Chapter 4
Using Abstract Data Types

Nothing remained in whose reality she could believe, save those abstract ideas.

— Virginia Woolf, *Night and Day,* 1919

Objectives

- To appreciate the advantages of abstract data types as a software development strategy.
- To develop significant facility with the **`Vector`**, **`Grid`**, **`Stack`**, **`Queue`**, **`Map`**, **`Lexicon`**, and **`Scanner`** classes from the client perspective.
- To understand the angle-bracket syntax required to use parameterized types in C++.
- To be able to explain the explain the advantages of the **`Vector`** and **`Grid`** classes over one- and two-dimensional arrays with regard to flexibility and safety.
- To understand the difference between the **`Stack`** and **`Queue`** classes in terms of the ordering of their elements.
- To gain some experience with the strategy of discrete-time simulation.
- To understand the **`Map`** class as a mapping from keys to values and to develop some intuition about when to use it.
- To know how to use the **`Lexicon`** class as an easily searchable list of words.
- To understand how to use the **`Scanner`** class to extract tokens from a string or an input file.
- To appreciate the role of iterators in working with abstract classes.

As you know from your programming experience, data structures can be assembled to form hierarchies. The atomic data types—such as **`int`**, **`char`**, **`double`**, and enumerated types—occupy the lowest level in the hierarchy. To represent more complex information, you combine the atomic types to form larger structures. These larger structures can then be assembled into even larger ones in an open-ended process. Collectively, these assemblages of information into more complex types are called **data structures.**

As you learn more about programming, however, you will discover that particular data structures are so useful that they are worth studying in their own right. Moreover, it is usually far more important to know how those structures behave than it is to understand their underlying representation. For example, even though a string might be represented inside the machine as an array of characters, it also has an abstract behavior that transcends its representation. A type defined in terms of its behavior rather than its representation is called an **abstract data type,** which is often abbreviated to **ADT.** Abstract data types are central to the object-oriented style of programming, which encourages thinking about data structures in a holistic way.

In this chapter, you will have a chance to learn about seven classes—**`Vector`**, **`Grid`**, **`Stack`**, **`Queue`**, **`Map`**, **`Lexicon`**, and **`Scanner`**—each of which represents an important abstract data type. For the moment, you will not need to understand how to implement those classes. In subsequent chapters, you'll have a chance to explore how each of these classes can be implementated and to learn about the algorithms and data structures necessary to make those implementations efficient.

Being able to separate the behavior of a class from its underlying implementation is a fundamental precept of object-oriented programming. As a design strategy, it offers the following advantages:

- *Simplicity*. Hiding the internal representation from the client means that there are fewer details for the client to understand.
- *Flexibility*. Because a class is defined in terms of its public behavior, the programmer who implements one is free to change its underlying private representation. As with any abstraction, it is appropriate to change the implementation as long as the interface remains the same.
- *Security*. The interface boundary acts as a wall that protects the implementation and the client from each other. If a client program has access to the representation, it can change the values in the underlying data structure in unexpected ways. Making the data private in a class prevents the client from making such changes.

The ADT classes used in this book are inspired by and draw much of their structure from a more advanced set of classes available for C++ called the **Standard Template Library,** or **STL** for short. Although the STL is extremely powerful and provides some capabilities beyond the somewhat simplified class library covered in this book, it is also more difficult to understand from both the client and implementation perspectives. One of the primary advantages of using the simplified class library is that you can easily understand the entire implementation by the time you finish this book. Understanding the implementation gives you greater insight into how classes work in general and what libraries like the Standard Template Library are doing for you behind the scenes. Experience has shown, however, that you will be able to understand the implementation of a class more easily if you have first had a chance to use with that class as a client.

4.1 The `Vector` class

One of the most valuable classes in the Standard Template Library and the simplified version of it used in this book is the **`Vector`** class, which represents a generalization of the array concept presented in section 2.4. To use the **`Vector`** class, you must include its interface, just as you would for any of the libraries in Chapter 3. The interfaces for each of the ADT classes introduced in this chapter is simply the name of the class spelled with a lowercase initial letter and followed with the extension **`.h`** at the end. Every program that wants to use the **`Vector`** class must therefore include the line

```
#include "vector.h"
```

Arrays are a fundamental type in almost all programming languages and have been part of programming language designs since the beginnings of the field. Arrays, however, have a number of weaknesses that can make using them difficult, such as the following:

- Arrays are allocated with a particular size that doesn't change after the array is allocated.
- Even though arrays have a fixed size, C++ does not in fact make that size available to the programmer. In most applications, you need to keep track of the effective size of the array, as discussed in Chapter 2.
- It is impossible to insert new elements into an array or to delete elements without writing a fair amount of code to shift the existing elements to new index positions.
- Many languages, including both C and C++, make no effort to ensure that the elements you select are actually present in the array. For example, if you create an array with 25 elements and then try to select the value at index position 50, C++ will not ordinarily detect this as an error. Instead, the program will blithely go on and look at the memory addresses at which element 50 would appear if the array were long enough. It would be far better if arrays in C++ (as they do in Java) implemented **bounds checking,** which means that every array access checks to see whether the index is valid.

The **`Vector`** class solves each of these problems by reimplementing the array concept in the form of an abstract data type. You can use the **`Vector`** class in place of arrays in any application, usually with surprisingly few changes in the source code and only a minor reduction in efficiency. In fact, once you have the **`Vector`** class, it's unlikely that you will have much occasion to use arrays at all, unless you actually have to *implement* classes like **`Vector`**, which, not surprisingly, uses arrays in its underlying structure. As a client of the **`Vector`** class, however, you are not interested in that underlying structure and can leave the array mechanics to the programmers who implement the abstract data type.

As a client of the **`Vector`** class, you are concerned with a different set of issues and need to answer the following questions:

1. How is it possible to specify the type of object contained in a **`Vector`**?
2. How does one create an object that is an instance of the **`Vector`** class?
3. What methods are available in the **`Vector`** class to implement its abstract behavior?

The next three sections explore the answers to each of these questions in turn.

Specifying the base type of a `Vector`

Most of the classes covered in this chapter contain other objects as part of a unified collection. Such classes are called either **container classes** or **collection classes.** In C++, container classes must specify the type of object they contain by including the type name in angle brackets following the class name. For example, the class `Vector<int>` represents a vector whose elements are integers, `Vector<char>` specifies a vector whose elements are single characters, while `Vector<string>` specifies one in which the elements are strings. The type enclosed within the angle brackets is called the **base type** for the collection and is analogous to the element type of a conventional array.

Classes that include a base-type specification are called **parameterized classes** in the object-oriented community. In C++, parameterized classes are more often called **templates,** which reflects the fact that C++ compilers treat `Vector<int>`, `Vector<char>`, and `Vector<string>` as independent classes that share a common structure. The name `Vector` acts as a template for stamping out a whole family of classes, in which the only difference is what type of value can appear as an element of the vector. For now, all you need to understand is how to *use* templates; the process of implementing basic templates is described in Chapter 9.

Declaring a new `Vector` object

One of the philosophical principles behind abstract data types is that clients should be able to think of them as if they were built-in primitive types. Thus, just as you would declare an integer variable by writing a declaration such as

```
int n;
```

it ought to be possible to declare a new vector by writing

```
Vector<int> vec;
```

In C++, that is precisely what you do. That declaration introduces a new variable named `vec`, which is—as the template marker in angle brackets indicates—a vector of integers.

As it happens, however, there is more going on in that declaration than meets the eye. Unlike declarations of a primitive type, declarations of a new class instance automatically initialize the object by invoking a special method called a **constructor.** The constructor for the `Vector` class initializes the underlying data structures so that they represent a vector with no elements, which is called an **empty vector,** to which you can later add any elements you need. As a client, however, you have no idea what those underlying data structures are. From your point of view, the constructor simply creates the `Vector` object and leave it ready for you to use.

Operations on the `Vector` class

Every abstract data type includes a suite of methods that define its behavior The methods exported by the `Vector` class appears in Table 4-1. As you can see, the `Vector` class includes methods that correspond directly to standard array operations (selecting an individual element and determining the length) along with new methods that extend the traditional array behavior (adding, inserting, and removing, elements).

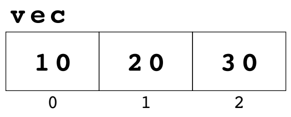
Given that every `Vector` you create starts out with no elements, one of the first things you need to learn is how to add new elements to a `Vector` object. The usual approach is to invoke the `add` method, which adds a new element at the end of the `Vector`. For example, if `vec` is an empty array of integers as declared in the preceding section, executing the code

Table 4-1 Methods exported by the `Vector` class

`size()`	Returns the number of elements in the vector.
`isEmpty()`	Returns **`true`** if the vector is empty.
`getAt(`*index***`)`**	Returns the element of the vector that appears at the specified index position. As a convenience, the **`Vector`** class also makes it possible to select an element using array notation, so that **`vec[i]`** selects the element of **`vec`** at index position **`i`**.
`setAt(`*index***`,`** *value***`)`**	Sets the element at the specified index to the new value. Attempting to reset an element outside the bounds of the vector generates an error.
`add(`*value***`)`**	Adds a new element at the end of the vector.
`insertAt(`*index***`,`** *value***`)`**	Inserts the new value before the specified index position.
`removeAt(`*index***`)`**	Deletes the element at the specified index position.
`clear()`	Removes all elements from the vector.
`iterator()`	Returns an *iterator* that cycles through the elements of the vector in turn. Iterators are discussed in section 4.8.

```
vec.add(10);
vec.add(20);
vec.add(30);
```

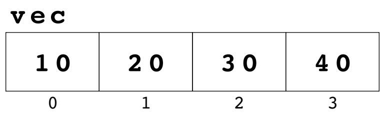
would create a three-element vector containing the values 10, 20, and 30. Conceptually, you could diagram the resulting structure just as if it were an array:



The major difference between the vector and the array is that you can add additional elements to the vector. For example, if you subsequently called

```
vec.add(40);
```

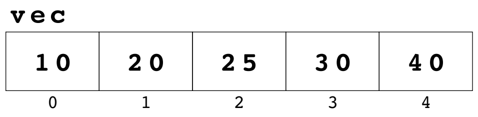
the vector would expand to make room for the new element, like this:



The **`insertAt`** method allows you to add new elements in the middle of a vector. The first argument to **`insertAt`** is an index number, and the new element is inserted before that position. For example, if you call

```
vec.insertAt(2, 25);
```

the value 25 is inserted before index position 2, as follows:

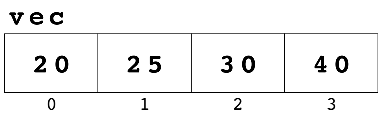


Internally, the implementation of the **`Vector`** class has to take care of moving the values 30 and 40 over one position to make room for the 25. From your perspective as a client, all of that is handled magically by the class.

The **`Vector`** class also lets you remove elements. For example, calling

```
vec.removeAt(0);
```

removes the element from position 0, leaving the following values:



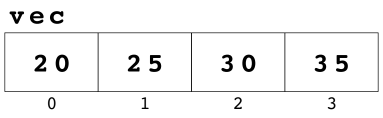
Once again, the implementation takes care of shifting elements to close the hole left by the deleted value.

Now that you have a way of getting elements into a vector, it would be useful to know how to examine them once they are there. The counterpart to array selection in the **`Vector`** class is the **`getAt`** method, which takes an index number and returns the value in that index position. For example, given the most recent of **`vec`**, calling **`vec.getAt(2)`** would return the value 30. If you were to call **`vec.getAt(5)`**, the bounds-checking code in the **`Vector`** implementation would signal an error because no such element exists.

Symmetrically, you can change the value of an element by calling the **`setAt`** method. Calling

```
vec.setAt(3, 35);
```

would change the value in index position 3 to 35, like this:



Even though the **`getAt`** and **`setAt`** methods are relatively simple to use, hardly anyone in fact calls these methods directly. One of the characteristics of C++ that sets it apart from most other languages is that classes can override the definition of the standard operators In particular, C++ allows classes to override the selection operator used to select elements in an array. This feature makes it possible for the **`Vector`** class to support exactly the same selection syntax as arrays. To select the element at position **`i`**, all you need to write is

```
vec[i];
```

To change an element, all you need to do is assign to the selected element. Thus, you can set element 3 in **`vec`** to 35 by writing

```
vec[3] = 35;
```

The resulting syntax is marginally shorter but considerably more evocative of the array operations that the **`Vector`** class tries to emulate.

Iterating through the elements of a `Vector`

As with conventional arrays, one of the most common programming patterns used with vectors is iteration, in which you cycle through each of the elements in turn. The basic idiom for doing so is

```
for (int i = 0; i < vec.size(); i++) {
   loop body
}
```

Inside the loop body, you can refer to the current element as **`vec[i]`**.

