17 Introduction to Data Structures Using the Standard Template Library

KNOWLEDGE GOALS

- To appreciate the distinction between the Standard Template Library (STL) and data structures in the abstract sense.
- To understand the operation of stacks, queues, and deques.
- To understand the set and map as abstract data types.
- To be aware of the structure and operations associated with tree and hash table abstract types.

SKILL GOALS

To be able to:

- Instantiate and use specializations of STL templates.
- Work with STL iterators.
- Apply STL algorithms to STL data structures.
- Use the basic STL sequence containers: `vector`, `list`, and `deque`.
- Use the STL sequence container adapters: `stack` and `queue`.
- Use the associative STL containers: `set` and `map`.

In Chapter 10, we began our exploration of user defined data types. In Chapter 11, we introduced the array. Chapters 13 and 14 focused on the list data structure as a vehicle for appreciating the general principles of how data can be organized in the abstract sense, and for recognizing how an ADT can have many different implementations that offer advantages under different circumstances. We are now ready to survey a broader range of ways to organize data.

The study of data structures is a major area of computer science research and education. Most computer science curricula include one or more complete courses on data structures, their related algorithms, and efficient implementation techniques. In this chapter, we can merely whet your appetite for those more advanced studies by giving you a preview of what's to come.

That doesn't mean, however, that we cannot delve into some practical uses of these more advanced structuring techniques. The C++ library provides a wealth of existing implementations of generic structured types, which we can use directly. In fact, the Standard Template Library (STL) is so rich that entire books have been devoted to explaining the subtleties of its operation! In this chapter, we concentrate on the basic principles and practical aspects of those STL types that support the more commonly used structured ADTs.

17.1 Abstract Data Structures versus Implementations

By this point you are quite familiar with the array and list data structures. We describe both of them as belonging to the family of linear data structures, because their components can be arranged in a straight sequence from one end to the other. The array is sequenced by its index values, and the list is sequenced by the logical relationship between each node and the nodes that immediately precede or follow it, as shown in **FIGURE 17.1**. As you know, lists may be implemented either with arrays or via pointers (linked lists).

With a little thought, you should also appreciate that nothing prevents us from implementing the abstract idea of the array using a linked list. Under normal circumstances, a linked implementation of an array would be inefficient. When an array with a large index range is sparsely filled with values, however, a linked implementation can save a lot of space. As an example, imagine that a wildlife biologist is plotting the location of geese on a lake using GPS coordinates received from the geese's tracking collars. The natural representation of this data is with a two-dimensional array indexed by the coordinates with each element representing the number of geese in that area of the lake. Of course, when there are only a few geese on the lake, an array implementation wastes a huge amount of memory space, which can be saved by switching to a linked implementation, as shown in **FIGURE 17.2**.

This discussion is simply meant to convey the idea that every data structure has two aspects: its abstract (conceptual) form and its implementation. Through the control and data abstraction supported in object-oriented programming using classes, it is possible to entirely divorce one of these aspects from the other. The advantage is that client code can use a natural representation that is easier to understand, while obtaining the benefits of an efficient underlying implementation.

As we proceed through this chapter, you might have thoughts such as, "But a stack is just a restricted kind of list. Why do we need it as a separate type?" We all fall into this trap of mixing abstract representation with implementation. It's true that once we have an implementation of a general list, it can do everything a stack can do. But as you'll see when we demonstrate the use of the stack, sometimes that simpler abstract type is all we need to solve a problem, and coding the solution is made easier by its simplicity. Underneath the stack interface, however, we can have either a list-based or an array-based implementation.

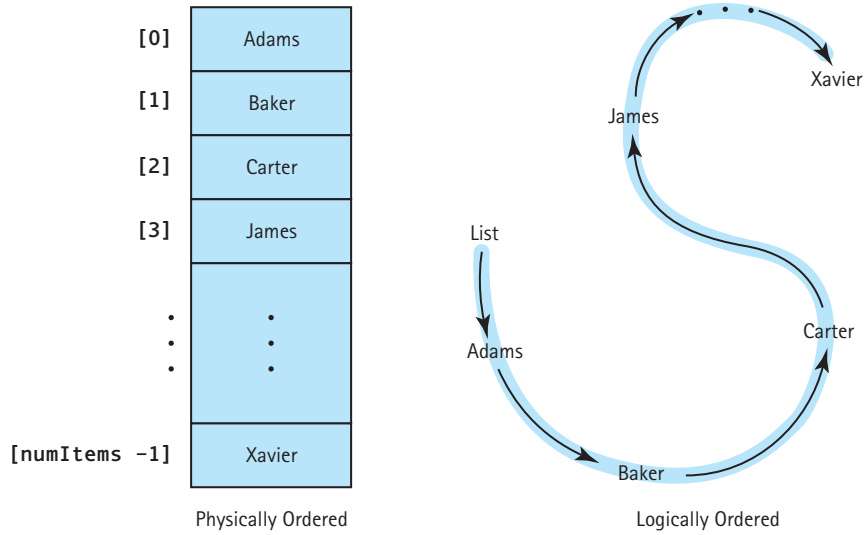


FIGURE 17.1 The Array and List as Linear Structures

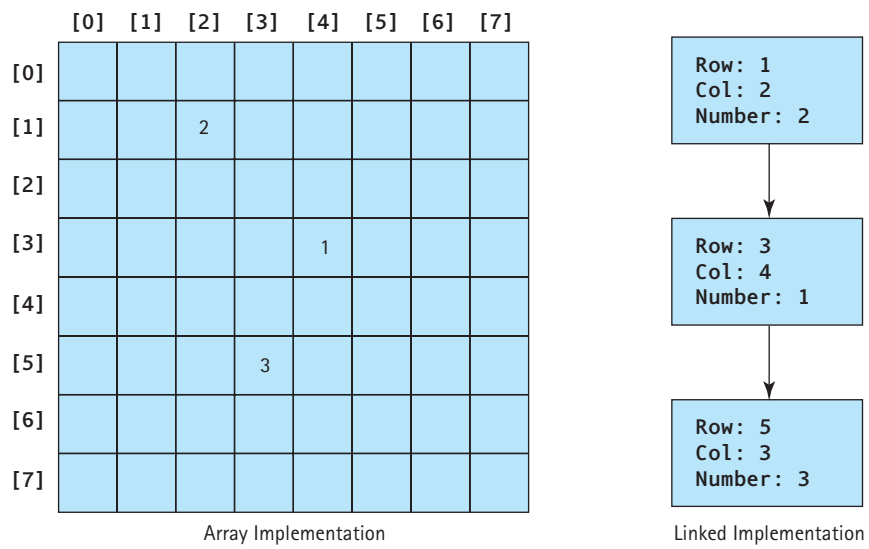


FIGURE 17.2 A Linked Implementation of a Sparsely Filled Array Conserves Space

Something else to keep in mind is that any implementation of a data structure is representing a corresponding ADT. For example, a `list` class has an interface that represents a list ADT. Because of this, when we look at a library of implementations such as the STL, it is tempting to automatically use the provided linked `list` type to implement the abstract list in a problem. But what if the problem requirements specify that the list is built from file data at the start of the run, with few subsequent deletions and additions? That argues for an array-based implementation.

The problem-solving strategy we use with data structures is as follows. In the problem-solving phase, we design the structure and interface that most naturally solves the problem. In the implementation phase, we consider how the structure will be used and which kind of structure actually provides the most efficient solution. Next we define a class that wraps the implementation structure with an interface for the abstract type. Lastly, we go to the library to see if it provides a suitable data structure implementation that can be used within the class.

Having just spelled out this strategy, we need to note that many of the goals of this chapter involve demonstrating the use of the STL. It's much easier to see how an STL type works without wrapping it in an interface for a different type. So be forewarned that our examples aren't necessarily meant to illustrate this approach: This is definitely a case of "Do as we say, not as we do!"

17.2 Additional Linear Structures

Outside of the context of a specific problem, it's difficult to imagine how we could invent a linear data structure that's not conceptually an array or a list—so let's consider an example problem. Do shopping carts at the supermarket wear out evenly? You've seen them arranged in their neat columns, with people taking them from the back, and clerks wheeling them back into place. But do the ones at the front of the column ever get used? Are they rearranged enough so that all carts get the same amount of wear in the long run? Or is there a reason that you always seem to get one with a sticky wheel that won't roll straight?

Other than curiosity, understanding how carts are used can help a supermarket decide how many to buy. If it turns out that they don't wear evenly, a manager might decide to have a clerk manually rearrange the carts periodically.

Solving such a problem involves simulation. We can model the carts in their rows and the time that they spend roaming the aisles, and record how much total time of use each one acquires. The speed of the computer enables us to evaluate years of use in a matter of seconds.

Which data structures should we use to represent this situation? Given what we've seen so far, an array would be a good choice for modeling a column of carts. When a simulated customer arrives, we remove a cart from the back end of the column. Then what do we do with it? We can randomly assign it a return time and put it into a sorted list. As each minute passes, we check the list to see if the cart at the head of the list is ready to be returned. Then we remove it from the head of the list, and place it at the back of the column array. **FIGURE 17.3** shows this arrangement.

Let's consider these structures more closely. Do we really need all of their capabilities to solve our problem? Not really. For example, we won't index into the array in random

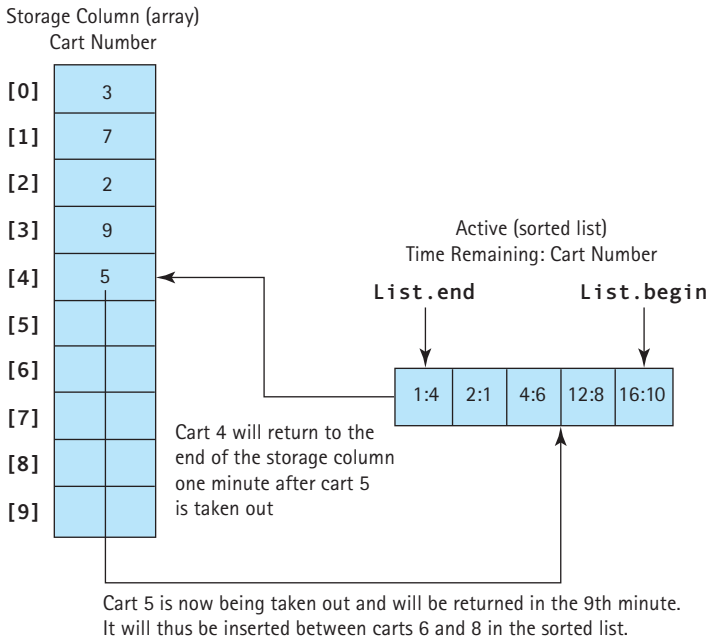


FIGURE 17.3 Using an Array and a Sorted List to Simulate Shopping Carts

locations; instead, we merely add and delete elements from one end of the array. In computer science, such a structure is called a stack. What about the list? Do we ever traverse it from one end to the other, or delete elements from the middle? No, we just insert elements into place and remove them from one end. This structure, which is called a priority queue (pronounced like the letter "Q"), is one member of a broader family of queue data structures.

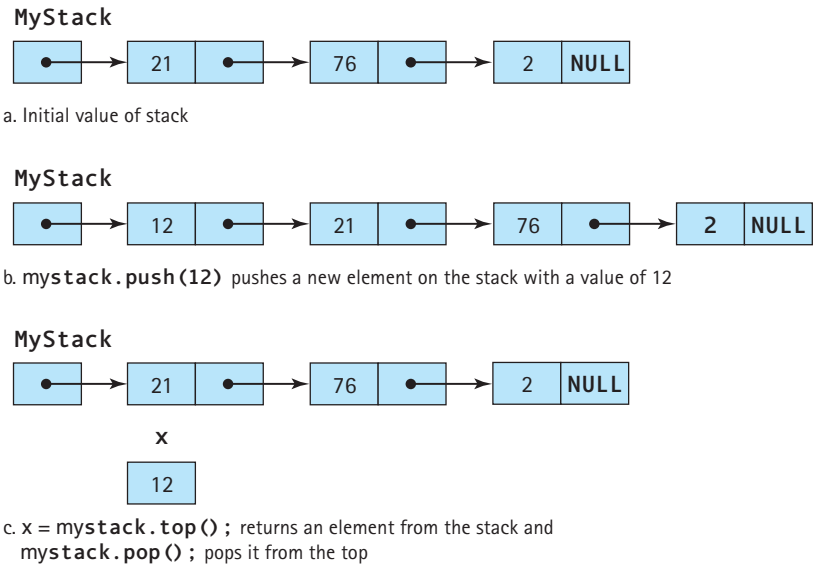
At this point, you should start to see that differences in linear data structures are based on how they are accessed. Arrays and lists support very general access. If access is restricted in some way, however, we can define a data structure with a simpler interface. Also, in some cases, the restrictions permit us to create a more efficient implementation.

Let's take a closer look at stacks and queues.

Stacks

A stack is a data structure that can be accessed from only one end. We can insert an element at the top (as the first element) and we can remove the top (first) element. This structure models a property commonly encountered in real life. Accountants call it LIFO, which stands for "last in, first out." The plate holder in a cafeteria has this property. You can take only the top plate. When you do, the one below it rises to the top so the next person can take a plate. Cars in a noncircular driveway exhibit this property: The last car in must be the first car out.

Stack A data structure in which insertions and deletions can be made from only one end.



a. Initial value of stack

b. `mystack.push(12)` pushes a new element on the stack with a value of 12

c. `x = mystack.top();` returns an element from the stack and `mystack.pop();` pops it from the top

FIGURE 17.4 A Stack, Showing Push, Top, and Pop Operations

The term **push** is used for the insertion operation, and the term **pop** is used for the deletion operation. In some implementations of a stack, **pop** also returns the top element. In other implementations, a separate operation, **top**, retrieves the top value and **pop** simply deletes it. **FIGURE 17.4** shows what happens when you **push** an element on a given stack and then **pop** the stack.

Stacks are used frequently in systems software. For example, C++ uses a stack at run time to keep track of nested method calls. The compiler uses a stack to translate arithmetic expressions. Stacks are used whenever we wish to remember a sequence of objects or actions in the reverse order from their original occurrence. An algorithm to read in a file containing a series of lines and print the lines out in reverse order using a stack of strings is shown here:

Reverse File

```
Create stack
Create file in
while in
   stack.push(in.getline)
while not stack.empty
   print stack.top
   stack.pop
```

As you can see from this algorithm, in addition to **push**, **pop**, and **top**, we need a way to test whether the stack is empty. We may also want a way to find the size of a stack. Even

with those additions, the stack interface has only five responsibilities, making it a very simple abstract type.

Queues

A queue is a data structure in which elements are entered at one end and removed from the other end. Accountants call this property FIFO, which stands for "first in, first out." A waiting line in a bank or supermarket and a line of cars on a one-way street are types of queues.[1] Indeed, queues are often used in computer simulations of similar situations.

Queue A data structure in which insertions are made at one end and deletions are made at the other.

Whereas the terminology for the insert and remove responsibilities on stacks is standard (**push**, **pop**), no such standardization exists with queues. The operation of inserting an element at the rear of the queue has many names in the literature: Insert, add, push, and enqueue are four common ones. Correspondingly, the operation for removing from the front of the queue is variously called delete, remove, pop, and dequeue.

C++ uses stack terminology for queues, where **push** applies to the back of the structure and **pop** applies to the front. In place of **top**, the operation **front** returns the element at the head of the queue. As with the stack, we need operations to check for an empty queue and to determine the size of a queue. The C++ implementation of a queue is atypical in that one more operation, **back**, returns the element at the rear of the queue. **FIGURE 17.5** shows an empty queue (a), insertion into a queue (b), and deletion from a queue (c).

Priority Queues

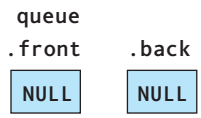
As we said in our shopping cart problem, a priority queue is a member of the queue family of structures. It behaves like a queue in that data is removed only from its front. Instead of inserting elements at the rear, however, elements are inserted so that they are in some given order, as shown in **FIGURE 17.6**. For example, passengers boarding an airplane are often called to line up in order of their seat assignments. Schoolchildren preparing to have their portraits taken may be told to line up in alphabetical order. In each case, the people enter the line at whatever point puts them in the proper order, but they leave the line only at the front.

Priority queue A data structure in which insertions are made according to a specified ordering and deletions are made at one end.

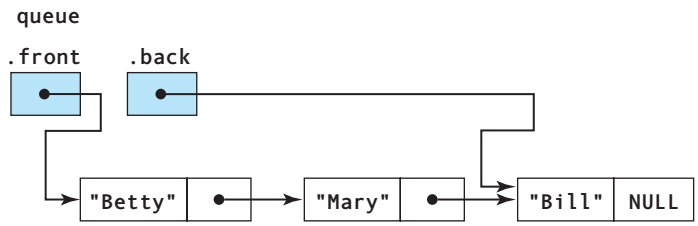
As with queues, there is no standard terminology for priority queue responsibilities. C++ simply uses the stack terminology: **push**, **pop**, **top**, **empty**, and **size**. As a result, it is important to be careful in naming stack and priority queue objects so that it's easy to tell what kind of object each one is. Unlike with other structures, we can't rely on the operation names to help us identify the data type.

