

Programming Abstractions in C++

Eric S. Roberts and Julie Zelenski

This course reader has had an interesting evolutionary history that in some ways mirrors the genesis of the C++ language itself. Just as Bjarne Stroustrup's first version of C++ was implemented on top of a C language base, this reader began its life as Eric Roberts's textbook *Programming Abstractions in C* (Addison-Wesley, 1998). In 2002-03, Julie Zelenski updated it for use with the C++ programming language, which we began using in CS106B and CS106X during that year.

Although the revised text worked fairly well at the outset, CS106B and CS106X have evolved in recent years so that their structure no longer tracks the organization of the book. This year, we're engaged in the process of rewriting the book so that students in these courses can use it as both a tutorial and a reference. As always, that process takes a considerable amount of time, and there are likely to be some problems as we update the reader. At the same time, we're convinced that the material in CS106B and CS106X is tremendously exciting and will be able to carry us through a quarter or two of instability, and we will end up with an even better course in the future.

We want to thank our colleagues at Stanford, several generations of section leaders (with special thanks to Dan Bentley and Keith Schwarz), and so many students over the years—all of whom have helped make it so exciting to teach this wonderful material.

Programming Abstractions in C++

Chapter 1. An Overview of C++ 1

1.1 What is C++? 2

The object-oriented paradigm; The compilation process

1.2 The structure of a C++ program 5

Comments; Library inclusions; Program-level definitions; Function prototypes; The main program; Function definitions

1.3 Variables, values, and types 9

Naming conventions; Local and global variables; The concept of a data type; Integer types; Floating-point types; Text types; Boolean type; Simple input and output

1.4 Expressions 16

Precedence and associativity; Mixing types in an expression; Integer division and the remainder operator; Type casts; The assignment operator; Increment and decrement operators; Boolean operators

1.5 Statements 24

Simple statements; Blocks; The **if** statement; The **switch** statement; The **while** statement; The **for** statement

1.6 Functions 32

Returning results from functions; Function definitions and prototypes; The mechanics of the function-calling process; Passing parameters by reference

Summary 38

Review questions 39

Programming exercises 41

Chapter 2. Data Types in C++ 45

2.1 Enumeration types 46

Internal representation of enumeration types; Scalar types

2.2 Data and memory 49

Bits; bytes; and words; Memory addresses

2.3 Pointers 51

Using addresses as data values; Declaring pointer variables; The fundamental pointer operations

2.4 Arrays 56

Array declaration; Array selection; Effective and allocated sizes; Initialization of arrays; Multidimensional arrays

2.5 Pointers and arrays 64

The relationship between pointers and arrays

2.6 Records 67

Defining a new structure type; Declaring structure variables; Record selection; Initializing records; Pointers to records

2.7 Dynamic allocation 71

Coping with memory limitations; Dynamic arrays; Dynamic records

Summary 74
Review questions 74
Programming exercises 77

Chapter 3. Libraries and Interfaces 85

3.1 The concept of an interface 86

Interfaces and implementations; Packages and abstractions; Principles of good interface design

3.2 A random number interface 89

The structure of the `random.h` interface; Constructing a client program; The ANSI functions for random numbers; The `random.cpp` implementation

3.3 Strings 98

The data type `string`; Operations on the `string` type ; The `strutils.h` interface; An aside about C-style strings

3.4 Standard I/O and file streams 105

Data files; Using file streams in C++; Standard streams; Formatted stream output; Formatted stream input; Single character I/O; Rereading characters from an input file; Line-oriented I/O

3.5 Other ANSI libraries 112

Summary 113
Review questions 113
Programming exercises 116

Chapter 4. Using Abstract Data Types 123

4.1 The `vector` class 125

Specifying the base type of a `vector`; Declaring a new `vector` object; Operations on the `vector` class; Iterating through the elements of a `vector`; Passing a `vector` as a parameter

4.2 The `grid` class 131

4.3 The `stack` class 133

The structure of the `stack` class

4.4 The `queue` class 136

Simulations and models; The waiting-line model; Discrete time; Events in simulated time; Implementing the simulation

4.5 The `map` class 146

The structure of the `map` class; Using maps in an application; Maps as associative arrays

4.6 The `lexicon` class 151

The structure of the `lexicon` class; A simple application of the `lexicon` class; Why are lexicons useful if maps already exist

4.7 The `scanner` class 154

Setting scanner options

4.8 Iterators 156

The standard iterator pattern; Iteration order; A simple iterator example; Computing word frequencies

Summary 163
Review questions 164
Programming exercises 165

Chapter 5. Introduction to recursion 173

5.1 A simple example of recursion 174

5.2 The factorial function 176

The recursive formulation of **fact**; Tracing the recursive process; The recursive leap of faith

5.3 The Fibonacci function 181

Computing terms in the Fibonacci sequence; Gaining confidence in the recursive implementation; Recursion is not to blame

5.4 Other examples of recursion 187

Detecting palindromes; Binary search; Mutual recursion

5.5 Thinking recursively 192

Maintaining a holistic perspective; Avoiding the common pitfalls

Summary 194
Review questions 195
Programming exercises 197

Chapter 6. Recursive procedures 201

6.1 The Tower of Hanoi 202

Framing the problem; Finding a recursive strategy; Validating the strategy; Coding the solution; Tracing the recursive process

6.2 Generating permutations 211

The recursive insight

6.3 Graphical applications of recursion 213

The graphics library; An example from computer art; Fractals

Summary 224
Review questions 225
Programming exercises 226

Chapter 7. Backtracking algorithms 235

7.1 Solving a maze by recursive backtracking 236

The right-hand rule; Finding a recursive approach; Identifying the simple cases; Coding the maze solution algorithm; Convincing yourself that the solution works

7.2 Backtracking and games 245

The game of nim; A generalized program for two-player games; The minimax strategy; Implementing the minimax algorithm; Using the general strategy to solve a specific game

Summary 269
Review questions 270
Programming exercises 271

Chapter 8. Algorithmic analysis 277

8.1 The sorting problem 278

The selection sort algorithm; Empirical measurements of performance; Analyzing the performance of selection sort

8.2 Computational complexity and big-O notation 282

Big-O notation; Standard simplifications of big-O; Predicting computational complexity from code structure; Worst-case versus average-case complexity; A formal definition of big-O

8.3 Recursion to the rescue 288

The power of divide-and-conquer strategies; Merging two vectors; The merge sort algorithm; The computational complexity of merge sort; Comparing N^2 and $N \log N$ performance

8.4 Standard complexity classes 294

8.5 The Quicksort algorithm 296

Partitioning the vector; Analyzing the performance of Quicksort

8.6 Mathematical induction 301

Summary 304

Review questions 305

Programming exercises 307

Chapter 9. Classes and objects 313

9.1 A simple example of a class definition 314

Defining a `Point` class; Implementing methods in a class; Constructors and destructors; The keyword `this`

9.2 Implementing a specialized version of the `stack` class 319

Defining the `CharStack` interface; Representing the stack data; The advantages of object encapsulation; Removing the maximum size limitation; Object copying

9.3 Implementing the `Scanner` class 328

Summary 328

Review questions 334

Programming exercises 335

Chapter 10. Efficiency and Data Representation 339

10.1 The concept of an editor buffer 340

10.2 Defining the buffer abstraction 341

The public interface of the `EditorBuffer` class; Coding the editor application

10.3 Implementing the editor using arrays 345

Defining the private data representation; Implementing the buffer operations; Assessing the computational complexity of the array implementation

10.4 Implementing the editor using stacks 352

Defining the private data representation for the stack-based buffer; Implementing the buffer operations; Comparing computational complexities

10.5 Implementing the editor using linked lists 357

The concept of a linked list; Designing a linked-list data structure; Using a linked list to represent the buffer; Insertion into a linked-list buffer; Deletion in a linked-list buffer; Cursor motion in the linked-list representation; Linked-list idioms; Completing the buffer implementation; Computational complexity of the linked-list buffer; Doubly linked lists; Time-space tradeoffs

Summary 371

Review questions 372

Programming exercises 373

Chapter 11. Linear Structures 381

11.1 Reimplementing stacks as a template class 382

The interface of a class template

11.2 Reimplementing stacks using linked lists 383

11.3 Implementing queues 391

An array-based implementation of queues; Linked-list representation of queues

11.4 Implementing vectors 404

Supporting insertion and deletion at arbitrary index positions; Implementing selection brackets; Implementing iterators

Summary 414

Review questions 415

Programming exercises 416

Chapter 12. Implementing Maps 419

12.1 An array-based implementation of the map interface 420

12.2 The advantage of knowing where to look 427

12.3 Hashing 429

Implementing the hash table strategy; Choosing a hash function; Determining the number of buckets; Using the `typename` keyword

12.4 Functions as data 438

A general plotting function; Declaring pointers to functions and function typedefs; Implementing `plot`; A generic sorting function

12.5 Mapping functions 444

Mapping over entries in a map; Implementing `mapAll`; Passing client information to a callback function; A note on function types and methods

Summary 448

Review questions 449

Programming exercises 450

Chapter 13. Trees 455

13.1 Family trees 456

Terminology used to describe trees; The recursive nature of a tree; Representing family trees in C++

13.2 Binary search trees 459

The underlying motivation for using binary search trees; Finding nodes in a binary search tree; Inserting new nodes in a binary search tree; Tree traversals

13.3 Balanced trees 466

Tree-balancing strategies; Illustrating the AVL idea; Single rotations; Double rotations; Implementing the AVL algorithm

13.4 Defining a general interface for binary search trees 477

Allowing the client to define the node data; Generalizing the types used for keys; Removing nodes; Implementing the binary search tree package; Implementing the `map.h` interface using binary trees; Using the `static` keyword

Summary 488

Review questions 489

Programming exercises 492

Chapter 14. Expression Trees 499

14.1 Overview of the interpreter 500

14.2 Understanding the abstract structure of expressions 505

A recursive definition of expressions; Expression trees

14.3 Class hierarchies and inheritance 509

14.4 Defining an inheritance hierarchy for expressions 510

Defining the interface for the expression subclasses

14.5 Implementing the node classes 518

Implementing the methods

14.6 Parsing an expression 522

Parsing and grammars; Parsing without precedence; Adding precedence to the parser

Summary 528

Review questions 528

Programming exercises 530

Chapter 15. Sets 535

15.1 Sets as a mathematical abstraction 536

Membership; Set operations; Identities on sets

15.2 Designing a set interface 539

Defining the element type; Writing the set interface; Character sets; Using sets to avoid duplication

15.3 Implementing the set class 544

15.4 Enhancing the efficiency of integer sets 548

Characteristic vectors; Packed arrays of bits; Bitwise operators; Implementing characteristic vectors using the bitwise operators; Implementing the high-level set operations; Using a hybrid implementation

Summary 555

Review questions 556

Programming exercises 558

Chapter 16. Graphs 563

16.1 The structure of a graph 564

Directed and undirected graphs; Paths and cycles; Connectivity

16.2 Implementation strategies for graphs 568

Representing connections using an adjacency list; Representing connections using an adjacency matrix; Representing connections using a set of arcs

16.3 Designing a low-level graph abstraction 571

Using the low-level **graph.h** interface

16.4 Graph traversals 575

Depth-first search; Breadth-first search

16.5 Defining a **Graph class 580**

Using classes for graphs, nodes, and arcs; Adopting an intermediate strategy

16.6 Finding minimum paths 589

16.7 An efficient implementation of priority queues 593

Summary 596

Review questions 597

Programming exercises 599

Appendix A. Library Interfaces 607

bst.h	608
cmpfn.h	611
extgraph.h	612
genlib.h	622
graph.h	623
graphics.h	627
grid.h	630
lexicon.h	634
map.h	638
queue.h	642
random.h	644
scanner.h	646
set.h	652
simpio.h	656
sound.h	657
stack.h	658
strutils.h	660
vector.h	662

Index 657

Chapter 1

An Overview of C++

Out of these various experiments come programs. This is our experience: programs do not come out of the minds of one person or two people such as ourselves, but out of day-to-day work.

— Stokely Carmichael and Charles V. Hamilton,
Black Power, 1967

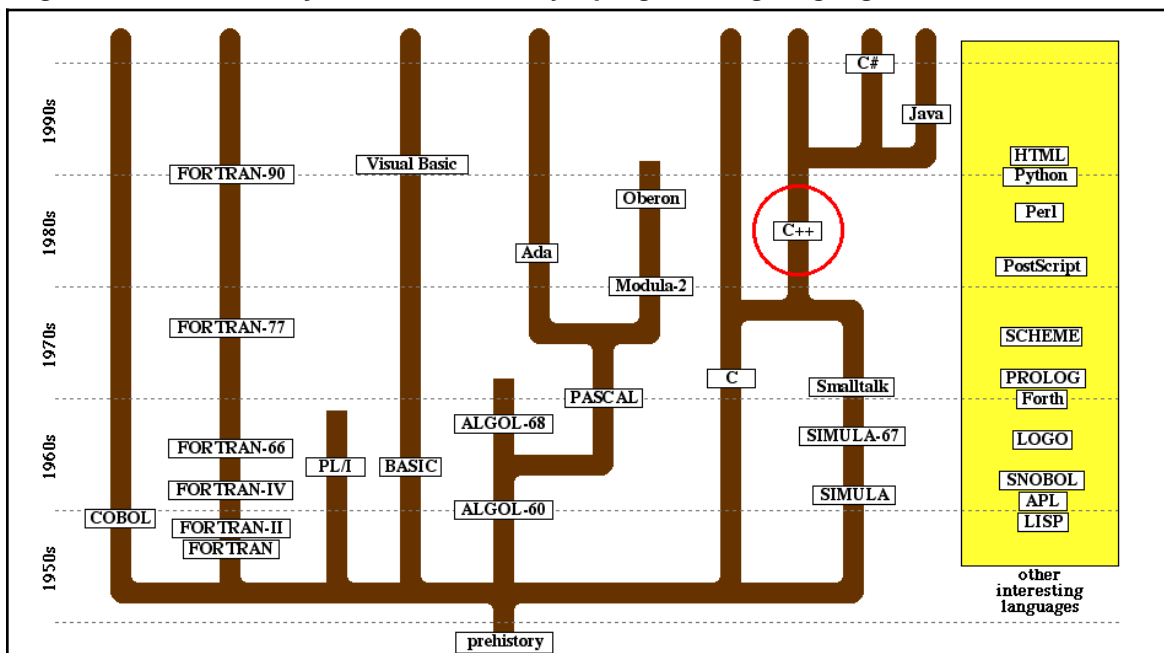
In Lewis Carroll's *Alice's Adventures in Wonderland*, the King asks the White Rabbit to "begin at the beginning and go on till you come to the end: then stop." Good advice, but only if you're starting from the beginning. This book is designed for a second course in computer science and therefore assumes that you have already begun your study of programming. At the same time, because first courses vary considerably in what they cover, it is difficult to rely on any specific material. Some of you, for example, will already have experience programming in C or C++. Many of you, however, are coming from a first course taught in some other language.

Because of this wide disparity in background, the best approach is to adopt the King's advice and begin at the beginning. The first three chapters in this text therefore move quickly through the material I consider to be essential background for the later chapters. Chapters 1 and 2 discuss C++ in general and may be skimmed if you've had experience with C++. Chapter 3 discusses standard interfaces and some interfaces particular to this text. By the end of these three chapters, you will be up to speed on the fundamentals of C++ programming.

1.1 What is C++?

In the early days of computing, programs were written in **machine language**, which consists of the primitive instructions that can be executed directly by the machine. Machine-language programs are difficult to understand, mostly because the structure of machine language reflects the design of the hardware rather than the needs of programmers. In the mid-1950s, a group of programmers under the direction of John Backus at IBM had an idea that profoundly changed the nature of computing. Would it be possible, they wondered, to write programs that resembled the mathematical formulas they were trying to compute and have the computer itself translate those formulas into machine language? In 1955, this team produced the initial version of Fortran (whose name is an abbreviation of *formula translation*), which was the first example of a **higher-level programming language**. Since that time, many new programming languages have been invented, as shown in the evolutionary diagram in Figure 1-1.

Figure 1-1 Evolutionary tree of several major programming languages



As Figure 1-1 illustrates, C++ represents the coming together of two branches in the evolution of programming languages. One of its ancestors is a language called C, which was designed at Bell Laboratories by Dennis Ritchie in 1972 and then later revised and standardized by the American National Standards Institute (ANSI) in 1989. But C++ also descends from another line of languages that have dramatically changed the nature of modern programming.

The object-oriented paradigm

Over the last decade or so, computer science and programming have gone through something of a revolution. Like most revolutions—whether political upheavals or the conceptual restructurings that Thomas Kuhn describes in his 1962 book *The Structure of Scientific Revolutions*—this change has been driven by the emergence of an idea that challenges an existing orthodoxy. Initially, the two ideas compete. For a while, the old order maintains its dominance. Over time, however, the strength and popularity of the new idea grows, until it begins to displace the older idea in what Kuhn calls a **paradigm shift**. In programming, the old order is represented by the **procedural paradigm**, in which programs consist of a collection of procedures and functions that operate on data. The challenger is the **object-oriented paradigm**, in which programs are viewed instead as a collection of data objects that exhibit particular behavior.

The idea of object-oriented programming is not really all that new. The first object-oriented language was SIMULA, a language for coding simulations designed in 1967 by the Scandinavian computer scientists Ole-Johan Dahl, Björn Myhrhaug, and Kristen Nygaard. With a design that was far ahead of its time, SIMULA anticipated many of the concepts that later became commonplace in programming, including the concept of abstract data types and much of the modern object-oriented paradigm. In fact, most of the terminology used to describe object-oriented systems comes from the original 1967 report on SIMULA.

For many years, however, SIMULA mostly just sat on the shelf. Few people paid much attention to it, and the only place you were likely to hear about it would be in a course on programming language design. The first object-oriented language to gain any significant level of recognition within the computing profession was Smalltalk, which was developed at the Xerox Palo Alto Research Center (more commonly known as Xerox PARC) in the late 1970s. The purpose of Smalltalk, which is described in the book *Smalltalk-80: The Language and Its Implementation* by Adele Goldberg and David Robson, was to make programming accessible to a wider audience. As such, Smalltalk was part of a larger effort at Xerox PARC that gave rise to much of the modern user-interface technology that is now standard on personal-computers.

Despite many attractive features and a highly interactive user environment that simplifies the programming process, Smalltalk never achieved much commercial success. The profession as a whole took an interest in object-oriented programming only when the central ideas were incorporated into variants of C, which had already become an industry standard. Although there were several parallel efforts to design an object-oriented language based on C, the most successful was the language C++, which was designed in the early 1980s by Bjarne Stroustrup at AT&T Bell Laboratories. By making it possible to integrate object-oriented techniques with existing C code, C++ enabled large communities of programmers to adopt the object-oriented paradigm in a gradual, evolutionary way.

Although object-oriented languages are undeniably gaining popularity at the expense of procedural ones, it would be a mistake to regard the object-oriented and procedural paradigms as mutually exclusive. Programming paradigms are not so much competitive

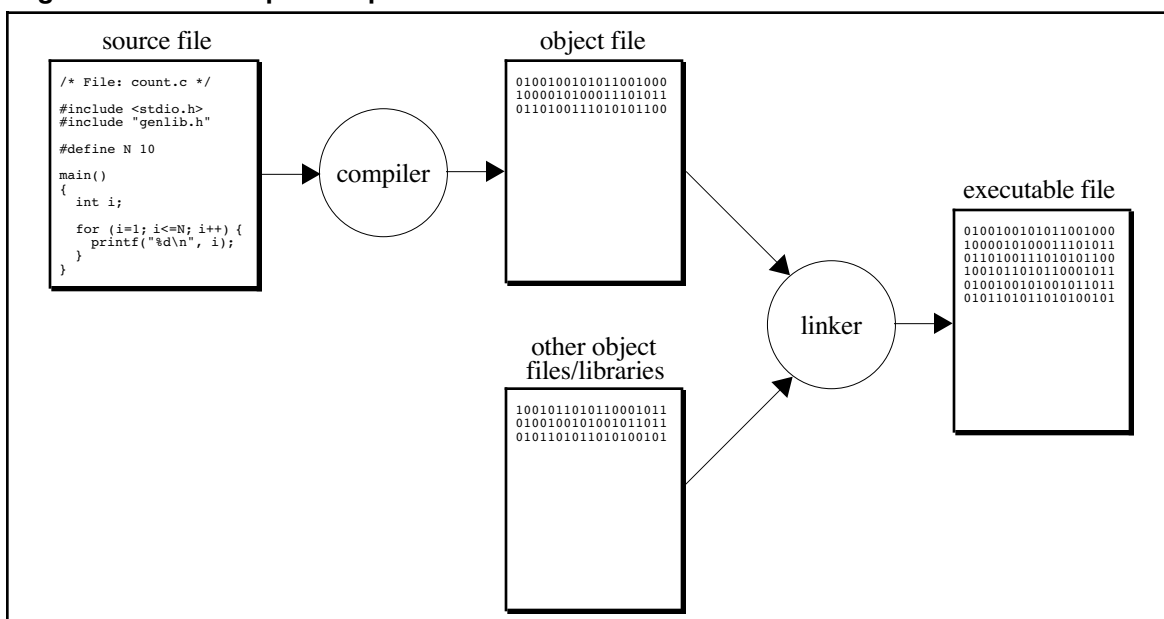
as they are complementary. The object-oriented and the procedural paradigm—along with other important paradigms such as the functional programming style embodied in LISP—all have important applications in practice. Even within the context of a single application, you are likely to find a use for more than one approach. As a programmer, you must master many different paradigms, so that you can use the conceptual model that is most appropriate to the task at hand.

The compilation process

When you write a program in C++, your first step is to create a file that contains the text of the program, which is called a **source file**. Before you can run your program, you need to translate the source file into an executable form. The first step in that process is to invoke a program called a **compiler**, which translates the source file into an **object file** containing the corresponding machine-language instructions. This object file is then combined with other object files to produce an **executable file** that can be run on the system. The other object files typically include predefined object files, called **libraries**, that contain the machine-language instructions for various operations commonly required by programs. The process of combining all the individual object files into an executable file is called **linking**. The process is illustrated by the diagram shown in Figure 1-2.

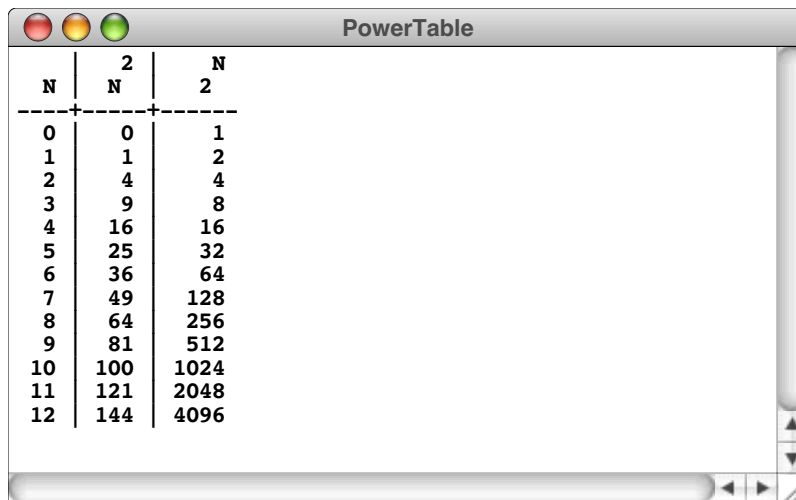
Unfortunately, the specific details of the compilation process vary considerably from one machine to another. There is no way that a general textbook like this can tell you exactly what commands you should use to run a program on your system. Because those commands are different for each system, you need to consult the documentation that comes with the compiler you are using on that machine. The good news, however, is that the C++ programs themselves will look the same. One of the principal advantages of programming in a higher-level language like C++ is that doing so often allows you to ignore the particular characteristics of the hardware and create programs that will run on many different machines.

Figure 1-2 The compilation process



1.2 The structure of a C++ program

The best way to get a feeling for the C++ programming language is to look at a sample program such as the one shown in Figure 1-3. This program generates a table comparing the values of N^2 and 2^N for various values of N —a comparison that will prove to be important in Chapter 8. The output of the program looks like this:



N	N^2	2^N
0	0	1
1	1	2
2	4	4
3	9	8
4	16	16
5	25	32
6	36	64
7	49	128
8	64	256
9	81	512
10	100	1024
11	121	2048
12	144	4096

As the annotations in Figure 1-3 indicate, the `powertab.cpp` program is divided into several components, which are discussed in the next few sections.

Comments

Much of the text in Figure 1-3 consists of English-language comments. A **comment** is text that is ignored by the compiler but which nonetheless conveys information to other programmers. A comment consists of text enclosed between the markers `/*` and `*/` and may continue over several lines. Alternatively, a single-line comment is begun by the marker `//` and continues until the end of the line. The `powertab.cpp` program includes a comment at the beginning that describes the operation of the program as a whole, one before the definition of the `RaiseIntToPower` function that describes what it does, and a couple of one-line comments that act very much like section headings in English text.

Library inclusions

The lines beginning with `#include` such as

```
#include "genlib.h"
#include <iostream>
#include <iomanip>
```

indicate that the compiler should read in definitions from a **header file**. The inclusion of a header file indicates that the program uses facilities from a **library**, which is a collection of prewritten tools that perform a set of useful operations. The different punctuation in these `#include` lines reflects the fact that the libraries come from different sources. The angle brackets are used to specify a system library, such as the standard input/output stream library (`iostream`) or the stream manipulator library (`iomanip`) that is supplied along with C++. The quotation marks are used for private libraries, including the general library (`genlib`), which was designed for use with the programs in this text. Every program in this book will include at least this library most will require other libraries as well and must contain an `#include` line for each one.

Figure 1-3 Sample program powertab.cpp

```

/*
 * File: powertab.cpp
 * -----
 * This program generates a table comparing values
 * of the functions n^2 and 2^n.
 */

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

/*
 * Constants
 * -----
 * LOWER_LIMIT -- Starting value for the table
 * UPPER_LIMIT -- Final value for the table
 */

const int LOWER_LIMIT = 0;
const int UPPER_LIMIT = 12;

/* Private function prototypes */

int RaiseIntToPower(int n, int k);

/* Main program */

int main() {
    cout << "      |      2 |      N" << endl;
    cout << "    N |    N |    2" << endl;
    cout << "----+-----" << endl;
    for (int n = LOWER_LIMIT; n <= UPPER_LIMIT; n++) {
        cout << setw(3) << n << " |" ;
        cout << setw(4) << RaiseIntToPower(n, 2) << " |" ;
        cout << setw(5) << RaiseIntToPower(2, n) << endl;
    }
    return 0;
}

/*
 * Function: RaiseIntToPower
 * Usage: p = RaiseIntToPower(n, k);
 * -----
 * This function returns n to the kth power.
 */

int RaiseIntToPower(int n, int k) {
    int result;

    result = 1;
    for (int i = 0; i < k; i++) {
        result *= n;
    }
    return result;
}

```

program comment

library inclusions

section comment

constant definitions

function prototype

main program

function comment

local variable declarations

statements forming body of function

function definition

Program-level definitions

After the `#include` lines for the libraries, many programs define constants that apply to the program as a whole. In the `powertab.cpp` program, the following lines

```
const int LOWER_LIMIT = 0;
const int UPPER_LIMIT = 12;
```

introduce two constants named `LOWER_LIMIT` and `UPPER_LIMIT`.

The syntax for declaring a constant resembles that for declaring a variable (discussed in section 1.3) that includes the type modifier `const`. The general form is

```
const type name = value;
```

which defines the constant *name* to be of type *type* and initialized to *value*. A constant must be initialized when it is defined and once initialized, it cannot be assigned a new value or changed in any way. Attempting to do so will result in a compiler error. After a named constant is defined, it is available to be used anywhere in the rest of the program. For example, after encountering the line

```
const double PI = 3.14159265;
```

any subsequent use of the name `PI` refers to the constant value `3.14159265`.

Giving symbolic names to constants has several important advantages in terms of programming style. First, the descriptive names give readers of the program a better sense of what the constant value means. Second, centralizing such definitions at the top of the file makes it easier to change the value associated with a name. For example, all you need to do to change the limits used for the table in the `powertab.cpp` program is change the values of the constants. And lastly, a `const` declaration protects from the value from any unintended modification.

In addition to constants, programs often define new data types in this section of the source file, as you will see in Chapter 2.

Function prototypes

Computation in a C++ program is carried out in the context of functions. A **function** is a unit of code that (1) performs a specific operation and (2) is identified by name. The `powertab.cpp` program contains two functions—`main` and `RaiseIntToPower`—which are described in more detail in the next two sections. The line

```
int RaiseIntToPower(int n, int k);
```

is an example of a **function prototype**, a declaration that tells the compiler the information it needs to know about a function to generate the proper code when that function is invoked. This prototype, for example, indicates that the function `RaiseIntToPower` takes two integers as arguments and returns an integer as its result.

You must provide the declaration or definition of each function before making any calls to that function. C++ requires this in order for the compiler to check whether calls to functions are compatible with the corresponding prototypes and can therefore aid you in the process of finding errors in your code.

The main program

Every C++ program must contain a function with the name **main**. This function specifies the starting point for the computation and is called when the program starts up. When **main** has finished its work and returns, execution of the program ends.

The first three statements of the **main** function in the **powertab.cpp** program are sending information to the **cout** stream to display output on the screen. A few useful notes about streams are included in the section on “Simple input and output” later in this chapter and more features are explored in detail in Chapter 3. At this point, you need to have an informal sense of how to display output to understand any programming example that communicates results to the user. In its simplest form, you use the insertion operator **<<** to put information into the output stream **cout**. If you insert a string enclosed in double quotes, it will display that string on the console. You must indicate explicitly that you want to move on to the next line by inserting the stream manipulator **endl**. Thus, the first three lines in **main** display the header for the table.

The rest of the function **main** consists of the following code, which is responsible for displaying the table itself:

```
for (int n = LOWER_LIMIT; n <= UPPER_LIMIT; n++) {
    cout << setw(3) << n << " |" ;
    cout << setw(4) << RaiseIntToPower(n, 2) << " |" ;
    cout << setw(5) << RaiseIntToPower(2, n) << endl;
}
```

This code is an example a **for loop**, which is used to specify repetition. In this case, the **for** statement indicates that the **body** of the loop should be repeated for each of the values of **n** from **LOWER_LIMIT** to **UPPER_LIMIT**. A section on the detailed structure of the **for** loop appears later in the chapter, but the example shown here represents a common idiomatic pattern that you can use to count between any specified limits.

The body of the loop illustrates an important new feature: the ability to include values as part of the output display. Rather than just displaying fixed strings, we can display numeric values and computed results. We can also use stream manipulators to format the output. Let’s examine the first statement in the body of the loop:

```
cout << setw(3) << n << " |" ;
```

This line will display the current value of the variable **n** followed by a string containing a space and a vertical bar. The **setw(3)** that is inserted into the stream just before **n** indicates that the stream should format the next value in a field that is three characters wide. Similarly, the next statement prints the formatted result taken from the expression **RaiseIntToPower(n, 2)**. Obtaining the value of the expression requires making a call on the **RaiseIntToPower** function, which is discussed in the following section. The value that **RaiseIntToPower** returns is displayed as part of the output. So is the result of the call to **RaiseIntToPower(2, n)**, which supplies the value for the third column in the table.

The last statement in **main** is

```
return 0;
```

which indicates that the function result is **0**. The return value from the **main** function is used to communicate the success or failure of the entire program. By convention, a result of 0 indicates success.

Function definitions

Because large programs are difficult to understand in their entirety, most programs are broken down into several smaller functions, each of which is easier to understand. In the `powertab.cpp` program, the function `RaiseIntToPower` is responsible for raising an integer to a power—an operation that is not built into C++ and must therefore be defined explicitly.

The first line of `RaiseIntToPower` is the variable declaration

```
int result;
```

which introduces a new variable named `result` capable of holding values of type `int`, the standard type used to represent integers. The syntax of variable declarations is discussed in more detail in the section on “Variables, values, and types” later in this chapter. For now, all you need to know is that this declaration creates space for an integer variable that you can then use in the body of the function.

The next line in the body of `RaiseIntToPower` is

```
result = 1;
```

This statement is a simple example of an **assignment statement**, which sets the variable on the left of the equal sign to the value of the expression on the right. In this case, the statement sets the variable `result` to the constant 1. The next statement in the function is a **for** loop that executes its body `k` times. The repeated code consists of the line

```
result *= n;
```

which is a C++ shorthand for the English sentence “Multiply `result` by `n`.” Because the function initializes the value of `result` to 1 and then multiplies `result` by `n` a total of `k` times, the variable `result` ends up with the value n^k .

The last statement in `RaiseIntToPower` is

```
return result;
```

which indicates that the function should return `result` as the value of the function.

1.3 Variables, values, and types

One of the fundamental characteristics of programs is that they manipulate data. To do so, programs must be able to store data as part of their operation. Moreover, programs today work with many different kinds of data, including numbers and text, along with many more sophisticated data structures, such as those introduced in Part 3 of this book. Learning how to store data of various types is an important part of mastering the basics of any language, including C++.

Variables

Data values in a program are usually stored in variables. In C++, if you want to use a variable to hold some information, you must **declare** that variable before you use it. Declaring a variable establishes the following properties:

- *Name.* Every variable has a name, which is formed according to the rules described in the section entitled “Naming conventions” later in this chapter. You use the name in the program to refer to the variable and the value it contains.

- *Type*. Each variable in a C++ program is constrained to hold values of a particular data type. C++ includes several predefined types and also allows you to define new types of your own, as discussed in Chapter 2.
- *Lifetime*. Depending on how they are declared, some variables persist throughout the entire program, while others are created and destroyed dynamically as the program moves through various levels of function call.
- *Scope*. The declaration of a variable also controls what parts of the program have access to the variable, which is called its **scope**.

The standard syntax for declaring a variable is

```
type namelist;
```

where *type* indicates the data type and *namelist* is a list of variable names separated by commas. For example, the function **RaiseIntToPower** in the **powertab.cpp** program contains the line

```
int result;
```

which declares the variable **result** to be of type **int**.

In C++, the initial contents of a variable are undefined. If you want a variable to have a particular initial value, you need to initialize it explicitly. One approach is to use an assignment statement in the body of the function to assign a value to each variable before you use it. You can, however, include initial values directly in a declaration by writing an equal sign and a value after a variable name. Thus, the declaration

```
int result = 0;
```

is a shorthand for the following code, in which the declaration and assignment are separate:

```
int result;  
result = 0;
```

An initial value specified as part of a declaration is called an **initializer**.

Naming conventions

The names used for variables, functions, types, constants, and so forth are collectively known as **identifiers**. In C++, the rules for identifier formation are

1. The name must start with a letter or the underscore character (`_`).
2. All other characters in the name must be letters, digits, or the underscore. No spaces or other special characters are permitted in names.
3. The name must not be one of the reserved keywords listed in Table 1-1

Uppercase and lowercase letters appearing in an identifier are considered to be different. Thus, the name **ABC** is not the same as the name **abc**. Identifiers can be of any length, but C++ compilers are not required to consider any more than the first 31 characters in determining whether two names are identical. Implementations may impose additional restrictions on identifiers that are shared between modules.

You can improve your programming style by adopting conventions for identifiers that help readers identify their function. In this text, names of variables and data types begin

Table 1-1 C++ reserved keywords

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

with a lowercase letter, such as **n1**, **total**, or **string**. By contrast, function names, such as **RaiseIntToPower**, usually begin with an uppercase letter. Moreover, whenever a name consists of several English words run together, the first letter in each word after the first is capitalized to make the name easier to read. By tradition, constant names, such as **LOWER_LIMIT** are written entirely in uppercase, with underscores between the words.

Local and global variables

Most variables are declared with the body of a function. Such variables are called **local variables**. The scope of a local variable extends to the end of the block in which it is declared. The lifetime of a local variable is the time during which that function is active. When the function is called, space for each local variable is allocated for the duration of that function call. When the function returns, all its local variables disappear.

If a variable declaration appears outside any function definition, that declaration introduces a **global variable**. The scope of a global variable is the remainder of the file in which it is declared. Its lifetime continues throughout the entire execution of a program. Global variables are therefore able to store values that persist across function calls. Although they have important applications, global variables can easily be overused. Because global variables can be manipulated by many different functions, it is harder to keep those functions from interfering with each other. Because of these dangers, global variables are used infrequently in this text.

The concept of a data type

One of the reasons C++ requires all variables to be declared is that doing so constrains their contents to values of a particular data type. From a formal perspective, a data type is defined by two properties: a *domain*, which is the set of values that belong to that type, and a *set of operations*, which defines the behavior of that type. For example, the domain of the type **int** includes all integers ($\dots -2, -1, 0, 1, 2 \dots$) up to the limits established by the hardware of the machine. The set of operations applicable to values of type **int** includes, for example, the standard arithmetic operations like addition and multiplication. Other types have a different domain and set of operations.

As you will learn in Chapter 2, much of the power of higher-level languages like C++ comes from the fact that you can define new data types from existing ones. To get that process started, C++ includes several fundamental types that are defined as part of the language (or, in the case of **string**, as part of the standard C++ libraries). These types, which act as the building blocks for the type system as a whole, are called **atomic types**. These predefined types are grouped into four categories—integer, floating-point, text, and Boolean—which are discussed in the sections that follow.

Integer types

Although the concept of an integer seems like a simple one, C++ actually includes several different data types for representing integer values. In most cases, all you need to know is the type **int**, which corresponds to the standard representation of an integer on the computer system you are using. In certain cases, however, you need to be more careful. Like all data, values of type **int** are stored internally in storage units that have a limited capacity. Those values therefore have a maximum size, which limits the range of integers you can use. To get around this problem, C++ defines three integer types—**short**, **int**, and **long**—distinguished from each other by the size of their domains.

Unfortunately, the language definition for C++ does not specify an exact range for these three types. As a result, the range for the different integer types depends on the machine and the compiler you're using. On many personal computers, the maximum value of type **int** is 32,767, which is rather small by computational standards. If you wanted, for example, to perform a calculation involving the number of seconds in a year, you could not use type **int** on those machines, because that value (31,536,000) is considerably larger than the largest available value of type **int**. The only properties you can rely on are the following:

- The internal size of an integer cannot decrease as you move from **short** to **int** to **long**. A compiler designer for C++ could, for example, decide to make **short** and **int** the same size but could not make **int** smaller than **short**.
- The maximum value of type **int** must be at least $32,767 (2^{15}-1)$.
- The maximum value of type **long** must be at least $2,147,483,647 (2^{31}-1)$.

The designers of C++ could have chosen to define the allowable range of type **int** more precisely. For example, they could have declared—as the designers of Java did—that the maximum value of type **int** would be $2^{31}-1$ on every machine. Had they done so, it would be easier to move a program from one system to another and have it behave in the same way. The ability to move a program between different machines is called **portability**, which is an important consideration in the design of a programming language.

In C++, each of the integer types **int**, **long**, and **short** may be preceded by the keyword **unsigned**. Adding **unsigned** creates a new data type in which only nonnegative values are allowed. Because unsigned variables do not need to represent negative values, declaring a variable to be one of the unsigned types allows it to hold twice as many positive values. For example, if the maximum value of type **int** is 32,767, the maximum value of type **unsigned int** will be 65,535. C++ allows the type **unsigned int** to be abbreviated to **unsigned**, and most programmers who use this type tend to follow this practice. Sometimes variables intended to store sizes are declared as unsigned, because a size will always be nonnegative.

An integer constant is ordinarily written as a string of digits representing a number in base 10. If the number begins with the digit 0, however, the compiler interprets the value as an octal (base 8) integer. Thus, the constant **040** is taken to be in octal and represents the decimal number 32. If you prefix a numeric constant with the characters **0x**, the compiler interprets that number as hexadecimal (base 16). Thus, the constant **0xFF** is equivalent to the decimal constant 255. You can explicitly indicate that an integer constant is of type **long** by adding the letter **L** at the end of the digit string. Thus, the constant **0L** is equal to 0, but the value is explicitly of type **long**. Similarly, if you use the letter **U** as a suffix, the constant is taken to be unsigned.

Floating-point types

Numbers that include a decimal fraction are called **floating-point numbers**, which are used to approximate real numbers in mathematics. As with integers, C++ defines three different floating-point types: **float**, **double**, and **long double**. Although ANSI C++ does not specify the exact representation of these types, the way to think about the difference is that types that appear later in the list allow numbers to be represented with greater precision but require more memory space. Unless you are doing exacting scientific calculation, the differences between these types will not matter a great deal. In keeping with a common convention among C++ programmers, this text uses the type **double** as its standard floating-point type.

Floating-point constants in C++ are written with a decimal point. Thus, if **2.0** appears in a program, the number is represented internally as a floating-point value if the programmer had written **2**, this value would be an integer. Floating-point values can also be written in a special programmer's style of scientific notation, in which the value is represented as a floating-point number multiplied by a integral power of 10. To write a number using this style, you write a floating-point number in standard notation, followed immediately by the letter **E** and an integer exponent, optionally preceded by a + or - sign. For example, the speed of light in meters per second can be written in C++ as

2.9979E+8

where the **E** stands for the words *times 10 to the power*.

Text types

In the early days, computers were designed to work only with numeric data and were sometimes called *number crunchers* as a result. Modern computers, however, work less with numeric data than they do with text data, that is, any information composed of individual characters that appear on the keyboard and the screen. The ability of modern computers to process text data has led to the development of word processing systems, on-line reference libraries, electronic mail, and a wide variety of other useful applications.

The most primitive elements of text data are individual characters, which are represented in C++ using the predefined data type **char**. The domain of type **char** is the set of symbols that can be displayed on the screen or typed on the keyboard: the letters, digits, punctuation marks, spacebar, Return key, and so forth. Internally, these values are represented inside the computer by assigning each character a numeric code. In most implementations of C++, the coding system used to represent characters is called **ASCII**, which stands for the *American Standard Code for Information Interchange*. The numeric values of the characters in the ASCII set are shown in Table 1-2.

Although it is important to know that characters are represented internally using a numeric code, it is not generally useful to know what numeric value corresponds to a particular character. When you type the letter A, the hardware logic built into the keyboard automatically translates that character into the ASCII code 65, which is then sent to the computer. Similarly, when the computer sends the ASCII code 65 to the screen, the letter A appears.

You can write a character constant in C++ by enclosing the character in single quotes. Thus, the constant **'A'** represents the internal code of the uppercase letter A. In addition to the standard characters, C++ allows you to write special characters in a two-character form beginning with a backward slash (****). These two-character combinations are called

Table 1-2 ASCII codes

	0	1	2	3	4	5	6	7	8	9
0	\000	\001	\002	\003	\004	\005	\006	\a	\b	\t
10	\n	\v	\f	\r	\016	\017	\020	\021	\022	\023
20	\024	\025	\026	\027	\030	\031	\032	\033	\034	\035
30	\036	\037	<i>space</i>	!	"	#	\$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~	\177		

escape sequences and you can see several listed in the first few rows of Table 1-2. A few of the more commonly used escape sequences are '\n' the newline character, '\t' the tab character, and '\\' the backslash character.

Characters are most useful when they are collected together into sequential units. In programming, a sequence of characters is called a **string**. Strings make it possible to display informational messages on the screen. You have already seen strings in the sample program **powertab.cpp**. It is important, however, to recognize that strings are data and that they can be manipulated and stored in much the same way that numbers can.

The standard C++ library defines a **string** type and operations that manipulate strings. The details of type **string** are not important at this point strings are considered in more detail in Chapter 3. In this chapter, strings are treated as atomic values and used exclusively to specify text that is displayed directly on the display screen.

You write string constants in C++ by enclosing the characters contained within the string in double quotes. C++ supports the same escape sequences for strings as for characters. If two or more string constants appear consecutively in a program, the compiler concatenates them together. The most important implication of this rule is that you can break a long string over several lines so that it doesn't end up running past the right margin of your program.

Boolean type

In the programs you write, it is often necessary to test a particular condition that affects the subsequent behavior of your code. Typically, that condition is specified using an expression whose value is either true or false. This data type—for which the only legal values are true and false—is called **Boolean data**, after the mathematician George Boole, who developed an algebraic approach for working with such values.

In C++, the Boolean type is called **bool** and its domain consists of the values **true** and **false**. You can declare variables of type **bool** and manipulate them in the same way as other data objects.

Simple input and output

Before you can write programs that interact with the user, you need to have some way of accepting input data from the user and displaying results on the screen. In C++, none of this functionality is provided directly within the language. Instead, all input and output operations—which are often referred to collectively as **I/O operations**—are performed by calling functions provided as part of a library.

Unfortunately, the standard library functions for reading input data from the user, which are described in Chapter 3, can be a bit complicated in their operation and provide more power than you need at this point. For most of the programs in this text, it is more convenient to use functions defined in a simplified I/O library that I designed to make it easier for beginning students to learn the important concepts of programming without getting bogged down in extraneous details. To use this library, you need to add the following line to the library-inclusion section at the beginning of your program:

```
#include "simpio.h"
```

The **simpio** library defines the functions **GetInteger**, **GetLong**, **GetReal**, and **GetLine**, which wait for the user to enter a line at the keyboard and then return a value of type **int**, **long**, **double**, and **string**, respectively. To let the user know what value is expected, it is conventional to display a message to the user, which is called a **prompt**, before calling the input function. Thus, if you need to request a value from the user for the integer variable **n**, you would typically use a pair of statements like this:

```
cout << "Enter an integer: ";
n = GetInteger();
```

Output operations in this book use the insertion operator **<<**. The operand on the left of the operator is a stream, such as the standard output stream **cout**. The operand on the right is the data that you wish to insert into the stream. Several insertions to the same stream can be chained together as shown here:

```
cout << "The result is " << val << endl;
```

Stream manipulators can be used to control the formatting of the output. A manipulator is inserted into the stream ahead of the value it affects. The manipulator does not print anything to the stream, but changes the state of the stream such that subsequent insertions will use the requested formatting. A few of the more common stream manipulators are shown in Table 1-3. To use these manipulators, you must include the **<iomanip>** interface file in the library-inclusion section of your program.

As an example of the use of the simple I/O facilities, the following main program reads in three floating-point values and displays their average:

```
int main() {
    cout << "This program averages three numbers." << endl;
    cout << "1st number: ";
    double n1 = GetReal();
    cout << "2nd number: ";
    double n2 = GetReal();
    cout << "3rd number: ";
    double n3 = GetReal();
    double average = (n1 + n2 + n3) / 3;
    cout << "The average is " << average << endl;
    return 0;
}
```

Table 1-3 Common output stream manipulators

setw(<i>n</i>)	Sets the field width to <i>n</i> characters. If the value to be formatted is too short to fill the entire field width, extra space is added. This stream property is transient it affects only the next value inserted into the stream.
setprecision(<i>n</i>)	Sets the precision to <i>n</i> places. The precision indicates how many digits should be displayed to the right of the decimal point. This property is persistent—once set, it applies to all subsequent values inserted into the stream.
left	Sets the justification. For values that are smaller than their field width, left justification will cause the values to line up along the left, padding with spaces on the right. This property is persistent.
right	Same as left, but to the right, padding with spaces on the left.

1.4 Expressions

Whenever you want a program to perform calculations, you need to write an expression that specifies the necessary operations in a form similar to that used for expressions in mathematics. For example, suppose that you wanted to solve the quadratic equation

$$ax^2 + bx + c = 0$$

As you know from high-school mathematics, this equation has two solutions given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The first solution is obtained by using + in place of the $\pm$ symbol the second is obtained by using – instead. In C++, you could compute the first of these solutions by writing the following expression:

```
(-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

There are a few differences in form—multiplication is represented explicitly by a *, division is represented by a /, and the square root function is spelled out—but the expression nonetheless captures the intent of its mathematical counterpart in a way that is quite readable, particularly if you’ve written programs in any modern programming language.

In C++, an expression is composed of terms and operators. A term, such as the variables **a**, **b**, and **c** or the constants 2 and 4 in the preceding expression, represents a single data value and must be either a constant, a variable, or a function call. An operator is a character (or sometimes a short sequence of characters) that indicates a computational operation. The complete list of operators available in C++ is shown in Table 1-4. The table includes familiar arithmetic operators like + and – along with several others that pertain only to types that are introduced in later chapters.

Precedence and associativity

The point of listing all the operators in a single table is to establish how they relate to one another in terms of **precedence**, which is a measure of how tightly an operator binds to its operands in the absence of parentheses. If two operators compete for the same

Table 1-4 Complete precedence table for C++ operators

() [] -> .	<i>left-associative</i>
<i>unary operators:</i> - ++ -- ! & * ~ (type) sizeof	<i>right-associative</i>
* / %	<i>left-associative</i>
+ -	<i>left-associative</i>
<< >>	<i>left-associative</i>
< <= > >=	<i>left-associative</i>
== !=	<i>left-associative</i>
&	<i>left-associative</i>
^	<i>left-associative</i>
	<i>left-associative</i>
&&	<i>left-associative</i>
	<i>left-associative</i>
? :	<i>right-associative</i>
= op=	<i>right-associative</i>
,	<i>left-associative</i>

operand, the one that appears higher in the precedence table is applied first. Thus, in the expression

```
(-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

the multiplications (**b * b** and **4 * a * c**) are performed before the subtraction because * has a higher precedence than -. It is, however, important to note that the - operator occurs in two forms. When it is written between two operands, it is a **binary operator** representing subtraction. When it is written in front of a single operand, as in is **-b**, it is a **unary operator** representing negation. The precedence of the unary and binary versions of an operator are different and are listed separately in the precedence table.

If two operators have the same precedence, they are applied in the order specified by their **associativity**, which indicates whether that operator groups to the left or to the right. Most operators in C++ are **left-associative**, which means that the leftmost operator is evaluated first. A few operators—primarily the assignment operator, which is discussed in more detail later in this chapter—are **right-associative**, which mean that they are evaluated from right to left. The associativity for each operator appears in Table 1-4.

The quadratic formula illustrates the importance of paying attention to associativity rules. Consider what would happen if you wrote the expression without the parentheses around **2 * a**, as follows:

```
(-b + sqrt(b * b - 4 * a * c)) / 2 * a
```



Should be **(2 * a)**

Without the parentheses, the division operator would be performed first because / and * have the same precedence and associate to the left.

Mixing types in an expression

In C++, you can write an expression that includes values of different numeric types. If C++ encounters an operator whose operands are of different numeric types, the compiler

automatically converts the operands to a common type by determining which of the two operand types appears closest to the top in Table 1-5. The result of applying the operation is always that of the arguments after any conversions are applied. This convention ensures that the result of the computation is as precise as possible.

As an example, suppose that **n** is declared as an **int**, and **x** is declared as a **double**. The expression

n + 1

is evaluated using integer arithmetic and produces a result of type **int**. The expression

x + 1

however, is evaluated by converting the integer 1 to the floating-point value 1.0 and adding the results together using double-precision floating-point arithmetic, which results in a value of type **double**.

Integer division and the remainder operator

The fact that applying an operator to two integer operands generates an integer result leads to an interesting situation with respect to the division operator. If you write an expression like

9 / 4

C++'s rules specify that the result of this operation must be an integer, because both operands are of type **int**. When C++ evaluates this expression, it divides 9 by 4 and discards any remainder. Thus, the value of this expression in C++ is 2, not 2.25.

If you want to compute the mathematically correct result of 9 divided by 4, at least one of the operands must be a floating-point number. For example, the three expressions

9.0 / 4
9 / 4.0
9.0 / 4.0

each produce the floating-point value 2.25. The decimal fraction is thrown away only if both operands are of type **int**. The operation of discarding a decimal fraction is called **truncation**.

There is an additional arithmetic operator that computes a remainder, which is indicated in C++ by the percent sign (%). The % operator returns the remainder when the first operand is divided by the second, and requires that both operands be of one of the

Table 1-5 Type conversion hierarchy for numeric types

long double double float unsigned long long unsigned int int unsigned short short char	<i>most precise</i>  <i>least precise</i>

integer types. For example, the value of

```
9 % 4
```

is 1, since 4 goes into 9 twice, with 1 left over. The following are some other examples of the % operator:

```
0 % 4 = 0      19 % 4 = 3
1 % 4 = 1      20 % 4 = 0
4 % 4 = 0      2001 % 4 = 1
```

The / and % operators turn out to be extremely useful in a wide variety of programming applications. The % operator, for example, is often used to test whether one number is divisible by another. For example, to determine whether an integer **n** is divisible by 3, you just check whether the result of the expression **n % 3** is 0.

It is, however, important to use caution if either or both of the operands to / and % might be negative, because the results may differ from machine to machine. On most machines, division truncates its result toward 0, but this behavior is not actually guaranteed by the ANSI standard. In general, it is good programming practice to avoid using these operators with negative values.

Type casts

In C++, you can specify explicit conversion by using what is called a **type cast**, a unary operator that consists of the desired type followed by the value you wish to convert in parentheses. For example, if **num** and **den** were declared as integers, you could compute the floating-point quotient by writing

```
quotient = double(num) / den;
```

The first step in evaluating the expression is to convert **num** to a **double**, after which the division is performed using floating-point arithmetic as described in the section on “Mixing types in an expression” earlier in this chapter.

As long as the conversion moves upward in the hierarchy shown in Table 1-5, the conversion causes no loss of information. If, however, you convert a value of a more precise type to a less precise one, some information may be lost. For example, if you use a type cast to convert a value of type **double** to type **int**, any decimal fraction is simply dropped. Thus, the value of the expression

```
int(1.9999)
```

is the integer 1.

The assignment operator

In C++, assignment of values to variables is built into the expression structure. The = operator takes two operands, just like + or *. The left operand must indicate a value that can change, which is typically a variable name. When the assignment operator is executed, the expression on the right-hand side is evaluated, and the resulting value is then stored in the variable that appears on the left-hand side. Thus, if you evaluate an expression like

```
result = 1
```

the effect is that the value 1 is assigned to the variable **result**. In most cases, assignment expressions of this sort appear in the context of simple statements, which are formed by adding a semicolon after the expression, as in the line

```
result = 1;
```

that appears in the **powertab.cpp** program. Such statements are often called **assignment statements**, although they in fact have no special status in the language definition.

The assignment operator converts the type of the value on the right-hand side so that it matches the declared type of the variable. Thus, if the variable **total** is declared to be of type **double**, and you write the assignment statement

```
total = 0;
```

the integer 0 is converted into a **double** as part of making the assignment. If **n** is declared to be of type **int**, the assignment

```
n = 3.14159265;
```

has the effect of setting **n** to 3, because the value is truncated to fit in the integer variable.

Even though assignment operators usually occur in the context of simple statements, they can also be incorporated into larger expressions, in which case the result of applying the assignment operator is simply the value assigned. For example, the expression

```
z = (x = 6) + (y = 7)
```

has the effect of setting **x** to 6, **y** to 7, and **z** to 13. The parentheses are required in this example because the **=** operator has a lower precedence than **+**. Assignments that are written as part of larger expressions are called **embedded assignments**.

Although there are contexts in which embedded assignments are extremely convenient, they often make programs more difficult to read because the assignment is easily overlooked in the middle of a complex expression. For this reason, this text limits the use of embedded assignments to a few special circumstances in which they seem to make the most sense. Of these, the most important is when you want to set several variables to the same value. C++'s definition of assignment as an operator makes it possible, instead of writing separate assignment statements, to write a single statement like

```
n1 = n2 = n3 = 0;
```

which has the effect of setting all three variables to 0. This statement works because C++ evaluates assignment operators from right to left. The entire statement is therefore equivalent to

```
n1 = (n2 = (n3 = 0));
```

The expression **n3 = 0** is evaluated, which sets **n3** to 0 and then passes 0 along as the value of the assignment expression. That value is assigned to **n2**, and the result is then assigned to **n1**. Statements of this sort are called **multiple assignments**.

As a programming convenience, C++ allows you to combine assignment with a binary operator to produce a form called a **shorthand assignment**. For any binary operator *op*, the statement

```
variable op= expression;
```

is equivalent to

```
variable = variable op (expression) ;
```

where the parentheses are required only if the expression contains an operator whose precedence is lower than that of *op*. Thus, the statement

```
balance += deposit;
```

is a shorthand for

```
balance = balance + deposit;
```

which adds **deposit** to **balance**.

Because this same shorthand applies to any binary operator in C++, you can subtract the value of **surcharge** from **balance** by writing

```
balance -= surcharge;
```

Similarly, you can divide the value of **x** by 10 using

```
x /= 10;
```

or double the value of **salary** by using

```
salary *= 2;
```

Increment and decrement operators

Beyond the shorthand assignment operators, C++ offers a further level of abbreviation for two particularly common programming operations—adding or subtracting 1 from a variable. Adding 1 to a variable is called **incrementing** it subtracting 1 is called **decrementing** it. To indicate these operations in an extremely compact form, C++ uses the operators ++ and --. For example, the statement

```
x++;
```

in C++ has the same ultimate effect as

```
x += 1;
```

which is itself short for

```
x = x + 1;
```

Similarly,

```
y--;
```

has the same effect as

```
y -= 1;
```

or

```
y = y - 1;
```

As it happens, these operators are more intricate than the previous examples would suggest. To begin with, each of these operators can be written in two ways. The operator

can come after the operand to which it applies, as in

x++

or before the operand, as in

++x

The first form, in which the operator follows the operand, is called the **postfix** form, the second, the **prefix** form.

If all you do is execute the ++ operator in isolation—as you do in the context of a separate statement or a typical **for** loop like those in the **powertab.cpp** example—the prefix and postfix operators have precisely the same effect. You notice the difference only if you use these operators as part of a larger expression. Then, like all operators, the ++ operator returns a value, but the value depends on where the operator is written relative to the operand. The two cases are as follows:

- x++** Calculates the value of **x** first, and then increments it. The value returned to the surrounding expression is the original value *before* the increment operation is performed.
- ++x** Increments the value of **x** first, and then uses the new value as the value of the ++ operation as a whole.

The -- operator behaves similarly, except that the value is decremented rather than incremented.

You may wonder why would anyone use such an arcane feature. The ++ and -- operators are certainly not essential. Moreover, there are not many circumstances in which programs that embed these operators in larger expressions are demonstrably better than those that use a simpler approach. On the other hand, ++ and -- are firmly entrenched in the historical tradition shared by C++ programmers. They are idioms, and programmers use them frequently. Because these operators are so common, you need to understand them so that you can make sense of existing code.

Boolean operators

C++ defines three classes of operators that manipulate Boolean data: the relational operators, the logical operators, and the ?: operator. The **relational operators** are used to compare two values. C++ defines six relational operators, as follows:

- ==** Equal
- !=** Not equal
- >** Greater than
- <** Less than
- >=** Greater than or equal to
- <=** Less than or equal to

When you write programs that test for equality, be careful to use the == operator, which is composed of two equal signs. A single equal sign is the assignment operator. Since the double equal sign violates conventional mathematical usage, replacing it with a single equal sign is a particularly common mistake. This mistake can also be very difficult to track down because the C++ compiler does not usually catch it as an error. A single equal sign usually turns the expression

COMMON PITFALLS

When writing programs that test for equality, be sure to use the == operator and not the single = operator, which signifies assignment. This error is extremely common and can lead to bugs that are very difficult to find, because the compiler cannot detect the error.

into an embedded assignment, which is perfectly legal in C++; it just isn't at all what you want.

The relational operators can be used to compare atomic data values like integers, floating-point numbers, Boolean values, and characters. Some of the types supplied in the libraries, such as **string**, also can be compared using the relational operators.

In addition to the relational operators, C++ defines three **logical operators** that take Boolean operands and combine them to form other Boolean values:

- !** Logical *not* (**true** if the following operand is **false**)
- &&** Logical *and* (**true** if both operands are **true**)
- ||** Logical *or* (**true** if either or both operands are **true**)

These operators are listed in decreasing order of precedence.

Although the operators **&&**, **||**, and **!** closely resemble the English words *and*, *or*, and *not*, it is important to remember that English can be somewhat imprecise when it comes to logic. To avoid that imprecision, it is often helpful to think of these operators in a more formal, mathematical way. Logicians define these operators using **truth tables**, which show how the value of a Boolean expression changes as the values of its operands change. The following truth table illustrates the result for each of the logical operators, given all possible values of the variables **p** and **q**:

p	q	p && q	p q	!p
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

C++ interprets the **&&** and **||** operators in a way that differs from the interpretation used in many other programming languages such as Pascal. Whenever a C++ program evaluates an expression of the form

*exp*₁ **&&** *exp*₂

or

*exp*₁ **||** *exp*₂

the individual subexpressions are always evaluated from left to right, and evaluation ends as soon as the answer can be determined. For example, if *exp*₁ is **false** in the expression involving **&&**, there is no need to evaluate *exp*₂ since the final answer will always be **false**. Similarly, in the example using **||**, there is no need to evaluate the second operand if the first operand is **true**. This style of evaluation, which stops as soon as the answer is known, is called **short-circuit evaluation**.

The C++ programming language provides another Boolean operator that can be extremely useful in certain situations: the **?:** operator. (This operator is referred to as *question-mark colon*, even though the two characters do not appear adjacent to one another in the code.) Unlike any other operator in C++, **?:** is written in two parts and requires three operands. The general form of the operation is

(*condition*) **?** *exp*₁ **:** *exp*₂

The parentheses around the condition are not technically required, but C++ programmers often include them to emphasize the boundaries of the conditional test.

When a C++ program encounters the `?:` operator, it first evaluates the condition. If the condition turns out to be **true**, *exp₁* is evaluated and used as the value of the entire expression if the condition is **false**, the value is the result of evaluating *exp₂*. For example, you can use the `?:` operator to assign to **max** either the value of **x** or the value of **y**, whichever is greater, as follows:

```
max = (x > y) ? x : y;
```

1.5 Statements

Programs in C++ are composed of functions, which are made up in turn of statements. As in most languages, statements in C++ fall into one of two principal classifications: **simple statements**, which perform some action, and **control statements**, which affect the way in which other statements are executed. The sections that follow review the principal statement forms available in C++ and give you the fundamental tools you need to write your own programs.

Simple statements

The most common statement in C++ is the **simple statement**, which consists of an expression followed by a semicolon:

```
expression;
```

In most cases, the expression is a function call, an assignment, or a variable followed by the increment or decrement operator.

Blocks

As C++ is defined, control statements typically apply to a single statement. When you are writing a program, you often want the effect of a particular control statement to apply to a whole group of statements. To indicate that a sequence of statements is part of a coherent unit, you can assemble those statements into a **block**, which is a collection of statements enclosed in curly braces, as follows:

```
{  
    statement1  
    statement2  
    . . .  
    statementn  
}
```

When the C++ compiler encounters a block, it treats the entire block as a single statement. Thus, whenever the notation *statement* appears in a pattern for one of the control forms, you can substitute for it either a single statement or a block. To emphasize that they are statements as far as the compiler is concerned, blocks are sometimes referred to as **compound statements**. In C++, the statements in any block may be preceded by declarations of variables. In this text, variable declarations are introduced only in the block that defines the body of a function.

The statements in the interior of a block are usually indented relative to the enclosing context. The compiler ignores the indentation, but the visual effect is extremely helpful to the human reader, because it makes the structure of the program jump out at you from

the format of the page. Empirical research has shown that indenting three or four spaces at each new level makes the program structure easiest to see; the programs in this text use four spaces for each new level. Indentation is critical to good programming, so you should strive to develop a consistent indentation style in your programs.

The only aspect of blocks that tends to cause any confusion for new students is the role of the semicolon. In C++, the semicolon is part of the syntax of a simple statement; it acts as a statement *terminator* rather than as a statement *separator*. While this rule is perfectly consistent, it can cause trouble for people who have previously been exposed to the language Pascal, which uses a different rule. In practical terms, the differences are:

1. In C++, there is always a semicolon at the end of the last simple statement in a block. In Pascal, the semicolon is usually not present although most compilers allow it as an option.
2. In C++, there is never a semicolon after the closing brace of a statement block. In Pascal, a semicolon may or may not follow the **END** keyword, depending on context.

The convention for using semicolons in C++ has advantages for program maintenance and should not cause any problem once you are used to it.

The **if** statement

In writing a program, you will often want to check whether some condition applies and use the result of that check to control the subsequent execution of the program. This type of program control is called **conditional execution**. The easiest way to express conditional execution in C++ is by using the **if** statement, which comes in two forms:

```
if (condition) statement
```

```
if (condition) statement else statement
```

You use the first form of the **if** statement when your solution strategy calls for a set of statements to be executed only if a particular Boolean condition is **true**. If the condition is **false**, the statements that form the body of the **if** statement are simply skipped. You use the second form of the **if** statement for situations in which the program must choose between two independent sets of actions based on the result of a test. This statement form is illustrated by the following program, which reads in a number and classifies it as either even or odd.

```
int main() {
    int n;

    cout << "This program classifies a num even or odd." << endl;
    cout << "Enter a number: ";
    n = GetInteger();
    if (n % 2 == 0) {
        cout << "That number is even." << endl;
    } else {
        cout << "That number is odd." << endl;
    }
    return 0;
}
```

As with any control statement, the statements controlled by the **if** statement can be either a single statement or a block. Even if the body of a control form is a single

statement, you are free to enclose it in a block if you decide that doing so improves the readability of your code. The programs in this book enclose the body of every control statement in a block unless the entire statement—both the control form and its body—is so short that it fits on a single line.