As an example, the following code writes out the contents of the vector **`vec`** as a comma-separated list enclosed in square brackets:

```
cout << "[";
for (int i = 0; i < vec.size(); i++) {
   if (i > 0) cout << ", ";
   cout << vec[i];
}
cout << "]" << endl;
```

If you were to execute this code given the most recent contents of **`vec`**, you would see the following output on the screen:

```
[20, 25, 30, 35]
```

Passing a `Vector` as a parameter

The code at the end of the preceding section is so useful (particularly when you're debugging and need to see what values the vector contains), that it would be worth defining a function that does it. At one level, encapsulating those lines as a single function is easy; all you have to do is add the appropriate function header, like this:

```
void PrintVector(Vector<int> & vec) {
   cout << "[";
   for (int i = 0; i < vec.size(); i++) {
      if (i > 0) cout << ", ";
      cout << vec[i];
   }
   cout << "]" << endl;
}
```

The header line, however, involves one subtlety that you absolutely have to understand before you can use the library classes effectively. As described in Chapter 1, the **`&`** before the parameter name indicates that the argument to **`PrintVector`** is passed by reference, which means that the vector in the caller is shared with the vector in the function. Passing by reference is more efficient than C++'s default model of passing by value, which specifies that the entire contents of the argument vector must be copied before passing it along to the function. More importantly, passing by reference makes it possible for you to write functions that change the contents of a vector. As an example, the following function adds the contents of an integer array to a vector, making it possible to convert an array into its vector counterpart:

```
void AddArrayToVector(Vector<int> & vec, int array[], int n) {
   for (int i = 0; i < n; i++) {
      vec.add(array[i]);
   }
}
```

If you had left out the ampersand in this header line, the function would have no effect at all. The code would happily add the first **`n`** elements from **`array`** to a vector, but that vector would be a copy of the one the caller supplied. As soon as **AddArrayToVector** returned, that copy would go away, leaving the original value unchanged. This kind of error is easy to make, and you should learn to look for it when your programs go awry.

The **revfile.cpp** program in Figure 4-1 shows a complete C++ program that uses **Vector** to reverse the lines in a file. The **AskUserForInputFile** and **ReadTextFile** in this example will probably come in handy in a variety of applications, and you might want to keep a copy of this program around so that you can cut-and-paste these functions into your own code.

COMMON PITFALLS

When you are using the classes in the template library, you should *always* pass them by reference. The C++ compiler won't notice if you don't, but the results are unlikely to be what you intend.

Figure 4-1 Program to print the lines of a file in reverse order

```
/*
 * File: revfile.cpp
 * -----------------
 * This program reads in a text file and then displays the lines of
 * the file in reverse order.
 */

#include "genlib.h"
#include "simpio.h"
#include "vector.h"
#include <string>
#include <iostream>
#include <fstream>

/* Function prototypes */

void ReadTextFile(ifstream & infile, Vector<string> & lines);
void AskUserForInputFile(string prompt, ifstream & infile);
void PrintReversed(Vector<string> & lines);

/* Main program */

int main() {
   ifstream infile;
   AskUserForInputFile("Input file: ", infile);
   Vector<string> lines;
   ReadTextFile(infile, lines);
   infile.close();
   PrintReversed(lines);
   return 0;
}

/*
 * Reads an entire file into the Vector<string> supplied by the user.
 */

void ReadTextFile(ifstream & infile, Vector<string> & lines) {
   while (true) {
      string line;
      getline(infile, line);
      if (infile.fail()) break;
      lines.add(line);
   }
}
```

```
/*
 * Opens a text file whose name is entered by the user.  If the file
 * does not exist, the user is given additional chances to enter a
 * valid file.  The prompt string is used to tell the user what kind
 * of file is required.
 */

void AskUserForInputFile(string prompt, ifstream & infile) {
   while (true) {
      cout << prompt;
      string filename = GetLine();
      infile.open(filename.c_str());
      if (!infile.fail()) break;
      cout << "Unable to open " << filename << endl;
      infile.clear();
   }
}

/*
 * Prints the lines from the Vector<string> in reverse order.
 */

void PrintReversed(Vector<string> & lines) {
   for (int i = lines.size() - 1; i >= 0; i--) {
      cout << lines[i] << endl;
   }
}
```

4.2 The `Grid` class

Just as the **`Vector`** class simulates—and improves upon—a single-dimensional array, the **`Grid`** class is designed to provide a better implementation of two-dimensional arrays in C++. As with **`Vector`**, the **`Grid`** class contains individual elements, which means that you need to use parameterized type templates to specify the base type. Thus, if you want to create a two-dimensional grid that contains characters, the appropriate class name would be **`Grid<char>`**.

The discipline for creating a new **`Grid`** object is slightly different from that of **`Vector`**, mostly because clients tend to use the **`Grid`** class in a different way. Particularly when you are reading in elements of a vector from a file, as was true in the **`revfile.cpp`** program in Figure 4-1, it seems natural to start with an empty vector and then add new elements as you go along. For the most part, that's not the way people use two-dimensional arrays. In most cases, you know the size of the two-dimensional array that you want to create. It therefore makes sense to specify the dimensions of a **`Grid`** object when you create it.

In the **`Grid`** class, the most commonly used constructor takes two arguments, which specify the number of rows and the number of columns. When you declare a **`Grid`** variable, you include these arguments in parentheses after the variable, as in a method call. For example, if you want to declare a grid with three rows and two columns in which each value is a **`double`**, you could do so by issuing the following declaration:

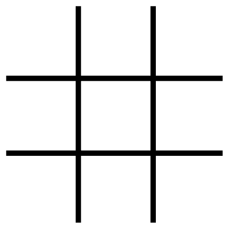
```
Grid<double> matrix(3, 2);
```

Even though the elements of the matrix are created by the constructor, they may not be initialized in any helpful way. If the elements of the grid are objects, they will be initialized by calling the **default constructor** for that class, which is simply the constructor that takes no arguments. If the elements, however, are of a primitive type like **double**, C++ does not initialize them, and their values depend on whatever happened be in the memory locations to which those variables were assigned. It is therefore a good programming practice to initialize the elements of a **Grid** explicitly before you use them.

To initialize the elements of a grid, you need to know what methods are available to manipulate the values the grid contains. The most common methods in the **Grid** class are shown in Table 4-2. As you can see from the table, the **Grid** class does include **getAt** and **setAt** methods that allow you to work with individual elements, but it is far more common to use the more familiar bracket-selection syntax. For example, if you want to set every element in the **matrix** grid to 0.0, you could do so with the following code:

```
for (int i = 0; i < matrix.numRows(); i++) {
   for (int j = 0; j < matrix.numCols(); j++) {
      matrix[i][j] = 0.0;
   }
}
```

Because so many games—including chess, checkers, and go—are played on two-dimensional boards, the **Grid** class is often useful in applications that play those games. One particularly simple grid-based game is tic-tac-toe, which is played on a board with three rows and three columns, as follows:



Players take turns placing the letters *X* and *O* in the empty squares, trying to line up three identical symbols horizontally, vertically, or diagonally.

If you want to represent a tic-tac-toe board using the classes provided in this chapter, the obvious approach is to use a **Grid** with three rows and three columns. Given that

Table 4-2 Methods exported by the Grid class

numRows()
numCols()
	These methods return the number of rows and the number of columns, respectively.

getAt(*row***,** *col***)**	Returns the element of the grid that appears at the specified row and column. As a convenience, the **Grid** class also makes it possible to select an element using array notation, so that **grid[row][col]** selects the element at the specified location.
setAt(*row***,** *col***,** *value***)**	Sets the element at the specified index to the new value. Attempting to reset an element outside the bounds of the vector generates an error.
resize(*rows***,** *cols***)**	Changes the dimensions of the grid as specified by the *rows* and *cols* parameters. Any previous contents of the grid are discarded.
iterator()	Returns an iterator that makes it easy to cycle through the elements of the grid.

each of the elements contains a character—an **X**, an **O**, or a space—the declaration of a board will presumably look like this:

```
Grid<char> board(3, 3);
```

You will have a chance to see an program that plays tic-tac-toe in Chapter 7, but for now, it is probably sufficient to look at how you might manipulate a tic-tac-toe board declared in this way. Figure 4-2, for example, contains the code for checking to see whether a player has won the game by looking to see whether the same character appears in every cell of a row, a column, or a diagonal.

4.3 The `Stack` class

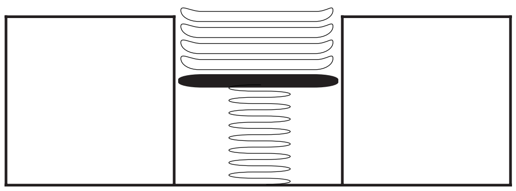
One of the simplest classes in the ADT library is the **Stack** class, which is used to model a data structure called a *stack*, which turns out to be particularly useful in a variety of programming applications. A **stack** provides storage for a collection of data values, subject to the restriction that values must be removed from a stack in the opposite order from which they were added, so that the last item added to a stack is always the first item that gets removed.

Figure 4-2 Program to check whether a player has won a tic-tac-toe game

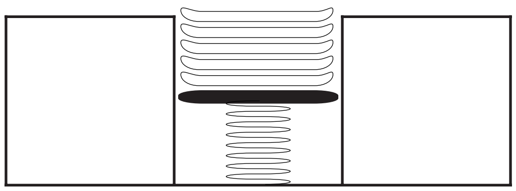
```
/*
 * Checks to see whether the specified player identified by mark
 * ('X' or 'O') has won the game.  To reduce the number of special
 * cases, this implementation uses the helper function CheckLine
 * so that the row, column, and diagonal checks are the same.
 */

bool CheckForWin(Grid<char> & board, char mark) {
   for (int i = 0; i < 3; i++) {
      if (CheckLine(board, mark, i, 0, 0, 1)) return true;
      if (CheckLine(board, mark, 0, i, 1, 0)) return true;
   }
   if (CheckLine(board, mark, 0, 0, 1, 1)) return true;
   return CheckLine(board, mark, 2, 0, -1, 1);
}

/*
 * Checks a line extending across the board in some direction.
 * The starting coordinates are given by the row and col
 * parameters.  The direction of motion is specified by dRow
 * and dCol, which show how to adjust the row and col values
 * on each cycle.  For rows, dRow is always 0; for columns,
 * dCol is 0.  For diagonals, the dRow and dCol values will
 * be +1 or -1 depending on the direction of the diagonal.
 */

bool CheckLine(Grid<char> & board, char mark, int row, int col,
               int dRow, int dCol) {
   for (int i = 0; i < 3; i++) {
      if (board[row][col] != mark) return false;
      row += dRow;
      col += dCol;
   }
   return true;
}
```

Because of their importance in computer science, stacks have acquired a terminology of their own. The values stored in a stack are called its **elements.** Adding a new element to a stack is called **pushing** that element; removing the most recent item from a stack is called **popping** the stack. Moreover, the order in which stacks are processed is sometimes called **LIFO,** which stands for "last in, first out."

The conventional (and possibly apocryphal) explanation for the words *stack, push,* and *pop* is that they come from the following metaphor. In many cafeterias, plates for the food are placed in spring-loaded columns that make it easy for people in the cafeteria line to take the top plate, as illustrated in the following diagram:

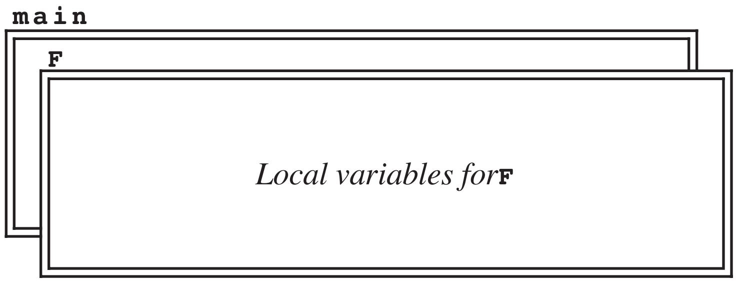


When a dishwasher adds a new plate, it goes on the top of the stack, pushing the others down slightly as the spring is compressed, as shown:

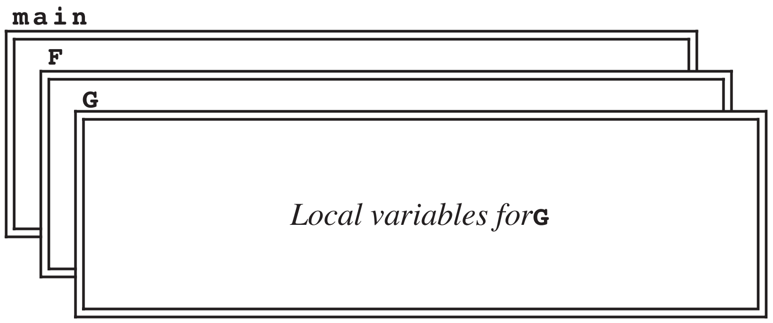


Customers can only take plates from the top of the stack. When they do, the remaining plates pop back up. The last plate added to the stack is the first one a customer takes.

The primary reason that stacks are important in programming is that nested function calls behave in a stack-oriented fashion. For example, if the main program calls a function named **`F`**, a stack frame for **`F`** gets pushed on top of the stack frame for **`main`**.



If **`F`** calls **`G`**, a new stack frame for **`G`** is pushed on top of the frame for **`F`**.



When **G** returns, its frame is popped off the stack, restoring **F** to the top of the stack as shown in the original diagram.

The structure of the **Stack** class

Like **Vector** and **Grid**, **Stack** is a collection class that requires you to specify the base type. For example, **Stack<int>** represents a stack whose elements are integers, and **Stack<string>** represents one in which the elements are strings. Similarly, if you had previously defined the classes **Plate** and **Frame** to contain all the information required to represent a dinner plate and the stack frame for a function, you could represent the stacks described in the preceding two sections with the parameterized classes **Stack<Plate>** and **Stack<Frame>**. The default constructor for the **Stack** class creates an empty stack. The complete list of methods exported by the **Stack** class appears in Table 4-3.