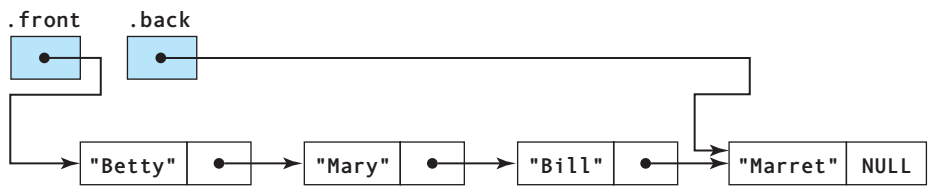
1. The word *queue* originates from French, but is in more widespread use around the globe. For example, where Americans would ask people to "line up," the British would ask them to "queue up."

a. An empty queue



b. Insertion into a queue





c. Deletion from a queue

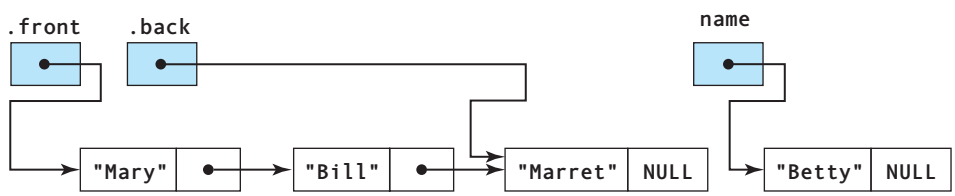


FIGURE 17.5 A Queue, Showing Insertion and Deletion Operations

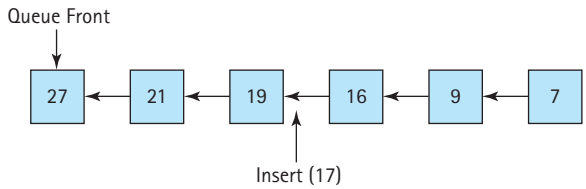


FIGURE 17.6 A Priority Queue, ADT Showing Insertion and Removal Points

Here we should reiterate a key point made in Section 17.1: Data structures have a conceptual structure and an implementation structure, which may differ. We have just described the priority queue from the conceptual perspective. The STL typically uses a more complex and more efficient structure to implement the priority queue. As we describe other data structures in this chapter, keep in mind that we are looking at their conceptual arrangement. Except for a few cases, we do not describe how the STL implements.

17.3 Bidirectional Linear Structures

Like the linked lists discussed in Chapter 14, queues are unidirectional structures. We traverse our lists from front to back. Elements in queues move from back to front. In a linked implementation, the components of the structures carry a link member that points to the next component. Even though elements in a stack may move down and up, they, too, can be implemented by links that point from the top down. It's more difficult to traverse the elements of these structures against the direction of their chain of links—analogous to petting a cat's fur in the wrong direction.

Bidirectional Lists

When we want a structure that can be traversed in either direction, an array is a natural choice. But when insertions and deletions in the middle of the data structure are common and random access is not as important, the array is appropriate neither as an ADT nor as an implementation. Instead, we want a list structure that provides for bidirectional traversal. Such a structure is known as a bidirectional list.

Bidirectional list A list data structure in which traversals can be made in either direction.

In a linked implementation of the bidirectional list, we augment our list nodes with a second link that points in the opposite direction. We keep external references to the front and the back of the list, and we can step through the nodes by going either forward or backward, as shown in **FIGURE 17.7**.

With this structure, every insertion and every deletion require twice as many pointer operations, and the nodes have twice as much space allocated to pointer members, so there

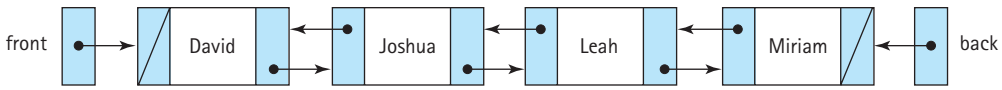


FIGURE 17.7 A Bidirectional Linked List

is an overall loss of efficiency. Thus, even though the bidirectional list is a more general structure with many attractive capabilities, it should be reserved for use only in those cases that need its extra features. Just as you wouldn't trim your houseplants with a chainsaw, similarly there are many places where a bidirectional linked list is overkill. Having said that, be aware that the STL provides only a bidirectional implementation of the list ADT. Instead of a unidirectional list, it provides similar functionality with the **`set`** type, which has a much different implementation that we describe later in the chapter.

Deques

Deque A data structure in which insertions and deletions are made at either end.

A deque (pronounced like "deck") is the bidirectional equivalent to the queue. With it, we can insert and delete elements at either end, as shown in **FIGURE 17.8**. In fact, the name is a word invented by computer scientists as a contraction of double-ended queue. If you want to simulate a waiting line in which people get frustrated and also leave from the back, a deque would be a good choice. It could also represent a train of railroad cars in a switchyard, where cars can be added and removed from either end of the train.

There is no consensus on the naming of deque operations. C++ uses **`push_front`** and **`push_back`** for insertion, **`pop_front`** and **`pop_back`** for deletions, and **`front`** and **`back`** for accessing the head and tail values. As we see when we look at the STL implementation, the C++ deque is really one huge class that combines a combination of the interfaces for the abstract bidirectional list, the deque, and the array.

Now it's time to take a look at the STL. We will focus on how it implements the interfaces for these linear ADTs—that is, how it implements the conceptual view of them. Data and control abstraction allow us to write whatever client code is needed to make use of these structures, without concerning ourselves about their internal implementations. We will return to our survey of abstract data structures later in the chapter, when we shift our focus to nonlinear organizations of data.

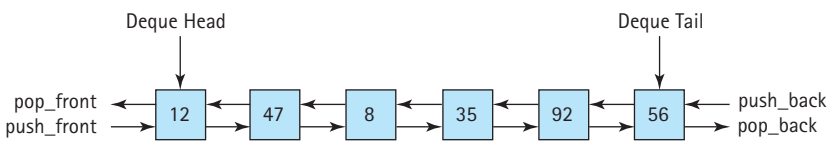


FIGURE 17.8 A Deque (Double-Ended Queue)

17.4 An Introduction to the STL

As the name Standard Template Library implies, the STL is made up primarily of templates. There are two main sections of the library: containers and algorithms. The containers portion has templates for 11 generic structured data types. Three of these data types are beyond the scope of this text: **bitset**, **multiset**, and **multimap**. The other 8 are summarized in the following table.

Container The name used with respect the C++ STL to refer to its generic data structures that hold an arbitrary number of elements.

Name	Description
deque	A double-ended queue. Also has bidirectional list and array (direct random access via an index) functionality.
list	A bidirectional list with similar capabilities to **deque**, but no direct index access.
map	Associates a lookup value (the key) with a value to be looked up. An example use would be a telephone directory.
priority_queue	A priority queue of elements ordered by a key value.
queue	A simple FIFO queue of elements.
set	An ordered collection of values that can be searched. Duplicates are not allowed. Similar in capability to the sorted list implemented in Chapter 13.
stack	A simple LIFO push-down stack of elements.
vector	Equivalent to an array, but able to change its size at runtime.

To declare and specialize any of these template classes, simply use its name, followed by the type in angle brackets. For example:

```
list<string> strList;       // Create an empty list of strings
vector<int> intVec;         // Create an empty vector of ints
```

We'll see different forms of constructors for each type as we look at them individually. The type of a container's elements can be either a built-in type or a class. The STL does impose some requirements on element objects. In particular, they must support at least a minimal subset of the overloaded relational operators. For most compilers, overloading < and == is sufficient, but a few require more of the operators to be overloaded.

The algorithms portion of the STL has a large number of function templates that can be used in many different contexts. In this text, we cover just a few of the algorithms in detail. The following table summarizes some of the more commonly used functions. Not all of these can be applied to every type of container.

Name	Description of Operation
`copy`	Copies a given range of elements in a container.
`copy_backward`	Copies a given range of elements in a container in reverse order.
`equal`	Compares a range of elements in two containers for equality.
`find`	Searches a range of elements in a container for a match with a specified value.
`max`	Given two elements comparable with the <, operator returns the greater one.
`min`	Given two elements comparable with the <, operator returns the lesser one.
`random_shuffle`	Randomly reorders the elements in a container within a specified range.
`replace`	Searches a container over a given range for elements that match a given value, replacing them with a second given value.
`search`	Searches a container within a given range for a series of elements that match a series from another container. Can be thought of as a generic equivalent of the string `find` operation for containers.
`set_difference`	Given two sets, returns their difference—that is, the values in the first set that are not present in the second set. Can be applied over specified ranges within each set.
`set_intersection`	Given two sets, returns their intersection—that is, the values in the first set that are also present in the second set. Can be applied over specified ranges within each set.
`set_union`	Given two sets, returns their union—that is, a set containing all of the values from both sets, but without duplicates. Can be applied over specified ranges within each set.
`sort`	Sorts the elements of a container within a given range.
`transform`	Applies a supplied function to all of the elements of a container within a given range.

We will look in more detail at each of the container classes, bringing in some of these algorithms as examples. Keep in mind that we are just skimming the surface of the capabilities of these templates so that you can get a sense of how they work. For a deeper treatment, you should consult one of the dedicated STL reference books.

Iterators

Before we begin, we must explore one key concept associated with container classes: the iterator. We've discussed iterators in the past, as one kind of responsibility a class can have. In the STL, an iterator is an object that points to some place in a container. One way to think of an iterator is as a pointer that has been wrapped in a class to provide additional functionality. It is called an iterator because the extra functionality includes operations that cause it to move in a step-by-step fashion through an STL container.

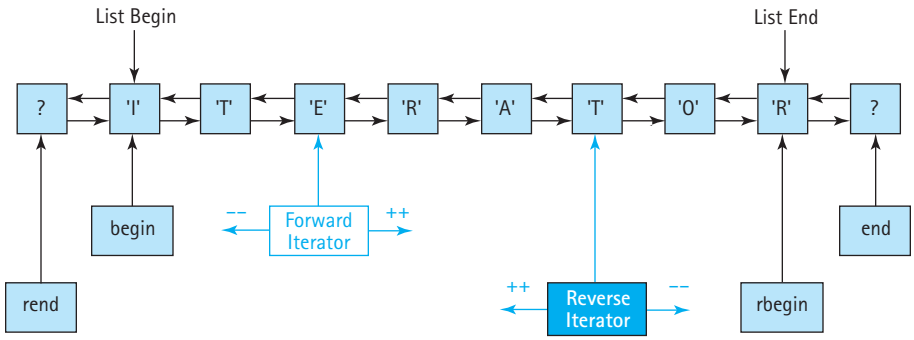


FIGURE 17.9 Forward and Reverse Iterators Associated With a List

Iterators in the STL come in two flavors: forward and reverse.[2] Typically, a container has methods called **`begin`** and **`end`** that return forward iterators pointing to its beginning and its end, respectively. You can use the `++` operator to advance a forward iterator through the container. Using `--` shifts the iterator backward.

Methods called **`rbegin`** and **`rend`** return reverse iterators. The iterators returned by **`rbegin`** and **`rend`** point to the end and the beginning of a container, respectively. Using `++` with a reverse iterator moves it from the back toward the front, and applying `--` causes a move toward the back. That is, the semantics of a reverse iterator are the opposite of those of a forward interator. A reverse iterator begins at the end of a container and "advances" toward the front. **FIGURE 17.9** illustrates this relationship.

The design of these iterators is meant to make it especially easy to traverse a container with a For loop. What **`end`** returns is actually an iterator referring to a point just beyond the last element. Similarly, the iterator returned by **`rend`** refers to a position just before the start of the container. As we'll see, this odd-sounding convention simplifies the termination condition in For loops. Be aware, however, that you cannot use the values returned by **`end`** and **`rend`** to directly access the ends of the container.

With random-access containers (such as **`deque`** and **`vector`**), the arithmetic operators `+` and `-` can be applied to iterators, just as we would use them with array indexes to compute the location of another element. Many C++ programmers think of an iterator as just another kind of pointer. In reality, the STL classifies a pointer the other way around—as a kind of iterator. Keep in mind that an iterator is an object. Not only does it provide extra operations, but it may also have restrictions that constrain its capabilities as compared to the capabilities of a pointer. For example, a unidirectional data structure limits its iterators to forward motion alone.

One very important aspect of iterators is the fact that they remain valid only as long as the contents of the container do not change. When an element is inserted into or deleted from a container, its iterators become invalid because C++ doesn't guarantee how the contents of a container are arranged after a change has occurred. The idea is that STL implementors

2. The STL actually defines five types of iterators. To keep things simple, however, we focus on just these two kinds of iterator behavior.

are free to reorganize the data so as to improve efficiency. The container class cannot keep track of every iterator that's pointing to it, so there is no way to update these iterators when a change takes place.

Each iterator must be declared for the specific type of container with which it will be used. For example, if we need iterators for a list of strings, we would declare them as follows:

```
list<string>::iterator place;          // Create a forward iterator
list<string>::reverse_iterator rplace; // Create a reverse iterator
```

The `vector` Template

Our tour of the STL's containers begins with one that, in the abstract, is very easy to understand. The **`vector`** class template implements an array. Why would we want another way to define an array? Well, there is one more thing about **`vector`**—it doesn't have to be fixed in size. As a consequence, we don't have to worry about subarray processing or decide how big to make the array when we declare it. Otherwise, however, you can almost directly substitute this container for the built-in array.

We say "almost" because a **`vector`** is an object rather than a built-in structure. Thus it must be instantiated with a constructor at run time. For this reason, we cannot declare it using an initialization list, as we can an array. Here are some simple calls to **`vector`** constructors:

```
vector<int> intVec(20);     // A vector holding 20 integers,
                            //  all initialized to 0
vector<string> noSizeVec;   // A vector holding 0 strings
```

The first statement makes sense—it's similar to writing this code:

```
int intVec[20];
```

But why would we want to declare a **`vector`** with no space in it? The reason is that a **`vector`** automatically expands to hold as much data as we put into it. Thus, if we don't have any idea of how many strings we will put into **`noSizeVec`**, we simply declare it with no elements and let the structure grow as we insert values.

If a **`vector`** always adjusts its size to hold the data, why do we ever need to bother with specifying a size? The answer is that this flexibility comes at a price. Every time a **`vector`** grows, it can take quite a bit of work. The underlying implementation is an array, and the template simply allocates a new, larger array and moves the old contents into it. Thus, when we know a maximum size in advance, it makes sense to indicate that value in the constructor.

Let's see an example of a **`vector`** in action. In Chapter 11, we wrote a demonstration program that allows the user to look up the number of occupants in a given apartment. Here is that program rewritten to use a **`vector`** directly with the changes highlighted. Recall that we fixed the number of apartments at 10.

```
//*****************************************************************
// This program allows a building owner to look up how many
// occupants are in a given apartment
//*****************************************************************
#include <iostream>
#include <fstream>                // For file I/O
#include <vector>                 // For vector template
using namespace std;

const int BUILDING_SIZE = 10;     // Number of apartments

int main()
{
  vector<int> occupants(BUILDING_SIZE);  // occupants[i] is the number of
                                         //   occupants in apartment i
  int totalOccupants;             // Total number of occupants
  int counter;                    // Loop control and index variable
  int apt;                        // An apartment number
  ifstream inFile;                // File of occupant data (one
                                  //   integer per apartment)
  inFile.open("apt.dat");
  totalOccupants = 0;
  for (counter = 0; counter < BUILDING_SIZE; counter++)
  {
    inFile >> occupants[counter];
    totalOccupants = totalOccupants + occupants[counter];
  }
  cout << "No. of apts. is " << BUILDING_SIZE << endl
       << "Total no. of occupants is " << totalOccupants << endl;

  cout << "Begin apt. lookup..." << endl;
  do
  {
    cout << "Apt. number (1 through " << BUILDING_SIZE
         << ", or 0 to quit): ";
    cin >> apt;
    if (apt > 0)
      cout << "Apt. " << apt << " has " << occupants[apt-1]
           << " occupants" << endl;
  } while (apt > 0);
  return 0;
}
```

As you can see, the **vector** almost directly replaces the array. Because the class overloads the **[]** operator, we can access the elements using index notation, both in storing values in the **vector** and in retrieving them. Our example, however, doesn't fully illustrate the potential power of the **vector** template. Let's rewrite the program so that it grows the **vector** to hold as much apartment data as is on the input file.

Of course, we have to change the loop from a For to a While, testing the state of the `inFile` stream and using a priming read. The constant `BUILDING_SIZE` is no longer useful, so we remove it. But how do we determine the size value to output? We use the `vector size` method. For example:

```
cout << "No. of apts. is " << occupants.size() << endl
```

Well, that's simple enough. But what about adding elements to the `vector`? For that we have to call the `push_back` method, which appends the element passed as its argument to the end of the `vector`, increasing its size if necessary.

In the original program, we didn't check whether the apartment number input by the user was in the valid range. We could use the `size` method again, with an If statement that guards the access to `occupants[apt-1]`, but that solution also ignores another useful feature of `vector`. If we use the `at` method instead of index access, the range is automatically checked, and we can easily catch an `out_of_range` exception (remember that we have to include `<stdexcept>` to access this exception class).