The **switch** statement

The **if** statement is ideal for those applications in which the program logic calls for a two-way decision point: some condition is either **true** or **false**, and the program acts accordingly. Some applications, however, call for more complicated decision structures involving several mutually exclusive cases: in one case, the program should do *x*; in another case, it should do *y*; in a third, it should do *z*; and so forth. In many applications, the most appropriate statement to use for such situations is the **switch** statement, which has the following syntactic form:

```
switch (e) {
    case c1:
        statements
        break;
    case c2:
        statements
        break;
    . . . more case clauses . . .
    default:
        statements
        break;
}
```

The expression *e* is called the **control expression**. When the program executes a **switch** statement, it evaluates the control expression and compares it against the values *c*₁, *c*₂, and so forth, each of which must be a constant. If one of the constants matches the value of the control expression, the statements in the associated **case** clause are executed. When the program reaches the **break** statement at the end of the clause, the operations specified by that clause are complete, and the program continues with the statement that follows the entire **switch** statement.

The **default** clause is used to specify what action occurs if none of the constants match the value of the control expression. The **default** clause, however, is optional. If none of the cases match and there is no **default** clause, the program simply continues on with the next statement after the **switch** statement without taking any action at all. To avoid the possibility that the program might ignore an unexpected case, it is good programming practice to include a **default** clause in every **switch** statement unless you are certain you have enumerated all the possibilities, even if the **default** clause is simply

```
default:
    Error("Unexpected case value");
```

The **Error** function is part of the **genlib** library and provides a uniform way of responding to errors. This function takes one string parameter, the error message. The **Error** function does not return; after the error message is displayed, the program terminates.

The code pattern I've used to illustrate the syntax of the **switch** statement deliberately suggests that **break** statements are required at the end of each clause. In fact, C++ is defined so that if the **break** statement is missing, the program starts executing statements

from the next clause after it finishes the selected one. While this design can be useful in some cases, it causes many more problems than it solves. To reinforce the importance of remembering to exit at the end of each **case** clause, the programs in this text always include a **break** or **return** statement in each such clause.

The one exception to this rule is that multiple **case** lines specifying different constants can appear together, one after another, before the same statement group. For example, a **switch** statement might include the following code:

```
case 1:
case 2:
    statements
    break;
```

which indicates that the specified statements should be executed if the **select** expression is either 1 or 2. The C++ compiler treats this construction as two **case** clauses, the first of which is empty. Because the empty clause contains no **break** statement, a program that selects the first path simply continues on with the second clause. From a conceptual point of view, however, you are better off if you think of this construction as a single **case** clause representing two possibilities.

The operation of the **switch** statement is illustrated by the following function, which computes the number of days for a given month and year:

```
int MonthDays(int month, int year) {
    switch (month) {
        case September:
        case April:
        case June:
        case November:
            return 30;
        case February:
            return (IsLeapYear(year)) ? 29 : 28;
        default:
            return 31;
    }
}
```

The code assumes that there is a function **IsLeapYear(year)** which tests whether **year** is a leap year and that the names of the months have been defined using constants, as follows:

```
const int JANUARY   = 1;
const int FEBRUARY  = 2;
const int MARCH     = 3;
const int APRIL     = 4;
const int MAY       = 5;
const int JUNE      = 6;
const int JULY      = 7;
const int AUGUST    = 8;
const int SEPTEMBER = 9;
const int OCTOBER   = 10;
const int NOVEMBER  = 11;
const int DECEMBER  = 12;
```

COMMON PITFALLS

It is good programming practice to include a **break** statement at the end of every **case** clause within a **switch** statement. Doing so will help you avoid programming errors that can be extremely difficult to find. It is also good practice to include a **default** clause unless you are sure you have covered all the cases.

The constants in a **switch** statement must be of integer type or a type that behaves like an integer. (The actual restriction is that the type must be a *scalar type*, which is defined in Chapter 2.) In particular, characters are often used as **case** constants, as illustrated by the following function, which tests to see if its argument is a vowel:

```
bool IsVowel(char ch) {
    switch (ch) {
        case 'A': case 'E': case 'I': case 'O': case 'U':
        case 'a': case 'e': case 'i': case 'o': case 'u':
            return true;
        default:
            return false;
    }
}
```

The while statement

In addition to the conditional statements **if** and **switch**, C++ includes several control statements that allow you to execute some part of the program multiple times to form a loop. Such control statements are called **iterative statements**. The simplest iterative construct in C++ is the **while** statement, which executes a statement repeatedly until a conditional expression becomes **false**. The general form for the **while** statement looks like this:

```
while (conditional-expression) {
    statements
}
```

When a program encounters a **while** statement, it first evaluates the conditional expression to see whether it is **true** or **false**. If it is **false**, the loop **terminates** and the program continues with the next statement after the entire loop. If the condition is **true**, the entire body is executed, after which the program goes back to the beginning of the loop to check the condition again. A single pass through the statements in the body constitutes a **cycle** of the loop.

There are two important principles about the operation of a **while** loop:

1. The conditional test is performed before every cycle of the loop, including the first. If the test is **false** initially, the body of the loop is not executed at all.
2. The conditional test is performed only at the *beginning* of a loop cycle. If that condition happens to become **false** at some point during the loop, the program doesn't notice that fact until a complete cycle has been executed. At that point, the program evaluates the test condition again. If it is still **false**, the loop terminates.

The operation of the **while** loop is illustrated by the following function, which computes the sum of the digits in an integer:

```
int DigitSum(int n) {
    int sum;

    sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
```

The function depends on the following observations:

- The expression `n % 10` always returns the last digit in a positive integer `n`.
- The expression `n / 10` returns a number without its final digit.

The **while** loop is designed for situations in which there is some test condition that can be applied at the beginning of a repeated operation, before any of the statements in the body of the loop are executed. If the problem you are trying to solve fits this structure, the **while** loop is the perfect tool. Unfortunately, many programming problems do not fit easily into the standard **while** loop structure. Instead of allowing a convenient test at the beginning of the operation, some problems are structured in such a way that the test you want to write to determine whether the loop is complete falls most naturally somewhere in the middle of the loop.

The most common example of such loops are those that read in data from the user until some special value, or **sentinel**, is entered to signal the end of the input. When expressed in English, the structure of the sentinel-based loop consists of repeating the following steps:

1. Read in a value.
2. If the value is equal to the sentinel, exit from the loop.
3. Perform whatever processing is required for that value.

Unfortunately, there is no test you can perform at the beginning of the loop to determine whether the loop is finished. The termination condition for the loop is reached when the input value is equal to the sentinel; in order to check this condition, the program must first read in some value. If the program has not yet read in a value, the termination condition doesn't make sense. Before the program can make any meaningful test, it must have executed the part of the loop that reads in the input value. When a loop contains some operations that must be performed before testing for completion, you have a situation that programmers call the **loop-and-a-half problem**.

One way to solve the loop-and-a-half problem in C++ is to use the **break** statement, which, in addition to its use in the **switch** statement, has the effect of immediately terminating the innermost enclosing loop. By using **break**, it is possible to code the loop structure for the sentinel problem in a form that follows the natural structure of the problem:

```
while (true) {  
    Prompt user and read in a value.  
    if (value == sentinel) break;  
    Process the data value.  
}
```

Note that the

```
while (true)
```

line itself seems to introduce an infinite loop because the value of the constant **true** can never become **false**. The only way this program can exit from the loop is by executing the **break** statement inside it. The loop-and-a-half strategy is illustrated by the **addlist.cpp** program in Figure 1-4, which computes the sum of a list of integers terminated by the sentinel value 0.

Figure 1-4 Program to add a list of integers

```

/*
 * File: addlist.cpp
 * -----
 * This program adds a list of numbers. The end of the
 * input is indicated by entering a sentinel value, which
 * is defined by setting the value of the constant SENTINEL.
 */

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

/*
 * Constant: SENTINEL
 * -----
 * This constant defines the value used to terminate the input
 * list and should therefore not be a value one would want to
 * include as a data value. The value 0 therefore makes sense
 * for a program that adds a series of numbers because the
 * user can simply skip any 0 values in the input.
 */

const int SENTINEL = 0;

/* Main program */

int main() {
    cout << "This program adds a list of numbers." << endl;
    cout << "Use " << SENTINEL << " to signal the end." << endl;
    int total = 0;
    while (true) {
        cout << " ? ";
        int value = GetInteger();
        if (value == SENTINEL) break;
        total += value;
    }
    cout << "The total is " << total << endl;
    return 0;
}

```

There are other strategies for solving the loop-and-a-half problem that involve copying part of the code outside the loop. However, empirical studies have demonstrated that students are more likely to write correct programs if they use a **break** statement to exit from the middle of the loop than if they are forced to use some other strategy. This evidence and my own experience have convinced me that using the **break** statement inside a **while** loop is the best solution to the loop-and-a-half problem.

The **for** statement

One of the most important control statements in C++ is the **for** statement, which is used in situations in which you want to repeat an operation a particular number of times. The general form is

```

for (init; test; step) {
    statements
}

```

which is equivalent to the **while** statement

```
init;
while (test) {
    statements
    step;
}
```

The operation of the **for** loop is determined by the three italicized expressions on the **for** control line: *init*, *test*, and *step*. The *init* expression indicates how the **for** loop should be initialized and usually declares and initializes the index variable. For example, if you write

```
for (int i = 0; . . .
```

the loop will begin by setting the index variable *i* to 0. If the loop begins

```
for (int i = -7; . . .
```

the variable *i* will start as -7, and so on.

Note the initialization expression does not have to also declare the index variable. It may instead just initialize an already declared variable, but typically it is convenient to place the declaration inside the **for** loop itself.¹

The *test* expression is a conditional test written exactly like the test in a **while** statement. As long as the test expression is **true**, the loop continues. Thus, the loop

```
for (int i = 0; i < n; i++)
```

begins with *i* equal to 0 and continues as long as *i* is less than *n*, which turns out to represent a total of *n* cycles, with *i* taking on the values 0, 1, 2, and so forth, up to the final value *n*-1. The loop

```
for (int i = 1; i <= n; i++)
```

begins with *i* equal to 1 and continues as long as *i* is less than or equal to *n*. This loop also runs for *n* cycles, with *i* taking on the values 1, 2, and so forth, up to *n*.

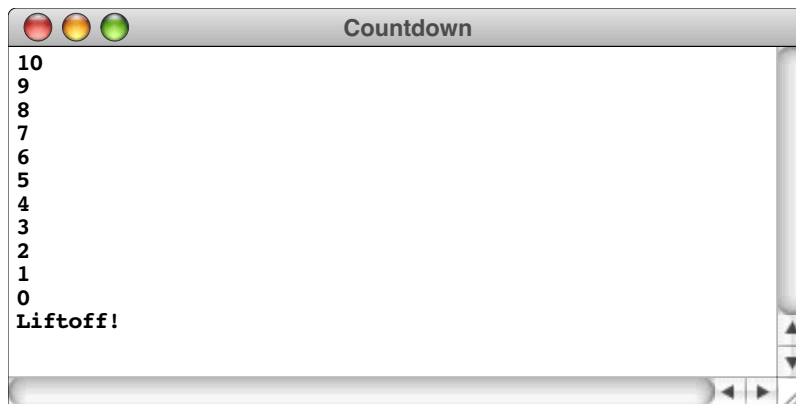
The *step* expression indicates how the value of the index variable changes from cycle to cycle. The most common form of step specification is to increment the index variable using the ++ operator, but this is not the only possibility. For example, one can count backward by using the -- operator, or count by twos by using += 2 instead of ++.

As an illustration of counting in the reverse direction, the program

```
int main() {
    for (int t = 10; t >= 0; t--) {
        cout << t << endl;
    }
    cout << "Liftoff!" << endl;
    return 0;
}
```

¹ The C++ standard states that the scope of an index variable declared in the initialization expression of a **for** loop extends to the end of the loop body and no further. Some C++ compilers, such as Visual C++ 6.0, mistakenly extend the scope to the end of the block enclosing the entire loop.

generates the following sample run:



The expressions *init*, *test*, and *step* in a **for** statement are each optional, but the semicolons must appear. If *init* is missing, no initialization is performed. If *test* is missing, it is assumed to be **true**. If *step* is missing, no action occurs between loop cycles.

1.6 Functions

A **function** consists of a set of statements that have been collected together and given a name. The act of using the name to invoke the associated statements is known as **calling** that function. To indicate a function call in C++, you write the name of the function, followed by a list of expressions enclosed in parentheses. These expressions, called **arguments**, allow the calling program to pass information to the function. For example, in the **powertab.cpp** program at the beginning of this chapter, the function **RaiseIntToPower** took two integer arguments, **n** and **k**, which are the values it needs to know in order to compute n^k . If a function requires no information from its caller, it need not have any arguments, but an empty set of parentheses must appear in the function call.

Once called, the function takes the data supplied as arguments, does its work, and then returns to the program step from which the call was made. Remembering what the calling program was doing and being able to get back precisely to that point is one of the defining characteristics of the function-calling mechanism. The operation of going back to the calling program is called **returning** from the function.

Returning results from functions

As they return, functions can send results back to the calling program. Thus, when the **RaiseIntToPower** function returns with the statement

```
return result;
```

the value of the local variable **result** is passed back to the main program as the value of the function. This operation is called **returning a value**.

Functions can return values of any type. The following function, for example, returns a value of type **bool**, which can then be used in conditional expressions:

```
bool IsLeapYear(int year) {
    return ((year % 4 == 0) && (year % 100 != 0))
        || (year % 400 == 0);
}
```


Functions that return Boolean results play an important role in programming and are called **predicate functions**.

Functions, however, do not need to return a value at all. A function that does not return a value and is instead executed for its effect is called a **procedure**. Procedures are indicated in the definition of a function by using the reserved word **void** as the result type.

The **return** statement in C++ has two forms. For procedures, you write the statement as

```
return;
```

For functions that return a value, the **return** keyword is followed by an expression, as follows:

```
return expression;
```

Executing either form of the **return** statement causes the current function to return immediately to its caller, passing back the value of the expression, if any, to its caller as the value of the function.

Function definitions and prototypes

A function definition has the following syntactic form:

```
result-type name (parameter-list) {  
    . . . body . . .  
}
```

In this example, *result-type* is the type of value returned by the function, *name* is the function name, and *parameter-list* is a list of declarations separated by commas, giving the type and name of each parameter to the function. **Parameters** are placeholders for the arguments supplied in the function call and act like local variables except for the fact that they are given initial values by the calling program. If a function takes no parameters, the entire parameter list in the function header line is empty. The body of the function is a block and typically contains declarations for the local variables required by the function.

Before you use a function in a C++ program, you declare it by specifying its prototype. A prototype has exactly the same form as a function definition, except that the entire body is replaced by a semicolon. The names of the parameter variables are optional in a prototype, but supplying them usually helps the reader.

The mechanics of the function-calling process

When you call a function in a program, the following steps occur:

1. The calling program computes values for each argument. Because the arguments are expressions, this computation can involve operators and other functions, all of which are evaluated before execution of the new function actually begins.
2. The system creates new space for all the local variables required by the new function, including the parameter variables. These variables are usually allocated together in a block, which is called a **stack frame**.
3. The value of each argument is copied into the corresponding parameter variable. If there is more than one argument, the arguments are copied into the parameters in

order; the first argument is copied into the first parameter, and so forth. If necessary, type conversions are performed between the argument values and the parameter variables, as in an assignment statement. For example, if you pass a value of type **int** to a function that expects a parameter of type **double**, the integer is converted into the equivalent floating-point value before it is copied into the parameter variable.

4. The statements in the function body are executed until a **return** statement is encountered or there are no more statements to execute.
5. The value of the **return** expression, if any, is evaluated and returned as the value of the function. If the value being returned does not precisely match the result type declared for the function, a type conversion is performed. Thus, if a **return** statement specifies a floating-point value in a function defined to return an **int**, the result is truncated to an integer.
6. The stack frame created for this function call is discarded. In the process, all local variables disappear.
7. The calling program continues, with the returned value substituted in place of the call.

Passing parameters by reference

In C++, whenever you pass a simple variable from one function to another, the function gets a copy of the calling value. Assigning a new value to the parameter as part of the function changes the local copy but has no effect on the calling argument. For example, if you try to implement a function that initializes a variable to zero using the code

```
void SetToZero(int var) {
    var = 0;
}
```



This function has no effect.

the function has no effect whatever. If you call

```
SetToZero(x);
```

the parameter **var** is initialized to a copy of whatever value is stored in **x**. The assignment statement

```
var = 0;
```

inside the function sets the local copy to 0 but leaves **x** unchanged in the calling program.

To address this problem, you can change the parameter to a **reference parameter** by adding an ampersand to the parameter declaration in the function header. Now the parameter value will not be copied, instead a reference is made to the original variable. Changes to the parameter are reflected in the original variable. The new coding is

```
void SetToZero(int &var) {
    var = 0;
}
```

To use this function, the caller must pass an assignable integer variable. To set **x** to 0, for example, you would need to make the following call:

```
SetToZero(x);
```

Passing an integer constant such as **3** would be an error because **3** is not an assignable integer variable.

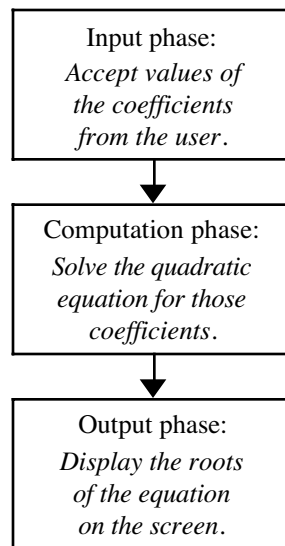
The use of reference parameters makes it possible for functions to change values in the frame of their caller. This mechanism is referred to as **call by reference**.

In C++, one of the common uses of call by reference occurs when a function needs to return more than one value to the calling program. A single result can easily be returned as the value of the function itself. If you need to return more than one result from a function, the return value is no longer appropriate. The standard approach to solving the problem is to turn that function into a procedure and pass values back and forth through the argument list.

As an example, suppose that you wanted to write a program to solve the quadratic equation

$$ax^2 + bx + c = 0$$

but that—because of your desire to practice good programming style—you were committed to dividing the work of the program into three phases as represented by the boxes in the following flowchart:



Decomposing this problem into separate functions that are responsible for each of these phases is somewhat tricky because several values must be passed from each phase to the next. Because there are three coefficients, you would like the input phase to return three values. Similarly, the computation phase must return two values, because a quadratic equation has two solutions.

Figure 1-5 shows how call by reference makes it possible to decompose the quadratic equation problem in this way. At each level, parameters that act as input to each function are passed in the conventional way; parameters that represent output from the function are passed by reference.

Stepwise refinement

Procedures and functions enable you to divide a large programming problem into smaller pieces that are individually easy to understand. The process of dividing a problem into manageable pieces, called **decomposition**, is a fundamental programming strategy. Finding the right decomposition, however, turns out to be a difficult task that requires considerable practice. If you choose the individual pieces well, each one will have

Figure 1-5 Implementation of quadeq.cpp that illustrates call by reference

```

/*
 * File: quadeq.cpp
 * -----
 * This program finds roots of the quadratic equation
 *
 *      2
 *      a x  + b x + c = 0
 *
 * If a is 0 or if the equation has no real roots, the
 * program exits with an error.
 */

#include <iostream>
#include <cmath>
#include "genlib.h"
#include "simpio.h"

/* Private function prototypes */

void GetCoefficients(double & a, double & b, double & c);
void SolveQuadratic(double a, double b, double c,
                    double & x1, double & x2);
void DisplayRoots(double x1, double x2);

/* Main program */

int main() {
    double a, b, c, r1, r2;

    GetCoefficients(a, b, c);
    SolveQuadratic(a, b, c, r1, r2);
    DisplayRoots(r1, r2);
    return 0;
}

/*
 * Function: GetCoefficients
 * Usage: GetCoefficients(a, b, c);
 * -----
 * This function is responsible for reading in the coefficients
 * of a quadratic equation. The values of the coefficients are
 * passed back to the main program in the variables a, b, and c,
 * which are reference parameters.
 */

void GetCoefficients(double & a, double & b, double & c) {
    cout << "Enter coefficients for the quadratic equation:" << endl;
    cout << "a: ";
    a = GetReal();
    cout << "b: ";
    b = GetReal();
    cout << "c: ";
    c = GetReal();
}

```

```

/*
 * Function: SolveQuadratic
 * Usage: SolveQuadratic(a, b, c, x1, x2);
 * -----
 * This function solves a quadratic equation. The coefficients
 * are supplied as the arguments a, b, and c, and the roots are
 * returned in x1 and x2, which are reference parameters.
 */

void SolveQuadratic(double a, double b, double c,
                    double & x1, double & x2) {
    if (a == 0) Error("The coefficient a must be nonzero");
    double disc = b * b - 4 * a * c;
    if (disc < 0) Error("The solutions are complex numbers");
    double sqrtDisc = sqrt(disc);
    x1 = (-b + sqrtDisc) / (2 * a);
    x2 = (-b - sqrtDisc) / (2 * a);
}

/*
 * Function: DisplayRoots
 * Usage: DisplayRoots(x1, x2);
 * -----
 * This function displays the values x1 and x2, which are
 * the roots of a quadratic equation.
 */

void DisplayRoots(double x1, double x2) {
    if (x1 == x2) {
        cout << "There is a double root at " << x1 << endl;
    } else {
        cout << "The roots are " << x1 << " and " << x2 << endl;
    }
}

```

conceptual integrity as a unit and make the program as a whole much simpler to understand. But if you choose unwisely, the decomposition can get in your way. There are no hard-and-fast rules for selecting a particular decomposition; you will learn how to apply this process through experience.

When you are faced with the task of writing a program, the best strategy is usually to start with the main program. At this level, you think about the problem as a whole and then try to identify the major pieces of the entire task. Once you figure out what the big pieces of the program are, you can define them as independent functions. Since some of these functions may themselves be complicated, it is often appropriate to decompose them into still smaller ones. You can continue this process until every piece of the problem is simple enough to be solved on its own. This process is called **top-down design**, or **stepwise refinement**.

Summary

This chapter is itself a summary, which makes it hard to condense it to a few central points. Its purpose was to introduce you to the C++ programming language and give you a crash course in how to write simple programs in that language. This chapter concentrated on the low-level structure of the language, proceeding in turn through the

topics of expressions, statements, and functions. The facilities that C++ offers for defining new data structures are detailed in Chapter 2.

Important points in this chapter include:

- In the 25 years of its existence, the C++ programming language has become one of the most widely used languages in the world.
- A typical C++ program consists of comments, library inclusions, program-level definitions, function prototypes, a function named **main** that is called when the program is started, and a set of auxiliary function definitions that work together with the main program to accomplish the required task.
- Variables in a C++ program must be declared before they are used. Most variables in C++ are *local variables*, which are declared within a function and can only be used inside the body of that function.
- A *data type* is defined by a domain of values and a set of operations. C++ includes several predefined types that allow programs to store data of several different types, such as integers, floating-point numbers, Booleans, and characters. In addition to these built-in types, the standard library defines the type **string**, which is treated in this book as if it were an integral part of the language.
- The easiest way to read input data from the user is to call functions in the simplified I/O library (**simpio**), which defines such functions as **GetInteger**, **GetReal**, and **GetLine**. To display output on the computer screen, the usual approach is to insert the values into the standard **cout** stream using the insertion operator **<<**.
- Expressions in C++ are written in a form similar to that in most programming languages, with individual terms connected by operators. A complete list of the operators available in C++ appears in Table 1-4, which also indicates the relative precedence of each operator.
- Statements in C++ fall into two classes: simple statements and control statements. A simple statement consists of an expression—typically an assignment or a function call—followed by a semicolon. The control statements described in this chapter are the **if**, **switch**, **while**, and **for** statements. The first two are used to express conditional execution, while the last two are used to specify repetition.
- C++ programs are typically subdivided into several functions. Each function consists of a sequence of statements that can be invoked by writing the name of the function, followed by a list of arguments enclosed in parentheses. These arguments are copied into the corresponding parameter variables inside the function. The function can return a result to the caller by using the **return** statement and can share values using reference parameters.

Review questions

1. What is the difference between a source file and an object file?
2. What characters are used to mark comments in a C++ program?
3. In an **#include** line, the name of the library header file can be enclosed in either angle brackets or double quotation marks. What is the difference between the two forms of punctuation?
4. How would you define a constant called **CENTIMETERS_PER_INCH** with the value 2.54?

5. What is the name of the function that must be defined in every C++ program?
6. What is the purpose of inserting **endl** into the output stream **cout**?
7. What four properties are established when you declare a variable?
8. Indicate which of the following are legal variable names in C++:
 - a. **x**
 - b. **formula1**
 - c. **average_rainfall**
 - d. **%correct**
 - e. **short**
 - f. **tiny**
 - g. **total output**
 - h. **aReasonablyLongVariableName**
 - i. **12MonthTotal**
 - j. **marginal-cost**
 - k. **b4hand**
 - l. **_stk_depth**
9. What are the two attributes that define a data type?
10. What is the difference between the types **short**, **int**, and **long**?
11. What does ASCII stand for?
12. List all possible values of type **bool**.
13. What statements would you include in a program to read a value from the user and store it in the variable **x**, which is declared as a **double**?
14. Suppose that a function contains the following declarations:

```
int i;  
double d;  
char c;  
string s;
```

Write a statement that displays the values of each of these variables on the screen.

15. Indicate the values and types of the following expressions:
 - a. **2 + 3**
 - b. **19 / 5**
 - c. **19.0 / 5**
 - d. **3 * 6.0**
 - e. **19 % 5**
 - f. **2 % 7**
16. What is the difference between the unary minus and the binary subtraction operator?
17. What does the term *truncation* mean?
18. Calculate the result of each of the following expressions:
 - a. **6 + 5 / 4 - 3**
 - b. **2 + 2 * (2 * 2 - 2) % 2 / 2**
 - c. **10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 * 2 + 1**
 - d. **1 + 2 + (3 + 4) * ((5 * 6 % 7 * 8) - 9) - 10**
19. How do you specify a shorthand assignment operation?
20. What is the difference between the expressions **++x** and **x++**?
21. What does the term *short-circuit evaluation* mean?

22. Write out the general syntactic form for each of the following control statements: **if**, **switch**, **while**, **for**.
23. Describe in English the general operation of the **switch** statement.
24. What is a sentinel?
25. What **for** loop control line would you use in each of the following situations?
 - a. Counting from 1 to 100
 - b. Counting by sevens starting at 0 until the number has more than two digits
 - c. Counting backward by twos from 100 to 0
26. What is a function prototype?
27. In your own words, describe what happens when you call a function in C++.
28. What is meant by the term *stepwise refinement*?

Programming exercises

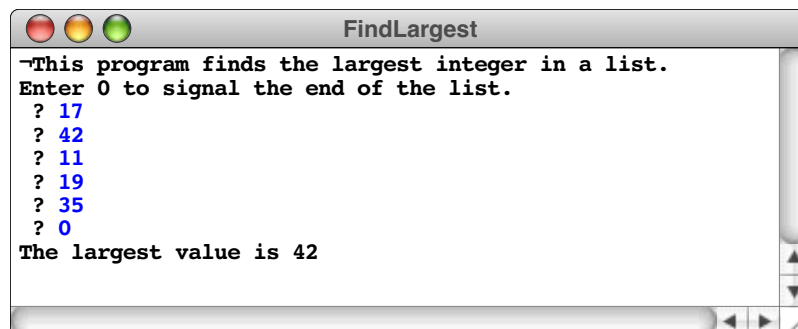
1. Write a program that reads in a temperature in degrees Celsius and displays the corresponding temperature in degrees Fahrenheit. The conversion formula is

$$F = \frac{9}{5}C + 32$$

2. Write a program that converts a distance in meters to the corresponding English distance in feet and inches. The conversion factors you need are

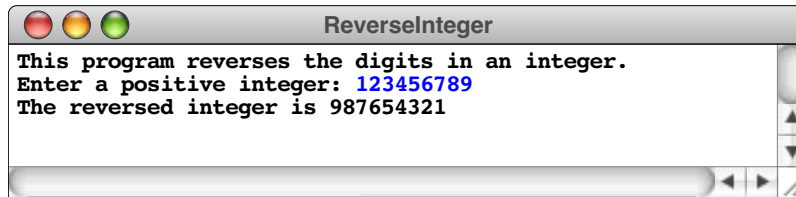
$$\begin{aligned} 1 \text{ inch} &= 0.0254 \text{ meters} \\ 1 \text{ foot} &= 12 \text{ inches} \end{aligned}$$

3. According to legend, the German mathematician Karl Friedrich Gauss (1777–1855) began to show his mathematical talent at a very early age. When he was in elementary school, Gauss was asked by his teacher to compute the sum of the numbers between 1 and 100. Gauss is said to have given the answer instantly: 5050. Write a program that computes the answer to the question Gauss’s teacher posed.
4. Write a program that reads in a positive integer N and then calculates and displays the sum of the first N odd integers. For example, if N is 4, your program should display the value 16, which is $1 + 3 + 5 + 7$.
5. Write a program that reads in a list of integers from the user until the user enters the value 0 as a sentinel. When the sentinel appears, your program should display the largest value in the list, as illustrated in the following sample run:



```
FindLargest
~This program finds the largest integer in a list.
Enter 0 to signal the end of the list.
? 17
? 42
? 11
? 19
? 35
? 0
The largest value is 42
```


6. Using the **DigitSum** function from the section entitled “The **while** statement” as a model, write a program that reads in an integer and then displays the number that has the same digits in the reverse order, as illustrated by this sample run:



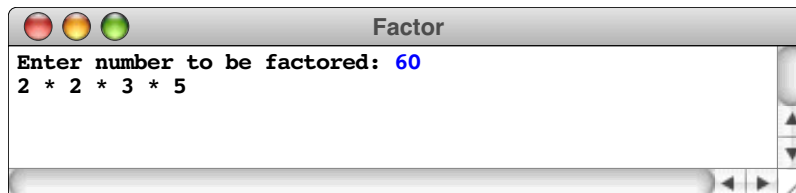
To make sure your program can handle integers as large as the one shown in the example, use the type **long** instead of **int** in your program.

7. Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of N is any divisor less than N itself). They called such numbers **perfect numbers**. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.

Write a predicate function **IsPerfect** that takes an integer n and returns **true** if n is perfect, and **false** otherwise. Test your implementation by writing a main program that uses the **IsPerfect** function to check for perfect numbers in the range 1 to 9999 by testing each number in turn. When a perfect number is found, your program should display it on the screen. The first two lines of output should be 6 and 28. Your program should find two other perfect numbers in the range as well.

8. Every positive integer greater than 1 can be expressed as a product of prime numbers. This factorization is unique and is called the **prime factorization**. For example, the number 60 can be decomposed into the factors $2 \times 2 \times 3 \times 5$, each of which is prime. Note that the same prime can appear more than once in the factorization.

Write a program to display the prime factorization of a number n , as illustrated by the following sample run:



9. When a floating-point number is converted to an integer in C++, the value is truncated by throwing away any fraction. Thus, when 4.99999 is converted to an integer, the result is 4. In many cases, it would be useful to have the option of *rounding* a floating-point value to the nearest integer. For a positive floating-point number x , the rounding operation can be achieved by adding 0.5 to x and then truncating the result to an integer. If the decimal fraction of x is less than .5, the truncated value will be the integer less than x ; if the fraction is .5 or more, the truncated value will be the next larger integer. Because truncation always moves toward zero, negative numbers must be rounded by subtracting 0.5 and truncating, instead of adding 0.5.

Write a function **Round(x)** that rounds a floating-point number x to the nearest integer. Show that your function works by writing a suitable main program to test it.

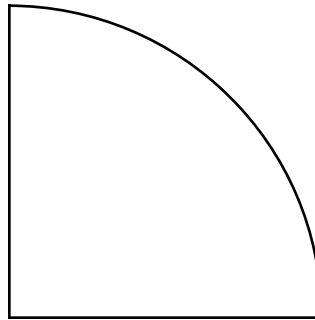
10. The German mathematician Leibniz (1646–1716) discovered the rather remarkable fact that the mathematical constant π can be computed using the following mathematical relationship:

$$\frac{\pi}{4} \cong 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots$$

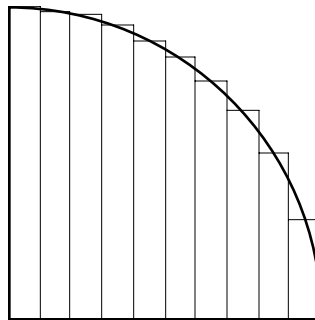
The formula to the right of the equal sign represents an infinite series; each fraction represents a term in that series. If you start with 1, subtract one-third, add one-fifth, and so on, for each of the odd integers, you get a number that gets closer and closer to the value of $\pi/4$ as you go along.

Write a program that calculates an approximation of π consisting of the first 10,000 terms in Leibniz's series.

11. You can also approximate π by approximating the area bounded by a circular arc. Consider the following quarter circle:



which has a radius r equal to two inches. From the formula for the area of a circle, you can easily determine that the area of the quarter circle should be π square inches. You can also approximate the area computationally by adding up the areas of a series of rectangles, where each rectangle has a fixed width and the height is chosen so that the circle passes through the midpoint of the top of the rectangle. For example, if you divide the area into 10 rectangles from left to right, you get the following diagram:



The sum of the areas of the rectangles approximates the area of the quarter circle. The more rectangles there are, the closer the approximation.

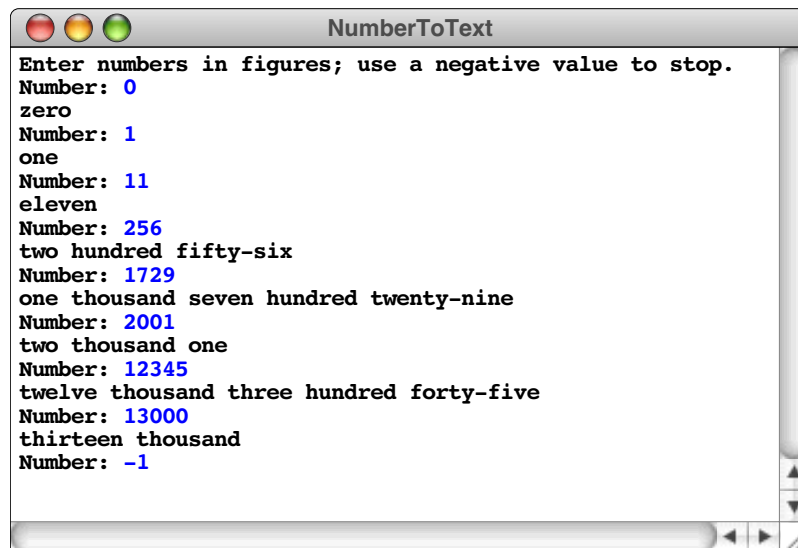
For each rectangle, the width w is a constant derived by dividing the radius by the number of rectangles. The height h , on the other hand, varies depending on the position of the rectangle. If the midpoint of the rectangle in the horizontal direction is given by x , the height of the rectangle can be computed using the distance formula

$$h = \sqrt{r^2 - x^2}$$

The area of each rectangle is then simply $h \times w$.

Write a program to compute the area of the quarter circle by dividing it into 100 rectangles.

12. When you write a check, the dollar amount appears twice: once as a number and once as English text. For example, if you write a check for \$1729, you need to translate that number to the English text “one thousand seven hundred twenty-nine.” Your task in this problem is to write a program that reads in integers from the user and writes out the equivalent value in figures on the next line, stopping when the user enters any negative number. For example, the following is a sample run of this program:



The key idea in this exercise is decomposition. The problem is not nearly as hard as it looks if you break it down into separate procedures that accomplish parts of the task. Many of these procedures will have a form that looks something like this:

```
void PrintOneDigit(int d) {
    switch (d) {
        case 0: cout << "zero"; break;
        case 1: cout << "one"; break;
        case 2: cout << "two"; break;
        case 3: cout << "three"; break;
        case 4: cout << "four"; break;
        case 5: cout << "five"; break;
        case 6: cout << "six"; break;
        case 7: cout << "seven"; break;
        case 8: cout << "eight"; break;
        case 9: cout << "nine"; break;
        default: Error("Illegal call to PrintOneDigit");
    }
}
```

In writing your program, you should keep the following points in mind:

- You don’t need to perform any string manipulation. All you have to do is display the value on the screen, which means that inserting `cout` is all you need.

- Your program need work only with values up to 999,999, although it should give the user some kind of error message if a number is outside of its range.
- It is perfectly acceptable for all the letters in the output to be lowercase. The problem is much harder if you try to capitalize the first word.
- You should remain on the lookout for functions that you can reuse. For example, printing the number of thousands is pretty much the same as printing out the last three digits, and you should be able to use the same procedure more than once.
- Several special cases arise in this problem. For example, the number 11 must be treated differently than 21 or 31, because *eleven* doesn't fit the pattern established by *twenty-one* and *thirty-one*.

Chapter 2

Data Types in C++

*It is a capital mistake to theorise before one has data.
Insensibly one begins to twist facts to suit theories, instead
of theories to suit facts.*

— Sherlock Holmes, in Sir Arthur Conan
Doyle's *A Scandal in Bohemia*, 1892

Chapter 1 of this text is a capsule summary of the features of C++ necessary to code the algorithmic structure of a program. The algorithmic structure, however, represents only part of the story. It is equally important to consider the structure of the data.

Like control statements and function calls—each of which can be nested hierarchically to represent increasingly complex algorithmic structures—data types in a language also form a hierarchy. The base of the hierarchy is composed of the atomic types that were introduced in Chapter 1, coupled with a new class of atomic types called *enumeration types* that are introduced in the following section. Starting from this base, you can extend the hierarchy using the following mechanisms for creating new types from existing ones:

- *Pointers*. A **pointer** is simply the internal machine address of a value inside the computer's memory. C++ allows you to work directly with pointers as data and makes them part of the type hierarchy, which means that you can define new types whose domains consist of pointers to values of some existing type.
- *Arrays*. An **array** is an ordered collection of data values, each of which has the same type.
- *Records*. A **record** is a collection of data values that represents some logically coherent whole. The individual components are identified by name rather than by order and may be of different types.

Each of these types is described in detail in a separate section later in this chapter. For now, the main point is that you can combine these mechanisms to generate new types at whatever level of complexity the program requires. You can, for example, create new types that are pointers to records containing arrays or any other nested structure you choose. The hierarchy of types in a program defines its **data structure**.

2.1 Enumeration types

Before moving on to the question of how to create new types from existing ones, it is important to complete the set of atomic types that form the base of the type hierarchy. The most common atomic types are the various built-in types described in Chapter 1: integers, floating-point numbers, characters, and so on. Like many languages, however, C++ makes it possible to define new atomic types by listing the elements that constitute their domains. Such types are called **enumeration types**.

In this text, new enumeration types are defined using the following syntactic form:

```
enum name { element-list };
```

where *element-list* is a list of identifiers, which are called **enumeration constants**, and *name* is the name of the new type. For example, the following enumeration type defines the four principal compass directions:

```
enum directionT { North, East, South, West };
```

Similarly, the following definition introduces the type **colorT**, which consists of the six primary and secondary colors available on a standard color monitor:

```
enum colorT {
    Red, Green, Blue, Yellow, Cyan, Magenta
};
```

Once you have defined an enumeration type, you can declare variables of that type just as you do with any of the built-in types. For example, the declaration

```
directionT dir;
```

declares the variable **dir** to be of type **directionT**, which means that it can take on any of the values **North**, **East**, **South**, or **West**.

Internal representation of enumeration types

The values of an enumeration type are stored internally as integers. When the compiler encounters a new enumeration type, it ordinarily assigns consecutive integers to the enumeration constants, starting with the integer 0. Thus, in the **directionT** example, the constants have the following values: **North** = 0, **East** = 1, **South** = 2, and **West** = 3. You can, however, control the encoding used for enumeration types by writing an equal sign and an integer constant after any of the element names. For example, after the definition

```
enum coinT {
    Penny = 1,
    Nickel = 5,
    Dime = 10,
    Quarter = 25,
    HalfDollar = 50
};
```

each of the enumeration constants **Penny**, **Nickel**, **Dime**, **Quarter**, and **HalfDollar** is represented internally as its corresponding monetary value. If the value of any enumeration constant is not specified, the compiler simply adds one to the value of the previous constant. Thus, in the definition,

```
enum monthT {
    January = 1, February, March, April, May, June,
    July, August, September, October, November, December
};
```

the constant **January** is represented internally as 1, **February** as 2, and so on.

Defining an enumeration type is in many ways similar to defining named constants. In the **monthT** example, you could achieve a similar effect by making the following definitions:

```
const int JANUARY    = 1;
const int FEBRUARY   = 2;
const int MARCH      = 3;
const int APRIL      = 4;
const int MAY        = 5;
const int JUNE       = 6;
const int JULY       = 7;
const int AUGUST     = 8;
const int SEPTEMBER  = 9;
const int OCTOBER    = 10;
const int NOVEMBER   = 11;
const int DECEMBER   = 12;
```

Inside the machine, the two strategies produce the same result: every element of the enumeration type is represented by an integer code. From the programmer's point of view, however, defining separate enumeration types has these advantages:

- The programmer does not need to specify the internal codes explicitly.
- The fact that there is a separate type name often makes the program easier to read because declarations can use a meaningful type name instead of the general-purpose designation `int`.
- A C++ compiler does some rudimentary checking for enumeration types. For example, an integer value cannot be assigned to an enum variable without a typecast, which helps to draw attention to possible mistakes such as assigning a value out of range for the enumeration.
- On many systems, programs that use enumeration types are easier to debug because the compiler makes the names of the enumeration constants available to the debugger. Thus, if you ask it about a value of type `monthT`, a well-designed debugger would be able to display the value `January` instead of the integer constant 1.

Scalar types

In C++, enumeration types, characters, and the various representations of integers form a more general type class called **scalar types**. When a value of a scalar type is used in a C++ expression, the compiler automatically converts it to the integer used to represent that value. The effect of this rule is that the operations you can perform on values of any scalar type are the same as those for integers.

As an example, suppose that you want to write a function `RightFrom(dir)` that takes a `directionT` and returns the direction you would be facing if you turned 90 degrees from that starting position. Thus, `RightFrom(North)` should return `East`. Because the directions appear in order as you move right around the compass points, turning right corresponds arithmetically to adding one to the underlying value, except for `RightFrom(West)`, which has to generate 0 instead of 4 as the underlying value. As is often the case with enumerated types that represent a value which is logically cyclical, you can use the `%` operator to write a one-line implementation of `RightFrom`, as follows:

```
directionT RightFrom(directionT dir) {  
    return directionT((dir + 1) % 4);  
}
```

C++ allows implicit conversion from enumeration type to integer, since every enumeration value has a corresponding integer representation. However, there is no implicit conversion in the other direction because most integer values do not have a representation in a particular enumeration. In the above example, the enumeration type is automatically converted to an integer when used in an arithmetic expression. Once you have computed the resulting integer value, you must use an explicit typecast to return that value as a `directionT`.

You can substitute scalar types in any context in which an integer might appear. For example, a variable of an enumeration type can be used as the control expression in a `switch` statement, so that you can define a function `DirectionName(dir)`, which returns the name of a direction as a string, like this:


```

string DirectionName(directionT dir) {
    switch (dir) {
        case North: return "North";
        case East:  return "East";
        case South: return "South";
        case West:  return "West";
        default:    Error("Illegal direction value");
    }
}

```

You can also use scalar types as index variables in **for** loops. For example, you can cycle through each of the four directions using the following loop control line:

```
for (directionT dir = North; dir <= West; dir = directionT(dir+1))
```

2.2 Data and memory

Before you can understand C++'s type system in any detail, you need to know how information is stored inside a computer. Every modern computer contains some amount of high-speed internal memory that is its principal repository for information. In a typical machine, that memory is built out of a special integrated-circuit chip called a **RAM**, which stands for *random-access memory*. Random-access memory allows the program to use the contents of any memory cell at any time. The technical details of how the RAM chip operates are not important to most programmers. What is important is how the memory is organized.

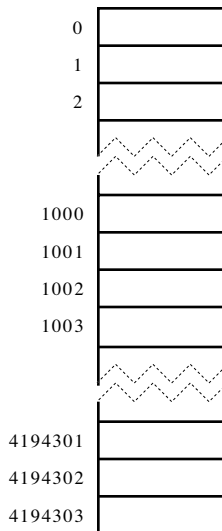
Bits, bytes, and words

At the most primitive level, all data values inside the computer are stored in the form of fundamental units of information called *bits*. A **bit** is a unit of memory that has only two possible states. If you think of the circuitry inside the machine as if it were a tiny light switch, you might label those states as *on* and *off*. Historically, the word *bit* is a contraction of *binary digit*, and it is therefore more common to label those states using the symbols 0 and 1, which are the two digits used in the binary number system on which computer arithmetic is based.

Since a single bit holds so little information, the bits themselves do not provide a particularly convenient mechanism for storing data. To make it easier to store such traditional types of information as numbers or characters, individual bits are collected together into larger units that are then treated as integral units of storage. The smallest such combined unit is called a **byte** and is large enough to hold a value of type **char**, which typically requires eight individual bits. On most machines, bytes are assembled into larger structures called **words**, where a word is usually defined to be the size required to hold a value of type **int**. Some machines use two-byte words (16 bits), some use four-byte words (32 bits), and some use less conventional sizes.

Memory addresses

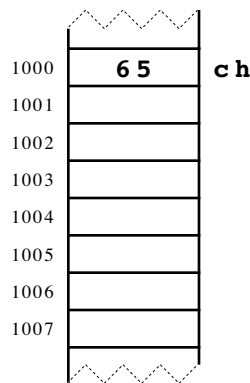
Within the memory system, every byte is identified by a numeric **address**. Typically, the first byte in the computer is numbered 0, the second is numbered 1, and so on, up to the number of bytes in the machine. For example, if your computer has four megabytes of memory (which actually means 4×2^{20} or 4,194,304 bytes), the addresses of the memory cells would look like this:



Each byte of memory is large enough to hold one character. For example, if you were to declare the character variable **ch**, the compiler would reserve one byte of storage for that variable as part of the current function frame. Suppose that this byte happened to be at address 1000. If the program then executed the statement

```
ch = 'A';
```

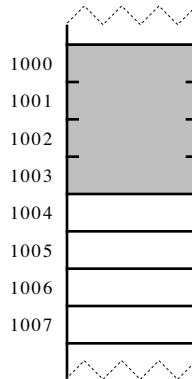
the internal representation of the character **'A'** would be stored in location 1000. Since the ASCII code for **'A'** is 65, the resulting memory configuration would look like this:



In most programming applications, you will have no way of predicting the actual address at which a particular variable is stored. In the preceding diagram, the variable **ch** is assigned to address 1000, but this choice is entirely arbitrary. Whenever your program makes a function call, the variables within the function are assigned to memory locations, but you have no way of predicting the addresses of those variables in advance. Even so, you may find it useful to draw pictures of memory and label the individual locations with addresses beginning at a particular starting point. These addresses—even though you choose them yourself—can help you to visualize what is happening inside the memory of the computer as your program runs.

Values that are larger than a single character are stored in consecutive bytes of memory. For example, if an integer takes up four bytes on your computer, that integer

requires four consecutive bytes of memory and might therefore be stored in the shaded area in the following diagram:



Data values requiring multiple bytes are identified by the address of the first byte, so the integer represented by the shaded area is the word stored at address 1000.

When you write a C++ program, you can determine how much memory will be assigned to a particular variable by using the `sizeof` operator. The `sizeof` operator takes a single operand, which must be a type name enclosed in parentheses or an expression. If the operand is a type, the `sizeof` operator returns the number of bytes required to store a value of that type; if the operand is an expression, `sizeof` returns the number of bytes required to store the value of that expression. For example, the expression

```
sizeof(int)
```

returns the number of bytes required to store a value of type `int`. The expression

```
sizeof x
```

returns the number of bytes required to store the variable `x`.

2.3 Pointers

One of the principles behind the design of C++ was that programmers should have as much access as possible to the facilities provided by the hardware itself. For this reason, C++ makes the fact that memory locations have addresses visible to the programmer. A data item whose value is an address in memory is called a **pointer**. In many high-level programming languages, pointers are used sparingly because those languages provide other mechanisms that eliminate much of the need for pointers; the Java programming language, for example, hides pointers from the programmer altogether. In C++, pointers are pervasive, and it is impossible to understand C++ programs without knowing how pointers work.

In C++, pointers serve several purposes, of which the following are the most important:

- *Pointers allow you to refer to a large data structure in a compact way.* Data structures in a program can become arbitrarily large. No matter how large they grow, however, the data structures still reside somewhere in the computer's memory and therefore have an address. Pointers allow you to use the address as a shorthand for the complete

value. Because a memory address typically fits in four bytes of memory, this strategy offers considerable space savings when the data structures themselves are large.

- *Pointers make it possible to reserve new memory during program execution.* Up to now, the only memory you could use in your programs was the memory assigned to variables that you have declared explicitly. In many applications, it is convenient to acquire new memory as the program runs and to refer to that memory using pointers. This strategy is discussed in the section on “Dynamic allocation” later in this chapter.
- *Pointers can be used to record relationships among data items.* In advanced programming applications, pointers are used extensively to model connections between individual data values. For example, programmers often indicate that one data item follows another in a conceptual sequence by including a pointer to the second item in the internal representation of the first.

Using addresses as data values

In C++, any expression that refers to an internal memory location capable of storing data is called an **lvalue** (pronounced “ell-value”). The *l* at the beginning of *lvalue* comes from the observation that lvalues can appear on the left side of an assignment statement in C++. For example, simple variables are lvalues because you can write a statement like

```
x = 1.0;
```

Many values in C++, however, are not lvalues. For example, constants are not lvalues because a constant cannot be changed. Similarly, although the result of an arithmetic expression is a value, it is not an lvalue, because you cannot assign a value to the result of an arithmetic expression.

The following properties apply to lvalues in C++:

- Every lvalue is stored somewhere in memory and therefore has an address.
- Once it has been declared, the address of an lvalue never changes, even though the contents of the lvalue may change.
- Depending on their data type, different lvalues require different amounts of memory.
- The address of an lvalue is a pointer value, which can be stored in memory and manipulated as data.

Declaring pointer variables

As with all other variables in C++, you must declare pointer variables before you use them. To declare a variable as a pointer, you precede its name with an asterisk (*) in a standard declaration. For example, the line

```
int *p;
```

declares the variable **p** to be of the conceptual type pointer-to-**int**. Similarly, the line

```
char *cptr;
```

declares the variable **cptr** to be of type pointer-to-**char**. These two types—pointer-to-**int** and pointer-to-**char**—are distinct in C++, even though each of them is represented internally as an address. To use the value at that address, the compiler needs to know how to interpret it and therefore requires that its type be specified explicitly. The type of the value to which a pointer points is called the **base type** of that pointer. Thus, the type pointer-to-**int** has **int** as its base type.

It is important to note that the asterisk used to indicate that a variable is a pointer belongs syntactically with the variable name and not with the base type. If you use the same declaration to declare two pointers of the same type, you need to mark each of the variables with an asterisk, as in

```
int *p1, *p2;
```

The declaration

```
int *p1, p2;
```

declares **p1** as a pointer to an integer, but declares **p2** as an integer variable.

The fundamental pointer operations

C++ defines two operators that allow you to move back and forth between values and pointers to those values:

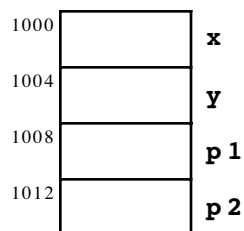
```
&      Address-of
*      Value-pointed-to
```

The **&** operator takes an lvalue as its operand and returns the memory address in which that lvalue is stored. The ***** operator takes a value of any pointer type and returns the lvalue to which it points. This operation is called **dereferencing** the pointer. The ***** operation produces an lvalue, which means that you can assign a value to a dereferenced pointer.

The easiest way to illustrate these operators is by example. Consider the declarations

```
int x, y;
int *p1, *p2;
```

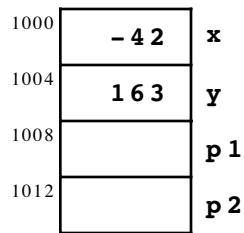
These declarations allocate memory for four words, two of type **int** and two of type pointer-to-**int**. For concreteness, let's suppose that these values are stored in the machine addresses indicated by the following diagram:



You can assign values to **x** and **y** using assignment statements, just as you always have. For example, executing the assignment statements

```
x = -42;
y = 163;
```

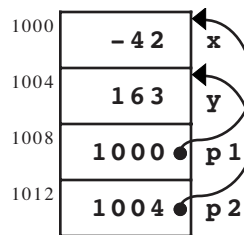
results in the following memory state:



To initialize the pointer variables **p1** and **p2**, you need to assign values that represent the addresses of some integer objects. In C++, the operator that produces addresses is the **&** operator, which you can use to assign the addresses of **x** and **y** to **p1** and **p2**, respectively:

```
p1 = &x;
p2 = &y;
```

These assignments leave memory in the following state:



The arrows in the diagram are used to emphasize the fact that the values of the variables **p1** and **p2** point to the cells indicated by the heads of the arrows. Drawing arrows makes it much easier to understand how pointers work, but it is important to remember that pointers are simply numeric addresses and that there are no arrows inside the machine.

To move from a pointer to the value it points to, you use the ***** operator. For example, the expression

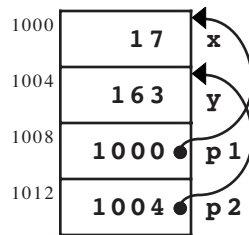
```
*p1
```

indicates the value in the memory location to which **p1** points. Moreover, since **p1** is declared as a pointer to an integer, the compiler knows that the expression ***p1** must refer to an integer. Thus, given the configuration of memory illustrated in the diagram, ***p1** turns out to be another name for the variable **x**.

Like the simple variable name **x**, the expression ***p1** is an lvalue, and you can assign new values to it. Executing the assignment statement

```
*p1 = 17;
```

changes the value in the variable **x** because that is where **p1** points. After you make this assignment, the memory configuration is

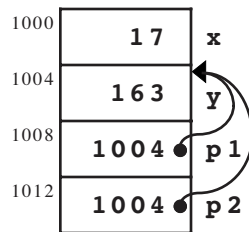


You can see that the value of **p1** itself is unaffected by this assignment. It continues to hold the value 1000 and therefore still points to the variable **x**.

It is also possible to assign new values to the pointer variables themselves. For instance, the statement

```
p1 = p2;
```

instructs the computer to take the value contained in the variable **p2** and copy it into the variable **p1**. The value contained in **p2** is the pointer value 1004. If you copy this value into **p1**, both **p1** and **p2** point to the variable **y**, as the following diagram shows:



In terms of the operations that occur inside the machine, copying a pointer is exactly the same as copying an integer. The value of the pointer is simply copied unchanged to the destination. From a conceptual perspective, the diagram shows that the effect of copying a pointer is to replace the destination pointer with a new arrow that points to the same location as the old one. Thus, the effect of the assignment

```
p1 = p2;
```

is to change the arrow leading from **p1** so that it points to the same memory address as the arrow originating at **p2**.

It is important to be able to distinguish the assignment of a pointer from that of a value. **Pointer assignment**, such as

```
p1 = p2;
```

makes **p1** and **p2** point to the same location. **Value assignment**, which is represented by the statement

```
*p1 = *p2;
```

copies the value from the memory location addressed by **p2** into the location addressed by **p1**.

The special pointer **NULL**

In many pointer applications, it is useful to be able to store in a pointer variable a special value indicating that the variable does not in fact point to any valid data, at least for the present. Such a pointer variable is called a **null pointer** and its value is assigned the address 0. The special constant called **NULL** can be used for this purpose.¹

It is important not to dereference a null pointer with the `*` operator. The intent of the null value is to indicate that the pointer does not point to valid data, so the idea of trying to find the data associated with a null pointer does not really make sense. Unfortunately, most compilers do not produce programs that explicitly check for this error. If you dereference a null pointer, many computers will interpret the 0 value as an address and return whatever data value is stored at address 0. If you happen to change that value by performing value assignment through a null pointer, the program can easily crash, giving no hint of the cause. The same problems can arise if you use pointer variables whose values have not yet been initialized.

The uses of the null pointer will be introduced in this text as they become relevant to a particular application. For now, the important thing to remember is that this constant exists.

2.4 Arrays

An **array** is a collection of individual data values with two distinguishing characteristics:

1. *An array is ordered.* You must be able to count off the individual components of an array in order: here is the first, here is the second, and so on.
2. *An array is homogeneous.* Every value stored in an array must be of the same type. Thus, you can define an array of integers or an array of floating-point numbers but not an array in which the two types are mixed.

From an intuitive point of view, it is best to think of an array as a sequence of boxes, one box for each data value in the array. Each of the values in an array is called an **element**. For example, the following diagram represents an array with five elements:



In C++, each array has two fundamental properties:

- The **element type**, which is the type of value that can be stored in the elements of the array
- The **array size**, which is the number of elements the array contains

Whenever you create a new array in your program, you must specify both the element type and the array size.

Array declaration

Like any other variable in C++, an array must be declared before it is used. The general form for an array declaration is

```
type name[size];
```

¹ The constant **NULL** is defined in the `<cstdlib>` header file. You may also just use the constant zero.

where *type* is the type of each element in the array, *name* is the name of the array variable, and *size* is a constant indicating the number of elements allocated to the array. For example, the declaration

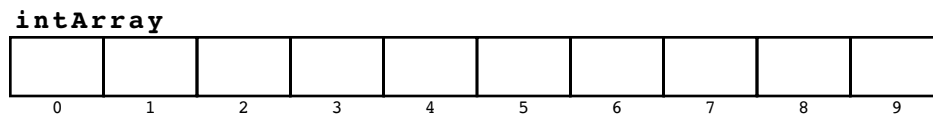
```
int intArray[10];
```

declares an array named **intArray** with 10 elements, each of which is of type **int**. In most cases, however, you should specify the size as a symbolic constant rather than an explicit integer so that the array size is easier to change. Thus, a more conventional declaration would look like this:

```
const int N_ELEMENTS = 10;
```

```
int intArray[N_ELEMENTS];
```

You can represent this declaration pictorially by drawing a row of ten boxes and giving the entire collection the name **intArray**:



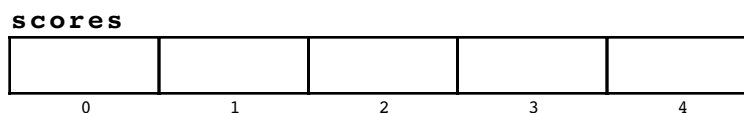
Each element in the array is identified by a numeric value called its **index**. In C++, the index numbers for an array always begin with 0 and run up to the array size minus one. Thus, in an array with 10 elements, the index numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, as the preceding diagram shows.

As is the case with any variable, you use the name of an array to indicate to other readers of the program what sort of value is being stored. For example, suppose that you wanted to define an array that was capable of holding the scores for a sporting event, such as gymnastics or figure skating, in which scores are assigned by a panel of judges. Each judge rates the performance on a scale from 0 to 10, with 10 being the highest score. Because a score may include a decimal fraction, as in 9.9, each element of the array must be of some floating-point type, such as **double**. Thus the declaration

```
const int N_JUDGES = 5;
```

```
double scores[N_JUDGES];
```

declares an array named **scores** with five elements, as shown in the following diagram:



Array selection

To refer to a specific element within an array, you specify both the array name and the index corresponding to the position of that element within the array. The process of identifying a particular element within an array is called **selection**, and is indicated in C++ by writing the name of the array and following it with the index written in square brackets. The result is a **selection expression**, which has the following form:

`array[index]`

Within a program, a selection expression acts just like a simple variable. You can use it in an expression, and, in particular, you can assign a value to it. Thus, if the first judge (judge #0, since C++ counts array elements beginning at zero) awarded the contestant a score of 9.2, you could store that score in the array by writing the assignment statement

```
scores[0] = 9.2;
```

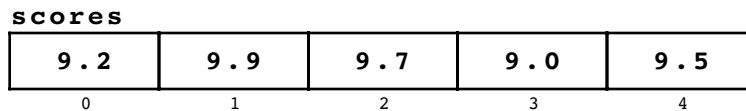
The effect of this assignment can be diagrammed as follows:



You could then go ahead and assign scores for each of the other four judges using, for example, the statements

```
scores[1] = 9.9;  
scores[2] = 9.7;  
scores[3] = 9.0;  
scores[4] = 9.5;
```

Executing these statements results in the following picture:



In working with arrays, it is essential to understand the distinction between the *index* of an array element and the *value* of that element. For instance, the first box in the array has index 0, and its value is 9.2. It is also important to remember that you can change the values in an array but never the index numbers.

The real power of array selection comes from the fact that the index value need not be constant, but can be any expression that evaluates to an integer or any other scalar type. In many cases, the selection expression is the index variable of a **for** loop, which makes it easy to perform an operation on each element of the array in turn. For example, you can set each element in the **scores** array to 0.0 with the following statement:

```
for (int i = 0; i < N_JUDGES; i++) {  
    scores[i] = 0.0;  
}
```

Effective and allocated sizes

At the time you write a program, you often will not know exactly how many elements an array will contain. The number of array elements usually depends on the user's data. Some users may require large amounts of array storage, while others need less. Unfortunately, you can't simply declare an array as

```
int dataValues[n];
```



Array bounds must be constant.

where **n** is a variable whose value changes in response to the needs of the application. C++ requires that arrays be declared with a constant size.

The usual strategy for solving this problem is to declare an array that is larger than you need and use only part of it. Thus, instead of declaring the array based on the *actual* number of elements—which you often do not know in advance—you define a constant indicating the *maximum* number of elements and use that constant in the declaration of the array. On any given use of the program, the actual number of elements is always less than or equal to this bound. When you use this strategy, you need to maintain a separate integer variable that keeps track of the number of values that are actually in use. The size of the array specified in the declaration is called the **allocated size**; the number of elements actively in use is called the **effective size**.