Stacks and pocket calculators

One interesting applications of stacks is in electronic calculators, where they are used to store intermediate results of a calculation. Although stacks play a central role in the operation of most calculators, that role is easiest to see in scientific calculators that require users to enter expressions in a form called *reverse Polish notation,* or *RPN.*

In reverse Polish notation, operators are entered after the operands to which they apply. For example, to compute the result of the expression

```
    50.0 * 1.5 + 3.8 / 2.0
```

on an RPN calculator, you would enter the operations in the following order:

$$50.0\ \boxed{\text{ENTER}}\ 1.5\ \boxed{\times}\ 3.8\ \boxed{\text{ENTER}}\ 2.0\ \boxed{/}\ \boxed{+}$$

When the **ENTER** button is pressed, the calculator takes the previous value and pushes it on a stack. When an operator button is pressed, the calculator first checks whether the user has just entered a value and, if so, automatically pushes it on the stack. It then computes the result of applying the operator by

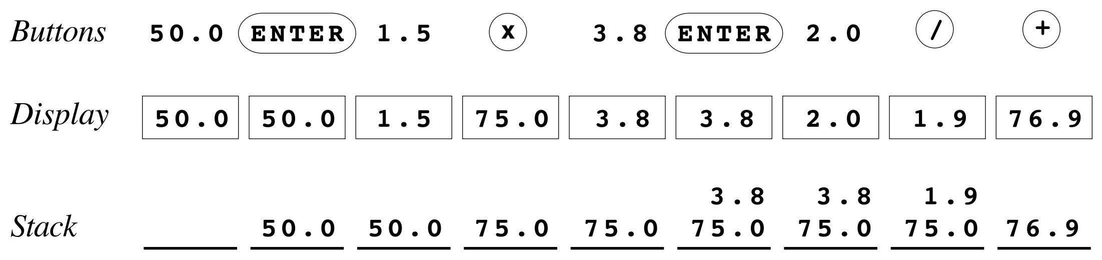
- Popping the top two values from the stack
- Applying the arithmetic operation indicated by the button to these values
- Pushing the result back on the stack

Except when the user is actually typing in a number, the calculator display shows the value at the top of the stack. Thus, at each point in the operation, the calculator display

Table 4-3 Methods exported by the Stack class

size()	Returns the number of elements currently on the stack.
isEmpty()	Returns **true** if the stack is empty.
push(*value***)**	Pushes *value* on the stack so that it becomes the topmost element.
pop()	Pops the topmost value from the stack and returns it to the caller. Calling **pop** on an empty stack generates an error.
peek()	Returns the topmost value on the stack without removing it. As with **pop**, calling **peek** on an empty stack generates an error.
clear()	Removes all the elements from a stack.

and stack contain the values shown:



To implement the RPN calculator described in the preceding section in C++ requires making some changes in the user-interface design. In a real calculator, the digits and operations appear on a keypad. In this implementation, it is easier to imagine that the user enters lines on the console, where those lines take one of the following forms:

- A floating-point number
- An arithmetic operator chosen from the set `+`, `-`, `*`, and `/`
- The letter `Q`, which causes the program to quit
- The letter `H`, which prints a help message
- The letter `C`, which clears any values left on the stack

A sample run of the calculator program might therefore look like this:

```
RPN Calculator Simulation (type H for help)
> 50.0
> 1.5
> *
75
> 3.8
> 2.0
> /
1.9
> +
76.9
> Q
```

Because the user enters each number on a separate line terminated with the RETURN key, there is no need for any counterpart to the calculator's **ENTER** button, which really serves only to indicate that a number is complete. The calculator program can simply push the numbers on the stack as the user enters them. When the calculator reads an operator, it pops the top two elements from the stack, applies the operator, displays the result, and then pushes the result back on the stack.

The complete implementation of the calculator application appears in Figure 4-3.

4.4 The `Queue` class

As you learned in section 4.3, the defining feature of a stack is that the last item pushed is always the first item popped. As noted in the introduction to that section, this behavior is often referred to in computer science as **LIFO,** which is an acronym for the phrase "last in, first out." The LIFO discipline is useful in programming contexts because it reflects the operation of function calls; the most recently called function is the first to return.

Figure 4-3 Implementation of a calculator that uses reverse Polish notation

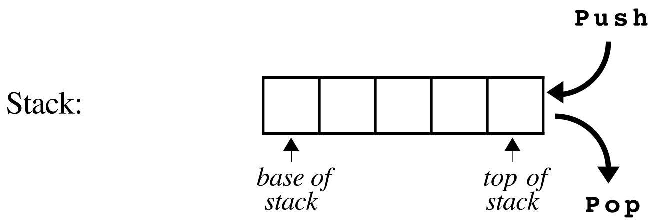
```
/*
 * File: rpncalc.cpp
 * -----------------
 * This program simulates an electronic calculator that uses
 * reverse Polish notation, in which the operators come after
 * the operands to which they apply.
 */

#include <iostream>
#include <cctype>
#include "genlib.h"
#include "simpio.h"
#include "strutils.h"
#include "stack.h"

/* Private function prototypes */

void ApplyOperator(char op, Stack<double> &operandStack);
void HelpCommand();
void ClearStack(Stack<double> &operandStack);

/* Main program */

int main() {
    Stack<double> operandStack;

    cout << "RPN Calculator Simulation (type H for help)" << endl;
    while (true) {
        cout << "> ";
        string line = GetLine();
        char ch = toupper(line[0]);
        if (ch == 'Q') {
            break;
        } else if (ch == 'C') {
            ClearStack(operandStack);
        } else if (ch == 'H') {
            HelpCommand();
        } else if (isdigit(ch)) {
            operandStack.push(StringToReal(line));
        } else {
            ApplyOperator(ch, operandStack);
        }
    }
    return 0;
}
```

```
/*
 * Function: ApplyOperator
 * Usage: ApplyOperator(op, operandStack);
 * ---------------------------------------
 * This function applies the operator to the top two elements on
 * the operand stack.  Because the elements on the stack are
 * popped in reverse order, the right operand is popped before
 * the left operand.
 */

void ApplyOperator(char op, Stack<double> &operandStack) {
    double result;

    double rhs = operandStack.pop();
    double lhs = operandStack.pop();
    switch (op) {
      case '+': result = lhs + rhs; break;
      case '-': result = lhs - rhs; break;
      case '*': result = lhs * rhs; break;
      case '/': result = lhs / rhs; break;
      default:  Error("Illegal operator");
    }
    cout << result << endl;
    operandStack.push(result);
}

/*
 * Function: HelpCommand
 * Usage: HelpCommand();
 * ---------------------
 * This function generates a help message for the user.
 */

void HelpCommand() {
    cout << "Enter expressions in Reverse Polish Notation," << endl;
    cout << "in which operators follow the operands to which" << endl;
    cout << "they apply.  Each line consists of a number, an" << endl;
    cout << "operator, or one of the following commands:" << endl;
    cout << "  Q -- Quit the program" << endl;
    cout << "  H -- Display this help message" << endl;
    cout << "  C -- Clear the calculator stack" << endl;
}

/*
 * Function: ClearStack
 * Usage: ClearStack(stack);
 * -------------------------
 * This function clears the stack by popping elements until empty.
 */

void ClearStack(Stack<double> &stack) {
    while (!stack.isEmpty()) {
        stack.pop();
    }
}
```

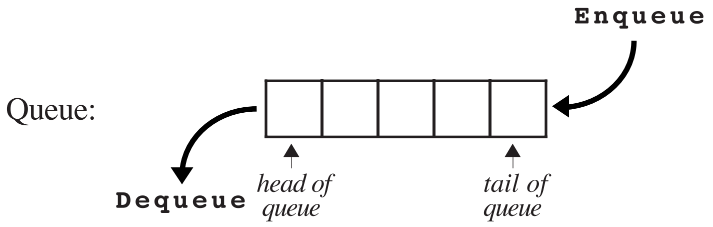
In real-world situations, however, its usefulness is more limited. In human society, our collective notion of fairness assigns some priority to being first, as expressed in the maxim “first come, first served.” In programming, the usual phrasing of this ordering strategy is “first in, first out,” which is traditionally abbreviated as **FIFO.**

A data structure that stores items using a FIFO discipline is called a **queue.** The fundamental operations on a queue—which are analogous to the **`push`** and **`pop`** operations for stacks—are called **`enqueue`** and **`dequeue`**. The **`enqueue`** operation adds a new element to the end of the queue, which is traditionally called its **tail.** The **`dequeue`** operation removes the element at the beginning of the queue, which is called its **head.**

The conceptual difference between these structures can be illustrated most easily with a diagram. In a stack, the client must add and remove elements from the same end of the internal data structure, as follows:



In a queue, the client adds elements at one end and removes them from the other, like this:



As you might expect from the fact that the conceptual are so similar, the structure of the **`Queue`** class looks very much like its **`Stack`** counterpart. The list of methods in the **`Queue`** class shown in Table 4-4 bears out that supposition. The only differences are in the terminology, which reflects the difference in the ordering of the elements.

The queue data structure has many applications in programming. Not surprisingly, queues turn up in many situations in which it is important to maintain a first-in/first-out discipline in order to ensure that service requests are treated fairly. For example, if you are working in an environment in which a single printer is shared among several

Table 4-4 Methods exported by the `Queue` class

`size()`	Returns the number of elements currently on the queue.
`isEmpty()`	Returns **`true`** if the queue is empty.
`enqueue(`*value***`)`**	Adds *value* to the tail of the queue.
`dequeue()`	Removes the element at the head of the queue and returns it to the caller. Calling **`dequeue`** on an empty queue generates an error.
`peek()`	Returns the value at the head of the queue without removing it. As with **`dequeue`** , calling **`peek`** on an empty queue generates an error.
`clear()`	Removes all the elements from a queue.

computers, the printing software is usually designed so that all print requests are entered in a queue. Thus, if several users decide to enter print requests, the queue structure ensures that each user's request is processed in the order received.

Queues are also common in programs that simulate the behavior of waiting lines. For example, if you wanted to decide how many cashiers you needed in a supermarket, it might be worth writing a program that could simulate the behavior of customers in the store. Such a program would almost certainly involve queues, because a checkout line operates in a first-in/first-out way. Customers who have completed their purchases arrive in the checkout line and wait for their turn to pay. Each customer eventually reaches the front of the line, at which point the cashier totals up the purchases and collects the money. Because simulations of this sort represent an important class of application programs, it is worth spending a little time understanding how such simulations work.

Simulations and models

Beyond the world of programming, there are an endless variety of real-world events and processes that—although they are undeniably important—are nonetheless too complicated to understand completely. For example, it would be very useful to know how various pollutants affect the ozone layer and how the resulting changes in the ozone layer affect the global climate. Similarly, if economists and political leaders had a more complete understanding of exactly how the national economy works, it would be possible to evaluate whether a cut in the capital-gains tax would spur investment or whether it would exacerbate the existing disparities of wealth and income.

When faced with such large-scale problems, it is usually necessary to come up with an idealized **model,** which is a simplified representation of some real-world process. Most problems are far too complex to allow for a complete understanding. There are just too many details. The reason to build a model is that, despite the complexity of a particular problem, it is often possible to make certain assumptions that allow you to simplify a complicated process without affecting its fundamental character. If you can come up with a reasonable model for a process, you can often translate the dynamics of the model into a program that captures the behavior of that model. Such a program is called a **simulation.**

It is important to remember that creating a simulation is usually a two-step process. The first step consists of designing a conceptual model for the real-world behavior you are trying to simulate. The second consists of writing a program that implements the conceptual model. Because errors can occur in both steps of the process, maintaining a certain skepticism about simulations and their applicability to the real world is probably wise. In a society conditioned to believe the "answers" delivered by computers, it is critical to recognize that the simulations can never be better than the models on which they are based.

The waiting-line model

Suppose that you want to design a simulation that models the behavior of a supermarket waiting line. By simulating the waiting line, you can determine some useful properties of waiting lines that might help a company make such decisions as how many cashiers are needed, how much space needs to be reserved for the line itself, and so forth.

The first step in the process of writing a checkout-line simulation is to develop a model for the waiting line, detailing the simplifying assumptions. For example, to make the initial implementation of the simulation as simple as possible, you might begin by assuming that there is one cashier who serves customers from a single queue. You might then assume that customers arrive with a random probability and enter the queue at the end of the line. Whenever the cashier is free and someone is waiting in line, the cashier begins to serve that customer. After an appropriate service period—which you must also model in some way—the cashier completes the transaction with the current customer, and is free to serve the next customer in the queue.

Discrete time

Another assumption often required in a model is some limitation on the level of accuracy. Consider, for example, the time that a customer spends being served by the cashier. One customer might spend two minutes; another might spend six. It is important, however, to consider whether measuring time in minutes allows the simulation to be sufficiently precise. If you had a sufficiently accurate stopwatch, you might discover that a customer actually spent 3.14159265 minutes. The question you need to resolve is how accurate you need to be.

For most models, and particularly for those intended for simulation, it is useful to introduce the simplifying assumption that all events within the model happen in discrete integral time units. Using discrete time assumes that you can find a time unit that—for the purpose of the model—you can treat as indivisible. In general, the time units used in a simulation must be small enough that the probability of more than one event occurring during a single time unit is negligible. In the checkout-line simulation, for example, minutes may not be accurate enough; two customers could easily arrive in the same minute. On the other hand, you could probably get away with using seconds as the time unit and discount the possibility that two customers arrive in precisely the same second.

Although the checkout-line example assumes that simulation time is measured in seconds, in general, there is no reason you have to measure time in conventional units. When you write a simulation, you can define the unit of time in any way that fits the structure of the model. For example, you could define a time unit to be five seconds and then run the simulation as a series of five-second intervals.