Here then, is the revised code with the changes highlighted, followed by a new data file containing 18 values, and the output from a sample run.

```
//*********************************************************************
// This program allows a building owner to look up how many
// occupants are in a given apartment
//*********************************************************************
#include <iostream>
#include <fstream>                // For file I/O
#include <vector>                 // For vector template
#include <stdexcept>              // For out_of_range exception
using namespace std;

int main()
{
  vector<int> occupants;          // occupants vector with 0 elements
  int totalOccupants;             // Total number of occupants
  int apt;                        // An apartment number
  int aptOccupants;               // Occupants in one apartment
  ifstream inFile;                // File of occupant data (one
                                  //   integer per apartment)
  inFile.open("apt.dat");
  totalOccupants = 0;
  inFile >> aptOccupants;
  while (inFile)
  {
    occupants.push_back(aptOccupants);
    totalOccupants = totalOccupants + aptOccupants;
    inFile >> aptOccupants;
  }
  cout << "No. of apts. is " << occupants.size() << endl
       << "Total no. of occupants is " << totalOccupants << endl;

  cout << "Begin apt. lookup..." << endl;
```

```
  do
  {
    cout << "Apt. number (1 through " << occupants.size()
         << ", or 0 to quit): ";
    cin >> apt;
    if (apt > 0)
      try
      {
         cout << "Apt. " << apt << " has " << occupants.at(apt-1)
              << " occupants" << endl;
      }
        catch(out_of_range)
      {
        cout << "Invalid apartment number. Enter ";
      }

  } while (apt > 0);
  return 0;
}
```

Data file:

```
3 4 0 3 4 1 1 2 3 2 1 4 2 3 0 5 2 3
```

Test run:

```
No. of apts. is 18
Total no. of occupants is 43
Begin apt. lookup...
Apt. number (1 through 18, or 0 to quit): 4
Apt. 4 has 3 occupants
Apt. number (1 through 18, or 0 to quit): 14
Apt. 14 has 3 occupants
Apt. number (1 through 18, or 0 to quit): 18
Apt. 18 has 3 occupants
Apt. number (1 through 18, or 0 to quit): 19
Invalid apartment number. Enter Apt. number (1 through 18, or 0 to quit): 15
Apt. 15 has 0 occupants
Apt. number (1 through 18, or 0 to quit): 0
```

The preceding example demonstrates just a few of the operators available with **vector**. The following table summarizes these and lists additional operators that are commonly used:

Name	Parameters	Returns	Description
at	**int**	reference	Returns a reference to the value at the specified location, checking the range.
back		reference	Returns a reference to the last element.
begin		**iterator**	Returns an iterator to the first element.
capacity		**int**	Returns the number of elements that can be held.
clear			Removes all elements. Sets **size** to 0.
empty		**bool**	Returns **true** if there are no elements in the **vector**.
end		**iterator**	Returns an iterator beyond the end of the **vector**.
erase	**iterator**		Removes the element at the iterator position. The **size** decreases.
front		reference	Returns a reference to the first element.
insert	**iterator**, value		Inserts the value at the location specified by the iterator. The **size** increases.
push_back	value		Inserts the value at the end of the vector. The **size** increases.
rbegin		**reverse_iterator**	Returns a reverse iterator to the last element.
rend		**reverse_iterator**	Returns a reverse iterator before the start of the **vector**.
reserve	**int**		Indicates that the **capacity** of the **vector** should increase to the amount specified.
size		**int**	Returns the number of elements in the **vector**.

There are a few additional things to note about these methods. The **at** method returns a reference that can be used for either access or assignment. That is, we can also write

```
occupants.at(3) = 5;
```

to set the fourth element of the **vector** to *5*, with range checking of the index.

The `capacity` of a `vector` is not the same as its `size`. The `capacity` is the number of spaces allocated for elements. In simple cases, the values of `capacity` and `size` may be the same. We can also use `reserve` to increase the `capacity` beyond the `size`, however. The advantage of doing so is that as the number of elements grows, the `vector` doesn't have to allocate larger and larger arrays, in addition to copying data between them. For example, if we insert

```
occupants.reserve(20);
```

after declaring `occupants`, then our program would not have to do any reallocations or make any copies of the data set we gave it. The test run would still show that `size` is 18, but if we were to insert

```
cout << occupants.capacity();
```

in the program, the value 20 would be output. **FIGURE 17.10** shows a `vector<char>` for which 10 spaces have been reserved and 6 elements have been inserted.

This is also different from calling the `vector` constructor with a number of elements specified, as we did in the previous version of our program:

```
vector<int> occupants(BUILDING_SIZE);
```

In this case, `occupants` is constructed with 10 elements, all initialized to the default value for the type (0 for `int`). Thus its initial `size` and `capacity` are both 10.

One other form of constructor is particularly useful. We may not be able to use an initializer list with a `vector`, but we can initialize it from an array of values. For example:

```
int occData[] = {3,4,0,3,4,1,1,2,3,2,1,4,2,3,0,5,2,3};
vector<int> occupants(occData, occData+sizeof(occData)/sizeof(int));
```

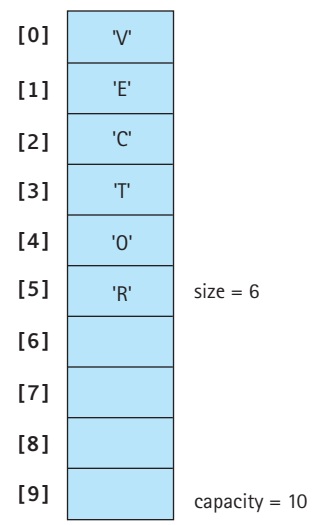


FIGURE 17.10 A `vector<char>` with `capacity` 10, After Insertion of 6 Elements

This constructor takes two iterators as its parameters. Recall that whenever we pass an array as an argument, what actually gets sent to the parameter is the memory address of the first element (the base address). Thus the first argument, `occData`, is the base address of the already initialized array. The second argument,

```
occData+sizeof(occData)/sizeof(int)
```

is the address that is one greater than the location of the last element of the array. The formula first determines the number of elements in `occData` by dividing its size (in bytes) by the size of its element (in bytes). The number of elements is added to the address for the beginning of the array to produce the address for the end of the array. C++ automatically casts these addresses into iterators of the appropriate type as they are passed to the constructor. The constructor is designed so that the address just beyond the end of the array is a sentinel value—it doesn't copy that location into the vector.

Lastly, the `vector` template provides a copy constructor that lets us simply pass another `vector` as an argument, and a new `vector` is created with the same size and content.

The `list` Template

Now that we've seen how `vector` works, we can quickly introduce the remaining containers. Many of their operations have similar functionality. For example, `list` also defines `back`, `begin`, `clear`, `empty`, `end`, `erase`, `front`, `insert`, `push_back`, `rbegin`, `rend`, and `size`, with essentially the same meanings as for `vector`. Because it is a linked implementation, `list` does not define a `capacity` or `reserve`. Also, because it doesn't support random access, the `[]` operator is not overloaded, nor is the `at` operation supported.

The same four forms of constructors are also defined for `list`. That is, we can create a `list` with zero elements, a `list` with a specific number of elements initialized to a default value, a `list` initialized from an array or another container, or a `list` created by using a copy constructor. The following table describes some additional operators that `list` supports.

Name	Parameters	Description
`merge`	`list`	Inserts the elements of the parameter `list` into the current `list`, in order. The elements are deleted from the parameter `list`. If the `lists` were already sorted, the result is a sorted `list` containing both sets of elements.
`pop_back`		Removes the last element.
`pop_front`		Removes the first element.
`push_front`	value	Inserts the value as the first element.
`remove`	value	Removes all elements with the specified value.
`reverse`		Reverses the order of the elements in the `list`.
`sort`		Sorts the elements from least to greatest.
`unique`		Removes all but the first element in any group of adjacent elements with the same value. If the `list` is sorted, the result is that it contains only unique values.

To see how `list` works, we present the following nonsense example program that creates a `list` from user input, displays its contents in different ways, applies various mutator operations, and merges it with another `list`. Here is the code:

```
//*********************************************************
// This program demonstrates various list methods
//*********************************************************
#include <iostream>
#include <list>
using namespace std;

int main ()
{
  list<string> demo;             // Create an empty list
  string word;
  cout << "Enter a line with six words:" << endl;
  for (int i = 1; i <= 6; i++)
  {
    cin >> word;
    demo.push_back(word);        // Insert elements at back
  }
  // Access front, back, and size
  cout << "Front element: " << demo.front() << endl
       << "Back element: " << demo.back() << endl
       << "Size of list: " << demo.size() << endl;
  // Create a forward iterator
  list<string>::iterator place;
  cout << "List contents from beginning to end: " << endl;
  // Traverse list in forward direction
  for (place = demo.begin(); place != demo.end(); place++)
    cout << *place << " ";
  // Create a reverse iterator
  list<string>::reverse_iterator rplace;
  cout << endl << "List contents from end to beginning: " << endl;
  // Traverse list in backward direction
  for (rplace = demo.rbegin(); rplace != demo.rend(); ++rplace)
    cout << *rplace << " ";
  cout << endl << "Enter a word to insert after the first word: ";
  cin >> word;
  place = demo.begin();          // Point forward iterator to front
  ++place;                       // Advance one place
  demo.insert(place, word);      // Insert an element
  place = demo.end();            // Point forward iterator past end
  --place;                       // Move back to last element
  --place;                       // Move back one more place
  demo.erase(place);             // Delete element
  cout << "Next to last word has been erased." << endl;
  cout << "Enter a word to add at the front: ";
  cin >> word;
  demo.push_front(word);         // Insert at front
  cout << "List contents from beginning to end: " << endl;
```

```
  for (place = demo.begin(); place != demo.end(); ++place)
    cout << *place << " ";
  demo.sort();                  // Sort the elements
  cout << endl
       << "After sorting, list contents from beginning to end: "
       << endl;
  for (place = demo.begin(); place != demo.end(); ++place)
    cout << *place << " ";
  // Create a three element list from an array of strings
  string init[] = {"large", "medium", "small"};
  list<string> demo2(init, init+sizeof(init)/sizeof(string));
  cout << endl << "After merging with: ";
  for (place = demo2.begin(); place != demo2.end(); ++place)
    cout << *place << " ";
  demo.merge(demo2);            // Merge the two lists
  cout << endl << "List contents are: " << endl;
  for (place = demo.begin(); place != demo.end(); ++place)
    cout << *place << " ";
  return 0;
}
```

Here is the output from a test run:

```
Enter a line with six words:
quick fox jumped over lazy dog
Front element: quick
Back element: dog
Size of list: 6
List contents from beginning to end:
quick fox jumped over lazy dog
List contents from end to beginning:
dog lazy over jumped fox quick
Enter a word to insert after the first word: brown
Next to last word has been erased.
Enter a word to add at the front: the
List contents from beginning to end:
the quick brown fox jumped over dog
After sorting, list contents from beginning to end:
brown dog fox jumped over quick the
After merging with: large medium small
List contents are:
brown dog fox jumped large medium over quick small the
```

The `stack` Template

Unlike **`list`**, which provides a very general set of container operations the goal of the **`stack`** type is to model a LIFO push-down stack as simply as possible. It has only five methods associated with it, which are listed in the following table.

Name	Parameters	Returns	Description
`empty`		`bool`	Returns `true` if there are no elements in the `stack`.
`top`		reference	Returns a reference to the value of the top element.
`push`	value		Inserts the value as the top element.
`pop`			Removes the top element.
`size`		`int`	Returns the number of elements in the `stack`.

There are two ways to create a `stack`, either as an empty container or as a copy of a `vector`, `list`, or `deque`.

```
stack<float> fltStack;          // Create an empty stack of floats
stack<string> strStack(demo);   // Create string stack from list demo
```

Here is a short demonstration program that inputs some words, pushing them on the stack and printing them in reverse order as they are popped from the top.

```
//********************************************************
// This program demonstrates various stack methods
//********************************************************
#include <iostream>
#include <stack>
using namespace std;

int main ()
{
  stack<string> strStack;          // Create an empty stack
  string word;
  cout << "Enter a line with six words:" << endl;
  for (int i = 1; i <= 6; i++)
  {
    cin >> word;
    strStack.push(word);           // Insert elements at top
  }
  // Access top and size
  cout << "Top element: " << strStack.top() << endl
       << "Size of stack: " << strStack.size() << endl;
  // Print the stack
  cout << "Stack contents:" << endl;
  while (!strStack.empty())
  {
    cout << strStack.top() << endl;
    strStack.pop();                // Remove elements from top
  }
  return 0;
}
```

Here is a sample run:

```
Enter a line with six words:
four score and seven years ago
Top element: ago
Size of stack: 6
Stack contents:
ago
years
seven
and
score
four
```

The queue Template

The design philosophy behind **queue** is the same as for **stack**: Keep it simple. The most significant difference is that **push** inserts elements at one end of the data structure and **pop** removes them from the other end. In addition to the observer, **front**, for the element that is about to be popped, **queue** provides **back** to allow us to look at the element most recently pushed. Here is a summary of its operations:

Name	Parameters	Returns	Description
empty		**bool**	Returns **true** if there are no elements in the **queue**.
front		reference	Returns a reference to the value of the front element.
back		reference	Returns a reference to the value of the back element.
push	value		Inserts the value as the back element.
pop			Removes the front element.
size		**int**	Returns the number of elements in the **queue**.

As with **stack**, there are two forms of constructor for **queue**. The default creates an empty **queue**, and the other form lets us initialize the **queue** with a copy of the contents of a **vector**, **list**, or **deque**.

Here is a sample program, very similar to the one we showed for **stack**, illustrating the basic differences between the two types.

```
//*********************************************************
// This program demonstrates various queue methods
//*********************************************************
#include <iostream>
#include <queue>
using namespace std;
```

```
int main ()
{
  queue<string> strQueue;          // Create an empty queue
  string word;
  cout << "Enter a line with six words:" << endl;
  for (int i = 1; i <= 6; i++)
  {
    cin >> word;
    strQueue.push(word);           // Insert elements at back
  }
  // Access front, back, and size
  cout << "Front element: " << strQueue.front() << endl
       << "Back element: " << strQueue.back() << endl
       << "Size of queue: " << strQueue.size() << endl;
  // Print the queue
  cout << "Queue contents:" << endl;
  while (!strQueue.empty())
  {
    cout << strQueue.front() << endl;
    strQueue.pop();                // Remove elements from front
  }
  return 0;
}
```

Here is a test run with the same data that we used for our **stack** example. As you can see, the words emerge in the order that they were entered, rather than in reverse order.

```
Enter a line with six words:
four score and seven years ago
Front element: four
Back element: ago
Size of queue: 6
Queue contents:
four
score
and
seven
years
ago
```

The priority_queue Template

The **priority_queue** is a variation on the idea of a queue that also bears some resemblance to a **stack**. The major difference is that the elements in the structure are kept in sorted order. Thus **push** inserts elements using an insertion sort so that they go directly to their proper places in the ordering. At one end, **top** observes the element with the greatest value and application of **pop** removes it. An example of a priority queue is what happens in an emergency room at the hospital: Cases are arranged with the most urgent at the head of the waiting line, and then the cases are taken in order. As subsequent cases arrive, they are inserted in the waiting line according to their urgency.

Here is a summary of the `priority_queue` operations:

Name	Parameters	Returns	Description
`empty`		`bool`	Returns `true` if the priority queue contains no elements.
`top`		reference	Returns a reference to the value of the top element.
`push`	value		Inserts the value according to the ordering of the type.
`pop`			Removes the top element.
`size`		`int`	Returns the number of elements in the priority queue.

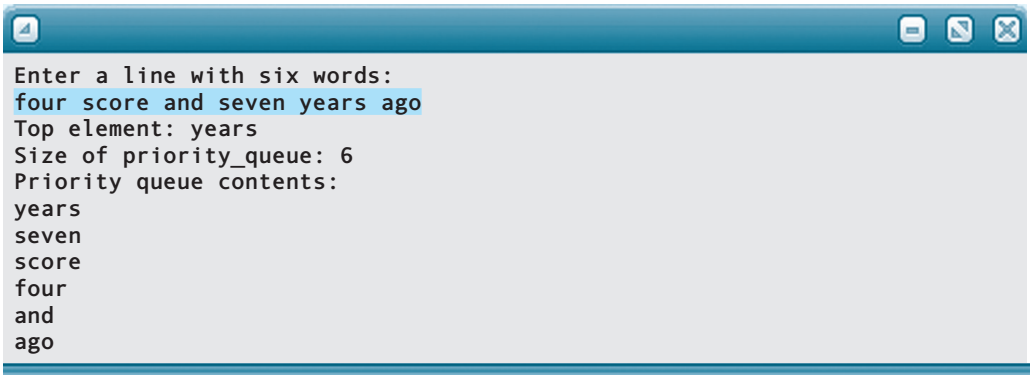
One of the quirks of the STL's organization is that `priority_queue` is not kept in its own header file. Instead, it shares the same file with `queue`; thus we include `<queue>` instead of `<priority_queue>`. This type supports the empty constructor, and more like `list` and company, it provides a second constructor that allows us to initialize the values from another container. Unlike with `stack` and `queue`, there is no straight copy constructor.

Here is a revision of our example `stack` program, adapted to the `priority_queue` so that we can appreciate the difference in this structure's behavior. Note that the queue outputs its contents in reverse alphabetical order.

```
//************************************************************
// This program demonstrates various priority_queue methods
//************************************************************
#include <iostream>
#include <queue>
using namespace std;

int main ()
{
  priority_queue<string> strQueue; // Create an empty priority_queue
  string word;
  cout << "Enter a line with six words:" << endl;
  for (int i = 1; i <= 6; i++)
  {
    cin >> word;
    strQueue.push(word);           // Insert elements in place
  }
  // Access top and size
  cout << "Top element: " << strQueue.top() << endl
       << "Size of priority_queue: " << strQueue.size() << endl;
  // Print the priority_queue
  cout << "Priority queue contents:" << endl;
  while (!strQueue.empty())
  {
    cout << strQueue.top() << endl;
    strQueue.pop();                // Remove elements from top
  }
  return 0;
}
```

Here is the output for the revised program:



The deque Template

Whereas **stack** and **queue** strive to keep things simple, the design for **deque** takes the opposite approach. With **deque**, we get a panoply of features taken from the containers we've already seen.