As an example, suppose that you wanted to change the declaration of the array **scores** introduced in the preceding section so that the program would work with any reasonable number of judges. Since you can't imagine that the number of judges at a sports event would ever be larger than 100, you might declare the array like this:

```
const int MAX_JUDGES = 100;

int scores[MAX_JUDGES];
```

To keep track of the effective size, you would need to declare an additional variable, as follows:

```
int nJudges;
```

Passing arrays as parameters

Functions in C++ can take entire arrays as parameters. When they do, it is common—particularly if the allocated and effective sizes might be different—to omit the maximum bound in the parameter declaration and use empty brackets instead. For example, the following function takes an array of type **double** and an integer indicating the effective size of the array and returns the **mean**, or arithmetic average, of the elements in the array:

```
double Mean(double array[], int n) {
    double total = 0;

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

When a function takes an array argument, the value of that argument is not copied in the way that simple variables are. Instead, the function always gets a pointer to the array, which means that the storage used for the parameter array is shared with that of the actual argument. Changing the value of an element of the parameter array therefore changes the value of the corresponding element in the argument array.

The use of arrays as parameters is illustrated by the **gymjudge.cpp** program shown in Figure 2-1, which asks the user to enter the score for each judge and then displays the average score. Note that the **ReadAllScores** function depends on the fact that arrays are passed as pointers. The whole point of the function is to fill up the elements in the array **scores**. If **ReadAllScores** were unable to make changes to the elements of the calling array, you would be forced to seek a different implementation strategy.

Initialization of arrays

Array variables can be given initial values at the time they are declared. In this case, the equal sign specifying the initial value is followed by a list of initializers enclosed in curly braces. For example, the declaration

```
int digits[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

declares an array called **digits** in which each of the 10 elements is initialized to its own index number. When initializers are provided for an array, it is legal to omit the array size from the declaration. Thus, you could also write the declaration of **digits** like this:

```
int digits[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

When the compiler encounters a declaration of this form, it counts the number of initializers and reserves exactly that many elements for the array.

In the **digits** example, there is little advantage in leaving out the array bound. You know that there are 10 digits and that new digits are not going to be added to this list. For arrays whose initial values may need to change over the life cycle of the program, having the compiler compute the array size from the initializers is useful for program maintenance because it frees the programmer from having to maintain the element count as the program evolves.

For example, imagine you're writing a program that requires an array containing the names of all U.S. cities with populations of over 1,000,000. Taking data from the 1990 census, you could declare and initialize **bigCities** as a global array using the following declaration:

```
string bigCities[] = {  
    "New York",  
    "Los Angeles",  
    "Chicago",  
    "Houston",  
    "Philadelphia",  
    "San Diego",  
    "Detroit",  
    "Dallas",  
};
```

When the figures are in from the 2000 census, it is likely that Phoenix and San Antonio will have joined this list. If they have, you can then simply add their names to the initializer list. The compiler will expand the array size to accommodate the new values. Note that a comma follows the last initializer for the **bigCities** array. This comma is optional, but it is good programming practice to include it. Doing so allows you to add new cities without having to change the existing entries in the initializer list.

If you write a program that uses the **bigCities** array, you will probably need to know how many cities the list contains. The compiler has this number because it counted the

Figure 2-1 Program to average a set of scores

```
/*
 * File: gymjudge.cpp
 * -----
 * This program averages a set of gymnastic scores.
 */

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

/* Constants */

const int MAX_JUDGES = 100;
const double MIN_SCORE = 0.0;
const double MAX_SCORE = 10.0;

/* Private function prototypes */

void ReadAllScores(double scores[], int nJudges);
double GetScore(int judge);
double Mean(double array[], int n);

/* Main program */

int main() {
    double scores[MAX_JUDGES];
    cout << "Enter number of judges: ";
    int nJudges = GetInteger();
    if (nJudges > MAX_JUDGES) Error("Too many judges");
    ReadAllScores(scores, nJudges);
    cout << "The average score is " << Mean(scores, nJudges) << endl;
    return 0;
}

/*
 * Function: ReadAllScores
 * Usage: ReadAllScores(scores, nJudges);
 * -----
 * This function reads in scores for each of the judges. The
 * array scores must be declared by the caller and must have
 * an allocated size that is at least as large as nJudges.
 * Because people tend to count starting at 1 rather than 0,
 * this program adds 1 to the argument to GetScore, which means
 * that the values the user sees will range from 1 to n instead
 * of between 0 and n-1.
 */

void ReadAllScores(double scores[], int nJudges) {
    for (int i = 0; i < nJudges; i++) {
        scores[i] = GetScore(i + 1);
    }
}
```

```

/*
 * Function: GetScore
 * Usage: score = GetScore(judge);
 * -----
 * This function reads in the score for the specified judge.
 * The implementation makes sure that the score is in the
 * legal range before returning.
 */

double GetScore(int judge) {
    while (true) {
        cout << "Score for judge #" << judge << ": ";
        double score = GetReal();
        if (score >= MIN_SCORE && score <= MAX_SCORE) return score;
        cout << "That score is out of range. Try again." << endl;
    }
}

/*
 * Function: Mean
 * Usage: mean = Mean(array, n);
 * -----
 * This function returns the statistical mean (average) of a
 * distribution stored in array, which has effective size n.
 */

double Mean(double array[], int n) {
    double total = 0;
    for (int i = 0; i < n; i++) {
        total += array[i];
    }
    return total / n;
}

```

initializers. The question is how to make that information available to the program. In C++, there is a standard idiom for determining the number of elements in an array whose size is established by static initialization. Given any array **a**, the number of elements in **a** can be computed using the expression

```
sizeof a / sizeof a[0]
```

Described in English, this expression takes the size of the entire array and divides it by the size of the initial element in the array. Because all elements of an array are the same size, the result is the number of elements in the array, regardless of the element type. Thus you could initialize a variable **nBigCities** to hold the number of cities in the **bigCities** array by writing

```
int nBigCities = sizeof bigCities / sizeof bigCities[0];
```

Multidimensional arrays

In C++, the elements of an array can be of any type. In particular, the elements of an array can themselves be arrays. Arrays of arrays are called **multidimensional arrays**. The most common form is the two-dimensional array, which is most often used to represent data in which the individual entries form a rectangular structure marked off into

rows and columns. This type of two-dimensional structure is called a **matrix**. Arrays of three or more dimensions are also legal in C++ but occur much less frequently.

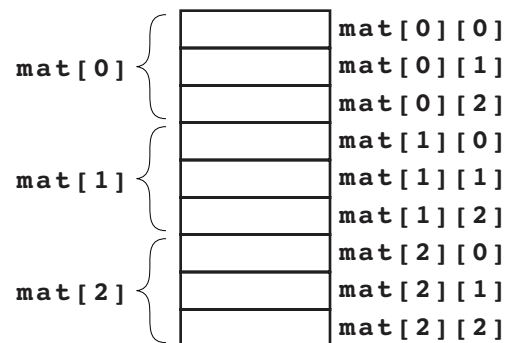
The following declaration, for example, introduces a 3×3 matrix, each of whose elements is of type **double**:

```
double mat[3][3];
```

Conceptually, the storage for **mat** forms a two-dimensional structure in which the individual elements are laid out like this:

mat[0][0]	mat[0][1]	mat[0][2]
mat[1][0]	mat[1][1]	mat[1][2]
mat[2][0]	mat[2][1]	mat[2][2]

Internally, C++ represents the variable **mat** as an array of three elements, each of which is an array of three floating-point values. The memory allocated to **mat** consists of nine cells arranged in the following form:



In the two-dimensional diagram, the first index is assumed to indicate the row number. This choice, however, is arbitrary because the two-dimensional geometry of the matrix is entirely conceptual; in memory, these values form a one-dimensional list. If you want the first index to indicate the column and the second to indicate the row, you do not need to change the declaration, only the way in which you select the elements. In terms of the internal arrangement, however, it is always true that the first index value varies least rapidly. Thus all the elements of **mat[0]** appear in memory before any elements of **mat[1]**.

Multidimensional arrays are passed between functions just as single-dimensional arrays are. The parameter declaration in the function header looks like the original declaration of the variable and includes the index information. C++ requires that you specify the size of each index in a parameter array, except for the first. However, because leaving out the first index bound makes the declaration unsymmetrical, it is common to include the array bounds for each index in the declaration of a multidimensional array parameter.

You can also use initializers with multidimensional arrays. To emphasize the overall structure, the values used to initialize each internal array are usually enclosed in an additional set of curly braces. For example, the declaration

```
double identityMatrix[3][3] = {
    { 1.0, 0.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 1.0 }
};
```

declares a 3 × 3 matrix of **doubles** and initializes it to contain the following values:

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

This particular matrix comes up frequently in mathematical applications and is called the **identity matrix**.

As in the case of parameters, the declaration of a statically initialized multidimensional array must specify all index bounds except possibly the first, which can be determined by counting the initializers. As was true with parameters, however, it is usually best to specify all the index bounds explicitly when you declare a multidimensional array.

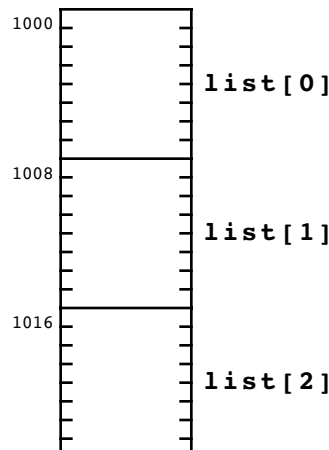
2.5 Pointers and arrays

In C++, arrays and pointers are defined in such a way that a complete understanding of either topic requires that you understand the other. Arrays, for example, are implemented internally as pointers. The operations on pointers only make sense if you consider them in relation to an array. Thus, in order to give you the full picture of how arrays and pointers work, it is important to consider the two concepts together.

To get a sense of the relationship, consider the simple array declaration

```
double list[3];
```

which reserves enough space for three values of type **double**. Assuming that a **double** is eight bytes long, the memory diagram for the array would look like this:



Because each of the elements in the array is an lvalue, it has an address that can be derived using the `&` operator. For example, the expression

```
&list[1]
```

has the pointer value 1008 because the element `list[1]` is stored at that address. Moreover, the index value need not be constant. The selection expression

```
list[i]
```

is an lvalue, and it is therefore legal to write

```
&list[i]
```

which indicates the address of the i^{th} element in `list`.

Because the address of the i^{th} element in `list` depends on the value of the variable `i`, the C++ compiler cannot compute this address when compiling the program. To determine the address, the compiler generates instructions that take the base address of the array and then add the value of `i` multiplied by the size of each array element in bytes. Thus, the numeric calculation necessary to find the address of `list[i]` is given by the formula

$$1000 + i \times 8$$

If `i` is 2, for example, the result of the address calculation is 1016, which matches the address shown in the diagram for `list[2]`. Because the process of calculating the address of an array element is entirely automatic, you don't have to worry about the details when writing your programs.

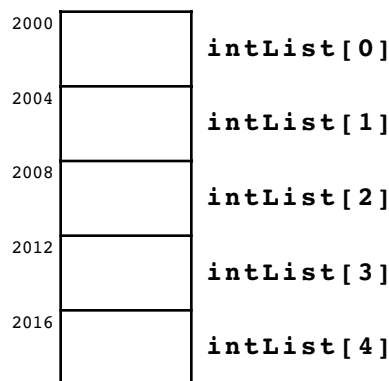
The relationship between pointers and arrays

Among the unusual characteristics of C++, one of the most interesting is that the name of an array is treated as being synonymous with a pointer to the initial element in that array. This concept is most easily illustrated by example.

The declaration

```
int intList[5];
```

allocates space for an array of five integers, which is assigned storage somewhere inside the computer's memory, as illustrated in the following diagram:



The name **intList** represents an array but can also be used directly as a pointer value. When it is used as a pointer, **intList** is defined to be the address of the initial element in the array. For any array **arr**, the following identity always holds in C++:

arr *is defined to be identical to* **&arr[0]**

Given any array name, you can assign its address directly to any pointer variable.

The most common example of this equivalence occurs when an array is passed from one function to another. For example, suppose that you make the function call

```
sum = SumIntegerArray(intList, 5);
```

where the definition of **SumIntegerArray** is

```
int SumIntegerArray(int array[], int n) {
    int sum = 0;

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

The **SumIntegerArray** function would work exactly the same way if the prototype had been written as

```
int SumIntegerArray(int *array, int n)
```

In this case, the first argument is declared as a pointer, but the effect is the same as in the preceding implementation, which declared this parameter as an array. The address of the first element in **intList** is copied into the formal parameter **array** and manipulated using pointer arithmetic. Inside the machine, the declarations are equivalent and the same operations can be applied in either case.

As a general rule, you should declare parameters in the way that reflects their use. If you intend to use a parameter as an array and select elements from it, you should declare that parameter as an array. If you intend to use the parameter as a pointer and dereference it, you should declare it as a pointer.

The crucial difference between arrays and pointers in C++ comes into play when variables are originally declared, not when those values are passed as parameters. The fundamental distinction between the declaration

```
int array[5];
```

and the declaration

```
int *p;
```

is one of memory allocation. The first declaration reserves five consecutive words of memory capable of holding the array elements. The second declaration reserves only a single word, which is large enough to hold a machine address. The implication of this difference is extremely important to you as a programmer. If you declare an array, you have storage to work with; if you declare a pointer variable, that variable is not associated with any storage until you initialize it.

Given your current level of understanding, the only way to use a pointer as an array is to initialize the pointer by assigning the base address of the array to the pointer variable. If, after making the preceding declarations, you were to write

```
p = array;
```

the pointer variable **p** would then point to the same address used for **array**, and you could use the two names interchangeably.

The technique of setting a pointer to the address of an existing array is rather limited. After all, if you already have an array name, you might as well use it. Assigning that name to a pointer does not really do you any good. The real advantage of using a pointer as an array comes from the fact that you can initialize that pointer to new memory that has not previously been allocated, which allows you to create new arrays as the program runs. This important programming technique is described in the section on “Dynamic allocation” later in this chapter.

2.6 Records

To understand the idea of a record, imagine for a moment that you are in charge of the payroll system for a small company. You need to keep track of various pieces of information about each employee. For example, in order to print a paycheck, you need to know the employee’s name, job title, Social Security number, salary, withholding status, and perhaps some additional data as well. These pieces of information, taken together, form the employee’s data record.

What do employee records look like? It is often easiest to think of records as entries in a table. For example, consider the case of the small firm of Scrooge and Marley, portrayed in Charles Dickens’s *A Christmas Carol*, as it might appear in this day of Social Security numbers and withholding allowances. The employee roster contains two records, which might have the following values:

Name	Job title	Soc. Sec. #	Salary	# With.
Ebenezer Scrooge	Partner	271-82-8183	250.00	1
Bob Cratchit	Clerk	314-15-9265	15.00	7

Each record is broken up into individual components that provide a specific piece of information about the employee. Each of these components is usually called a **field**, although the term **member** is also used, particularly in the context of C++ programming. For example, given an employee record, you can talk about the name field or the salary field. Each of the fields is associated with a type, which may be different for different fields. The name and title field are strings, the salary field might well be represented as a floating-point number, and the number of withholding exemptions is presumably an integer. The Social Security number could be represented as either an integer or a string; because Social Security numbers are too big to fit within the limits imposed on integers by many systems, they are represented here as strings.

Even though a record is made up of individual fields, it must have meaning as a coherent whole. In the example of the employee roster, the fields in the first line of the table represent a logically consistent set of data referring to Ebenezer Scrooge; those in the second line refer to Bob Cratchit. The conceptual integrity of each record suggests that the data for that employee should be collected into a compound data structure. Moreover, since the individual fields making up that structure are of different types, arrays are not suited to the task. In cases such as this, you need to define the set of data for each employee as a record.

Defining a new structure type

Creating new records in C++ is conceptually a two-step process.

1. *Define a new structure type.* Before you declare any variables, you must first define a new structure type. The type definition specifies what fields make up the record, what the names of those fields are, and what type of information each field contains. This structure type defines a model for all objects that have the new type but does not by itself reserve any storage.
2. *Declare variables of the new type.* Once you have defined the new type, you can then declare variables of that type so that you can store actual data values.

The general form for defining a new structure type looks like this:

```
struct name {
    field-declarations
};
```

where *field-declarations* are standard variable declarations used to define the fields of the structure and *name* indicates the name of the newly defined type. For example, the following code defines a new structure type called **employeeRecordT** to represent employee records:

```
struct employeeRecordT {
    string name;
    string title;
    string ssnum;
    double salary;
    int withholding;
};
```

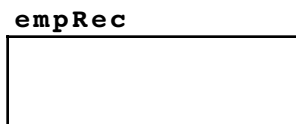
This definition provides a template for all objects that have the new type **employeeRecordT**. Each such object will have five fields, starting with a **name** field, which is a **string**, and continuing through a **withholding** field, which is an **int**.

Declaring structure variables

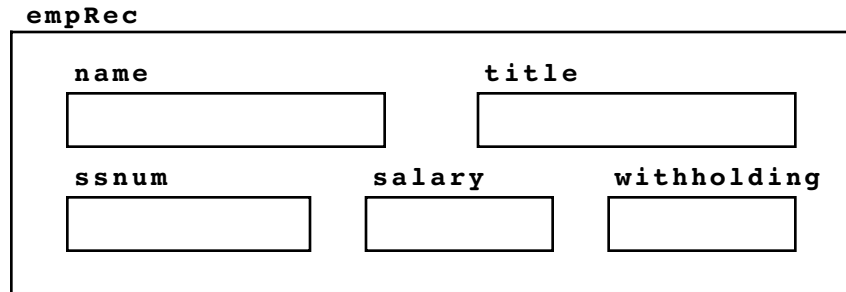
Now that you have defined a new type, the next step is to declare variables of that type. For example, given the type **employeeRecordT**, you can declare **empRec** to be a variable of that type by writing

```
employeeRecordT empRec;
```

If you want to illustrate this variable using a box diagram, you can choose to represent it in either of two ways. If you take a very general view of the situation—which corresponds conceptually to looking at the diagram from a considerable distance—what you see is just a box named **empRec**:



If, on the other hand, you step close enough to see the details, you discover that the box labeled **empRec** is composed internally of five individual boxes:



Record selection

Once you have declared the variable `empRec` by writing

```
employeeRecordT empRec;
```

you can refer to the record as a whole simply by using its name. To refer to a specific field within a record, you write the name of the complete record, followed by a period, followed by the name of the field. Thus, to refer to the job title of the employee stored in `empRec`, you need to write

```
empRec.title
```

When used in this context, the period is invariably called a *dot*, so that you would read this expression aloud as “`empRec` dot `title`.” Selecting a field using the dot operator is called **record selection**.

Initializing records

As with any other type of variable, you can initialize the contents of a record variable by assigning values to its components. The dot operator returns an lvalue, which means that you can assign values to a record selection expression. For example, if you were to execute the statements

```
empRec.name = "Ebenezer Scrooge";
empRec.title = "Partner";
empRec.ssnum = "271-82-8183";
empRec.salary = 250.00;
empRec.withholding = 1;
```

you would create the employee record for Ebenezer Scrooge used in the earlier examples.

You can also initialize its contents at the time the record is declared, using much the same syntax as you use to initialize the elements of an array. Initializers for a record are specified in the order in which they appear in the structure definition. Thus, you could declare and initialize a record named `manager` that contains the data for Mr. Scrooge, as follows:

```
employeeRecordT manager = {
    "Ebenezer Scrooge", "Partner", "271-82-8183", 250.00, 1
};
```

Pointers to records

Although small records are sometimes used directly in C++, variables that hold structured data in C++ are often declared to be pointers to records rather than the records

themselves. A pointer to a record is usually smaller and more easily manipulated than the record itself.

Suppose, for example, that you want to declare a variable that points to employee records. The syntax for such a declaration is the same as that for any other pointer. The line

```
employeeRecordT *empPtr;
```

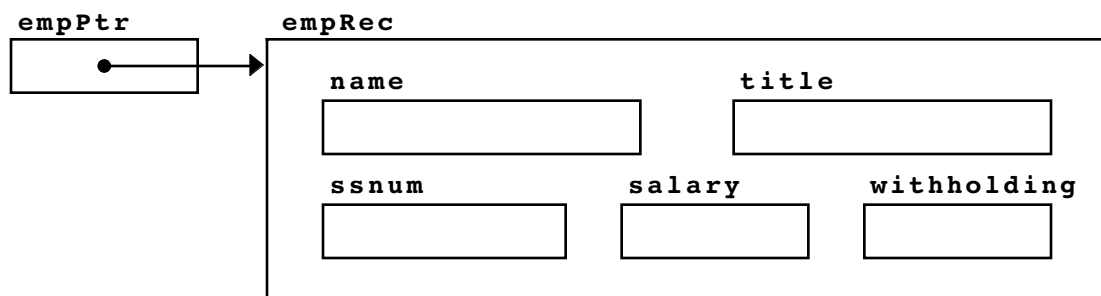
declares the variable **empPtr** as a pointer to an object of type **employeeRecordT**. When you make this declaration, space is reserved only for the pointer itself. Before using **empPtr**, you still need to provide the actual storage for the fields in the complete record. The best approach is to allocate space for a new record as described in the next section, on “Dynamic allocation.” For the moment, let’s assume that the earlier declaration

```
employeeRecordT empRec;
```

is still around, which means that you can make **empPtr** point to the **empRec** record by writing

```
empPtr = &empRec;
```

The conceptual picture of memory now looks like this:



Starting from **empPtr**, how can you refer to an individual field in the underlying record? In seeking an answer to this question, it is easy to be misled by your intuition. It is *not* appropriate to write, for example,

```
*empPtr.salary
```



The order of operations is incorrect.

Contrary to what you might have expected, this statement does not select the salary component of the object to which **empPtr** points, because the precedence of the operators in the expression does not support that interpretation. The selection operator takes precedence over dereferencing, so the expression has the meaningless interpretation

```
*(empPtr.salary)
```

rather than the intended

```
(*empPtr).salary
```

The latter form has the desired effect but is much too cumbersome for everyday use. Pointers to structures are used all the time. Forcing programmers to include parentheses in every selection would make records much less convenient. For this reason, C++ defines the operator `->`, which combines the operations of dereference and selection into a single operator. Thus, the conventional way to refer to the salary in the record to which `empPtr` points is to write

```
empPtr->salary
```

2.7 Dynamic allocation

Up to this point in the text, you have seen two mechanisms for assigning memory to variables. When you declare a global variable, the compiler allocates memory space for that variable which persists throughout the entire program. This style of allocation is called **static allocation** because the variables are assigned to fixed locations in memory. When you declare a local variable inside a function, the space for that variable is allocated on the system stack. Calling the function assigns memory to the variable; that memory is freed when the function returns. This style of allocation is called **automatic allocation**. There is also a third way of allocating memory that permits you to acquire new memory when you need it and to free it explicitly when it is no longer needed. The process of acquiring new storage while the program is running is called **dynamic allocation**.

When a program is loaded into memory, it usually occupies only a fraction of the available storage. In most systems, you can allocate some of the unused storage to the program whenever it needs more memory. For example, if you need space for a new array while the program is running, you can reserve part of the unallocated memory, leaving the rest for subsequent allocations. The pool of unallocated memory available to a program is called the **heap**.

In C++, you use the **new** operator to allocate memory from the heap. In its simplest form, the **new** operator takes a type and allocates space for a variable of that type located in the heap. For example, if you want to allocate an integer in the heap, you call

```
int *ip = new int;
```

The call to **new** operator will return the address of a storage location in the heap that has been set aside to hold an integer.

The **new** operator can also be used to allocate variables of compound type. To allocate an employee record in the heap, you could use the call:

```
employeeRecordT *empPtr = new employeeRecordT;
```

The **new[]** operator is a variant that is used to allocate an array in the heap. Within the square brackets, you specify the number of array elements, as shown in these examples:

```
int *intList = new int[12];  
employeeRecordT *empList = new employeeRecordT[1000];
```

The address returned by **new[]** is the base address of a contiguous piece of memory large enough for the entire array. You can index into dynamic array using ordinary subscript notation just as you would for a static array.

Coping with memory limitations

Although they are getting larger all the time, computer memory systems are finite in size. As a result, the heap will eventually run out of space. When this occurs, the **new** operator will eventually be unable to allocate a block of the requested size. The failure to fulfill an allocation request is such a serious error that the default behavior is for the program to immediately halt.

One way to help ensure that you don't run out of memory is to free any storage you have allocated when you are finished using it. C++ supplies the operator **delete**, which takes a pointer previously allocated by **new** and returns the memory associated with that pointer to the heap. If, for example, you determine that you are completely finished using the storage allocated for **ptr**, you can free that storage for later reuse by calling

```
delete ptr;
```

The **delete[]** operator is used to free storage that was allocated using the **new[]** operator. You do not indicate the number of elements in the square brackets when using **delete[]**.

```
int *arr = new int[45];  
...  
delete[] arr;
```

Knowing when to free a piece of memory is not always an easy task, particularly as programs become large. If several parts of a program share some data structure that has been allocated in the heap, it may not be possible for any single part to recognize that the memory can be freed. Given the size of memories today, however, you can often allocate whatever memory you need without ever bothering to free it again. The problem of limited memory typically becomes critical only when you design an application that needs to run for a long period of time, such as the operating system on which all the other facilities of the system depend. In these applications, it is important to free memory when you no longer need it.

Some languages, including Java, support a system for dynamic allocation that actively goes through memory to see what parts of it are in use, freeing any storage that is no longer needed. This strategy is called **garbage collection**. Garbage-collecting allocators exist for C++, and it is likely that their use will increase in coming years, particularly as people become more familiar with their advantages. If it does, the policy of ignoring deallocation will become reasonable even in long-running applications because you can rely on the garbage collector to perform the deallocation operations automatically.

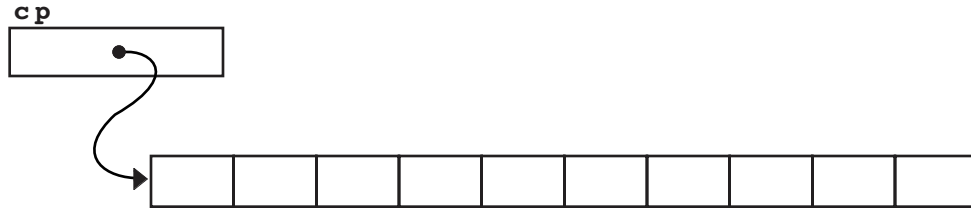
For the most part, this text assumes that your applications fall into the class of problems for which allocating memory whenever you need it is a workable strategy. This assumption will simplify your life considerably and make it easier for you to concentrate on algorithmic details.

Dynamic arrays

From a conceptual perspective, an assignment of the form

```
cp = new char[10];
```

creates the following configuration in memory:



The variable `cp` points to a set of 10 consecutive bytes that have been allocated in the heap. Because pointers and arrays are freely interchangeable in C++, the variable now acts exactly as if it had been declared as an array of 10 characters.

Arrays that you allocate on the heap and reference using a pointer variable are called **dynamic arrays** and play a significant role in modern programming. The principal differences between declared arrays and dynamic arrays are that

- The memory associated with a declared array is allocated automatically as part of the declaration process. When the frame for the function declaring the array is created, all the elements of that array are allocated as part of the frame. In the case of a dynamic array, the actual memory is not allocated until you invoke the `new[]` operator.
- The size of a declared array must be a constant in the program. In contrast, because its memory comes from the heap, a dynamic array can be of any size. Moreover, you can adjust the size of a dynamic array according to the amount of data. If you know you need an array with precisely N elements, you can reserve just the right amount of storage.

You can allocate a dynamic array using the `new[]` operator. For example, if you wanted to initialize the variable `darray` to a dynamic array of `n` values of type `double`, you would declare `darray` using the line

```
double *darray;
```

and then execute the following code:

```
darray = new double[n];
```

Dynamic records

Dynamic memory allocation is just as useful for records as it is for arrays. If you declare a pointer to a record, you can allocate memory to store the actual data in the record by calling `new`. For example, if the type `employeeRecordT` is defined as

```
struct employeeRecordT {
    string name;
    string title;
    string ssn;
    double salary;
    int withholding;
};
```

you can assign space for a newly allocated record to the variable `empPtr` as follows:

```
employeeRecordT *empPtr;

empPtr = new employeeRecordT;
empPtr->name = "Eric S. Roberts";
```

Summary

In this chapter, you have learned how to use the data structure definition capabilities of C++ to create new data types. The data structures presented in this chapter—pointers, arrays, and records—form the foundation for abstract data types, which are presented in later in this text. The principal advantage of these structures is that you can use them to represent data in a way that reflects the real-world structure of an application. Moreover, by combining pointers, arrays, and records in the right way, you can create hierarchical structures that allow you to manage data complexity in much the same way that decomposing a large program into separate functions allows you to manage algorithmic complexity.

Important points in this chapter include:

- C++ allows you to define new atomic types by listing the elements that comprise the domain of the type. Such types are called *enumeration types* and are part of a more general class called *scalar types* that also includes characters and integers.
- Data values inside a computer are represented as collections of bits that are organized into larger structures called *bytes* and *words*. Every byte in memory is associated with a numeric address that can be used to refer to the data contained at that location.
- Addresses of data in memory are themselves data values and can be manipulated as such by a program. A data value that is the address of some other piece of data is called a *pointer*. Pointer variables are declared in C++ by writing an asterisk in front of the variable name in its declaration line.
- The fundamental operations on pointers are `&` and `*`, which indicate the address of a stored value and the value stored at a particular address, respectively.
- There is a special pointer value called `NULL`, which is used to indicate that a pointer does not refer to anything.
- An *array* is an ordered, homogeneous collection of data values composed of elements indicated by an index number. In C++, index numbers in an array always begin with 0. Arrays are declared by specifying a constant size in square brackets after the name of the array. The size specified in the declaration is called the *allocated size* of the array and is typically larger than the *effective size*, which is the number of elements actually in use.
- When an array is passed as a parameter, the elements of the array are not copied. Instead, the function is given the address of the actual array. As a result, if a function changes the values of any elements of an array passed as a parameter, those changes will be visible to the caller, which is working with the same array.
- A *record* is a heterogeneous collection of data values that forms a logically consistent unit.
- The storage for arrays and records can be allocated dynamically from a pool of unused memory called the *heap*. The standard operators `new` and `new[]` are used to allocate records and dynamic arrays, respectively.

Review questions

1. Define each of the following terms: *pointer*, *array*, *record*.
2. What type definition would you use to define a new enumeration type `POLYGON` consisting of the following elements: `Triangle`, `Square`, `Pentagon`, `Hexagon`, `Octagon`? How would you change the definition so that internal representation for each constant name corresponded to the number of sides for that polygon?

3. What three advantages are cited in the text for defining a new enumeration type as opposed to defining named constants to achieve an equivalent effect?
4. True or false: In C++, you may apply any operation defined on integers to values of any scalar type.
5. At first glance, the following function looks very much like **RightFrom**, which is defined in the section on “Scalar types”:

```
directionT LeftFrom(directionT dir) {
    return directionT((dir - 1) % 4);
}
```



The **RightFrom** implementation works fine, but this one has a small bug. Identify the problem, and then rewrite the function so that it correctly calculates the compass direction that is 90 degrees to the left of **dir**.

6. In your own words, explain what the keyword **struct** does.
7. Define the following terms: *bit*, *byte*, *word*, *address*.
8. True or false: In C++, a value of type **int** always requires four bytes of memory.
9. True or false: In C++, a value of type **char** always requires one byte of memory.
10. What is the purpose of the **sizeof** operator? How do you use it?
11. What reasons for using pointers are cited in this chapter?
12. What is an lvalue?
13. What are the types of the variables introduced by the following declaration:

```
int * p1, p2;
```

14. What declaration would you use to declare a variable named **pFlag** as a pointer to a Boolean value?
15. What are the two fundamental pointer operations? Which one corresponds to the term *dereferencing*?
16. Explain the difference between pointer assignment and value assignment.
17. Draw diagrams showing the contents of memory after each line of the following code:

```
v1 = 10; v2 = 25; p1 = &v1; p2 = &v2;
*p1 += *p2;
p2 = p1;
*p2 = *p1 + *p2;
```

18. What is the internal representation of the constant **NULL**?
19. What does the phrase *call by reference* mean?

20. Write array declarations for the following array variables:

- An array **realArray** consisting of 100 floating-point values
- An array **inUse** consisting of 16 Boolean values
- An array **lines** that can hold up to 1000 strings

Remember that the upper bounds for these arrays should be defined as constants to make them easier to change.

21. Write the variable declaration and **for** loop necessary to create and initialize the following integer array:

squares										
0	1	4	9	16	25	36	49	64	81	100
0	1	2	3	4	5	6	7	8	9	10

22. What is the difference between allocated size and effective size?

23. Assuming that the base address for **rectangular** is 1000 and that values of type **int** require four bytes of memory, draw a diagram that shows the address of each element in the array declared as follows:

int rectangular[2][3];

24. Write a variable declaration that you could use to record the state of a chessboard, which consists of an 8x8 array of squares, each of which may contain any one of the following symbols:

K	white king	k	black king
Q	white queen	q	black queen
R	white rook	r	black rook
B	white bishop	b	black bishop
N	white knight	n	black knight
P	white pawn	p	black pawn
—	empty square		

Explain how you could initialize this array so that it holds the standard starting position for a chess game:

r	n	b	q	k	b	n	r
p	p	p	p	p	p	p	p
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

25. Assuming that **intArray** is declared as

```
int intArray[10];
```

and that **j** is an integer variable, describe the steps the computer would take to determine the value of the following expression:

```
&intArray[j + 3];
```

26. True or false: If **arr** is declared to be an array, then the expressions

```
arr
```

and

```
&arr[0]
```

are equivalent.

27. Assume that variables of type **double** take up eight bytes on the computer system you are using. If the base address of the array **doubleArray** is 1000, what is the address value of **doubleArray + 5**?

28. Describe the distinction between the declarations

```
int array[5];
```

and

```
int *p;
```

29. What steps are necessary to declare a record variable?
30. If the variable **p** is declared as a pointer to a record that contains a field called **cost**, what is wrong with the expression

```
*p.cost
```

as a means of following the pointer from **p** to its value and then selecting the **cost** field? What expression would you write in C++ to accomplish this dereference-and-select operation?

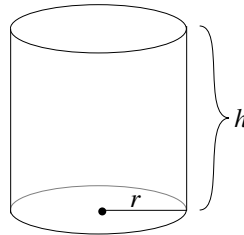
31. What is the heap?
32. Describe the effect of the following operators: **new** and **delete**.
33. What is the purpose of the **delete[]** operator?
34. What is meant by the term *garbage collection*?

Programming exercises

1. Define an enumeration type **weekdayT** whose elements are the days of the week. Write functions **NextDay** and **PreviousDay** that take a value of type **weekdayT** and return the day of the week that comes after or before the specified day, respectively.

Also write a function **IncrementDay(startDay, delta)** that returns the day of the week that comes **delta** days after **startDay**. Thus, **IncrementDay(Thursday, 4)** should return **Monday**. Your implementation of **IncrementDay** should work if the value of **delta** is negative, in which case it should proceed backward in time.

- Write a program that computes the surface area and volume of a cylinder, given the height (h) and radius of the base (r) as shown in the following diagram:



The formulas for calculating surface area and volume are

$$A = 2\pi hr$$

$$V = \pi hr^2$$

In this exercise, design your main program so that it consists of three function calls: one to read the input data, one to compute the results, and one to display the answers. When appropriate, use call by reference to communicate data between the functions and the main program.

- Because individual judges may have some bias, it is common practice to throw out the highest and lowest score before computing the average. Modify the **gymjudge.cpp** program from Figure 2-1 to discard the highest and lowest scores before computing the average score.
- Write a predicate function **IsSorted(array, n)** that takes an integer array and its effective size as parameters and returns **true** if the array is sorted in nondecreasing order.
- In the third century B.C., the Greek astronomer Eratosthenes developed an algorithm for finding all the prime numbers up to some upper limit N . To apply the algorithm, you start by writing down a list of the integers between 2 and N . For example, if N were 20, you would begin by writing down the following list:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Next you circle the first number in the list, indicating that you have found a prime. You then go through the rest of the list and cross off every multiple of the value you have just circled, since none of those multiples can be prime. Thus, after executing the first step of the algorithm, you will have circled the number 2 and crossed off every multiple of two, as follows:

② 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~

From this point, you simply repeat the process by circling the first number in the list that is neither crossed off nor circled, and then crossing off its multiples. In this

example, you would circle 3 as a prime and cross off all multiples of 3 in the rest of the list, which would result in the following state:

Eventually, every number in the list will either be circled or crossed out, as shown in this diagram:

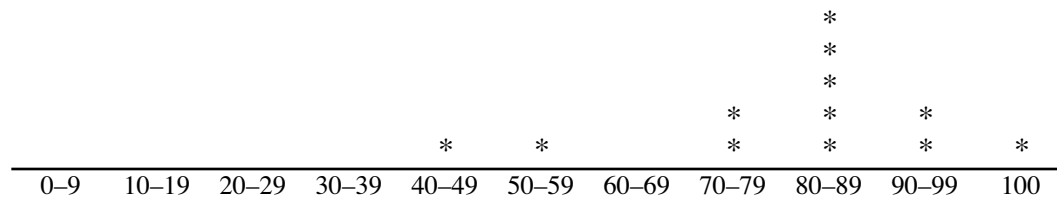
The circled numbers are the primes; the crossed-out numbers are composites. This algorithm for generating a list of primes is called the **sieve of Eratosthenes**.

Write a program that uses the sieve of Eratosthenes to generate a list of the primes between 2 and 1000.

6. A **histogram** is a graphical way of displaying data by dividing the data into separate ranges and then indicating how many data values fall into each range. For example, given the set of exam scores

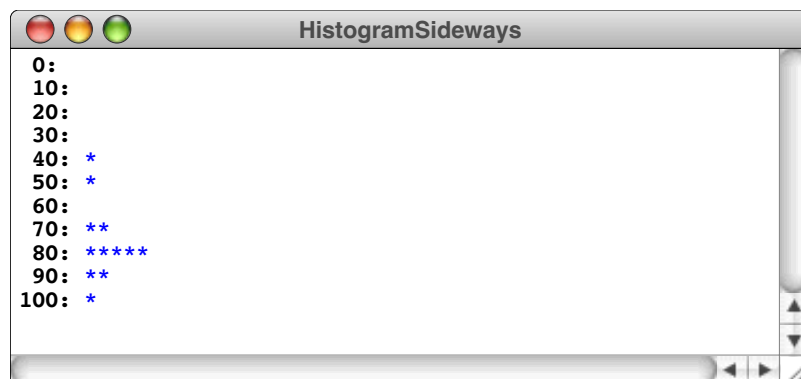
100, 95, 47, 88, 86, 92, 75, 89, 81, 70, 55, 80

a traditional histogram would have the following form:



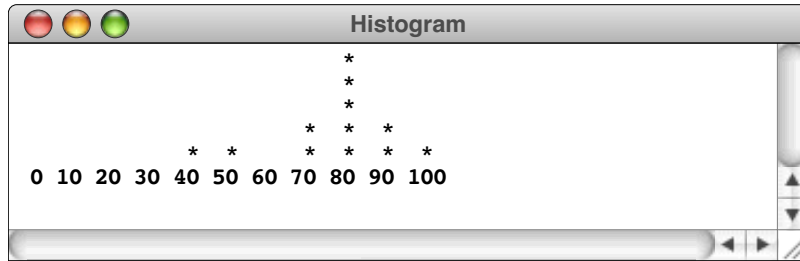
The asterisks in the histogram indicate one score in the 40s, one score in the 50s, five scores in the 80s, and so forth.

When you generate histograms using a computer, however, it is usually much easier to display them sideways on the page, as in this sample run:



Write a program that reads in an array of integers and then displays a histogram of those numbers, divided into the ranges 0–9, 10–19, 20–29, and so forth, up to the range containing only the value 100. Your program should generate output that looks as much like the sample run as possible.

7. Rewrite the histogram program from the preceding exercise so that it displays the histogram in a more traditional vertical orientation, like this:



8. Write a function **RemoveZeroElements(array, n)** that goes through an array of integers and eliminates any elements whose value is 0. Because this operation changes the effective size of the array, **RemoveZeroElements** should take the effective size as a reference parameter and adjust it accordingly. For example, suppose that **scores** contains an array of scores on an optional exam and that **nScores** indicates the effective size of the array, as shown:

nScores												
13												
scores												
65	0	95	0	0	79	82	0	84	94	86	90	0
0	1	2	3	4	5	6	7	8	9	10	11	12

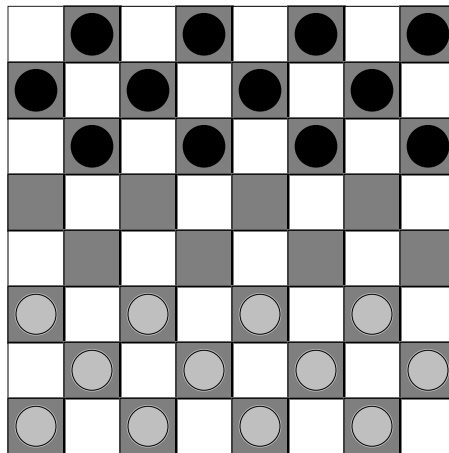
At this point, the statement

```
RemoveZeroElements(scores, nScores);
```

should remove the 0 scores, compressing the array into the following configuration:

nScores												
8												
scores												
65	95	79	82	84	94	86	90					
0	1	2	3	4	5	6	7	8	9	10	11	12

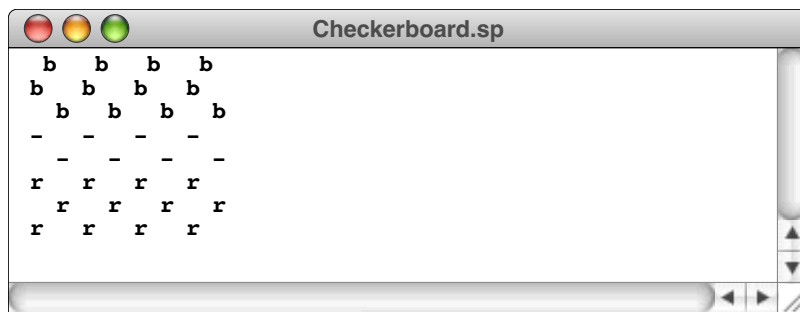
9. The initial state of a checkers game is shown in the following diagram:



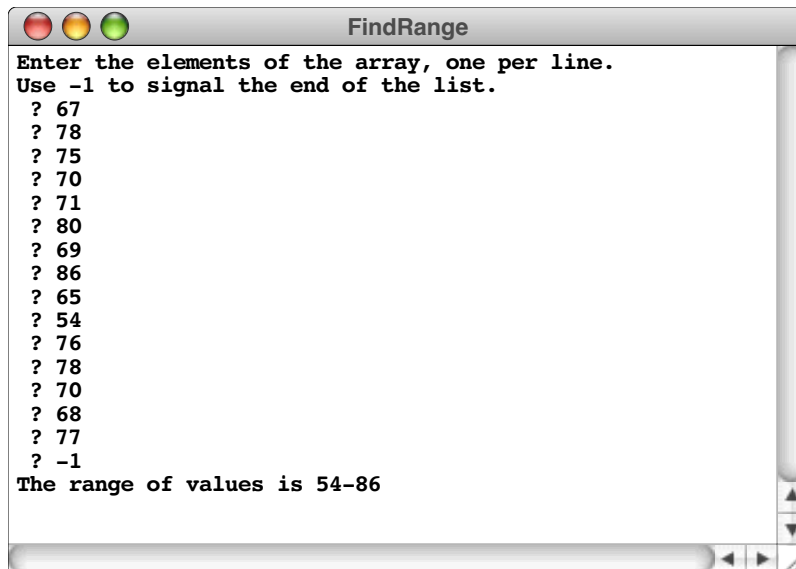
The dark squares in the bottom three rows are occupied by red checkers; the dark squares in the top three rows contain black checkers. The two center rows are unoccupied.

If you want to store the state of a checkerboard in a computer program, you need a two-dimensional array indexed by rows and columns. The elements of the array could be of various different types, but a reasonable approach is to use characters. For example, you could use the letter *r* to represent a red checker and the letter *b* to represent a black checker. Empty squares could be represented as spaces or hyphens depending on whether the color of the square was light or dark.

Implement a function **InitCheckerboard** that initializes a checkerboard array so that it corresponds to the starting position of a checkers game. Implement a second function **DisplayCheckerboard** that displays the current state of a checkerboard on the screen, as follows:



10. Design a function prototype that would allow a single function to find and return simultaneously both the lowest and highest values in an array of type **double**. Implement and test your function as shown in the following sample run:



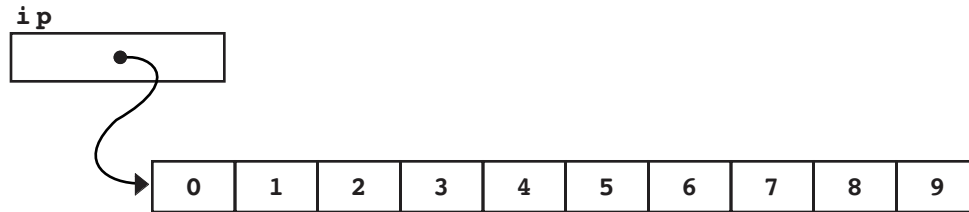
11. Write a function **IndexArray(n)** that returns a pointer to a dynamically allocated integer array with **n** elements, each of which is initialized to its own index. For example, assuming that **ip** is declared as

```
int *ip;
```

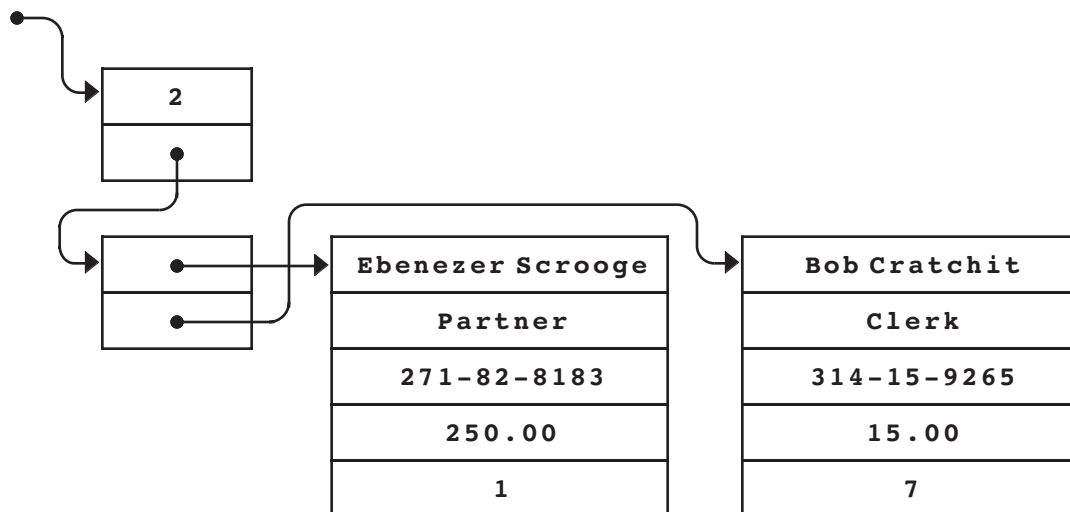
the statement

```
ip = IndexArray(10);
```

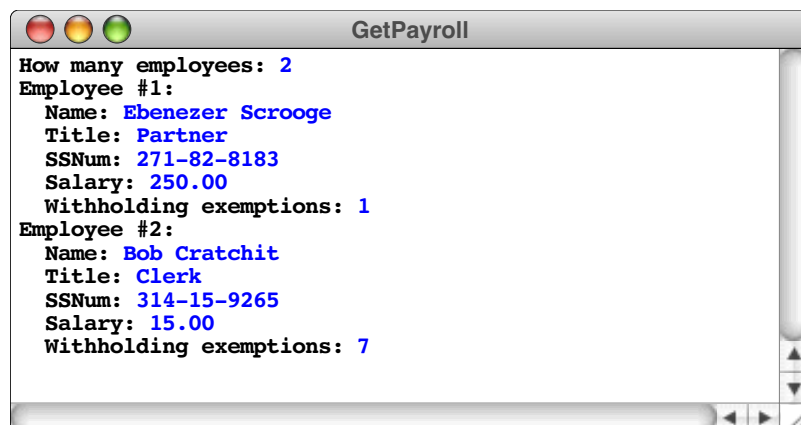
should produce the following memory configuration:



12. Design a new type called **payrollT** that is capable of holding the data for a list of employees, each of which is represented using the **employeeT** type introduced in the section on “Dynamic records” at the end of the chapter. The type **payrollT** should be a pointer type whose underlying value is a record containing the number of employees and a dynamic array of the actual **employeeT** values, as illustrated by the following data diagram:



After writing the types that define this data structure, write a function **GetPayroll** that reads in a list of employees, as shown in the following sample run:

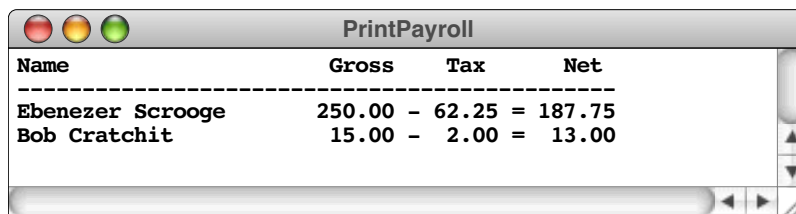


After the input values have been entered, the **GetPayroll** function should return a value of type **payrollT** that matches the structure shown in the diagram.

13. Write a program that generates the weekly payroll for a company whose employment records are stored using the type **payrollT**, as defined in the preceding exercise. Each employee is paid the salary given in the employee record, after deducting taxes. Your program should compute taxes as follows:
 - Deduct \$1 from the salary for each withholding exemption. This figure is the *adjusted income*. (If the result of the calculation is less than 0, use 0 as the adjusted income.)
 - Multiply the adjusted income by the *tax rate*, which you should assume is a flat 25 percent.

For example, Bob Cratchit has a weekly income of \$15. Because he has seven dependents, his adjusted income is $\$15 - (7 \times \$1)$, or \$8. Twenty-five percent of \$8 is \$2, so Mr. Cratchit's net pay is $\$15 - \$2$, or \$13.

The payroll listing should consist of a series of lines, one per employee, each of which contains the employee's name, gross pay, tax, and net pay. The output should be formatted in columns, as shown in the following sample run:



Name	Gross	Tax	Net
Ebenezer Scrooge	250.00	62.25	187.75
Bob Cratchit	15.00	2.00	13.00

14. Suppose that you have been assigned the task of computerizing the card catalog system for a library. As a first step, your supervisor has asked you to develop a prototype capable of storing the following information for each of 1000 books:
 - The title
 - A list of up to five authors
 - The Library of Congress catalog number
 - A list of up to five subject headings
 - The publisher
 - The year of publication
 - Whether the book is circulating or noncirculating

Design the data structures that would be necessary to keep all the information required for this prototype library database. Given your definition, it should be possible to write the declaration

```
libraryT libdata;
```

and have the variable **libdata** contain all the information you would need to keep track of up to 1000 books. Remember that the actual number of books will usually be less than this upper bound.

Write an additional procedure **SearchBySubject** that takes as parameters the library database and a subject string. For each book in the library that lists the subject string as one of its subject headings, **SearchBySubject** should display the title, the name of the first author, and the Library of Congress catalog number of the book.

15. Write a function

```
int *GetDynamicIntegerArray(int sentinel, int & n);
```

that returns a dynamically allocated array of integers read in from the user. The **sentinel** argument indicates the value used to signal the end of the input. The second argument is an integer reference parameter, which is used to return the effective size of the array. Note that it is impossible to know in advance how many values the user will enter. As a result, the implementation must allocate new array space as needed.

Chapter 3

Libraries and Interfaces

My library / Was dukedom large enough.

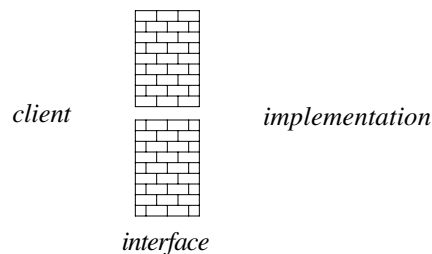
— Shakespeare, *The Tempest*, c. 1612

In modern programming, it is impossible to write interesting programs without calling library functions. In fact, as the science of programming advances, programmers depend more and more on library functions. Today, it is not at all unusual for 90 percent or more of a program to consist of library code, with only a few parts specifically tailored for a particular application. As a programmer, you must understand not only how to write new code but also how to avoid doing so by making appropriate use of existing libraries.

The main purpose of this chapter is to encourage you to think about libraries in a way that emphasizes the distinction between the library itself and other programs that make use of it, which are called its **clients**. To do so, the chapter focuses on the boundary between a library and its clients, which is called the **interface**. An interface provides a channel of communication but also acts as a barrier that prevents complex details on one side from affecting the other. Interfaces are central to a modern treatment of libraries, and it is important—even if you’ve done a lot of programming—for you to understand libraries from this perspective.

3.1 The concept of an interface

In English, the word *interface* means a common boundary between two distinct entities. The surface of a pond, for example, is the interface between the water and the air. In programming, an interface constitutes a conceptual boundary rather than a physical one: an interface is the boundary between the implementation of a library and its clients. The purpose of the interface is to provide each client with the information it needs to use the library without revealing the details required by the implementation. Thus, it is important to think of an interface not only as a channel for communication between client and implementation, but also as a barrier that keeps them separated, as illustrated by the following diagram:



By mediating the communication between the two sides, an interface reduces the conceptual complexity of the programming process by ensuring that details that lie on one side of the interface boundary do not escape to complicate the code on the other side.

Interfaces and implementations

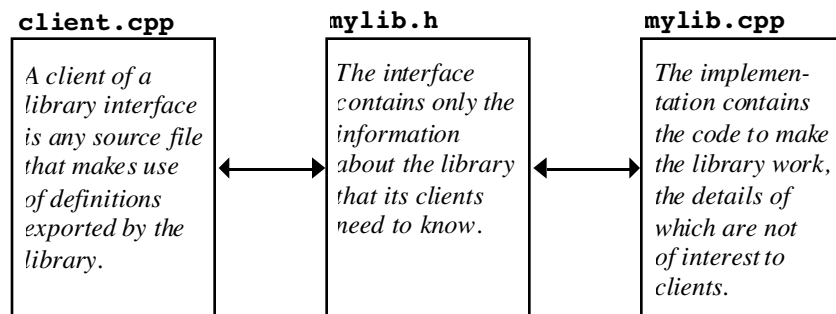
In computer science, an interface is a conceptual entity. It consists of an understanding between the programmer who implements a library and the programmer who uses it, spelling out the information that is required by both sides. When you write a C++ program, however, you must have some way to represent the conceptual interface as part of the actual program. In C++, an interface is represented by a **header file**, which traditionally has the same name as the file that implements it with the **.cpp** extension replaced by **.h**.

As an example, suppose that you have created a collection of functions that you want to make available to clients as a library. To do so, you need to create two files: an interface that you might call **mylib.h** and the corresponding implementation **mylib.cpp**.

The code for each of your functions goes in the `mylib.cpp` implementation file. The `mylib.h` interface contains only the function prototypes, which contain the information the compiler needs to interpret any calls to those functions. Putting the prototypes in the interface makes them available to clients and is called **exporting** those functions.

Although function prototypes are the most common component of an interface, interfaces can export other definitions as well. In particular, interfaces often export data types and constants. A single definition exported by an interface is called an **interface entry**.

Once you have written the interface and implementation for a library, you—or some other programmer with whom you are collaborating—can then write separate source files that act as clients of the `mylib.h` interface. The relationship between the files representing the client, interface, and implementation is illustrated in the following diagram:



The distinction between the abstract concept of an interface and the header file that represents it may at first seem subtle. In many ways, the distinction is the same as that between an algorithm and a program that implements it. The algorithm is an abstract strategy; the program is the concrete realization of that algorithm. Similarly, in C++, header files provide a concrete realization of an interface.

Packages and abstractions

The same distinction between a general concept and its programming manifestation also comes up in the definition of two other terms that are often used in discussions of interfaces. In computer science, you will often hear the term *package* used to describe the software that defines a library. If you are assigned to develop a library, part of your job consists of producing a `.h` file to serve as the library interface and a corresponding `.cpp` file that provides the underlying implementation. Those files constitute the **package**. To get a full understanding of a library, however, you must look beyond the software. Libraries embody a specific conceptual approach that transcends the package itself. The conceptual basis of a library is called an **abstraction**.

The relationship between an abstraction and a package is best illustrated by an example. The programs in the first two chapters of this text use the insertion operator and the `cout` stream from the `iostream` interface for all output operations. For input operations, however, those programs use functions exported by the `simpio.h` interface such as `GetInteger`, `GetReal`, and `GetLine`. The `iostream` interface provides functions for accepting user input, but they are more difficult for beginning programmers to use than their counterparts in `simpio.h`. The two libraries embody different approaches to input operations: the `iostream` interface emphasizes power and flexibility, while the `simpio.h` interface emphasizes simplicity of structure and ease of use. The approach used in each of these interfaces is part of the abstraction. The associated packages

implement those abstractions and make them real, in the sense that they can then be used by programmers.

Principles of good interface design

Programming is hard because programs reflect the complexity of the problems they solve. As long as we use computers to solve problems of ever-increasing sophistication, the process of programming will need to keep becoming more sophisticated as well.

Writing a program to solve a large or difficult problem forces you to manage an enormous amount of complexity. There are algorithms to design, special cases to consider, user requirements to meet, and innumerable details to get right. To make programming manageable, you must reduce the complexity of the programming process as much as possible.

One of the primary purposes of defining new functions is to reduce complexity by dividing up the entire program into more manageable pieces. Interfaces offer a similar reduction in programming complexity but at a higher level of detail. A function gives its caller access to a set of steps that together implement a single operation. An interface gives its client access to a set of functions that together implement a programming abstraction. The extent to which the interface simplifies the programming process, however, depends largely on how well it is designed.

To design an effective interface, you must balance several criteria. In general, you should try to develop interfaces that are

- *Unified*. A single interface should define a consistent abstraction with a clear unifying theme. If a function does not fit within that theme, it should be defined in a separate interface.
- *Simple*. To the extent that the underlying implementation is itself complex, the interface must hide as much of that complexity from the client as possible.
- *Sufficient*. When clients use an abstraction, the interface must provide sufficient functionality to meet their needs. If some critical operation is missing from an interface, clients may decide to abandon it and develop their own, more powerful abstraction. As important as simplicity is, the designer must avoid simplifying an interface to the point that it becomes useless.
- *General*. A well-designed interface should be flexible enough to meet the needs of many different clients. An interface that performs a narrowly defined set of operations for one client is not as useful as one that can be used in many different situations.
- *Stable*. The functions defined in an interface should continue to have precisely the same structure and effect, even if their underlying implementation changes. Making changes in the behavior of an interface forces clients to change their programs, which compromises the value of the interface.

Because it is important to maintain stable interfaces, the designer of an interface must exercise special care to get it right. As more and more clients start to depend on a particular interface, the cost of changing that interface increases. In fact, it is quite common to discover interfaces that cannot be changed at all, even if they have serious design flaws. Certain functions in the standard libraries, for example, exhibit behaviors that are widely considered flaws by the C++ programming community. Even so, it is impossible to change the design of these functions because too many clients depend on the current behavior.

Some interface changes, however, are more drastic than others. For example, adding an entirely new function to an interface is usually a relatively straightforward process, since no clients already depend on that function. Changing an interface in a way that requires no changes to existing programs is called **extending** the interface. If you find that you need to make evolutionary changes over the lifetime of an interface, it is usually best to make those changes by extension.

3.2 A random number interface

The easiest way to illustrate the structure of an interface is to provide a simple example. The libraries for this book and its C-based predecessor include an interface called **random.h** that makes it easier for client programs to simulate random processes like flipping a coin or rolling a die. Getting a computer—which ordinarily operates deterministically in the sense that running the same program with the same input always produces the same result—to behave in a random way involves a certain amount of complexity. For the benefit of client programmers, you want to hide this complexity behind an interface so that the client has access to the capabilities it needs without having to confront the complexity of the complete implementation.

The structure of the **random.h** interface

Figure 3-1 shows the **random.h** interface, which exports the functions **RandomInteger**, **RandomReal**, **RandomChance**, and **Randomize**. For each of these functions, the interface contains a one-line prototype along with a comment that describes the purpose of the function from the perspective of the client. As is typical for the interfaces defined in this text, comments comprise most of the interface and provide all the information clients need. If the comments in an interface are well designed, clients should be able to rely on them without having to read the underlying implementation.

After the initial comment that describes the interface, the **random.h** interface includes a somewhat cryptic set of lines that is part of the conventional syntax for an interface. The lines

```
#ifndef _random_h
#define _random_h
```

operate in conjunction with the last line of the interface, which is

```
#endif
```

These three lines are often referred to as **interface boilerplate**. When you design your own interfaces, you should be sure to include similar boilerplate lines, substituting the name of your own interface for the name **random** in this example.

The purpose of the interface boilerplate is to prevent the compiler from reading the same interface many times during a single compilation. The line

```
#ifndef _random_h
```

causes the compiler to skip any text up to the **#endif** line if the symbol **_random_h** has been previously defined. When the compiler reads this interface for the first time, **_random_h** is undefined, which means that the compiler goes ahead and reads the contents of the file. The compiler immediately thereafter encounters the definition

```
#define _random_h
```

Figure 3-1 The `random.h` interface

```
/*
 * File: random.h
 * -----
 * This interface provides several functions for generating
 * pseudorandom numbers.
 */

#ifndef _random_h
#define _random_h

/*
 * Function: RandomInteger
 * Usage: n = RandomInteger(low, high);
 * -----
 * This function returns a random integer in the range low to high,
 * inclusive.
 */

int RandomInteger(int low, int high);

/*
 * Function: RandomReal
 * Usage: d = RandomReal(low, high);
 * -----
 * This function returns a random real number in the half-open
 * interval [low .. high), meaning that the result is always
 * greater than or equal to low but strictly less than high.
 */

double RandomReal(double low, double high);

/*
 * Function: RandomChance
 * Usage: if (RandomChance(p)) . . .
 * -----
 * The RandomChance function returns true with the probability
 * indicated by p, which should be a floating-point number between
 * 0 (meaning never) and 1 (meaning always). For example, calling
 * RandomChance(.30) returns true 30 percent of the time.
 */

bool RandomChance(double p);

/*
 * Function: Randomize
 * Usage: Randomize();
 * -----
 * This function initializes the random-number generator so that
 * its results are unpredictable. If this function is not called,
 * the other functions will return the same values on each run.
 */

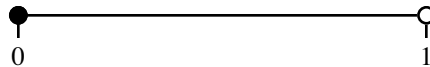
void Randomize();

#endif
```

which defines the symbol `_random_h`. If the compiler reads the `random.h` interface a second time, the symbol `_random_h` has already been defined, which means that the compiler ignores the entire contents of the file on the second pass.

The remainder of the interface consists of function prototypes and their associated comments. The first prototype is for the function `RandomInteger(low, high)`, which returns a randomly chosen integer in the range between `low` and `high`, inclusive. For example, calling `RandomInteger(1, 6)` would return 1, 2, 3, 4, 5, or 6 and could be used to simulate rolling a die. Similarly, calling `RandomInteger(0, 36)` returns an integer between 0 and 36 and could be used to model a European roulette wheel, which is marked with those 37 numbers.

The function `RandomReal(low, high)` is conceptually similar to `RandomInteger` and returns a floating-point value r subject to the condition that `low ≤ r < high`. For example, calling `RandomReal(0, 1)` returns a random number that can be as small as 0 but is always strictly less than 1. In mathematics, a range of real numbers that can be equal to one endpoint but not the other is called a **half-open interval**. On a number line, a half-open interval is marked using an open circle to show that the endpoint is excluded, like this:



In text, the standard convention is to use square brackets to indicate closed ends of intervals and parentheses to indicate open ones, so that the notation `[0, 1)` indicates the half-open interval corresponding to this diagram.

The function `RandomChance(p)` is used to simulate random events that occur with some fixed probability. To be consistent with the conventions of statistics, a probability is represented as a number between 0 and 1, where 0 means that the event never occurs and 1 means that it always does. The `RandomChance` function returns a Boolean value that is `true` with probability `p`, so that `RandomChance(0.75)` returns `true` 75 percent of the time. You can use `RandomChance` to simulate flipping a coin, as illustrated by the following function, which returns either `"heads"` or `"tails"`:

```
string FlipCoin() {
    if (RandomChance(0.50)) {
        return "heads";
    } else {
        return "tails";
    }
}
```

The last function in the `random.h` interface requires a little more explanation. Because computers are deterministic machines, random numbers are usually computed by going through a deterministic calculation that nonetheless appears random to the user. Random numbers computed in this way are called **pseudorandom numbers**. If you take no special action, the computer always applies the same process to generate its sequence of random numbers, which means that the results will be the same every time the program is run. The purpose of the `Randomize` function is to initialize the internal pseudorandom number generator so that each run of the program produces a different sequence, which is what you want if you are writing a program that plays a game.

At first, it may seem hard to understand why a random number package should return the same values on each run. After all, deterministic behavior of this sort seems to defeat

the whole purpose of the package. There is, however, a good reason behind this behavior: programs that behave deterministically are easier to debug. To illustrate this fact, suppose you have just written a program to play an intricate game, such as Monopoly™. As is always the case with newly written programs, the odds are good that your program has a few bugs. In a complex program, bugs can be relatively obscure, in the sense that they only occur in rare situations. Suppose you are playing the game and discover that the program is starting to behave in a bizarre way. As you begin to debug the program, it would be very convenient if you could regenerate the same state and take a closer look at what is going on. Unfortunately, if the program is running in a nondeterministic way, a second run of the program will behave differently from the first. Bugs that showed up the first time may not occur on the second pass.

In general, it is extremely difficult to reproduce the conditions that cause a program to fail if the program is behaving in a truly random fashion. If, on the other hand, the program is operating deterministically, it will do the same thing each time it is run. This behavior makes it possible for you to recreate the conditions under which the problem occurred. When you write a program that works with random numbers, it is usually best to leave out the call to **Randomize** during the debugging phase. When the program seems to be working well, you can insert a call to **Randomize** at the beginning of the main program to make the program change its behavior from one run to the next.

Constructing a client program

Once you know what the interface looks like, you can immediately begin to code applications that use it. The interface provides all the information that any client needs to know. From the client's perspective, the important questions are what the functions in the library do and how to call them. The details of how they work are important only for the implementation.

As a simple illustration of how clients can use the random number package, let's consider the **craps.cpp** program shown in Figure 3-2, which simulates the casino game of craps. Because the program uses the random number library to simulate rolling the dice, the **craps.cpp** file needs to include the **random.h** interface. Moreover, in order to ensure that the outcome is not the same every time, the program calls **Randomize** at the beginning of its operation. The rest of the program is simply an encoding of the rules of the game, which are outlined in the comments at the beginning of the program.

