results.

Input and output iterators connect directly to I/O streams, thus allowing data to be piped directly between I/O devices and containers. Specialized iterators allow backward iteration and can also change the behavior of some algorithms so that they insert data rather than overwriting existing data.

Algorithms are standalone functions that can work on many different containers. In addition, each container has its own specific member functions. In some cases the same function is available as both an algorithm and a member function.

STL containers and algorithms will work with objects of any class, provided certain member functions, such as the < operator, are overloaded for that class.

The behavior of certain algorithms such as `find_if()` can be customized using function objects. A function object is instantiated from a class containing only an `()` operator.

## Questions

Answers to these questions can be found in Appendix G.

1. An STL container can be used to

   a. hold objects of class `employee`.

   b. store elements in a way that makes them quickly accessible.

   c. compile C++ programs.

   d. organize the way objects are stored in memory.

2. The STL sequence containers are v_______, l_______, and d________.
3. Two important STL associative containers are s_______ and ma_______.
4. An STL algorithm is

   a. a standalone function that operates on containers.

   b. a link between member functions and containers.

   c. a friend function of appropriate container classes.

   d. a member function of appropriate container classes.

5. True or false: One purpose of an iterator in the STL is to connect algorithms and containers.
6. The `find()` algorithm

   a. finds matching sequences of elements in two containers.

   b. finds a container that matches a specified container.

   c. takes iterators as its first two arguments.

   d. takes container elements as its first two arguments.

7. True or false: Algorithms can be used only on STL containers.
8. A range is often supplied to an algorithm by two i_______ values.
9. What entity is often used to customize the behavior of an algorithm?

10. A vector is an appropriate container if you
    a. want to insert lots of new elements at arbitrary locations in the vector.
    b. want to insert new elements, but always at the front of the container.
    c. are given an index number and you want to quickly access the corresponding element.
    d. are given an element's key value and you want to quickly access the corresponding element.
11. True or false: The `back()` member function removes the element at the back of the container.
12. If you define a vector `v` with the default constructor, and define another vector `w` with a one-argument constructor to a size of 11, and insert 3 elements into each of these vectors with `push_back()`, then the `size()` member function will return ______ for `v` and _____ for `w`.
13. The `unique()` algorithm removes all _________ element values from a container.
14. In a deque
    a. data can be quickly inserted or deleted at any arbitrary location.
    b. data can be inserted or deleted at any arbitrary location, but the process is relatively slow.
    c. data can be quickly inserted or deleted at either end.
    d. data can be inserted or deleted at either end, but the process is relatively slow.
15. In iterator ________ a specific element in a container.
16. True or false: An iterator can always move forward or backward through a container.
17. You must use at least a ____________ iterator for a list.
18. If `iter` is an iterator to a container, write an expression that will have the value of the object pointed to by `iter`, and will then cause `iter` to point to the next element.
19. The `copy()` algorithm returns an iterator to
    a. the last element copied from.
    b. the last element copied to.
    c. the element one past the last element copied from.
    d. the element one past the last element copied to.
20. To use a `reverse_iterator`, you should
    a. begin by initializing it to `end()`.
    b. begin by initializing it to `rend()`.
    c. increment it to move backward through the container.
    d. decrement it to move backward through the container.

21. True or false: The `back_inserter` iterator always causes the new elements to be inserted following the existing ones.
22. Stream iterators allow you to treat the display and keyboard devices, and files, as if they were _____________.
23. What does the second argument to an `ostream_iterator` specify?
24. In an associative container
    a. values are stored in sorted order.
    b. keys are stored in sorted order.
    c. sorting is always in alphabetical or numerical order.
    d. you must use the `sort()` algorithm to keep the contents sorted.
25. When defining a set, you must specify how ______________.
26. True or false: In a set, the `insert()` member function inserts a key in sorted order.
27. A map stores __________ of objects (or values).
28. True or false: A map can have two or more elements with the same key value.
29. If you store pointers to objects, instead of objects, in a container
    a. the objects won't need to be copied to implement storage in the container.
    b. only associative containers can be used.
    c. you can't sort the objects using object attributes as keys.
    d. the containers will often require less memory.
30. If you want an associative container such as `set` to order itself automatically, you can define the ordering in a function object and specify that function object in the container's ___________.