For example, **deque** replicates most of the operations of **list**: **back**, **begin**, **clear**, **empty**, **end**, **erase**, **front**, **insert**, **pop_back**, **pop_front**, **push_back**, **push_front**, **rbegin**, **rend**, and **size**, with essentially the same meanings. Like **vector**, it overloads **[]** and **at** to provide random access. Being linked, it does not need to support **capacity** or **reserve** operations. Nor does it directly support **merge**, **remove**, **reverse**, **sort**, or **unique**, because equivalent function templates with the same names are defined in the algorithms section of the STL, all of which can be applied to **deque**. This template provides constructors of the same four forms that **list** supplies: We can create a **deque** with zero elements, a **deque** with a specific number of elements initialized to a default value, a **deque** initialized from an array or another container, and a **deque** created with a copy constructor.

If **deque** encompasses **list** and **vector**, why shouldn't we always use it? As always, this increase in capability costs us in terms of efficiency. The underlying array implementation of a **vector** is usually much faster than that of a **deque**. We say "usually" because, if a large **vector** must change its **capacity** frequently, the linked implementation of a **deque** can outperform it. Random access in a **deque** is also not as efficient as for a **vector**. How does **deque** compare with **list**? Because a **list** doesn't have to support random access, deletions and insertions at points other than its ends can be much faster. However, when you really need the combination of **vector** and **list** features in one structure, **deque** is quite useful to have in the library.

The naming of **list** and **deque** in the STL is really somewhat misleading, because both can act as either a list or a deque in the abstract sense. The STL **list** is a general bidirectional list with support for deque operations. The STL **deque** is also a bidirectional list, with support for both deque operations and random-access indexing.

Because we've already seen the features of **deque** with other containers, an operation summary table is redundant. We will demonstrate the use of a **deque** in the Problem-Solving Case Study at the end of the chapter.

MAY WE INTRODUCE Sir Charles Antony Richard Hoare

Tony Hoare's interest in computing was awakened in the early fifties, when he studied philosophy (together with Latin and Greek) at Oxford University, under the tutelage of John Lucas. He was fascinated by the power of mathematical logic as an explanation of the apparent certainty of mathematical truth. During his National Service (1956-1958), he studied Russian in the Royal Navy. Then he took a qualification in statistics (and incidentally) a course in programming given by Leslie Fox. In 1959, as a graduate student at Moscow State University, he studied the machine translation of languages (together with probability theory) in the school of Kolmogorov. To assist in efficient look-up of words in a dictionary, he discovered the well-known sorting algorithm Quicksort.

On return to England in 1960, he worked as a programmer for Elliott Brothers, a small scientific computer manufacturer. He led a team (including his later wife Jill) in the design and delivery of the first commercial compiler for the programming language Algol 60. He attributes the success of the project to the use of Algol itself as the design language for the compiler, although the implementation used decimal machine code. Promoted to the rank of Chief Engineer, he then led a larger team on a disastrous project to implement an operating system. After managing a recovery from the failure, he moved as Chief Scientist to the computing research division, where he worked on the hardware and software architecture for future machines.

These machines were cancelled when the company merged with its rivals, and in 1968 Tony took a chance to apply for the Professorship of Computing Science at the Queen's University, Belfast. His research goal was to understand why operating systems were so much more difficult than compilers, and to see if advances in programming theory and languages could help with the problems of concurrency. In spite of civil disturbances, he built up a strong teaching and research department, and published a series of papers on the use of assertions to prove correctness of computer programs. He knew that this was long term research, unlikely to achieve industrial application within the span of his academic career.

In 1977 he moved to Oxford University, and undertook to build up the Programming Research Group, founded by Christopher Strachey. With the aid of external funding from government initiatives, industrial collaborations, and charitable donations, Oxford now teaches a range of degree courses in Computer Science, including an external Master's degree for software engineers from industry. The research of his teams at Oxford pursued an ideal that takes provable correctness as the driving force for the accurate specification, design and development of computing systems, both critical and non-critical. Well-known results of the research include the Z specification language, and the CSP concurrent programming model. A recent personal research goal has been the unification of a diverse range of theories applying to different programming languages, paradigms, and implementation technologies.

Throughout more than thirty years as an academic, Tony has maintained strong contacts with industry, through consultancy, teaching, and collaborative research projects. He took a particular interest in the sustenance of legacy code, where assertions are now playing a vital role, not for his original purpose of program proof, but rather in instrumentation of code for testing purposes. On reaching retirement age at Oxford, he welcomed an opportunity to go back to industry as a senior researcher with Microsoft Research in Cambridge. He hopes to expand the opportunities for industrial application of good academic research, and to encourage academic researchers to continue

MAY WE INTRODUCE Sir Charles Antony Richard Hoare, continued

the pursuit of deep and interesting questions in areas of long-term interest to the software industry and its customers.

The above biographical sketch was written by Sir Tony Hoare himself and reprinted with his permission. What he does not say is that he received the Turing Award in 1980, at the age of 46, for his fundamental contributions to the definition and design of programming languages and was awarded a Knighthood in 1999 for his services to education and computer science.

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

MAINTENANCE TASK: In Chapter 15, we developed a set of classes to support an appointment calendar application, and a simple driver with which to test them. We used our linked implementation of a list to hold the calendar data. Now that we have learned how to use the STL `list` class, let's see how much work it will take to convert our prior work to make use of it. In theory, this should be fairly easy.

EXISTING CODE: Recall that our appointment calendar project had six underlying classes. Three of these provide the attributes of an entry: `Name`, `TimeOfDay`, and `Date`. We originally developed a base class called `Entry`, containing just `Name` and `TimeOfDay`. We then derived a subclass from `Entry`, called `EntryWithDate`, that added support for a `Date` attribute. The `AppointmentCalendar` class used our `List` class to create a list of `EntryWithDate` objects that we could search and update. To keep things simple, we did not actually make use of the `TimeOfDay` attribute, but distinguished appointments only by their date and name.

Because our `List` class kept its data unsorted, we took the approach of performing a linear search on the list, removing the element to be updated, and then appending the new version to the end of the list.

In Chapter 16, we changed the `Date` class so that it uses the overloaded relational operators rather than the `ComparedTo` function. Let's also make this update to our appointment calendar. That step will necessitate some changes to `EntryWithDate`, but we know that its use of `ComparedTo` likewise must be revised to overload at least the < and == operators so that we can use this class with the `list` container.

DISCUSSON: Looking over the classes in our original solution, even at the abstract level, we can see that `Name`, `TimeOfDay`, and `Entry` do not depend on any of the changes we are making. We should be able to use them unchanged. Our goal should be to hide the changes from the driver, within the encapsulating interface of `AppointmentCalendar`.

We know that `EntryWithDate` must be updated to use the relational operators supplied by the revised `Date` class, and that it must itself supply overloaded relational operators for at least < and == to enable the STL `list` to work with it. For generality's sake, we can just include all of the relational operators. Once we have < and == implemented, we can write the others by cutting and pasting from them with minor changes (often called cookie-cutter coding).

Recall that we defined equality for entries as having the same date and name. How should we define "less than"? For now, let's just use the date to order entries. The implication of this choice is that within

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

a single date, the ordering of entries isn't significant. In the future, we would clearly amend this definition to take into account the time and perhaps to alphabetically order elements by name.

We now have enough information to code the new version of this class:

```
//*****************************************
// SPECIFICATION File for class EntryWithDate
//*****************************************
#include "Date.h"
#include "Entry.h"
class EntryWithDate : public Entry
{
public:
  // Constructors
  EntryWithDate();
  EntryWithDate(Date initDate, TimeOfDay initTime, Name initName);
  // Knowledge responsibility
  Date GetDate() const;
  bool operator<(const EntryWithDate& otherEntry) const;
  // Post: Returns true if instance comes before otherEntry
  bool operator>(const EntryWithDate& otherEntry) const;
  // Post: Returns true if instance comes after otherEntry
  bool operator==(const EntryWithDate& otherEntry) const;
  // Post: Returns true if instance is the same date as otherEntry
  bool operator<=(const EntryWithDate& otherEntry) const;
  // Post: Returns true if instance is <= otherEntry
  bool operator>=(const EntryWithDate& otherEntry) const;
  // Post: Returns true if instance is >= otherEntry
  bool operator!=(const EntryWithDate& otherEntry) const;
  // Post: Returns true if instance is != otherEntry
private:
  Date date;
};
```

```
//*****************************************************

// IMPLEMENTATION FILE for class EntryWithDate
//*****************************************************

#include "EntryWithDate.h"

EntryWithDate::EntryWithDate()
{ };

//*****************************************************
```

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

```
EntryWithDate::EntryWithDate
  (Date initDate, TimeOfDay initTime, Name initName) :
    Entry(initName.GetFirstName(), initName.GetMiddleName(),
          initName.GetLastName(), initTime.GetHours(),
          initTime.GetMinutes(), initTime.GetSeconds())

{
  date = initDate;
}

//*********************************************************

Date EntryWithDate::GetDate() const
{ return date; }

//*********************************************************

bool EntryWithDate::operator<(const EntryWithDate& otherEntry) const
{
  return date < otherEntry.GetDate();
}
//*********************************************************

bool EntryWithDate::operator>(const EntryWithDate& otherEntry) const
{
  return date > otherEntry.GetDate();
}

//*********************************************************

bool EntryWithDate::operator==(const EntryWithDate& otherEntry) const
{
  return (date == otherEntry.GetDate() &&
          (GetName().ComparedTo(otherEntry.GetName()) == SAME));
}

//*********************************************************

bool EntryWithDate::operator<=(const EntryWithDate& otherEntry) const
{
  return
    date < otherEntry.GetDate() || date == otherEntry.GetDate();
}
//*********************************************************

bool EntryWithDate::operator>=(const EntryWithDate& otherEntry) const
{
  return
    date > otherEntry.GetDate() || date == otherEntry.GetDate();
}
```

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

```
//*****************************************************

bool EntryWithDate::operator!=(const EntryWithDate& otherEntry) const
{
  return !(otherEntry == *this);
}
```

Next we turn our attention to the **AppointmentCalendar** class, which makes use of the **list**. Let's look first at the original version of the specification file:

```
//*****************************************************
// SPECIFICATION FILE for class AppointmentCalendar
//*****************************************************

#include "List.h"
#include <fstream>
using namespace std;

class AppointmentCalendar
{
public:
  // Constructor
  AppointmentCalendar(ifstream& inFile);
  // Knowledge responsibility
  bool AppointmentCalendar::IsThere(Name name, Date date)
  // Returns true if an entry exists with the given name and date

  // Action responsibilities
  EntryWithDate GetEntry(Name name, Date date);
  // Returns entry with time field equal to time
  // Pre:  entry with time field equal to time exists
  void InsertEntry(EntryWithDate initEntry);
  // Inserts entry into list
  void WriteListToFile(ofstream& outFile);
  // Writes list to outFile

private:
  List list;
};
```

Obviously, we need to include **<list>** rather than "**List.h**" and we will use **list<EntryWithDate>** as the type for **list**. Oh no! We gave our entry list the same name as the STL type. We should change it to **entryList**. Is there anything else to change here? If we plan to hide the changes from the driver, then the only changes should be to the private part of the interface; that means we are done with updating the specification file.

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

Here is the new version with the changes highlighted:

```
//*****************************************************
// SPECIFICATION FILE for class AppointmentCalendar
//*****************************************************

#include <list>
#include <fstream>
#include "EntryWithDate.h"
using namespace std;

class AppointmentCalendar
{
public:
  // Constructor
  AppointmentCalendar(ifstream& inFile);
  // Knowledge responsibility
  bool IsThere(Name name, Date date);
  // Returns true if an entry exists with the given name and date

  // Action responsibilities
  EntryWithDate GetEntry(Name name, Date date);
  // Returns entry with time field equal to time
  // Pre:  entry with time field equal to time exists
  void InsertEntry(EntryWithDate initEntry);
  // Inserts entry into list
  void WriteListToFile(ofstream& outFile);
  // Writes list to outFile

private:
  list<EntryWithDate> entryList;
};
```

Thus far the changes have been very straightforward. As we expected, all of the significant changes are confined to the implementation of **AppointmentCalendar**. Let's go through it method-by-method to see what we need to revise.

Here is the original version of the constructor, which reads a file and inserts it into the list. Everything in the code is concerned with building up the pieces of an entry, which hasn't changed (even though the implementation of **Date** is different, it still uses the same form of constructor).

```
AppointmentCalendar::AppointmentCalendar(ifstream& inFile)
{
  int hours, minutes, seconds;  // for class TimeOfDay
  int month, day, year;         // for class Date
  string first, middle, last;   // for class Name
  int numberEntries;
  inFile >> numberEntries;
```

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

```
  for (int counter = 0; counter < numberEntries; counter++)
  {
    inFile >> hours >> minutes >> seconds;
    TimeOfDay time(hours, minutes, seconds);
    inFile >> month >> day >> year;
    Date date(month, day, year);
    inFile >> first >> middle >> last;
    Name name(first, middle, last);
    EntryWithDate entry(date, time, name);
    list.Insert(entry);
  }
}
```

Only the very last statement makes reference to **list**. Our class used **Insert** rather than **push_back** to add elements to the end of the list. We change this one line to the following and then move on to the next method:

```
entryList.push_back(entry);
```

The **IsThere** method checks whether an entry is in the calendar merely by constructing an entry from its parameters and calling **IsThere** for the list. Now we have a problem: The STL **list** template doesn't provide an operation equivalent to a search. But wait! There are more than 60 algorithms in the other portion of the STL. Maybe we can find something there that will help us. Searching through our STL reference, we discover an algorithm called **find** that sounds promising.

The documentation for **find** says that it searches a container, starting from a beginning **iterator** up to an ending **iterator**, looking for a specified element. If it finds the element, it returns an **iterator** pointing to it (when there are multiple copies, **find** stops with the first one). If the element isn't found, the iterator points to the same place that the **end** method returns. Because we want to search the entire list, we can use **begin** and **end** to get the range iterators. To convert the result of the search into a **bool** indicating whether the entry is there, we just compare it with **end**. All we have to do is insert

```
#include <algorithm>
```

at the top of the file and change the return statement to the following:

```
return find(entryList.begin(), entryList.end(), entry)
         != entryList.end();
```

Next comes the workhorse method of this class, **GetEntry**, which searches for an entry to update and deletes it from the list.

```
EntryWithDate AppointmentCalendar::GetEntry(Name name, Date date)
{
  TimeOfDay time;
  EntryWithDate otherEntry(date, time, name);
  EntryWithDate entry;
  list.ResetList();
  entry = list.GetNextItem();
  while (entry.ComparedTo(otherEntry) != SAME)
    entry = list.GetNextItem();
  list.Delete(entry);
  return entry;
}
```

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

Our **List** class kept an iterator internally and allowed us to indirectly manage it by providing methods called **GetNextItem** and **ResetList**. The client code was responsible for looping through the list using these operations. The paradigm for the STL containers is to allow any number of iterators to be kept externally and to supply algorithms that traverse the data. We must now adapt this method to that alternate paradigm.

We already know that we can use **find** to replace the search operation. Obviously, that means we can delete the entire looping structure. But what do we do with the result from **find**? It returns a pointer to the entry, rather than the entry itself. We should declare an iterator of the appropriate type, assign the **find** result to it, and then dereference it before passing the entry back. At least the **Delete** operation has an equivalent form in **list**: We just call **erase**.

As we start coding, one other stylistic change comes to light. We named **otherEntry** because of how it would be used in the **ComparedTo** function. Given that it is the target of the **find** function, that name now seems inappropriate. Let's call it **searchEntry** instead.

Here is the revised code for **GetEntry**, with the changes highlighted.

```
EntryWithDate AppointmentCalendar::GetEntry(Name name, Date date)
{
  TimeOfDay time;
  EntryWithDate searchEntry(date, time, name);
  list<EntryWithDate>::iterator location =
    find(entryList.begin(), entryList.end(), searchEntry);
  EntryWithDate entry = *location;
  entryList.erase(location);
  return entry;
}
```

The change to the original **InsertEntry** method is trivial. We just replace the call to **Insert** with a call to **push_back**. Lastly, we review the **WriteListToFile** method.

```
void AppointmentCalendar::WriteListToFile(ofstream& outFile)
{
  EntryWithDate entry;
  Name name;
  Date date;
  TimeOfDay time;
  list.ResetList();
  outFile << list.GetLength() << endl;
  while (list.HasNext())
  {
    entry = list.GetNextItem();
    time = entry.GetTime();
    outFile << time.GetHours() << ' ' << time.GetMinutes() << ' '
            << time.GetSeconds() << ' ';
    date = entry.GetDate();
    outFile << date.GetMonth() << ' ' << date.GetDay() << ' '
            << date.GetYear() << ' ';
    name = entry.GetName();
```

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

```
    outFile << name.GetFirstName() << ' ' << name.GetMiddleName()
            << ' ' << name.GetLastName() << endl;
  }
  outFile.close();
}
```

As before, we have to account for the change in the paradigm of how we create and manage iterators in the STL versus our List ADT. However, we already know how to traverse a list from beginning to end: We simply use a For loop along with calls to **begin** and **end**. We also need to declare an iterator of the appropriate type, which we call **location**. Because **location** is a pointer to an entry, we must dereference it and assign the result to **entry**, instead of assigning the result of **GetNextItem**. Finally, our use of the **GetLength** method becomes a call to **size**.