The ANSI functions for random numbers

Although you can design the **craps.cpp** program by relying on the interface description, you cannot run the program until you have an implementation for the random number library. To make sense of that implementation, you first need to understand the facilities provided by the standard libraries for generating random numbers. The basic tools are already provided in the form of a small set of functions exported by the **cstdlib** interface. Unfortunately, that interface is not well suited to the needs of clients, largely because the results returned by the functions differ from machine to machine. One of the advantages of **random.h** is that it provides clients with a machine-independent interface that is considerably easier to use than the underlying facilities on which it is based.

The **random.cpp** implementation is based on a more primitive random number facility provided as part of the standard library interface **cstdlib**. The function on which the implementation depends is called **rand** and has the following prototype:

```
int rand();
```

Figure 3-2 A program to play the game of craps

```
/*
 * File: craps.cpp
 * -----
 * This program plays the casino game called craps, which is
 * played using a pair of dice. At the beginning of the game,
 * you roll the dice and compute the total. If your first roll
 * is 7 or 11, you win with what gamblers call a "natural."
 * If your first roll is 2, 3, or 12, you lose by "crapping
 * out." In any other case, the total from the first roll
 * becomes your "point," after which you continue to roll
 * the dice until one of the following conditions occurs:
 *
 * a) You roll your point again. This is called "making
 *    your point," which wins.
 *
 * b) You roll a 7, which loses.
 *
 * Other rolls, including 2, 3, 11, and 12, have no effect
 * during this phase of the game.
 */

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

/* Function prototypes */

bool TryToMakePoint(int point);
int RollTwoDice();

/* Main program */

int main() {
    Randomize();
    cout << "This program plays a game of craps." << endl;
    int point = RollTwoDice();
    switch (point) {
        case 7: case 11:
            cout << "That's a natural. You win." << endl;
            break;
        case 2: case 3: case 12:
            cout << "That's craps. You lose." << endl;
            break;
        default:
            cout << "Your point is " << point << "." << endl;
            if (TryToMakePoint(point)) {
                cout << "You made your point. You win." << endl;
            } else {
                cout << "You rolled a seven. You lose." << endl;
            }
    }
    return 0;
}
```

```

/*
 * Function: TryToMakePoint
 * Usage: flag = TryToMakePoint(point);
 * -----
 * This function is responsible for the part of the game
 * during which you roll the dice repeatedly until you either
 * make your point or roll a 7. The function returns true if
 * you make your point and false if a 7 comes up first.
 */

bool TryToMakePoint(int point) {
    while (true) {
        int total = RollTwoDice();
        if (total == point) return true;
        if (total == 7) return false;
    }
}

/*
 * Function: RollTwoDice
 * Usage: total = RollTwoDice();
 * -----
 * This function rolls two dice and both prints and returns their sum.
 */

int RollTwoDice() {
    cout << "Rolling the dice . . ." << endl;
    int d1 = RandomInteger(1, 6);
    int d2 = RandomInteger(1, 6);
    int total = d1 + d2;
    cout << "You rolled " << d1 << " and " << d2 << " - that's "
        << total << endl;
    return total;
}

```

Unlike most functions, **rand** returns a different result each time it is called. The result of **rand** is guaranteed to be nonnegative and no larger than the constant **RAND_MAX**, which is also defined in the **cstdlib** interface. Thus, each time **rand** is called, it returns a different integer between 0 and **RAND_MAX**, inclusive.

The value of **RAND_MAX** depends on the computer system. When you write programs that work with random numbers, you should not make any assumptions about the precise value of **RAND_MAX**. Instead, your programs should be prepared to use whatever value of **RAND_MAX** the system defines.

You can get a sense of how **rand** behaves on your own system by running the program

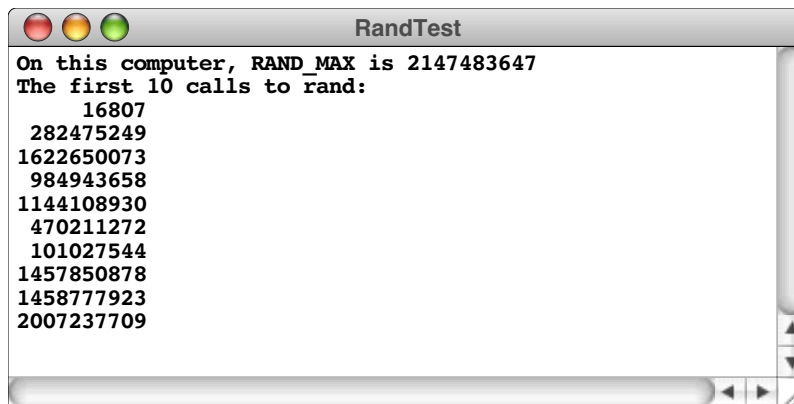
```

const int N_TRIALS = 10;

int main() {
    cout << "On this computer, RAND_MAX is " << RAND_MAX << endl;
    cout << "The first " << N_TRIALS << " calls to rand:" << endl;
    for (int i = 0; i < N_TRIALS; i++) {
        cout << setw(10) << rand() << endl;
    }
    return 0;
}

```

On the computer in my office, the program generates the following output:



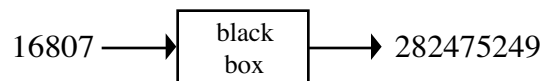
```

RandTest
On this computer, RAND_MAX is 2147483647
The first 10 calls to rand:
16807
282475249
1622650073
984943658
1144108930
470211272
101027544
1457850878
1458777923
2007237709

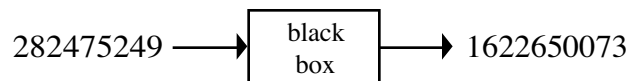
```

You can see that the program is generating integers, all of which are positive and none of which is greater than 2147483647, which the sample run shows as the value of **RAND_MAX** for this machine. Although this number itself seems rather arbitrary, it happens to be $2^{31} - 1$, which is the largest value of type **int** on my computer. Because the numbers are pseudorandom, you know that there must be some pattern, but it is unlikely that you can discern one. From your point of view, the numbers appear to be random, because you don't know what the underlying pattern is.

The **rand** function generates each new random value by applying a set of mathematical calculations to the last value it produced. Because you don't know what those calculations are, it is best to think of the entire operation as a black box where old numbers go in on one side and new pseudorandom numbers pop out on the other. Since, the first call to **rand** produces the number 16807, the second call to **rand** corresponds to putting 16807 into one end of the black box and having 282475249 appear on the other:

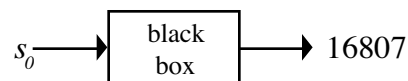


Similarly, on the next call to **rand**, the implementation puts 282475249 into the black box, which returns 1622650073:



This same process is repeated on each call to **rand**. The computation inside the black box is designed so that (1) the numbers are uniformly distributed over the legal range, and (2) the sequence goes on for a long time before it begins to repeat.

But what about the first call to **rand**—the one that returns 16838? The implementation must have a starting point. There must be an integer, s_0 , that goes into the black box and produces 16807:



This initial value—the value that is used to get the entire process started—is called a **seed** for the random number generator. The ANSI library implementation sets the initial seed

to a constant value every time a program is started, which is why the library always generates the same sequence of values. To change the sequence, you need to set the seed to a different value, which is done by calling the function `srand(seed)`.

The `srand` function is essential to the implementation of `Randomize`, which resets the seed so that the sequence of random numbers is different each time. The usual strategy is to use the value of the internal system clock as the initial seed. Because the time keeps changing, the random number sequence will change as well. You can retrieve the current value of the system clock by calling the function `time`, which is defined in the library interface `ctime`, and then converting the result to an integer. This technique allows you to write the following statement, which has the effect of initializing the pseudorandom number generator to some unpredictable point:

```
srand(int(time(NULL)));
```

Although it requires only a single line, the operation to set the random seed to an unpredictable value based on the system clock is relatively obscure. If this line were to appear in the client program, the client would have to understand the concept of a random number seed, along with the functions `srand` and `time`. To make things simpler for the client, it is much better to give this operation a simple name like `Randomize` and make it part of the random number library. By doing so, the client simply needs to call

```
Randomize();
```

which is certainly easier to explain.

The `random.cpp` implementation

The ANSI functions described in the preceding section provide all the tools you need to implement the `random.h` interface. The implementation is contained in a separate source file called `random.cpp` which is shown in Figure 3-3.

After the initial comment, the implementation file lists the `#include` files it needs. The implementation needs `cstdlib` and `ctime` so that it has access to the functions `rand`, `srand`, and `time`. Finally, every implementation needs to include its own interface so the compiler can check the prototypes against the actual definitions.

The rest of the file consists of the functions exported by the interface, along with any comments that would be useful to the programmers who may need to maintain this program in the future. Like all other forms of expository writing, comments must be written with their audience in mind. When you write comments, you must put yourself in the role of the reader so that you can understand what information that reader will want to see. Comments in the `.cpp` file have a different audience than their counterparts in the `.h` file. The comments in the implementation are written for another implementer who may have to modify the implementation in some way. They therefore must explain how the implementation works and provide any details that later maintainers would want to know. Comments in the interface, on the other hand, are written for the client. A client should never have to read the comments inside the implementation. The comments in the interface should suffice.

The use of comments to explain the operation of the code are best illustrated by the `RandomInteger` implementation, which converts a random number in the range 0 to `RAND_MAX` into one that lies in the interval between the parameters `low` and `high`. As the comments indicate, the implementation uses the following four-step process:

Figure 3-3 The implementation of the random number library

```
/*
 * File: random.cpp
 * -----
 * This file implements the random.h interface.
 */

#include <cstdlib>
#include <ctime>
#include "random.h"

/*
 * Function: RandomInteger
 * -----
 * This function begins by using rand to select an integer
 * in the interval [0, RAND_MAX] and then converts it to the
 * desired range by applying the following steps:
 *
 * 1. Normalize the value to a real number in the interval [0, 1)
 * 2. Scale the resulting value to the appropriate range size
 * 3. Truncate the scaled value to an integer
 * 4. Translate the integer to the appropriate starting point
 */

int RandomInteger(int low, int high) {
    double d = double(rand()) / (double(RAND_MAX) + 1);
    int k = int(d * (high - low + 1));
    return low + k;
}

/*
 * Function: RandomReal
 * -----
 * The implementation of RandomReal is similar to that
 * of RandomInteger, without the truncation step.
 */

double RandomReal(double low, double high) {
    double d = double(rand()) / (double(RAND_MAX) + 1);
    return low + d * (high - low);
}

/*
 * Function: RandomChance
 * -----
 * This function uses RandomReal to generate a real number
 * in the interval [0, 1) and then compares that value to p.
 */

bool RandomChance(double p) {
    return RandomReal(0, 1) < p;
}
```

```

/*
 * Function: Randomize
 * -----
 * This function operates by setting the random number
 * seed to the current time. The srand function is
 * provided by the <cstdlib> library and requires an
 * integer argument. The time function is exported by
 * the <ctime> interface.
 */

void Randomize() {
    srand(int(time(NULL)));
}

```

- *Normalization.* The first step in the process is to convert the integer result from **rand** into a floating-point number **d** in the half-open interval $[0, 1)$. To do so, all you need to do is convert the result of **rand** to a **double** and then divide it by the number of elements in the range. Because **RAND_MAX** is often the largest integer the machine can hold, it is important to convert it to a **double** before adding the constant 1, which ensures that the division produces a value that is strictly less than 1.
- *Scaling.* The second step consists of multiplying the value **d** by the size of the desired range, so that it spans the correct number of integers. Because the desired range includes both the endpoints, **low** and **high**, the number of integers in the range is given by the expression **high - low + 1**.
- *Truncation.* The third step consists of using a type cast to convert the number back to an integer by throwing away any fraction. This step gives you a random integer with a lower bound of 0.
- *Translation.* The final step consists of adding the value **low** so that the range begins at the desired lower bound.

The steps in this process are illustrated by the diagram in Figure 3-4, which shows how a call to **RandomInteger(1, 6)** converts the result of **rand** into an integer in the desired range of 1 to 6.

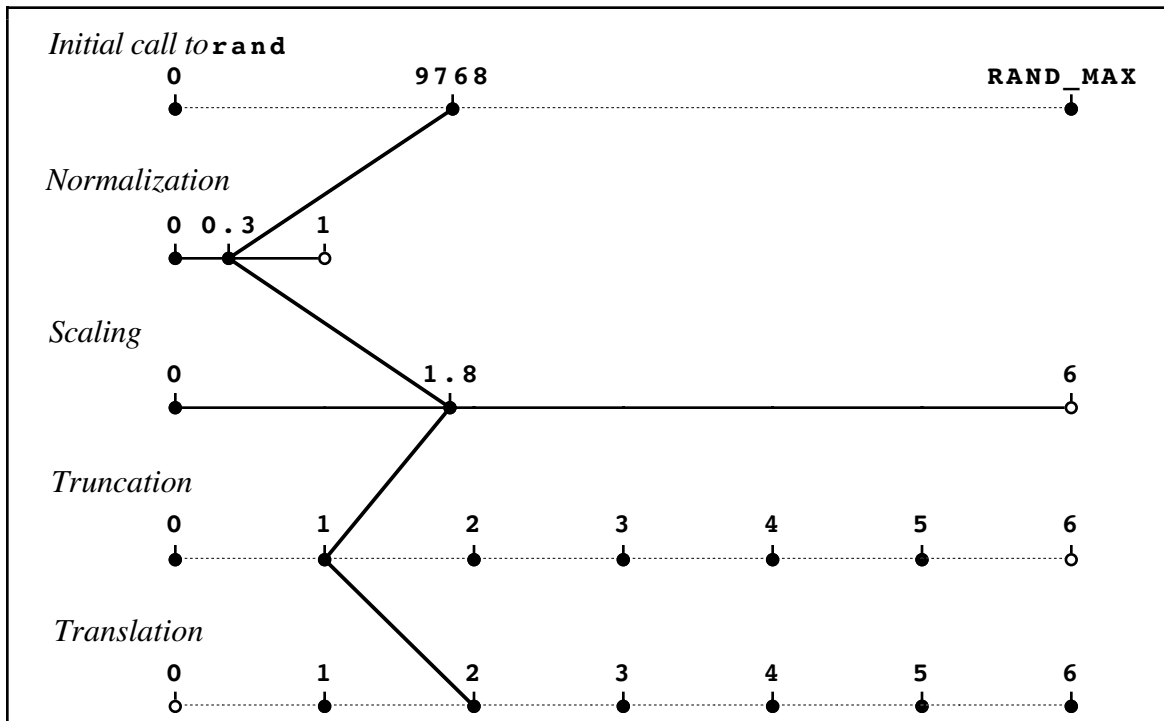
3.3 Strings

In any programming language, strings are one of the most important data types because they come up so frequently in applications. Conceptually, a string is just a sequence of characters. For example, the string "**hello, world**" is a sequence of 12 characters including ten letters, a comma, and a space. In C++, the **string** data type and its associated operations are defined in the **<string>** interface, and you must therefore include this interface in any source file that manipulates string data.

In many ways, **string** is an ideal example of an abstract data type, which is the topic of the following chapter. However, given that strings are essential to many applications and that they are more deeply integrated into the syntax of C++ than the classes described in Chapter 4, it makes sense to introduce them before the others. In this chapter, you will look at strings only from a client perspective, which means that you will be able to use them as a tool without having to understand how their underlying implementation.

The data type **string**

In Chapter 1, you learned that a data type is defined by two properties: a domain and a set of operations. If you think about strings as a separate data type, the domain is easy to

Figure 3-4 Steps required to calculate a random integer

identify; the domain of type **string** is the set of all sequences of characters. The more interesting problem is to identify an appropriate set of operations. Early versions of C++ and its predecessor language C did not have much support for manipulating strings. The only facilities provided were fairly low-level operations that required you to be very familiar with the precise representation of strings and work at a detailed level. The C++ standard introduced the standard **string** type that presents an abstract view of string and a richer set of facilities for operating on strings. This allows us to focus on manipulating strings using their published interface without concern for how strings might be internally represented.

Operations on the **string** type

One common way to initialize a string variable is with a **string literal**, a sequence of characters enclosed in double quotes:

```
string str = "Hello";
```

You can also create strings by concatenating existing strings using the `+` operator:

```
string str2 = str + "World";
```

The addition operator has been **overloaded**, or given multiple meanings. When `+` is applied to numbers, it performs addition, when applied to strings, it performs string concatenation. It is an error to attempt to add two operands of incompatible type such as a double and a string.

The shorthand `+=` form can be used to append onto an existing string. Both `+` and `+=` allow concatenation of another string or a single character:

```
string str = "Stanford";
str += '!';
```

As much as possible, operations on strings were defined so as to mimic the behavior of those operations on the built-in primitive types. For example, assigning one string variable to another copies the string contents:

```
str2 = str1;
```

This assignment overwrites any previous contents of **str2** with a copy of the characters contained in **str1**. The variables **str1** and **str2** remain independent. Changing the characters in **str1** would not cause any changes to **str2**.

When passing a string as parameter, a copy is made, just as with a primitive type. Changes to the parameter made within a function are not reflected in the calling function unless the string parameter is explicitly passed by reference.

Two strings can be compared using the relational operators, **==**, **!=**, **<**, **>**, **<=**, and **>=**. These operators compare strings **lexicographically**, which follows the order given by the character codes. This ordering means case is significant, so **"abc"** is not equal to **"ABC"**.

```
if (str == "quit") ...
```

You can retrieve the number of characters contained in a string by invoking the **length** function on the string using the dot operator:

```
int numChars = str.length();
```

The syntax for calling the **length** function is different from what you have seen so far. If **length** were like the functions in the **random.h**, you would expect to call it like this:

```
numChars = length(str);
```



This syntax is illegal.

What makes **length** different is that it is an integral part of the **string** class and must therefore be applied to a particular string object. In the expression

```
str.length()
```

the object is the variable **str**. In the language of the object-oriented paradigm, this object to the left of the dot is called the **receiver**, and the syntax is intended to emphasize that the object stored in **str** is receiving a request to perform the **length** operation. Informally, it makes sense to think of this operation as sending a message to the object asking for its length and having it respond by providing the requested information.

In C++, functions that are directed to a specific objects were originally called **member functions**, but that designation is gradually giving way to the term **method**, which is more common in other object-oriented languages. Functions that are independent of an object—such as those you’ve seen up to now—are called **free functions**. Other than the use of dot notation to identify the receiver, the mechanics of calling a method and calling a free function are pretty much the same.

It is possible to access the **i**th character from a string **str** using the notation like that of array selection, as follows:

```
str[i]
```

As with arrays, index numbers in a string begin at 0. Thus, if **str** contains the string **"Hello"**, then **str[0]** has the value **'H'**, **str[1]** has the value **'e'**, and so on.

The standard idiom for processing every character in a string looks like this:

```
for (int i = 0; i < str.length(); i++) {
    . . . body of loop that manipulates str[i] . . .
}
```

On each loop cycle, the selection expression `str[i]` refers to the i^{th} character in the string. Because the purpose of the loop is to process every character, the loop continues until `i` reaches the length of the string. Thus, you can count the number of spaces in a string using the following function:

```
int CountSpaces(string str) {
    int nSpaces = 0;

    for (int i = 0; i < str.length(); i++) {
        if (str[i] == ' ') nSpaces++;
    }
    return nSpaces;
}
```

As with array selection, selecting a character position that is out of range for the string is not automatically recognized as an error but will typically lead to incorrect results.

The `string` interface exports a large number of methods, many of which come up only in relatively specialized applications. Table 3-1 lists several of the more useful methods.

Table 3-1 Common string methods

<code>str.length()</code>	This method returns the number of characters in the receiver string <i>str</i> .
<code>str.at(pos)</code> <code>str[pos]</code>	Both of these expressions return the character at index <i>pos</i> within the receiver string <i>str</i> . However, if used to access an invalid position, the <code>at</code> method raises an error whereas bracket-selection has undefined behavior.
<code>str.substr(pos, len)</code>	This method returns a new string made of up <i>len</i> characters starting at <i>pos</i> copied from the receiver string <i>str</i> . If necessary, <i>len</i> is truncated at the end of <i>str</i> . The second parameter is optional; if not given, all characters to the end of the receiver string are assumed. A new string is returned; the receiver string is unchanged.
<code>str.find(ch, pos)</code>	This method searches the receiver string <i>str</i> starting at <i>pos</i> for the character <i>ch</i> and returns the index of the first location in which it appears. If <i>ch</i> does not appear, the function returns the constant <code>string::npos</code> . The second parameter is optional; if not given, zero is assumed.
<code>str.find(pattern, pos)</code>	An overloaded version of <code>find</code> that searches for a substring <i>pattern</i> instead of a single character.
<code>str.insert(pos, txt)</code>	This method copies the characters from <i>txt</i> and inserts them into the receiver string <i>str</i> starting at <i>pos</i> . Note this operation destructively modifies the receiver string.
<code>str.replace(pos, count, txt)</code>	This method replaces the <i>count</i> characters starting at <i>pos</i> in the receiver string <i>str</i> with the entire contents of string <i>txt</i> . If necessary, <i>count</i> is truncated to the end of <i>str</i> . Note this operation destructively modifies the receiver string.

The code in Figure 3-5 provides a more extensive example of how you can use the **string** interface to perform simple string manipulation. The principal function is **PigLatin**, which converts an English word to Pig Latin by applying the following rules:

- If the word contains no vowels, no translation is done, which means that the translated word is the same as the original.
- If the word begins with a vowel, the function adds the string "**way**" to the end of the original word. Thus, the Pig Latin equivalent of *alley* is *alleyway*.
- If the word begins with a consonant, the function extracts the string of consonants up to the first vowel, moves that collection of consonants to the end of the word, and adds the string "**ay**". For example, the Pig Latin equivalent of *trash* is *ashtray*.

The implementation of **PigLatin** uses the **substr** function to extract pieces of the original word and string concatenation to reassemble them. The **FindFirstVowel** function returns the integer index of the first vowel, or **-1** if no vowel appears.

Figure 3-5 Implementation of Pig Latin translation using the **string interface**

```

/*
 * Function: PigLatin
 * Usage: translation = PigLatin(word);
 * -----
 * This function translates a word from English to Pig Latin using
 * the rules specified in Chapter 3. The translated word is
 * returned as the value of the function.
 */

string PigLatin(string word) {
    int vp = FindFirstVowel(word);
    if (vp == -1) {
        return word;
    } else if (vp == 0) {
        return word + "way";
    } else {
        string head = word.substr(0, vp);
        string tail = word.substr(vp);
        return tail + head + "ay";
    }
}

/*
 * Function: FindFirstVowel
 * Usage: k = FindFirstVowel(word);
 * -----
 * This function returns the index position of the first vowel
 * in word. If word does not contain a vowel, FindFirstVowel
 * returns -1. The code for IsVowel appears in Chapter 1.
 */

int FindFirstVowel(string word) {
    for (int i = 0; i < word.length(); i++) {
        if (IsVowel(word[i])) return i;
    }
    return -1;
}

```

The principal advantage of the **string** interface is that strings are treated as abstract values. When a function needs to return a string to its caller, it does so by returning a string as its value. Any operations on that string are performed using the operations exported by the library. The client is free from having to worry about the details of string representation and allocation.

Even though this text relies extensively on the functions in the **string** interface, it is not necessary to understand that code from the implementation side. The whole point of interfaces is that they protect clients from the complexity of the implementation. As you continue with your programming career, you will often make use of libraries even though you have no understanding of their implementation, so it is best to become comfortable with that process as early as you can.

The **strutils.h** interface

There are a few minor convenience operations on strings that are used in this text as a supplement to the standard library. These functions are simple adaptations of existing standard functionality, but packaged in a way to be more convenient for use. This interface is called **strutils.h**, and is presented in its complete form in Figure 3-6.

An aside about C-style strings

Prior to the introduction of the **string** class into C++, strings were implemented as raw

Figure 3-6 Interface to the string utilities library

```
/*
 * File: strutils.h
 * -----
 * The strutils.h file defines the interface for a library of string
 * utilities.
 */

#ifndef _strutils_h
#define _strutils_h

#include "genlib.h"

/*
 * Function: ConvertToLowerCase
 * Usage: s = ConvertToLowerCase(s);
 * -----
 * This function returns a new string with all
 * alphabetic characters converted to lower case.
 */

string ConvertToLowerCase(string s);

/*
 * Function: ConvertToUpperCase
 * Usage: s = ConvertToUpperCase(s);
 * -----
 * This function returns a new string with all
 * alphabetic characters converted to upper case.
 */

string ConvertToUpperCase(string s);
```

```

/*
 * Function: IntegerToString
 * Usage: s = IntegerToString(n);
 * -----
 * This function converts an integer into the corresponding
 * string of digits. For example, IntegerToString(123)
 * returns "123" as a string.
 */

string IntegerToString(int n);

/*
 * Function: StringToInteger
 * Usage: n = StringToInteger(s);
 * -----
 * This function converts a string of digits into an integer.
 * If the string is not a legal integer or contains extraneous
 * characters, StringToInteger signals an error condition.
 */

int StringToInteger(string s);

/*
 * Function: RealToString
 * Usage: s = RealToString(d);
 * -----
 * This function converts a floating-point number into the
 * corresponding string form. For example, calling
 * RealToString(23.45) returns "23.45".
 */

string RealToString(double d);

/*
 * Function: StringToReal
 * Usage: d = StringToReal(s);
 * -----
 * This function converts a string representing a real number
 * into its corresponding value. If the string is not a
 * legal floating-point number or if it contains extraneous
 * characters, StringToReal signals an error condition.
 */

double StringToReal(string s);

#endif

```

arrays of characters with a sentinel value, the null character, used to mark the end of the string. You can recognize older programs that operate on these C-style strings by their use of the data type `char*` or `char[]`. Working with C-style strings is error-prone because of the exposed-pointer implementation and issues of allocation. Using the C++ `string` type frees the programmer from managing string storage and decreases the probability of difficult pointer bugs. Thus, you should use C++ strings wherever possible. That said, there are some facilities in the standard libraries that predate the introduction of the C++ string type and require the use of the older C-style strings. You can obtain the C-style equivalent from a C++ string using the method `c_str`:


```
string str = ...  
char *cstr = str.c_str();
```

Some of the legacy of C++ is demonstrated by the fact that string literals are in fact C-style strings. However, they are automatically converted to C++ strings as needed in most contexts, so you can use a string literal in most places where a C++ string is expected. Where needed, you can explicitly convert a string literal to a C++ string using a typecast-like notation:

```
string str = string("Hello");
```

3.4 Standard I/O and file streams

The most commonly used library is the I/O stream library. Every program in the text includes `iostream`, in order to display output using the `cout` stream. The `iostream` interface is extended in the `fstream` interface with additional features that make it possible to read and write files, which in turn enable you to store data values that persist even after your program completes its operation.

Data files

Whenever you want to store information on the computer for longer than the running time of a program, the usual approach is to collect the data into a logically cohesive whole and store it on a permanent storage medium as a file. Ordinarily, a file is stored using magnetic or optical media, such as on a removable floppy disk, a compact disc, or a hard disk. The important point is that the permanent data objects you store on the computer—documents, games, executable programs, source code, and the like—are all stored in the form of files.

On most systems, files come in a variety of types. For example, in the programming domain, you work with source files, object files, and executable files, each of which has a distinct representation. When you use a file to store data for use by a program, that file usually consists of text and is therefore called a **text file**. You can think of a text file as a sequence of characters stored in a permanent medium and identified by a file name. The name of the file and the characters it contains have the same relationship as the name of a variable and its contents.

As an example, the following text file contains the first stanza of Lewis Carroll's nonsense poem "Jabberwocky," which appears in *Through the Looking Glass*:

jabberwocky.txt

```
'Twas brillig, and the slithy toves  
Did gyre and gimble in the wabe;  
All mimsy were the borogoves,  
And the mome raths outgrabe.
```

When you look at a file, it is often convenient to regard it as a two-dimensional structure—a sequence of lines composed of individual characters. Internally, however, text files are represented as a one-dimensional sequence of characters. In addition to the printing characters you can see, files also contain the newline character '`\n`', which marks the end of each line. The file system also keeps track of the length of the file so programs that read files can recognize where the file ends.

In many respects, text files are similar to strings. Each consists of an ordered collection of characters with a specified endpoint. On the other hand, strings and files differ in several important respects. The most important difference is the permanence of

the data. A string is stored temporarily in the computer's memory during the time that a program runs; a file is stored permanently on a long-term storage device until it is explicitly deleted. There is also a difference in the way you use data in strings and files. The individual characters in a string may be accessed in any order by specifying the appropriate index. The characters in a text file are usually accessed in a sequential fashion, with a program starting at the beginning of the file and either reading or writing characters from the start to the end in order.

Using file streams in C++

To read or write a file as part of a C++ program, you must use the following steps:

1. *Declare a stream variable.* The **fstream** interface exports two types, **ifstream**, for files being read from, and **ofstream**, for files being written to. Each type keeps track of the information the system needs to manage file processing activity. Thus, if you are writing a program that reads an input file and creates a second file for output, you need to declare two variables, as follows:

```
ifstream infile;  
ofstream outfile;
```

2. *Open the file.* Before you can use a stream variable, you need to establish an association between that variable and an actual file. This operation is called **opening** the file and is performed by calling the stream method **open**. Like the methods seen previously on **string**, you invoke it using dot notation on the stream. For example, if you wanted to read the text of the **jabberwocky.txt** file, you would execute the statement

```
infile.open("jabberwocky.txt");
```

One minor glitch to note is that the **open** method expects a C-style string as the argument. A string literal is acceptable as is, but if you have the filename in a C++ string variable, you will need to pass its C-style equivalent as shown:

```
string str = ...  
infile.open(str.c_str());
```

If a requested input file is missing, the open operation will fail. You can test the current state of a stream by using the **fail** method. As a programmer, you have a responsibility to check for this error and report it to the user.

```
if (infile.fail()) Error("Could not open file.");
```

In some cases, you may want to recover from such an input file failure (for example, if the user accidentally typed a non-existent filename), and try to open the input stream once again. In such cases, you must first clear the "fail state" that the input stream variable will be in after such a failure. You can do this by calling the **clear** method on the stream, as exemplified below.

```
infile.clear();
```

3. *Transfer the data.* Once you have opened the data files, you then use the appropriate stream operations to perform the actual I/O operations. For an input file, these operations read data from the file into your program; for an output file, the operations transfer data from the program to the file. To perform the actual transfers, you can choose any of several strategies, depending on the application. At the simplest level, you can read or write files character by character. In some cases, however, it is more convenient to process files line by line. At a still higher level, you can choose to read

and write formatted data. Doing so allows you to intermix numeric data with strings and other data types.

4. *Close the file.* When you have finished all data transfers, you need to indicate that fact to the file system by calling the stream method **close**. This operation, called **closing** the file, breaks the association between a **stream** variable and the actual file.

```
infile.close();
```

Standard streams

The standard I/O library defines three special identifiers—**cin**, **cout**, and **cerr**—that act as stream variables and are available to all programs. These variables are referred to as **standard streams**. The constant **cin** designates the standard input stream, which is the source for user input. The constant **cout** indicates the standard output stream and represents the device on which output data is written to the user. The constant **cerr** represents the standard error stream and is used to report any error messages the user should see. Typically, the standard streams all refer to the computer console. When you read data from **cin**, the input comes from the keyboard; when you write data to **cout** or **cerr**, the output appears on the screen. You do not open or close these streams; they are handled automatically.

Formatted stream output

Stream output is usually accomplished by using the insertion operator **<<**. The operand on the left is the output stream; the operand on the right is the data being inserted into the stream. Thus far, you have only written to **cout**, but any open output stream can be used as the destination.

```
outfile << "some text " << num << endl;
```

The insertion operator will accept any primitive data type on the right. Arrays are treated as pointers in this context and only the base address is printed. Strings print the entire sequence of characters. By default, complex types such as structs cannot be printed with a single insertion operation; you must output each of the struct members one by one. Several insertions can be chained together by multiple uses of the insertion operator. The manipulator **endl** is used to output a newline.

Values are formatted according to the stream's default rules unless you specify otherwise. For explicit control, you insert I/O manipulators to set the field width, alignment, precision, and other features on a per-stream basis. A few common output manipulators options are described in the section on "Simple input and output" in Chapter 1. Although the mechanisms for formatted I/O in any programming language can be quite useful, they also tend to be detail-ridden. I recommend having a reference on hand for looking up details on as-needed basis rather than memorizing the entirety of the options.

Formatted stream input

The extraction operator **>>** provides an input counterpart to the insertion operator **<<** and allows programs to read in values of the various basic types. Here is a sample usage of reading an integer from an input stream:

```
int num;  
infile >> num;
```

The operand on the left is the input stream; the operand on the right is the variable that will store the value read from the stream. The previous value of the variable is overwritten by the next integer read from the stream.

The default setting for an input stream is to skip characters that appear as blank space, such as the space, tab, and newline characters. When reading the next value, the insertion operator will skip over any leading blank space and start reading from the first non-blank character. When extracting a string from an input stream, a sequence of non-blank characters is read, stopping at the first blank. To illustrate how blank space is handled, consider the a text file with the following contents:

```
data.txt
3      45
some text
```

The code below attempts to extract two integers, a string, and character from this file:

```
int num1, num2;
string str;
char ch;
infile >> num1 >> num2 >> str >> ch;
```

After executing this code, **num1** would hold the value 3, **num2** the value 45, **str** would be "some", and **ch** would be 't'.

The extraction operator will fail if there is no input remaining or if the next input is not compatible with the type you are attempting to extract. The stream method **fail** can be used to test whether an extraction operation was successful.

```
if (infile.fail()) ...
```

If you were attempting to read and sum integers from a file until you reached the end, you could use a loop such as this one:

```
int total, num;
total = 0;
while (true) {
    infile >> num;
    if (infile.fail()) break;
    total += num;
}
```

For a program to be robust, it typically must take care to handle malformed input. This is especially important when reading from **cin**, where the human user is prone to typing errors at the console. To avoid cluttering the code with error-handling, this text does not use the extraction operator on **cin** and relies instead on the facilities provided by the **simpio.h** interface described in Chapter 1 that have error-handling built-in.

Single character I/O

Most I/O is handled with the insertion and extraction operators, but there are times when it is more convenient to process files as a sequence of characters or lines. The method **get** is used to read a single character from an input stream. Although the idea of **get** seems simple enough, there is a confusing aspect in its design. If you look at the formal definition of **get**, its prototype looks like this:

```
int get();
```

At first glance, the result type seems odd. The prototype indicates that **get** returns an integer, even though conceptually the function returns a character. The reason for this design decision is that returning a character would make it impossible for a program to detect the end-of-file mark. There are only 256 possible character codes, and a data file might contain any of those values. There is no value—or at least no value of type **char**—that you could use as a sentinel to indicate the end-of-file condition. By extending the definition so that **get** returns an integer, the implementation can return a value outside the range of legal character codes to indicate the end-of-file condition. That value is given the symbolic name of **EOF**.

COMMON PITFALLS

Remember that **get** returns an **int**, not a **char**. If you use a variable of type **char** to store the result of **get**, your program will be unable to detect the end-of-file condition.

For output streams, the stream method **put** takes just one argument, a single character, and writes that character to the stream:

```
outfile.put(ch);
```

As an example of the use of **get** and **put**, you can copy one file to another by calling the following function:

```
void CopyFile(istream & infile, ostream & outfile) {
    int ch;
    while ((ch = infile.get()) != EOF) {
        outfile.put(ch);
    }
}
```

The **while** loop in **CopyFile** is highly idiomatic and deserves some consideration. The test expression for the **while** loop uses embedded assignment to combine the operations of reading in a character and testing for the end-of-file condition. When the program evaluates the **while** condition, it begins by evaluating the subexpression

```
ch = infile.get()
```

which reads a character and assigns it to **ch**. Before executing the loop body, the program then goes on to make sure the result of the assignment is not **EOF**. The parentheses around the assignment are required; without them, the expression would incorrectly assign to **ch** the result of comparing the character against **EOF**.

Note that both stream parameters are passed by reference to the function **CopyFile**. Reading from and writing to a stream changes its internal state, and it is essential to pass streams by reference so that the stream state is consistently maintained throughout the context of the entire program.

Rereading characters from an input file

When you are reading data from an input file, you will often find yourself in the position of not knowing that you should stop reading characters until you have already read more than you need. For example, suppose that you are asked to write a program that copies a program from one file to another, removing all comments as it does so. As you know, a comment in C++ begins with the character sequence **/*** and ends with the sequence ***/**. A program to remove them must copy characters until it detects the initial **/*** sequence and then read characters without copying them until it detects the ***/** at the end. The only aspect of this problem that poses any difficulty is the fact that the comment markers are two characters long. If you are copying the file a character at a time, what do you do when you encounter a slash? It might be the beginning of a comment, in which case you

should not copy it to the output file. On the other hand, it might be the division operator. The only way to determine which of these cases applies is to look at the next character. If it is an asterisk, you need to ignore both characters and make note of the fact that a comment is in progress. If it not, however, what you would most like to do is forget that you ever read that character and then treat it normally on the next cycle of the loop.

The stream interface provides a function that allows you to do just that. The method is called **unget** and has the following form:

```
infile.unget();
```

The effect of this call is to “push” the last character read back into the input stream so that it is returned on the next call to **get**. The C++ libraries only guarantee the ability to push back one character into the input file, so you should not rely on being able to read several characters ahead and then push them all back. Fortunately, being able to push back one character is sufficient in the vast majority of cases.

An implementation of a function **CopyRemovingComments** that uses **unget** is shown in Figure 3-7.

Line-oriented I/O

Because files are usually subdivided into individual lines, it is often useful to read an entire line of data at a time. The stream function that performs this operation is called **getline**. (not to be confused with the similarly-named **GetLine** function from the **simpio.h** interface). Unlike the other stream operations we’ve discussed, **getline** is not

Figure 3-7 Implementation of CopyRemovingComments

```
void CopyRemovingComments(istream & infile, ostream & outfile) {
    int ch, nch;
    bool commentFlag;

    commentFlag = false;
    while ((ch = infile.get()) != EOF) {
        if (commentFlag) {
            if (ch == '*') {
                nch = infile.get();
                if (nch == '/') {
                    commentFlag = false;
                } else {
                    infile.unget();
                }
            }
        } else {
            if (ch == '/') {
                nch = infile.get();
                if (nch == '*') {
                    commentFlag = true;
                } else {
                    infile.unget();
                }
            }
            if (!commentFlag) outfile.put(ch);
        }
    }
}
```

a method and is not invoked using dot notation. **getline** takes two arguments, the input stream to read from, and a string; both are reference parameters.

```
getline(infile, str);
```

The effect of this function is to copy the next line of the file into the string parameter. **getline** removes the newline character used to signal the end of the line, so that the string contains simply the characters on the line. The string will be the empty string if the next character coming up in the stream is a newline. The **getline** function will fail if there are no more lines to be read from the file. As always, you can test whether the stream is in failure state using the **fail** method

Some of the most common operations exported by the **iostream** and **fstream** interfaces are summarized in Table 3-3.

Table 3-3 Common operations for the *iostream* and *fstream* classes

Call	Operations for all streams
<i>file</i> . open (<i>filename</i>)	This method attempts to open the named file and attach it to the receiver stream <i>file</i> . Note that the <i>filename</i> parameter is a C-style string, not a C++ string object. You can convert a C++ string to its C-style equivalent with the string method c_str . You can check whether open fails by calling the fail method.
<i>file</i> . close ()	This method closes the file attached to the stream <i>file</i> .
<i>file</i> . fail ()	This method returns a boolean indicating the error state of the stream <i>file</i> . A true result means that a previous operation on this stream failed for some reason. Once an error occurs, the error state must be cleared before any further operations will succeed.
<i>file</i> . clear ()	This method clears the error state of the stream <i>file</i> . After an error, you must clear the stream before executing subsequent operations.
Operations for input streams	
<i>infile</i> . get ()	This method reads and returns the next character from the input stream <i>infile</i> . If there are no more characters, get returns the constant EOF . Note that the return value is of type int , not char .
<i>infile</i> . unget ()	This method pushes the last character read back onto the input stream <i>infile</i> .
getline (<i>infile</i> , <i>str</i>)	This function reads the next line from the input stream <i>infile</i> into the reference parameter <i>str</i> , discarding the newline. Note this is not a method, but an ordinary free function. (This is a different function than GetLine from simpio.h .)
<i>infile</i> >> <i>var</i>	The stream extraction operation reads a value into <i>var</i> from the input stream <i>infile</i> . By default, leading whitespace is skipped. If there is no input remaining or the next input is not compatible with the type of <i>var</i> , the stream is set into an error state.
Operations for output streams	
<i>outfile</i> . put (<i>ch</i>)	This method writes the character <i>ch</i> to the output stream <i>outfile</i> .
<i>outfile</i> << <i>expr</i>	This stream insertion operations writes the value of <i>expr</i> to the output stream <i>outfile</i> . The expression can evaluate to any type that has a defined stream insertion behavior, generally this means all primitive types and string.

3.5 Other ANSI libraries

In addition to the **string** and **fstream** interfaces described earlier in this chapter, there are a few other interfaces in the standard library that are important for you to know. The most important of these are the **cctype** interface, which exports a set of functions for working with characters, and the **cmath** interface, which exports several common mathematical functions. The most important functions exported by these interfaces are shown in Tables 3-4 and 3-5, respectively.

Table 3-4 Common functions exported by the cctype interface

isupper (<i>ch</i>) islower (<i>ch</i>) isalpha (<i>ch</i>)	These functions return true if <i>ch</i> is an uppercase letter, a lowercase letter, or any type of letter, respectively.
isdigit (<i>ch</i>)	This function returns true if <i>ch</i> is one of the digit characters.
isalnum (<i>ch</i>)	This function returns true if <i>ch</i> is alphanumeric , which means that it is either a letter or a digit.
ispunct (<i>ch</i>)	This function returns true if <i>ch</i> is a punctuation symbol.
isspace (<i>ch</i>)	This function returns true if <i>ch</i> is a whitespace character . These characters are ' ' (the space character), '\t', '\n', '\f', or '\v', all of which appear as blank space on the screen.
isprint (<i>ch</i>)	This function returns true if <i>ch</i> is any printable character, including the whitespace characters.
toupper (<i>ch</i>) tolower (<i>ch</i>)	If <i>ch</i> is a letter, these functions return <i>ch</i> converted to the desired case, so that tolower ('A') returns 'a'; if not, <i>ch</i> is returned unchanged.

Table 3-5 Common functions exported by the math interfaces

fabs (<i>x</i>)	This function returns the absolute value of a real number <i>x</i> . (Note: The function abs , which takes the absolute value of an integer, is exported by cstdlib)
floor (<i>x</i>)	This function returns the floating-point representation of the largest integer less than or equal to <i>x</i> .
ceil (<i>x</i>)	This function returns the floating-point representation of the smallest integer greater than or equal to <i>x</i> .
fmod (<i>x</i> , <i>y</i>)	This function returns the floating-point remainder of <i>x</i> / <i>y</i> .
sqrt (<i>x</i>)	This function returns the square root of <i>x</i> .
pow (<i>x</i> , <i>y</i>)	This function returns x^y .
exp (<i>x</i>)	This function returns e^x .
log (<i>x</i>)	This function returns the natural logarithm of <i>x</i> .
sin (<i>theta</i>) cos (<i>theta</i>)	These functions return the sine and cosine of the angle <i>theta</i> , which is expressed in radians. You can convert from degrees to radians by multiplying by $\pi/180$.
atan (<i>x</i>)	This function returns the trigonometric arctangent of the value <i>x</i> . The result is an angle expressed in radians between $-\pi/2$ and $+\pi/2$.
atan2 (<i>y</i> , <i>x</i>)	This function returns the angle formed between the <i>x</i> -axis and the line extending from the origin through the point (<i>x</i> , <i>y</i>). As with the other trigonometric functions, the angle is expressed in radians.

In some cases, the functions in these interfaces are easy to implement on your own. Even so, it is good programming practice to use the library functions instead of writing your own. There are three principal reasons for doing so:

1. Because the library functions are standard, programs you write will be easier for other programmers to read. Assuming that the programmers are at all experienced in C++, they will recognize these functions and know exactly what they mean.
2. It is easier to rely on library functions for correctness than on your own. Because the C++ libraries are used by millions of client programmers, there is considerable pressure on the implementers to get the functions right. If you rewrite library functions yourself, the chance of introducing a bug is much larger.
3. The library implementations of functions are often more efficient than those you can write yourself. Because these libraries are standard and their performance affects many clients, the implementers have a large incentive to optimize the performance of the libraries as much as possible.

Summary

In this chapter, you have learned about *interfaces*, which are the points of connection between the client of a library abstraction and the corresponding implementation. Interfaces are one of the central themes of modern programming and will be used extensively throughout the rest of this text. You have also learned how to use several specific interfaces including **random.h**, **string**, **fstream**, **cctype**, and **cmath**.

Important points in this chapter include:

- An interface provides a channel of communication between a client of a library and its implementation but also acts as a barrier to keep unnecessary information from crossing the boundary between the two sides.
- Interfaces in C++ are represented using header files.
- The definitions exported by an interface are called *interface entries*. The most common interface entries are function prototypes, constant definitions, and type definitions. The interface should contain extensive comments for each entry so that the client can understand how to use that entry but should not expose details that are relevant only to the implementation.
- A well-designed interface must be *unified*, *simple*, *sufficient*, *general*, and *stable*. Because these criteria sometimes conflict with each other, you must learn to strike an appropriate balance in your interface design.
- You can use the **random.h** interface to generate pseudorandom numbers that appear to be random even though they are generated deterministically. The **random.h** interface exports four entries: **RandomInteger**, **RandomReal**, **RandomChance**, and **Randomize**.
- The interface **string** provides the string class, an abstraction used for manipulating sequences of characters. The **strutils.h** interface adds some convenience operations on strings.
- The **fstream** interface exports operations that allow you to read and write data files. The process of using a data file consists of four steps: declaring a stream variable, opening the file, transferring data, and closing the file.

Review questions

1. Define the following terms: *interface*, *package*, *abstraction*, *implementation*, *client*.

2. In your own words, describe the difference in perspective between a programmer who writes the implementation of a library and one who writes programs that are clients of that library.
3. How are interfaces represented in C++?
4. What are the most common interface entries?
5. What are the five criteria for good interface design listed in this chapter?
6. Why is it important for an interface to be stable?
7. What is meant by the term *extending an interface*?
8. Why are comments particularly important in interfaces?
9. True or false: The comments in an interface should explain in detail how each of the exported functions is implemented.
10. If you were defining an interface named **magic.h**, what would the interface boilerplate look like? What is the purpose of these lines?
11. Why are the values generated by the **rand** function called *pseudorandom numbers*?
12. How would you use **RandomInteger** to generate a random four-digit positive number?
13. Could you use the multiple assignment statement

```
d1 = d2 = RandomInteger(1, 6);
```

to simulate the process of rolling two dice?

14. The **rand** function ordinarily generates the same sequence of random numbers every time a program is run. What is the reason for this behavior?
15. What is meant by the term *seed* in the context of random numbers?
16. What four steps are necessary to convert the result of **rand** into an integer value between the limits **low** and **high**?
17. Why is it not necessary to understand how a string is stored in memory?
18. What idiom can you use to process each character in a string?
19. What is the purpose of the **strutils.h** interface?
20. Assuming that **s1** and **s2** are strings, describe the effect of the conditional test in the following **if** statement:

```
if (s1 == s2) . . .
```

21. If you call **s1.replace(0, 1, s2)**, which string is the *receiver*?
22. What happens if you attempt to access a character at an invalid index of a string?
23. How are two strings of different cases handled when being compared using the relational operators on two strings?

24. What is the result of calling each of the following functions from the **string** interface?

```
string s = "ABCDE", t = "";
```

- a. **s.length()**;
- b. **t.length()**;
- c. **s[2]**
- d. **s + t**
- e. **t += 'a'**
- f. **s.replace(0, 2, "Z")**
- g. **s.substr(0, 3)**
- h. **s.substr(4)**
- i. **s.substr(3, 9)**
- j. **s.substr(3, 3)**

25. What is the result of calling each of the following functions?

```
string s = "ABCDE", t = "Abracadabra";
```

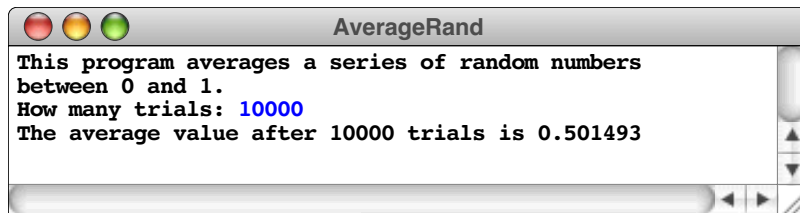
- a. **s == "abcde"**
- b. **s == "ABCDE"**
- c. **s == "ABC"**
- d. **s < "abcde"**
- e. **t.find('a', 0)**
- f. **t.find("ra", 3)**
- g. **t.find("cad")**
- h. **t.find("CAD", 1);**
- i. **ConvertToLowerCase("Catch-22")**
- j. **RealToString(3.140)**

26. What is the purpose of the types **ifstream** and **ofstream**? Is understanding the underlying structure of these types important to most programmers?
27. The argument to **open** must be a C-style string. What is the significance of this requirement?
28. How can you determine if an open operation on a stream was unsuccessful?
29. The **iostream** interface automatically defines three standard streams. What are their names? What purpose does each one serve?
30. When you are using the **get** method, how do you detect the end of a file?
31. Why is the return type of **get** declared as **int** instead of **char**?
32. What is the purpose of the **unget** method?
33. How can you determine whether an extraction operation on a stream was successful?
34. What are the differences between the functions **GetLine** and **getline**?

35. True or false: It is very worthwhile to memorize all the features of the standard I/O library since they are used so extensively.
36. What is the result of each of the following calls from **cctype**:
- a. `isdigit(7)`
 - b. `isdigit('7')`
 - c. `isalnum('7')`
 - d. `toupper('7')`
 - e. `toupper('A')`
 - f. `tolower('A')`
37. When using the trigonometric functions in the **cmath** interface, how can you convert an angle from degrees to radians?

Programming exercises

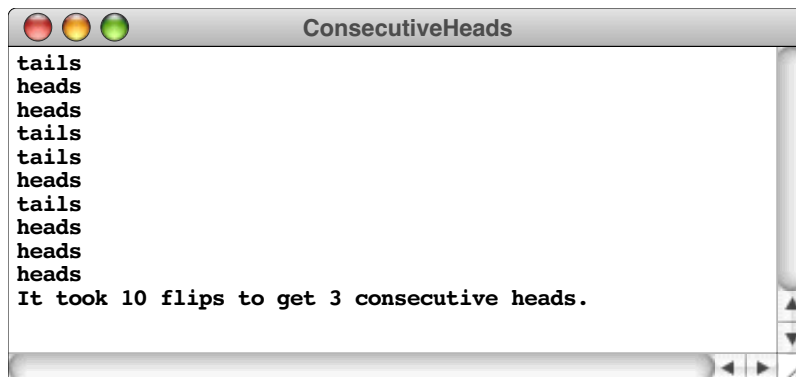
1. Write a program that repeatedly generates a random real number between 0 and 1 and then displays the average after a certain number of runs, as illustrated by the following sample run:



If the random number generator is working well, the average should get closer to 0.5 as the number of trials increases.

2. *Heads...*
Heads...
Heads...
A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.
— Tom Stoppard, *Rosencrantz and Guildenstern Are Dead*, 1967

Write a program that simulates flipping a coin repeatedly and continues until three *consecutive* heads are tossed. At that point, your program should display the total number of coin flips that were made. The following is one possible sample run of the program:



3. In casinos from Monte Carlo to Las Vegas, one of the most common gambling devices is the slot machine—the “one-armed bandit.” A typical slot machine has three wheels that spin around behind a narrow window. Each wheel is marked with the following symbols: **CHERRY**, **LEMON**, **ORANGE**, **PLUM**, **BELL**, and **BAR**. The window, however, allows you to see only one symbol on each wheel at a time. For example, the window might show the following configuration:

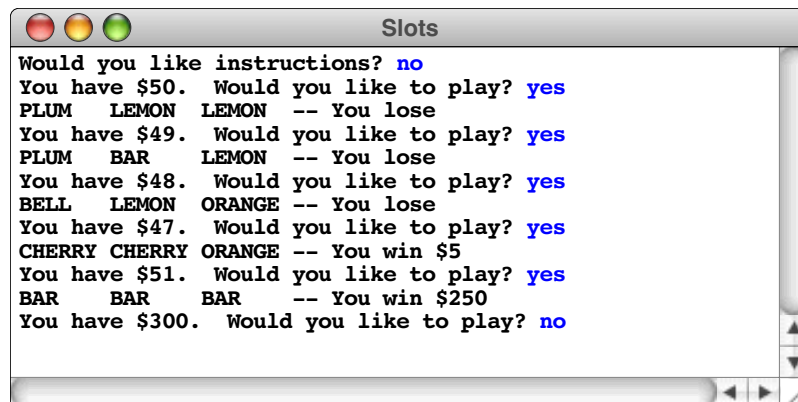


If you put a dollar into a slot machine and pull the handle on its side, the wheels spin around and eventually come to rest in some new configuration. If the configuration matches one of a set of winning patterns printed on the front of the slot machine, you get back some money. If not, you’re out a dollar. The following table shows a typical set of winning patterns, along with their associated payoffs:

BAR	BAR	BAR	<i>pays</i>	\$250
BELL	BELL	BELL/BAR	<i>pays</i>	\$20
PLUM	PLUM	PLUM/BAR	<i>pays</i>	\$14
ORANGE	ORANGE	ORANGE/BAR	<i>pays</i>	\$10
CHERRY	CHERRY	CHERRY	<i>pays</i>	\$7
CHERRY	CHERRY	—	<i>pays</i>	\$5
CHERRY	—	—	<i>pays</i>	\$2

The notation **BELL/BAR** means that either a **BELL** or a **BAR** can appear in that position, and the dash means that any symbol at all can appear. Thus, getting a **CHERRY** in the first position is automatically good for \$2, no matter what appears on the other wheels. Note that there is never any payoff for the **LEMON** symbol, even if you happen to line up three of them.

Write a program that simulates playing a slot machine. Your program should provide the user with an initial stake of \$50 and then let the user play until the money runs out or the user decides to quit. During each round, your program should take away a dollar, simulate the spinning of the wheels, evaluate the result, and pay the user any appropriate winnings. For example, a user might be lucky enough to see the following sample run:



Even though it’s not realistic (and would make the slot machine unprofitable for the casino), you should assume that the six symbols are equally likely on each wheel.

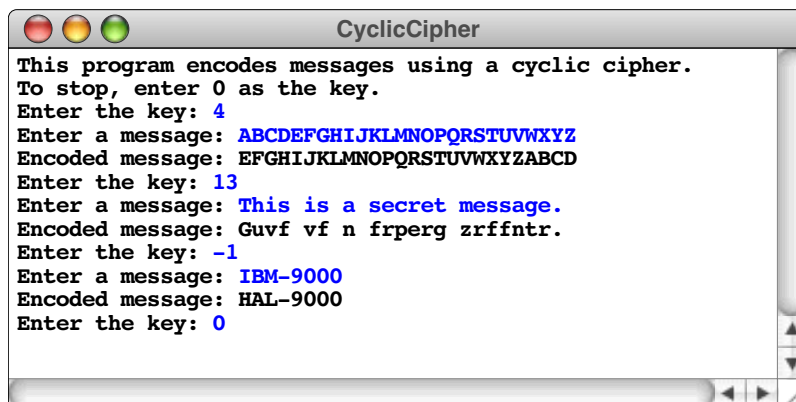
4. Implement a function **EqualIgnoringCase(str1, str2)** that returns **true** if the two string parameters contain the same sequence of characters ignoring case distinctions. Implement this once using the convenience functions from **strutils.h** and again without.
5. Implement a function **Capitalize(str)** that returns a string in which the initial character is capitalized (if it is a letter) and all other letters are converted so that they appear in lowercase form. Characters other than letters are not affected. For example, **Capitalize("BOOLEAN")** and **Capitalize("boolean")** should each return the string **"Boolean"**.
6. A **palindrome** is a word that reads identically backward and forward, such as *level* or *noon*. Write a predicate function **IsPalindrome(str)** that returns **true** if the string **str** is a palindrome. In addition, design and write a test program that calls **IsPalindrome** to demonstrate that it works.
7. One of the simplest types of codes used to make it harder for someone to read a message is a **letter-substitution cipher**, in which each letter in the original message is replaced by some different letter in the coded version of that message. A particularly simple type of letter-substitution cipher is a **cyclic cipher**, in which each letter is replaced by its counterpart a fixed distance ahead in the alphabet. The word *cyclic* refers to the fact that if the operation of moving ahead in the alphabet would take you past Z, you simply circle back to the beginning and start over again with A.

Using the string functions provided by the **string** interface, implement a function **EncodeString** with the prototype

```
string EncodeString(string str, int key);
```

The function returns the new string formed by shifting every letter in **str** forward the number of letters indicated by **key**, cycling back to the beginning of the alphabet if necessary. For example, if **key** has the value 4, the letter *A* becomes *E*, *B* becomes *F*, and so on up to *Z*, which becomes *D* because the coding cycles back to the beginning. If **key** is negative, letter values should be shifted toward the beginning of the alphabet instead of the end.

After you have implemented **EncodeString**, write a test program that duplicates the examples shown in the following sample run:



```

CyclicCipher
This program encodes messages using a cyclic cipher.
To stop, enter 0 as the key.
Enter the key: 4
Enter a message: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Encoded message: EFGHIJKLMNOPQRSTUVWXYZABCD
Enter the key: 13
Enter a message: This is a secret message.
Encoded message: Guvf vf n frperg zrffntr.
Enter the key: -1
Enter a message: IBM-9000
Encoded message: HAL-9000
Enter the key: 0

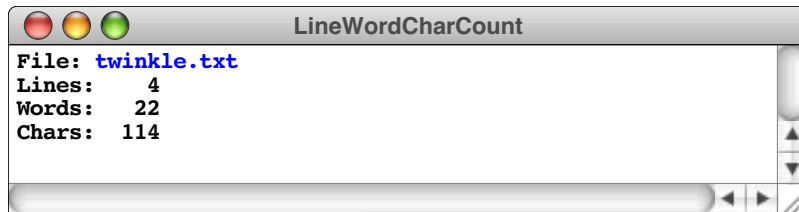
```

Note that the coding operation applies only to letters; any other character is included unchanged in the output. Moreover, the case of letters is unaffected: lowercase letters come out as lowercase, and uppercase letters come out as uppercase.

8. Without using the string method `substr`, implement your own function `SubString(s, pos, len)`, which returns the substring of `s`, beginning at position `pos` and including at most `len` characters. Make sure that your function correctly applies the following rules:
 - a. If `pos` is negative, it is set to 0 so that it indicates the first character in the string.
 - b. If `len` is greater than `s.length() - pos`, it is set to `strlen(s) - pos` so that it stops at last character.
 - c. If `pos` is greater than `s.length() - 1`, `SubString` returns the empty string.
9. Write a program `wc.cpp` that reads a file and reports how many lines, words, and characters appear in it. For the purposes of this program, a word consists of a consecutive sequence of any characters except whitespace characters. For example, if the file `twinkle.txt` contains the following verse from *Alice in Wonderland*,

```
Twinkle, twinkle, little bat!
How I wonder what you're at!
Up above the world you fly,
Like a teatray in the sky.
```

your program should be able to generate the following sample run:

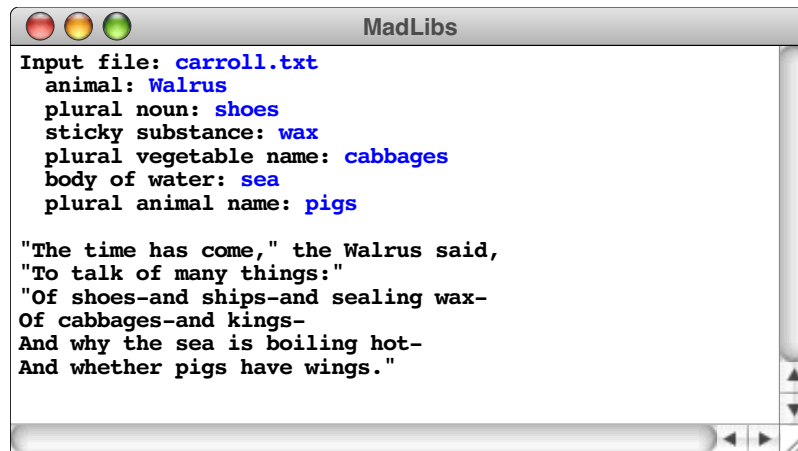


10. In the 1960s, entertainer Steve Allen often played a game called *madlibs* as part of his comedy routine. Allen would ask the audience to supply words that fit specific categories—a verb, an adjective, or a plural noun, for example—and then use these words to fill in blanks in a previously prepared text that he would then read back to the audience. The results were usually nonsense, but often very funny nonetheless.

In this exercise, your task is to write a program that plays madlibs with the user. The text for the story comes from a text file that includes occasional placeholders enclosed in angle brackets. For example, suppose the input file `carroll.txt` contains the following excerpt from Lewis Carroll's poem "The Walrus and the Carpenter," with a few key words replaced by placeholders as shown:

```
"The time has come," the <animal> said,
"To talk of many things:"
"Of <plural noun>-and ships-and sealing <sticky substance>-
Of <plural vegetable name>-and kings-
And why the <body of water> is boiling hot-
And whether <plural animal name> have wings."
```

Your program must read this file and display it on the console, giving the user the chance to fill in the placeholders with new strings. If Carroll himself had used the program, he would likely have obtained the following sample run:



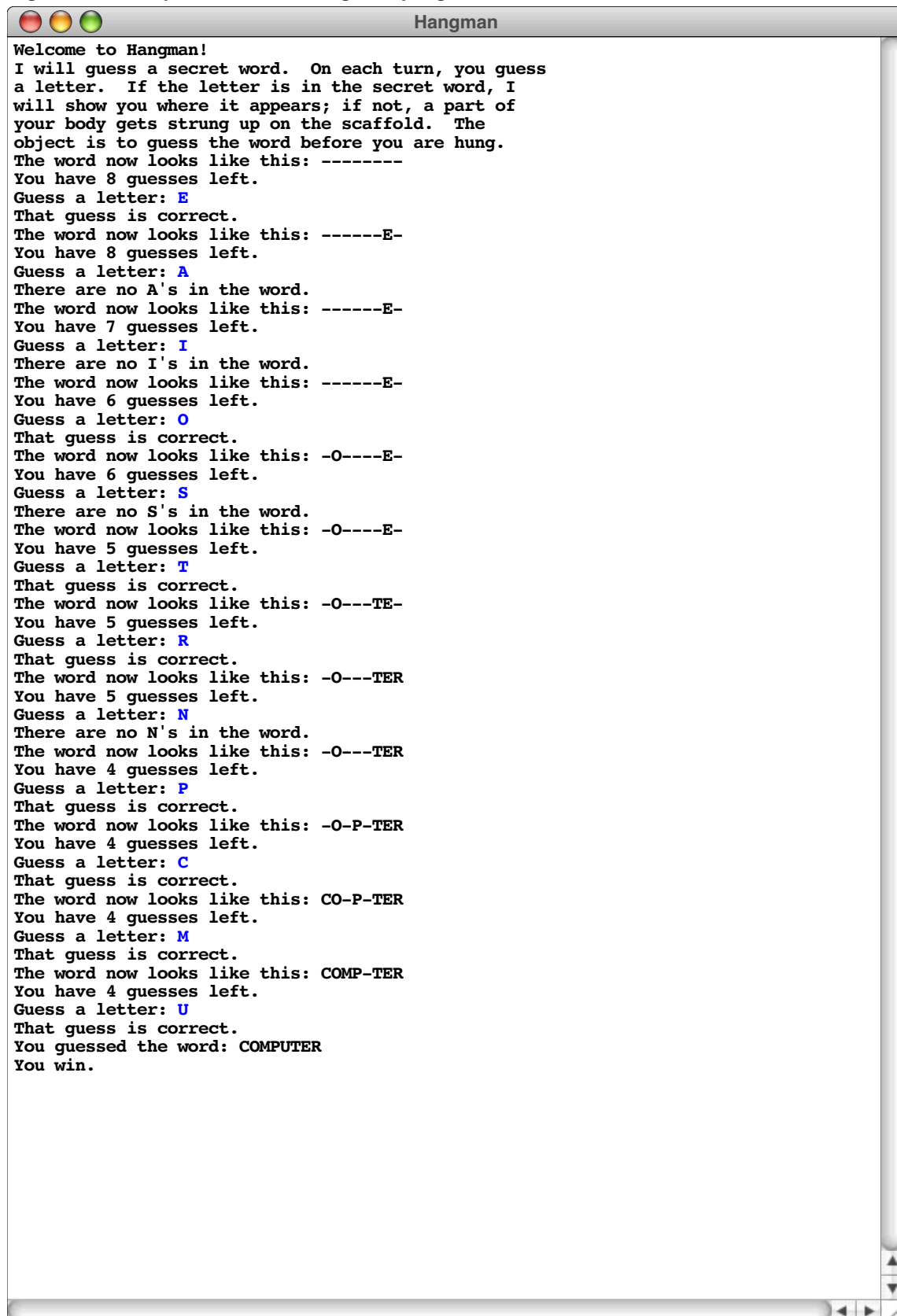
Note that the user must provide all the substitutions before any of the text is displayed. This design complicates the program structure slightly because it is impossible to display the output text as you go. The simplest strategy is to write the output to a temporary file first and then copy the contents of the temporary file back to the screen.