Events in simulated time

The real advantage of using discrete time units is not that it makes it possible to work with variables of type **int** instead of being forced to use type **double**. The most important property of discrete time is that it allows you to structure the simulation as a loop in which each time unit represents a single cycle. When you approach the problem in this way, a simulation program has the following form:

```
for (int time = 0; time < SIMULATION_TIME; time++) {
    Execute one cycle of the simulation.
}
```

Within the body of the loop, the program performs the operations necessary to advance through one unit of simulated time.

Think for a moment about what events might occur during each time unit of the checkout-line simulation. One possibility is that a new customer might arrive. Another is that the cashier might finish with the current customer and go on the serve the next person in line. These events bring up some interesting issues. To complete the model, you need to say something about how often customers arrive and how much time they spend at the cash register. You could (and probably should) gather approximate data by watching a real checkout line in a store. Even if you collect that information, however, you will need to simplify it to a form that (1) captures enough of the real-world behavior to be useful and (2) is easy to understand in terms of the model. For example, your surveys might show that customers arrive at the line on average once every 20 seconds. This average arrival rate is certainly useful input to the model. On the other hand, you would not have much confidence in a simulation in which customers arrived exactly once every 20 seconds. Such an implementation would violate the real-world condition that customer arrivals have some random variability and that they sometimes bunch together.

For this reason, the arrival process is usually modeled by specifying the probability that an arrival takes place in any discrete time unit instead of the average time between arrivals. For example, if your studies indicated that a customer arrived once every 20 seconds, the average probability of a customer arriving in any particular second would be 1/20 or 0.05. If you assume that arrivals occur randomly with an equal probability in each unit of time, the arrival process forms a pattern that mathematicians call a **Poisson distribution.**

You might also choose to make simplifying assumptions about how long it takes to serve a particular customer. For example, the program is easier to write if you assume that the service time required for each customer is uniformly distributed within a certain range. If you do, you can use the **`RandomInteger`** function from the **`random.h`** interface to pick the service time.

Implementing the simulation

Even though it is longer than the other programs in this chapter, the code for the simulation program is reasonably easy to write and appears in Figure 4-4. The core of the simulation is a loop that runs for the number of seconds indicated by the parameter **`SIMULATION_TIME`**. In each second, the simulation performs the following operations:

1. Determine whether a new customer has arrived and, if so, add that person to the queue.
2. If the cashier is busy, note that the cashier has spent another second with the current customer. Eventually, the required service time will be complete, which will free the cashier.
3. If the cashier is free, serve the next customer in the waiting line.

The simulation is controlled by the following parameters:

- **`SIMULATION_TIME`**—This parameter specifies the duration of the simulation.
- **`ARRIVAL_PROBABILITY`**—This parameter indicates the probability that a new customer will arrive at the checkout line during a single unit of time. In keeping with standard statistical convention, the probability is expressed as a real number between 0 and 1.
- **`MIN_SERVICE_TIME`**, **`MAX_SERVICE_TIME`**—These parameters define the legal range of customer service time. For any particular customer, the amount of time spent at the cashier is determined by picking a random integer in this range.

In addition to these parameters, there is also a variable **`traceFlag`** that controls whether the program produces a debugging log. The flag is set to **`false`** in the finished version of the application, but it is probably a good idea to leave the debugging code in the program. That way, if someone needs to modify the program later, they can set **`traceFlag`** to **`true`** and see the debugging output.

When the simulation is complete, the program reports the simulation parameters along with the following results:

- The number of customers served
- The average amount of time customers spent in the waiting line
- The average length of the waiting line

Figure 4-4 Program to simulate a checkout line

```
/*
 * File: checkout.cpp
 * ------------------
 * This program simulates a checkout line, such as one you
 * might encounter in a grocery store.  Customers arrive at
 * the checkout stand and get in line.  Those customers wait
 * in the line until the cashier is free, at which point
 * they are served and occupy the cashier for some period
 * of time.  After the service time is complete, the cashier
 * is free to serve the next customer in the line.
 *
 * In each unit of time, up to the parameter SIMULATION_TIME,
 * the following operations are performed:
 *
 * 1. Determine whether a new customer has arrived.
 *    New customers arrive randomly, with a probability
 *    determined by the parameter ARRIVAL_PROBABILITY.
 *
 * 2. If the cashier is busy, note that the cashier has
 *    spent another minute with that customer.  Eventually,
 *    the customer's time request is satisfied, which frees
 *    the cashier.
 *
 * 3. If the cashier is free, serve the next customer in line.
 *    The service time is taken to be a random period between
 *    MIN_SERVICE_TIME and MAX_SERVICE_TIME.
 *
 * At the end of the simulation, the program displays the
 * parameters and the following computed results:
 *
 * o  The number of customers served
 * o  The average time spent in line
 * o  The average number of people in line
 */

#include "genlib.h"
#include "random.h"
#include "queue.h"
#include <iostream>
#include <iomanip>

/* Simulation parameters */

const int SIMULATION_TIME = 2000;
const double ARRIVAL_PROBABILITY = 0.1;
const int MIN_SERVICE_TIME =  5;
const int MAX_SERVICE_TIME = 15;
```

```
/*
 * Type: customerT
 * ---------------
 * A customer is represented using a record
 * containing the following information:
 *
 * o The customer number (for debugging traces)
 * o The arrival time (to compute the waiting time)
 * o The time required for service
 */

struct customerT {
    int customerNumber;
    int arrivalTime;
    int serviceTime;
};

/*
 * Type: simDataT
 * --------------
 * This type stores the data required for the simulation.  The
 * main program declares a variable of this type and then passes
 * it by reference to every other function in the program.
 */

struct simDataT {
    Queue<customerT> queue;
    customerT activeCustomer;
    bool hasActiveCustomer;
    int time;
    int numCustomers;
    int numServed;
    long totalWaitTime;
    long totalLineLength;
};

/*
 * Debugging option: traceFlag
 * ---------------------------
 * This variable controls whether the simulation produces a
 * debugging trace.
 */

static bool traceFlag = true;

/* Private function prototypes */

void InitializeSimulation(simDataT & sd);
void RunSimulation(simDataT & sd);
void EnqueueCustomer(simDataT & sd);
void ProcessQueue(simDataT & sd);
void ServeCustomer(simDataT & sd);
void DismissCustomer(simDataT & sd);
void ReportResults(simDataT & sd);
```

```
/* Main program */

int main() {
   simDataT simData;

   Randomize();
   InitializeSimulation(simData);
   RunSimulation(simData);
   ReportResults(simData);
   return 0;
}

/*
 * Function: InitializeSimulation
 * Usage: InitializeSimulation(simData);
 * -------------------------------------
 * This function initializes the simulation data block whose
 * address is passed as the argument.
 */

void InitializeSimulation(simDataT & sd) {
   sd.hasActiveCustomer = false;
   sd.numServed = 0;
   sd.totalWaitTime = 0;
   sd.totalLineLength = 0;
}

/*
 * Function: RunSimulation
 * Usage: RunSimulation(simData);
 * ------------------------------
 * This function runs the actual simulation.  In each time unit,
 * the program first checks to see whether a new customer arrives.
 * Then, if the cashier is busy (indicated by the boolean flag
 * in the hasActiveCustomer field), the program decrements
 * the service time counter for that customer.  Finally, if the
 * cashier is free, it serves another customer from the queue
 * and updates the necessary bookkeeping data.
 */

void RunSimulation(simDataT & sd) {
   for (sd.time = 0; sd.time < SIMULATION_TIME; sd.time++) {
      if (RandomChance(ARRIVAL_PROBABILITY)) {
         EnqueueCustomer(sd);
      }
      ProcessQueue(sd);
   }
}
```

```
/*
 * Function: EnqueueCustomer
 * Usage: EnqueueCustomer(simData);
 * --------------------------------
 * This function simulates the arrival of a new customer.
 */

void EnqueueCustomer(simDataT & sd) {
   customerT c;

   sd.numCustomers++;
   c.customerNumber = sd.numCustomers;
   c.arrivalTime = sd.time;
   c.serviceTime = RandomInteger(MIN_SERVICE_TIME, MAX_SERVICE_TIME);
   sd.queue.enqueue(c);
   if (traceFlag) {
      cout << setw(4) << sd.time << ": Customer " << c.customerNumber
           << " arrives and gets in line." << endl;
   }
}

/*
 * Function: ProcessQueue
 * Usage: ProcessQueue(simData);
 * -----------------------------
 * This function processes a single time cycle for the queue.
 */

void ProcessQueue(simDataT & sd) {
   if (!sd.hasActiveCustomer) {
      if (!sd.queue.isEmpty()) {
         ServeCustomer(sd);
      }
   } else {
      if (sd.activeCustomer.serviceTime == 0) {
         DismissCustomer(sd);
      } else {
         sd.activeCustomer.serviceTime--;
      }
   }
   sd.totalLineLength += sd.queue.size();
}

/*
 * Function: ServeCustomer
 * Usage: ServeCustomer(simData);
 * ------------------------------
 * This function is called when the cashier is free and a
 * customer is waiting.  The effect is to serve the first
 * customer in the line and update the total waiting time.
 */
```

```
void ServeCustomer(simDataT & sd) {
   customerT c = sd.queue.dequeue();
   sd.activeCustomer = c;
   sd.hasActiveCustomer = true;
   sd.numServed++;
   sd.totalWaitTime += (sd.time - c.arrivalTime);
   if (traceFlag) {
      cout << setw(4) << sd.time << ": Customer "
           << c.customerNumber << " reaches cashier." << endl;
   }
}

/*
 * Function: DismissCustomer
 * Usage: DismissCustomer(simData);
 * --------------------------------
 * This function is called when the active customer's service
 * time has dropped to 0. The cashier becomes free and the
 * program no longer needs to hold the customer's storage.
 */

void DismissCustomer(simDataT & sd) {
   if (traceFlag) {
      cout << setw(4) << sd.time << ": Customer "
           << sd.activeCustomer.customerNumber
           << " leaves cashier." << endl;
   }
   sd.hasActiveCustomer = false;
}

/*
 * Function: ReportResults
 * Usage: ReportResults(simData);
 * ------------------------------
 * This function reports the results of the simulation.
 */

void ReportResults(simDataT & sd) {
   cout << "Simulation results given the following parameters:"
        << endl;
   cout << fixed << setprecision(2);
   cout << "  SIMULATION_TIME:     " << setw(4)
        << SIMULATION_TIME << endl;
   cout << "  ARRIVAL_PROBABILITY: " << setw(7)
        << ARRIVAL_PROBABILITY << endl;
   cout << "  MIN_SERVICE_TIME:     " << setw(4)
        << MIN_SERVICE_TIME << endl;
   cout << "  MAX_SERVICE_TIME:     " << setw(4)
        << MAX_SERVICE_TIME << endl;
   cout << endl;
   cout << "Customers served:     " << setw(4)
        << sd.numServed << endl;
   cout << "Average waiting time: " << setw(7)
        << double(sd.totalWaitTime) / sd.numServed << endl;
   cout << "Average line length:  " << setw(7)
        << double(sd.totalLineLength) / SIMULATION_TIME << endl;
}
```

For example, the following sample run shows the results of the simulation for the indicated parameter values:

```
Simulation results given the following parameters:
  SIMULATION_TIME:     2000
  ARRIVAL_PROBABILITY:    0.05
  MIN_SERVICE_TIME:       5
  MAX_SERVICE_TIME:      15

Customers served:       117
Average waiting time:    15.24
Average line length:      0.90
```

The behavior of the simulation depends significantly on the values of its parameters. Suppose, for example, that the probability of a customer arriving increases from 0.05 to 0.10. Running the simulation with these parameters gives the following results:

```
Simulation results given the following parameters:
  SIMULATION_TIME:     2000
  ARRIVAL_PROBABILITY:    0.10
  MIN_SERVICE_TIME:       5
  MAX_SERVICE_TIME:      15

Customers served:       166
Average waiting time:   237.47
Average line length:     23.35
```

As you can see, doubling the probability of arrival causes the average waiting time to grow from approximately 15 seconds to nearly four minutes, which is obviously a dramatic increase. The reason for the poor performance is that the arrival rate in the second run of the simulation means that new customers arrive at the same rate at which they are served. When this arrival level is reached, the length of the queue and the average waiting time begin to grow very quickly. Simulations of this sort make it possible to experiment with different parameter values. Those experiments, in turn, make it possible to identify potential sources of trouble in the corresponding real-world systems.

4.5 The `Map` class

This section introduces another generic collection called a **map,** which is conceptually similar to a dictionary. A dictionary allows you to look up a word to find its meaning. A map is a generalization of this idea that provides an association between an identifying tag called a **key** and an associated **value,** which may be a much larger and more complicated structure. In the dictionary example, the key is the word you're looking up, and the value is its definition.

Maps have many applications in programming. For example, an interpreter for a programming language needs to be able to assign values to variables, which can then be referenced by name. A map makes it easy to maintain the association between the name of a variable and its corresponding value. When they are used in this context, maps are often called **symbol tables,** which is just another name for the same concept.

The structure of the `Map` class

As with the collection classes introduced earlier in this chapter, **`Map`** is implemented as a template class than must be parameterized with its value type. For example, if you want to simulate a dictionary in which individual words are associated with their definitions, you can start by declaring a **`dictionary`** variable as follows:

```
Map<string> dictionary;
```

Similarly, if you use a **`Map`** to store values of floating-point variables in a programming language, you could start with the following definition:

```
Map<double> symbolTable;
```

These definitions create empty maps that contain no keys and values. In either case, you would subsequently need to add key/value pairs to the map. In the case of the dictionary, you could read the contents from a data file. For the symbol table, you would add new associations whenever an assignment statement appeared.