Here, then, is the complete implementation for **AppointmentCalendar**, with all of the changes highlighted.

```
//****************************************************
// IMPLEMENTATION FILE for AppointmentCalendar
//****************************************************

#include <string>
#include <fstream>
#include <iostream>
#include <algorithm>
#include "AppointmentCalendar.h"
using namespace std;

AppointmentCalendar::AppointmentCalendar(ifstream& inFile)
{
  int hours, minutes, seconds;  // for class TimeOfDay
  int month, day, year;         // for class Date
  string first, middle, last;   // for class Name
  int numberEntries;
  inFile >> numberEntries;
  for (int counter = 0; counter < numberEntries; counter++)
  {
    inFile >> hours >> minutes >> seconds;
    TimeOfDay time(hours, minutes, seconds);
    inFile >> month >> day >> year;
    Date date(month, day, year);
    inFile >> first >> middle >> last;
    Name name(first, middle, last);
    EntryWithDate entry(date, time, name);
    entryList.push_back(entry);
  }
}

//****************************************************
```

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

```
bool AppointmentCalendar::IsThere(Name name, Date date)
{
  TimeOfDay time;
  EntryWithDate entry(date, time, name);
  return find(entryList.begin(), entryList.end(), entry)
           != entryList.end();
}

//********************************************************

EntryWithDate AppointmentCalendar::GetEntry(Name name, Date date)
{
  TimeOfDay time;
  EntryWithDate searchEntry(date, time, name);
  list<EntryWithDate>::iterator location =
    find(entryList.begin(), entryList.end(), searchEntry);
  EntryWithDate entry = *location;
  entryList.erase(location);
  return entry;
}

//********************************************************

void AppointmentCalendar::InsertEntry(EntryWithDate entry)
{ entryList.push_back(entry); }

//********************************************************

void AppointmentCalendar::WriteListToFile(ofstream& outFile)
{
  EntryWithDate entry;
  Name name;
  Date date;
  TimeOfDay time;
  outFile << entryList.size() << endl;
  list<EntryWithDate>::iterator location;
  for (location = entryList.begin();
       location != entryList.end(), location++)
  {
    entry = *location;
    time = entry.GetTime();
    outFile << time.GetHours() << ' ' << time.GetMinutes() << ' '
            << time.GetSeconds() << ' ';
    date = entry.GetDate();
    outFile << date.GetMonth() << ' ' << date.GetDay() << ' '
            << date.GetYear() << ' ';
    name = entry.GetName();
    outFile << name.GetFirstName() << ' ' << name.GetMiddleName()
            << ' ' << name.GetLastName() << endl;
  }
  outFile.close();
}
```

SOFTWARE MAINTENANCE CASE STUDY: Appointment Calendar Using STL List

TESTING: Here is the driver's output, using the same test data as before. The results match what was output by the version in Chapter 15.

Here is the input file:

```
5
8 20 0 10 12 2009 Boris Becker Brown
8 45 0 10 11 2009 Sara Jane Jones
12 30 0 10 11 2009 Susy Smiley Baker
8 30 0 10 12 2009 Bill Bradley Britton
9 30 0 11 5 2010 Mary Jane Smith
```

Here is a test run:

```
Enter name of entry to change:
Enter name as first, middle, last
Mary Jane Smith
Enter date of entry to change
Enter date as month, day, and year
11 5 2010
Entry retrieved:
Time  9:30
Date  11/5/2010
Name Mary Smith

Enter field to change: 'T' (time) 'D' (date) 'N' (name)
D
Enter date as month, day, and year
11 5 2009
Do you wish to continue changing entries? 'Y' or 'N'
Y
Enter name of entry to change:
Enter name as first, middle, last
Bill Bradley Britton
Enter date of entry to change
Enter date as month, day, and year
10 12 2009
Entry retrieved:
Time  8:30
Date  10/12/2009
Name Bill Britton

Enter field to change: 'T' (time) 'D' (date) 'N' (name)
T
Enter time as hours, minutes, seconds
9 30 0
Do you wish to continue changing entries? 'Y' or 'N'
y
```

SOFTWARE MAINTENANCE CASE STUDY: An Introduction to Software Maintenance

```
Enter name of entry to change:
Enter name as first, middle, last
Sara Jane Jones
Enter date of entry to change
Enter date as month, day, and year
10 11 2009
Entry retrieved:
Time  8:45
Date  10/11/2009
Name Sara Jones

Enter field to change: 'T' (time) 'D' (date) 'N' (name)
N
Enter name as first, middle, last
Sarah Jane Jones
Do you wish to continue changing entries? 'Y' or 'N'
y
Enter name of entry to change:
Enter name as first, middle, last
Sara Jane Jones
Enter date of entry to change
Enter date as month, day, and year
10 11 2009
No entry exists with this name and date
Do you wish to continue changing entries? 'Y' or 'N'
N
```

Here is the output file:

```
5
8 20 0 10 12 2009 Boris Becker Brown
12 30 0 10 11 2009 Susy Smiley Baker
9 30 0 11 5 2009 Mary Jane Smith
9 30 0 10 12 2009 Bill Bradley Britton
8 45 0 10 11 2009 Sarah Jane Jones
```

17.5 Nonlinear Structures

Early in the chapter, we noted that all of the structured types we had seen were linear in nature. That's still true: Vectors, lists, stacks, queues, deques, and priority queues all arrange their elements one after another, either by indexing with an integral value or by following links. In this section, we open the door to some other arrangements. If you were surprised by how many different variations could be created within the limitations of a linear arrangement, it should be clear that when we remove this restriction, the number of possibilities grows tremendously. Here we can touch on only a few nonlinear arrangements.

We start with one of the simplest, in which elements have links that branch out in different directions. This branching behavior gives the structure its name: tree. In particular, we focus on the case where each node has just two branches and, therefore, is called a binary tree.

Binary Trees

Binary tree A linked data structure, each of whose nodes contains links to a left and right child node.

We can expand the concept of a linked list to structures containing nodes with more than one forward link field. One of these structures is known as a binary tree (**FIGURE 17.11**). Such a tree is referenced by an external pointer to a specific node, called the *root* of the tree. The root has two pointers: one to its *left child* and one to its *right child*. Each child also has two pointers: one to its left child and one to its right child. The left and right children of a node are called *siblings*.

For any node in a tree, the left child of the node is the root of the *left subtree* of the node. Likewise, the right child is the root of the *right subtree*. Nodes whose left and right children are both `NULL` are called *leaf nodes*.

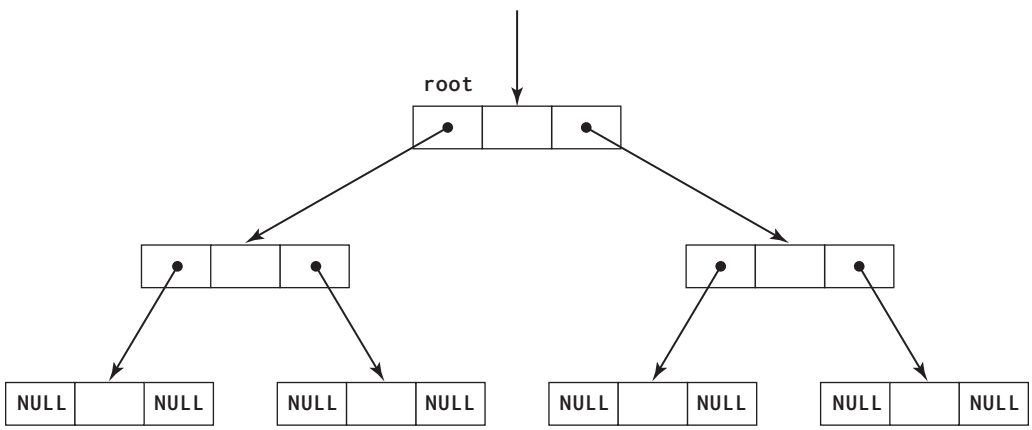
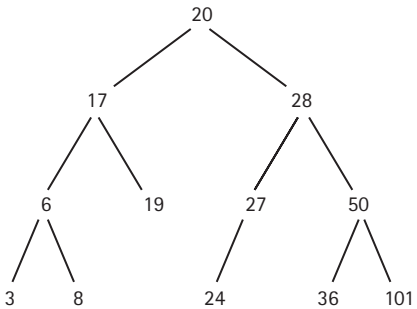


FIGURE 17.11 A Binary Tree

Although Figure 17.11 shows a binary tree with only seven nodes, there is no theoretical limit on the number of nodes in a tree. If you turn the figure upside down, you can see why it is called a tree.

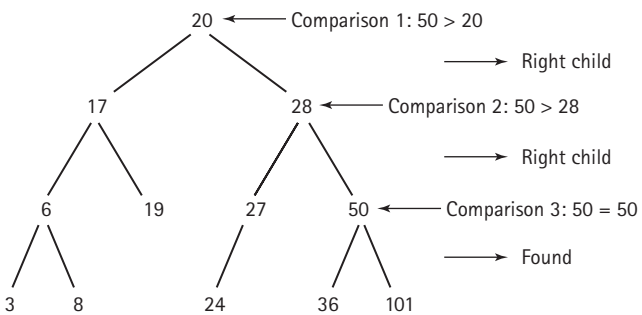
A binary search tree is a special kind of binary tree with the additional property that the values are arranged in a particular order. In a binary search tree, the component in any node is greater than the component in its left child and any of its children (left subtree) and less than the component in its right child and any of its children (right subtree). This definition assumes no duplicates. The tree shown below is an example of a binary search tree.

Binary search tree A binary tree in which the value in any node is greater than the value in its left child and any of its children and less than the value in its right child and any of its children.

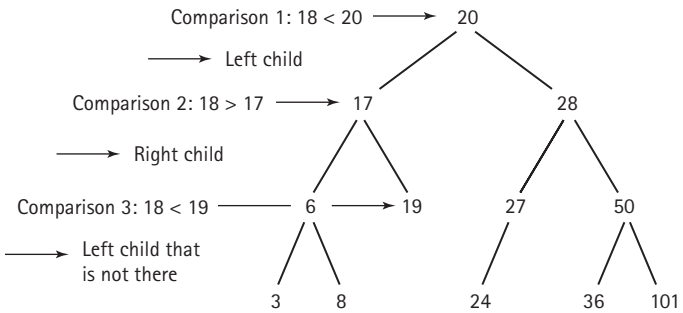


A binary search tree is useful because, if we are looking for a certain item, we can tell which half of the tree it is in by using just one comparison. We can then tell which half of that half the item is in with one more comparison. This process continues until either we find the item (a number in this case) or we determine that the item is not there. The process is analogous to a binary search of a sorted array.

Let's search for the number 50 in our binary search tree.



Now let's look for 18, a number that is not there.



The left child of 19 is `NULL`, so we know that 18 isn't in the tree. Not only do we know that 18 is not there, but we are also at the right place to insert 18 if we want to do so.

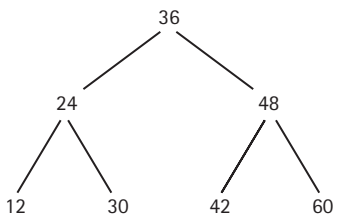
To summarize, we compared the value we were looking for with the item in the tree and took the appropriate branch if the value and the item were not the same. When we started to take a branch that was `NULL`, we knew that the value was not in the tree. Just as the binary search has algorithmic complexity of $O(\log_2 N)$, so does searching in a binary search tree.

If we want to print all of the values in a binary search tree in order, we traverse it as follows:

Tree Traversal

If the left child isn't null, traverse the left subtree.
Print the current item.
If the right child isn't null, traverse the right subtree.

We start at the root of the tree. If it has a left child, we move to that node and repeat the process. We continue in the same way down the left side of the tree until we reach a leaf, which we print. Then we move back up to the parent of that leaf and print it before traversing the subtree to its right. As an example, suppose we are given the following binary search tree:



Traversal begins with the root (36), proceeds to its left child (24) and then to the left child of 24, which is 12. Because 12 is a leaf, it is printed, and we back up to 24, which we also print. Then we traverse its right subtree, which is the leaf, 30. After printing 30, we go back to 24. Because we are done with this subtree, we back up to 36 and print it. Then the process is repeated with the right subtree, printing 42, 48, and 60.

As we see in Chapter 18, this kind of algorithm, which calls itself to process a smaller portion of the problem, is said to be *recursive*. In that chapter, we'll also see that a stack is the natural choice of structure for keeping track of where we are in the tree.

This particular form of traversal (visit left subtree, then root, then right subtree) is called an in-order traversal. There are two other commonly used traversal patterns for a tree.

In-order traversal A traversal of a binary tree that proceeds in the pattern of "left subtree, visit the root, then visit right subtrees."

Pre-order Traversal

```
Visit the root.
Visit the left subtree.
Visit the right subtree.
```

Post-order Traversal

```
Visit the left subtree.
Visit the right subtree.
Visit the root.
```

The names come from the place within the order of traversal that we visit the root. If we visit it before we visit the subtrees, then it is pre-order traversal. If we visit the root after the subtrees, then it is post-order traversal. Depending on the data stored in a binary tree, each order of traversal may have a particular use.

Trees are not limited to a binary branching factor. If we define a node to have three link fields, we get a tree that has three branches per node. Nor are trees limited to having the same number of branches in each node. For example, we can construct a tree that represents a family genealogy, where the number of links from a node depends on the number of children who were born to a given set of parents. To store a variable number of child links, we might use a list as a field within a tree node.

Pre-order traversal A traversal of a binary tree that proceeds in the pattern of "visit the root, then visit the left and right subtrees."

Post-order traversal A traversal of a binary tree that proceeds in the pattern of "visit the left and right subtrees, then visit the root."

Graph A data structure in which the links can be arranged in any pattern.

Lists, stacks, queues, and trees all have the property that the links flow in one direction (from head to tail, from root to leaf, and so on). We can form even more sophisticated structures, called graphs, in which links can flow in any direction. We might use a graph to represent the roads connecting a set of cities, or the network connecting a set of computers. As you can see, linked structures can become quite complex.

Just as we were able to implement a list as an array or a dynamic structure, so we can implement stacks, queues, trees, and graphs with dynamic structures or arrays. It is important to remember that a data structure has both a logical organization and an implementation structure. The two are distinct, and the choice of each depends on the requirements of any given problem.

The STL does not actually provide a container that implements a tree in the abstract sense. However, the `set` container that we examine in Section 17.6, "Associative Containers," is often implemented with a binary search tree.

Next we look at a different kind of logical structure that provides very fast searching under certain conditions. We'll again see that it can have multiple implementations.

Hash Tables

Your summer job at a drive-in theater includes changing the movie title on the sign each week. The letters are kept in a box in the projection booth. Your first day on the job, you discover that the letters are in random order, and you have to search through nearly all of them to find the ones you need. You sort the letters into alphabetical order, which allows you to use a binary search to locate each letter, but that's still tedious. Finally, you make a set of 26 cards to separate the letters, and then you are able to directly pull each letter from its place in the box.

Because you are searching for values within a known range, you've divided up the space so that you can directly locate them. In general, if the items you are searching have this property, you can merely index into an array based on their values and retrieve the items. Given a set of N values, the searching algorithms we've seen so far have a maximum of either N steps (linear search) or $\log_2 N$ steps (binary search). This new technique typically takes just a few steps, regardless of the amount of data. We refer to this approach as hashing.

Hashing A technique used to perform insertion and access of elements in a data structure in approximately constant time, by using the value of each element to identify its location in the structure.

Hash function A function used to manipulate the value of an element to produce an index that identifies its location in a data structure.

Collision The condition resulting when a hash function maps multiple values to the same location.

Our letter-finding example is a perfect situation for using hashing because there are a small number of possible values, and we can use them as indexes. Unfortunately, most data are not equally well behaved. For example, if our data values are real numbers representing temperatures, then the number of potential values is immense. C++ does not let us create an array big enough to use a `float` value as an index.

Nevertheless, the temperatures might have a very limited range, such as 90.0 to 110.0 degrees Fahrenheit. If we are interested in precision only to one-tenth of a degree, then there are actually just 201 distinct values. We could develop a method that takes a temperature and returns an index in the range of 0 to 200; we could then store the temperatures using hashing. Such a method is called a hash function. Assuming that the hash function takes a small amount of time to compute, searching with hashing is still a very fast operation.

The preceding examples have ignored the problem of having duplicates in the data set. Also, some data sets can have an irregular distribution of values. For example, if you are sorting a group of names, you will find that many more begin with "A," "M," or "S" than with "Q," "X," or "Z" (although those frequencies change with where you are in the world). Thus we might end up trying to store multiple values into the same place in the array, a condition called a collision.

Several approaches can be taken to deal with collisions. For example, when we attempt to store a value into a location that's already full, we could simply increment the index until we find an empty location. Unfortunately, that strategy tends to produce clusters of filled locations. In the worst case, it can take as many accesses to find an empty slot as there are elements in the array. In another approach, known as rehashing, we feed the output of the hash function back into the function (or into another function) to select a new index.

Until now we've assumed the use of a simple linear array implementation, but hashing isn't limited to this kind of structure. We could, for example, use an array of linked lists. The hash function might then index into this array and add the item to the linked list at that location. With each collision on a given array slot, the associated list would simply grow one node longer. If the hash function is well designed, the lists would remain reasonably short (few collisions), so the search would still take just a few operations. **FIGURE 17.12** shows such a structure, called a chained hash table.