11. Write a program that plays the game of hangman. In hangman, the computer begins by selecting a secret word at random from a list of possibilities. It then prints out a row of dashes—one for each letter in the secret word—and asks the user to guess a letter. If the user guesses a letter that appears in the word, the word is redisplayed with all instances of that letter shown in the correct positions, along with any letters guessed correctly on previous turns. If the letter does not appear in the word, the player is charged with an incorrect guess. The player keeps guessing letters until either (1) the player has correctly guessed all the letters in the word or (2) the player has made eight incorrect guesses. A sample run of the hangman program is shown in Figure 3-8.

To separate the process of choosing a secret word from the rest of the game, define and implement an interface called **randword.h** that exports two functions: **InitDictionary** and **ChooseRandomWord**. **InitDictionary** should take the name of a data file containing a list of words, one per line, and read it into an array declared as a static global variable in the implementation. **ChooseRandomWord** takes no arguments and returns a word chosen at random from the internally maintained array.

12. Design and implement an interface called **card.h** that exports the following interface entries:
 - A type **rankT** that allows you to represent the rank of a card. The values of type **rankT** include the integers between 2 and 10 but should also include the constants **Ace**, **Jack**, **Queen**, and **King**.
 - A type **suitT** consisting of the four suits: **Clubs**, **Diamonds**, **Hearts**, and **Spades**.
 - A type **cardT** that combines a rank and a suit.
 - A function **NewCard(rank, suit)** that creates a **cardT** from the rank and suit values.
 - Two functions, **Rank(card)** and **Suit(card)**, that allow the client to select the rank and suit of a **cardT** value. These functions could easily be replaced by code that selected the appropriate components of the card, but defining them as functions means that the client need not pay attention to the underlying structure of the type.

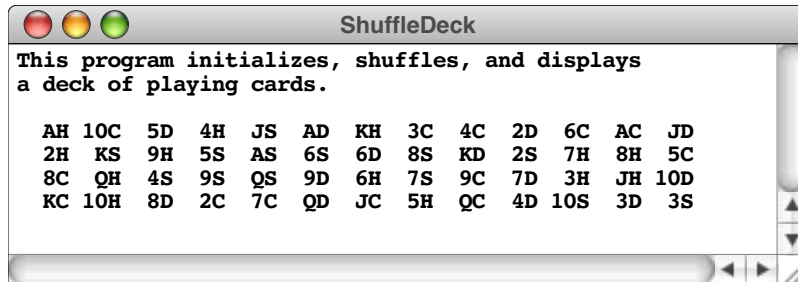
Figure 3-8 Sample run of the hangman program



```

Welcome to Hangman!
I will guess a secret word.  On each turn, you guess
a letter.  If the letter is in the secret word, I
will show you where it appears; if not, a part of
your body gets strung up on the scaffold.  The
object is to guess the word before you are hung.
The word now looks like this: -----
You have 8 guesses left.
Guess a letter: E
That guess is correct.
The word now looks like this: -----E-
You have 8 guesses left.
Guess a letter: A
There are no A's in the word.
The word now looks like this: -----E-
You have 7 guesses left.
Guess a letter: I
There are no I's in the word.
The word now looks like this: -----E-
You have 6 guesses left.
Guess a letter: O
That guess is correct.
The word now looks like this: -O----E-
You have 6 guesses left.
Guess a letter: S
There are no S's in the word.
The word now looks like this: -O----E-
You have 5 guesses left.
Guess a letter: T
That guess is correct.
The word now looks like this: -O---TE-
You have 5 guesses left.
Guess a letter: R
That guess is correct.
The word now looks like this: -O---TER
You have 5 guesses left.
Guess a letter: N
There are no N's in the word.
The word now looks like this: -O---TER
You have 4 guesses left.
Guess a letter: P
That guess is correct.
The word now looks like this: -O-P-TER
You have 4 guesses left.
Guess a letter: C
That guess is correct.
The word now looks like this: CO-P-TER
You have 4 guesses left.
Guess a letter: M
That guess is correct.
The word now looks like this: COMP-TER
You have 4 guesses left.
Guess a letter: U
That guess is correct.
You guessed the word: COMPUTER
You win.
```

- A function **CardName**(*card*) that returns a string identifying the card. The result of **CardName** begins with a rank indicator (which is one of A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, or K), followed by a one-character suit (C, D, H, or S). Note that the result is usually a two-character string, but contains three characters if the rank is a 10.
13. Using the **card.h** interface from the preceding exercise, write a program that initializes a complete deck of 52 cards, shuffles it, and then displays the shuffled values, as shown in the following sample run:



One of the easiest ways to shuffle the contents of an array is to adopt the strategy represented by the following pseudocode:

```

for (each position  $p_1$  in the array) {
    Pick a random position  $p_2$  between  $p_1$  and the end of the array.
    Exchange the values at positions  $p_1$  and  $p_2$  .
}

```

Chapter 4

Using Abstract Data Types

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

— Virginia Woolf, *Night and Day*, 1919

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 implemented 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

<code>size()</code>	Returns the number of elements in the vector.
<code>isEmpty()</code>	Returns <code>true</code> if the vector is empty.
<code>getAt(index)</code>	Returns the element of the vector that appears at the specified index position. As a convenience, the <code>Vector</code> class also makes it possible to select an element using array notation, so that <code>vec[i]</code> selects the element of <code>vec</code> at index position <code>i</code> .
<code>setAt(index, value)</code>	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.
<code>add(value)</code>	Adds a new element at the end of the vector.
<code>insertAt(index, value)</code>	Inserts the new value before the specified index position.
<code>removeAt(index)</code>	Deletes the element at the specified index position.
<code>clear()</code>	Removes all elements from the vector.
<code>iterator()</code>	Returns an <i>iterator</i> 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:

vec

10	20	30
0	1	2

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:

vec

10	20	30	40
0	1	2	3

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:

vec

10	20	25	30	40
0	1	2	3	4

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:

vec			
20	25	30	40
0	1	2	3

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:

vec			
20	25	30	35
0	1	2	3

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:



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 elements of an integer array to a vector:

```

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

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;
    }
}

```

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.

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:

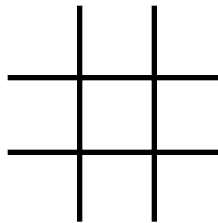
```
Grid<double> matrix(3, 2);
```

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.

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 <i>rows</i> and <i>cols</i> 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.

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 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 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.
*/

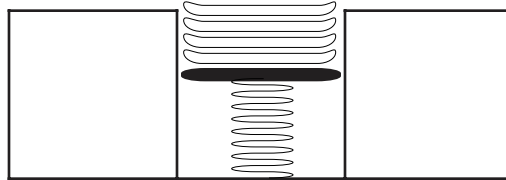
bool CheckForWin(Grid<char> & board, char mark) {
    for (int i = 0; i < 3; i++) {
        if (CheckLine(board, mark, i, 0, 0, 1)) return true;
        if (CheckLine(board, mark, 0, i, 1, 0)) return true;
    }
    if (CheckLine(board, mark, 0, 0, 1, 1)) return true;
    return CheckLine(board, mark, 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,
* these values will be +1 or -1 depending on the direction.
*/

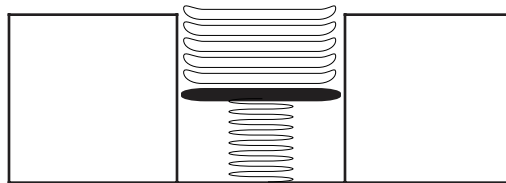
bool CheckLine(Grid<char> & board, char mark, int row, int col,
    int dRow, int dCol) {
    for (int i = 0; i < 3; i++) {
        if (board[row][col] != mark) return false;
        row += dRow;
        col += dCol;
    }
    return true;
}
```

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 **poppping** 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:

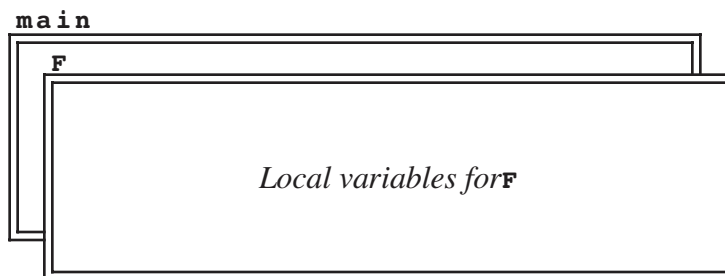


When a dishwasher adds a new plate, it goes on the top of the stack, pushing the others down slightly as the spring is compressed, as shown:

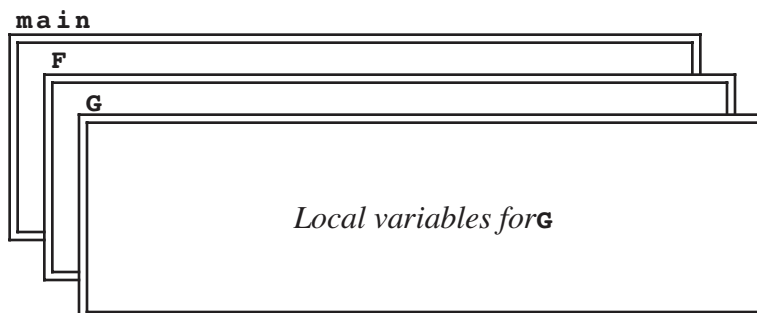


Customers can only take plates from the top of the stack. When they do, the remaining plates pop back up. The last plate added to the stack is the first one a customer takes.

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 **ENTER** **1.5** **x** **3.8** **ENTER** **2.0** **/** **+**

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 <i>value</i> 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.

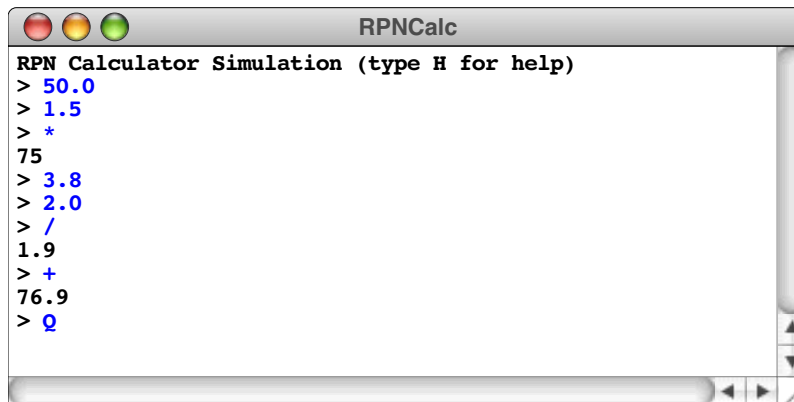
and stack contain the values shown:

<i>Buttons</i>	50.0	ENTER	1.5	(x)	3.8	ENTER	2.0	(/)	(+)
<i>Display</i>	50.0	50.0	1.5	75.0	3.8	3.8	2.0	1.9	76.9
<i>Stack</i>		50.0	50.0	75.0	75.0	3.8	3.8	1.9	76.9

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:



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;
}

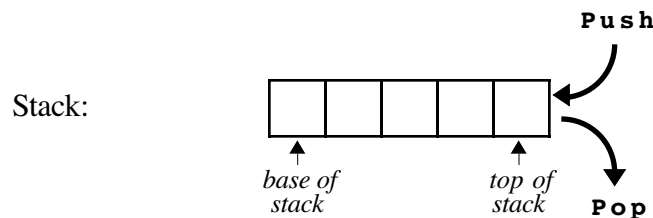
/*
 * Function: ClearStack
 * Usage: ClearStack(stack);
 * -----
 * This function clears the stack by popping elements until empty.
 */

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

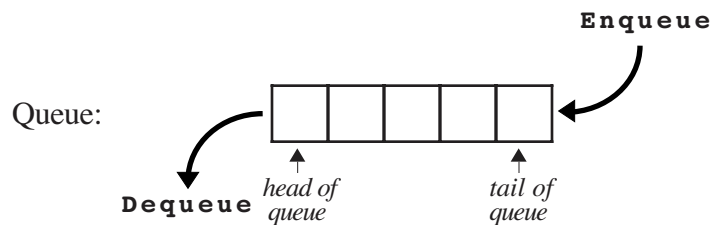
In real-world situations, however, its usefulness is more limited. In human society, our collective notion of fairness assigns some priority to being first, as expressed in the maxim “first come, first served.” In programming, the usual phrasing of this ordering strategy is “first in, first out,” which 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 <i>value</i> 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 to 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 waiting line itself is represented, naturally enough, as a queue. The value stored in the queue is the time at which that customer arrived in the queue, which makes it possible to determine how many seconds that customer spent in line before reaching the head of the queue.

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.

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 a randomly
 * chosen period of time. After the service time is complete,
 * the cashier is free to serve the next customer in the line.
 *
 * The waiting line is represented by a Queue<int> in which the
 * integer value stored in the queue is the time unit in which
 * that customer arrived. Storing this time makes it possible
 * to determine the average waiting time for each customer.
 *
 * 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 customers in the 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 = 0.10;
const int MIN_SERVICE_TIME = 5;
const int MAX_SERVICE_TIME = 15;

```

```
/* Private function prototypes */

void RunSimulation();
void ReportResults(int nServed, long totalWait, long totalLength);

/* Main program */

int main() {
    Randomize();
    RunSimulation();
    return 0;
}

/*
 * Function: RunSimulation
 * Usage: RunSimulation();
 * -----
 * 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 a nonzero value for
 * serviceTimeRemaining), the program decrements that variable to
 * indicate that one more time unit has passed. Finally, if the
 * cashier is free, the simulation serves another customer from
 * the queue after recording the waiting time for that customer.
 */

void RunSimulation() {
    Queue<int> queue;
    int serviceTimeRemaining = 0;
    int nServed = 0;
    long totalWait = 0;
    long totalLength = 0;
    for (int t = 0; t < SIMULATION_TIME; t++) {
        if (RandomChance(ARRIVAL_PROBABILITY)) {
            queue.enqueue(t);
        }
        if (serviceTimeRemaining > 0) {
            serviceTimeRemaining--;
            if (serviceTimeRemaining == 0) nServed++;
        } else if (!queue.isEmpty()) {
            totalWait += t - queue.dequeue();
            serviceTimeRemaining =
                RandomInteger(MIN_SERVICE_TIME, MAX_SERVICE_TIME);
        }
        totalLength += queue.size();
    }
    ReportResults(nServed, totalWait, totalLength);
}
```



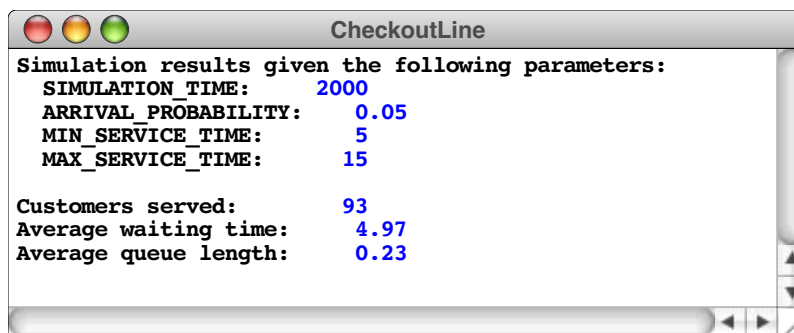
```

/*
 * Function: ReportResults
 * Usage: ReportResults(nServed, totalWait, totalLength);
 * -----
 * This function reports the results of the simulation.
 */

void ReportResults(int nServed, long totalWait, long totalLength) {
    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) << nServed << endl;
    cout << "Average waiting time:    " << setw(7)
        << double(totalWait) / nServed << endl;
    cout << "Average queue length:    " << setw(7)
        << double(totalLength) / SIMULATION_TIME << endl;
}

```

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



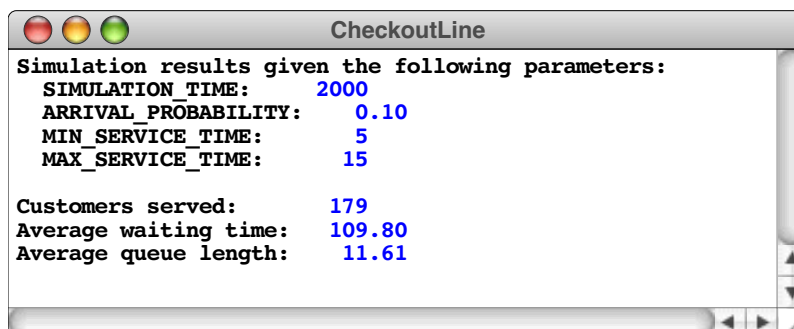
```

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

Customers served:    93
Average waiting time: 4.97
Average queue length: 0.23

```

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:



```

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

Customers served:    179
Average waiting time: 109.80
Average queue length: 11.61

```

As you can see, doubling the probability of arrival causes the average waiting time to grow from approximately 5 seconds to over a minute and a half, 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 that 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 avoids 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.

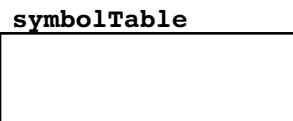
The most common methods used with the **Map** class appear in Table 4-5. Of these, the ones that implement the fundamental behavior of the map concept are **put** and **get**. The

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 <i>key</i> and <i>value</i> in the map. If <i>key</i> has no previous definition, a new entry is added; if a previous association exists, the old value is discarded and replaced by the new one.
get(key)	Returns the value currently associated with <i>key</i> in the map. If there is no such value, get generates an error.
remove(key)	Removes <i>key</i> from the map along with any associated value. If <i>key</i> does not exist, this call leaves the map unchanged.
containsKey(key)	Checks to see whether <i>key</i> 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.

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 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.

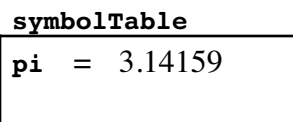
A few simple diagrams may help to illustrate the operation of the **Map** class in more detail. Suppose that you have declared the **symbolTable** variable to be a **Map<double>** as you saw earlier in the section. That declaration creates an empty map with no associations, which can be represented as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

symbolTable	
pi	= 3.14159
e	= 2.71828

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

symbolTable	
pi	= 3.2
e	= 2.71828

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 in 2009, as compiled by Airports Council International, 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:



Figure 4-5. The first page of the AirportCodes.txt data file

```
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
JFK=New York, NY, USA
AMS=Amsterdam, Netherlands
LAS=Las Vegas, NV, USA
HKG=Hong Kong, Hong Kong
IAH=Houston, TX, USA
PHX=Phoenix, AZ, USA
BKK=Bangkok, Thailand
SIN=Singapore, Singapore
EWR=Newark, NJ, USA
MCO=Orlando, FL, USA
DTW=Detroit, MI, USA
SFO=San Francisco, CA, USA
NRT=Tokyo, Japan
LGW=London, England, United Kingdom
MSP=Minneapolis, MN, USA
DXB=Dubai, United Arab Emirates
MUC=Munich, Germany
MIA=Miami, FL, USA
CLT=Charlotte, NC, USA
FCO=Rome, Italy
BCN=Barcelona, Spain
SYD=Sydney, New South Wales, Australia
PHL=Philadelphia, PA, USA
CGK=Jakarta, Indonesia
YYZ=Toronto, Ontario, Canada
ICN=Incheon [Seoul], South Korea
SEA=Seattle, WA, USA
CAN=Guangzhou, China
PVG=Shanghai, China
BOS=Boston, MA, USA
KUL=Kuala Lumpur, Malaysia
ORY=Paris, France
MEX=Mexico City, Distrito Federal, Mexico
IST=Istanbul, Turkey
BOM=Bombay (Mumbai), India
LGA=New York, NY, USA
IAD=Washington, DC, USA
MXP=Milan, Italy
STN=London, England, United Kingdom
TPE=Taipei, Taiwan
DEL=Delhi, India
DUB=Dublin, Ireland
PMI=Palma Mallorca, Mallorca Island, Spain
.
.
.
```

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

scores				
9.2	9.9	9.7	9.0	9.5
0	1	2	3	4

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 following 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 <i>word</i> 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:

This		line		contains		10		tokens	.

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

```
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 **** (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 (<i>str</i>)	Sets the scanner input source so that new tokens are taken from the string <i>str</i> . Any unread tokens remaining from the preceding call to setInput are discarded.
setInput (<i>stream</i>)	Sets the scanner input source so that new tokens are taken from the input stream <i>stream</i> . 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 (<i>token</i>)	Stores the specified token in the private state of the scanner so that it will be returned the next time nextToken is called.
setSpaceOption (<i>option</i>) 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 <i>option</i> parameter must be one of the following constant names: Scanner::PreserveSpaces Scanner::IgnoreSpaces
setNumberOption (<i>option</i>) getNumberOption ()	Allows the client to control whether numeric strings are recognized as single tokens. The <i>option</i> 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 (<i>option</i>) getStringOption ()	Allows the client to control whether quoted strings are recognized as single tokens. The <i>option</i> 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 (<i>option</i>) getBracketOption ()	Allows the client to control whether HTML tags enclosed in angle brackets (such as <p> or) are recognized as single tokens. The <i>option</i> 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**.

The foreach idiom

Even though the iterator pattern introduced in the preceding section takes up just a few lines of code, I have discovered over the years that the details of the iterator mechanism get in the way of understanding the purpose of the iterator, which is simply to step through the elements of a collection, one at a time. If you were trying to express the algorithmic function of an iterator in English pseudocode, you would want to write something like this:

```
For each element in a particular collection {
    Process that element
}
```

Some languages, most notably C# and Java, define a new syntactic form that expresses precisely that idea. Unfortunately, the syntax of C++ does not such a facility.

The good news, however, is that it is possible to use the macro-definition capabilities of the C++ preprocessor to achieve exactly what you would like to see in the language. Although the implementation details are beyond the scope of this text, the collection classes that support iteration also define a **foreach** macro that has the following form:

```
foreach (string word in english) {
    if (word.length() == 2) {
        cout << word << endl;
    }
}
```

The advantage of the **foreach** syntax is not that the code is a couple of lines shorter, but rather that the revised code tells the reader exactly what is going on. If you compare the two versions of this loop, I'm sure you'll immediately recognize just how much clearer the **foreach** version is. In my experience, once students have been given the opportunity to use **foreach** in their code, they *never* go back to using the iterator form. Even so, it is useful to know that the iterator form exists and that the iterator mechanism is in fact what is going on inside the implementation of **foreach**.

Iteration order

When you work with iterators or the **foreach** macro, it is often useful to understand the order in which the iterator generates each of the individual values. There is no universal rule. Each container class defines its own policy about iteration order, usually based on considerations of efficiency. 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.

An example of the **foreach** mechanism

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 english("EnglishWords.dat");
    foreach (string word in english) {
        string pig = PigLatin(word);
        if (pig != word && english.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`, you 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 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

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.next_token());
        if (IsAllAlpha(word)) {
            if (wordCounts.containsKey(word)) {
                wordCounts[word]++;
            } else {
                wordCounts[word] = 1;
            }
        }
    }
}

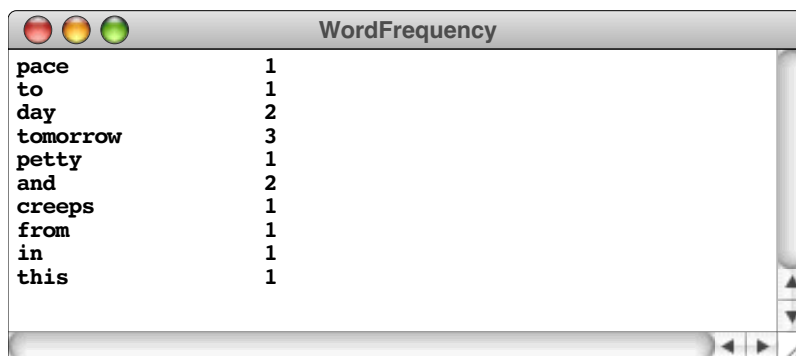
/*
 * Displays the count for each word in the frequency table.
 */

void DisplayWordCounts(Map<int> & wordCounts) {
    foreach (string word in wordCounts) {
        cout << left << setw(15) << word
              << right << setw(5) << wordCounts[word] << endl;
    }
}

```

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 8.

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 than 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 8×8 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. If you use the **iterator** method explicitly, what is the standard idiom for using an iterator to cycle through the elements of a collection?
32. What is the **foreach** version of this standard iterator idiom?
33. What is the principal advantage of using **foreach** in preference to the explicit iterator-based code?
34. True or false: The iterator for the **Map** class guarantees that individual keys will be delivered in alphabetical order.
35. True or false: The iterator for the **Lexicon** class guarantees that individual words will be delivered in alphabetical order.
36. 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 asked 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(istream & 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 **istream** on that file. In addition, suppose that you have declared 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(istream & 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 3x3 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 4×4 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 3×3 magic square in which each of the rows, columns, and diagonals add up to 15, as shown:

8	1	6	=15
3	5	7	=15
4	9	2	=15

=15	8	1	6	=15
=15	3	5	7	=15
=15	4	9	2	

8	1	6	=15
3	5	7	
4	9	2	=15

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 9×9 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 3×3 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		

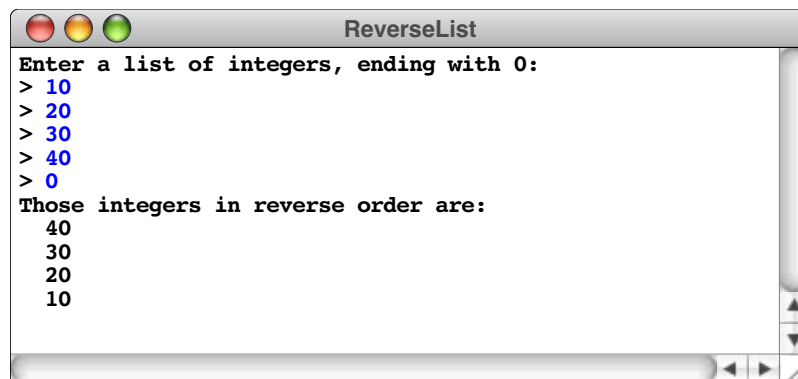
3	9	2	4	6	5	8	1	7
7	4	1	8	9	3	6	2	5
6	8	5	2	7	1	4	3	9
2	5	4	1	3	8	7	9	6
8	3	9	6	2	7	1	5	4
1	7	6	9	5	4	2	8	3
9	6	7	5	8	2	3	4	1
4	2	3	7	1	9	5	6	8
5	1	8	3	4	6	9	7	2

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.

- 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:



- 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:

(((1))	<i>The line is missing a close parenthesis.</i>
) (	<i>The close parenthesis comes before the open parenthesis.</i>
{ (}	<i>The bracketing operators are improperly nested.</i>

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:

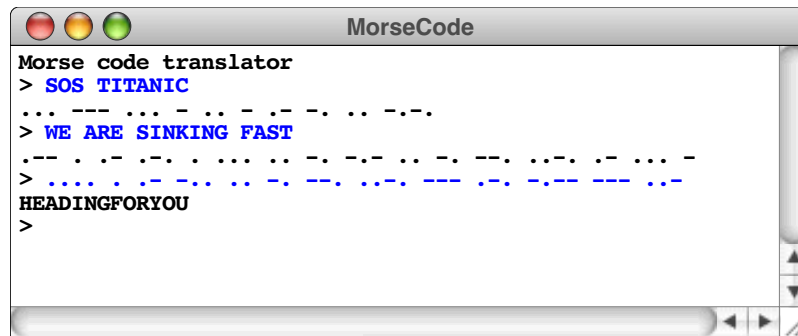
A	• —	J	• — — —	S	• • •
B	— • • •	K	— • —	T	—
C	— • — •	L	• — • •	U	• • —
D	— • •	M	— —	V	• • • —
E	•	N	— •	W	• — —
F	• • — •	O	— — —	X	— • • —
G	— — •	P	• — — •	Y	— • — —
H	• • • •	Q	— — • —	Z	— — • •
I	• •	R	• — •		

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. Because there is no encoding for the space between words, the characters of the translated message will be run together when your program translates in this direction.

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:



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**.

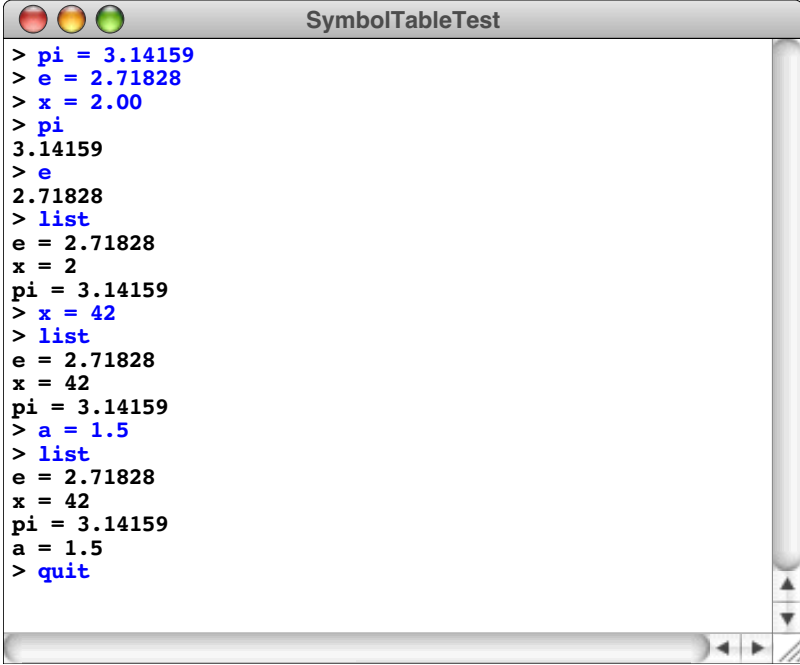
- The name of a variable alone on a line. When your program reads in such a line, it should print out the current value in the symbol table associated with that name. Thus, if **pi** has been defined as shown in the preceding example, the command

```
pi
```

should display the value 3.14159.

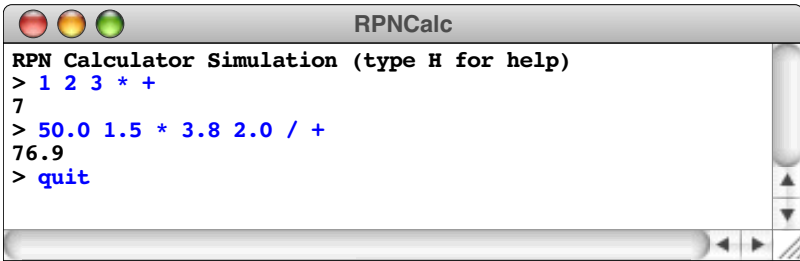
- The command **list**, which is interpreted by the program as a request to display all variable/value pairs currently stored in the symbol table, not necessarily in any easily discernable order.
- The command **quit**, which should exit from the program.

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



```
> 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
```

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 (type H for help)
> 1 2 3 * +
7
> 50.0 1.5 * 3.8 2.0 / +
76.9
> quit
```

Chapter 5

Introduction to Recursion

*And often enough, our faith beforehand in a certain result
is the only thing that makes the result come true.*

— William James, *The Will To Believe*, 1897

Most algorithmic strategies used to solve programming problems have counterparts outside the domain of computing. When you perform a task repeatedly, you are using iteration. When you make a decision, you exercise conditional control. Because these operations are familiar, most people learn to use the control statements **for**, **while**, and **if** with relatively little trouble.

Before you can solve many sophisticated programming tasks, however, you will have to learn to use a powerful problem-solving strategy that has few direct counterparts in the real world. That strategy, called **recursion**, is defined as any solution technique in which large problems are solved by reducing them to smaller problems *of the same form*. The italicized phrase is crucial to the definition, which otherwise describes the basic strategy of stepwise refinement. Both strategies involve decomposition. What makes recursion special is that the subproblems in a recursive solution have the same form as the original problem.

If you are like most beginning programmers, the idea of breaking a problem down into subproblems of the same form does not make much sense when you first hear it. Unlike repetition or conditional testing, recursion is not a concept that comes up in day-to-day life. Because it is unfamiliar, learning how to use recursion can be difficult. To do so, you must develop the intuition necessary to make recursion seem as natural as all the other control structures. For most students of programming, reaching that level of understanding takes considerable time and practice. Even so, learning to use recursion is definitely worth the effort. As a problem-solving tool, recursion is so powerful that it at times seems almost magical. In addition, using recursion often makes it possible to write complex programs in simple and profoundly elegant ways.

5.1 A simple example of recursion

To gain a better sense of what recursion is, let's imagine you have been appointed as the funding coordinator for a large charitable organization that is long on volunteers and short on cash. Your job is to raise \$1,000,000 in contributions so the organization can meet its expenses.

If you know someone who is willing to write a check for the entire \$1,000,000, your job is easy. On the other hand, you may not be lucky enough to have friends who are generous millionaires. In that case, you must raise the \$1,000,000 in smaller amounts. If the average contribution to your organization is \$100, you might choose a different tack: call 10,000 friends and ask each of them for \$100. But then again, you probably don't have 10,000 friends. So what can you do?

As is often the case when you are faced with a task that exceeds your own capacity, the answer lies in delegating part of the work to others. Your organization has a reasonable supply of volunteers. If you could find 10 dedicated supporters in different parts of the country and appoint them as regional coordinators, each of those 10 people could then take responsibility for raising \$100,000.

Raising \$100,000 is simpler than raising \$1,000,000, but it hardly qualifies as easy. What should your regional coordinators do? If they adopt the same strategy, they will in turn delegate parts of the job. If they each recruit 10 fundraising volunteers, those people will only have to raise \$10,000. The delegation process can continue until the volunteers are able to raise the money on their own; because the average contribution is \$100, the volunteer fundraisers can probably raise \$100 from a single donor, which eliminates the need for further delegation.

If you express this fundraising strategy in pseudocode, it has the following structure:

```
void CollectContributions(int n) {
    if (n <= 100) {
        Collect the money from a single donor.
    } else {
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
    }
}
```

The most important thing to notice about this pseudocode translation is that the line

Get each volunteer to collect n/10 dollars.

is simply the original problem reproduced at a smaller scale. The basic character of the task—raise n dollars—remains exactly the same; the only difference is that n has a smaller value. Moreover, because the problem is the same, you can solve it by calling the original function. Thus, the preceding line of pseudocode would eventually be replaced with the following line:

```
CollectContributions(n / 10);
```

It's important to note that the **CollectContributions** function ends up calling itself if the contribution level is greater than \$100. In the context of programming, having a function call itself is the defining characteristic of recursion.

The structure of the **CollectContributions** procedure is typical of recursive functions. In general, the body of a recursive function has the following form:

```
if (test for simple case) {
    Compute a simple solution without using recursion.
} else {
    Break the problem down into subproblems of the same form.
    Solve each of the subproblems by calling this function recursively.
    Reassemble the solutions to the subproblems into a solution for the whole.
}
```

This structure provides a template for writing recursive functions and is therefore called the **recursive paradigm**. You can apply this technique to programming problems as long as they meet the following conditions:

1. You must be able to identify **simple cases** for which the answer is easily determined.
2. You must be able to identify a **recursive decomposition** that allows you to break any complex instance of the problem into simpler problems of the same form.

The **CollectContributions** example illustrates the power of recursion. As in any recursive technique, the original problem is solved by breaking it down into smaller subproblems that differ from the original only in their scale. Here, the original problem is to raise \$1,000,000. At the first level of decomposition, each subproblem is to raise \$100,000. These problems are then subdivided in turn to create smaller problems until the problems are simple enough to be solved immediately without recourse to further subdivision. Because the solution depends on dividing hard problems into simpler ones, recursive solutions are often called **divide-and-conquer** strategies.

5.2 The factorial function

Although the **CollectContributions** example illustrates the concept of recursion, it gives little insight into how recursion is used in practice, mostly because the steps that make up the solution, such as finding 10 volunteers and collecting money, are not easily represented in a C++ program. To get a practical sense of the nature of recursion, you need to consider problems that fit more easily into the programming domain.

For most people, the best way to understand recursion is to start with simple mathematical functions in which the recursive structure follows directly from the statement of the problem and is therefore easy to see. Of these, the most common is the factorial function—traditionally denoted in mathematics as $n!$ —which is defined as the product of the integers between 1 and n . In C++, the equivalent problem is to write an implementation of a function with the prototype

```
int Fact(int n);
```

that takes an integer n and returns its factorial.

As you probably discovered in an earlier programming course, it is easy to implement the **Fact** function using a **for** loop, as illustrated by the following implementation:

```
int Fact(int n) {  
    int product;  
  
    product = 1;  
    for (int i = 1; i <= n; i++) {  
        product *= i;  
    }  
    return product;  
}
```

This implementation uses a **for** loop to cycle through each of the integers between 1 and n . In the recursive implementation this loop does not exist. The same effect is generated instead by the cascading recursive calls.

Implementations that use looping (typically by using **for** and **while** statements) are said to be **iterative**. Iterative and recursive strategies are often seen as opposites because they can be used to solve the same problem in rather different ways. These strategies, however, are not mutually exclusive. Recursive functions sometimes employ iteration internally, and you will see examples of this technique in Chapter 6.

The recursive formulation of **Fact**

The iterative implementation of **Fact**, however, does not take advantage of an important mathematical property of factorials. Each factorial is related to the factorial of the next smaller integer in the following way:

$$n! = n \times (n - 1)!$$

Thus, $4!$ is $4 \times 3!$, $3!$ is $3 \times 2!$, and so on. To make sure that this process stops at some point, mathematicians define $0!$ to be 1. Thus, the conventional mathematical definition of the factorial function looks like this:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

This definition is recursive, because it defines the factorial of n in terms of the factorial of $n-1$. The new problem—finding the factorial of $n-1$ —has the same form as the original problem, which is the fundamental characteristic of recursion. You can then use the same process to define $(n-1)!$ in terms of $(n-2)!$. Moreover, you can carry this process forward step by step until the solution is expressed in terms of $0!$, which is equal to 1 by definition.

From your perspective as a programmer, the practical impact of the mathematical definition is that it provides a template for a recursive implementation. In C++, you can implement a function **Fact** that computes the factorial of its argument as follows:

```
int Fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * Fact(n - 1);
    }
}
```

If n is 0, the result of **Fact** is 1. If not, the implementation computes the result by calling **Fact**($n-1$) and then multiplying the result by n . This implementation follows directly from the mathematical definition of the factorial function and has precisely the same recursive structure.

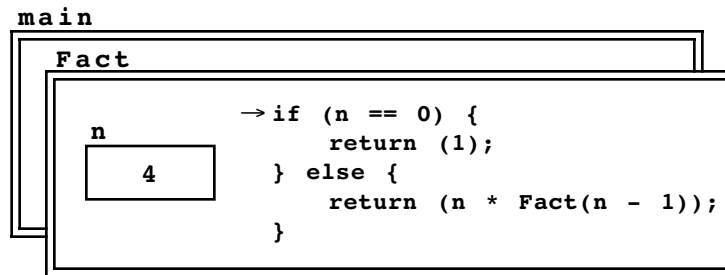
Tracing the recursive process

If you work from the mathematical definition, writing the recursive implementation of **Fact** is straightforward. On the other hand, even though the definition is easy to write, the brevity of the solution may seem suspicious. When you are learning about recursion for the first time, the recursive implementation of **Fact** seems to leave something out. Even though it clearly reflects the mathematical definition, the recursive formulation makes it hard to identify where the actual computational steps occur. When you call **Fact**, for example, you want the computer to give you the answer. In the recursive implementation, all you see is a formula that transforms one call to **Fact** into another one. Because the steps in that calculation are not explicit, it seems somewhat magical when the computer gets the right answer.

If you follow through the logic the computer uses to evaluate any function call, however, you discover that no magic is involved. When the computer evaluates a call to the recursive **Fact** function, it goes through the same process it uses to evaluate any other function call. To visualize the process, suppose that you have executed the statement

```
f = Fact(4);
```

as part of the function **main**. When **main** calls **Fact**, the computer creates a new stack frame and copies the argument value into the formal parameter **n**. The frame for **Fact** temporarily supersedes the frame for **main**, as shown in the following diagram:

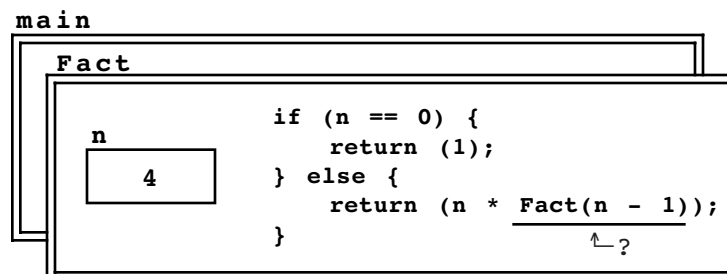


In the diagram, the code for the body of **Fact** is shown inside the frame to make it easier to keep track of the current position in the program, which is indicated by an arrow. In the current diagram, the arrow appears at the beginning of the code because all function calls start at the first statement of the function body.

The computer now begins to evaluate the body of the function, starting with the **if** statement. Because **n** is not equal to 0, control proceeds to the **else** clause, where the program must evaluate and return the value of the expression

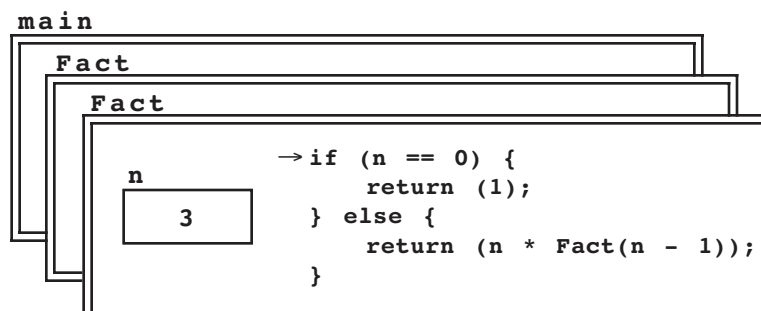
n * Fact(n - 1)

Evaluating this expression requires computing the value of **Fact(n - 1)**, which introduces a recursive call. When that call returns, all the program has to do is to multiply the result by **n**. The current state of the computation can therefore be diagrammed as follows:



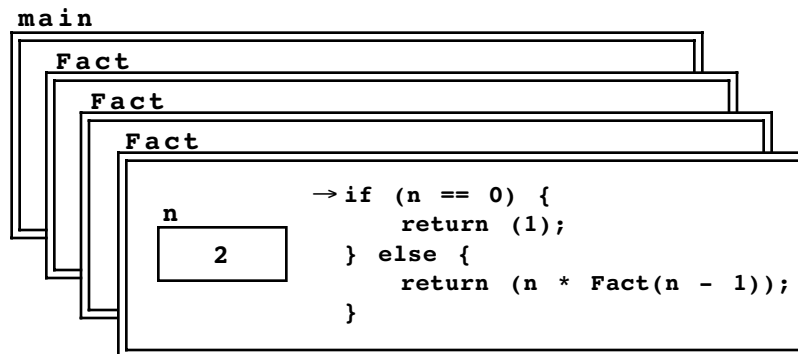
As soon as the call to **Fact(n - 1)** returns, the result is substituted for the expression underlined in the diagram, allowing computation to proceed.

The next step in the computation is to evaluate the call to **Fact(n - 1)**, beginning with the argument expression. Because the current value of **n** is 4, the argument expression **n - 1** has the value 3. The computer then creates a new frame for **Fact** in which the formal parameter is initialized to this value. Thus, the next frame looks like this:

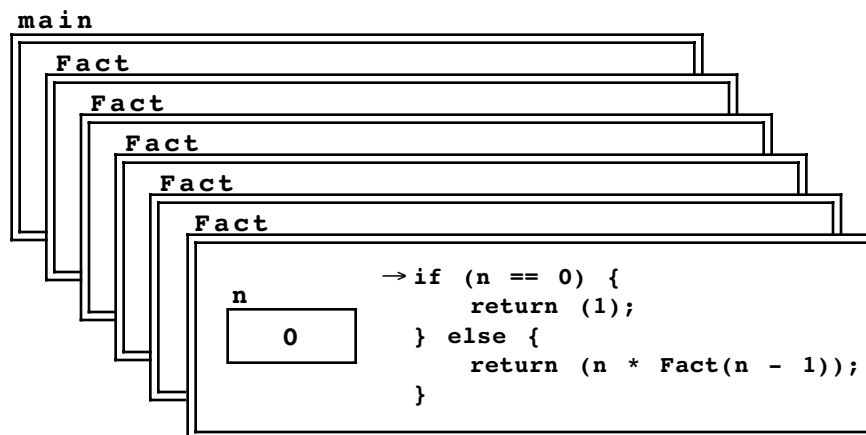


There are now two frames labeled **Fact**. In the most recent one, the computer is just starting to calculate **Fact(3)**. In the preceding frame, which the newly created frame hides, the **Fact** function is awaiting the result of the call to **Fact(n - 1)**.

The current computation, however, is the one required to complete the topmost frame. Once again, **n** is not 0, so control passes to the **else** clause of the **if** statement, where the computer must evaluate **Fact(n - 1)**. In this frame, however, **n** is equal to 3, so the required result is that computed by calling **Fact(2)**. As before, this process requires the creation of a new stack frame, as shown:



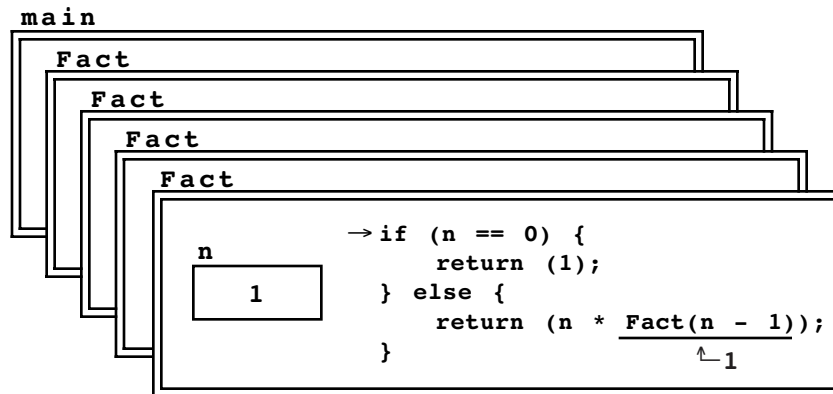
Following the same logic, the program must now call **Fact(1)**, which in turn calls **Fact(0)**, thereby creating two new stack frames. The resulting stack configuration looks like this:



At this point, however, the situation changes. Because the value of **n** is 0, the function can return its result immediately by executing the statement

```
return 1;
```

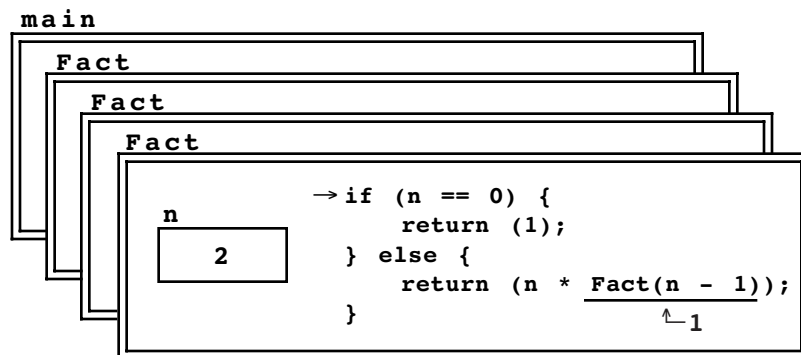
The value 1 is returned to the calling frame, which resumes its position on top of the stack, as shown:



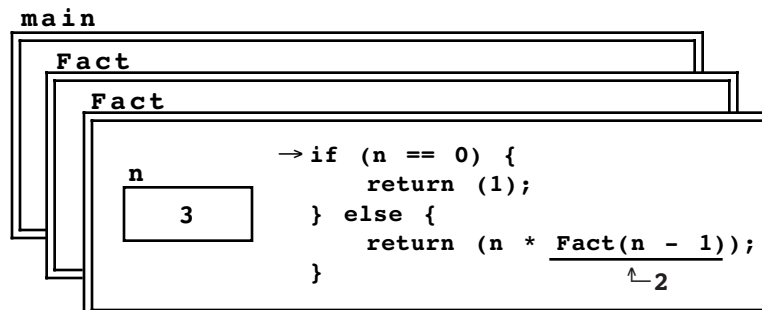
From this point, the computation proceeds back through each of the recursive calls, completing the calculation of the return value at each level. In this frame, for example, the call to `Fact(n - 1)` can be replaced by the value 1, so that the result at this level can be expressed as follows:

```
return n * 1);
```

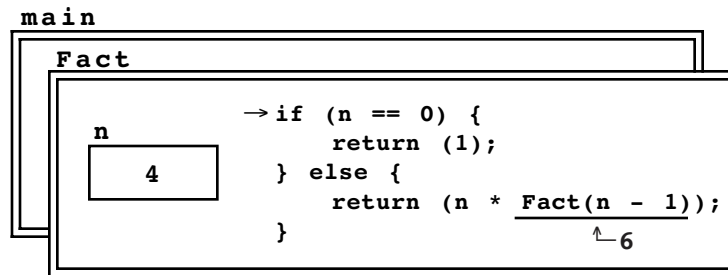
In this stack frame, `n` has the value 1, so the result of this call is simply 1. This result gets propagated back to its caller, which is represented by the top frame in the following diagram:



Because `n` is now 2, evaluating the `return` statement causes the value 2 to be passed back to the previous level, as follows:



At this stage, the program returns 3×2 to the previous level, so that the frame for the initial call to `Fact` looks like this:



The final step in the calculation process consists of calculating 4×6 and returning the value 24 to the main program.

The recursive leap of faith

The point of the long **Fact(4)** example in the preceding section is to show you that the computer treats recursive functions just like all other functions. When you are faced with a recursive function, you can—at least in theory—mimic the operation of the computer and figure out what it will do. By drawing all the frames and keeping track of all the variables, you can duplicate the entire operation and come up with the answer. If you do so, however, you will usually find that the complexity of the process ends up making the problem much harder to understand.

When you try to understand a recursive program, you must be able to put the underlying details aside and focus instead on a single level of the operation. At that level, you are allowed to assume that any recursive call automatically gets the right answer as long as the arguments to that call are simpler than the original arguments in some respect. This psychological strategy—assuming that any simpler recursive call will work correctly—is called the **recursive leap of faith**. Learning to apply this strategy is essential to using recursion in practical applications.

As an example, consider what happens when this implementation is used to compute **Fact(n)** with **n** equal to 4. To do so, the recursive implementation must compute the value of the expression

n * Fact(n - 1)

By substituting the current value of **n** into the expression, you know that the result is

4 * Fact(3)

Stop right there. Computing **Fact(3)** is simpler than computing **Fact(4)**. Because it is simpler, the recursive leap of faith allows you to assume that it works. Thus, you should assume that the call to **Fact(3)** will correctly compute the value of $3!$, which is $3 \times 2 \times 1$, or 6. The result of calling **Fact(4)** is therefore 4×6 , or 24.

As you look at the examples in the rest of this chapter, try to focus on the big picture instead of the morass of detail. Once you have made the recursive decomposition and identified the simple cases, be satisfied that the computer can handle the rest.

5.3 The Fibonacci function

In a mathematical treatise entitled *Liber Abbaci* published in 1202, the Italian mathematician Leonardo Fibonacci proposed a problem that has had a wide influence on many fields, including computer science. The problem was phrased as an exercise in

population biology—a field that has become increasingly important in recent years. Fibonacci’s problem concerns how the population of rabbits would grow from generation to generation if the rabbits reproduced according to the following, admittedly fanciful, rules:

- Each pair of fertile rabbits produces a new pair of offspring each month.
- Rabbits become fertile in their second month of life.
- Old rabbits never die.

If a pair of newborn rabbits is introduced in January, how many pairs of rabbits are there at the end of the year?

You can solve Fibonacci’s problem simply by keeping a count of the rabbits at each month during the year. At the beginning of January, there are no rabbits, since the first pair is introduced sometime in that month, which leaves one pair of rabbits on February 1. Since the initial pair of rabbits is newborn, they are not yet fertile in February, which means that the only rabbits on March 1 are the original pair of rabbits. In March, however, the original pair is now of reproductive age, which means that a new pair of rabbits is born. The new pair increases the colony’s population—counting by pairs—to two on April 1. In April, the original pair goes right on reproducing, but the rabbits born in March are as yet too young. Thus, there are three pairs of rabbits at the beginning of May. From here on, with more and more rabbits becoming fertile each month, the rabbit population begins to grow more quickly.

Computing terms in the Fibonacci sequence

At this point, it is useful to record the population data so far as a sequence of terms, indicated here by the subscripted value t_i , each of which shows the number of rabbit pairs at the beginning of the i th month from the start of the experiment on January 1. The sequence itself is called the **Fibonacci sequence** and begins with the following terms, which represent the results of our calculation so far:

t_0	t_1	t_2	t_3	t_4
0	1	1	2	3

You can simplify the computation of further terms in this sequence by making an important observation. Because rabbits in this problem never die, all the rabbits that were around in the previous month are still around. Moreover, all of the fertile rabbits have produced a new pair. The number of fertile rabbit pairs capable of reproduction is simply the number of rabbits that were alive in the month before the previous one. The net effect is that each new term in the sequence must simply be the sum of the preceding two. Thus, the next several terms in the Fibonacci sequence look like this:

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
0	1	1	2	3	5	8	13	21	34	55	89	144

The number of rabbit pairs at the end of the year is therefore 144.

From a programming perspective, it helps to express the rule for generating new terms in the following, more mathematical form:

$$t_n = t_{n-1} + t_{n-2}$$

An expression of this type, in which each element of a sequence is defined in terms of earlier elements, is called a **recurrence relation**.

The recurrence relation alone is not sufficient to define the Fibonacci sequence. Although the formula makes it easy to calculate new terms in the sequence, the process has to start somewhere. In order to apply the formula, you need to have at least two terms in hand, which means that the first two terms in the sequence— t_0 and t_1 —must be defined explicitly. The complete specification of the terms in the Fibonacci sequence is therefore

$$t_n = \begin{cases} n & \text{if } n \text{ is 0 or 1} \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

This mathematical formulation is an ideal model for a recursive implementation of a function **Fib(n)** that computes the n th term in the Fibonacci sequence. All you need to do is plug the simple cases and the recurrence relation into the standard recursive paradigm. The recursive implementation of **Fib(n)** is shown in Figure 5-1, which also includes a test program that displays the terms in the Fibonacci sequence between two specified indices.

Gaining confidence in the recursive implementation

Now that you have a recursive implementation of the function **Fib**, how can you go about convincing yourself that it works? You can always begin by tracing through the logic. Consider, for example, what happens if you call **Fib(5)**. Because this is not one of the simple cases enumerated in the **if** statement, the implementation computes the result by evaluating the line

```
return Fib(n - 1) + Fib(n - 2);
```

which is in this case equivalent to

```
return Fib(4) + Fib(3);
```

At this point, the computer calculates the result of **Fib(4)**, adds that to the result of calling **Fib(3)**, and returns the sum as the value of **Fib(5)**.

But how does the computer go about evaluating **Fib(4)** and **Fib(3)**? The answer, of course, is that it uses precisely the same strategy. The essence of recursion is to break problems down into simpler ones that can be solved by calls to exactly the same function. Those calls get broken down into simpler ones, which in turn get broken down into even simpler ones, until at last the simple cases are reached.

On the other hand, it is best to regard this entire mechanism as irrelevant detail. Remember the recursive leap of faith. Your job at this level is to understand how the call to **Fib(5)** works. In the course of walking through the execution of that function, you have managed to transform the problem into computing the sum of **Fib(4)** and **Fib(3)**. Because the argument values are smaller, each of these calls represents a simpler case. Applying the recursive leap of faith, you can assume that the program correctly computes each of these values, without going through all the steps yourself. For the purposes of validating the recursive strategy, you can just look the answers up in the table. **Fib(4)** is 3 and **Fib(3)** is 2, so the result of calling **Fib(5)** is 3 + 2, or 5, which is indeed the correct answer. Case closed. You don't need to see all the details, which are best left to the computer.

Figure 5-1 Recursive implementation of the Fibonacci function

```
/*
 * File: fib.cpp
 * -----
 * This program lists the terms in the Fibonacci sequence with
 * indices ranging from MIN_INDEX to MAX_INDEX.
 */

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

/*
 * Constants
 * -----
 * MIN_INDEX -- Index of first term to generate
 * MAX_INDEX -- Index of last term to generate
 */

const int MIN_INDEX = 0;
const int MAX_INDEX = 12;

/* Private function prototypes */

int Fib(int n);

/* Main program */

int main() {
    cout << "This program lists the Fibonacci sequence." << endl;
    for (int i = MIN_INDEX; i <= MAX_INDEX; i++) {
        cout << "Fib(" << i << ")";
        if (i < 10) cout << " ";
        cout << " = " << Fib(i) << endl;
    }
    return 0;
}

/*
 * Function: Fib
 * Usage: t = Fib(n);
 * -----
 * This function returns the nth term in the Fibonacci sequence
 * using a recursive implementation of the recurrence relation
 *
 *      Fib(n) = Fib(n - 1) + Fib(n - 2)
 */

int Fib(int n) {
    if (n < 2) {
        return n;
    } else {
        return Fib(n - 1) + Fib(n - 2);
    }
}
```


0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . .

comes from defining $t_0 = 0$ and $t_1 = 1$. If, for example, you defined $t_0 = 3$ and $t_1 = 7$, you would get this sequence instead:

3, 7, 10, 17, 27, 44, 71, 115, 186, 301, 487, 788, 1275, . . .

Similarly, defining $t_0 = -1$ and $t_1 = 2$ gives rise to the following sequence:

-1, 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, . . .

These sequences all use the same recurrence relation, which specifies that each new term is the sum of the previous two. The only way the sequences differ is in the choice of the first two terms. As a general class, the sequences that follow this pattern are called **additive sequences**.

This concept of an additive sequence makes it possible to convert the problem of finding the n^{th} term in the Fibonacci sequence into the more general problem of finding the n^{th} term in an additive sequence whose initial terms are t_0 and t_1 . Such a function requires three arguments and might be expressed in C++ as a function with the following prototype:

```
int AdditiveSequence(int n, int t0, int t1);
```

If you had such a function, it would be easy to implement **Fib** using it. All you would need to do is supply the correct values of the first two terms, as follows:

```
int Fib(int n) {
    return AdditiveSequence(n, 0, 1);
}
```

The body consists of a single line of code that does nothing but call another function, passing along a few extra arguments. Functions of this sort, which simply return the result of another function, often after transforming the arguments in some way, are called **wrapper** functions. Wrapper functions are extremely common in recursive programming. In most cases, a wrapper function is used—as it is here—to supply additional arguments to a subsidiary function that solves a more general problem.

From here, the only remaining task is to implement the function **AdditiveSequence**. If you think about this more general problem for a few minutes, you will discover that additive sequences have an interesting recursive character of their own. The simple case for the recursion consists of the terms t_0 and t_1 , whose values are part of the definition of the sequence. In the C++ implementation, the value of these terms are passed as arguments. If you need to compute t_0 , for example, all you have to do is return the argument **t0**.

But what if you are asked to find a term further down in the sequence? Suppose, for example, that you want to find t_6 in the additive sequence whose initial terms are 3 and 7. By looking at the list of terms in the sequence

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	
3	7	10	17	27	44	71	115	186	301	. . .

you can see that the correct value is 71. The interesting question, however, is how you can use recursion to determine this result.

The key insight you need to discover is that the n th term in any additive sequence is simply the $n-1$ st term in the additive sequence which begins one step further along. For example, t_6 in the sequence shown in the most recent example is simply t_5 in the additive sequence

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	...
7	10	17	27	44	71	115	186	301	...

that begins with 7 and 10.

This discovery makes it possible to implement the function **AdditiveSequence** as follows:

```
int AdditiveSequence(int n, int t0, int t1) {
    if (n == 0) return t0;
    if (n == 1) return t1;
    return AdditiveSequence(n - 1, t1, t0 + t1);
}
```

If you trace through the steps in the calculation of **Fib(5)** using this technique, you will discover that the calculation involves none of the redundant computation that plagued the earlier recursive formulation. The steps lead directly to the solution, as shown in the following diagram:

```
Fib(5)
  = AdditiveSequence(5, 0, 1)
    = AdditiveSequence(4, 1, 1)
      = AdditiveSequence(3, 1, 2)
        = AdditiveSequence(2, 2, 3)
          = AdditiveSequence(1, 3, 5)
            = 5
```

Even though the new implementation is entirely recursive, it is comparable in efficiency to the standard iterative version of the Fibonacci function.

5.4 Other examples of recursion

Although the factorial and Fibonacci functions provide excellent examples of how recursive functions work, they are both mathematical in nature and may therefore convey the incorrect impression that recursion is applicable only to mathematical functions. In fact, you can apply recursion to any problem that can be decomposed into simpler problems of the same form. It is useful to consider a few additional examples, including several that are far less mathematical in their character.

Detecting palindromes

A **palindrome** is a string that reads identically backward and forward, such as "**level**" or "**noon**". Although it is easy to check whether a string is a palindrome by iterating through its characters, palindromes can also be defined recursively. The insight you need to do so is that any palindrome longer than a single character must contain a shorter palindrome in its interior. For example, the string "**level**" consists of the palindrome "**eve**" with an "**l**" at each end. Thus, to check whether a string is a palindrome—assuming the string is sufficiently long that it does not constitute a simple case—all you need to do is

1. Check to see that the first and last characters are the same.
2. Check to see whether the substring generated by removing the first and last characters is itself a palindrome.

If both conditions apply, the string is a palindrome.

The only other question you must consider before writing a recursive solution to the palindrome problem is what the simple cases are. Clearly, any string with only a single character is a palindrome because reversing a one-character string has no effect. The one-character string therefore represents a simple case, but it is not the only one. The empty string—which contains no characters at all—is also a palindrome, and any recursive solution must operate correctly in this case as well.

Figure 5-3 contains a recursive implementation of the predicate function **IsPalindrome(str)** that returns **true** if and only if the string **str** is a palindrome. The function first checks to see whether the length of the string is less than 2. If it is, the string is certainly a palindrome. In not, the function checks to make sure that the string meets both of the criteria listed earlier.

This implementation in Figure 5-3 is somewhat inefficient, even though the recursive decomposition is easy to follow. You can write a more efficient implementation of **IsPalindrome** by making the following changes:

- *Calculate the length of the argument string only once.* The initial implementation calculates the length of the string at every level of the recursive decomposition, even though the structure of the solution guarantees that the length of the string decreases by two on every recursive call. By calculating the length of the string at the beginning and passing it down through each of the recursive calls, you can eliminate many calls to the **length** method. To avoid changing the prototype for **IsPalindrome**, you need

Figure 5-3 Recursive implementation of IsPalindrome

```
/*
 * Function: IsPalindrome
 * Usage: if (IsPalindrome(str)) . . .
 * -----
 * This function returns true if and only if the string is a
 * palindrome. This implementation operates recursively by noting
 * that all strings of length 0 or 1 are palindromes (the simple
 * case) and that longer strings are palindromes only if their first
 * and last characters match and the remaining substring is a
 * palindrome.
 */

bool IsPalindrome(string str) {
    int len = str.length();
    if (len <= 1) {
        return true;
    } else {
        return (str[0] == str[len - 1]
                && IsPalindrome(str.substr(1, len - 2)));
    }
}
```

to define **IsPalindrome** as a wrapper function and have it pass the information to a second recursive function that does all the actual work.

- *Don't make a substring on each call.* Instead of calling **substr** to make copy of the interior of the string, you can pass the first and last position for the substring as parameters and allow those positions to define the subregion of the string being checked.

The revised implementation of **IsPalindrome** appears in Figure 5-4.

Binary search

When you work with arrays or vectors, one of the most common algorithmic operations consists of searching the array for a particular element. For example, if you were working with arrays of strings, it would be extremely useful to have a function

Figure 5-4 More efficient implementation of IsPalindrome