It is important to note that the parameter for the **`Map`** class specifies the type of the *value,* and not the type of the *key*. In many implementations of collection classes—including, for example, the one in the Standard Template Library and its counterpart in the Java collection classes—you can specify the type of the key as well. The **`Map`** class used in this book avoid considerable complexity by insisting that all keys be strings. Strings are certainly the most common type for keys, and it is usually possible to convert other types to strings if you want to use them as map keys. For example, if you want to use integers as keys, you can simply call the **`IntegerToString`** function on the integer version of the key and then use the resulting string for all map operations.

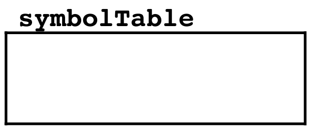
The most common methods used with the **`Map`** class appear in Table 4-5. Of these, the ones that implement the fundamental behavior of the map concept are **`put`** and **`get`**. The **`put`** method creates an association between a key and a value. Its operation is analogous to assigning a value to a variable in C++: if there is a value already associated with the key, the old value is replaced by the new one. The **`get`** method retrieves the value most recently associated with a particular key and therefore corresponds to the act of using a

Table 4-5 Methods exported by the `Map` class

`size()`	Returns the number of key/value pairs contained in the map.
`isEmpty()`	Returns **`true`** if the map is empty.
`put(`*key***`,`** *value***`)`**	Associates the specified *key* and *value* in the map. If *key* has no previous definition, a new entry is added; if a previous association exists, the old value is discarded and replace by the new one.
`get(`*key***`)`**	Returns the value currently associated with *key* in the map. If there is no such value, **`get`** generates an error.
`remove(`*key***`)`**	Removes *key* from the map along with any associated value. If *key* does not exist, this call leaves the map unchanged.
`containsKey(`*key***`)`**	Checks to see whether *key* is associated with a value. If so, this method returns **`true`**; if not, it returns **`false`**.
`clear()`	Removes all the key/value pairs from the map.
`iterator()`	Returns an iterator that makes it easy to cycle through the keys in the map.

variable name to retrieve its value. If no value appears in the map for a particular key, calling **get** with that key generates an error condition. You can check for that condition by calling the **containsKey** method, which returns **true** or **false** depending on whether the key exists in the map.

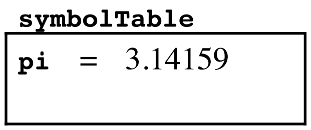
A few simple diagrams may help to illustrate the operation of the **Map** class in more detail. Suppose that you have declared the **symbolTable** variable to be a **Map<double>** as you saw earlier in the section. That declaration creates an empty map with no associations, which can be represented as the following empty box:



Once you have the map, you can use **put** to establish new associations. For example, if you were to call

```
symbolTable.put("pi", 3.14159);
```

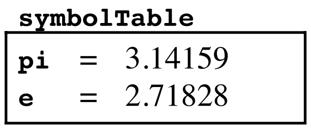
the conceptual effect would be to add an association inside the box between the key **"pi"** and the value 3.14159, as follows:



If you then called

```
symbolTable.put("e", 2.71828);
```

a new association would be added between the key **"e"** and the value 2.71828, like this:

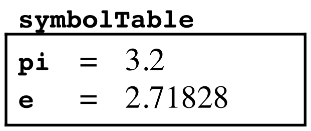


You can then use **get** to retrieve these values. Calling **symbolTable.get("pi")** would return the value 3.14159, and calling **symbolTable.get("pi")** would return 2.71828.

Although it hardly makes sense in the case of mathematical constants, you could change the values in the map by making additional calls to **put**. You could, for example, reset the value associated with **"pi"** (as an 1897 bill before the Indiana State General Assembly sought to do) by calling

```
symbolTable.put("pi", 3.2);
```

which would leave the map in the following state:



At this point, calling **symbolTable.containsKey("pi")** would return **true**; by contrast, calling **symbolTable.containsKey("x")** would return **false**.

Using maps in an application

If you fly at all frequently, you quickly learn that every airport in the world has a three-letter code assigned by the International Air Transport Association (IATA). For example, John F. Kennedy airport in New York City is assigned the three-letter code JFK. Other codes, however, are considerably harder to recognize. Most web-based travel systems offer some means of looking up these codes as a service to their customers.

Suppose that you have been asked to write a simple C++ program that reads a three-letter airport code from the user and responds with the location of that airport. The data you need is in the form of a text file called `AirportCodes.txt`, which contains a list of the several thousand airport codes that IATA has assigned. Each line of the file consists of a three-letter code, an equal sign, and the location of the airport. If the file were sorted in descending order by passenger traffic as of 2007, the file would begin with the lines in Figure 4-5.

Given the `Map` class, the code for this application fits on a single page, as shown in Figure 4-6. That program makes it possible for the user to see the following sample run:

```
Airport code: LHR
LHR is in London, England, United Kingdom
Airport code: SFO
SFO is in San Francisco, CA, USA
Airport code: XXX
There is no such airport code
Airport code:
```

Figure 4-5. The first 25 lines in `AirportCodes.txt`

```
ATL=Atlanta, GA, USA
ORD=Chicago, IL, USA
LHR=London, England, United Kingdom
HND=Tokyo, Japan
LAX=Los Angeles, CA, USA
CDG=Paris, France
DFW=Dallas/Ft Worth, TX, USA
FRA=Frankfurt, Germany
PEK=Beijing, China
MAD=Madrid, Spain
DEN=Denver, CO, USA
AMS=Amsterdam, Netherlands
JFK=New York, NY, USA
HKG=Hong Kong, Hong Kong
LAS=Las Vegas, NV, USA
IAH=Houston, TX, USA
PHX=Phoenix, AZ, USA
BKK=Bangkok, Thailand
SIN=Singapore, Singapore
MCO=Orlando, FL, USA
EWR=Newark, NJ, USA
DTW=Detroit, MI, USA
SFO=San Francisco, CA, USA
NRT=Tokyo, Japan
LGW=London, England, United Kingdom
   .
   .
   .
```

Figure 4-6 Program to look up three-letter airport codes

```
/*
 * File: airports.cpp
 * ------------------
 * This program looks up a three-letter airport code in a Map object.
 */

#include "genlib.h"
#include "simpio.h"
#include "strutils.h"
#include "map.h"
#include <iostream>
#include <fstream>
#include <string>

/* Private function prototypes */

void ReadCodeFile(Map<string> & map);

/* Main program */

int main() {
   Map<string> airportCodes;
   ReadCodeFile(airportCodes);
   while (true) {
      cout << "Airport code: ";
      string code = ConvertToUpperCase(GetLine());
      if (code == "") break;
      if (airportCodes.containsKey(code)) {
         cout << code << " is in " << airportCodes.get(code) << endl;
      } else {
         cout << "There is no such airport code" << endl;
      }
   }
   return 0;
}

/* Reads the data file into the map */

void ReadCodeFile(Map<string> & map) {
   ifstream infile;
   infile.open("AirportCodes.txt");
   if (infile.fail()) Error("Can't read the data file");
   while (true) {
      string line;
      getline(infile, line);
      if (infile.fail()) break;
      if (line.length() < 4 || line[3] != '=') {
         Error("Illegal data file line: " + line);
      }
      string code = ConvertToUpperCase(line.substr(0, 3));
      map.put(code, line.substr(4));
   }
   infile.close();
}
```

Maps as associative arrays

The **`Map`** class overloads the square bracket operators used for array selection so that the statement

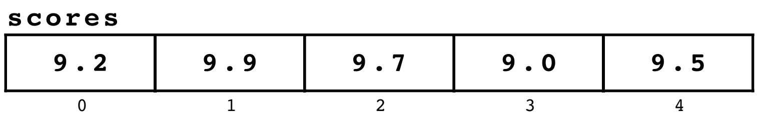
```
map[key] = value;
```

acts as a shorthand for

```
map.put(key, value);
```

and the expression **`map[key]`** returns the value from **`map`** associated with **`key`** in exactly the same way that **`map.get(key)`** does. While these shorthand forms of the **`put`** and **`get`** methods are certainly convenient, using array notation for maps is initially somewhat surprising, given that maps and arrays seem to be rather different in their structure. If you think about maps and arrays in the right way, however, they turn out to be more alike than you might at first suspect.

The insight necessary to unify these two seemingly different structures is that you can think of arrays as structures that map index positions to elements. For example, the array



used as an example in Chapter 2 maps the key 0 into the value 9.2, the key 1 into 9.9, the key 2 into 9.7, and so forth. Thus, an array is in some sense just a map with integer keys. Conversely, you can think of a map as an array that uses strings as index positions, which is precisely what the overloaded selection syntax for the **`Map`** class suggests.

Using array syntax to perform map-like operations is becoming increasingly common in programming languages beyond the C++ domain. Many popular scripting languages implement all arrays internally as maps, making it possible use index values that are not necessarily integers. Arrays implemented using maps as their underlying representation are called **associative arrays.**

4.6 The `Lexicon` class

A **lexicon** is conceptually a dictionary from which the definitions have been removed. Given a lexicon, you can only tell whether a word exists; there are no definitions or values associated with the individual words.

The structure of the `Lexicon` class

The **`Lexicon`** class offers two different forms of the constructor. The default constructor creates an empty lexicon to which you can add new words. In many applications, however, it is convenient to provide the constructor with the name of a data file that contains the words you want to include. For example, if the file **`EnglishWords.dat`** contains a list of all English words, you could use that file to create a lexicon using the followng declaration:

```
Lexicon english("EnglishWords.dat");
```

The implementation of the **`Lexicon`** class allows these data files to be in either of two formats:

1. A text file in which the words appear in any order, one word per line.

2. A precompiled data file that mirrors the internal representation of the lexicon. Using precompiled files (such as `EnglishWords.dat`) is more efficient, both in terms of space and time.

Unlike the classes presented earlier in this chapter, `Lexicon` does not require a type parameter, because a lexicon doesn't contain any values. It does, of course, contain a set of words, but the words are always strings.

The methods available in the `Lexicon` class appear in Table 4-6. The most commonly used method is `containsWord`, which checks to see if a word is in the lexicon. Assuming that you have initialized the variable `english` so that it contains a lexicon of all English words, you could see if a particular word exists by writing a test such as the following:

```
if (english.containsWord(word)) . . .
```

And because it is useful to make such tests in a variety of applications, you can also determine whether any English words begin with a particular substring by calling

```
if (english.containsPrefix(prefix)) . . .
```

A simple application of the `Lexicon` class

In many word games, such as the popular Scrabble™ crossword game, it is critical to memorize as many two letter words as you can, because knowing the two-letter words makes it easier to attach new words to the existing words on the board. Given that you have a lexicon containing English words, you could create such a list by generating all two-letter strings and then using the lexicon to check which of the resulting combinations are actually words. The code to do so appears in Figure 4-7.

As you will discover in section 4.8, it is also possible to solve this problem by going through the lexicon and printing out the words whose length is two. Given that there are more than 100,000 English words in the lexicon and only 676 (26×26) combinations of two letters, the strategy used in Figure 4-7 is probably more efficient.

Table 4-6 Methods exported by the `Lexicon` class

`size()`	Returns the number of words in the lexicon.
`isEmpty()`	Returns `true` if the lexicon is empty.
`add(`*word*`)`	Adds a new word to the lexicon. If the word is already in the lexicon, this call has no effect; each word may appear only once. All words in a lexicon are stored in lower case.
`addWordsFromFile(`*name*`)`	Adds all the words in the named file to the lexicon. The file must either be a text file, in which case the words are listed on separate lines, or a precompiled data file, in which the contents of the file match the internal structure of the lexicon. The `addWordsFromFile` method can read a precompiled file only if the lexicon is empty.
`containsWord(`*word*`)`	Returns `true` if *word* is in the lexicon.
`containsPrefix(`*prefix*`)`	Returns `true` if any of the words in the lexicon start with the specified prefix.
`clear()`	Removes all the elements from a lexicon.
`iterator()`	Returns an iterator that makes it easy to cycle through the words in the lexicon.

Figure 4-7 Program to display all two-letter English words

```
/*
 * File: twoletters.cpp
 * --------------------
 * This program generates a list of the two-letter words.
 */

#include "genlib.h"
#include "lexicon.h"
#include <iostream>

int main() {
   Lexicon english("EnglishWords.dat");
   string word = "xx";
   for (char c0 = 'a'; c0 <= 'z'; c0++) {
      word[0] = c0;
      for (char c1 = 'a'; c1 <= 'z'; c1++) {
         word[1] = c1;
         if (english.containsWord(word)) {
            cout << word << endl;
         }
      }
   }
   return 0;
}
```

Why are lexicons useful if maps already exist

If you think about it, a lexicon is in many ways just a simplified version of a map in which you ignore the values altogether. It would therefore be easy enough to build most of the `Lexicon` class on top of the `Map` class. Adding a word to the lexicon corresponds to calling `put` using the word as the key; checking whether a word exists corresponds to calling `containsKey`.