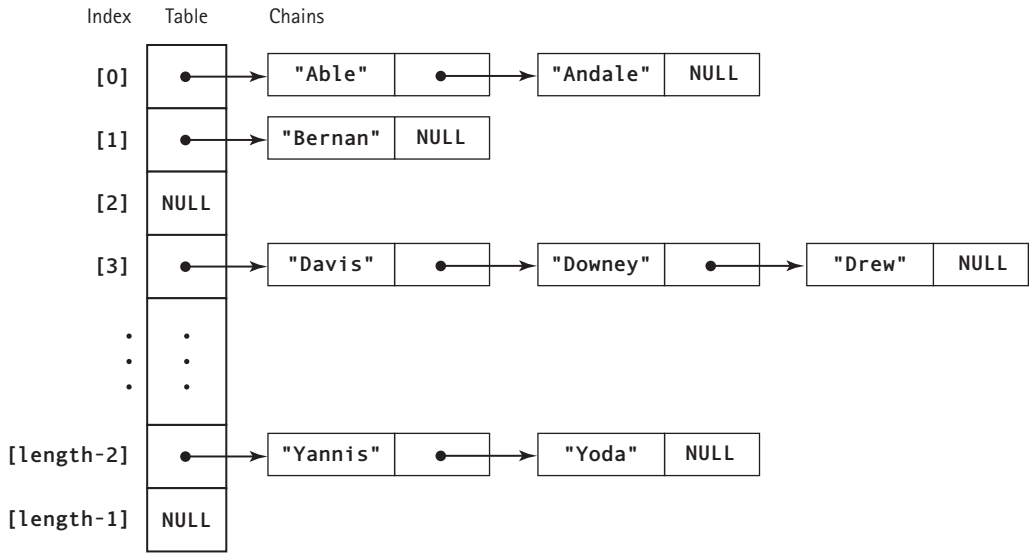


FIGURE 17.12 A Chained Hash Table

The efficiency of hashing depends on having an array or table that is large enough to hold the data set, with enough extra room to keep collisions to a minimum. Sometimes this is an impractical condition to satisfy; for many other problems, hashing is a very effective means of organizing data. For example, we could use a hash table to implement a list that has fast insertion, deletion, and access capabilities, similar to the STL `deque`. A hash table is also a candidate implementation structure for another kind of container that we look at in the next section, the `map`.

Developing a good hash function can require a considerable amount of data analysis. Our goal here is not to show you how to develop such functions, but merely to give you an appreciation for both the benefits of hashing and its limitations.

17.6 Associative Containers

The STL divides its containers into three groups. The vector, list, and deque are referred to as *sequence containers*. The simpler stack, queue, and priority queue types are called *container adapters* because they are actually built on top of the sequence containers to provide a simpler and more convenient interface. The third group includes the so-called *associative containers*.

We locate values within an associative container not by their positions, but by their values. Imagine, for example, that you have an array of structs, each of which contains a

name and a phone number. You can access a number by name by conducting a search of the array and comparing a given name to the name stored in each location; you may eventually find a match with this approach. Now, suppose instead that you could just write

```
cout << phonebook["Sylva Jones"];
```

to look up the number. That is, suppose you could find the phone number just by referring to its associated value.

Associative lookup (content-addressable access) The process of retrieving a value based on an associated value called the key.

Computer scientists refer to this kind of lookup as access by association, associative lookup, or content addressable access. The value that is used for searching is called the key, and collectively these values must be ordered in a manner that allows two keys to be compared. In many cases, the keys must also be unique. Associative lookup is the basis for many kinds of database processing as well as having a wide variety of other uses.

The STL offers five different associative containers. Here we examine just two of them: `set` and `map`.

The `set` Template

The STL `set` template takes its name from the fact that, like the mathematical set, it stores only unique values. If you try to insert a value that is already in a set, the set will be unchanged. Not surprisingly, the STL takes the implementation of the set far beyond the mathematical concept.

Internally, the values in a `set` are sorted from least to greatest, so the STL provides the ability to use an `iterator` with `begin` and `end`, or a `reverse_iterator` with `rbegin` and `rend`, to traverse the set as if it was a sorted list (most implementations use a binary search tree as shown in **FIGURE 17.13**). However, this is less efficient than traversing a sequence container.

What the `set` excels at is finding elements by their values. Thus the `find` operation is built into the `set` interface, rather than being treated as a separate algorithm. After `find` provides

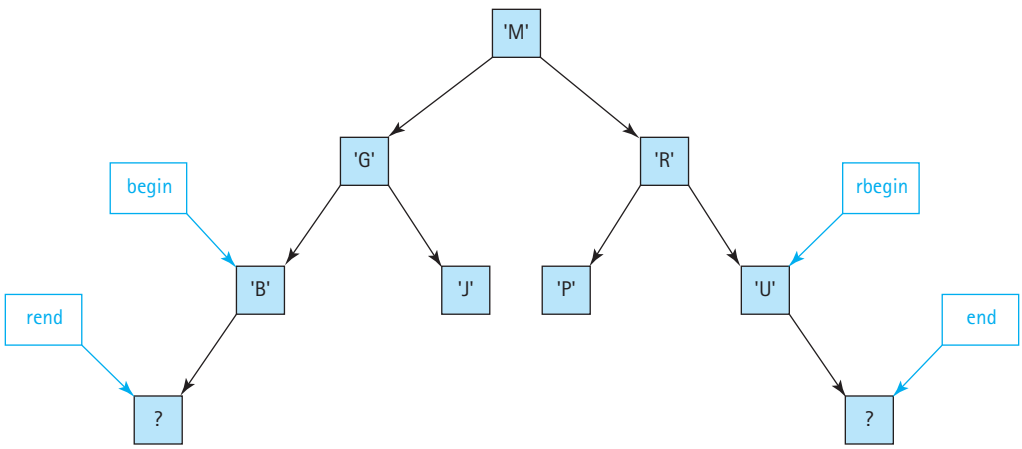


FIGURE 17.13 An Implementation of a `set` Using a Binary Search Tree

an `iterator` pointing to a value in a `set`, you can use it to `erase` the element or retrieve adjacent elements. Here is a table of some of the more commonly used `set` operations.

Name	Parameters	Returns	Description
`begin`		`iterator`	Returns an `iterator` to the first element.
`clear`			Removes all elements. Sets `size` to 0.
`count`	value	`int`	Returns the number of instances of this value in the `set`. Will be 1 if the value is in the set and 0 otherwise.
`empty`		`bool`	Returns `true` if there are no elements in the `set`.
`end`		`iterator`	Returns an `iterator` beyond the end of the `set`.
`erase`	`iterator`		Removes the element at the `iterator` position. The `size` decreases.
`find`	value	`iterator`	Returns an `iterator` to the element with the value.
`insert`	value		Inserts the value into the `set`. The `size` increases.
`lower_bound`	value	`iterator`	Returns an `iterator` to the first element that is greater than or equal to the value.
`rbegin`		`reverse_iterator`	Returns a reverse `iterator` to the last element.
`rend`		`reverse_iterator`	Returns a reverse `iterator` before the start of the `set`.
`size`		`int`	Returns the number of elements in the `set`.
`upper_bound`	value	`iterator`	Returns an `iterator` to the first element that is less than the value.

The `set` type provides a default constructor that creates an empty `set`. A second form of constructor follows the pattern of the sequence containers, in which another container or an array can be used to initialize the `set`. Of course, any duplicate elements in the container are not inserted into the `set`. A straight copy constructor that takes another `set` as a parameter is available for creating a duplicate `set`.

In addition to the class methods in the preceding table, three of the algorithms described in Section 17.4 apply to `set`: `set_difference`, `set_intersection`, and `set_union`. Keep in mind that you must include `<algorithm>` to use these algorithms, and they are called like traditional functions, rather than object methods. The example program later in this section demonstrates the use of `set_difference`. All of the `set` algorithms use the same arrangement of five parameters, all of which are iterators:

```
set_difference(start1, end1, start2, end2, output);
```

The first two iterators specify a range in the first **set**, the next two specify a range in the second **set**, and the last points to the beginning of the place where the result **set** should be placed.

Oddly enough, the **set** algorithms are not designed to output their results directly as a value of class **set**. Instead, they were written to be more generally applicable to other containers, so their output can't go directly into a **set**. (You can write the result directly to a **vector**, for example.)

Another function template in the **<iterator>** package of algorithms, **inserter**, must be used to force the **set** algorithms to generate their output in a way that enables a **set** to insert the output into itself. The **inserter** takes two arguments: the destination container, followed by an iterator that points to the place where the insertion should occur. Because **set** picks its own insertion point, it doesn't matter what the second value is, as long as it points to a place in the set. We can use **begin** or **end**, for example. Here is how a call to **set_difference** looks, once all of this has been factored in:

```
set_difference(set1.begin(), set1.end(),      // First set
               set2.begin(), set2.end(),      // Second set
               inserter(set3, set3.end()));   // Result set
```

In this example, **set3** contains all of the elements of **set1** that are not found in **set2**. We've shown this behavior as a void function call. In reality, each of the set algorithms returns an iterator pointing to the end of the result. Because we don't need this result in our example, we simply let the return value be discarded.

Here is a sample program using **set**:

```
//*********************************************************
// This program demonstrates various set methods
//*********************************************************
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

int main ()
{
  set<string> strSet;          // Create an empty set
  string word;
  cout << "Enter a line with six words:" << endl;
  for (int i = 1; i <= 6; i++)
  {
    cin >> word;
    strSet.insert(word);       // Insert elements
  }
  // Access size
  cout << "Size of set: " << strSet.size() << endl;
  // Print the set
  set<string>::iterator place;
  cout << "Set contents from beginning to end: " << endl;
  // Traverse set in forward order
```

```
  for (place = strSet.begin(); place != strSet.end(); place++)
    cout << *place << " ";
  // Create a reverse iterator
  set<string>::reverse_iterator rplace;
  cout << endl << "Set contents from end to beginning: " << endl;
  // Traverse set in reverse order
  for (rplace = strSet.rbegin(); rplace != strSet.rend(); ++rplace)
    cout << *rplace << " ";
  // Prepare to illustrate set difference
  set<string> strSet2(strSet);        // Create a copy of strSet
  strSet2.erase(strSet2.begin());     // Delete first element
  strSet2.erase(--strSet2.end());     // Delete last element
  set<string> strSet3;                // Create a set for the result
  // Need to use inserter because set doesn't allow copy iterator
  set_difference(strSet.begin(), strSet.end(),      // First set
                 strSet2.begin(), strSet2.end(),    // Second set
                 inserter(strSet3, strSet3.end())); // Result set
  // Print result set
  cout << endl << "Set difference from beginning to end: " << endl;
  for (place = strSet3.begin(); place != strSet3.end(); place++)
    cout << *place << " ";
  return 0;
}
```

Here is a sample run of the program:

```
Enter a line with six words:
four score and seven years ago
Size of set: 6
Set contents from beginning to end:
ago and four score seven years
Set contents from end to beginning:
years seven score four and ago
Set difference from beginning to end:
ago years
```

The `map` Template

An STL **`map`** type is like a **`set`**, but with one major difference: Whereas the **`set`** stores only key values, a **`map`** associates a second value with each key. The **`map`** is, therefore, the standard example of an associative container. It lets us implement the kind of name-indexed array that we described at the beginning of this section, as shown in **FIGURE 17.14**. Although the **`map`** is conceptually an array, internally it may be implemented with a linked structure such as a binary search tree or a chained hash table.

The operations for a **`map`** are nearly identical to those for a **`set`**. Its elements are sorted and it supports **`begin`**, **`clear`**, **`count`**, **`empty`**, **`end`**, **`erase`**, **`find`**, **`insert`**, **`lower_bound`**, **`rbegin`**, **`rend`**, **`size`**, and **`upper_bound`**. As you might guess, however, the **`insert`** operation does not take a simple value as its argument. Instead, it expects a value of the template type **`pair`**.

["Helmand"]	5550101
["Jeng"]	5552345
["Kellaug"]	8881077
["Lepoutre"]	9994280
["Marini"]	7714444
["McCartney"]	5550003
["Meely"]	8884902
["Melchiore"]	5557391
["Smite"]	8883129
["Teernsma"]	7713618

FIGURE 17.14 A Conceptual Illustration of a `map`

Here is an example definition of a `map` that could be used for a phone book, and an `insert` operation that places an entry into it:

```
map<string, int> phonebook;
phonebook.insert(pair<string, int>("Eben Johnson", 5550001));
```

The following statements

```
map<string, int>::iterator place;
place = phonebook.find("Eben Johnson");
cout << *place.first << ": " << *place.second << endl;
phonebook.erase(place);
```

cause `find` to return an `iterator` pointing to this `pair` element of `phonebook`. After dereferencing the iterator, we can use `first` and `second` to output the two parts of the `pair`. The last statement erases the element.

This is all well and good, but a bit tedious to code. Wouldn't it be nice if we could just use array syntax? That's the other big difference between the `set` and the `map`. With `map`, the `[]` operator is overloaded, enabling us to write the preceding code segment as follows:

```
map<string, int> phonebook;
string name = "Eben Johnson";
phonebook[name] = 5550001;
cout << name << ": " << phonebook[name] << endl;
phonebook.erase(name);
```

As we see in the following case study, a `map` can also be used to convert input strings to `enum` values.

Problem-Solving Case Study

Creating a Deck of Cards

PROBLEM: For a probability and statistics class, you are doing a project to empirically explore the odds of certain kinds of hands occurring in various card games. You want to begin by developing a set of classes that will allow you to simulate a deck of 52 cards consisting of the four suits (hearts, clubs, diamonds, spades) with 13 values (ace through king) in each suit. Once you have these working, you can easily construct the necessary drivers to try out your hypotheses.

IDENTIFYING INITIAL CLASSES: Here is the list of potential classes, based on the problem statement:

Deck
Card
Suits
Values
Hands
Games

FILTERING: With a little reflection, it's clear that the last two classes pertain to the larger problem rather than to the task immediately at hand. Clearly, we need a structure to represent the card deck as well as an individual card. Do we really need separate classes to represent suits and values, or can we just use a pair of `enum` values? You anticipate wanting to be able to enter names of cards for some of the experiments, and it would be nice to encapsulate the conversion of the `enum` values back into strings. So, yes, it makes sense to build classes representing these objects.

Here is our filtered list:

Deck
Card
Suits
Values

INITIAL CLASS DESIGNS: Let's begin by looking at the simplest classes, which represent the suits and values. They are likely to be nearly identical, with the exception of the particular enum. We can start with `Suits`, because it has a smaller range than `Values`. We would like to be able to construct a suit value from a string or from an integer. As a default, it should use the smallest value, `CLUBS`. We want observers to convert the value to an `int` or a `string`. We can call these responsibilities `toString` and `toInt`. For comparison purposes, we should also provide tests for less than and equal.