```

/*
 * Function: IsPalindrome
 * Usage: if (IsPalindrome(str)) . . .
 * -----
 * This function returns true if and only if the character string
 * str is a palindrome. This level of the implementation is
 * just a wrapper for the CheckPalindrome function, which
 * does the real work.
 */

bool IsPalindrome(string str) {
    return CheckPalindrome(str, 0, str.length() - 1);
}

/*
 * Function: CheckPalindrome
 * Usage: if (CheckPalindrome(str, firstPos, lastPos)) . . .
 * -----
 * This function returns true if the characters from firstPos
 * to lastPos in the string str form a palindrome. The
 * implementation uses the recursive insight that all
 * strings of length 0 or 1 are palindromes (the simple
 * case) and that longer strings are palindromes only if
 * their first and last characters match and the remaining
 * substring is a palindrome. Recursively examining the
 * interior substring is performed by adjusting the indexes
 * of the range to examine. The interior substring
 * begins at firstPos+1 and ends at lastPos-1.
 */

bool CheckPalindrome(string str, int firstPos, int lastPos) {
    if (firstPos >= lastPos) {
        return true;
    } else {
        return (str[firstPos] == str[lastPos]
                && CheckPalindrome(str, firstPos + 1, lastPos - 1));
    }
}

```

```
int FindStringInArray(string key, string array[], int n);
```

that searches through each of the **n** elements of **array**, looking for an element whose value is equal to **key**. If such an element is found, **FindStringInArray** returns the index at which it appears (if the key appears more than once in the array, the index of any matching is fine). If no matching element exists, the function returns **-1**.

If you have no specific knowledge about the order of elements within the array, the implementation of **FindStringInArray** must simply check each of the elements in turn until it either finds a match or runs out of elements. This strategy is called the **linear search algorithm**, which can be time-consuming if the arrays are large. On the other hand, if you know that the elements of the array are arranged in alphabetical order, you can adopt a much more efficient approach. All you have to do is divide the array in half and compare the key you're trying to find against the element closest to the middle of the array, using the order defined by the ASCII character codes, which is called **lexicographic order**. If the key you're looking for precedes the middle element, then the key—if it exists at all—must be in the first half. Conversely, if the key follows the middle element in lexicographic order, you only need to look at the elements in the second half. This strategy is called the **binary search algorithm**. Because binary search makes it possible for you to discard half the possible elements at each step in the process, it turns out to be much more efficient than linear search for sorted arrays.

The binary search algorithm is also a perfect example of the divide-and-conquer strategy. It is therefore not surprising that binary search has a natural recursive implementation, which is shown in Figure 5-5. Note that the function **FindStringInSortedArray** is implemented as a wrapper, leaving the real work to the recursive function **BinarySearch**, which takes two indices—**low** and **high**—that limit the range of the search.

The simple cases for **BinarySearch** are

1. *There are no elements in the active part of the array.* This condition is marked by the fact that the index **low** is greater than the index **high**, which means that there are no elements left to search.
2. *The middle element (or an element to one side of the middle if the array contains an even number of elements) matches the specified key.* Since the key has just been found, **FindStringInSortedArray** can simply return the index of the middle value.

If neither of these cases applies, however, the implementation can simplify the problem by choosing the appropriate half of the array and call itself recursively with an updated set of search limits.

Mutual recursion

In each of the examples considered so far, the recursive functions have called themselves directly, in the sense that the body of the function contains a call to itself. Although most of the recursive functions you encounter are likely to adhere to this style, the definition of *recursion* is actually somewhat broader. To be recursive, a function must call itself at some point during its evaluation. If a function is subdivided into subsidiary functions, the recursive call can actually occur at a deeper level of nesting. For example, if a function *f* calls a function *g*, which in turn calls *f*, the function calls are still considered to be recursive. Because the functions *f* and *g* call each other, this type of recursion is called **mutual recursion**.

Figure 5-5 Divide-and-conquer implementation of binary search

```

/*
 * Function: FindStringInSortedArray
 * Usage: index = FindStringInSortedArray(key, array, n);
 * -----
 * This function searches the array looking for the specified
 * key. The argument n specifies the effective size of the
 * array, which must be sorted according to lexicographic
 * order. If the key is found, the function returns the
 * index in the array at which that key appears. (If the key
 * appears more than once in the array, any of the matching
 * indices may be returned). If the key does not exist in
 * the array, the function returns -1. In this implementation,
 * FindStringInSortedArray is simply a wrapper; all the work
 * is done by the recursive function BinarySearch.
 */

int FindStringInSortedArray(string key, string array[], int n) {
    return BinarySearch(key, array, 0, n - 1);
}

/*
 * Function: BinarySearch
 * Usage: index = BinarySearch(key, array, low, high);
 * -----
 * This function does the work for FindStringInSortedArray.
 * The only difference is that BinarySearch takes both the
 * upper and lower limit of the search.
 */

int BinarySearch(string key, string array[], int low, int high) {
    if (low > high) return -1;
    int mid = (low + high) / 2;
    if (key == array[mid]) return mid;
    if (key < array[mid]) {
        return BinarySearch(key, array, low, mid - 1);
    } else {
        return BinarySearch(key, array, mid + 1, high);
    }
}

```

As a simple example, let's investigate how to use recursion to test whether a number is even or odd. If you limit the domain of possible values to the set of **natural numbers**, which are defined simply as the set of nonnegative integers, the even and odd numbers can be characterized as follows:

- A number is *even* if its predecessor is odd.
- A number is *odd* if it is not even.
- The number 0 is even by definition.

Even though these rules seem simplistic, they constitute the basis of an effective, if inefficient, strategy for distinguishing odd and even numbers. A mutually recursive implementation of the predicate functions **IsEven** and **IsOdd** appears in Figure 5-6.

Figure 5-6 Mutually recursive definitions of IsEven and IsOdd

```
/*
 * Function: IsEven
 * Usage: if (IsEven(n)) . . .
 * -----
 * This function returns true if n is even. The number 0
 * is considered even by definition; any other number is
 * even if its predecessor is odd. Note that this function
 * is defined to take an unsigned argument and is therefore
 * not applicable to negative integers.
 */

bool IsEven(unsigned int n) {
    if (n == 0) {
        return true;
    } else {
        return IsOdd(n - 1);
    }
}

/*
 * Function: IsOdd
 * Usage: if (IsOdd(n)) . . .
 * -----
 * This function returns true if n is odd, where a number
 * is defined to be odd if it is not even. Note that this
 * function is defined to take an unsigned argument and is
 * therefore not applicable to negative integers.
 */

bool IsOdd(unsigned int n) {
    return !IsEven(n);
}
```

5.5 Thinking recursively

For most people, recursion is not an easy concept to grasp. Learning to use it effectively requires considerable practice and forces you to approach problems in entirely new ways. The key to success lies in developing the right mindset—learning how to think recursively. The remainder of this chapter is designed to help you achieve that goal.

Maintaining a holistic perspective

In Chapter 2 of *The Art and Science of C*, I devote one section to the philosophical concepts of holism and reductionism. Simply stated, **reductionism** is the belief that the whole of an object can be understood merely by understanding the parts that make it up. Its antithesis is **holism**, the position that the whole is often greater than the sum of its parts. As you learn about programming, it helps to be able to interleave these two perspectives, sometimes focusing on the behavior of a program as a whole, and at other times delving into the details of its execution. When you try to learn about recursion, however, this balance seems to change. Thinking recursively requires you to think holistically. In the recursive domain, reductionism is the enemy of understanding and invariably gets in the way.

To maintain the holistic perspective, you must become comfortable adopting the recursive leap of faith, which was introduced in its own section earlier in this chapter.

Whenever you are writing a recursive program or trying to understand the behavior of one, you must get to the point where you ignore the details of the individual recursive calls. As long as you have chosen the right decomposition, identified the appropriate simple cases, and implemented your strategy correctly, those recursive calls will simply work. You don't need to think about them.

Unfortunately, until you have had extensive experience working with recursive functions, applying the recursive leap of faith does not come easily. The problem is that it requires to suspend your disbelief and make assumptions about the correctness of your programs that fly in the face of your experience. After all, when you write a program, the odds are good—even if you are an experienced programmer—that your program won't work the first time. In fact, it is quite likely that you have chosen the wrong decomposition, messed up the definition of the simple cases, or somehow messed things up trying to implement your strategy. If you have done any of these things, your recursive calls won't work.

When things go wrong—as they inevitably will—you have to remember to look for the error in the right place. The problem lies somewhere in your recursive implementation, not in the recursive mechanism itself. If there is a problem, you should be able to find it by looking at a single level of the recursive hierarchy. Looking down through additional levels of recursive calls is not going to help. If the simple cases work and the recursive decomposition is correct, the subsidiary calls will work correctly. If they don't, there is something you need to fix in the definition of the recursive function itself.

Avoiding the common pitfalls

As you gain experience with recursion, the process of writing and debugging recursive programs will become more natural. At the beginning, however, finding out what you need to fix in a recursive program can be difficult. The following is a checklist that will help you identify the most common sources of error.

- *Does your recursive implementation begin by checking for simple cases?* Before you attempt to solve a problem by transforming it into a recursive subproblem, you must first check to see if the problem is so simple that such decomposition is unnecessary. In almost all cases, recursive functions begin with the keyword **if**. If your function doesn't, you should look carefully at your program and make sure that you know what you're doing.¹
- *Have you solved the simple cases correctly?* A surprising number of bugs in recursive programs arise from having incorrect solutions to the simple cases. If the simple cases are wrong, the recursive solutions to more complicated problems will inherit the same mistake. For example, if you had mistakenly defined **Fact(0)** as 0 instead of 1, calling **Fact** on any argument would end up returning 0.
- *Does your recursive decomposition make the problem simpler?* For recursion to work, the problems have to get simpler as you go along. More formally, there must be some **metric**—a standard of measurement that assigns a numeric difficulty rating to the problem—that gets smaller as the computation proceeds. For mathematical functions like **Fact** and **Fib**, the value of the integer argument serves as a metric. On each recursive call, the value of the argument gets smaller. For the **IsPalindrome** function, the appropriate metric is the length of the argument string, because the string gets shorter on each recursive call. If the problem instances do not get simpler, the

¹ At times, as in the case of the **IsPalindrome** implementation, it may be necessary to perform some calculations prior to making the simple-case test. The point is that the simple-case test must precede any recursive decomposition.

decomposition process will just keep making more and more calls, giving rise to the recursive analogue of the infinite loop, which is called **nonterminating recursion**.

- *Does the simplification process eventually reach the simple cases, or have you left out some of the possibilities?* A common source of error is failing to include simple case tests for all the cases that can arise as the result of the recursive decomposition. For example, in the **IsPalindrome** implementation presented in Figure 5-3, it is critically important for the function to check the zero-character case as well as the one-character case, even if the client never intends to call **IsPalindrome** on the empty string. As the recursive decomposition proceeds, the string arguments get shorter by two characters at each level of the recursive call. If the original argument string is even in length, the recursive decomposition will never get to the one-character case.
- *Do the recursive calls in your function represent subproblems that are truly identical in form to the original?* When you use recursion to break down a problem, it is essential that the subproblems be of the same form. If the recursive calls change the nature of the problem or violate one of the initial assumptions, the entire process can break down. As several of the examples in this chapter illustrate, it is often useful to define the publicly exported function as a simple wrapper that calls a more general recursive function which is private to the implementation. Because the private function has a more general form, it is usually easier to decompose the original problem and still have it fit within the recursive structure.
- *When you apply the recursive leap of faith, do the solutions to the recursive subproblems provide a complete solution to the original problem?* Breaking a problem down into recursive subinstances is only part of the recursive process. Once you get the solutions, you must also be able to reassemble them to generate the complete solution. The way to check whether this process in fact generates the solution is to walk through the decomposition, religiously applying the recursive leap of faith. Work through all the steps in the current function call, but assume that every recursive call generates the correct answer. If following this process yields the right solution, your program should work.

Summary

This chapter has introduced the idea of *recursion*, a powerful programming strategy in which complex problems are broken down into simpler problems of the same form. The important points presented in this chapter include:

- Recursion is similar to stepwise refinement in that both strategies consist of breaking a problem down into simpler problems that are easier to solve. The distinguishing characteristic of recursion is that the simpler subproblems must have the same form as the original.
- In C++, recursive functions typically have the following paradigmatic form:

```

if (test for simple case) {
    Compute a simple solution without using recursion.
} else {
    Break the problem down into subproblems of the same form.
    Solve each of the subproblems by calling this function recursively.
    Reassemble the solutions to the subproblems into a solution for the whole.
}

```

- To use recursion, you must be able to identify *simple cases* for which the answer is easily determined and a *recursive decomposition* that allows you to break any complex instance of the problem into simpler problems of the same type.

- Recursive functions are implemented using exactly the same mechanism as any other function call. Each call creates a new stack frame that contains the local variables for that call. Because the computer creates a separate stack frame for each function call, the local variables at each level of the recursive decomposition remain separate.
- Before you can use recursion effectively, you must learn to limit your analysis to a single level of the recursive decomposition and to rely on the correctness of all simpler recursive calls without tracing through the entire computation. Trusting these simpler calls to work correctly is called the *recursive leap of faith*.
- Mathematical functions often express their recursive nature in the form of a *recurrence relation*, in which each element of a sequence is defined in terms of earlier elements.
- Although some recursive functions may be less efficient than their iterative counterparts, recursion itself is not the problem. As is typical with all types of algorithms, some recursive strategies are more efficient than others.
- In order to ensure that a recursive decomposition produces subproblems that are identical in form to the original, it is often necessary to generalize the problem. As a result, it is often useful to implement the solution to a specific problem as a simple *wrapper* function whose only purpose is to call a subsidiary function that handles the more general case.
- Recursion need not consist of a single function that calls itself but may instead involve several functions that call each other in a cyclical pattern. Recursion that involves more than one function is called *mutual recursion*.
- You will be more successful at understanding recursive programs if you can maintain a holistic perspective rather than a reductionistic one.

Thinking about recursive problems in the right way does not come easily. Learning to use recursion effectively requires practice and more practice. For many students, mastering the concept takes years. But because recursion will turn out to be one of the most powerful techniques in your programming repertoire, that time will be well spent.

Review questions

1. Define the terms *recursive* and *iterative*. Is it possible for a function to employ both strategies?
2. What is the fundamental difference between recursion and stepwise refinement?
3. In the pseudocode for the **CollectContributions** function, the **if** statement looks like this:

```
if (n <= 100)
```

Why is it important to use the `<=` operator instead of simply checking whether `n` is exactly equal to 100?

4. What is the standard recursive paradigm?
5. What two properties must a problem have for recursion to make sense as a solution strategy?
6. Why is the term *divide and conquer* appropriate to recursive techniques?

7. What is meant by the *recursive leap of faith*? Why is this concept important to you as a programmer?
8. In the section entitled “Tracing the recursive process,” the text goes through a long analysis of what happens internally when **Fact(4)** is called. Using this section as a model, trace through the execution of **Fib(4)**, sketching out each stack frame created in the process.
9. Modify Fibonacci’s rabbit problem by introducing the additional rule that rabbit pairs stop reproducing after giving birth to three litters. How does this assumption change the recurrence relation? What changes do you need to make in the simple cases?
10. How many times is **Fib(1)** called when calculating **Fib(n)** using the recursive implementation given in Figure 5-1?
11. What would happen if you eliminated the **if (n == 1)** check from the function **AdditiveSequence**, so that the implementation looked like this:

```
int AdditiveSequence(int n, int t0, int t1) {
    if (n == 0) return t0;
    return AdditiveSequence(n - 1, t1, t0 + t1);
}
```

Would the function still work? Why or why not?

12. What is a wrapper function? Why are they often useful in writing recursive functions?
13. Why is it important that the implementation of **IsPalindrome** in Figure 5-3 check for the empty string as well as the single character string? What would happen if the function didn’t check for the single character case and instead checked only whether the length is 0? Would the function still work correctly?
14. Explain the effect of the function call

```
CheckPalindrome(str, firstPos + 1, lastPost - 1)
```

in the **IsPalindrome** implementation given in Figure 5-4.

15. What is mutual recursion?
16. What would happen if you defined **IsEven** and **IsOdd** as follows:

```
bool IsEven(unsigned int n) {
    return !IsOdd(n);
}

bool IsOdd(unsigned int n) {
    return !IsEven(n);
}
```

Which of the errors explained in the section “Avoiding the common pitfalls” is illustrated in this example?

17. The following definitions of **IsEven** and **IsOdd** are also incorrect:

```
bool IsEven(unsigned int n) {
    if (n == 0) {
        return true;
    } else {
        return IsOdd(n - 1);
    }
}

bool IsOdd(unsigned int n) {
    if (n == 1) {
        return true;
    } else {
        return IsEven(n - 1);
    }
}
```

Give an example that shows how this implementation can fail. What common pitfall is illustrated here?

Programming exercises

1. Spherical objects, such as cannonballs, can be stacked to form a pyramid with one cannonball at the top, sitting on top of a square composed of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth. Write a recursive function **Cannonball** that takes as its argument the height of the pyramid and returns the number of cannonballs it contains. Your function must operate recursively and must not use any iterative constructs, such as **while** or **for**.
2. Unlike many programming languages, C++ does not include a predefined operator that raises a number to a power. As a partial remedy for this deficiency, write a recursive implementation of a function

```
int RaiseIntToPower(int n, int k)
```

that calculates n^k . The recursive insight that you need to solve this problem is the mathematical property that

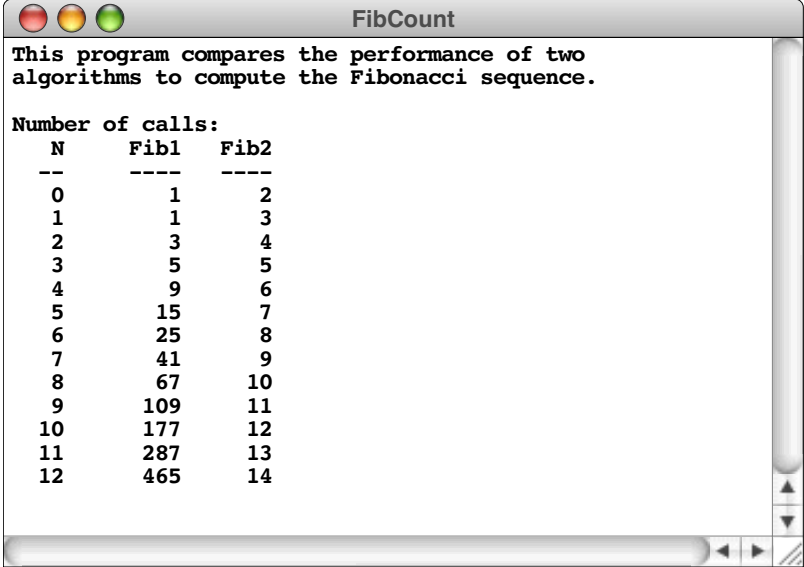
$$n^k = \begin{cases} 1 & \text{if } k = 0 \\ n \times n^{k-1} & \text{otherwise} \end{cases}$$

3. The **greatest common divisor** (g.c.d.) of two nonnegative integers is the largest integer that divides evenly into both. In the third century B.C., the Greek mathematician Euclid discovered that the greatest common divisor of x and y can always be computed as follows:
 - If x is evenly divisible by y , then y is the greatest common divisor.
 - Otherwise, the greatest common divisor of x and y is always equal to the greatest common divisor of y and the remainder of x divided by y .

Use Euclid's insight to write a recursive function **GCD(x, y)** that computes the greatest common divisor of x and y .

4. Write an iterative implementation of the function **Fib(n)**.

5. For each of the two recursive implementations of the function **Fib(n)** presented in this chapter, write a recursive function (you can call these **CountFib1** and **CountFib2** for the two algorithms) that counts the number of function calls made during the evaluation of the corresponding Fibonacci calculation. Write a main program that uses these functions to display a table showing the number of calls made by each algorithm for various values of **n**, as shown in the following sample run:



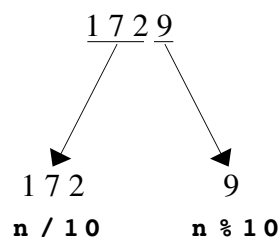
This program compares the performance of two algorithms to compute the Fibonacci sequence.

Number of calls:

N	Fib1	Fib2
0	1	2
1	1	3
2	3	4
3	5	5
4	9	6
5	15	7
6	25	8
7	41	9
8	67	10
9	109	11
10	177	12
11	287	13
12	465	14

6. Write a recursive function **DigitSum(n)** that takes a nonnegative integer and returns the sum of its digits. For example, calling **DigitSum(1729)** should return $1 + 7 + 2 + 9$, which is 19.

The recursive implementation of **DigitSum** depends on the fact that it is very easy to break an integer down into two components using division by 10. For example, given the integer 1729, you can divide it into two pieces as follows:



Each of the resulting integers is strictly smaller than the original and thus represents a simpler case.

7. The **digital root** of an integer n is defined as the result of summing the digits repeatedly until only a single digit remains. For example, the digital root of 1729 can be calculated using the following steps:

Step 1:	$1 + 7 + 2 + 9$	$\rightarrow$	19
Step 2:	$1 + 9$	$\rightarrow$	10
Step 3:	$1 + 0$	$\rightarrow$	1

Because the total at the end of step 3 is the single digit 1, that value is the digital root.

Write a function **DigitalRoot(n)** that returns the digital root of its argument. Although it is easy to implement **DigitalRoot** using the **DigitSum** function from exercise 6 and a **while** loop, part of the challenge of this problem is to write the function recursively without using any explicit loop constructs.

8. The mathematical combinations function $C(n, k)$ is usually defined in terms of factorials, as follows:

$$C(n, k) = \frac{n!}{k! \times (n-k)!}$$

The values of $C(n, k)$ can also be arranged geometrically to form a triangle in which n increases as you move down the triangle and k increases as you move from left to right. The resulting structure, which is called *Pascal's Triangle* after the French mathematician Blaise Pascal, is arranged like this:

$$\begin{array}{ccccccc} & & & & C(0,0) & & \\ & & & & C(1,0) & C(1,1) & \\ & & & C(2,0) & C(2,1) & C(2,2) & \\ & & C(3,0) & C(3,1) & C(3,2) & C(3,3) & \\ C(4,0) & C(4,1) & C(4,2) & C(4,3) & C(4,4) & & \end{array}$$

Pascal's Triangle has the interesting property that every entry is the sum of the two entries above it, except along the left and right edges, where the values are always 1. Consider, for example, the circled entry in the following display of Pascal's Triangle:

$$\begin{array}{cccccccc} & & & & & & 1 & \\ & & & & & 1 & & 1 \\ & & & 1 & & 2 & & 1 \\ & & 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ 1 & & 1 & & 5 & & 10 & & 10 & & 5 & & 1 \\ & 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1 \end{array}$$

This entry, which corresponds to $C(6, 2)$, is the sum of the two entries—5 and 10—that appear above it to either side. Use this relationship between entries in Pascal's Triangle to write a recursive implementation of the **Combinations** function that uses no loops, no multiplication, and no calls to **Fact**.

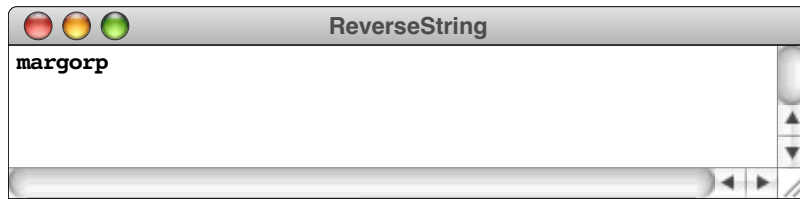
9. Write a recursive function that takes a string as argument and returns the reverse of that string. The prototype for this function should be

```
string Reverse(string str);
```

and the statement

```
cout << Reverse("program") << endl;
```

should display



Your solution should be entirely recursive and should not use any iterative constructs such as **while** or **for**.

10. The **strutils.h** library contains a function **IntegerToString**. You might have wondered how the computer actually goes about the process of converting an integer into its string representation. As it turns out, the easiest way to implement this function is to use the recursive decomposition of an integer outlined in exercise 6. Rewrite the **IntegerToString** implementation so that it operates recursively without using use any of the iterative constructs such as **while** and **for**.

Chapter 6

Recursive Procedures

*Nor would I consider the magnitude and complexity of my
plan as any argument of its impracticability.*

— Mary Shelley, *Frankenstein*, 1818

When a recursive decomposition follows directly from a mathematical definition, as it does in the case of the **Fact** and **Fib** functions in Chapter 5, applying recursion is not particularly hard. In most cases, you can translate the mathematical definition directly into a recursive implementation by plugging the appropriate expressions into the standard recursive paradigm. The situation changes, however, as you begin to solve more complex problems.

This chapter introduces several programming problems that seem—at least on the surface—much more difficult than those in Chapter 5. In fact, if you try to solve these problems without using recursion, relying instead on more familiar iterative techniques, you will find them quite difficult to solve. Even so, each of the problems has a recursive solution that is surprisingly short. If you exploit the power of recursion, a few lines of code are sufficient for each task.

The brevity of these solutions endows them with a deceptive aura of simplicity. The hard part of solving these problems does not lie in the intricacy or length of the code. What makes these programs difficult is identifying the appropriate recursive decomposition in the first place. Doing so occasionally requires some cleverness, but what you need even more is confidence. You have to accept the recursive leap of faith. As you develop your solution, you must strive to come to a point at which you are faced with a problem that is identical to the original in form but simpler in scale. When you do, you have to be willing to stop and declare the problem solved, without trying to trace the program further.

6.1 The Tower of Hanoi

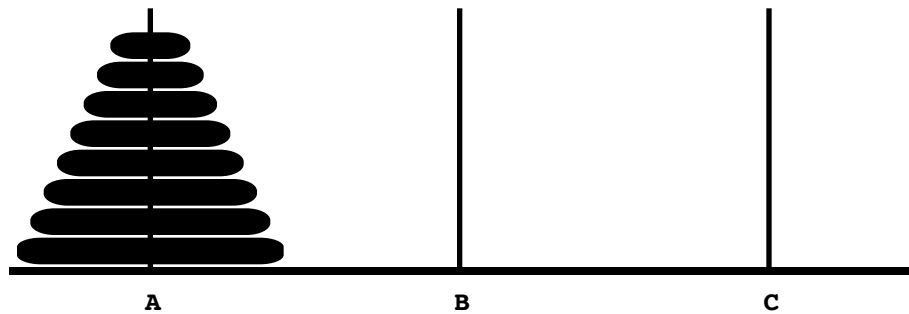
The first example in this chapter is a simple puzzle that has come to be known as the **Tower of Hanoi**. Invented by French mathematician Edouard Lucas in the 1880s, the Tower of Hanoi puzzle quickly became popular in Europe. Its success was due in part to the legend that grew up around the puzzle, which was described as follows in *La Nature* by the French mathematician Henri De Parville (as translated by the mathematical historian W. W. R. Ball):

In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When all the sixty-four disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

Over the years, the setting has shifted from India to Vietnam, but the puzzle and its legend remain the same.

As far as I know, the Tower of Hanoi puzzle has no practical use except one: teaching recursion to computer science students. In that domain, its value is unquestioned because the solution involves almost nothing besides recursion. In contrast to most practical examples of recursion, the Tower of Hanoi problem has no extraneous complications that might interfere with your understanding and keep you from seeing how the recursive solution works. Because it works so well as an example, the Tower of Hanoi is included in most textbooks that treat recursion and has become part of the cultural heritage that computer programmers share.

In most commercial versions of the puzzle, the 64 golden disks of legend are replaced with eight wooden or plastic ones, which makes the puzzle considerably easier to solve, not to mention cheaper. The initial state of the puzzle looks like this:



At the beginning, all eight disks are on spire A. Your goal is to move these eight disks from spire A to spire B, but you must adhere to the following rules:

- You can only move one disk at a time.
- You are not allowed to move a larger disk on top of a smaller disk.

Framing the problem

In order to apply recursion to the Tower of Hanoi problem, you must first frame the problem in more general terms. Although the ultimate goal is moving eight disks from A to B, the recursive decomposition of the problem will involve moving smaller subtowers from spire to spire in various configurations. In the more general case, the problem you need to solve is moving a tower of a given height from one spire to another, using the third spire as a temporary repository. To ensure that all subproblems fit the original form, your recursive procedure must therefore take the following arguments :

1. The number of disks to move
2. The name of the spire where the disks start out
3. The name of the spire where the disks should finish
4. The name of the spire used for temporary storage

The number of disks to move is clearly an integer, and the fact that the spires are labeled with the letters *A*, *B*, and *C* suggests the use of type **char** to indicate which spire is involved. Knowing the types allows you to write a prototype for the operation that moves a tower, as follows:

```
void MoveTower(int n, char start, char finish, char temp);
```

To move the eight disks in the example, the initial call is

```
MoveTower(8, 'A', 'B', 'C');
```

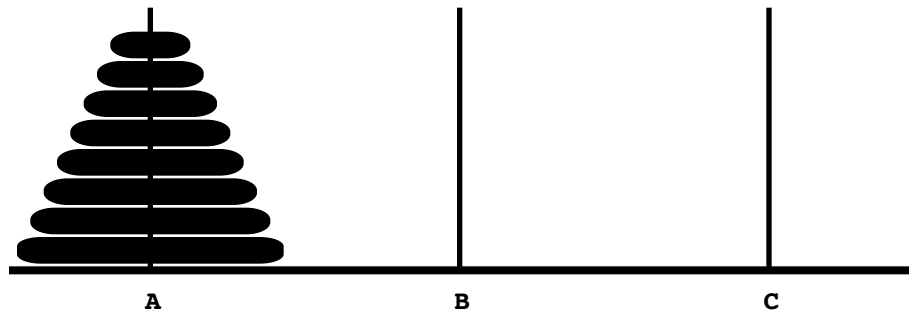
This function call corresponds to the English command “Move a tower of size 8 from spire A to spire B using spire C as a temporary.” As the recursive decomposition proceeds, **MoveTower** will be called with different arguments that move smaller towers in various configurations.

Finding a recursive strategy

Now that you have a more general definition of the problem, you can return to the problem of finding a strategy for moving a large tower. To apply recursion, you must first make sure that the problem meets the following conditions:

1. *There must be a simple case.* In this problem, there is an obvious simple case. Because the rules of the puzzle require you to move only one disk at a time, any tower with more than one disk must be moved in pieces. If, however, the tower only contains one disk, you can go ahead and move it, as long as you obey the other rules of the game. Thus, the simple case occurs when n is equal to 1.
2. *There must be a way to break the problem down into a simpler problem in such a way that solving the smaller problem contributes to solving the original one.* This part of the problem is harder and will require closer examination.

To see how solving a simpler subproblem helps solve a larger problem, it helps to go back and consider the original example with eight disks.

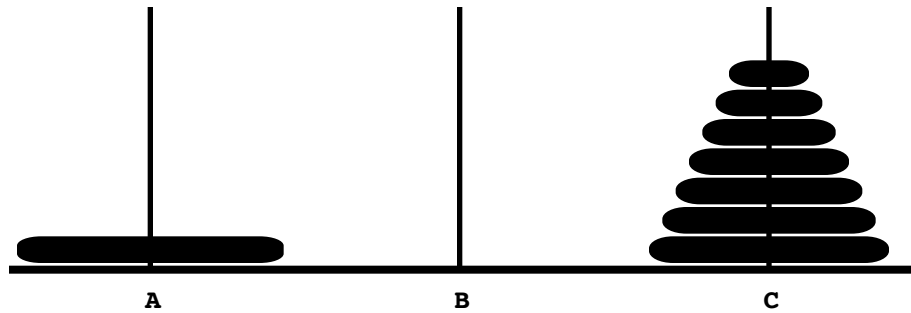


The goal here is to move eight disks from spire A to spire B. You need to ask yourself how it would help if you could solve the same problem for a smaller number of disks. In particular, you should think about how being able to move a stack of seven disks would help you to solve the eight-disk case.

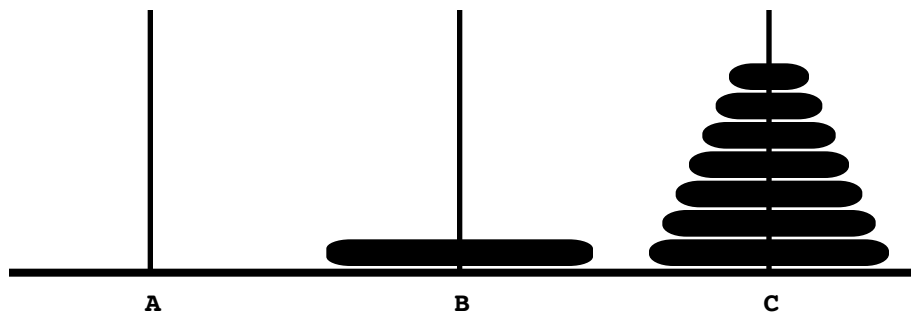
If you think about the problem for a few moments, it becomes clear that you can solve the problem by dividing it into these three steps:

1. Move the entire stack consisting of the top seven disks from spire A to spire C.
2. Move the bottom disk from spire A to spire B.
3. Move the stack of seven disks from spire C to spire B.

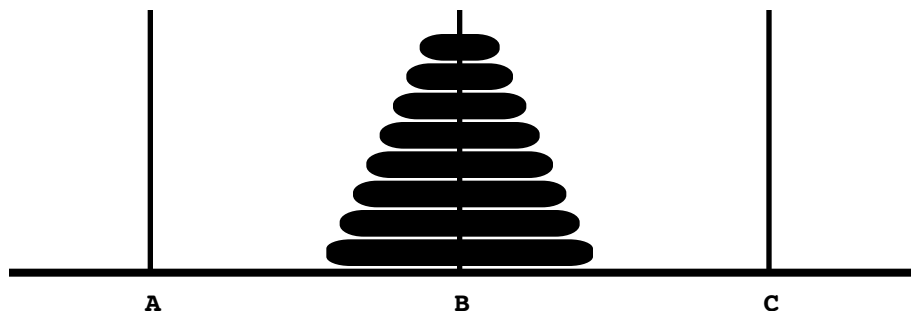
Executing the first step takes you to the following position:



Once you have gotten rid of the seven disks on top of the largest disk, the second step is simply to move that disk from spire A to spire B, which results in the following configuration:



All that remains is to move the tower of seven disks back from spire C to spire B, which is again a smaller problem of the same form. This operation is the third step in the recursive strategy, and leaves the puzzle in the desired final configuration:



That's it! You're finished. You've reduced the problem of moving a tower of size eight to one of moving a tower of size seven. More importantly, this recursive strategy generalizes to towers of size N , as follows:

1. Move the top $N-1$ disks from the start spire to the temporary spire.
2. Move a single disk from the start spire to the finish spire.
3. Move the stack of $N-1$ disks from the temporary spire back to the finish spire.

At this point, it is hard to avoid saying to yourself, “Okay, I can reduce the problem to moving a tower of size $N-1$, but how do I accomplish that?” The answer, of course, is that you move a tower of size $N-1$ in precisely the same way. You break that problem down into one that requires moving a tower of size $N-2$, which further breaks down into

the problem of moving a tower of size $N-3$, and so forth, until there is only a single disk to move. Psychologically, however, the important thing is to avoid asking that question altogether. The recursive leap of faith should be sufficient. You've reduced the scale of the problem without changing its form. That's the hard work. All the rest is bookkeeping, and it's best to let the computer take care of that.

Once you have identified the simple cases and the recursive decomposition, all you need to do is plug them into the standard recursive paradigm, which results in the following pseudocode procedure:

```
void MoveTower(int n, char start, char finish, char temp) {
    if (n == 1) {
        Move a single disk from start to finish.
    } else {
        Move a tower of size n - 1 from start to temp.
        Move a single disk from start to finish.
        Move a tower of size n - 1 from temp to finish.
    }
}
```

Validating the strategy

Although the pseudocode strategy is in fact correct, the derivation up to this point has been a little careless. Whenever you use recursion to decompose a problem, you must make sure that the new problems are identical in form to the original. The task of moving $N-1$ disks from one spire to another certainly sounds like an instance of the same problem and fits the **MoveTower** prototype. Even so, there is a subtle but important difference. In the original problem, the destination and temporary spires are empty. When you move a tower of size $N-1$ to the temporary spire as part of the recursive strategy, you've left a disk behind on the starting spire. Does the presence of that disk change the nature of the problem and thus invalidate the recursive solution?

To answer this question, you need to think about the subproblem in light of the rules of the game. If the recursive decomposition doesn't end up violating the rules, everything should be okay. The first rule—that only one disk can be moved at a time—is not an issue. If there is more than a single disk, the recursive decomposition breaks the problem down to generate a simpler case. The steps in the pseudocode that actually transfer disks move only one disk at a time. The second rule—that you are not allowed to place a larger disk on top of a smaller one—is the critical one. You need to convince yourself that you will not violate this rule in the recursive decomposition.

The important observation to make is that, as you move a subtower from one spire to the other, the disk you leave behind on the original spire—and indeed any disk left behind at any previous stage in the operation—must be larger than anything in the current subtower. Thus, as you move those disks among the spires, the only disks below them will be larger in size, which is consistent with the rules.

Coding the solution

To complete the Tower of Hanoi solution, the only remaining step is to substitute function calls for the remaining pseudocode. The task of moving a complete tower requires a recursive call to the **MoveTower** function. The only other operation is moving a single disk from one spire to another. For the purposes of writing a test program that displays the steps in the solution, all you need is a function that records its operation on the console. For example, you can implement the function **MoveSingleDisk** as follows:

```
void MoveSingleDisk(char start, char finish) {
    cout << start << " -> " << finish << endl;
}
```

The **MoveTower** code itself looks like this:

```
void MoveTower(int n, char start, char finish, char temp) {
    if (n == 1) {
        MoveSingleDisk(start, finish);
    } else {
        MoveTower(n - 1, start, temp, finish);
        MoveSingleDisk(start, finish);
        MoveTower(n - 1, temp, finish, start);
    }
}
```

Tracing the recursive process

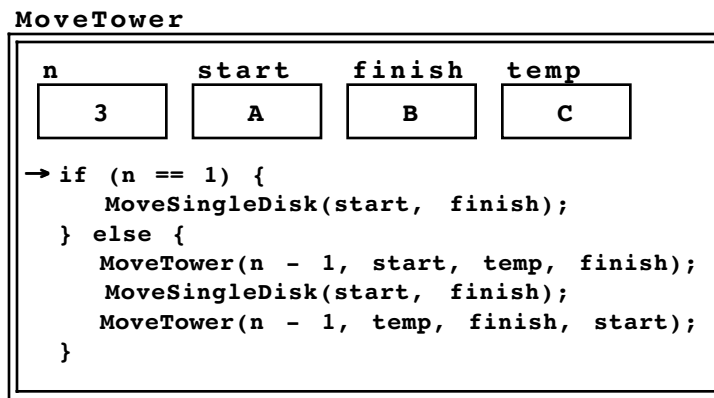
The only problem with this implementation of **MoveTower** is that it seems like magic. If you're like most students learning about recursion for the first time, the solution seems so short that there must be something missing. Where is the strategy? How can the computer know which disk to move first and where it should go?

The answer is that the recursive process—breaking a problem down into smaller subproblems of the same form and then providing solutions for the simple cases—is all you need to solve the problem. If you trust the recursive leap of faith, you're done. You can skip this section of the book and go on to the next. If you're still suspicious, it may be necessary for you to go through the steps in the complete process and watch what happens.

To make the problem more manageable, consider what happens if there are only three disks in the original tower. The main program call is therefore

```
MoveTower(3, 'A', 'B', 'C');
```

To trace how this call computes the steps necessary to transfer a tower of size 3, all you need to do is keep track of the operation of the program, using precisely the same strategy as in the factorial example from Chapter 5. For each new function call, you introduce a stack frame that shows the values of the parameters for that call. The initial call to **MoveTower**, for example, creates the following stack frame:



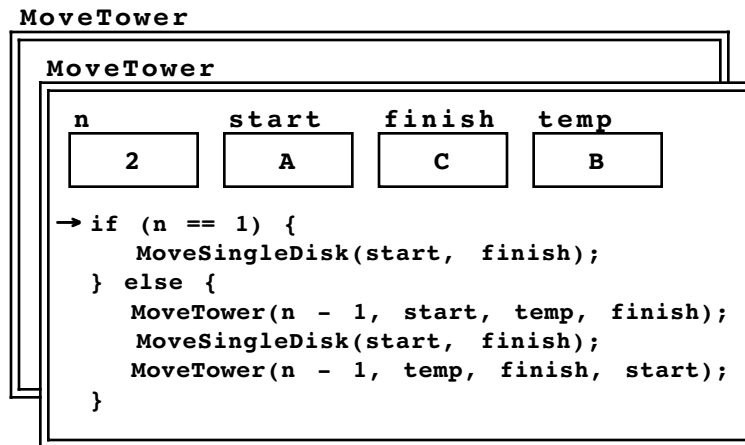
As the arrow in the code indicates, the function has just been called, so execution begins with the first statement in the function body. The current value of **n** is not equal to 1, so the program skips ahead to the **else** clause and executes the statement

```
MoveTower(n-1, start, temp, finish);
```

As with any function call, the first step is to evaluate the arguments. To do so, you need to determine the values of the variables **n**, **start**, **temp**, and **finish**. Whenever you need to find the value of a variable, you use the value as it is defined in the current stack frame. Thus, the **MoveTower** call is equivalent to

```
MoveTower(2, 'A', 'C', 'B');
```

This operation, however, indicates another function call, which means that the current operation is suspended until the new function call is complete. To trace the operation of the new function call, you need to generate a new stack frame and repeat the process. As always, the parameters in the new stack frame are initialized by copying the calling arguments in the order in which they appear. Thus, the new stack frame looks like this:



As the diagram illustrates, the new stack frame has its own set of variables, which temporarily supersede the variables in frames that are further down on the stack. Thus, as long as the program is executing in this stack frame, **n** will have the value 2, **start** will be 'A', **finish** will be 'C', and **temp** will be 'B'. The old values in the previous frame will not reappear until the subtask represented by this call is created and the function returns.

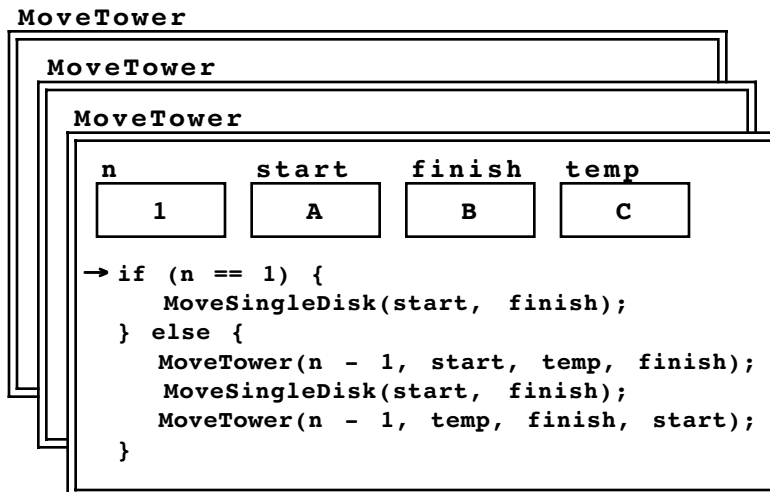
The evaluation of the recursive call to **MoveTower** proceeds exactly like the original one. Once again, **n** is not 1, which requires another call of the form

```
MoveTower(n-1, start, temp, finish);
```

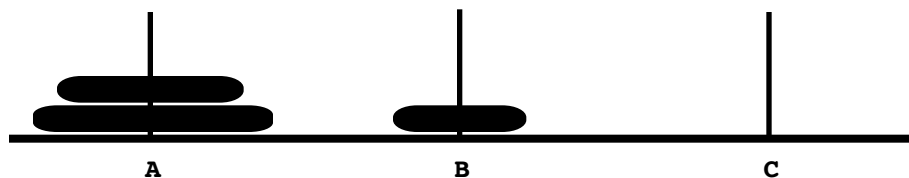
Because this call comes from a different stack frame, however, the value of the individual variables are different from those in the original call. If you evaluate the arguments in the context of the current stack frame, you discover that this function call is equivalent to

```
MoveTower(1, 'A', 'B', 'C');
```

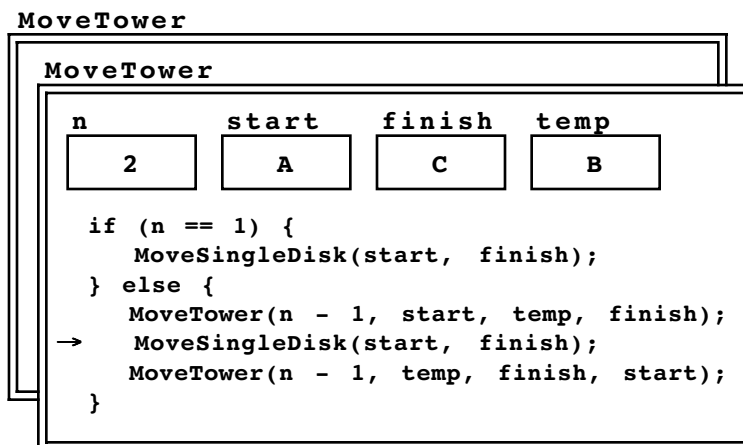
The effect of making this call is to introduce yet another stack frame for the **MoveTower** function, as follows:



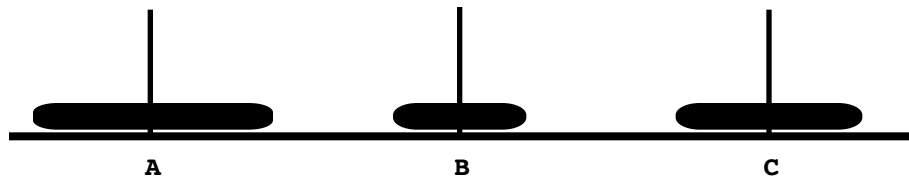
This call to **MoveTower**, however, does represent the simple case. Since n is 1, the program calls the **MoveSingleDisk** function to move a disk from A to B, leaving the puzzle in the following configuration:



At this point, the most recent call to **MoveTower** is complete and the function returns. In the process, its stack frame is discarded, which brings the execution back to the previous stack frame, having just completed the first statement in the **else** clause:



The call to **MoveSingleDisk** again represents a simple operation, which leaves the puzzle in the following state:



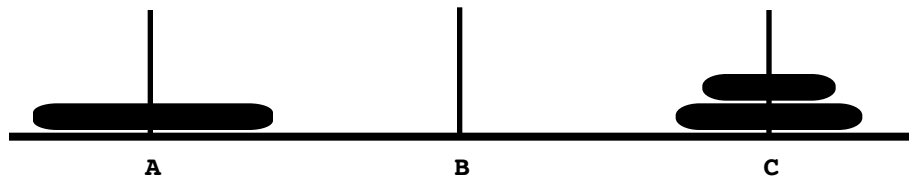
With the **MoveSingleDisk** operation completed, the only remaining step required to finish the current call to **MoveTower** is the last statement in the function:

```
MoveTower(n-1, temp, finish, start);
```

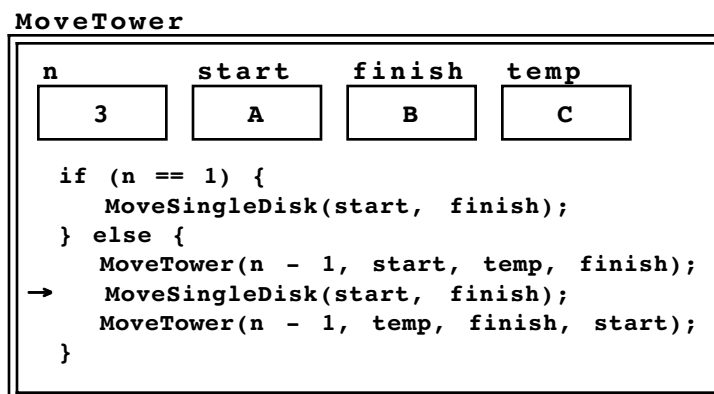
Evaluating these arguments in the context of the current frame reveals that this call is equivalent to

```
MoveTower(1, 'B', 'C', 'A');
```

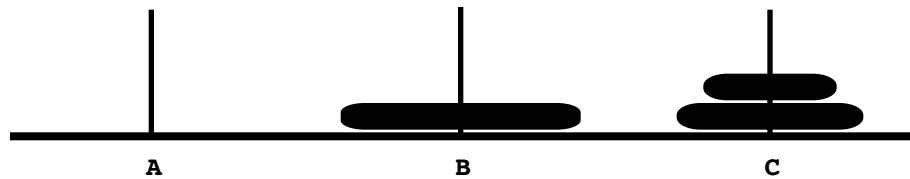
Once again, this call requires the creation of a new stack frame. By this point in the process, however, you should be able to see that the effect of this call is simply to move a tower of size 1 from B to C, using A as a temporary repository. Internally, the function determines that **n** is 1 and then calls **MoveSingleDisk** to reach the following configuration:



This operation again completes a call to **MoveTower**, allowing it to return to its caller having completed the subtask of moving a tower of size 2 from A to C. Discarding the stack frame from the just-completed subtask reveals the stack frame for the original call to **MoveTower**, which is now in the following state:



The next step is to call **MoveSingleDisk** to move the largest disk from A to B, which results in the following position:



The only operation that remains is to call

```
MoveTower(n-1, temp, finish, start);
```

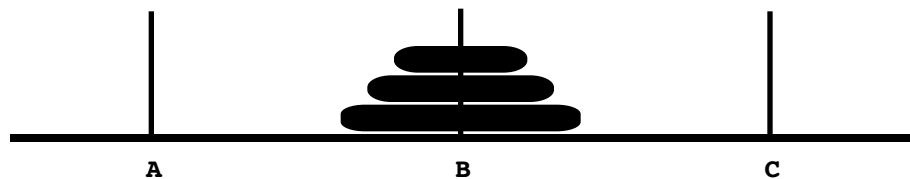
with the arguments from the current stack frame, which are

```
MoveTower(2, 'C', 'B', 'A');
```

If you're still suspicious of the recursive process, you can draw the stack frame created by this function call and continue tracing the process to its ultimate conclusion. At some point, however, it is essential that you trust the recursive process enough to see that function call as a single operation having the effect of the following command in English:

Move a tower of size 2 from C to B, using A as a temporary repository.

If you think about the process in this holistic form, you can immediately see that completion of this step will move the tower of two disks back from C to B, leaving the desired final configuration:



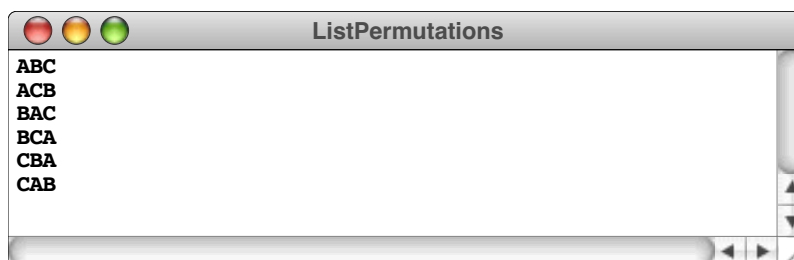
6.2 Generating permutations

Many word games such as Scrabble® require the ability to rearrange a set of letters to form a word. Thus, if you wanted to write a Scrabble program, it would be useful to have a facility for generating all possible arrangements of a particular set of tiles. In word games, such arrangements are generally called **anagrams**. In mathematics, they are known as **permutations**.

Let's suppose you want to write a function **ListPermutations(s)** that displays all permutations of the string **s**. For example, if you call

```
ListPermutations("ABC");
```

your program should display the six arrangements of **"ABC"**, as follows:



The order of the output is unimportant, but each of the possible arrangements should appear exactly once.

How would you go about implementing the **ListPermutations** function? If you are limited to iterative control structures, finding a general solution that works for strings of any length is difficult. Thinking about the problem recursively, on the other hand, leads to a relatively straightforward solution.

As is usually the case with recursive programs, the hard part of the solution process is figuring out how to divide the original problem into simpler instances of the same problem. In this case, to generate all permutations of a string, you need to discover how being able to generate all permutations of a shorter string might contribute to the solution.

Stop and think about this problem for a few minutes. When you are first learning about recursion, it is easy to look at a recursive solution and believe that you could have generated it on your own. Without trying it first, however, it is hard to know whether you would have come up with the same insight.

To give yourself more of a feel for the problem, you need to consider a concrete case. Suppose you want to generate all permutations of a five-character string, such as "**ABCDE**". In your solution, you can apply the recursive leap of faith to generate all permutations of any shorter string. Just assume that the recursive calls work and be done with it. Once again, the critical question is how being able to permute shorter strings helps you solve the problem of permuting the original five-character string.

The recursive insight

The key to solving the permutation problem is recognizing that the permutations of the five-character string "**ABCDE**" consist of the following strings:

- The character '**A**' followed by every possible permutation of "**BCDE**"
- The character '**B**' followed by every possible permutation of "**ACDE**"
- The character '**C**' followed by every possible permutation of "**ABDE**"
- The character '**D**' followed by every possible permutation of "**ABCE**"
- The character '**E**' followed by every possible permutation of "**ABCD**"

More generally, to display all permutations of a string of length n , you can take each of the n characters in turn and display that character followed by every possible permutation of the remaining $n - 1$ characters.

The only difficulty with this solution strategy is that the recursive subproblem does not have exactly the same form as the original. The original problem requires you to display all permutations of a string. The subproblem requires you to display a character from a string followed by all permutations of the remaining letters. As the recursion proceeds, the character in front will become two characters, then three, and so forth. The general subproblem, therefore, is to generate all permutations of a string, with some characters at the beginning of the string already fixed in their positions.

As discussed in Chapter 5, the easiest way to solve the problem of asymmetry between the original problem and its recursive subproblems is to define **ListPermutations** as a simple wrapper function that calls a subsidiary function to solve the more general case. In this example, the general problem can be solved by a new procedure **RecursivePermute**, which generates all permutations of the remaining characters in a string having already chosen some characters as the prefix. The prefix starts empty and

all the original letters still remain to be examined, which gives you the original problem. As the prefix grows and there are fewer characters remaining, the problem becomes simpler. When there are no characters remaining to be permuted, all characters have been placed in the prefix, and it can be displayed exactly as it appears. The definition of **ListPermutations** itself looks like this:

```
void ListPermutations(string str) {
    RecursivePermute("", str);
}
```

The **RecursivePermute** procedure follows the outline of the recursive permutation algorithm and has the following pseudocode form:

```
void RecursivePermute(string prefix, string rest) {
    if (rest is empty) {
        Display the prefix string.
    } else {
        For each character in rest {
            Add the character to the end of prefix.
            Remove character from rest.
            Use recursion to generate permutations with the updated values for prefix and rest.
        }
    }
}
```

Translating this function from pseudocode to C++ is reasonably simple. The full definition of **RecursivePermute** looks like this:

```
void RecursivePermute(string prefix, string rest) {
    if (rest == "") {
        cout << prefix << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string newPrefix = prefix + rest[i];
            string newRest = rest.substr(0, i) + rest.substr(i+1);
            RecursivePermute(newPrefix, newRest);
        }
    }
}
```

6.3 Graphical applications of recursion

Back in the early 1990s, I developed a simple graphics library that makes it possible to construct simple drawings on a computer display using line segments and circular arcs.¹ Using the graphics library makes programming more fun, but it also has proven to be useful in illustrating the concepts of recursion. Many of the figures you can create with the graphics library are fundamentally recursive in nature and provide excellent examples of recursive programming.

The remainder of this chapter includes an overview of the graphics library, followed by two illustrations of how recursion can be applied in the graphics domain. This material is not essential to learning about recursion, and you can skip it if you don't have ready access to the graphics library. On the other hand, working through these examples might make recursion seem a lot more powerful, not to mention more fun.

¹ Note that the C-based graphics library uses a very different drawing model from the **acm.graphics** package available in Java. We're working on a new graphics library for C++ that looks like the Java one.

The graphics library

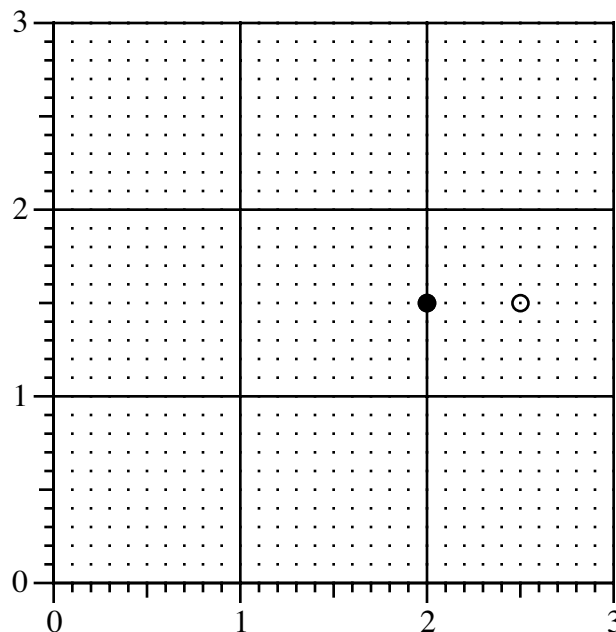
The **graphics.h** interface provides you with access to a collection of functions that enable you to create simple line drawings on the computer screen. When you initialize the graphics package, a new rectangular window called the **graphics window** is created on the screen and used as the drawing surface. Whenever you call procedures and functions in the graphics library, the results are displayed in the graphics window.

To specify points within the graphics window, the graphics library uses an approach that should be familiar from high-school geometry or algebra. All drawing in the graphics window takes place on a conceptual grid, as illustrated in Figure 6-1. As in traditional geometry, points are identified by specifying their position relative to the **origin**, which is the point at the lower left corner of the graphics window. The horizontal and vertical lines that emanate from the origin along the edges of the graphics window are called the **axes**; the x -axis runs along the bottom of the window, and the y -axis runs up the left side. Every point in the graphics window is identified by a pair of values, usually written as (x, y) , that specifies the position of that point along the x and y axes. These values are called the **coordinates** of the point. Coordinates are measured in inches relative to the origin, which is the point $(0, 0)$. From there, x values increase as you move to the right, and y values increase as you move up.

Coordinates in the graphics library come in two forms:

- **Absolute coordinates** specify a point in the window by giving its coordinates with respect to the origin. For example, the solid dot in Figure 6-1 is at absolute coordinates $(2.0, 1.5)$.
- **Relative coordinates** specify a position in the window by indicating how far away that point is along each axis from the last position specified. For example, the open dot in Figure 6-1 has absolute coordinates $(2.5, 1.5)$. If, however, you express its coordinates in relation to the solid dot, this point is shifted by the relative coordinates $(0.5, 0.0)$. If you want to connect these dots with a line, the standard approach is to specify the first point in absolute coordinates and the endpoint of the line in relative coordinates.

Figure 6-1 Coordinates in the graphics library



The best mental model to use for the drawing process is to imagine that there is a pen positioned over a piece of transparent graph paper covering the screen. You can move the pen to any location on the screen by specifying the absolute coordinates. You then draw a straight line by moving the pen to a new point specified using relative coordinates, making sure that the pen continuously touches the graph paper as you draw the line. From there, you can start another line beginning where the last one ended.

The functions exported by the **graphics.h** interface are shown in Table 6-1. Graphics applications begin by calling **InitGraphics**, after which the graphical image itself is created by calls to **MovePen**, **DrawLine**, and **DrawArc**. The remaining functions—**GetWindowWidth**, **GetWindowHeight**, **GetCurrentX**, and **GetCurrentY**—make it possible to retrieve information about the dimensions and state of the graphics window. These functions come up less frequently, but are nonetheless useful enough that it makes sense to include them in the interface.

To get a better sense of how the graphics library works, consider the following program, which draws a simple archway:

Table 6-1 Functions exported by `graphics.h`

Function call	Operation
InitGraphics()	This procedure creates the graphics window on the screen. The call to InitGraphics must precede any output operations of any kind and is usually the first statement in the function main .
MovePen (<i>x</i> , <i>y</i>)	This procedure picks up the pen and moves it—without drawing any lines—to the position (<i>x</i> , <i>y</i>), which is specified in absolute coordinates.
DrawLine (<i>dx</i> , <i>dy</i>)	This procedure draws a line extending from the current point by moving the pen <i>dx</i> inches in the <i>x</i> direction and <i>dy</i> inches in the <i>y</i> direction. The final position becomes the new current point.
DrawArc (<i>r</i> , <i>start</i> , <i>sweep</i>)	This procedure draws a circular arc, which always begins at the current point. The arc itself has radius <i>r</i> , and starts at the angle specified by the parameter <i>start</i> , relative to the center of the circle. This angle is measured in degrees counterclockwise from the 3 o'clock position along the <i>x</i> -axis, as in traditional mathematics. For example, if <i>start</i> is 0, the arc begins at the 3 o'clock position; if <i>start</i> is 90, the arc begins at the 12 o'clock position; and so on. The fraction of the circle drawn is specified by the parameter <i>sweep</i> , which is also measured in degrees. If <i>sweep</i> is 360, DrawArc draws a complete circle; if <i>sweep</i> is 90, it draws a quarter of a circle. If the value of <i>sweep</i> is positive, the arc is drawn counterclockwise from the current point; if <i>sweep</i> is negative, the arc is drawn clockwise. The current point at the end of the DrawArc operation is the final position of the pen along the arc.
GetWindowWidth() GetWindowHeight()	These functions return the width and height of the graphics window, respectively.
GetCurrentX() GetCurrentY()	These functions return the absolute coordinates of the current point.