Given that the `Map` class already provides most of the functionality of the `Lexicon`, it may seem odd that both classes are included in this book. When we designed the ADT library, we chose to include a separate `Lexicon` class for the following reasons:

- Not having to worry about the values in a map makes it possible to implement the lexicon in a more efficient way, particularly in terms of how much memory is required to store the data. Given the data structure used in the `Lexicon` implementation, the entire English dictionary requires approximately 350,000 bytes. If you were to use a `Map` to store the words, the storage requirements would be more than five times greater.
- The underlying representation used in the lexicon makes it possible to check not only whether a word exists in the lexicon, but also to find out whether any word in the lexicon starts with a particular set of letters (the `containsPrefix` method).
- The lexicon representation ensures that the words remain in alphabetical order.

Although these characteristics of the `Lexicon` class are clearly advantages, it is still somewhat surprising that these reasons focus on what seem to be implementation details, particularly in light of the emphasis this chapter places on ignoring such details. By choosing to include the `Lexicon` class, those details are not being revealed to the client. After reading the justification for the `Lexicon` class, you have no idea *why* it might be more efficient than the `Map` class for storing a list of words, but you might well choose to take advantage of that fact.

4.7 The `Scanner` class

The last of the abstract data types introduced in this chapter is the **`Scanner`** class, which provides a useful tool for dividing up a string into meaningful units that are larger than a single character. After all, when you read text on a page, you don't ordinarily pay much attention to the individual letters. Your eye instead groups letters to form words, which it then recognizes as independent units. The **`Scanner`** class does much the same thing: it divides an input string into its component **tokens,** which are ordinarily either

- A sequence of consecutive alphanumeric characters (letters or digits), or
- A single-character string consisting of a space or punctuation mark

For example, if the scanner were initialized to extract tokens from the string

```
This line contains 10 tokens.
```

successive calls to the scanner package would return those ten individual tokens as shown by the boxes on the following line:

$\boxed{\texttt{This}}\ \boxed{}\ \boxed{\texttt{line}}\ \boxed{}\ \boxed{\texttt{contains}}\ \boxed{}\ \boxed{\texttt{10}}\ \boxed{}\ \boxed{\texttt{tokens}}\ \boxed{\texttt{.}}$

Using the `Scanner` class

When you want to use the **`Scanner`** class, you typically go through the following steps:

1. *Create a new scanner object.* As with the abstract data types introduced earlier in this chapter, your first responsibility when using a scanner is to declare a new scanner object, like this:

   ```
   Scanner scanner;
   ```

   The scanner object, which is called **`scanner`** in this example, keeps track of all the state information it needs to know the order in which tokens should be delivered. The methods in the **`Scanner`** class all operate on a particular scanner object, which means that you must always specify **`scanner`** as a receiver in any calls that you make.

2. *Initialize the input to be scanned.* Once you have a scanner instance, you can then initialize your scanner input by calling

   ```
   scanner.setInput(str);
   ```

   where **`str`** is the string from which the tokens are scanned. Alternatively, if you want to read tokens from a file, you can call call

   ```
   scanner.setInput(infile);
   ```

   where **`infile`** is an **`ifstream`** object, as described in section 3.4.

3. *Read tokens from the scanner.* Once you have initialized the scanner input, you can process each of its tokens individually by calling **`scanner.nextToken()`** for each token in the input. When all the tokens have been consumed, the method **`scanner.hasMoreTokens()`** returns **`false`**, which means that the standard idiom for iterating through each token in turn looks like this:

   ```
   while (scanner.hasMoreTokens()) {
       string token = scanner.nextToken();
       Do something with the token you've found.
   }
   ```

Because it is often easier to check for a sentinel value, the **nextToken** method returns the empty string if you call it after the last token has been read.

You can use the same scanner instance many times by calling **setInput** for each string you want to split into tokens, so you don't need to declare a separate scanner for each string that you have.

Figure 4-8 offers a simple example of how to use the scanner to create a program that reports all words in a text file that aren't in the English lexicon.

Figure 4-8 Program to check spelling in a text file

```
/*
 * File: spellcheck.cpp
 * --------------------
 * This program checks the spelling of words in an input file.
 */

#include "genlib.h"
#include "simpio.h"
#include "lexicon.h"
#include "scanner.h"
#include <string>
#include <cctype>
#include <iostream>
#include <fstream>

/* Function prototypes */

bool IsAllAlpha(string & str);
void AskUserForInputFile(string prompt, ifstream & infile); (see page 131)

/* Main program */

int main() {
   ifstream infile;
   Lexicon english("EnglishWords.dat");
   Scanner scanner;
   AskUserForInputFile("Input file: ", infile);
   scanner.setInput(infile);
   while (scanner.hasMoreTokens()) {
      string word = scanner.nextToken();
      if (IsAllAlpha(word) && !english.containsWord(word)) {
         cout << word << " is not in the dictionary" << endl;
      }
   }
   infile.close();
   return 0;
}

/* Returns true if a string contains only alphabetic characters. */

bool IsAllAlpha(string & str) {
   for (int i = 0; i < str.length(); i++) {
      if (!isalpha(str[i])) return false;
   }
   return true;
}
```

Setting scanner options

As you can see from Table 4-7, most of the methods in the **Scanner** class allow clients to redefine what character sequences count as individual tokens. By default, the scanner recognizes only two classes of tokens: sequences of alphanumeric characters and single characters that fall outside the alphanumeric set. Depending on the application, clients may want to change this interpretation. For example, a C++ compiler has to recognize numbers like **3.14159265** and quoted strings like **"hello, world"** as single tokens. A browser must do the same with the tags like **<p>** (start a new paragraph) and **<b>** (switch to a boldface font) that are part of the Hypertext Markup Language (HTML) in which most web pages are written. In many contexts, including both the compiler and the web browser, it is useful to ignore whitespace characters—spaces, tabs, and end-of-line markers—because these characters serve only to separate tokens and have no semantic value in themselves.

These capabilities are incorporated into the **Scanner** class as option settings. You enable a particular option by calling a method with an argument that specified the behavior you want. To enhance readability, the **Scanner** class defines a set of enumerated type constants whose names describe as closely as possible exactly what the option does. For example, if you want to ignore whitespace characters, you can call

```
scanner.setSpaceOption(Scanner::IgnoreSpaces);
```

The names of the constants used to set each option are described in Table 4-7 along with the method to which those constants apply.

The **Scanner** class also exports a method called **saveToken** that comes in handy in a variety of applications. This method solves the problem that arises from the fact that you often don't know that you want to stop reading a sequence of tokens until you've read the token that follows that sequence. Unless your application is prepared to deal with the new token at that point in the code, it is convenient to put that token back in the scanner stream where it can be read again when the program is ready to do so.

4.8 Iterators

The **twoletter.cpp** program introduced in Figure 4-7 earlier in this chapter computes a list of all two-letter words by generating every possible combination of two letters and then looking up each one to see whether that two-letter string appears in the lexicon of English words. Another strategy that accomplishes the same result is to go through every word in the lexicon and display the words whose length is equal to 2. To do so, all you need is some way of stepping through each word in a **Lexicon** object, one at a time.

Stepping through the elements of a collection class is a fundamental operation that each class must provide through its interface. Moreover, if the package of collection classes is well designed, clients should be able to use the same strategy to perform that operation, no matter whether they are cycling through all elements in a vector or a grid, all keys in a map, or all words in a lexicon. In most modern software packages, including the library ADTs used in this book and the Standard Template Library on which those classes are based, the process of cycling through the elements of a collection is provided through a class called an **iterator.** Each abstract collection class in the library—with the exception of **Stack** and **Queue**, for which being able to process the elements out of order would violate the LIFO or FIFO discipline that defines those type—exports its own **Iterator** class, but defines that class so that all iterators behave in the same way. Once you learn how to use an iterator for one class, you can easily transfer that knowledge to any of the other classes.

Table 4-7 Methods exported by the `Scanner` class

`setInput(`*str***`)`**	Sets the scanner input source so that new tokens are taken from the string *str*. Any unread tokens remaining from the preceding call to **`setInput`** are discarded.
`setInput(`*stream***`)`**	Sets the scanner input source so that new tokens are taken from the input stream *stream*. Opening and closing the input stream are the responsibility of the client. As with the string version of this method, any unread tokens from a previous call are discarded.
`hasMoreTokens()`	Returns **`true`** if the scanner contains additional tokens.
`nextToken()`	Reads the next token from the input source (either a string or a file) and returns it as a string. If no more tokens exist, **`nextToken`** returns the empty string.
`saveToken(`*token***`)`**	Stores the specified token in the private state of the scanner so that it will be returned the next time **`nextToken`** is called.

`setSpaceOption(`*option***`)`**
`getSpaceOption()`
	Allows the client to control whether whitespace characters (spaces, tabs, and ends of lines) are returned as single-character tokens or skipped entirely. The *option* parameter must be one of the following constant names:
	`Scanner::PreserveSpaces`
	`Scanner::IgnoreSpaces`

`setNumberOption(`*option***`)`**
`getNumberOption()`
	Allows the client to control whether numeric strings are recognized as single tokens. The *option* parameter must be one of the following constant names:
	`Scanner::ScanNumbersAsLetters`
	`Scanner::ScanNumbersAsIntegers`
	`Scanner::ScanNumbersAsReals`
	By default, digits are considered just like letters; the other options allow the scanner to read an complete number.

`setStringOption(`*option***`)`**
`getStringOption()`
	Allows the client to control whether quoted strings are recognized as single tokens. The *option* parameter must be one of the following constant names:
	`Scanner::ScanQuotesAsPunctuation`
	`Scanner::ScanQuotesAsStrings`
	By default, quotation marks act like any other punctuation mark; if **`ScanQuotesAsStrings`** is in effect, a quoted string is treated as a single token.

`setBracketOption(`*option***`)`**
`getBracketOption()`
	Allows the client to control whether HTML tags enclosed in angle brackets (such as **`<p>`** or **`<font>`**) are recognized as single tokens. The *option* parameter must be one of the following constant names:
	`Scanner::ScanBracketsAsPunctuation`
	`Scanner::ScanBracketsAsTag`
	By default, angle brackets act like any other punctuation mark; if **`ScanBracketsAsTag`** is in effect, the entire tag (including the angle brackets) is treated as a single token.

The standard iterator pattern

The general pattern for using an iterator is illustrated by the following code fragment, which iterates over every word in the now-familiar lexicon of English words:

```
Lexicon::Iterator iter = english.iterator();
while (iter.hasNext()) {
    string word = iter.next();
    code to work with that word
}
```

The **`iterator`** method in the **`Lexicon`** class returns an iterator that provides each word in the lexicon, one at a time. The iterator class itself is defined as a nested subclass within **`Lexicon`**, so its full name is **`Lexicon::Iterator`**. The first line in this example therefore applies the **`iterator`** method to the lexicon stored in the variable **`english`** and then stores the resulting iterator object in the variable **`iter`**, which has been suitably declared with the full name of its type.

Once you have the iterator variable, you then enter a loop that continues as long as the iterator has more elements to process. The **`hasNext`** method returns a Boolean value that indicates whether any additional elements remain, which is exactly what you need for the condition in the **`while`** loop. Inside the loop, the **`next`** method returns the next element in the collection. In this example, calling **`iter.next()`** returns the next word from the English language lexicon, which is then stored in the string variable **`word`**.

The code that needs to go into the body of the loop depends, of course, on what you're trying to do. If, for example, you want to list all two-letter English words using the iterator model, the code to do so will look like this:

```
Lexicon::Iterator iter = english.iterator();
while (iter.hasNext()) {
    string word = iter.next();
    if (word.length() == 2) {
        cout << word << endl;
    }
}
```

The type of value produced by the **`next`** method depends on the class in which the iterator is created. In the **`Map`** and **`Lexicon`** classes, **`next`** always returns a value of type **`string`**. In the **`Array`** and **`Grid`** classes, **`next`** returns a value whose type matches the base type of that collection. Thus, an iterator for an **`Array<int>`** will produce values of type **`int`**, and an iterator for a **`Grid<char>`** will produce values of type **`char`**.

Iteration order

More interesting than the issue of what *type* of value an iterator produces is the question of the *order* in which those values are generated. Each container class defines its own policy about iteration order, which is usually chosen so that the underlying implementation can make iterators efficient. The classes you've already seen make the following guarantees about the order of values:

- The iterator for the **`Vector`** class generates the elements in the order of the index position, so that the element in position 0 comes first, followed by the element in position 1, and so on, up to the end of the vector. The order in which elements are returned by the iterator is therefore the same as the order in which elements are processed by the standard **`for`** loop pattern for iterating through an array:

```
for (int i = 0; i < vec.size(); i++) {
    code to process vec[i]
}
```

- The iterator for the **`Grid`** class steps through the elements of row 0 in order, then the elements of row 1, and so forth. This order is iteration strategy for **`Grid`** is thus analogous to using the following **`for`** loop pattern:

```
for (int row = 0; row < grid.numRows(); row++) {
    for (int col = 0; col < grid.numCols(); col++) {
        code to process grid[row][col]
    }
}
```

  This order, in which the **`row`** subscript appears in the outer loop, is called **row-major order.**

- The iterator for the **`Map`** class makes no guarantees about the order in which the keys are returned. As you will discover in Chapter 12, the most efficient representation for storing keys in a map is incompatible with, for example, keeping them in alphabetical order.
- The iterator for the **`Lexicon`** class always returns words in alphabetical order, with all words converted to lower case. The ability to process words alphabetically is one of the principal advantages of the **`Lexicon`** class.

When you use an iterator, it is important that you do not modify the contents of the collection object over which the iteration is performed, because such changes may invalidate the data structures stored within the iterator. If, for example, you are iterating over the keys in a map, deleting the current key may make it impossible for the iterator to figure out how to get to the next key. The implementations of the iterators used in this text check that the structure has not changed as the iterator proceeds through it, but that may not be the case for iterators that exist in other packages.