A quick walk-through of the responsibilities shows that most of them are straightforward. In `toString` we can use an array of strings, indexed by the `enum` to convert the value to a string. But how do we go in the other direction for the constructor that takes a string? As we've just seen, a `map` is a perfect choice for this task. But there's one more problem: We need to convert an input string into all uppercase to properly match it. We could use a loop to convert the string character-by-character to uppercase, but a quick brows-

Problem-Solving Case Study

ing of the STL reveals an algorithm called **transform** that can do the work for us. Here is a CRC card for this class design:

Class Name: *Suits*	Superclass: *None*	Subclasses: *None*
Responsibilities	Collaborations	
Create Suit()		
Create Suit(string)	*map, transform*	
Create Suit(int)		
toString() returns string		
toInt() returns int		
Operator < returns bool		
Operator == returns bool		

The **Values** class will be identical except for the name, so we don't show its CRC card. Here are the specification files for these two classes, based on the CRC card design. We wait until after we've finished the rest of the design before considering their implementations.

```
//*****************************************
// SPECIFICATION File for class Suits
//*****************************************
#include <string>
using namespace std;

class Suits
{
public:
  // Constructors
  Suits();
  Suits(string initString);
  Suits(int initInt);
  // Observers
  string toString() const;
  int toInt() const;
  // Relational operators
  bool operator<(Suits otherSuit) const;
  bool operator==(Suits otherSuit) const;

  enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
private:
  Suit mySuit;
};
```

Problem-Solving Case Study

```
//****************************************
// SPECIFICATION File for class Values
//****************************************
#include <string>
using namespace std;

class Values
{
public:
  // Constructors
  Values();
  Values(string initString);
  Values(int initInt);
  // Observers
  string toString() const;
  int toInt() const;
  // Relational operators
  bool operator<(Values otherValue) const;
  bool operator==(Values otherValue) const;

  enum Value {ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
              EIGHT, NINE, TEN, JACK, QUEEN, KING};
private:
  Value myValue;
};
```

Now let's move up to considering the responsibilities for a Card. We should provide a constructor that takes a value and a suit. We also want to be able to observe both of these members. Because it will be necessary to compare cards against one another, we should also support relational operators. Let's go ahead and include all of them, just for future convenience. Again a quick walk-through reveals that most of these responsibilities are trivial. However, comparisons require computing the rank of a card within the deck (its suit value × 13 plus its face value). It will simplify our work if we create a helper function to do this computation. Here is the CRC card for **Card**:

Problem-Solving Case Study

Class Name: *Suits*	Superclass: *None*	Subclasses: *None*
Responsibilities	Collaborations	
Create Card()		
Create Card(Values, Suits)	*Values, Suits*	
Get Value returns Values	*Values*	
GetSuit returns Suits	*Suits*	
Operator < returns bool		
Operator > returns bool		
Operator == returns bool		
Operator <= returns bool		
Operator >= returns bool		
Operator != returns bool		

In fact, this class is so simple that we can implement it directly:

```
//*****************************************
// SPECIFICATION File for class Card
//*****************************************
#include "Values.h"
#include "Suits.h"
class Card
{
public:
  // Constructors
  Card();
  Card(Values initValue, Suits initSuit);
  // Observers
  Values GetValue() const;
  Suits GetSuit() const;
  // Relational operators
  bool operator<(const Card& otherCard) const;
  bool operator>(const Card& otherCard) const;
  bool operator==(const Card& otherCard) const;
  bool operator<=(const Card& otherCard) const;
  bool operator>=(const Card& otherCard) const;
  bool operator!=(const Card& otherCard) const;
private:
  Values myValue;
  Suits mySuit;
  int rank() const;  // Helper function for relational ops
};
```

Problem-Solving Case Study

```
//**********************************************************
// IMPLEMENTATION FILE for class Card
//**********************************************************

#include "Card.h"

Card::Card()
{ }

//**********************************************************

Card::Card(Values initValue, Suits initSuit)
{
  myValue = initValue;
  mySuit = initSuit;
}

//**********************************************************

Values Card::GetValue() const
{ return myValue; }

//**********************************************************

Suits Card::GetSuit() const
{ return mySuit; }

//**********************************************************

bool Card::operator<(const Card& otherCard) const
{ return rank() < otherCard.rank(); }

//**********************************************************

bool Card::operator>(const Card& otherCard) const
{ return rank() > otherCard.rank(); }

//**********************************************************

bool Card::operator==(const Card& otherCard) const
{ return rank() == otherCard.rank(); }

//**********************************************************

bool Card::operator<=(const Card& otherCard) const
{ return rank() <= otherCard.rank(); }

//**********************************************************
```

Problem-Solving Case Study

```
bool Card::operator>=(const Card& otherCard) const
{ return rank() >= otherCard.rank(); }

//*******************************************************

bool Card::operator!=(const Card& otherCard) const
{ return rank() != otherCard.rank(); }

//*******************************************************

int Card::rank() const
{
  return mySuit.toInt() * 13 + myValue.toInt();
}
```

All that remains is determining the responsibilities for class **Deck**. Initially it will be sufficient to have the default constructor build a full deck of cards in order. We want the ability to shuffle the deck, to deal out an individual card, to ask whether the deck is empty, and to find out whether a given card is still in the deck. It's also helpful to know how many cards are left in the deck.

A walk-through reveals that we have a choice of STL containers with which we could collaborate to represent the deck. Let's use the **deque**, just to see it in action. We'll need to loop through the values and suits, creating the 52 cards to push into the **deque**. Once that's done, we can use the STL **find** algorithm to implement **isThere**. What about shuffling? Again, the STL algorithms come to the rescue with a **random_shuffle** operation. Dealing is just a matter of getting the front element of the **deque** and popping the card off of it. Here is a CRC card that outlines those responsibilities:

Class Name: *Deck*	Superclass: *None*	Subclasses: *None*
Responsibilities	Collaborations	
Create Deck()	*deque, Values, Suits, Card*	
Empty returns bool	*deque*	
IsThere returns bool	*find*	
Size returns int	*deque*	
void Shuffle	*Random_shuffle*	
Deal returns Card	*Card*	

Problem-Solving Case Study

Here is the specification file for class **Deck**:

```
//*****************************************
// SPECIFICATION File for class Deck
//*****************************************
#include "Card.h"
#include <deque>

class Deck
{
public:
  // Constructor
  Deck();
  // Observers
  bool empty() const;
  bool isThere(Card aCard) const;
  int size () const;
  // Action responsibilities
  void Shuffle();     // Reorders the cards
  Card Deal();        // Takes one card from deck
private:
  deque<Card> theDeck;
};
```

The implementations of most of these methods are straightforward. We can use the **empty** and **size** functions from **deque** to implement the corresponding functions in this class. We already know that **find** and **random_shuffle** from the STL algorithms can implement the **isThere** and **Shuffle** functions, respectively. To deal a card, we just use the **front** and **pop_front** operations from **deque**. The only part that requires much thought is the constructor. It's clear that we'll use **push_back** to insert the cards into the **deque**—but how do we generate the cards in order? A nested For loop, counting through the enum values, with **Values** in the inner loop and **Suits** in the outer loop, can accomplish this task. A little refresher on how to iterate through an **enum**, from Chapter 10, and we're all set to write the implementation.

```
//*****************************************************
// IMPLEMENTATION FILE for class Deck
//*****************************************************

#include "Deck.h"
#include <algorithm>

Deck::Deck()
{
  for(Suits::Suit suit = Suits::CLUBS;
      suit <= Suits::SPADES; suit=Suits::Suit(suit+1))
    for(Values::Value value = Values::ACE;
        value <= Values::KING; value=Values::Value(value+1))
      theDeck.push_back(Card(Values(value), Suits(suit)));
}
```

Problem-Solving Case Study

```
//*******************************************************

bool Deck::empty() const
{ return theDeck.empty(); }

//*******************************************************

bool Deck::isThere(Card aCard) const
{ return find(theDeck.begin(), theDeck.end(), aCard)
           != theDeck.end(); }

//*******************************************************

int Deck::size() const
{ return theDeck.size(); }

//*******************************************************

void Deck::Shuffle()
{
  random_shuffle(theDeck.begin(), theDeck.end());
}

//*******************************************************

Card Deck::Deal()
{
  Card next = theDeck.front();  // Get the next card
  theDeck.pop_front();          // Delete it from the deck
  return next;                  // Return it
}
```

Now we can return to the implementation of **Suits**. As we noted, the only method that needs further attention is the constructor that is initialized by a string. What we can do is assign the **string-enum** pairs to a **map** using statements like

```
map<string, Suit> lookup;
lookup["CLUBS"] = CLUBS;
```

To get the suit from the string, we just reverse the process:

```
mySuit = lookup[initString];
```

The one thing that stands in our way is converting the string to uppercase. The STL **transform** algorithm can handle this task for us. It takes four arguments: The first two are iterators to the start and end of the source container, the third is an iterator to the start of the destination container, and the last is a function that is applied to each element of the container as it is copied from the source to the destination. We can use the **toupper** function to perform the conversion. But can we actually apply **transform** to a string?

This is where we see that **string** really is a class in the C++ library. It is actually another container in the STL (albeit with some special properties). As a consequence, **string** also supports **begin** and **end** opera-

Problem-Solving Case Study

tions, and it can have many of the STL algorithms applied to its objects. The necessary call to **transform** looks like this:

```
transform(initString.begin(), initString.end(),
          initString.begin(), toupper);
```

At last, we can implement **Suits**:

```
//*********************************************************
// IMPLEMENTATION FILE for class Suits
//*********************************************************

#include "Suits.h"
#include <map>
#include <cctype>            // For toupper
#include <algorithm>         // For transform
#include <string>

Suits::Suits()
{ mySuit = CLUBS; }

//*********************************************************

Suits::Suits(string initString)
{
  // Create a mapping between strings and enums
  map<string, Suit> lookup;
  lookup["CLUBS"] = CLUBS;
  lookup["DIAMONDS"] = DIAMONDS;
  lookup["HEARTS"] = HEARTS;
  lookup["SPADES"] = SPADES;
  // Convert the string to uppercase
  transform(initString.begin(), initString.end(),
            initString.begin(), toupper);
  // Get the corresponding enum
  mySuit = lookup[initString];
}

//*********************************************************

Suits::Suits(int initInt)
{ mySuit = Suit(initInt); }

//*********************************************************

string Suits::toString() const
{
  string suitString[4] = {"CLUBS", "DIAMONDS", "HEARTS", "SPADES"};
  return suitString[mySuit];
}
```

Problem-Solving Case Study

```
//*********************************************************

int Suits::toInt() const
{ return int(mySuit); }

//*********************************************************

bool Suits::operator<(Suits otherSuit) const
{
  return mySuit < otherSuit.toInt();
}

//*********************************************************

bool Suits::operator==(Suits otherSuit) const
{
  return mySuit == otherSuit.toInt();
}
```

As we noted, the implementation of **`Values`** is virtually identical, except for the change in the variable names and the different **`enum`** values.

```
//*********************************************************
// IMPLEMENTATION FILE for class Values
//*********************************************************

#include "Values.h"
#include <map>
#include <cctype>            // For toupper
#include <algorithm>         // For transform
#include <string>

Values::Values()
{ myValue = ACE; }

//*********************************************************

Values::Values(string initString)
{
  // Create a mapping between strings and enums
  map<string, Value> lookup;
  lookup["ACE"] = ACE;
  lookup["TWO"] = TWO;
  lookup["THREE"] = THREE;
  lookup["FOUR"] = FOUR;
  lookup["FIVE"] = FIVE;
  lookup["SIX"] = SIX;
  lookup["SEVEN"] = SEVEN;
  lookup["EIGHT"] = EIGHT;
```

Problem-Solving Case Study

```
  lookup["NINE"] = NINE;
  lookup["TEN"] = TEN;
  lookup["JACK"] = JACK;
  lookup["QUEEN"] = QUEEN;
  lookup["KING"] = KING;
  // Convert the string to uppercase
  transform(initString.begin(), initString.end(),
            initString.begin(), toupper);
  // Get the corresponding enum
  myValue = lookup[initString];
}

//*******************************************************

Values::Values(int initInt)
{ myValue = Value(initInt); }

//*******************************************************

string Values::toString() const
{
  string valueString[13] = {"ACE", "TWO", "THREE", "FOUR",
    "FIVE", "SIX", "SEVEN", "EIGHT", "NINE", "TEN", "JACK",
    "QUEEN", "KING"};
  return valueString[myValue];
}

//*******************************************************

int Values::toInt() const
{ return int(myValue); }

//*******************************************************

bool Values::operator<(Values otherValue) const
{
  return myValue < otherValue.toInt();
}

//*******************************************************

bool Values::operator==(Values otherValue) const
{
  return myValue == otherValue.toInt();
}
```

TESTING: A proper driver for these classes should exercise them as much as possible. Here we are limited by space to some basic tests. We can create a deck and deal out some cards to see that they are initially in order. Then the deck can be shuffled before we test **`isThere`** with a random query that is input by the user. Using **`empty`**, we can control a second loop that deals out the rest of the cards. At each stage, we can test **`size`** by outputting the size of the deck. That covers all of the operations in **`Deck`**, which in turn runs through all of the values in **`Card`**, although it does not cover the relational operators (except as they are used

Problem-Solving Case Study

by `find`). Again, with the exception of the relational operators, this strategy covers all of the operations in `Values` and `Suits`.

Here is the driver that implements these tests:

```
//*********************************************************
// Driver for class Deck
//*********************************************************
#include <iostream>
#include "Deck.h"
using namespace std;

int main ()
{
  // Create a deck of 52 cards in order
  Deck myDeck;
  cout << "The deck has " << myDeck.size() << " cards."
       << endl << endl;
  // Deal out the first ten in order
  for (int i=1 ; i<=10; ++i)
  {
    Card aCard = myDeck.Deal();
    cout << aCard.GetValue().toString() << " of "
         << aCard.GetSuit().toString() << endl;
  }
  cout << endl << "The deck has " << myDeck.size()
       << " cards." << endl;
  // Shuffle the remaining cards
  cout << "Shuffling" << endl << endl;
  myDeck.Shuffle();
  // Input a card
  cout << "Enter the name of a suit: ";
  string suit;
  cin >> suit;
  cout << "Enter the name of a card (ace, two,..., king): ";
  string value;
  cin >> value;
  // Check if card is in the deck
  if (myDeck.isThere(Card(Values(value), Suits(suit))))
    cout << "The card is in the deck." << endl;
  else
    cout << "The card has already been dealt." << endl;
  // Deal out the remaining cards
  cout << "The remaining contents of the deck are:" << endl;
  while (!myDeck.empty())
  {
    Card aCard = myDeck.Deal();
    cout << aCard.GetValue().toString() << " of "
         << aCard.GetSuit().toString() << endl;
  }
  cout << endl << "The deck has " << myDeck.size()
       << " cards." << endl;
}
```

Problem-Solving Case Study

Following is a run of the driver with the user input shaded. As you can see, the cards are in order to start, and the last 42 are output in shuffled order. The user input mixes uppercase and lowercase and still matches the card. At each point the correct size is reported, and the final loop terminates when the deck is empty. After a little more testing, these classes can go into production use for your experiments.

```
The deck has 52 cards.

ACE of CLUBS
TWO of CLUBS
THREE of CLUBS
FOUR of CLUBS
FIVE of CLUBS
SIX of CLUBS
SEVEN of CLUBS
EIGHT of CLUBS
NINE of CLUBS
TEN of CLUBS

The deck has 42 cards.
Shuffling

Enter the name of a suit: Clubs
Enter the name of a card (ace, two,..., king): jack
The card is in the deck.
The remaining contents of the deck are:
QUEEN of SPADES
NINE of SPADES
TWO of SPADES
KING of DIAMONDS
EIGHT of DIAMONDS
FOUR of SPADES
SEVEN of HEARTS
NINE of HEARTS
SIX of DIAMONDS
TWO of DIAMONDS
FIVE of HEARTS
FIVE of DIAMONDS
TEN of HEARTS
KING of SPADES
QUEEN of CLUBS
JACK of HEARTS
TEN of SPADES
SIX of HEARTS
JACK of DIAMONDS
JACK of SPADES
TEN of DIAMONDS
EIGHT of HEARTS
FOUR of DIAMONDS
ACE of HEARTS
TWO of HEARTS
```

Problem-Solving Case Study

```
THREE of SPADES
SEVEN of SPADES
SIX of SPADES
THREE of DIAMONDS
SEVEN of DIAMONDS
QUEEN of DIAMONDS
EIGHT of SPADES
KING of CLUBS
FOUR of HEARTS
QUEEN of HEARTS
ACE of SPADES
KING of HEARTS
ACE of DIAMONDS
JACK of CLUBS
THREE of HEARTS
FIVE of SPADES
NINE of DIAMONDS

The deck has 0 cards.
```

Testing and Debugging

One of the advantages of working with the STL is that the classes and algorithms it provides have already been thoroughly tested. It's easy, however, to let this fact give you a false sense of security. In this chapter, we've been able to touch on only the highlights of the STL; there are many more features that we haven't seen. For example, many of the operations and constructors are overloaded by other versions or have default parameters. A mistake in writing a call may not trigger a compiler error, but instead call a version of an operation with which you're not familiar. If you will work extensively with the STL, it is important to have access to a complete reference for the STL, and to take the time to understand the sometimes cryptic notation of templatized functions and parameters.

When testing a program that uses a container, we must watch out for some special cases. Obviously, attempting to operate on an empty container and going beyond the bounds of the container are among those situations. Also watch for iterators that are moving the wrong direction or by the wrong distance on each step. Popping the first element off of a stack or queue at the start of a loop results in the equivalent of an off-by-one error, because the first element of the container is never processed. Be aware of the special properties and boundary conditions associated with each type of container you are using, and take the time to design tests that ensure your code is not inadvertently violating them.

Testing and Debugging Hints

1. When accessing a container, remember that the values returned by `end` and `rend` point outside of the structure.
2. An iterator must be dereferenced to obtain the value it points to.
3. When declaring an iterator, remember that it must be qualified by the type that it will point to—for example, `list<string>::iterator`.
4. Forward iterators are incremented to advance them and decremented to move them backward. Reverse iterators move backward when they are incremented.
5. Whenever the underlying structure of a container changes, its iterators become invalid. Accessing an invalid iterator after an insertion or deletion operation will cause errors.
6. Whenever you are deleting elements from a container, make sure that you check whether it is empty.
7. Keep in mind that not all algorithms work with all containers. Check the documentation carefully.
8. Remember to copy the top element of a stack or queue before you pop it.
9. Choose a container that has the functionality you need. Additional capability often comes at a cost in performance.
10. Many of the containers require that user-defined element types support some subset of the relational operators. Be sure that your classes provide this support.

Summary

In this chapter, we explored the realm of data structures beyond arrays and singly linked lists. Along the way we encountered vectors, bidirectional lists, stacks, queues, priority queues, deques, trees, graphs, hash tables, sets, and maps. The programming universe is full of other data structures, to the point that the options can sometimes feel overwhelming. Always keep in mind that there are two sides to every data structure. The abstract side is the interface that we use to clearly and conveniently solve a problem. The implementation side is chosen for efficiency of operation. It is usually possible to wrap an implementation within an interface in such a manner that we obtain the best of both worlds. In the end, almost any choice of data structure will be a compromise in one way or another, so the best strategy is to choose the abstract structure that best fits the problem, and then to determine whether the same or another structure provides the most effective implementation.

The stack is a structure in which we insert and delete elements at one end. It is useful when we need to recall information in the order that is the reverse of how it was inserted. We thus say that a stack is a "last in, first out" (LIFO) structure. The queue is another linear structure, but in this case elements are added at one end (the back) and removed from the other end (the front). Therefore, a queue is a "first in, first out" (FIFO) structure. Queues are often used in simulations of real-world waiting lines. A priority queue inserts information according to a given ordering, keeping the data sorted and returning it in order.