```
int main() {  
    InitGraphics();  
    MovePen(2.0, 0.5);  
    DrawLine(1.0, 0.0);  
    DrawLine(0.0, 1.0);  
    DrawArc(0.5, 0, 180);  
    DrawLine(0.0, -1.0);  
    return 0;  
}
```

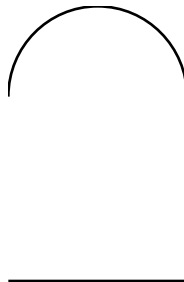
The program begins, like all graphics programs, with a call to **InitGraphics**, which creates an empty graphics window. The next two statements then move the pen to the point (2.0, 0.5) and draw a line with the relative coordinates (1.0, 0.0). The effect of these statements is to draw a 1-inch horizontal line near the bottom of the window. The next call to **DrawLine** adds a vertical line that begins where the first line ended. Thus, at this point, the graphics window contains two lines in the following configuration:



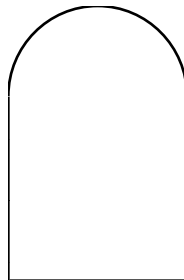
The next statement

```
DrawArc(0.5, 0, 180);
```

draws a circular arc with a radius of 0.5 inches. Because the second argument is 0, the arc begins at the 0 degree mark, which corresponds to the 3 o'clock position. From there, the third argument indicates that the arc runs in the positive direction (counterclockwise) for a total of 180 degrees, or halfway around the circle. Adding this semicircle to the line segments generated earlier makes the graphics window look like this:



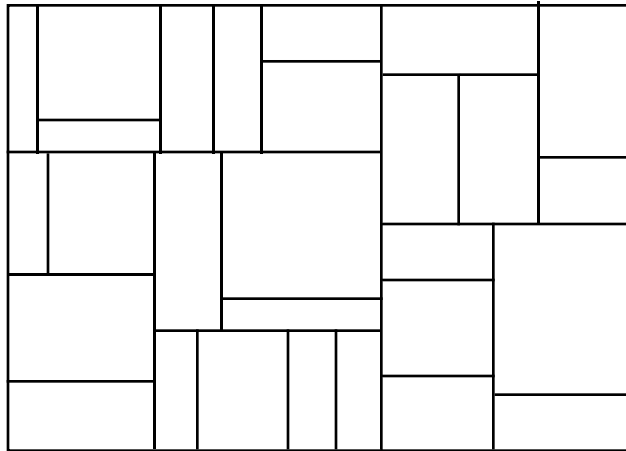
The last line in the program draws a 1-inch vertical line in the downward direction, which completes the archway, as shown:



An example from computer art

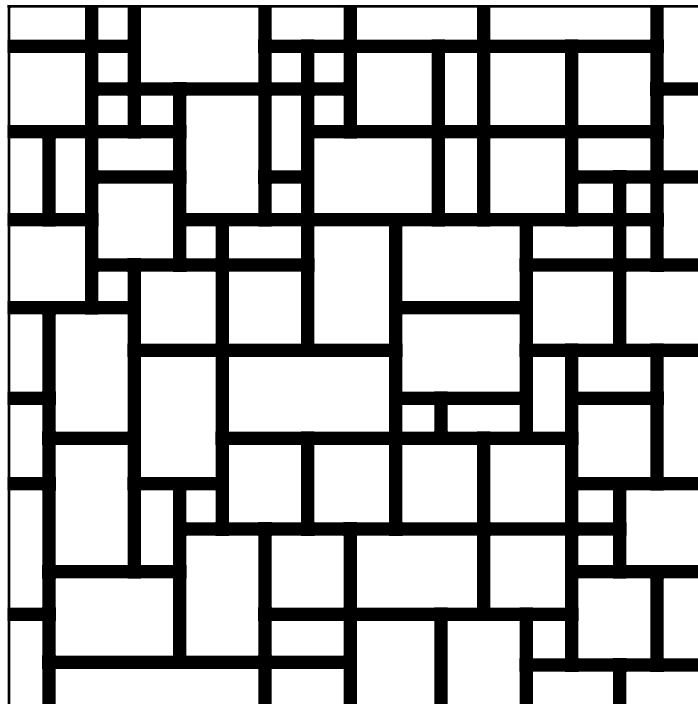
In the early part of the twentieth century, a controversial artistic movement arose in Paris, largely under the influence of Pablo Picasso and Georges Braque. The Cubists—as they were called by their critics—rejected classical artistic notions of perspective and representationalism and instead produced highly fragmented works based on primitive geometrical forms. Strongly influenced by Cubism, the Dutch painter Piet Mondrian (1872–1944) produced a series of compositions based on horizontal and vertical lines, such as the one shown in Figure 6-2.

Suppose that you want to generate compositions such as the following, which—like much of Mondrian’s work—consists only of horizontal and vertical lines:

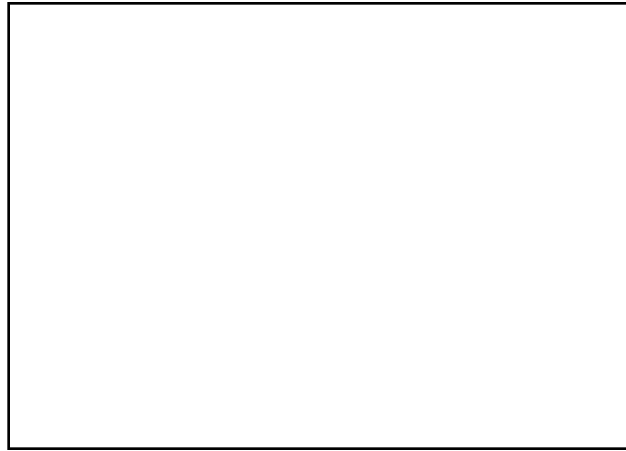


How would you go about designing a general strategy to create such a figure using the graphics library?

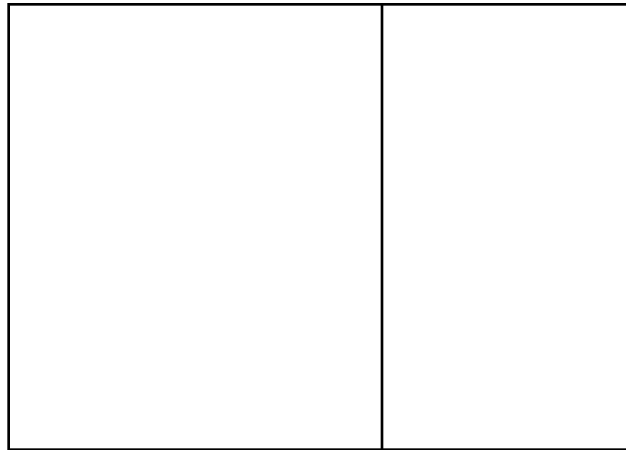
Figure 6-2 Grid pattern from Piet Mondrian, “Composition with Grid 6,” 1919



To understand how a program might produce such a figure, it helps to think about the process as one of successive decomposition. At the beginning, the canvas was simply an empty rectangle that looked like this:



If you want to subdivide the canvas using a series of horizontal and vertical lines, the easiest way to start is by drawing a single line that divides the rectangle in two:



If you're thinking recursively, the thing to notice at this point is that you now have two empty rectangular canvases, each of which is smaller in size. The task of subdividing these rectangles is the same as before, so you can perform it by using a recursive implementation of the same procedure. Since the new rectangles are taller than they are wide, you might choose to use a horizontal dividing line, but the basic process remains the same.

At this point, the only thing needed for a complete recursive strategy is a simple case. The process of dividing up rectangles can't go on indefinitely. As the rectangles get smaller and smaller, at some point the process has to stop. One approach is to look at the area of each rectangle before you start. Once the area of a rectangle falls below some threshold, you needn't bother to subdivide it any further.

The `mondrian.cpp` program shown in Figure 6-3 implements the recursive algorithm, using the entire graphics window as the initial canvas.

Figure 6-3 Program to generate a Mondrian-style drawing

```
/*
 * File: mondrian.cpp
 * -----
 * This program creates a random line drawing in a style reminiscent
 * of the Dutch painter Piet Mondrian. The picture is generated by
 * recursively subdividing the canvas into successively smaller
 * rectangles with randomly chosen horizontal and vertical lines.
 */

#include "genlib.h"
#include "graphics.h"
#include "random.h"

/*
 * Constants
 * -----
 * MIN_AREA -- Smallest square that will be split
 * MIN_EDGE -- Minimum fraction on each side of dividing line
 */

const double MIN_AREA = 0.5;
const double MIN_EDGE = 0.15;

/* Private function prototypes */

void SubdivideCanvas(double x, double y, double width, double height);

/* Main program */

int main() {
    InitGraphics();
    Randomize();
    SubdivideCanvas(0, 0, GetWindowWidth(), GetWindowHeight());
    return 0;
}

/*
 * Function: SubdivideCanvas
 * Usage: SubdivideCanvas(x, y, width, height);
 * -----
 * This function decomposes a canvas by recursive subdivision. The
 * lower left corner of the canvas is the point (x, y), and the
 * dimensions are given by the width and height parameters. The
 * function first checks for the simple case, which is obtained
 * when the size of the rectangular canvas is too small to subdivide
 * (area < MIN_AREA). In the simple case, the function does nothing.
 * If the area is larger than the minimum, the function first
 * decides whether to split the canvas horizontally or vertically,
 * choosing the larger dimension. The function then chooses a
 * random dividing line, making sure to leave at least MIN_EDGE on
 * each side. The program then uses a divide-and-conquer strategy
 * to subdivide the two new rectangles.
 */
```

```

void SubdivideCanvas(double x, double y, double width, double height) {
    if (width * height >= MIN_AREA) {
        if (width > height) {
            double mid = width * RandomReal(MIN_EDGE, 1 - MIN_EDGE);
            MovePen(x + mid, y);
            DrawLine(0, height);
            SubdivideCanvas(x, y, mid, height);
            SubdivideCanvas(x + mid, y, width - mid, height);
        } else {
            double mid = height * RandomReal(MIN_EDGE, 1 - MIN_EDGE);
            MovePen(x, y + mid);
            DrawLine(width, 0);
            SubdivideCanvas(x, y, width, mid);
            SubdivideCanvas(x, y + mid, width, height - mid);
        }
    }
}

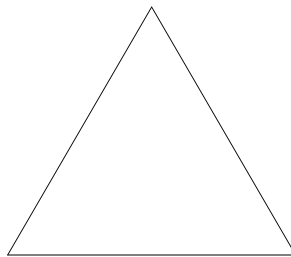
```

In `mondrian.cpp`, the recursive function `SubdivideCanvas` does all the work. The arguments give the position and dimensions of the current rectangle on the canvas. At each step in the decomposition, the function simply checks to see whether the rectangle is large enough to split. If it is, the function checks to see which dimension—width or height—is larger and accordingly divides the rectangle with a vertical or horizontal line. In each case, the function draws only a single line; all remaining lines in the figure are drawn by subsequent recursive calls.

Fractals

In the late 1970s, a researcher at IBM named Benoit Mandelbrot generated a great deal of excitement by publishing a book on **fractals**, which are geometrical structures in which the same pattern is repeated at many different scales. Although mathematicians have known about fractals for a long time, there was a resurgence of interest in the subject during the 1980s, partly because the development of computers made it possible to do so much more with fractals than had ever been possible before.

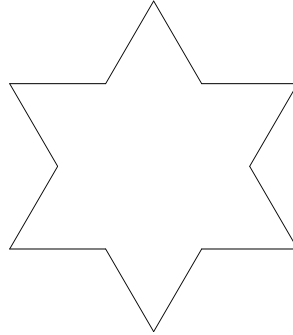
One of the earliest examples of fractal figures is called the **Koch snowflake** after its inventor, Helge von Koch. The Koch snowflake begins with an equilateral triangle like this:



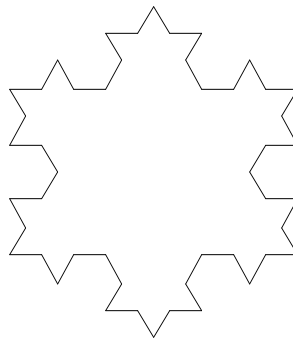
This triangle, in which the sides are straight lines, is called the Koch fractal of order 0. The figure is then revised in stages to generate fractals of successively higher orders. At each stage, every straight-line segment in the figure is replaced by one in which the middle third consists of a triangular bump protruding outward from the figure. Thus, the first step is to replace each line segment in the triangle with a line that looks like this:



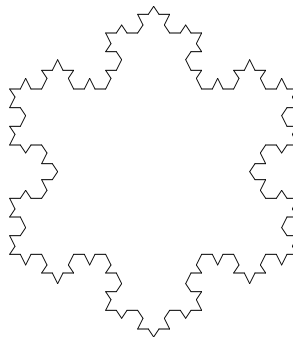
Applying this transformation to each of the three sides of the original triangle generates the Koch fractal of order 1, as follows:



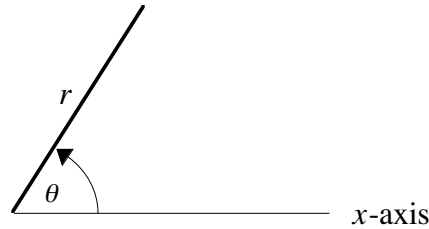
If you then replace each line segment in this figure with a new line that again includes a triangular wedge, you create the following order-2 Koch fractal:



Replacing each of these line segments gives the order-3 fractal shown in the following diagram, which has started to resemble a snowflake:



Because figures like the Koch fractal are much easier to draw by computer than by hand, it makes sense to write a program that uses the graphics library to generate this design. Before designing the program itself, however, it helps to introduce a new procedure that will prove useful in a variety of graphical applications. The **DrawLine** primitive in the graphics library requires you to specify the relative coordinates of the new endpoint as a pair of values, dx and dy . In many graphical applications, it is much easier to think of lines as having a length and a direction. For example, the solid line in the following diagram can be identified by its length (r) and its angle from the x -axis (θ):



In mathematics, the parameters r and θ are called the **polar coordinates** of the line. Converting from polar coordinates to the more traditional Cartesian coordinates used in the graphics library requires a little trigonometry, as shown in the following implementation of the procedure **DrawPolarLine**, which draws a line of length r in the direction θ , measured in degrees counterclockwise from the x -axis:

```
const double PI = 3.1415926535;

void DrawPolarLine(double r, double theta) {
    double radians = theta / 180 * PI;
    DrawLine(r * cos(radians), r * sin(radians));
}
```

If you don't understand trigonometry, don't worry. You don't need to understand the implementation of **DrawPolarLine** to use it. If you have had a trigonometry course, most of this implementation should be straightforward; the only complexity comes from the fact that the library functions **sin** and **cos** are defined to take their arguments in radian measure, which means that the implementation must convert the **theta** parameter from degrees to radians prior to calling the trigonometric functions.

Given **DrawPolarLine**, it is very easy to draw the equilateral triangle that represents the Koch snowflake of order 0. If **size** is the length of one side of the triangle, all you need to do is position the pen at the lower left corner of the figure and make the following calls:

```
DrawPolarLine(size, 0);
DrawPolarLine(size, 120);
DrawPolarLine(size, 240);
```

But how would you go about drawing a more complicated Koch fractal, that is, one of a higher order? The first step is simply to replace each of the calls to **DrawPolarLine** with a call to a new procedure that draws a fractal line of a specified order. Thus, the three calls in the main program look like this:

```
DrawFractalLine(size, 0, order);
DrawFractalLine(size, 120, order);
DrawFractalLine(size, 240, order);
```

The next task is to implement **DrawFractalLine**, which is easy if you think about it recursively. The simple case for **DrawFractalLine** occurs when **order** is 0, in which case the function simply draws a straight line with the specified length and direction. If **order** is greater than 0, the fractal line is broken down into four components, each of which is itself a fractal line of the next lower order. Thus, the implementation of **DrawFractalLine** looks like this:

```

void DrawFractalLine(double len, double theta, int order) {
    if (order == 0) {
        DrawPolarLine(len, theta);
    } else {
        DrawFractalLine(len/3, theta, order - 1);
        DrawFractalLine(len/3, theta - 60, order - 1);
        DrawFractalLine(len/3, theta + 60, order - 1);
        DrawFractalLine(len/3, theta, order - 1);
    }
}

```

The complete implementation of the `koch.cpp` program is shown in Figure 6-4.

Figure 6-4 Program to draw a Koch fractal snowflake

```

/*
 * File: koch.cpp
 * -----
 * This program draws a Koch fractal.
 */

#include <iostream>
#include <cmath>
#include "simpio.h"
#include "graphics.h"
#include "genlib.h"

/* Constants */

const double PI = 3.1415926535;

/* Private function prototypes */

void KochFractal(double size, int order);
void DrawFractalLine(double len, double theta, int order);
void DrawPolarLine(double r, double theta);

/* Main program */

int main() {
    InitGraphics();
    cout << "Program to draw Koch fractals" << endl;
    cout << "Enter edge length in inches: ";
    double size = GetReal();
    cout << "Enter order of fractal: ";
    int order = GetInteger();
    KochFractal(size, order);
    return 0;
}

/*
 * Function: KochFractal
 * Usage: KochFractal(size, order);
 * -----
 * This function draws a Koch fractal snowflake centered in
 * the graphics window of the indicated size and order.
 */

```

```

void KochFractal(double size, int order) {
    double x0 = GetWindowWidth() / 2 - size / 2;
    double y0 = GetWindowHeight() / 2 - sqrt(3) * size / 6;
    MovePen(x0, y0);
    DrawFractalLine(size, 0, order);
    DrawFractalLine(size, 120, order);
    DrawFractalLine(size, 240, order);
}

/*
 * Function: DrawFractalLine
 * Usage: DrawFractalLine(len, theta, order);
 * -----
 * This function draws a fractal line of the given length, starting
 * from the current point and moving in direction theta. If order
 * is 0, the fractal line is a straight line. If order is greater
 * than zero, the line is divided into four line segments, each of
 * which is a fractal line of the next lower order. These segments
 * connect the same endpoints as the straight line, but include
 * a triangular wedge replacing the center third of the segment.
 */

void DrawFractalLine(double len, double theta, int order) {
    if (order == 0) {
        DrawPolarLine(len, theta);
    } else {
        DrawFractalLine(len/3, theta, order - 1);
        DrawFractalLine(len/3, theta - 60, order - 1);
        DrawFractalLine(len/3, theta + 60, order - 1);
        DrawFractalLine(len/3, theta, order - 1);
    }
}

/*
 * Function: DrawPolarLine
 * Usage: DrawPolarLine(r, theta);
 * -----
 * This function draws a line of length r in the direction
 * specified by the angle theta, measured in degrees.
 */

void DrawPolarLine(double r, double theta) {
    double radians = theta / 180 * PI;
    DrawLine(r * cos(radians), r * sin(radians));
}

```

Summary

Except for the discussion of the graphics library in the section entitled “Graphical applications of recursion” earlier in this chapter, relatively few new concepts have been introduced in Chapter 6. The fundamental precepts of recursion were introduced in Chapter 5. The point of Chapter 6 is to raise the sophistication level of the recursive examples to the point at which the problems become difficult to solve in any other way. Because of this increase in sophistication, beginning students often find these problems much harder to comprehend than those in the preceding chapter. Indeed, they are harder, but recursion is a tool for solving hard problems. To master it, you need to practice with problems at this level of complexity.

The important points in this chapter include:

- Whenever you want to apply recursion to a programming problem, you have to devise a strategy that transforms the problem into simpler instances of the same problem. Until you find the correct insight that leads to the recursive strategy, there is no way to apply recursive techniques.
- Once you identify a recursive approach, it is important for you to check your strategy to ensure that it does not violate any conditions imposed by the problem.
- When the problems you are trying to solve increase in complexity, the importance of accepting the recursive leap of faith increases.
- Recursion is not magical. If you need to do so, you can simulate the operation of the computer yourself by drawing the stack frames for every procedure that is called in the course of the solution. On the other hand, it is critical to get beyond the skepticism that forces you to look at all the underlying details.
- Wrapper functions are useful in complex recursive procedures, just as they are for the simple recursive functions presented in Chapter 5.
- The **graphics.h** library makes it possible to display simple graphical drawings, many of which have an interesting recursive structure.

Review questions

1. In your own words, describe the recursive insight necessary to solve the Tower of Hanoi puzzle.
2. What is wrong with the following strategy for solving the recursive case of the Tower of Hanoi puzzle:
 - a. Move the top disk from the start spire to the temporary spire.
 - b. Move a stack of $N-1$ disks from the start spire to the finish spire.
 - c. Move the top disk now on the temporary spire back to the finish spire.
3. If you call

MoveTower(16, 'A', 'B', 'C')

what line is displayed by **MoveSingleDisk** as the first step in the solution? What is the last step in the solution?

4. What is a permutation?
5. In your own words, explain the recursive insight necessary to enumerate the permutations of the characters in a string.
6. How many permutations are there of the string "**WXYZ**"?
7. Why is it necessary to define both **ListPermutations** and **RecursivePermute** in the permutation problem?
8. Where is the origin located in the graphics window?
9. What is the difference between absolute and relative coordinates?
10. What are the eight functions exported by the **graphics.h** interface?
11. What simple case is used to terminate the recursion in **mondrian.cpp**?

12. Draw a picture of the order-1 Koch fractal.
13. How many line segments appear in the order-2 Koch fractal?
14. From the caller's point of view, describe the effect of the function **DrawPolarLine**.

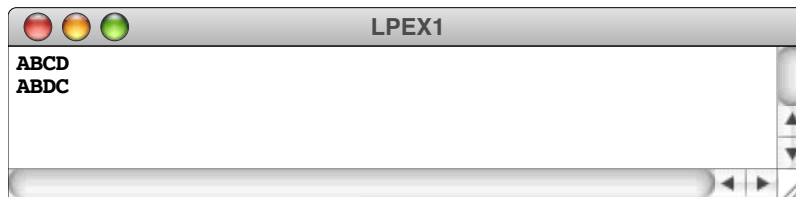
Programming exercises

1. Following the logic of the **MoveTower** function, write a recursive function **NHanoiMoves(n)** that calculates the number of individual moves required to solve the Tower of Hanoi puzzle for **n** disks.
2. To make the operation of the program somewhat easier to explain, the implementation of **MoveTower** in this chapter uses

```
if (n == 1)
```

as its simple case test. Whenever you see a recursive program use 1 as its simple case, it pays to be a little skeptical; in most applications, 0 is a more appropriate choice. Rewrite the Tower of Hanoi program so that the **MoveTower** function checks whether **n** is 0 instead. What happens to the length of the **MoveTower** implementation?

3. Rewrite the Tower of Hanoi program so that it uses an explicit stack of pending tasks instead of recursion. In this context, a task can be represented most easily as a structure containing the number of disks to move and the names of the spires used for the start, finish, and temporary repositories. At the beginning of the process, you push onto your stack a single task that describes the process of moving the entire tower. The program then repeatedly pops the stack and executes the task found there until no tasks are left. Except for the simple cases, the process of executing a task results in the creation of more tasks that get pushed onto the stack for later execution.
4. As presented in the text, the function **RecursivePermute** takes two strings, which indicate how far the permutation process has progressed. You could also design the program so that **RecursivePermute** takes a string and an integer, where the string is the concatenation of the fixed prefix with a suffix whose characters can still be permuted. The integer indicates the number of characters in the prefix and thus the index at which the remaining characters begin within the string. For example, if you call the redesigned **RecursivePermute** function on the arguments "**ABCD**" and **2**, the output should be

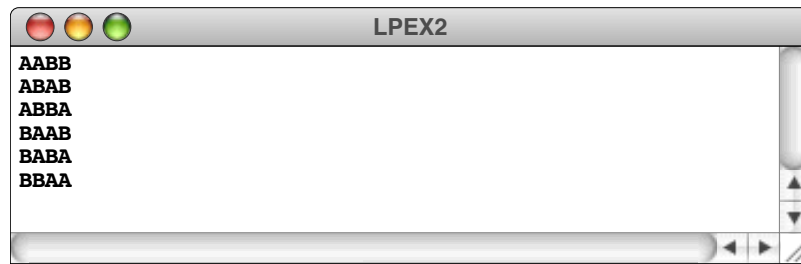


which is all strings beginning with "**AB**" followed by some permutation of "**CD**".

Rewrite the permutation program so that it uses this new design.

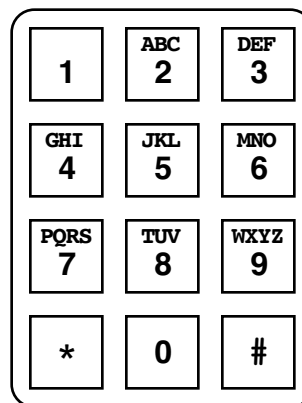
5. Update the permutation algorithm from the text to generate the correct list of permutations even if the string contains repeated letters. For example, if you call **ListPermutations** on the string "**AABB**", your program should not generate as many permutations as it does for the string "**ABCD**" because some of the strings

generated by the standard algorithm would be indistinguishable from others. Your program should instead generate the following six:



Write a new implementation of **ListPermutations** that works correctly even if the string contains duplicated letters. In writing this implementation, you should not merely keep a list of the permutations that have already been encountered and avoid generating duplicates. Instead, you should think carefully about the recursive structure of the problem and find a way to avoid generating the extra permutations in the first place.

6. On a telephone keypad, the digits are mapped onto the alphabet as shown in the diagram below:



In order to make their phone numbers more memorable, service providers like to find numbers that spell out some word (called a **mnemonic**) appropriate to their business that makes that phone number easier to remember. For example, the phone number for a recorded time-of-day message in some localities is 637-8687 (NERVOUS).

Imagine that you have just been hired by a local telephone company to write a function **ListMnemonics** that will generate all possible letter combinations that correspond to a given number, represented as a string of digits. For example, the call

ListMnemonics("723")

should generate the following 36 possible letter combinations that correspond to that prefix:

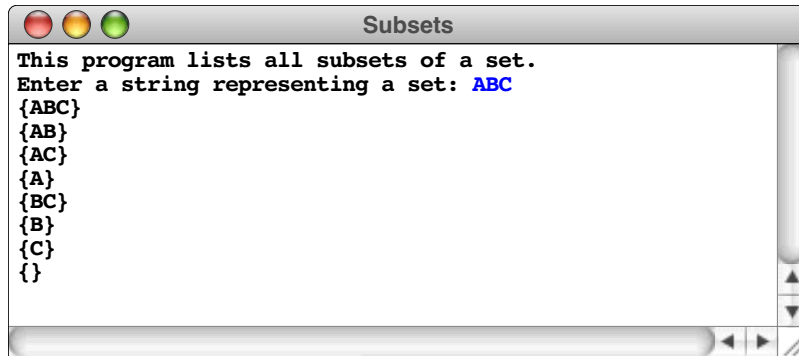
PAD	PBD	PCD	QAD	QBD	QCD	RAD	RBD	RCD	SAD	SBD	SCD
PAE	PBE	PCE	QAE	QBE	QCE	RAE	RBE	RCE	SAE	SBE	SCE
PAF	PBF	PCF	QAF	QBF	QCF	RAF	RBF	RCF	SAF	SBF	SCF

7. Rewrite the program from exercise 6 so that it uses the **Lexicon** class and the **EnglishWords.dat** file from Chapter 4 so that the program only lists mnemonics that are valid English words.

8. Using the **ListPermutations** example as a starting point, write a function **ListSubsets** that generates all possible subsets of a given set, where the set is represented by a string of letters. For example, if you call the function

```
ListSubsets("ABC");
```

your function should produce the following output:



Like permutations, the subset problem has a recursive formulation. If you represent a set of characters using a string that either contains or does not contain a given letter, you can calculate all possible subsets by (1) including the first character in the subset and concatenating it onto the front of all subsets of the remaining $N-1$ characters and then (2) displaying the subsets of the remaining $N-1$ without this character.

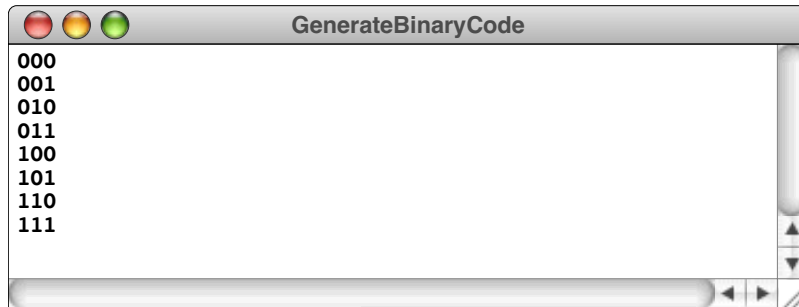
9. Inside a computer system, integers are represented as a sequence of bits, each of which is a single digit in the binary number system and can therefore have only the value 0 or 1. With N bits, you can represent 2^N distinct integers. For example, three bits are sufficient to represent the eight (2^3) integers between 0 and 7, as follows:

0 0 0	→	0
0 0 1	→	1
0 1 0	→	2
0 1 1	→	3
1 0 0	→	4
1 0 1	→	5
1 1 0	→	6
1 1 1	→	7

Each entry in the left side of the table is written in its standard binary representation, in which each bit position counts for twice as much as the position to its right. For instance, you can demonstrate that the binary value **110** represents the decimal number 6 by following the logic shown in the following diagram:

$$\begin{array}{rcccc}
 \text{place value} & \rightarrow & 4 & 2 & 1 \\
 & & \times & \times & \times \\
 \text{binary digits} & \rightarrow & \mathbf{1} & \mathbf{1} & \mathbf{0} \\
 & & \parallel & \parallel & \parallel \\
 & & 4 & + & 2 & + & 0 & = & 6
 \end{array}$$

Write a recursive function **GenerateBinaryCode(nBits)** that generates the bit patterns for the standard binary representation of all integers that can be represented using the specified number of bits. For example, calling **GenerateBinaryCode(3)** should produce the following output:



10. Although the binary coding used in exercise 7 is ideal for most applications, it has certain drawbacks. As you count in standard binary notation, there are some points in the sequence at which several bits change at the same time. For example, in the three-bit binary code, the value of every bit changes as you move from 3 (**011**) to 4 (**100**).

In some applications, this instability in the bit patterns used to represent adjacent numbers can lead to problems. Imagine for the moment that you are using some hardware measurement device that produces a three-bit value from some real-world phenomenon that happens to be varying between 3 and 4. Sometimes, the device will register **011** to indicate the value 3; at other times, it will register **100** to indicate 4. For this device to work correctly, the transitions for each of the individual bits must occur simultaneously. If the first bit changes more quickly than the others, for example, there may be an intermediate state in which the device reads **111**, which would be a highly inaccurate reading.

It is interesting to discover that you can avoid this problem simply by changing the numbering system. If instead of using binary representation in the traditional way, you can assign three-bit values to each of the numbers 0 through 7 with the highly useful property that only one bit changes in the representation between every pair of adjacent integers. Such an encoding is called a *Gray code* (after its inventor, the mathematician Frank Gray) and looks like this:

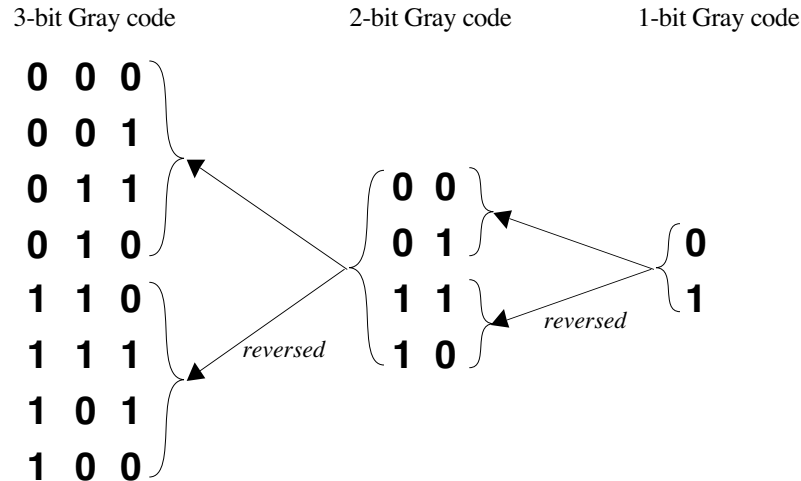
0 0 0	→	0
0 0 1	→	1
0 1 1	→	2
0 1 0	→	3
1 1 0	→	4
1 1 1	→	5
1 0 1	→	6
1 0 0	→	7

Note that, in the Gray code representation, the bit patterns for 3 and 4 differ only in their leftmost bit. If the hardware measurement device used Gray codes, a value oscillating between 3 and 4 would simply turn that bit on and off, eliminating any problems with synchronization.

The recursive insight that you need to create a Gray code of N bits is summarized in the following informal procedure:

1. Write down the Gray code for $N - 1$ bits.
2. Copy that same list *in reverse order* below the original one.
3. Add a **0** bit in front of the codings in the original half of the list and a **1** bit in front of those in the reversed copy.

This procedure is illustrated in the following derivation of the Gray code for three bits:



Write a recursive function **GenerateGrayCode(nBits)** that generates the Gray code patterns for the specified number of bits.

11. Given a set of numbers, the *partition problem* is to find a subset of the numbers that add up to a specific target number. For example, there are two ways to partition the set {1, 3, 4, 5} so that the remaining elements add up to 5:
 - Select the 1 and the 4
 - Select just the 5

By contrast, there is no way to partition the set {1, 3, 4, 5} to get 11.

Write a function **NumberOfPartitions** that takes an array of integers, the length of that array, and a target number, and returns the number of partitions of that set of integers which add up to the target. For example, suppose that the array **sampleSet** has been initialized as follows:

```
int sampleSet[] = {1, 3, 4, 5};
```

Given this definition of **sampleSet**, calling

```
NumberOfPartitions(sampleSet, 4, 5);
```

should return 2 (there are two ways to make 5), and calling

```
NumberOfPartitions(sampleSet, 4, 11)
```

should return 0 (there are no ways to make 11).

The prototype for **NumberOfPartitions** is

```
int NumberOfPartitions(int array[], int length, int target);
```

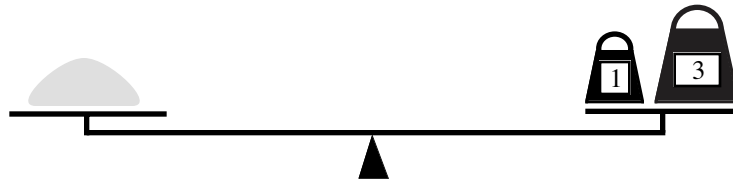
In order to see the recursive nature of this problem, think about any specific element in the set, such as the first element. If you think about all the partitions of a particular target number, some of them will include the first element and some won't. If you count those that do include the first element and then add that total to the number of those which leave out that element, you get the total number of partitions. Each of these two computations, however, can be expressed as a problem in the same form as the outer partition problem and can therefore be solved recursively.

12. *I am the only child of parents who weighed, measured, and priced everything; for whom what could not be weighed, measured, and priced had no existence.*

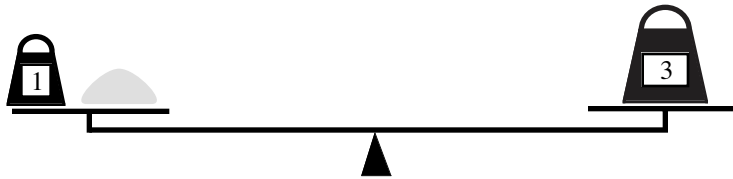
— Charles Dickens, *Little Dorrit*, 1857

In Dickens's time, merchants measured many commodities using weights and a two-pan balance—a practice that continues in many parts of the world today. If you are using a limited set of weights, however, you can only measure certain quantities accurately.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces, as shown:



It is somewhat more interesting to discover that you can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:



Write a recursive function

```
bool IsMeasurable(int target, int weights[], int nWeights)
```

that determines whether it is possible to measure out the desired target amount with a given set of weights. The available weights are stored in the array **weights**, which has **nWeights** as its effective size. For instance, the sample set of two weights illustrated above could be represented using the following pair of variables:

```
int sampleWeights[] = { 1, 3 };  
int nSampleWeights = 2;
```

Given these values, the function call

IsMeasurable(2, sampleWeights, nSampleWeights)

should return **true** because it is possible to measure out 2 ounces using the sample weight set as illustrated in the preceding diagram. On the other hand, calling

IsMeasurable(5, sampleWeights, nSampleWeights)

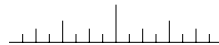
should return **false** because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces.

The fundamental observation you need to make for this problem is that each weight in the array can be either

1. Put on the opposite side of the balance from the sample
2. Put on the same side of the balance as the sample
3. Left off the balance entirely

If you consider one of the weights in the array and determine how choosing one of these three options affects the rest of the problem, you should be able to come up with the recursive insight you need to solve the problem.

13. In countries like the United States that still use the traditional English system of measurement, each inch on a ruler is marked off into fractions using tick marks that look like this:



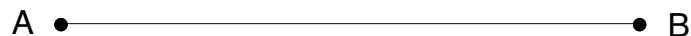
The longest tick mark falls at the half-inch position, two smaller tick marks indicate the quarter inches, and even smaller ones are used to mark the eighths and sixteenths. Write a recursive program that draws a 1-inch line at the center of the graphics window and then draws the tick marks shown in the diagram. Assume that the length of the tick mark indicating the half-inch position is given by the constant definition

```
const double HALF_INCH_TICK = 0.2;
```

and that each smaller tick mark is half the size of the next larger one.

14. One of the reasons that fractals have generated so much interest is that they turn out to be useful in some surprising practical contexts. For example, the most successful techniques for drawing computer images of mountains and certain other landscape features involve using fractal geometry.

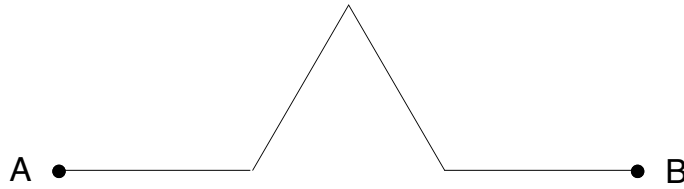
As a simple example of where this issue comes up, consider the problem of connecting two points A and B with a fractal that looks like a coastline on a map. The simplest possible strategy would be to draw a straight line between the two points:



This is the order 0 coastline and represents the base case of the recursion.

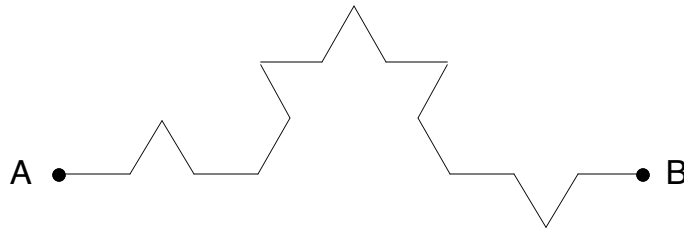
Of course, a real coastline will have small peninsulas or inlets somewhere along its length, so you would expect a realistic drawing of a coastline to jut in or out occasionally like a real one. As a first approximation, you could replace the straight

line with precisely the same fractal line used to create the Koch snowflake in the program described in the section on “Fractals” earlier in the chapter, as follows:

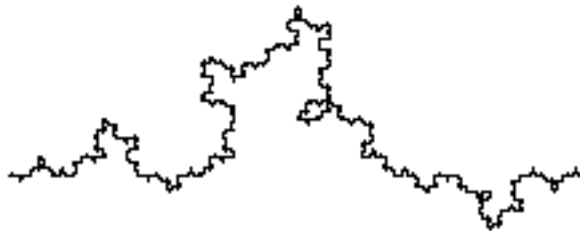


This process gives the order 1 coastline. However, in order to give the feeling of a traditional coastline, it is important for the triangular wedge in this line sometimes to point up and sometimes down, with equal probability.

If you then replace each of the straight line segments in the order 1 fractal with a fractal line in a random direction, you get the order 2 coastline, which might look like this:



Continuing this process eventually results in a drawing that conveys a remarkably realistic sense, as in this order 5 coastline:



Write a function **DrawCoastline** that fits the following interface description:

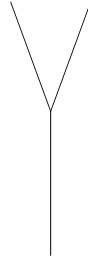
```
/*
 * Function: DrawCoastline
 * Usage: DrawCoastline(length, theta, order);
 * -----
 * The DrawCoastline function starts at the current (x, y)
 * position and draws a fractal coastline of the specified
 * length moving in the direction given by the angle theta
 * (as defined in the definition of DrawPolarLine in the
 * preceding problem); order gives the number of recursive
 * subdivisions into which each segment will be divided.
 */

void DrawCoastline(double length, double theta, int order);
```

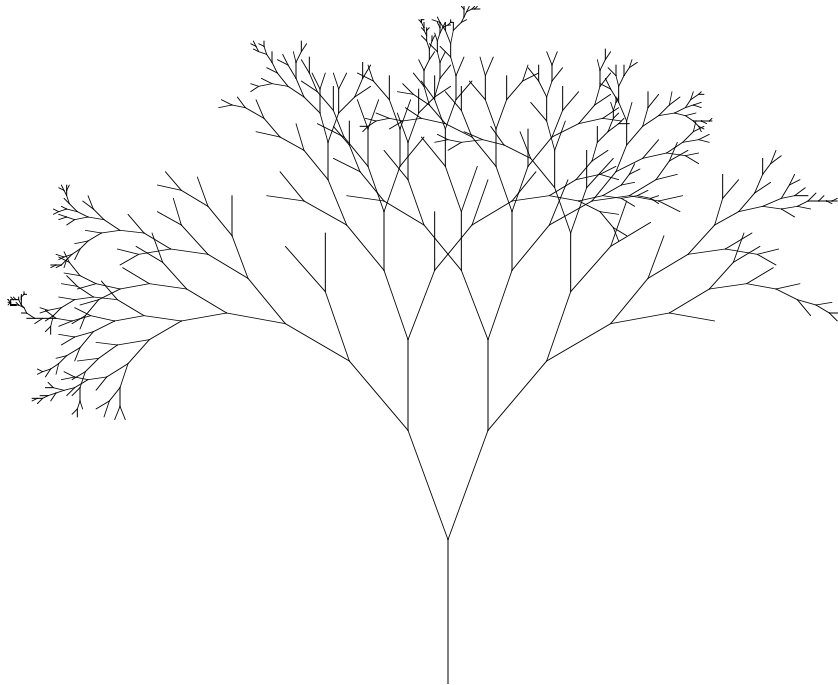
15. Recursive decomposition can also be used to draw a stylized representation of a tree. The tree begins as a simple trunk indicated by a straight vertical line, as follows:



The trunk may branch at the top to form two lines that veer off at an angle, as shown:



These branches may themselves split to form new branches, which split to form new ones, and so on. If the decision to branch is made randomly at each step of the process, the tree will eventually become unsymmetrical and will end up looking a little more like trees in nature, as illustrated by the following diagram:



If you think about this process recursively, however, you can see that all trees constructed in this way consist of a trunk, optionally topped by two trees that veer off at an angle. If the probability of branching is a function of the length of the current branch, the process will eventually terminate as the branches get progressively shorter.

Write a program **drawtree.cpp** that uses this recursive strategy and the graphics library to draw a stylized line drawing of a tree.

Chapter 7

Backtracking Algorithms

Truth is not discovered by proofs but by exploration. It is always experimental.

— Simone Weil, *The New York Notebook*, 1942

For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path. If you make the correct set of choices, you end up at the solution. On the other hand, if you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different path. Algorithms that use this approach are called **backtracking algorithms**.

If you think about a backtracking algorithm as the process of repeatedly exploring paths until you encounter the solution, the process appears to have an iterative character. As it happens, however, most problems of this form are easier to solve recursively. The fundamental recursive insight is simply this: a backtracking problem has a solution if and only if at least one of the smaller backtracking problems that results from making each possible initial choice has a solution. The examples in this chapter are designed to illustrate this process and demonstrate the power of recursion in this domain.

7.1 Solving a maze by recursive backtracking

Once upon a time, in the days of Greek mythology, the Mediterranean island of Crete was ruled by a tyrannical king named Minos. From time to time, Minos demanded tribute from the city of Athens in the form of young men and women, whom he would sacrifice to the Minotaur—a fearsome beast with the head of a bull and the body of a man. To house this deadly creature, Minos forced his servant Daedalus (the engineering genius who later escaped the island by constructing a set of wings) to build a vast underground labyrinth at Knossos. The young sacrifices from Athens would be led into the labyrinth, where they would be eaten by the Minotaur before they could find their way out. This tragedy continued until young Theseus of Athens volunteered to be one of the sacrifices. Following the advice of Minos’s daughter Ariadne, Theseus entered the labyrinth with a sword and a ball of string. After slaying the monster, Theseus was able to find his way back to the exit by unwinding the string as he went along.

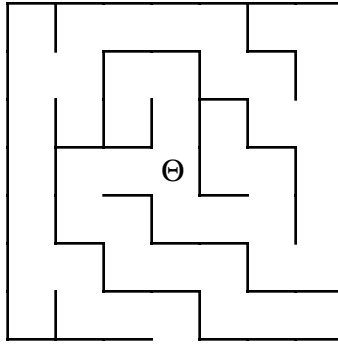
The right-hand rule

Theseus’s strategy represents an algorithm for escaping from a maze, but not everyone in such a predicament is lucky enough to have a ball of string or an accomplice clever enough to suggest such an effective approach. Fortunately, there are other strategies for escaping from a maze. Of these strategies, the best known is called the **right-hand rule**, which can be expressed in the following pseudocode form:

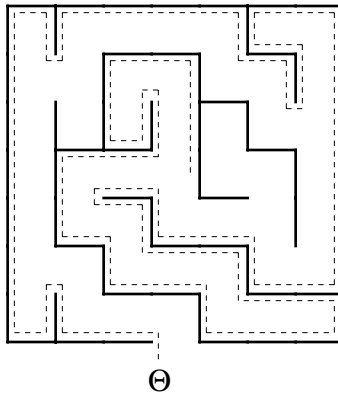
```
Put your right hand against a wall.
while (you have not yet escaped from the maze) {
    Walk forward keeping your right hand on a wall.
}
```

As you walk, the requirement that you keep your right hand touching the wall may force you to turn corners and occasionally retrace your steps. Even so, following the right-hand rule guarantees that you will always be able to find an opening to the outside of any maze.

To visualize the operation of the right-hand rule, imagine that Theseus has successfully dispatched the Minotaur and is now standing in the position marked by the first character in Theseus’s name, the Greek letter theta (Θ):



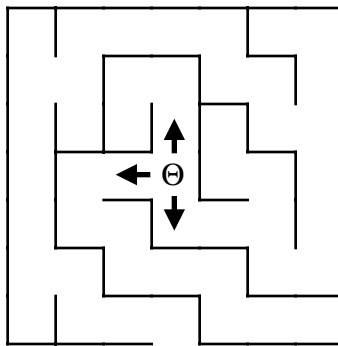
If Theseus puts his right hand on the wall and then follows the right-hand rule from there, he will trace out the path shown by the dashed line in this diagram:



Finding a recursive approach

As the **while** loop in its pseudocode form makes clear, the right-hand rule is an *iterative* strategy. You can, however, also think about the process of solving a maze from a *recursive* perspective. To do so, you must adopt a different mindset. You can no longer think about the problem in terms of finding a complete path. Instead, your goal is to find a recursive insight that simplifies the problem, one step at a time. Once you have made the simplification, you use the same process to solve each of the resulting subproblems.

Let's go back to the initial configuration of the maze shown in the illustration of the right-hand rule. Put yourself in Theseus's position. From the initial configuration, you have three choices, as indicated by the arrows in the following diagram:



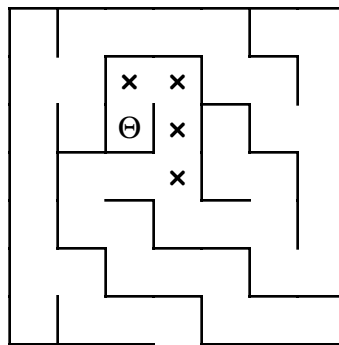
The exit, if any, must lie along one of those paths. Moreover, if you choose the correct direction, you will be one step closer to the solution. The maze has therefore become

simpler along that path, which is the key to a recursive solution. This observation suggests the necessary recursive insight. The original maze has a solution if and only if it is possible to solve at least one of the new mazes shown in Figure 7-1. The $\times$ in each diagram marks the original starting square and is off-limits for any of the recursive solutions because the optimal solution will never have to backtrack through this square.

By looking at the mazes in Figure 7-1, it is easy to see—at least from your global vantage point—that the submazes labeled (a) and (c) represent dead-end paths and that the only solution begins in the direction shown in the submaze (b). If you are thinking recursively, however, you don't need to carry on the analysis all the way to the solution. You have already decomposed the problem into simpler instances. All you need to do is rely on the power of recursion to solve the individual subproblems, and you're home free. You still have to identify a set of simple cases so that the recursion can terminate, but the hard work has been done.

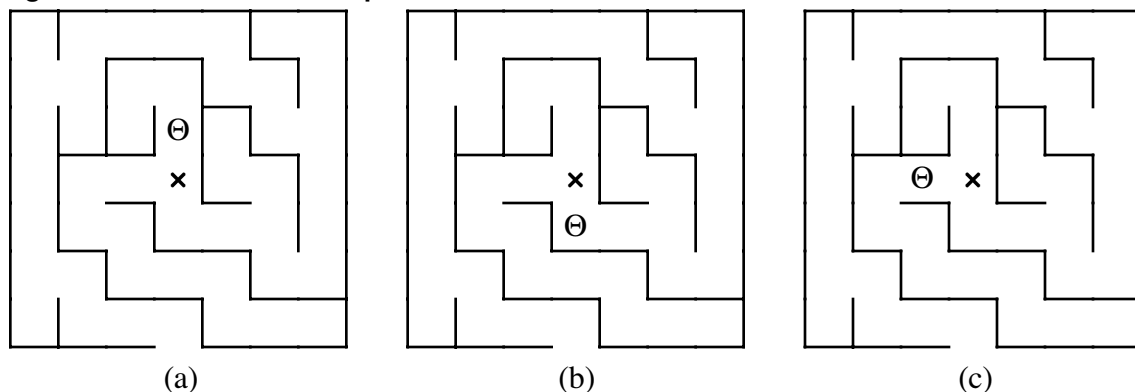
Identifying the simple cases

What constitutes the simple case for a maze? One possibility is that you might already be standing outside the maze. If so, you're finished. Clearly, this situation represents one simple case. There is, however, another possibility. You might also reach a blind alley where you've run out of places to move. For example, if you try to solve the sample maze by moving north and then continue to make recursive calls along that path, you will eventually be in the position of trying to solve the following maze:



At this point, you've run out of room to maneuver. Every path from the new position is either marked or blocked by a wall, which makes it clear that the maze has no solution from this point. Thus, the maze problem has a second simple case in which every direction from the current square is blocked, either by a wall or a marked square.

Figure 7-1 Recursive decomposition of a maze



It is easier to code the recursive algorithm if, instead of checking for marked squares as you consider the possible directions of motion, you go ahead and make the recursive calls on those squares. If you check at the beginning of the procedure to see whether the current square is marked, you can terminate the recursion at that point. After all, if you find yourself positioned on a marked square, you must be retracing your path, which means that the solution must lie in some other direction.

Thus, the two simple cases for this problem are as follows:

1. If the current square is outside the maze, the maze is solved.
2. If the current square is marked, the maze is unsolvable.

Coding the maze solution algorithm

Although the recursive insight and the simple cases are all you need to solve the problem on a conceptual level, writing a complete program to navigate a maze requires you to consider a number of implementation details as well. For example, you need to decide on a representation for the maze itself that allows you, for example, to figure out where the walls are, keep track of the current position, indicate that a particular square is marked, and determine whether you have escaped from the maze. While designing an appropriate data structure for the maze is an interesting programming challenge in its own right, it has very little to do with understanding the recursive algorithm, which is the focus of this discussion. If anything, the details of the data structure are likely to get in the way and make it more difficult for you to understand the algorithmic strategy as a whole.

Fortunately, it is possible to put such details aside by introducing a new abstraction layer. The purpose of the abstraction is to provide the main program with access to the information it needs to solve a maze, even though the details are hidden. An interface that provides the necessary functionality is the `mazelib.h` interface shown in Figure 7-2.

Once you have access to the `mazelib.h` interface, writing a program to solve a maze becomes much simpler. The essence of the problem is to write a function `SolveMaze` that uses recursive backtracking to solve a maze whose specific characteristics are maintained by the `mazelib` module. The argument to `SolveMaze` is the starting position, which changes for each of the recursive subproblems. To ensure that the recursion can terminate when a solution is found, the `SolveMaze` function must also return some indication of whether it has succeeded. The easiest way to convey this information is to define `SolveMaze` as a predicate function that returns `true` if a solution has been found, and `false` otherwise. Thus, the prototype for `SolveMaze` looks like this:

```
bool SolveMaze(pointT pt);
```

Given this definition, the main program is simply

```
int main() {
    ReadMazeMap(MazeFile);
    if (SolveMaze(GetStartPosition())) {
        cout << "The marked squares show a solution path." << endl;
    } else {
        cout << "No solution exists." << endl;
    }
    return 0;
}
```

Figure 7-2 The mazelib.h interface

```

/*
 * File: mazelib.h
 * -----
 * This interface provides a library of primitive operations
 * to simplify the solution to the maze problem.
 */

#ifndef _mazelib_h
#define _mazelib_h

#include "genlib.h"

/*
 * Type: directionT
 * -----
 * This type is used to represent the four compass directions.
 */

enum directionT { North, East, South, West };

/*
 * Type: pointT
 * -----
 * The type pointT is used to encapsulate a pair of integer
 * coordinates into a single value with x and y components.
 */

struct pointT {
    int x, y;
};

/*
 * Function: ReadMazeMap
 * Usage: ReadMazeMap(filename);
 * -----
 * This function reads in a map of the maze from the specified
 * file and stores it in private data structures maintained by
 * this module. In the data file, the characters '+', '-', and
 * '|' represent corners, horizontal walls, and vertical walls,
 * respectively; spaces represent open passageway squares. The
 * starting position is indicated by the character 'S'. For
 * example, the following data file defines a simple maze:
 *
 *      +-+-+-+-+
 *      |      |
 *      + -+ + -+
 *      |S |   |
 *      +-+-+-+-+
 *
 * Coordinates in the maze are numbered starting at (0,0) in
 * the lower left corner. The goal is to find a path from
 * the (0,0) square to the exit east of the (4,1) square.
 */

void ReadMazeMap(string filename);

```



```

/*
 * Function: GetStartPosition
 * Usage: pt = GetStartPosition();
 * -----
 * This function returns a pointT indicating the coordinates of
 * the start square.
 */

pointT GetStartPosition();

/*
 * Function: OutsideMaze
 * Usage: if (OutsideMaze(pt)) . . .
 * -----
 * This function returns true if the specified point is outside
 * the boundary of the maze.
 */

bool OutsideMaze(pointT pt);

/*
 * Function: WallExists
 * Usage: if (WallExists(pt, dir)) . . .
 * -----
 * This function returns true if there is a wall in the indicated
 * direction from the square at position pt.
 */

bool WallExists(pointT pt, directionT dir);

/*
 * Functions: MarkSquare, UnmarkSquare, IsMarked
 * Usage: MarkSquare(pt);
 *         UnmarkSquare(pt);
 *         if (IsMarked(pt)) . . .
 * -----
 * These functions mark, unmark, and test the status of the
 * square specified by the coordinates pt.
 */

void MarkSquare(pointT pt);
void UnmarkSquare(pointT pt);
bool IsMarked(pointT pt);

#endif

```

The code for the **SolveMaze** function itself turns out to be extremely short and is shown in Figure 7-3. The entire algorithm fits into approximately 10 lines of code with the following pseudocode structure:

```

If the current square is outside the maze, return true to indicate that a solution has been found.
If the current square is marked, return false to indicate that this path has already been tried.
Mark the current square.
for (each of the four compass directions) {
    if (this direction is not blocked by a wall) {
        Move one step in the indicated direction from the current square.
        Try to solve the maze from there by making a recursive call.
        If this call shows the maze to be solvable, return true to indicate that fact.
    }
}
Unmark the current square.
Return false to indicate that none of the four directions led to a solution.

```

The only function called by **SolveMaze** that is not exported by the **mazelib.h** interface is the function **AdjacentPoint(pt, dir)**, which returns the coordinates of the square that is one step away from **pt** in the direction **dir**. The following is a simple implementation of **AdjacentPoint** that copies the original point and then adjusts the appropriate coordinate value:

Figure 7-3 The **SolveMaze** function

```

/*
 * Function: SolveMaze
 * Usage: if (SolveMaze(pt)) . . .
 * -----
 * This function attempts to generate a solution to the current
 * maze from point pt. SolveMaze returns true if the maze has
 * a solution and false otherwise. The implementation uses
 * recursion to solve the submazes that result from marking the
 * current square and moving one step along each open passage.
 */

bool SolveMaze(pointT pt) {
    if (OutsideMaze(pt)) return true;
    if (IsMarked(pt)) return false;
    MarkSquare(pt);
    for (int i = 0; i < 4; i++) {
        directionT dir = directionT(i);
        if (!WallExists(pt, dir)) {
            if (SolveMaze(AdjacentPoint(pt, dir))) {
                return true;
            }
        }
    }
    UnmarkSquare(pt);
    return false;
}

```

```

pointT AdjacentPoint(pointT pt, directionT dir) {
    pointT newpt = pt;
    switch (dir) {
        case North: newpt.y++; break;
        case East:  newpt.x++; break;
        case South: newpt.y--; break;
        case West:  newpt.x--; break;;
    }
    return newpt;
}

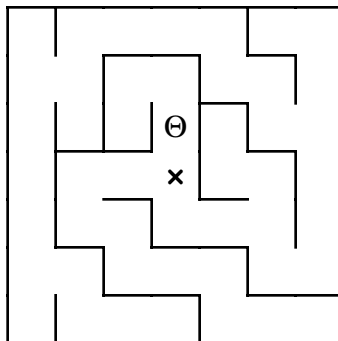
```

The code to unmark the current square at the end of the **for** loop is not strictly necessary in this implementation and in fact can reduce the performance of the algorithm if there are loops in the maze (see exercise 3). The principal advantage of including it is that doing so means that the solution path ends up being recorded by a chain of marked squares from the original starting position to the exit. If you are using a graphical implementation of this algorithm, erasing the marks as you retreat down a path makes it much easier to see the current path.

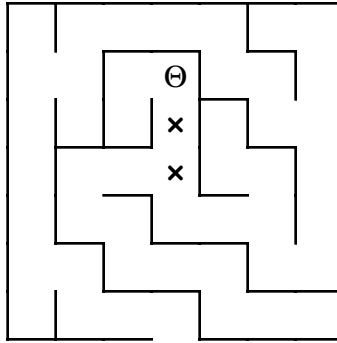
Convincing yourself that the solution works

In order to use recursion effectively, at some point you must be able to look at a recursive function like the **SolveMaze** example in Figure 7-3 and say to yourself something like this: “I understand how this works. The problem is getting simpler because more squares are marked each time. The simple cases are clearly correct. This code must do the job.” For most of you, however, that confidence in the power of recursion will not come easily. Your natural skepticism makes you want to see the steps in the solution. The problem is that, even for a maze as simple as the one shown earlier in this chapter, the complete history of the steps involved in the solution is far too large to think about comfortably. Solving that maze, for example, requires 66 calls to **SolveMaze** that are nested 27 levels deep when the solution is finally discovered. If you attempt to trace the code in detail, you will inevitably get lost.

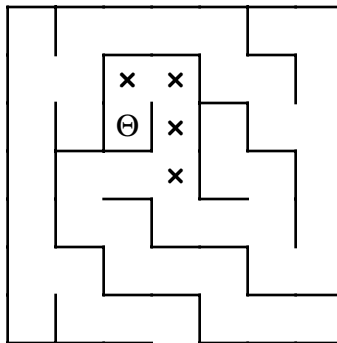
If you are not yet ready to accept the recursive leap of faith, the best you can do is track the operation of the code in a more general sense. You know that the code first tries to solve the maze by moving one square to the north, because the **for** loop goes through the directions in the order defined by the **directionT** enumeration. Thus, the first step in the solution process is to make a recursive call that starts in the following position:



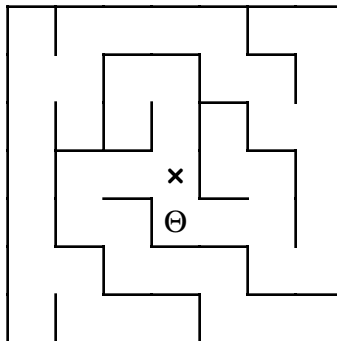
At this point, the same process occurs again. The program again tries to move north and makes a new recursive call in the following position:



At this level of the recursion, moving north is no longer possible, so the **for** loop cycles through the other directions. After a brief excursion southward, upon which the program encounters a marked square, the program finds the opening to the west and proceeds to generate a new recursive call. The same process occurs in this new square, which in turn leads to the following configuration:

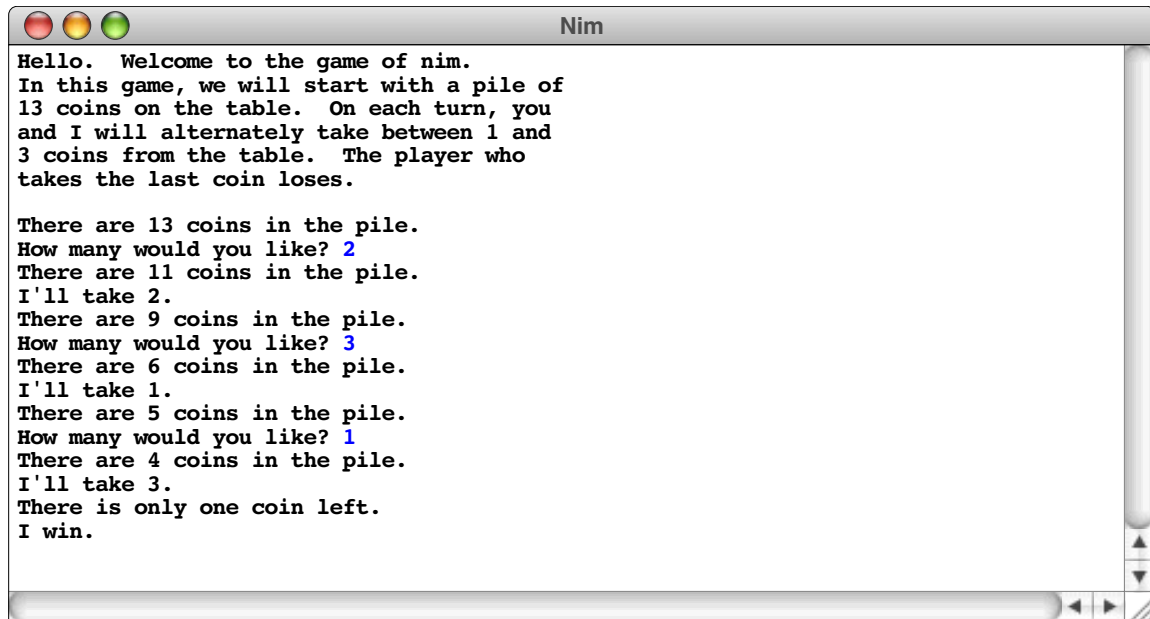


In this position, none of the directions in the **for** loop do any good; every square is either blocked by a wall or already marked. Thus, when the **for** loop at this level exits at the bottom, it unmarks the current square and returns to the previous level. It turns out that all the paths have also been explored in this position, so the program once again unmarks the square and returns to the next higher level in the recursion. Eventually, the program backtracks all the way to the initial call, having completely exhausted the possibilities that begin by moving north. The **for** loop then tries the eastward direction, finds it blocked, and continues on to explore the southern corridor, beginning with a recursive call in the following configuration:



From here on, the same process ensues. The recursion systematically explores every corridor along this path, backing up through the stack of recursive calls whenever it reaches a dead end. The only difference along this route is that eventually—after

Figure 7-4 Sample game of nim



How would you go about writing a program to play a winning game of nim? The mechanical aspects of the game—keeping track of the number of coins, asking the player for a legal move, determining the end of the game, and so forth—are a straightforward programming task. The interesting part of the program consists of figuring out how to give the computer a strategy for playing the best possible game.

Finding a successful strategy for nim is not particularly hard, particularly if you work backward from the end of the game. The rules of nim state that the loser is the player who takes the last coin. Thus, if you ever find yourself with just one coin on the table, you're in a bad position. You have to take that coin and lose. On the other hand, things look good if you find yourself with two, three, or four coins. In any of these cases, you can always take all but one of the remaining coins, leaving your opponent in the unenviable position of being stuck with just one coin. But what if there are five coins on the table? What can you do then? After a bit of thought, it's easy to see that you're also doomed if you're left with five coins. No matter what you do, you have to leave your opponent with two, three, or four coins—situations that you've just discovered represent good positions from your opponent's perspective. If your opponent is playing intelligently, you will surely be left with a single coin on your next turn. Since you have no good moves, being left with five coins is clearly a bad position.

This informal analysis reveals an important insight about the game of nim. On each turn, you are looking for a good move. A move is good if it leaves your opponent in a bad position. But what is a bad position? A bad position is one in which there is no good move. Although these definitions of *good move* and *bad position* are circular, they nonetheless constitute a complete strategy for playing a perfect game of nim. All you have to do is rely on the power of recursion. If you have a function **FindGoodMove** that takes the number of coins as its argument, all it has to do is try every possibility, looking for one that leaves a bad position for the opponent. You can then assign the job of determining whether a particular position is bad to the predicate function **IsBadPosition**, which calls **FindGoodMove** to see if there is one. The two functions call each other back and forth, evaluating all possible branches as the game proceeds.

The **FindGoodMove** function has the following pseudocode formulation:

```

int FindGoodMove(int nCoins) {
    for (each possible move) {
        Evaluate the position that results from making that move.
        If the resulting position is bad, return that move.
    }
    Return a sentinel value indicating that no good move exists.
}

```

The legal values returned by **FindGoodMove** are 1, 2, and 3. The sentinel indicating that no good move exists can be any integer value outside that range. For example, you can define the constant **NO_GOOD_MOVE** as follows:

```

const int NO_GOOD_MOVE = -1;

```

The code for **FindGoodMove** then looks like this:

```

int FindGoodMove(int nCoins) {
    for (int nTaken = 1; nTaken <= MAX_MOVE; nTaken++) {
        if (IsBadPosition(nCoins - nTaken)) return nTaken;
    }
    return NO_GOOD_MOVE;
}

```

The code for the **IsBadPosition** function is even easier. After checking for the simple case that occurs when there is only a single coin to take, the function simply calls **FindGoodMove** to see if a good move exists. The code for **IsBadPosition** is therefore simply

```

bool IsBadPosition(int nCoins) {
    if (nCoins == 1) return true;
    return FindGoodMove(nCoins) == NO_GOOD_MOVE;
}

```

This function encapsulates the following ideas:

- Being left with a single coin indicates a bad position.
- A position is bad if there are no good moves.

The functions **FindGoodMove** and **IsBadPosition** provide all the strategy that the nim program needs to play a perfect game. The rest of the program just takes care of the mechanics of playing nim with a human player, as shown in Figure 7-5.

A generalized program for two-player games

The code for nim shown in Figure 7-5 is highly specific to that game. The **FindGoodMove** function, for example, is written so that it incorporates directly into the structure of the code the knowledge that the computer may take one, two, or three coins. The basic idea, however, is far more general. Many two-player games can be solved using the same overall strategy, even though different games clearly require different codings of the details.

Figure 7-5 Program to play the game of nim

```
/*
 * File: nim.cpp
 * -----
 * This program simulates a simple variant of the game of nim.
 * In this version, the game starts with a pile of 13 coins
 * on a table. Players then take turns removing 1, 2, or 3
 * coins from the pile. The player who takes the last coin
 * loses. This simulation allows a human player to compete
 * against the computer.
 */

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

/*
 * Constants
 * -----
 * N_COINS      -- Initial number of coins
 * MAX_MOVE     -- The maximum number of coins a player may take
 * NO_GOOD_MOVE -- Sentinel indicating no good move is available
 */

const int N_COINS = 13;
const int MAX_MOVE = 3;
const int NO_GOOD_MOVE = -1;