## Exercises

Answers to exercises can be found in Appendix G.

*1. Write a program that applies the `sort()` algorithm to an array of floating point values entered by the user, and displays the result.

*2. Apply the `sort()` algorithm to an array of words entered by the user, and display the result. Use `push_back()` to insert the words, and the `[]` operator and `size()` to display them.

*3. Start with a list of `int` values. Use two normal (not reverse) iterators, one moving forward through the list and one moving backward, in a `while` loop, to reverse the contents of the list. You can use the `swap()` algorithm to save a few statements. (Make sure your solution works for both even and odd numbers of items.) To see how the experts do it, look at the `reverse()` function in your compiler's ALGORITHM header file.

*4. Start with the `person` class, and create a multiset to hold pointers to `person` objects. Define the multiset with the `comparePersons` function object, so it will be sorted automatically by names of persons. Define a half-dozen persons, put them in the multiset, and display its contents. Several of the persons should have the same name, to verify that the multiset stores multiple objects with the same key.

5. Fill an array with even numbers and a set with odd numbers. Use the `merge()` algorithm to merge these containers into a vector. Display the vector contents to show that all went well.

6. In Exercise 3, two ordinary (non-reverse) iterators were used to reverse the contents of a container. Now use one forward and one reverse iterator to carry out the same task, this time on a vector.

7. We showed the four-argument version of the `accumulate()` algorithm in the PLUSAIR example. Rewrite this example using the three-argument version.

8. You can use the `copy()` algorithm to copy sequences within a container. However, you must be careful when the destination sequence overlaps the source sequence. Write a program that lets you copy any sequence to a different location within an array, using `copy()`. Have the user enter values for `first1`, `last1`, and `first2`. Use the program to verify that you can shift a sequence that overlaps its destination to the left, but not to the right. (For example, you can shift several items from 10 to 9, but not from 10 to 11.) This is because `copy()` starts with the leftmost element.

9. We listed the function objects corresponding to the C++ operators in Table 15.10, and, in the PLUSAIR program earlier in this chapter, we showed the function object `plus<>()` used with the `accumulate()` algorithm. It wasn't necessary to provide arguments to the function objects in that example, but sometimes it is. However, you can't put the argument within the parentheses of the function object, as you might expect. Instead, you use a function adapter called `bind1st` or `bind2nd` to bind the argument to the function. For example, suppose you were looking for a particular string (call it `searchName`) in a container of strings (called `names`). You can say

```
ptr = find_if(names.begin(), names.end(),
              bind2nd(equal_to<string>(), searchName) );
```

Here `equal_to<>()` and `searchName` are arguments to `bind2nd()`. This statement returns an iterator to the first string in the container equal to `searchName`. Write a program that incorporates this statement or a similar one to find a string in a container of strings. It should display the position of `searchName` in the container.

10. You can use the `copy_backward()` algorithm to overcome the problem described in Exercise 7 (that is, you can't shift a sequence to the left if any of the source overlaps any of the destination). Write a program that uses both `copy()` and `copy_backward()` to enable shifting any sequence anywhere within a container, regardless of overlap.

11. Write a program that copies a source file of integers to a destination file, using stream iterators. The user should supply both source and destination filenames to the program. You can use a `while` loop approach. Within the loop, read each integer value from the input iterator and write it immediately to the output iterator, then increment both iterators. The ITER.DAT file created by the FOUTITER program in this chapter makes a suitable source file.
12. A frequency table lists words and the number of times each word appears in a text file. Write a program that creates a frequency table for a file whose name is entered by the user. You can use a map of `string-int` pairs. You may want to use the C library function `ispunct()` (in header file CTYPE.H) to check for punctuation so you can strip it off the end of a word, using the string member function `substr()`. Also, the `tolower()` function may prove handy for uncapitalizing words.