Linear structures that have links running in both directions are said to be bidirectional. We can traverse a bidirectional list from back to front as easily as we can go from front to back. A second form of bidirectional structure is the double-ended queue, or deque. With it, we can insert or remove elements at either end.

A binary tree is a branching data structure that has a root node with left and right subtrees. Each of the subtrees has a pair of links to its own subtrees, and so on. Nodes that have two `NULL` links are called leaves. A node with non-`NULL` links is said to be a parent, and the nodes directly below it are called its children. By arranging the values in a binary tree so that

every left child's value is less than its parent's value, and every right child's value is greater than its parent's value, we create a binary search tree. Finding a value in a binary search tree containing N values takes $\log_2(N)$ steps. An in-order traversal of the tree will visit all of the elements in order; a binary tree may also be traversed in pre-order or post-order. Additional links enable us to build arbitrarily complex networks of nodes, called graphs.

Hashing is a technique that computes an index for storing and retrieving values in a structure in roughly constant time. The keys to effective hashing are developing a good hash function and having enough extra space in the structure to minimize collisions. A chained hash table uses an array of linked lists, where a collision results in a node being added to the list associated with the array element where the collision occurred.

Binary search trees and hash tables are often used to implement associative lookup, in which an arbitrarily complex key value may be used like an index to access associated data. The set is a structure that stores unique values associatively, enabling us to determine whether a given value is present in the set. A map extends the idea of a set by pairing another value with the key so that we can retrieve it once the key has been located.

The C++ Standard Template Library provides templatized classes and functions that represent many common data structures. Once you understand the principles behind the STL, working with data structures becomes much easier. However, it is important to be aware of the underlying properties of the STL classes so that you can choose the ones that are most appropriate for a given problem.

Quick Check

1. Why would you choose one data structure in the problem-solving phase and then implement it with a different structure? (pp. 870–872)
2. What are the two basic operations on a stack? (pp. 873–875)
3. Is a queue a LIFO structure or a FIFO structure? (p. 875)
4. Why do we call a map an associative structure? (p. 917)
5. In what order does post-order traversal visit the nodes of a binary tree? (pp. 910–911)
6. What is a collision in a hash table? (pp. 912–913)
7. How would you declare an STL `queue` called `nums`, with a `float` specialization? (pp. 892–893)
8. Which `list` operation returns an iterator that points just beyond the last element? (pp. 888–890)
9. Which STL algorithm would you use to search a `deque` for a value? (pp. 902, 904)
10. The STL `stack` and `queue` containers both have `pop` operations. How do they differ? (pp. 890–892)
11. How would you insert the number 42 into a `map` called `matches`, at a location with the key value "`answer`"? (pp. 917–918)

Answers

1. In problem solving, we want a structure that naturally fits the problem. In implementation, efficiency is also a concern. 2. `push` and `pop` 3. FIFO. 4. Because it associates key–value pairs and allows us to look up values using their associated keys. 5. Left subtree, right subtree, root. 6. The condition in which the hash function maps two values to the same place in the structure. 7. `queue<float> nums;` 8. `end()` 9. `find` 10. The `stack pop` removes the element at the back; the `queue pop` removes the element at the front. 11. `matches["answer"] = 42;`

Exam Preparation Exercises

1. Given the following operations on a stack:

```
push(5), push(4), push(3), pop(), push(2), push(1), pop(), pop()
```

   a. How many elements are on the stack?
   b. What is the value at the top of the stack?

2. Given the following operations on a queue:

```
push(5), push(4), push(3), pop(), push(2), push(1), pop(), pop()
```

   a. How many elements are in the queue?
   b. What is the value at the front of the queue?

3. Given the following operations on a priority queue:

```
push(1), push(7), push(3), push(2), push(5), pop(), push(10), pop()
```

   a. How many elements are in the priority queue?
   b. What is the value at the front of the priority queue?
   c. What was the last value popped from the priority queue?
   d. Where would the value 4 be inserted in the priority queue?

4. Given the following operations on a deque:

```
push_back(5), push_back(4), push_front(3), push_front(2), pop_back(),
push_back(1), pop_front(), pop_front()
```

   a. How many elements are in the deque?
   b. What is the value at the front of the deque?
   c. What is the value at the back of the deque?
   d. Write the current contents of the deque, indicating `front` and `back`.

5. a. Write an algorithm to perform in-order traversal of a binary tree.
   b. Write an algorithm to perform pre-order traversal of a binary tree.
   c. Write an algorithm to perform post-order traversal of a binary tree.

6. a. What is the relationship of the values in the parent node and its two children in a binary search tree?
   b. If a binary search tree contains 63 nodes, what is the maximum time required to search for a matching value among its nodes?
   c. How many leaves are there in a binary search tree with 7 nodes?

7. a. What problem can result from resolving a collision in a hash table by incrementing the index and using the next location in the table?
   b. What do we mean by the "chain" in a chained hash table?

8. a. Is an `iterator` the same as a pointer? Explain.
   b. What happens to an `iterator` after a deletion from a container?
   c. Which operator do you use to advance an `iterator`?
   d. Which operator causes an `iterator` to move backward?
   e. In which direction does a `reverse_iterator` move when it is incremented?

9. a. Which aspect of `vector` does `deque` implement?
   b. Which aspect of `list` does `deque` implement?

c. What happens to a `deque` when the `clear` operation is called?
d. What happens to a `deque` when the `erase` operation is called?

10. a. What are the five operations provided by `stack`?
    b. What are the six operations provided by `queue`?
11. a. What is the difference between the `size` and `capacity` of a `vector`?
    b. How does the `vector at` operator differ from the `[]` operator?
    c. When would you use the `reserve` operation with a `vector`?
12. a. For `map` and `set`, what is returned by the `upper_bound` operation?
    b. Which primary capability does a `map` add to those of the `set`?
    c. Name three set-theoretic operations from the STL algorithms library.
    d. You want a data structure that keeps track of whether shareholders are present at a corporate meeting. Would you choose a `map` or a `set`? Explain.

Programming Warm-Up Exercises

1. Given the following `vector` initialization:

```
int fill[] = {1,2,3,4,5,6,7,8,9,10};
vector<int> ex1(fill, fill+sizeof(fill)/sizeof(int)));
```

What is output by each of the following code segments?

a.
```
cout << ex1.size() << " " << ex1.front() << " " << ex1.back();
```

b.
```
cout << ex1[3] << " " << ex1[5] << " " << ex1[9];
```

c.
```
ex1.erase(ex1.begin()+4);
  cout << ex1[3] << " " << ex1[4] << " " << ex1[5];
```

d.
```
vector<int>::iterator i;

  ex1.erase(ex1.rbegin());
  for (i = ex1.begin()+2; i != ex1.end(); i++)
    cout << *i << endl;
```

2. Given the following `list` initialization:

```
int fill[] = {12,15,7,42,43,9,11,52,30,10};
list<int> ex2(fill, fill+sizeof(fill)/sizeof(int)));
```

What is output by each of the following code segments?

a.
```
cout << ex2.size() << " " << ex2.front() << " " << ex2.back();
```

b.
```
cout << *ex2.begin() << " " << *(--ex2.end()) << " "
      << *ex2.rbegin() << " " << *(--ex2.rend());
```

c.
```
ex2.insert(++ex2.begin(), 99);
ex2.pop_back();
cout << *ex2.begin();
```

d.
```
list<int>::iterator i = ex2.begin();
while (i != ex2.end())
  if (*i%2 == 0)
    i = ex2.erase(i);
  else
    i++;
for (i = ex2.begin(); i != ex2.end(); i++)
  cout << *i << endl;
```

e.
```
list<int>::iterator i;
ex2.sort();
for (i = ex2.begin()+2; i != ex2.end(); i++)
  cout << *i << endl;
```

3. Given the following `deque` initialization:

```
int fill[] = {27,16,81,42,22,75,3,49,12,3,7,2,4};
deque<int> ex3(fill, fill+sizeof(fill)/sizeof(int)));
```

What is output by each of the following code segments?

a.
```
cout << ex3.size() << " " << ex3.front() << " " << ex3.back();
```

b.
```
cout << *ex3.begin() << " " << *(--ex3.end()) << " "
        << *ex3.rbegin() << " " << *(--ex3.rend());
```

c.
```
cout << ex3[2] << " " << ex3[6] << " " << ex3[8];
```

d.
```
ex1.erase(ex1.begin()+4);
  cout << ex3[3] << " " << ex3[4] << " " << ex3[5];
```

e.
```
ex2.insert(++ex3.begin(), 99);
ex3.pop_back();
cout << *ex3.begin();
```

f.
```
list<int>::iterator i = ex3.begin();
while (i != ex3.end())
  if (*i%2 == 1)
    i = ex3.erase(i);
  else
    i++;
for (i = ex3.begin(); i != ex3.end(); i++)
  cout << *i << endl;
```

4. Given the following `list` initialization:

```
int fill[] = {1,2,3,4,5};
list<int> ex4(fill, fill+sizeof(fill)/sizeof(int)));
```

What is wrong with the following statement?

```
cout << *ex4.rend() << endl;
```

5. Given the following `stack` operations:

```
stack<char> ex5;
ex5.push('r'); ex5.push('a'); ex5.push('h'); ex5.push('c');
while(!ex5.empty())
  { cout << ex5.top(); ex5.pop(); }
```

What is output?

6. Given the following `queue` operations:

```
queue<string> ex6;
ex6.push("Co"); ex6.push("nt"); ex6.push("ai"); ex6.push("ner");
while(!ex6.empty())
  { cout << ex6.top(); ex6.pop(); }
```

What is output?

7. Given the following `priority_queue` operations:

```
priority_queue<float> ex7;
ex7.push(1.8); ex7.push(5.2); ex7.push(3.4); ex7.push(6.7);
while(!ex7.empty())
  { cout << ex7.top() << " "; ex7.pop(); }
```

What is output?

8. a. Write the declaration for a `vector` of `int` values, initialized to the numbers 1 through 10. You will need to define an array as part of the initialization process.
   b. Write the declaration for an empty `deque` of `int` values.
   c. Write statements that twice copy the contents of the `vector` declared in part (a) into the `deque` declared in part (b). The first time, the elements should be pushed in order at the back of the `deque`; the second time, they should be pushed at the front so that they are in reverse order within the `deque`. Afterward, the values of `front` and `back`, as applied to the `deque`, should both be 10, and the `size` of the `deque` should be 20.
   d. Assuming the operations in part (c) have been performed, write statements to output the `size` of the `deque`, its first and last elements, and the values in locations 9 and 10.
   e. Assuming the operations in part (c) have been performed, write a loop that pops an element from both ends of the `deque` on each iteration and outputs these values in two columns. The loop terminates when the `deque` is empty.
9. a. Write the declarations for three `set` containers that can hold `int` values. Initialize the first `set` to hold the odd numbers in the range 1 through 9. Initialize the second `set` to hold the even numbers from 2 to 10.
   b. Given the declarations in part (a), write statements to store their difference in the third `set`, and then output the contents of the third `set`. Does it matter whether you pass the first `set` or the second `set` as the first argument to the difference algorithm? That is, would the output differ if their places were reversed in the call?
   c. Given the declarations in part (a), write statements to store the intersection of the first two `set`s in the third `set` and then output its contents.
   d. Given the declarations in part (a), write statements to store the union of the first two `set`s in the third `set` and then output its contents.
10. a. Write the declaration for a `map` that associates values of type `long int` with values of type `string`. The `long int` will be the key (index), and the `string` will be its associated data. You can imagine this data structure as holding Social Security numbers (SSNs) and names.
    b. Write a simple driver that inputs ten SSNs, with corresponding names, and stores them in the `map` declared in part (a). It should then allow the user to input an SSN, after which the matching name is output. Be sure that you handle cases of SSNs that aren't in the `map`.
    c. Given the use of the `map` in part (b), explain why it is a better choice of data structure than an array indexed by the SSN.
11. a. Which STL algorithm would you use to search through a `list`?
    b. If you want to convert a `string` to uppercase, which STL algorithm makes that task easier?

c. You want to rearrange the elements of `list` into random order. Which STL algorithm can take care of this task for you?

d. You want to search through a `deque` of `int` values, replacing every occurrence of 21 with 18. Which STL algorithm would be a good choice?

Programming Problems

1. You are registering people for a campus club that charges dues. New members receive a discount, and returning members pay the full fee. You have a file that contains the ID numbers (which are integers) of existing members, and a file containing the ID numbers for people who have signed up for this year. Write an application that determines which of the people who have signed up for this year are new members and which are returning members. The application should combine the two files so that there are no duplicates and write the result out as a new file of existing members. It should also write out three additional files: the first contains the ID numbers of past members who did not return, the second contains the ID numbers of returning members, and the third contains the ID numbers of new members.
2. A local movie theater has three ticket windows and two computerized ticket kiosks. Some transactions, such as group discounts, can only be done at the ticket windows. An arriving customer looks at the lines, and chooses to stand in the shortest one that can handle his or her transaction. Group sales take four minutes to process. Normal ticket sales take two minutes at a window and three minutes at a kiosk. Internet ticket sales pickups take one minute at a kiosk and two minutes at a window. Write a simulation of this process, using queues to represent the five lines. The data set should consist of randomly arranged customer arrivals, with 5% group sales, 20% Internet ticket sales pickups, and 75% regular ticket sales. A data set will contain 200 customers. All of the customers tend to arrive shortly before the start of a movie, so we are simplifying the simulation to say that they all arrive at the same time. From this simulation, the theater wants you to determine the maximum length of each line, the average waiting time for each line, and the time required for each line to empty.

   As a hint, think about using a `list` to create the data set by adding 10 group, 40 Internet, and 150 regular customers and then shuffling the list. The list is then traversed, transferring each customer to the shortest queue that can handle the customer's transaction.
3. Implement the shopping cart simulation described in Section 17.2 so that it runs for a specified number of minutes. A counter can be used to keep track of the current minute in the simulation. Each time the counter advances, the simulation code handles all of the actions to be performed during that minute. You can use a stack to represent the column of waiting carts, and a priority queue to represent the active carts.

   Each `Cart` object should keep track of the total time it has been active. Use the random-number generator to determine how many minutes (1 to 60) that a cart will be active when it is taken out, and how many customers take carts out in each minute. Keep in mind that the priority queue pops the element with the greatest value, so you will want the `Cart` object's < operator to indicate that carts with a later return time have lower values. One way to do so is to represent the time as a negative number. For example, a simulation of 1000 minutes would start at −1 and end at −1000. When a cart is taken out, its return time becomes the current minute minus the time it will be

active. That way, the cart at the head of the priority queue will have the return time that is earliest in the simulation.

At the start of every minute, check the front of the priority queue to see if the front cart's return time is the same as the current minute, and move it back to the stack. Be sure to check for more than one cart being returned in a given minute. Then determine the number of carts to be taken out during that minute and their activity times, and push them into the priority queue. Remember to watch out for empty container situations. At the end of the simulation, any carts that remain active should be transferred back to the stack. Then, for each cart, output its ID number and total activity time.

4. An application is needed to keep track of shareholders at an annual corporate meeting. The application takes as input a file containing the shareholder names. When a shareholder arrives, his or her name is checked against the list for permission to enter the meeting. That name is then moved to a list of those present, which management can request to see at any time. If the person leaves the meeting, his or her name should be moved from the present list back to the shareholder list. We've described these data structures as lists, but it should be apparent that a `set` is also a natural way of representing the data, given that the only operations are to search the `set` and move names in and out. You can also use the `Name` class that we developed for this book, but the `ComparedTo` function should be replaced with overloaded relational operators.
5. In Chapter 7, Programming Problem 2 involved an application to compute a person's weight on a different planet. Reimplement the application using a `map`, so that the planet's name (a `string`) can be used to retrieve the weight multiplication factor (a `float`).
6. Use the `Deck` class developed in this chapter as part of implementing a program that allows the user to play a game of solitaire. A `list` is a natural way to represent the columns of cards, because it provides an operator called `splice` that lets us move ranges of elements from one `list` to another. Here's an example call:

```
column5.splice(column5.end(), column3, column3.place, column3.end());
```

The result of the call is that the elements of the `column3 list`, starting from the one pointed to by the iterator `place` and going to its end, are removed from `column3` and inserted at the end of `column5`. The `Deck` class will provide a convenient way to deal out the initial arrangement, and the `Card` class can be used to create the `list` elements.

Case Study Follow-Up

1. Expand the test driver for the `Deck` class so that it exercises the relational operators supported by `Deck`, `Suits`, and `Values`.
2. Implement the remaining relational operators for the `Suits` and `Values` classes.
3. Some card games involve shuffling multiple decks. Add a parameterized constructor to the `Deck` class that lets the caller specify how many standard 52-card decks should be in the working deck.
4. Add support for `begin` and `end` operations that return iterators to the ends of the deck. Also implement overloading of the `[]` operator to enable access to deck elements. You can do so by simply calling the corresponding operators in the underlying `deque`.
5. After completing Exercise 4, implement a driver that lets you issue various commands to the deck: deal a given number of cards, reshuffle the deck, and view the current contents of the deck.

6. Write a simulation that empirically determines the odds of being dealt four of a kind in a 5-card hand. As input, it should take the number of hands to try. Note that if more than 10 hands are requested, the program will have to create a new deck after 50 cards have been dealt. You may have to simulate thousands of hands to get significant results. For each hand, check whether four cards with the same value are present, and increment a count. At the end, divide the total number of hands by this count to get the odds. For example, if the result is 35, you can report the odds as 1 in 35 hands.