A simple iterator example

In the discussion of Pig Latin in section 3.3, the words used to illustrate the rules for forming Pig Latin words were *alley* and *trash*. These words have the interesting property that their Pig Latin forms—*alleyway* and *ashtray*—happen to be other English words. Such words are not all that common; in the lexicon stored in the file **`EnglishWords.dat`**, there are only 27 words with that property out of over 100,000 English words. Given iterators and the **`PigLatin`** function from Figure 3-5, it is easy to write a program that lists them all:

```
int main() {
    cout << "This program finds words that remain words when "
         << "translated to Pig Latin." << endl;
    Lexicon dictionary("EnglishWords.dat");
    Lexicon::Iterator iter = dictionary.iterator();
    while (iter.hasNext()) {
        string word = iter.next();
        string pig = PigLatin(word);
        if (pig != word && dictionary.containsWord(pig)) {
            cout << word << " -> " << pig << endl;
        }
    }
    return 0;
}
```

Computing word frequencies

Computers have revolutionized many fields of academic inquiry, including some in which the use of such modern tools might at first seem surprising. Over the last few decades, computer analysis has become central to resolving questions of disputed authorship. For example, there are plays from the Elizabethan era that might have been written by Shakespeare, even though they are not part of the traditional canon. Conversely, several Shakespearean plays that are attributed to Shakespeare have parts that don't sound like his other works and may have in fact been written by someone else. To resolve such questions, Shakespearean scholars often compute the frequency of particular words that appear in the text and see whether those frequencies match what we expect to find based on an analysis of Shakespeare's known works.

Suppose, for example, that you have a text file containing a passage from Shakespeare, such as the following well-known lines from Act 5 of *Macbeth:*

macbeth.txt

```
Tomorrow, and tomorrow, and tomorrow
Creeps in this petty pace from day to day
```

If you are trying to determine the relative frequency of words in Shakespeare's writing, you need to have a program that counts how many times each word appears in the data file. Thus, given the file **macbeth.txt**, your would like your program to produce something like the following output:

```
and                 2
creeps              1
day                 2
from                1
in                  1
pace                1
petty               1
this                1
to                  1
tomorrow            3
```

The code for the word frequency program appears in Figure 4-9. Given the tools you have at your disposal from the earlier sections in this chapter, the code required to

Figure 4-9 Program to keep track of the frequency of words in a text file

```
/*
 * File: wordfreq.cpp
 * ------------------
 * This program computes the frequency of words in a text file.
 */

#include "genlib.h"
#include "simpio.h"
#include "map.h"
#include "scanner.h"
#include <string>
#include <cctype>
#include <iostream>
#include <fstream>
#include <iomanip>
```

```
/* Private function prototypes */

void CreateFrequencyTable(ifstream & infile, Map<int> & wordCounts);
void DisplayWordCounts(Map<int> & wordCounts);
void AskUserForInputFile(string prompt, ifstream & infile); (see page 131)
bool IsAllAlpha(string & str);                              (see page 157)

/* Main program */

int main() {
   ifstream infile;
   Map<int> wordCounts;
   AskUserForInputFile(infile);
   CreateFrequencyTable(infile, wordCounts);
   infile.close();
   DisplayWordCounts(wordCounts);
   return 0;
}

/*
 * Creates a frequency table that reads through the input file
 * and counts how often each word appears.  The client supplies
 * both the input file stream and the map used to keep track of
 * the word count.
 */

void CreateFrequencyTable(ifstream & infile, Map<int> & wordCounts) {
   Scanner scanner;
   scanner.setInput(infile);
   scanner.setSpaceOption(Scanner::IgnoreSpaces);
   while (scanner.hasMoreTokens()) {
      string word = ConvertToLowerCase(scanner.nextToken());
      if (IsAllAlpha(word)) {
         if (wordCounts.containsKey(word)) {
            wordCounts[word]++;
         } else {
            wordCounts[word] = 1;
         }
      }
   }
}

/*
 * Displays the count associated with each word in the frequency
 * table.
 */

void DisplayWordCounts(Map<int> & wordCounts) {
   Map<int>::Iterator iter = wordCounts.iterator();
   while (iter.hasNext()) {
      string word = iter.next();
      cout << left << setw(15) << word
           << right << setw(5) << wordCounts[word] << endl;
   }
}
```

tabulate word frequencies is quite straightforward. The **`Scanner`** class is clearly the right mechanism for going through the words in the file, just as it was for the spelling checker in Figure 4-8. To keep track of the mapping between words and their associated counts, a **`Map<int>`** is precisely what you need. And when you need to go through the entries in the map to list the word counts, **`Iterator`** provides just the right tool.

The only minor problem with this implementation is that the words don't appear in alphabetical order as they did in the proposed sample run created during the design phase. Because the iterator for the **`Map`** class is allowed to produce the keys in any order, they will ordinarily come out in some jumbled fashion. Given the implementation of the **`Map`** class as it exists, the program happens to produce the output

```
pace                1
to                  1
day                 2
tomorrow            3
petty               1
and                 2
creeps              1
from                1
in                  1
this                1
```

but any other order would be equally possible.

It's not hard to get the output to come out in alphabetical order. In fact, as you will have a chance to discover in exercise 16, the library classes in this chapter make it possible to display this list alphabetically with just a few additional lines of code. It might, however, be even more useful to present the list in descending order of frequency. To do that, it will be useful to understand the sorting algorithms presented in Chapter 6.

Summary

This chapter introduced seven C++ classes—**`Vector`**, **`Grid`**, **`Stack`**, **`Queue`**, **`Map`**, **`Lexicon`**, and **`Scanner`**—that form a powerful collection of programming tools. For the moment, you have looked at these classes only as a client. In subsequent chapters, you will have a chance to learn more about how they are implemented. Important points in this chapter include:

- Data structures that are defined in terms of their behavior rather their representation are called *abstract data types*. Abstract data types have several important advantages over more primitive data structures such as arrays and records. These advantages include:
    1. *Simplicity*. The representation of the underlying data representation is not accessible, which means that there are fewer details for the client to understand.
    2. *Flexibility*. The implementer is free to enhance the underlying representation as long as the methods in the interface continue to behave in the same way.
    3. *Security*. The interface barrier prevents the client from making unexpected changes in the internal structure.
- Classes that contain other objects as elements of an integral collection are called *container classes* or, equivalently, *collection classes*. In C++, container classes are usually defined using a *template* or *parameterized type,* in which the type name of the element appears in angle brackets after the name of the container class. For example, the class **`Vector<int>`** signifies a vector containing values of type **`int`**.

- The **`Vector`** class is an abstract data type that behaves in much the same fashion as a one-dimensional array but is much more powerful. Unlike arrays, a **`Vector`** can grow dynamically as elements are added and removed. They are also more secure, because the implementation of **`Vector`** checks to make sure that selected elements exist.
- The **`Grid`** class provides a convenient abstraction for working with two-dimensional arrays.
- The **`Stack`** class represents a collection of objects whose behavior is defined by the property that items are removed from a stack in the opposite order from which they were added: last in, first out (LIFO). The fundamental operations on a stack are **`push`**, which adds a value to the stack, and **`pop`**, which removes and returns the value most recently pushed.
- The **`Queue`** class is similar to the **`Stack`** class except for the fact that elements are removed from a queue in the same order in which they were added: first in, first out (FIFO). The fundamental operations on a queue are **`enqueue`**, which adds a value to the end of a queue, and **`dequeue`**, which removes and returns the value from the front.
- The **`Map`** class makes it possible to associate *keys* with *values* in a way that makes it possible to retrieve those associations very efficiently. The fundamental operations on a map are **`put`**, which adds a key/value pair, and **`get`**, which returns the value associated with a particular key.
- The **`Lexicon`** class represents a word list. The fundamental operations on a map are **`add`**, which stores a new word in the list, and **`containsWord`**, which checks to see whether a word exists in the lexicon.
- The **`Scanner`** class simplifies the problem of breaking up a string or an input file into *tokens* that have meaning as a unit. The fundamental operations on an scanner are **`hasMoreTokens`**, which determines whether more tokens can be read from the scanner, and **`nextToken`**, which returns the next token from the input source.
- Most collection classes define an internal class named **`Iterator`** that makes it easy to cycle through the contents of the collection. The fundamental operations on an iterator are **`hasNext`**, which determines whether more elements exist, and **`next`**, which returns the next element from the collection.

Review questions

1. True or false: An abstract data type is one defined in terms of its behavior rather than its representation.

2. What three advantages does this chapter cite for separating the behavior of a class from its underlying implementation?

3. What is the STL?

4. If you want to use the **`Vector`** class in a program, what **`#include`** line do you need to add to the beginning of your code?

5. List at least three advantages of the **`Vector`** class over the more primitive array mechanism available in C++.

6. What is meant by the term *bounds-checking?*

7. What type name would you use to store a vector of Boolean values?

8. True or false: The default constructor for the **`Vector`** class creates a vector with ten elements, although you can make it longer later.

9. What method do you call to determine the number of elements in a **Vector**?
10. If a **Vector** object has *N* elements, what is the legal range of values for the first argument to **insertAt**? What about for the first argument to **removeAt**?
11. What feature of the **Vector** class makes it possible to avoid explicit use of the **getAt** and **setAt** methods?
12. Why is it important to pass vectors and other collection object by reference?
13. What declaration would you use to initialize a variable called **chessboard** to an 8 x 8 grid, each of whose elements is a character?
14. Given the **chessboard** variable from the preceding exercise, how would you assign the character **'R'** (which stands for a white rook in standard chess notation) to the squares in the lower left and lower right corners of the board?
15. What do the acronyms *LIFO* and *FIFO* stand for? How do these terms apply to stacks and queues?
16. What are the names of the two fundamental operations for a stack?
17. What are the names for the corresponding operations for a queue?
18. What does the **peek** operation do in each of the **Stack** and **Queue** classes?
19. What are the names for the corresponding operations for a queue?
20. Describe in your own words what is meant by the term *discrete time* in the context of a simulation program.
21. True or false: In the **Map** class used in this book, the keys are always strings.
22. True or false: In the **Map** class used in this book, the values are always strings.
23. What happens if you call **get** for a key that doesn't exist in a map?
24. What are the syntactic shorthand forms for **get** and **put** that allow you to treat a map as an associative array?
25. Why do the libraries for this book include a separate **Lexicon** class even though it is easy to implement the fundamental operations of a lexicon using the **Map** class?
26. What are the two kinds of data files supported by the constructor for the **Lexicon** class?
27. What is the purpose of the **Scanner** class?
28. What options are available for controlling the definition of the tokens recognized by the **Scanner** class?
29. What is an *iterator?*
30. What reason is offered for why there is no iterator for the **Stack** and **Queue** class?
31. What is the standard idiom for using an iterator to cycle through the elements of a collection?

32. True or false: The iterator for the **`Map`** class guarantees that individual keys will be delivered in alphabetical order.

33. True or false: The iterator for the **`Lexicon`** class guarantees that individual words will be delivered in alphabetical order.

34. What would happen if you removed the call to **`scanner.setSpaceOption`** in the implementation of **`createFrequencyTable`** in Figure 4-9? Would the program still work?

Programming exercises

1. In Chapter 2, exercise 8, you were askd to write a function **`RemoveZeroElements`** that eliminated any zero-valued elements from an integer array. That operation is much easier in the context of a vector, because the **`Vector`** class makes it possible to add and remove elements dynamically. Rewrite **`RemoveZeroElements`** so that the function header looks like this:

```
void RemoveZeroElements(Vector<int> & vec);
```

2. Write a function

```
bool ReadVector(ifstream & infile, Vector<double> & vec);
```

that reads lines from the data file specified by **`infile`**, each of which consists of a floating-point number, and adds them to the vector **`vec`**. The end of the vector is indicated by a blank line or the end of the file. The function should return **`true`** if it successfully reads the vector of numbers; if it encounters the end of the data file before it reads any values, the function should return **`false`**.

To illustrate the operation of this function, suppose that you have the data file

`SquareAndCubeRoots.txt`

```
1.0000
1.4142
1.7321
2.0000

1.0000
1.2599
1.4422
1.5874
1.7100
1.8171
1.9129
2.0000
```

and that you have opened **`infile`** as an **`ifstream`** on that file. In addition, suppose that you have declares the variable **`roots`** as follows:

```
Vector<double> roots;
```

The first call to **`ReadVector(infile, roots)`** should return **`true`** after initializing **`roots`** so that it contains the four elements shown at the beginning of the file. The second call would also return **`true`** and change the value of **`roots`** to contain the eight elements shown at the bottom of the file. Calling **`ReadVector`** a third time would return **`false`**.

3. Given that it is possible to insert new elements at any point, it is not difficult to keep elements in order as you create a **`Vector`**. Using **`ReadTextFile`** as a starting point, write a function

```
void SortTextFile(ifstream & infile, Vector<string> & lines);
```

that reads the lines from the file into the vector **`lines`**, but keeps the elements of the vector sorted in lexicographic order instead of the order in which they appear in the file. As you read each line, you need to go through the elements of the vector you have already read, find out where this line belongs, and then insert it at that position.

4. The code in Figure 4-2 shows how to check the rows, columns, and diagonals of a tic-tac-toe board using a single helper function. That function, however, is coded in such a way that it only works for 3 x 3 boards. As a first step toward creating a program that can play tic-tac-toe on larger grids, reimplement the **`CheckForWin`** and **`CheckLine`** functions so that they work for square grids of any size.

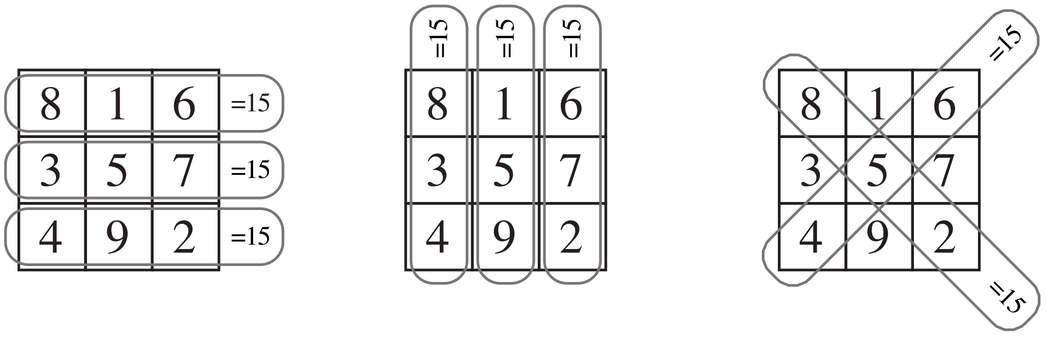
5. A **magic square** is a two-dimensional grid of integers in which the rows, columns, and diagonals all add up to the same value. One of the most famous magic squares appears in the 1514 engraving "Melencolia I" by Albrecht Dürer shown in Figure 4-10, in which a 4 x 4 magic square appears in the upper right, just under the bell. In Dürer's square, which can be read more easily as

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

all four rows, all four columns, and both diagonals add up to 34.

A more familiar example is the following 3 x 3 magic square in which each of the rows, columns, and diagonals add up to 15, as shown:

Figure 4-10 Dürer etching with a magic square



Implement a function

```
bool IsMagicSquare(Grid<int> & square);
```

that tests to see whether the grid contains a magic square. Note that your program should work for square grids of any size. If you call **`IsMagicSquare`** with a grid in which the number of rows and columns are different, it should simply return **`false.`**

6. In the last several years, a new logic puzzle called *Sudoku* has become quite popular throughout the world. In Sudoku, you start with a 9 x 9 grid of integers in which some of the cells have been filled in with digits between 1 and 9. Your job in the puzzle is to fill in each of the empty spaces with a digit between 1 and 9 so that each digit appears exactly once in each row, each column, and each of the smaller 3 x 3 squares. Each Sudoku puzzle is carefully constructed so that there is only one solution. For example, given the puzzle shown on the left of the following diagram, the unique solution is shown on the right:

		2	4		5	8		
	4	1	8				2	
6				7			3	9
2				3			9	6
		9	6		7	1		
1	7			5				3
9	6			8				1
	2				9	5	6	
		8	3		6	9		

3	9	2	4	6	5	8	1	7
7	4	1	8	9	3	6	2	5
6	8	5	2	7	1	4	3	9
2	5	4	1	3	8	7	9	6
8	3	9	6	2	7	1	5	4
1	7	6	9	5	4	2	8	3
9	6	7	5	8	2	3	4	1
4	2	3	7	1	9	5	6	8
5	1	8	3	4	6	9	7	2

Although you won't have learned the algorithmic strategies you need to solve Sudoku puzzles until later in this book, you can easily write a method that checks to see whether a proposed solution follows the Sudoku rules against duplicating values in a row, column, or outlined 3 x 3 square. Write a function

```
bool CheckSudokuSolution(Grid<int> puzzle);
```

that performs this check and returns **`true`** if the **`puzzle`** is a valid solution. Your program should check to make sure that **`puzzle`** contains a 9 x 9 grid of integers and report an error if this is not the case.

7. Write a program that uses a stack to reverse a sequence of integers read in one per line from the console, as shown in the following sample run:

```
Enter a list of integers, ending with 0:
> 10
> 20
> 30
> 40
> 0
Those integers in reverse order are:
  40
  30
  20
  10
```

8. Write a C++ program that checks whether the bracketing operators (parentheses, brackets, and curly braces) in a string are properly matched. As an example of proper matching, consider the string

```
{ s = 2 * (a[2] + 3); x = (1 + (2)); }
```

If you go through the string carefully, you discover that all the bracketing operators are correctly nested, with each open parenthesis matched by a close parenthesis, each open bracket matched by a close bracket, and so on. On the other hand, the following strings are all unbalanced for the reasons indicated:

`(([])`	*The line is missing a close parenthesis.*
`)(`	*The close parenthesis comes before the open parenthesis.*
`{(})`	*The bracketing operators are improperly nested.*

The reason that this exercise fits in this chapter is that one of the simplest strategies for implementing this program is to store the unmatched operators on a stack.

9. Bob Dylan's 1963 song "The Times They Are A-Changin'" contains the following lines, which are themselves paraphrased from Matthew 19:30:

And the first one now
Will later be last
For the times they are a-changin'

In keeping with this revolutionary sentiment, write a function

```
void ReverseQueue(Queue<string> & queue);
```

that reverses the elements in the queue. Remember that you have no access to the internal representation of the queue and will need to come up with an algorithm, presumably involving other structures, that accomplishes the task.

10. The checkout-line simulation in Figure 4-4 can be extended to investigate important practical questions about how waiting lines behave. As a first step, rewrite the simulation so that there are several independent queues, as is usually the case in supermarkets. A customer arriving at the checkout area finds the shortest checkout line and enters that queue. Your revised simulation should calculate the same results as the simulation in the chapter.

11. As a second extension to the checkout-line simulation, change the program from the previous exercise so that there is a single waiting line served by multiple cashiers—a practice that has become more common in recent years. In each cycle of the simulation, any cashier that becomes idle serves the next customer in the queue. If you compare the data produced by this exercise and the preceding one, what can you say about the relative advantages of these two strategies?

12. If waiting lines become too long, customers can easily become frustrated and may decide to take their business elsewhere. Simulations may make it possible to reduce the risk of losing customers by allowing managers to determine how many cashiers are required to reduce the average waiting time below a predetermined threshold. Rewrite the checkout-line simulation from exercise 11 so that the program itself determines how many cashiers are needed. To do so, your program must run the complete simulation several times, holding all parameters constant except the number of cashiers. When it finds a staffing level that reduces the average wait to an acceptable level, your program should display the number of cashiers used on that simulation run.

13. Write a program to simulate the following experiment, which was included in the 1957 Disney film, *Our Friend the Atom,* to illustrate the chain reactions involved in nuclear fission. The setting for the experiment is a large cubical box, the bottom of which is completely covered with an array of 625 mousetraps, arranged to form a square grid 25 mousetraps on a side. Each of the mousetraps is initially loaded with two ping-pong balls. At the beginning of the simulation, an additional ping-pong ball is released from the top of the box and falls on one of the mousetraps. That mousetrap springs and shoots its two ping-pong balls into the air. The ping-pong balls bounce around the sides of the box and eventually land on the floor, where they are likely to set off more mousetraps.

    In writing this simulation, you should make the following simplifying assumptions:

    - Every ping-pong ball that falls always lands on a mousetrap, chosen randomly by selecting a random row and column in the grid. If the trap is loaded, its balls are released into the air. If the trap has already been sprung, having a ball fall on it has no effect.
    - Once a ball falls on a mousetrap—whether or not the trap is sprung—that ball stops and takes no further role in the simulation.
    - Balls launched from a mousetrap bounce around the room and land again after a random number of simulation cycles have gone by. That random interval is chosen independently for each ball and is always between one and four cycles.

    Your simulation should run until there are no balls in the air. At that point, your program should report how many time units have elapsed since the beginning, what percentage of the traps have been sprung, and the maximum number of balls in the air at any time in the simulation.

14. In May of 1844, Samuel F. B. Morse sent the message "What hath God wrought!" by telegraph from Washington to Baltimore, heralding the beginning of the age of electronic communication. To make it possible to communicate information using only the presence or absence of a single tone, Morse designed a coding system in which letters and other symbols are represented as coded sequences of short and long tones, traditionally called *dots* and *dashes*. In Morse code, the 26 letters of the alphabet are represented by the following codes:

A	• —	J	• — — —	S	• • •
B	— • • •	K	— • —	T	—
C	— • — •	L	• — • •	U	• • —
D	— • •	M	— —	V	• • • —
E	•	N	— •	W	• — —
F	• • — •	O	— — —	X	— • • —
G	— — •	P	• — — •	Y	— • — —
H	• • • •	Q	— — • —	Z	— — • •
I	• •	R	• — •		

    If you want to convert from letters to Morse code, you can store the strings for each letter in an array with 26 elements; to convert from Morse code to letters, the easiest approach is to use a map.

    Write a program that reads in lines from the user and translates each line either to or from Morse code depending on the first character of the line:

    - If the line starts with a letter, you want to translate it to Morse code. Any characters other than the 26 letters should simply be ignored.

- If the line starts with a period (dot) or a hyphen (dash), it should be read as a series of Morse code characters that you need to translate back to letters. Each sequence of dots and dashes is separated by spaces, but any other characters should be ignored.

The program should end when the user enters a blank line. A sample run of this program (taken from the messages between the Titanic and the Carpathia in 1912) might look like this:

```
Morse code translator
> SOS TITANIC
... --- ... - .. - .- -. .. -.-.
> WE ARE SINKING FAST
.-- . .- .-. . ... .. -. -.- .. -. --. ..-. .- ... -
> .... . .- -.. .. -. --. ..-. --- .-. -.-- --- ..-
HEADING FOR YOU
>
```

15. In Chapter 3, exercise 6, you were asked to write a function **IsPalindrome** that checks whether a word is a *palindrome,* which means that it reads identically forward and backward. Use that function together with the lexicon of English words to print out a list of all words in English that are palindromes.

16. As noted in the chapter, it is actually rather easy to change the **wordfreq.cpp** program from Figure 4-8 so that the words appear in alphabetical order. The only thing you need to do is think creatively about the tools that you already have. Make the necessary modifications to the program to accomplish this change.

17. As noted in section 4.5, a map is often called a *symbol table* when it is used in the context of a programming language, because it is precisely the structure you need to store variables and their values. For example, if you are working in an application in which you need to assign floating-point values to variable names, you could do so using a map declared as follows:

```
Map<double> symbolTable;
```

Write a C++ program that declares such a symbol table and then reads in command lines from the user, which must be in one of the following forms:

- A simple assignment statement of the form

  var **=** *number*

  This statement should store the value represented by the token *number* in the symbol table under the name *var*. Thus, if the user were to enter

  ```
  pi = 3.14159
  ```

  the string **pi** should be assigned a value of 3.14159 in **symbolTable**.

- The name of a variable alone on a line. When your program reads in such a line, it should print out the current value in the symbol table associated with that name. Thus, if **pi** has been defined as shown in the preceding example, the command

  ```
  pi
  ```

  should display the value 3.14159.

- The command **`list`**, which is interpreted by the program as a request to display all variable/value pairs currently stored in the symbol table, not necessarily in any easily discernable order.
- The command **`quit`**, which should exit from the program.

Once you have implemented each of these command forms, your program should be able to produce the following sample run:

```
Symbol Table Test
> pi = 3.14159
> e = 2.71828
> x = 2.00
> pi
3.14159
> e
2.71828
> list
e = 2.71828
x = 2
pi = 3.14159
> x = 42
> list
e = 2.71828
x = 42
pi = 3.14159
> a = 1.5
> list
e = 2.71828
x = 42
pi = 3.14159
a = 1.5
> quit
```

Note that the output of the **`list`** command does not appear in any easily discernable order.

18. Rewrite the RPN calculator from Figure 4-3 so that it uses the **`Scanner`** class to read its input tokens from a single line, as illustrated by the following sample run:

```
RPN Calculator Simulation
> 1 2 3 * +
7
> 50.0 1.5 * 3.8 2.0 / +
76.9
> quit
```

19. Probably because solving problems by computer can generate such intense frustration, computer science courses seem to generate more than their share of plagiarism. In several universities, the situation has gotten so bad that computer science departments have had to develop software to help detect cases of academic misconduct. The usual approach taken in such programs is to compare the structure of two programs, ignoring differences that are easy for students to change, such as the names of variables and procedures.

Consider, for example, the two program fragments shown below, each of which sums the elements in an integer array.

student1.dat

```
int Total(int array[], int n) {
    int i, total;

    total = 0;
    for (i = 0; i < n; i++) {
        total += array[i];
    }
    return total;
}
```

student2.dat

```
int FindSum(int list[], int nList) {
    int i, sum;

    sum = 0;
    for (i = 0; i < nList; i++) {
        sum += list[i];
    }
    return sum;
}
```

The names of the functions and many of the variables are different, but the two programs are otherwise exactly the same. In code samples this short, it is likely that two programs were created independently, but one would start to get suspicious if the structural similarity went on for page after page.

Write a program that uses two instances of the **`Scanner`** class to perform a line-by-line comparison of two input files. The output of the program should be the percentage of lines in the files that "match." Two lines are defined as matching if their corresponding tokens match all the way across. Two tokens match if either of the following is true:

- The tokens are the same string.
- The tokens both begin with a letter.

For example, the tokens **`"sum"`** and **`"total"`** match because both begin with a letter. In the **`student1.dat`** and **`student2.dat`** sample files, every line matches perfectly under this definition, so the program should report that 100 percent of the lines match.