/*
 * Type: playerT
 * -----
 * This enumeration type distinguishes the turns for the human
 * player from those for the computer.
 */

enum playerT { Human, Computer };

/* Private function prototypes */

void GiveInstructions();
void AnnounceWinner(int nCoins, playerT whoseTurn);
int GetUserMove(int nCoins);
bool MoveIsLegal(int nTaken, int nCoins);
int ChooseComputerMove(int nCoins);
int FindGoodMove(int nCoins);
bool IsBadPosition(int nCoins);
```



```
/*
 * Main program
 * -----
 * This program plays the game of nim. In this implementation,
 * the human player always goes first.
 */

int main() {
    int nCoins, nTaken;
    playerT whoseTurn;

    GiveInstructions();
    nCoins = N_COINS;
    whoseTurn = Human;
    while (nCoins > 1) {
        cout << "There are " << nCoins << " coins in the pile."<<endl;
        switch (whoseTurn) {
            case Human:
                nTaken = GetUserMove(nCoins);
                whoseTurn = Computer;
                break;
            case Computer:
                nTaken = ChooseComputerMove(nCoins);
                cout << "I'll take " << nTaken << "." << endl;
                whoseTurn = Human;
                break;
        }
        nCoins -= nTaken;
    }
    AnnounceWinner(nCoins, whoseTurn);
    return 0;
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This function explains the rules of the game to the user.
 */

void GiveInstructions() {
    cout << "Hello. Welcome to the game of nim." << endl;
    cout << "In this game, we will start with a pile of" << endl;
    cout << N_COINS << " coins on the table. " << endl;
    cout << "On each turn, you" << endl;
    cout << "and I will alternately take between 1 and" << endl;
    cout << MAX_MOVE << " coins from the table." << endl;
    cout << "The player who" << endl;
    cout << "takes the last coin loses." << endl;
    cout << endl;
}
```

```

/*
 * Function: AnnounceWinner
 * Usage: AnnounceWinner(nCoins, whoseTurn);
 * -----
 * This function announces the final result of the game.
 */

void AnnounceWinner(int nCoins, playerT whoseTurn) {
    if (nCoins == 0) {
        cout << "You took the last coin.  You lose." << endl;
    } else {
        cout << "There is only one coin left." << endl;
        switch (whoseTurn) {
            case Human:    cout << "I win." << endl; break;
            case Computer: cout << "I lose." << endl; break;
        }
    }
}

/*
 * Function: GetUserMove
 * Usage: nTaken = GetUserMove(nCoins);
 * -----
 * This function is responsible for the human player's turn.
 * It takes the number of coins left in the pile as an argument,
 * and returns the number of coins that the player removes
 * from the pile.  The function checks the move for legality
 * and gives the player repeated chances to enter a legal move.
 */

int GetUserMove(int nCoins) {
    int nTaken, limit;

    while (true) {
        cout << "How many would you like? ";
        nTaken = GetInteger();
        if (MoveIsLegal(nTaken, nCoins)) break;
        limit = (nCoins < MAX_MOVE) ? nCoins : MAX_MOVE;
        cout << "That's cheating!  Please choose a number";
        cout << " between 1 and " << limit << endl;
        cout << "There are " << nCoins << " coins in the pile."<<endl;
    }
    return nTaken;
}

/*
 * Function: MoveIsLegal
 * Usage: if (MoveIsLegal(nTaken, nCoins)) . . .
 * -----
 * This predicate function returns true if it is legal to take
 * nTaken coins from a pile of nCoins.
 */

bool MoveIsLegal(int nTaken, int nCoins) {
    return (nTaken > 0 && nTaken <= MAX_MOVE && nTaken <= nCoins);
}

```

```
/*
 * Function: ChooseComputerMove
 * Usage: nTaken = ChooseComputerMove(nCoins);
 * -----
 * This function figures out what move is best for the computer
 * player and returns the number of coins taken. The function
 * first calls FindGoodMove to see if a winning move exists.
 * If none does, the program takes only one coin to give the
 * human player more chances to make a mistake.
 */

int ChooseComputerMove(int nCoins) {
    int nTaken = FindGoodMove(nCoins);
    if (nTaken == NO_GOOD_MOVE) nTaken = 1;
    return nTaken;
}

/*
 * Function: FindGoodMove
 * Usage: nTaken = FindGoodMove(nCoins);
 * -----
 * This function looks for a winning move, given the specified
 * number of coins. If there is a winning move in that
 * position, the function returns that value; if not, the
 * function returns the constant NoWinningMove. This function
 * depends on the recursive insight that a good move is one
 * that leaves your opponent in a bad position and a bad
 * position is one that offers no good moves.
 */

int FindGoodMove(int nCoins) {
    for (int nTaken = 1; nTaken <= MAX_MOVE; nTaken++) {
        if (IsBadPosition(nCoins - nTaken)) return nTaken;
    }
    return NO_GOOD_MOVE;
}

/*
 * Function: IsBadPosition
 * Usage: if (IsBadPosition(nCoins)) . . .
 * -----
 * This function returns true if nCoins is a bad position.
 * A bad position is one in which there is no good move.
 * Being left with a single coin is clearly a bad position
 * and represents the simple case of the recursion.
 */

bool IsBadPosition(int nCoins) {
    if (nCoins == 1) return true;
    return FindGoodMove(nCoins) == NO_GOOD_MOVE;
}
```

One of the principal ideas in this text is the notion of **abstraction**, which is the process of separating out the general aspects of a problem so that they are no longer obscured by the details of a specific domain. You may not be terribly interested in a program that plays nim; after all, nim is rather boring once you figure it out. What you would probably enjoy more is a program that is general enough to be adapted to play nim, or tic-tac-toe, or any other two-player strategy game you choose.

The first step in creating such a generalization lies in recognizing that there are several concepts that are common to all games. The most important such concept is **state**. For any game, there is some collection of data that defines exactly what is happening at any point in time. In the nim game, for example, the state consists of the number of coins on the table and whose turn it is to move. For a game like chess, the state would instead include what pieces were currently on which squares. Whatever the game, however, it should be possible to combine all the relevant data together into a single record structure and then refer to it using a single variable. Another common concept is that of a **move**. In nim, a move consists of an integer representing the number of coins taken away. In chess, a move might consist of a pair indicating the starting and ending coordinates of the piece that is moving, although this approach is in fact complicated by the need to represent various esoteric moves like castling or the promotion of a pawn. In any case, a move can also be represented by a single structure that includes whatever information is appropriate to that particular game. The process of abstraction consists partly of defining these concepts as general types, with names like **stateT** and **moveT**, that transcend the details of any specific game. The internal structure of these types will be different for different games, but the abstract algorithm can refer to these concepts in a generic form.

Consider, for example, the following main program, which comes from the tic-tac-toe example introduced in Figure 7-6 at the end of this chapter:

```
int main() {
    GiveInstructions();
    stateT state = NewGame();
    moveT move;
    while (!GameIsOver(state)) {
        DisplayGame(state);
        switch (WhoseTurn(state)) {
            case Human:
                move = GetUserMove(state);
                break;
            case Computer:
                move = ChooseComputerMove(state);
                DisplayMove(move);
                break;
        }
        MakeMove(state, move);
    }
    AnnounceResult(state);
    return 0;
}
```

At this level, the program is easy to read. It begins by giving instructions and then calls **NewGame** to initialize a new game, storing the result in the variable **state**. It then goes into a loop, taking turns for each side until the game is over. On the human player's turns, it calls a function **GetUserMove** to read in the appropriate move from the user. On its own turns, the program calls **ChooseComputerMove**, which has the task of finding the best move in a particular state. Once the move has been determined by one of these two functions, the main program then calls **MakeMove**, which updates the state of the game to

show that the indicated move has been made and that it is now the other player's turn. At the end, the program displays the result of the game and exits.

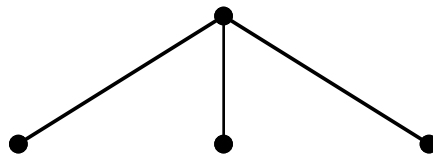
It is important to notice that the main program gives no indication whatsoever about what the actual game is. It could just as easily be nim or chess as tic-tac-toe. Each game requires its own definitions for `stateT`, `moveT`, and the various functions like `GiveInstructions`, `MakeMove`, and `GameIsOver`. Even so, the implementation of the main program as it appears here is general enough to work for many different games.

The minimax strategy

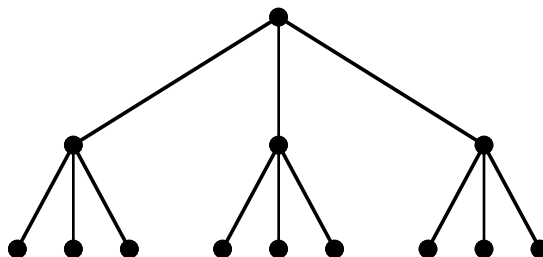
The main program, however, is hardly the most interesting part of a game. The real challenge consists of providing the computer with an effective strategy. In the general program for two-player games, the heart of the computer's strategy is the function `FindBestMove`, which is called by the function `ChooseComputerMove` in the main program. Given a particular state of the game, the role of `FindBestMove` is to return the optimal move in that position.

From the discussion of nim earlier in this chapter, you should already have some sense of what constitutes an optimal move. The best move in any position is simply the one that leaves your opponent in the worst position. The worst position is likewise the one that offers the weakest best move. This idea—finding the position that leaves your opponent with the worst possible best move—is called the **minimax** strategy because the goal is to find the move that minimizes your opponent's maximum opportunity.

The best way to visualize the operation of the minimax strategy is to think about the possible future moves in a game as forming a branching diagram that expands on each turn. Because of their branching character, such diagrams are called **game trees**. The current state is represented by a dot at the top of the game tree. If there are, for example, three possible moves from this position, there will be three lines emanating down from the current state to three new states that represent the results of these moves, as shown in the following diagram:



For each of these new positions, your opponent will also have options. If there are again three options from each of these positions, the next generation of the game tree looks like this:



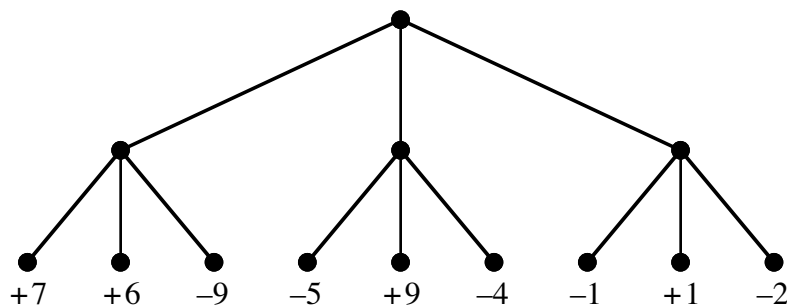
Which move do you choose in the initial position? Clearly, your goal is to achieve the best outcome. Unfortunately, you only get to control half of the game. If you were able to select your opponent's move as well as your own, you could select the path to the state two turns away that left you in the best position. Given the fact that your opponent is also

trying to win, the best thing you can do is choose the initial move that leaves your opponent with as few winning chances as possible.

In order to get a sense of how you should proceed, it helps to add some quantitative data to the analysis. Determine whether a particular move is better than some alternative is much easier if it is possible to assign a numeric score to each possible move. The higher the numeric score, the better the move. Thus, a move that had a score of +7, for example, is better than a move with a rating of -4 . In addition to rating each possible move, it makes sense to assign a similar numeric rating to each position in the game. Thus, one position might have a rating of +9 and would therefore be better than a position with a score of only +2.

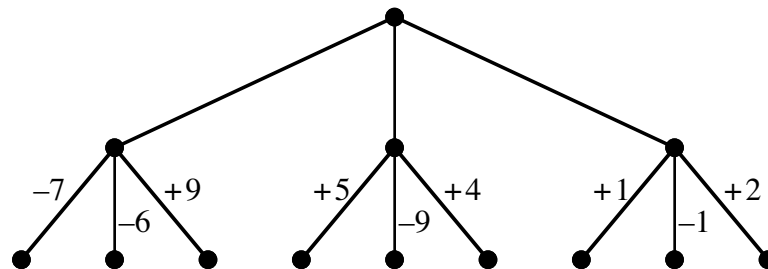
Both positions and moves are rated from the perspective of the player having the move. Moreover, the rating system is designed to be symmetric around 0, in the sense that a position that has a score of +9 for the player to move would have a score of -9 from the opponent's point of view. This interpretation of rating numbers captures the idea that a position that is good for one player is therefore a bad position for the opponent, as you saw in the discussion of the Nim game earlier in this chapter. More importantly, defining the rating system in this way makes it easy to express the relationship between the scores for moves and positions. The score for any move is simply the negative of the score for the resulting position when rated by your opponent. Similarly, the rating of any position can be defined as the rating of its best move.

To make this discussion more concrete, let's consider a simple example. Suppose that you have looked two steps ahead in the game, covering one move by you and the possible responses from your opponent. In computer science, a single move for a single player is called a **ply** to avoid the ambiguity associated with the words *move* and *turn*, which sometimes imply that both players have a chance to play. If you rate the positions at the conclusion of the two-ply analysis, the game tree might look like this:



Because the positions at the bottom of this tree are again positions in which—as at the top of the tree—you have to move, the rating numbers in those positions are assigned from your perspective. Given these ratings of the potential positions, what move should you make from the original configuration? At first glance, you might be attracted by the fact that the leftmost branch has the most positive total score or that the center one contains a path that leads to a +9, which is an excellent outcome for you. Unfortunately, none of these considerations matter much if your opponent is playing rationally. If you choose the leftmost branch, your opponent will surely take the rightmost option from there, which leaves you with a -9 position. The same thing happens if you choose the center branch; your opponent finds the worst possible position, which has a rating of -5 . The best you can do is choose the rightmost branch, which only allows your opponent to end up with a -2 rating. While this position is hardly ideal, it is better for you than the other outcomes.

The situation becomes easier to follow if you add the ratings for each of your opponent's responses at the second level of the tree. The rating for a move—from the perspective of the player making it—is the negative of the resulting position. Thus, the move ratings from your opponent's point of view look like this:



In these positions, your opponent will seek to play the move with the best score. By choosing the rightmost path, you minimize the maximum score available to your opponent, which is the essence of the *minimax* strategy.

Implementing the minimax algorithm

The minimax algorithm is quite general and can be implemented in a way that does not depend on the specific characteristics of the game. In many respects, the implementation of the minimax algorithm is similar to the strategy section in `nim.cpp` because it consists of two mutually recursive functions, one that finds the best move and another that evaluates the quality of a position. If you want to make your algorithm as general as possible, however, you must modify the approach used in the `nim` program to accommodate the following extensions:

- *It must be possible to limit the depth of the recursive search.* For games that involve any significant level of complexity, it is impossible to search the entire game tree in a reasonable amount of time. If you try to apply this approach to chess, for example, a program running on the fastest computers available would require many times the lifetime of the universe to make the first move. As a result, a practical implementation of the minimax algorithm must include a provision for cutting off the search at a certain point. One possible approach is to limit the depth of the recursion to a certain number of moves. You could, for example, allow the recursion to proceed until each player had made three moves and then evaluate the position at that point using some nonrecursive approach.
- *It must be possible to assign ratings to moves and positions.* Every position in `nim` is either good or bad; there are no other options. In a more complex game, it is necessary—particularly if you can't perform a complete analysis—to assign ratings to positions and moves so that the algorithm has a standard for comparing them against other possibilities. The rating scheme used in this implementation assigns integers to positions and moves. Those integers extend in both the positive and the negative direction and are centered on zero, which means that a rating of `-5` for one player is equivalent to a rating of `+5` from the opponent's point of view. Zero is therefore the neutral rating and is given the name `NeutralPosition`. The maximum positive rating is the constant `WinningPosition`, which indicates a position in which the player whose turn it is to move will invariably win; the corresponding extreme in the negative direction is `LosingPosition`, which indicates that the player will always lose.

Taking these general considerations into account requires some changes in the design of the mutually recursive functions that implement the minimax algorithm, which are called `FindBestMove` and `EvaluatePosition`. Both functions take the state of the game

as an argument, but each also requires the current depth of the recursion so that the recursive search can be restricted if necessary. Moreover, in order to avoid a considerable amount of redundant calculation, it is extremely useful if **FindBestMove** can return a rating along with the best move, so it uses a reference parameter along with the return value to return the two pieces of information.. Given these design decisions, the prototypes for **FindBestMove** and **EvaluatePosition** look like this:

```
moveT FindBestMove(stateT state, int depth, int & rating);
int EvaluatePosition(stateT state, int depth);
```

The strategy for **FindBestMove** can be expressed using the following pseudocode:

```
moveT FindBestMove(stateT state, int depth, int & rating) {
    for (each possible move or until you find a forced win) {
        Make the move.
        Evaluate the resulting position, adding one to the depth indicator.
        Keep track of the minimum rating so far, along with the corresponding move.
        Retract the move to restore the original state.
    }
    Store the move rating into the reference parameter.
    Return the best move.
}
```

The corresponding implementation, which follows this pseudocode outline, looks like this:

```
moveT FindBestMove(stateT state, int depth, int & rating) {
    Vector<moveT> moveList;
    GenerateMoveList(state, moveList);
    int nMoves = moveList.size();
    if (nMoves == 0) Error("No moves available");
    moveT bestMove;
    int minRating = WINNING_POSITION + 1;
    for (int i = 0; i < nMoves && minRating != LOSING_POSITION; i++) {
        moveT move = moveList[i];
        MakeMove(state, move);
        int curRating = EvaluatePosition(state, depth + 1);
        if (curRating < minRating) {
            bestMove = move;
            minRating = curRating;
        }
        RetractMove(state, move);
    }
    rating = -minRating;
    return bestMove;
}
```

The function **GenerateMoveList(state, moveArray)** is implemented separately for each game and has the effect of filling the elements in **moveArray** with a list of the legal moves in the current position; the result of the function is the number of available moves. The only other parts of this function that require some comment are the line

```
minRating = WinningPosition + 1;
```

which initializes the value of **minRating** to a number large enough to guarantee that this value will be replaced on the first cycle through the **for** loop, and the line


```
rating = -minRating;
```

which stores the rating of the best move in the reference parameter. The negative sign is included because the perspective has shifted: the positions were evaluated from the point-of-view of your opponent, whereas the ratings express the value of a move from your own point of view. A move that leaves your opponent with a negative position is good for you and therefore has a positive value.

The **EvaluatePosition** function is considerably simpler. The simple cases that allow the recursion to terminate occur when the game is over or when the maximum allowed recursive depth has been achieved. In these cases, the program must evaluate the current state as it exists without recourse to further recursion. This evaluation is performed by the function **EvaluateStaticPosition**, which is coded separately for each game. In the general case, however, the rating of a position is simply the rating of the best move available, given the current state. Thus, the following code is sufficient to the task:

```
int EvaluatePosition(stateT state, int depth) {
    int rating;
    if (GameIsOver(state) || depth >= MAX_DEPTH) {
        return EvaluateStaticPosition(state);
    }
    FindBestMove(state, depth, rating);
    return rating;
}
```

Using the general strategy to solve a specific game

The minimax strategy embodied in the functions **FindBestMove** and **EvaluatePosition** takes care of the conceptually complicated work of finding the best move from a given position. Moreover, because it is written in an abstract way, the code does not depend on the details of a particular game. Once you have these functions, the task of coding a new two-player strategy game is reduced to the problem of designing **moveT** and **stateT** structures that are appropriate to the game and then writing code for the functions that must be supplied independently for each game.

The long program in Figure 7-6 illustrates how to use the general minimax facility to construct a program that plays tic-tac-toe. The implementations of the main program and the functions **FindBestMove** and **EvaluatePosition** are completely independent of the details of tic-tac-toe, even though they are responsible for calculating most of the strategy. What makes this program play tic-tac-toe is the definition of the types and functions that surround the basic framework.

The types **moveT** and **stateT** are defined so that they are appropriate for tic-tac-toe. If you number the squares in the board like this:

1	2	3
4	5	6
7	8	9

a move can be represented as a single integer, which means that the appropriate definition for the type **moveT** is simply

```
typedef int moveT;
```

Figure 7-6 Program to play the game of tic-tac-toe

```

/*
 * File: tictactoe.cpp
 * -----
 * This program plays a game of tic-tac-toe with the user. The
 * code is designed to make it easy to adapt the general structure
 * to other games.
 */

#include "genlib.h"
#include "simpio.h"
#include "vector.h"
#include "grid.h"
#include <iostream>

/*
 * Constants: WINNING_POSITION, NEUTRAL_POSITION, LOSING_POSITION
 * -----
 * These constants define a rating system for game positions. A
 * rating is an integer centered at 0 as the neutral score: ratings
 * greater than 0 are good for the current player, ratings less than
 * 0 are good for the opponent. The constants WINNING_POSITION and
 * LOSING_POSITION are opposite in value and indicate a position that
 * is a forced win or loss, respectively. In a game in which the
 * analysis is complete, no intermediate values ever arise. If the
 * full tree is too large to analyze, the EvaluatePosition function
 * returns integers that fall between the two extremes.
 */

const int WINNING_POSITION = 1000;
const int NEUTRAL_POSITION = 0;
const int LOSING_POSITION = -WINNING_POSITION;

/*
 * Type: playerT
 * -----
 * This type is used to distinguish the human and computer
 * players and keep track of who has the current turn.
 */

enum playerT { Human, Computer };

/*
 * Type: moveT
 * -----
 * For any particular game, the moveT type must keep track of the
 * information necessary to describe a move. For tic-tac-toe,
 * a moveT is simply an integer identifying the number of one of
 * the nine squares, as follows:
 *
 *      1 | 2 | 3
 *      ---+---+---
 *      4 | 5 | 6
 *      ---+---+---
 *      7 | 8 | 9
 */

typedef int moveT;

```

```

/*
 * Type: stateT
 * -----
 * For any game, the stateT structure records the current state of
 * the game. As in Chapter 4, the tic-tac-toe board is represented
 * using a Grid<char>; the elements must be either 'X', 'O', or ' '.
 * In addition to the board array, the code stores a playerT value
 * to indicate whose turn it is. In this game, the stateT structure
 * also contains the total number of moves so that functions can
 * find this value without counting the number of occupied squares.
 */

struct stateT {
    Grid<char> board;
    playerT whoseTurn;
    int turnsTaken;
};

/*
 * Constant: MAX_DEPTH
 * -----
 * This constant indicates the maximum depth to which the recursive
 * search for the best move is allowed to proceed.
 */

const int MAX_DEPTH = 10000;

/*
 * Constant: FIRST_PLAYER
 * -----
 * This constant indicates which player goes first.
 */

const playerT FIRST_PLAYER = Computer;

/* Private function prototypes */

void GiveInstructions();
moveT FindBestMove(stateT state, int depth, int & pRating);
int EvaluatePosition(stateT state, int depth);
stateT NewGame();
void DisplayGame(stateT state);
void DisplayMove(moveT move);
char PlayerMark(playerT player);
moveT GetUserMove(stateT state);
moveT ChooseComputerMove(stateT state);
void GenerateMoveList(stateT state, Vector<moveT> & moveList);
bool MoveIsLegal(moveT move, stateT state);
void MakeMove(stateT & state, moveT move);
void RetractMove(stateT & state, moveT move);
bool GameIsOver(stateT state);
void AnnounceResult(stateT state);
playerT WhoseTurn(stateT state);
playerT Opponent(playerT player);
int EvaluateStaticPosition(stateT state);
bool CheckForWin(stateT state, playerT player);
bool CheckForWin(Grid<char> & board, char mark);
bool CheckLine(Grid<char> & board, char mark,
               int row, int col, int dRow, int dCol);

```

```

/*
 * Main program
 * -----
 * The main program, along with the functions FindBestMove and
 * EvaluatePosition, are general in their design and can be
 * used with most two-player games. The specific details of
 * tic-tac-toe do not appear in these functions and are instead
 * encapsulated in the stateT and moveT data structures and a
 * variety of subsidiary functions.
 */

int main() {
    GiveInstructions();
    stateT state = NewGame();
    moveT move;
    while (!GameIsOver(state)) {
        DisplayGame(state);
        switch (WhoseTurn(state)) {
            case Human:
                move = GetUserMove(state);
                break;
            case Computer:
                move = ChooseComputerMove(state);
                DisplayMove(move);
                break;
        }
        MakeMove(state, move);
    }
    AnnounceResult(state);
    return 0;
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * -----
 * This function gives the player instructions about how to
 * play the game.
 */

void GiveInstructions() {
    cout << "Welcome to tic-tac-toe. The object of the game" << endl;
    cout << "is to line up three symbols in a row," << endl;
    cout << "vertically, horizontally, or diagonally." << endl;
    cout << "You'll be " << PlayerMark(Human) << " and I'll be "
        << PlayerMark(Computer) << "." << endl;
}

```

```
/*
 * Function: NewGame
 * Usage: state = NewGame();
 * -----
 * This function starts a new game and returns a stateT that
 * has been initialized to the defined starting configuration.
 */

stateT NewGame() {
    stateT state;
    state.board.resize(3, 3);
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            state.board[i][j] = ' ';
        }
    }
    state.whoseTurn = FIRST_PLAYER;
    state.turnsTaken = 0;
    return state;
}

/*
 * Function: DisplayGame
 * Usage: DisplayGame(state);
 * -----
 * This function displays the current state of the game.
 */

void DisplayGame(stateT state) {
    if (GameIsOver(state)) {
        cout << "The final position looks like this:" << endl << endl;
    } else {
        cout << "The game now looks like this:" << endl << endl;
    }
    for (int i = 0; i < 3; i++) {
        if (i != 0) cout << "----+----+----" << endl;
        for (int j = 0; j < 3; j++) {
            if (j != 0) cout << "|";
            cout << " " << state.board[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

/*
 * Function: DisplayMove
 * Usage: DisplayMove(move);
 * -----
 * This function displays the computer's move.
 */

void DisplayMove(moveT move) {
    cout << "I'll move to " << move << endl;
}
```

```

/*
 * Function: FindBestMove
 * Usage: move = FindBestMove(state, depth, pRating);
 * -----
 * This function finds the best move for the current player, given
 * the specified state of the game. The depth parameter and the
 * constant MAX_DEPTH are used to limit the depth of the search
 * for games that are too difficult to analyze in full detail.
 * The function returns the best move by storing an integer in
 * the variable to which pRating points.
 */

moveT FindBestMove(stateT state, int depth, int & rating) {
    Vector<moveT> moveList;
    GenerateMoveList(state, moveList);
    int nMoves = moveList.size();
    if (nMoves == 0) Error("No moves available");
    moveT bestMove;
    int minRating = WINNING_POSITION + 1;
    for (int i = 0; i < nMoves && minRating != LOSING_POSITION; i++) {
        moveT move = moveList[i];
        MakeMove(state, move);
        int curRating = EvaluatePosition(state, depth + 1);
        if (curRating < minRating) {
            bestMove = move;
            minRating = curRating;
        }
        RetractMove(state, move);
    }
    rating = -minRating;
    return bestMove;
}

/*
 * Function: EvaluatePosition
 * Usage: rating = EvaluatePosition(state, depth);
 * -----
 * This function evaluates a position by finding the rating of
 * the best move in that position. The depth parameter and the
 * constant MAX_DEPTH are used to limit the depth of the search.
 */

int EvaluatePosition(stateT state, int depth) {
    int rating;
    if (GameIsOver(state) || depth >= MAX_DEPTH) {
        return EvaluateStaticPosition(state);
    }
    FindBestMove(state, depth, rating);
    return rating;
}

```

```
/*
 * Function: NewGame
 * Usage: state = NewGame();
 * -----
 * This function starts a new game and returns a stateT that
 * has been initialized to the defined starting configuration.
 */

stateT NewGame() {
    stateT state;

    for (int i = 1; i <= 9; i++) {
        state.board[i] = ' ';
    }
    state.whoseTurn = FIRST_PLAYER;
    state.turnsTaken = 0;
    return state;
}

/*
 * Function: DisplayGame
 * Usage: DisplayGame(state);
 * -----
 * This function displays the current state of the game.
 */

void DisplayGame(stateT state) {
    if (GameIsOver(state)) {
        cout << "The final position looks like this:" << endl << endl;
    } else {
        cout << "The game now looks like this:" << endl << endl;
    }
    for (int row = 0; row < 3; row++) {
        if (row != 0) cout << "----+----+----" << endl;
        for (int col = 0; col < 3; col++) {
            if (col != 0) cout << "|";
            cout << " " << state.board[row * 3 + col + 1] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

/*
 * Function: DisplayMove
 * Usage: DisplayMove(move);
 * -----
 * This function displays the computer's move.
 */

void DisplayMove(moveT move) {
    cout << "I'll move to " << move << endl;
}
```

```
/*
 * Function: PlayerMark
 * Usage: mark = PlayerMark(player);
 * -----
 * This function returns the mark used on the board to indicate
 * the specified player. By convention, the first player is
 * always X, so the mark used for each player depends on who
 * goes first.
 */

char PlayerMark(playerT player) {
    if (player == FIRST_PLAYER) {
        return 'X';
    } else {
        return 'O';
    }
}

/*
 * Function: GetUserMove
 * Usage: move = GetUserMove(state);
 * -----
 * This function allows the user to enter a move and returns the
 * number of the chosen square. If the user specifies an illegal
 * move, this function gives the user the opportunity to enter
 * a legal one.
 */

moveT GetUserMove(stateT state) {
    cout << "Your move." << endl;
    moveT move;
    while (true) {
        cout << "What square? ";
        move = GetInteger();
        if (MoveIsLegal(move, state)) break;
        cout << "That move is illegal. Try again." << endl;
    }
    return move;
}

/*
 * Function: ChooseComputerMove
 * Usage: move = ChooseComputerMove(state);
 * -----
 * This function chooses the computer's move and is primarily
 * a wrapper for FindBestMove. This function also makes it
 * possible to display any game-specific messages that need
 * to appear at the beginning of the computer's turn. The
 * rating value returned by FindBestMove is simply discarded.
 */

moveT ChooseComputerMove(stateT state) {
    int rating;
    cout << "My move." << endl;
    return FindBestMove(state, 0, rating);
}
```



```

/*
 * Function: GenerateMoveList
 * Usage: GenerateMoveList(state, moveList);
 * -----
 * This function generates a list of the legal moves available in
 * the specified state. The list of moves is returned in the
 * vector moveList, which must be allocated by the client.
 */

void GenerateMoveList(stateT state, Vector<moveT> & moveList) {
    for (int i = 1; i <= 9; i++) {
        moveT move = moveT(i);
        if (MoveIsLegal(move, state)) {
            moveList.add(moveT(i));
        }
    }
}

/*
 * Function: MoveIsLegal
 * Usage: if (MoveIsLegal(move, state)) . . .
 * -----
 * This function returns true if the specified move is legal.
 */

bool MoveIsLegal(moveT move, stateT state) {
    if (move < 1 || move > 9) return false;
    int row = (move - 1) / 3;
    int col = (move - 1) % 3;
    return state.board[row][col] == ' ';
}

/*
 * Function: MakeMove
 * Usage: MakeMove(state, move);
 * -----
 * This function changes the state by making the indicated move.
 */

void MakeMove(stateT & state, moveT move) {
    int row = (move - 1) / 3;
    int col = (move - 1) % 3;
    state.board[row][col] = PlayerMark(state.whoseTurn);
    state.whoseTurn = Opponent(state.whoseTurn);
    state.turnsTaken++;
}

```

```
/*
 * Function: RetractMove
 * Usage: RetractMove(state, move);
 * -----
 * This function changes the state by "unmaking" the indicated move.
 */

void RetractMove(stateT & state, moveT move) {
    int row = (move - 1) / 3;
    int col = (move - 1) % 3;
    state.board[row][col] = ' ';
    state.whoseTurn = Opponent(state.whoseTurn);
    state.turnsTaken--;
}

/*
 * Function: GameIsOver
 * Usage: if (GameIsOver(state)) . . .
 * -----
 * This function returns true if the game is complete.
 */

bool GameIsOver(stateT state) {
    return (state.turnsTaken == 9
           || CheckForWin(state, state.whoseTurn)
           || CheckForWin(state, Opponent(state.whoseTurn)));
}

/*
 * Function: AnnounceResult
 * Usage: AnnounceResult(state);
 * -----
 * This function announces the result of the game.
 */

void AnnounceResult(stateT state) {
    DisplayGame(state);
    if (CheckForWin(state, Human)) {
        cout << "You win." << endl;
    } else if (CheckForWin(state, Computer)) {
        cout << "I win." << endl;
    } else {
        cout << "Cat's game." << endl;
    }
}
```

```
/*
 * Function: WhoseTurn
 * Usage: player = WhoseTurn(state);
 * -----
 * This function returns whose turn it is, given the current
 * state of the game. The reason for making this a separate
 * function is to ensure that the common parts of the code do
 * not need to refer to the internal components of the state.
 */

playerT WhoseTurn(stateT state) {
    return state.whoseTurn;
}

/*
 * Function: Opponent
 * Usage: opp = Opponent(player);
 * -----
 * This function returns the playerT value corresponding to the
 * opponent of the specified player.
 */

playerT Opponent(playerT player) {
    return (player == Human) ? Computer : Human;
}

/*
 * Function: EvaluateStaticPosition
 * Usage: rating = EvaluateStaticPosition(state);
 * -----
 * This function gives the rating of a position without looking
 * ahead any further in the game tree. Although this function
 * duplicates much of the computation of GameIsOver and therefore
 * introduces some runtime inefficiency, it makes the algorithm
 * somewhat easier to follow.
 */

int EvaluateStaticPosition(stateT state) {
    if (CheckForWin(state, state.whoseTurn)) {
        return WINNING_POSITION;
    }
    if (CheckForWin(state, Opponent(state.whoseTurn))) {
        return LOSING_POSITION;
    }
    return NEUTRAL_POSITION;
}
```

```

/*
 * Function: CheckForWin
 * Usage: if (CheckForWin(state, player)) . . .
 * -----
 * This function returns true if the specified player has won
 * the game. The check on turnsTaken increases efficiency,
 * because neither player can win the game until the fifth move.
 */

bool CheckForWin(stateT state, playerT player) {
    if (state.turnsTaken < 5) return false;
    return CheckForWin(state.board, PlayerMark(player));
}

/*
 * 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;
}

```

The state of the tic-tac-toe game is determined by the symbols on the nine squares of the board and whose turn it is to move. Thus, you can represent the essential components of the state as follows:

```
struct stateT {
    char board[(3 * 3) + 1];
    playerT whoseTurn;
};
```

The board is represented as a single-dimensional array to correspond with the numbering of the squares from 1 to 9; the fact that position 0 is unused makes it necessary to add an extra element to the array. As the complete program shows, adding a **turnsTaken** component to the state makes it much easier to determine whether the game is complete.

The rules of tic-tac-toe are expressed in the implementation of functions like **GenerateMoveList**, **MoveIsLegal**, and **EvaluateStaticPosition**, each of which is coded specifically for this game. Even though these functions require some thought, they tend to be less conceptually complex than those that implement the minimax strategy. By coding the difficult functions once and then reusing them in various different game programs, you can avoid having to go through the same logic again and again.

Summary

In this chapter, you have learned to solve problems that require making a sequence of choices as you search for a goal, as illustrated by finding a path through a maze or a winning strategy in a two-player game. The basic strategy is to write programs that can backtrack to previous decision points if those choices lead to dead ends. By exploiting the power of recursion, however, you can avoid coding the details of the backtracking process explicitly and develop general solution strategies that apply to a wide variety of problem domains.

Important points in this chapter include:

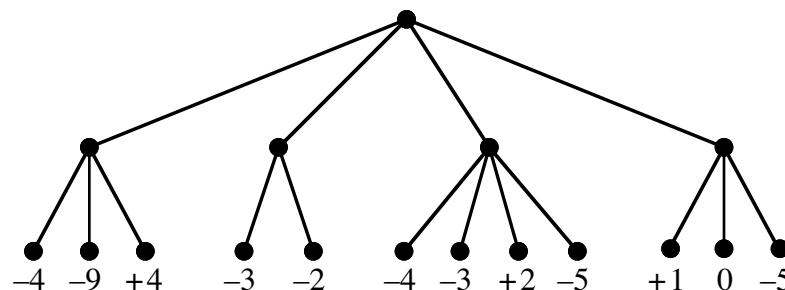
- You can solve most problems that require backtracking by adopting the following recursive approach:

```
If you are already at a solution, report success.
for (every possible choice in the current position ) {
    Make that choice and take one step along the path.
    Use recursion to solve the problem from the new position.
    If the recursive call succeeds, report the success to the next higher level.
    Back out of the current choice to restore the state at the beginning of the loop.
}
Report failure.
```

- The complete history of recursive calls in a backtracking problem—even for relatively simple applications—is usually too complex to understand in detail. For problems that involve any significant amount of backtracking, it is essential to accept the recursive leap of faith.
- You can often find a winning strategy for two-player games by adopting a recursive-backtracking approach. Because the goal in such games involves minimizing the winning chances for your opponent, the conventional strategic approach is called the *minimax algorithm*.
- It is possible to code the minimax algorithm in a general way that keeps the details of any specific game separate from the implementation of the minimax strategy itself. This approach makes it easier to adapt an existing program to new games.

Review questions

1. What is the principal characteristic of a backtracking algorithm?
2. Using your own words, state the right-hand rule for escaping from a maze. Would a left-hand rule work equally well?
3. What is the insight that makes it possible to solve a maze by recursive backtracking?
4. What are the simple cases that apply in the recursive implementation of **SolveMaze**?
5. Why is important to mark squares as you proceed through the maze? What would happen in the **SolveMaze** function if you never marked any squares?
6. What is the purpose of the **UnmarkSquare** call at the end of the **for** loop in the **SolveMaze** implementation? Is this statement essential to the algorithm?
7. What is the purpose of the Boolean result returned by **SolveMaze**?
8. In your own words, explain how the backtracking process actually takes place in the recursive implementation of **SolveMaze**.
9. In the simple nim game, the human player plays first and begins with a pile of 13 coins. Is this a good or a bad position? Why?
10. Write a simple C++ expression based on the value of **nCoins** that has the value **true** if the position is good for the current player and **false** otherwise. (Hint: Use the **%** operator.)
11. What is the minimax algorithm? What does its name signify?
12. Suppose you are in a position in which the analysis for the next two moves shows the following rated outcomes from your opponent's point-of-view:



If you adopt the minimax strategy, what is the best move to make in this position? What is the rating of that move from your perspective?

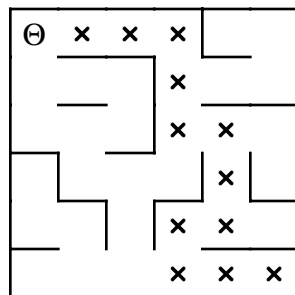
13. Why is it useful to develop an abstract implementation of the minimax algorithm?
14. What two data structures are used to make the minimax implementation independent of the specific characteristics of a particular game?
15. What is the role of each of the three arguments to the **FindBestMove** function?
16. Explain the role of the **EvaluateStaticPosition** function in the minimax implementation.

Programming exercises

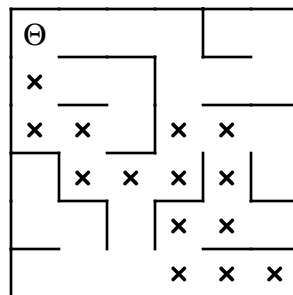
1. The **SolveMaze** function shown in Figure 7-3 implements a recursive algorithm for solving a maze but is not complete as it stands. The solution depends on several functions exported by the **mazelib.h** interface, which is specified in Figure 7-2 but never actually implemented.

Write a file **mazelib.cpp** that implements this interface. Your implementation should store the data representing the maze as part of the private state of the module. This design requires you to declare static global variables within the implementation that are appropriate to the data you need to implement the operations. If you design the data structure well, most of the individual function definitions in the interface are quite simple. The hard parts of this exercise are designing an appropriate internal representation and implementing the function **ReadMapFile**, which initializes the internal structures from a data file formatted as described in the interface documentation.

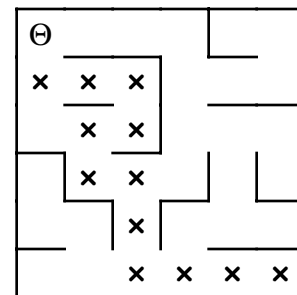
2. In many mazes, there are multiple paths. For example, the diagrams below show three solutions for the same maze:



length = 13

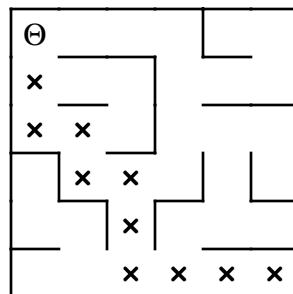


length = 15



length = 13

None of these solutions, however, is optimal. The shortest path through the maze has a path length of 11:



Write a function

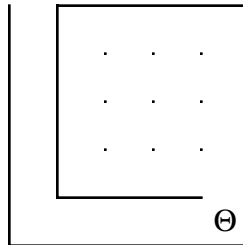
```
int ShortestPathLength(pointT pt);
```

that returns the length of the shortest path in the maze from the specified position to any exit. If there is no solution to the maze, **ShortestPathLength** should return the constant **NO_SOLUTION**, which is defined to have a value larger than the maximum permissible path length, as follows:

```
const int NO_SOLUTION = 10000;
```

3. As implemented in Figure 7-3, the **SolveMaze** function unmarks each square as it discovers there are no solutions from that point. Although this design strategy has the advantage that the final configuration of the maze shows the solution path as a series of marked squares, the decision to unmark squares as you backtrack has a cost in terms of the overall efficiency of the algorithm. If you've marked a square and then backtracked through it, you've already explored the possibilities leading from that square. If you come back to it by some other path, you might as well rely on your earlier analysis instead of exploring the same options again.

To give yourself a sense of how much these unmarking operations cost in terms of efficiency, extend the **SolveMaze** program so that it records the number of recursive calls as it proceeds. Use this program to calculate how many recursive calls are required to solve the following maze if the call to **UnmarkSquare** remains part of the program:



Run your program again, this time without the call to **UnmarkSquare**. What happens to the number of recursive calls?

4. As the result of the preceding exercise makes clear, the idea of keeping track of the path through a maze by using the **MarkSquare** facility in **mazelib.h** has a substantial cost. A more practical approach is to change the definition of the recursive function so that it keeps track of the current path as it goes. Following the logic of **SolveMaze**, write a function

```
bool FindPath(pointT pt, Vector<pointT> path);
```

that takes, in addition to the coordinates of the starting position, a vector of **pointT** values called **path**. Like **SolveMaze**, **FindPath** returns a Boolean value indicating whether the maze is solvable. In addition, **FindPath** initializes the elements of the **path** vector to a sequence of coordinates beginning with the starting position and ending with the coordinates of the first square that lies outside the maze. For example, if you use **FindPath** with the following main program

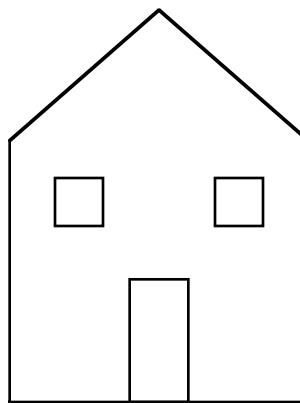
```
int main() {
    ReadMazeMap(MazeFile);
    Vector<pointT> path;
    if (FindPath(GetStartPosition(), path)) {
        cout << "The following path is a solution:" << endl;
        for (int i = 0; i < path.size(); i++) {
            cout << "(" << path[i].x << ", "
                << path[i].y << ")" << endl;
        }
    } else {
        cout << "No solution exists." << endl;
    }
    return 0;
}
```

it will display the coordinates of the points in the solution path on the screen.

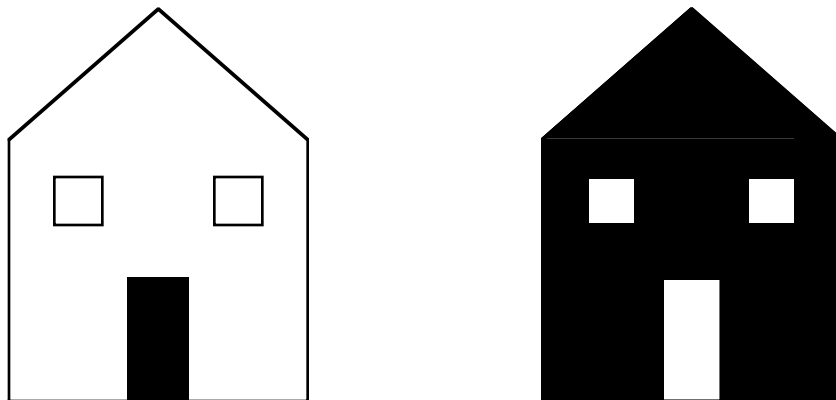
For this exercise, it is sufficient for **FindPath** to find any solution path. It need not find the shortest one.

5. If you have access to the graphics library described in Chapter 6, extend the **maze.cpp** program so that, in addition to keeping track of the internal data structures for the maze, it also displays a diagram of the maze on the screen and shows the final solution path.
6. Most drawing programs for personal computers make it possible to fill an enclosed region on the screen with a solid color. Typically, you invoke this operation by selecting a paint-bucket tool and then clicking the mouse, with the cursor somewhere in your drawing. When you do, the paint spreads to every part of the picture it can reach without going through a line.

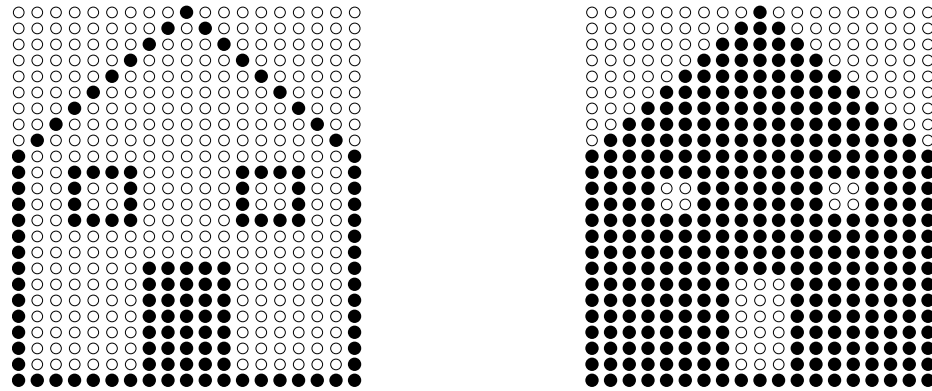
For example, suppose you have just drawn the following picture of a house:



If you select the paint bucket and click inside the door, the drawing program fills the area bounded by the door frame as shown at the left side of the following diagram. If you instead click somewhere on the front wall of the house, the program fills the entire wall space except for the windows and doors, as shown on the right:



In order to understand how this process works, it is important to understand that the screen of the computer is actually broken down into an array of tiny dots called **pixels**. On a monochrome display, pixels can be either white or black. The paint-fill operation consists of painting black the starting pixel (i.e., the pixel you click while using the paint-bucket tool) along with any pixels connected to that starting point by an unbroken chain of white pixels. Thus, the patterns of pixels on the screen representing the preceding two diagrams would look like this:



Write a program that simulates the operation of the paint-bucket tool. To simplify the problem, assume that you have access to the enumerated type

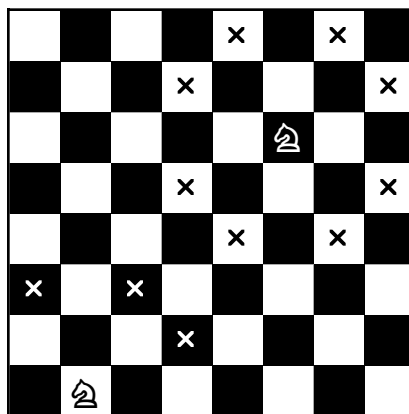
```
enum pixelStateT { WHITE, BLACK };
```

and the following functions:

```
pixelStateT GetPixelState(pointT pt);
void SetPixelState(pointT pt, pixelStateT state);
bool OutsidePixelBounds(pointT pt);
```

The first function is used to return the state of any pixel, given its coordinates in the pixel array. The second allows you to change the state of any pixel to a new value. The third makes it possible to determine whether a particular coordinate is outside the pixel array altogether, so that the recursion can stop at the edges of the screen.

7. In chess, a knight moves in an L-shaped pattern: two squares in one direction horizontally or vertically, and then one square at right angles to that motion. For example, the knight in the upper right side of the following diagram can move to any of the eight squares marked with a black cross:



The mobility of a knight decreases near the edge of the board, as illustrated by the bottom knight, which can reach only the three squares marked by white crosses.

It turns out that a knight can visit all 64 squares on a chessboard without ever moving to the same square twice. A path for the knight that moves through all the squares without repeating a square is called a **knight's tour**. One such tour is shown in the following diagram, in which the numbers in the squares indicate the order in which they were visited:

52	47	56	45	54	5	22	13
57	44	53	4	23	14	25	6
48	51	46	55	26	21	12	15
43	58	3	50	41	24	7	20
36	49	42	27	62	11	16	29
59	2	37	40	33	28	19	8
38	35	32	61	10	63	30	17
1	60	39	34	31	18	9	64

Write a program that uses backtracking recursion to find a knight's tour.

8. The most powerful piece in the game of chess is the queen, which can move any number of squares in any direction, horizontally, vertically, or diagonally. For example, the queen shown in this chessboard can move to any of the marked squares:

×					×		
	×				×		
		×			×		
			×		×		×
				×	×	×	
×	×	×	×	×	♙	×	×
				×	×	×	
			×		×		×

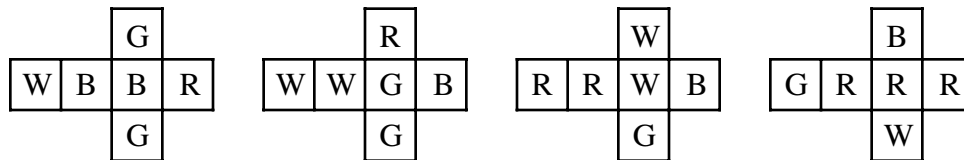
Even though the queen can cover a large number of squares, it is possible to place eight queens on a 8x8 chessboard so that none of them attacks any of the others, as shown in the following diagram:

	♙						
					♙		
							♙
			♙				
♙							

Write a program that solves the more general problem of whether it is possible to place N queens on an $N \times N$ chessboard so that none of them can move to a square occupied by any of the others in a single turn. Your program should either display a solution if it finds one or report that no solutions exist.

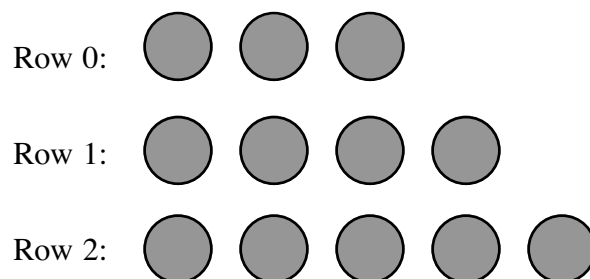
9. In the 1960s, a puzzle called *Instant Insanity* was popular for some years before it faded from view. The puzzle consisted of four cubes whose faces were each painted with one of the colors red, blue, green, and white, represented in the rest of this problem by their initial letter. The goal of the puzzle was to arrange the cubes into a line so that if you looked at the line from any of its edges, you would see no duplicated colors.

Cubes are hard to draw in two dimensions, but the following diagram shows what the cubes would look like if you unfolded them and placed them flat on the page:



Write a program that uses backtracking to solve the Instant Insanity puzzle.

10. Rewrite the simple nim game from Figure 7-5 so that it uses the more general structure developed for tic-tac-toe. Your new program should not change the implementations of the functions **main**, **FindBestMove**, and **EvaluatePosition**, which should remain exactly as they appear in Figure 7-6. Your job is to come up with appropriate definitions of **stateT**, **moveT**, and the various game-specific functions in the implementation so that the program plays nim instead of tic-tac-toe.
11. Modify the code for the simple nim game you wrote for exercise 10 so that it plays a different variant of nim. In this version, the pile begins with 17 coins. On each turn, players alternate taking one, two, three, or four coins from the pile. In the simple nim game, the coins the players took away were simply ignored; in this game, the coins go into a pile for each player. The player whose pile contains an even number of coins after the last one is taken wins the game.
12. In the most common variant of nim, the coins are not combined into a single pile but are instead arranged in three rows like this:



A move in this game consists of taking any number of coins, subject to the condition that all the coins must come from the same row. The player who takes the last coin loses.

Write a program that uses the minimax algorithm to play a perfect game of three-pile nim. The starting configuration shown here is a typical one, but your program should be general enough so that you can easily change the number of coins in each row.

Chapter 8

Algorithmic Analysis

Without analysis, no synthesis.

— Friedrich Engels, *Herr Eugen Dühring's
Revolution in Science*, 1878

In Chapter 5, you were introduced to two different recursive implementations of the function **Fib(n)**, which computes the n^{th} Fibonacci number. The first is based directly on the mathematical definition

$$\mathbf{Fib}(n) = \begin{cases} n & \text{if } n \text{ is } 0 \text{ or } 1 \\ \mathbf{Fib}(n-1) + \mathbf{Fib}(n-2) & \text{otherwise} \end{cases}$$

and turns out to be wildly inefficient. The second implementation, which uses the notion of additive sequences to produce a version of **Fib(n)** that is comparable in efficiency to traditional iterative approaches, demonstrates that recursion is not really the root of the problem. Even so, examples like the simple recursive implementation of the Fibonacci function have such high execution costs that recursion sometimes gets a bad name as a result.

As you will see in this chapter, the ability to think recursively about a problem often leads to new strategies that are considerably *more* efficient than anything that would come out of an iterative design process. The power of recursive divide-and-conquer algorithms is enormous and has a profound impact on many problems that arise in practice. By using recursive algorithms of this form, it is possible to achieve dramatic increases in efficiency that can cut the solution times, not by factors of two or three, but by factors of a thousand or more.

Before looking at these algorithms, however, it is important to ask a few questions. What does the term *efficiency* mean in an algorithmic context? How would you go about measuring that efficiency? These questions form the foundation for the subfield of computer science known as **analysis of algorithms**. Although a detailed understanding of algorithmic analysis requires a reasonable facility with mathematics and a lot of careful thought, you can get a sense of how it works by investigating the performance of a few simple algorithms.

8.1 The sorting problem

The easiest way to illustrate the analysis of algorithms is to consider a problem domain in which different algorithms vary widely in their performance. Of these, one of the most interesting is the problem of **sorting**, which consists of reordering the elements in an array or vector so that they fall in some defined sequence. For example, suppose you have stored the following integers in the variable **vec**, which is a **Vector<int>**:

vec							
56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

Your mission is to write a function **Sort(vec)** that rearranges the elements into ascending order, like this:

vec							
19	25	30	37	56	58	73	95
0	1	2	3	4	5	6	7

The selection sort algorithm

There are many algorithms you could choose to sort an vector of integers into ascending order. One of the simplest is called **selection sort**. Given a vector of size N , the selection sort algorithm goes through each element position and finds the value which should occupy that position in the sorted vector. When it finds the appropriate element, the algorithm exchanges it with the value which previously occupied the desired position to ensure that no elements are lost. Thus, on the first cycle, the algorithm finds the smallest element and swaps it with the first vector position. On the second cycle, it finds the smallest remaining element and swaps it with the second position. Thereafter, the algorithm continues this strategy until all positions in the vector are correctly ordered. An implementation of **Sort** that uses selection sort is shown in Figure 8-1.

For example, if the initial contents of the vector are

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

the first cycle through the outer **for** loop identifies the 19 in index position 5 as the smallest value in the entire vector and then swaps it with the 56 in index position 0 to leave the following configuration:

Figure 8-1 Implementation of the selection sort algorithm

```

/*
 * Function: Sort
 * -----
 * This implementation uses an algorithm called selection sort,
 * which can be described in English as follows. With your left
 * hand (lh), point at each element in the vector in turn,
 * starting at index 0. At each step in the cycle:
 *
 * 1. Find the smallest element in the range between your left
 *    hand and the end of the vector, and point at that element
 *    with your right hand (rh).
 *
 * 2. Move that element into its correct position by exchanging
 *    the elements indicated by your left and right hands.
 */

void Sort(Vector<int> & vec) {
    int n = vec.size();
    for (int lh = 0; lh < n; lh++) {
        int rh = lh;
        for (int i = lh + 1; i < n; i++) {
            if (vec[i] < vec[rh]) rh = i;
        }
        int temp = vec[lh];
        vec[lh] = vec[rh];
        vec[rh] = temp;
    }
}

```

19	25	37	58	95	56	73	30
0	1	2	3	4	5	6	7

On the second cycle, the algorithm finds the smallest element between positions 1 and 7, which turns out to be the 25 in position 1. The program goes ahead and performs the exchange operation, leaving the vector unchanged from the preceding diagram. On each subsequent cycle, the algorithm performs a swap operation to move the next smallest value into its appropriate final position. When the **for** loop is complete, the entire vector is sorted.

Empirical measurements of performance

How efficient is the selection sort algorithm as a strategy for sorting? To answer this question, it helps to collect empirical data about how long it takes the computer to sort a vector of various sizes. When I did this experiment some years ago on a computer that is significantly slower than machines available today, I observed the following timing data for selection sort, where N represents the number of elements in the vector:

N	Running time
10	0.12 msec
20	0.39 msec
40	1.46 msec
100	8.72 msec
200	33.33 msec
400	135.42 msec
1000	841.67 msec
2000	3.35 sec
4000	13.42 sec
10,000	83.90 sec

For an vector of 10 integers, the selection sort algorithm completes its work in a fraction of a millisecond. Even for 1000 integers, this implementation of **Sort** takes less than a second, which certainly seems fast enough in terms of our human sense of time. As the vector sizes get larger, however, the performance of selection sort begins to go downhill. For an vector of 10,000 integers, the algorithm requires over a minute of computing time. If you're sitting in front of your computer waiting for it to reply, a minute seems like an awfully long time.

Even more disturbing is the fact that the performance of selection sort rapidly gets worse as the vector size increases. As you can see from the timing data, every time you double the number of values in the vector, the running time increases by about a factor of four. Similarly, if you multiply the number of values by 10, the time required to sort the vector goes up a hundredfold. If this pattern continues, sorting a list of 100,000 numbers would take two and a half hours. Sorting a vector of a million numbers by this method would take approximately 10 days. Thus, if your business required you to solve sorting problems on this scale, you would have no choice but to find a more efficient approach.

Analyzing the performance of selection sort

What makes selection sort perform so badly as the number of values to be sorted becomes large? To answer this question, it helps to think about what the algorithm has to do on each cycle of the outer loop. To correctly determine the first value in the vector, the selection sort algorithm must consider all N elements as it searches for the smallest value.

Thus, the time required on the first cycle of the loop is presumably proportional to N . For each of the other elements in the vector, the algorithm performs the same basic steps but looks at one fewer element each time. It looks at $N-1$ elements on the second cycle, $N-2$ on the third, and so on, so the total running time is roughly proportional to

$$N + N-1 + N-2 + \dots + 3 + 2 + 1$$

Because it is difficult to work with an expression in this expanded form, it is useful to simplify it by applying a bit of mathematics. As you may have learned in an algebra course, the sum of the first N integers is given by the formula

$$\frac{N(N+1)}{2}$$

or, multiplying out the numerator,

$$\frac{N^2 + N}{2}$$

You will learn how to prove that this formula is correct in the section on “Mathematical induction” later in this chapter. For the moment, all you need to know is that the sum of the first N integers can be expressed in this more compact form.

If you write out the values of the function

$$\frac{N^2 + N}{2}$$

for various values of N , you get a table that looks like this:

N	$\frac{N^2 + N}{2}$
10	55
20	210
40	820
100	5050
200	20,100
400	80,200
1000	500,500
2000	2,001,000
4000	8,002,000
10,000	50,005,000

Because the running time of the selection sort algorithm is presumably related to the amount of work the algorithm needs to do, the values in this table should be roughly proportional to the observed execution time of the algorithm, which turns out to be true. If you look at the measured timing data for selection sort, for example, you discover that the algorithm requires 83.90 seconds to sort 10,000 numbers. In that time, the selection sort algorithm has to perform 50,005,000 operations in its innermost loop. Assuming that there is indeed a proportionality relationship between these two values, dividing the time by the number of operations gives the following estimate of the proportionality constant:

$$\frac{83.90 \text{ seconds}}{50,005,000} = 0.00167 \text{ msec}$$

If you then apply this same proportionality constant to the other entries in the table, you discover that the formula

$$0.00167 \text{ msec} \times \frac{N^2 + N}{2}$$

is indeed a good approximation of the running time, at least for large values of N . The observed times and the estimates calculated using this formula are shown in Table 8-1, along with the relative error between the two.

8.2 Computational complexity and big-O notation

The problem with carrying out a detailed analysis like the one shown in Table 8-1 is that you end up with too much information. Although it is occasionally useful to have a formula for predicting exactly how long a program will take, you can usually get away with more qualitative measures. The reason that selection sort is impractical for large values of N has little to do with the precise timing characteristics of a particular implementation running on a specific machine. The problem is much simpler and more fundamental. At its essence, the problem with selection sort is that doubling the size of the input vector increases the running time of the selection sort algorithm by a factor of four, which means that the running time grows more quickly than the number of elements in the vector.

The most valuable qualitative insights you can obtain about algorithmic efficiency are usually those that help you understand how the performance of an algorithm responds to changes in problem size. Problem size is usually easy to quantify. For algorithms that operate on numbers, it generally makes sense to let the numbers themselves represent the problem size. For most algorithms that operate on arrays or vectors, you can use the number of elements. When evaluating algorithmic efficiency, computer scientists traditionally use the letter N to indicate the size of the problem, no matter how it is calculated. The relationship between N and the performance of an algorithm as N becomes large is called the **computational complexity** of that algorithm. In general, the most important measure of performance is execution time, although it also possible to apply complexity analysis to other concerns, such as the amount of memory space required. Unless otherwise stated, all assessments of complexity used in this text refer to execution time.

Table 8-1 Observed and estimated times for selection sort

N	Observed time	Estimated time	Error
10	0.12 msec	0.09 msec	23%
20	0.39 msec	0.35 msec	10%
40	1.46 msec	1.37 msec	6%
100	8.72 msec	8.43 msec	3%
200	33.33 msec	33.57 msec	1%
400	135.42 msec	133.93 msec	1%
1000	841.67 msec	835.84 msec	1%
2000	3.35 sec	3.34 sec	< 1%
4000	13.42 sec	13.36 sec	< 1%
10,000	83.90 sec	83.50 sec	< 1%

Big-O notation

Computer scientists use a special notation to denote the computational complexity of algorithms. Called **big-O notation**, it was introduced by the German mathematician Paul Bachmann in 1892—long before the development of computers. The notation itself is very simple and consists of the letter *O*, followed by a formula enclosed in parentheses. When it is used to specify computational complexity, the formula is usually a simple function involving the problem size *N*. For example, in this chapter you will soon encounter the big-O expression

$$O(N^2)$$

which is read aloud as “big-oh of *N* squared.”

Big-O notation is used to specify qualitative approximations and is therefore ideal for expressing the computational complexity of an algorithm. Coming as it does from mathematics, big-O notation has a precise definition, which appears later in this chapter in the section entitled “A formal definition of big-O.” At this point, however, it is far more important for you—no matter whether you think of yourself as a programmer or a computer scientist—to understand what big-O means from a more intuitive point of view.

Standard simplifications of big-O

When you use big-O notation to estimate the computational complexity of an algorithm, the goal is to provide a *qualitative* insight as to how changes in *N* affect the algorithmic performance as *N* becomes large. Because big-O notation is not intended to be a quantitative measure, it is not only appropriate but desirable to reduce the formula inside the parentheses so that it captures the qualitative behavior of the algorithm in the simplest possible form. The most common simplifications that you can make when using big-O notation are as follows:

1. *Eliminate any term whose contribution to the total ceases to be significant as *N* becomes large.* When a formula involves several terms added together, one of those terms often grows much faster than the others and ends up dominating the entire expression as *N* becomes large. For large values of *N*, this term alone will control the running time of the algorithm, and you can ignore the other terms in the formula entirely.
2. *Eliminate any constant factors.* When you calculate computational complexity, your main concern is how running time changes as a function of the problem size *N*. Constant factors have no effect on the overall pattern. If you bought a machine that was twice as fast as your old one, any algorithm that you executed on your machine would run twice as fast as before for every value of *N*. The growth pattern, however, would remain exactly the same. Thus, you can ignore constant factors when you use big-O notation.

The computational complexity of selection sort

You can apply the simplification rules from the preceding section to derive a big-O expression for the computational complexity of selection sort. From the analysis in the section “Analyzing the performance of selection sort” earlier in the chapter, you know that the running time of the selection sort algorithm for a vector of *N* elements is proportional to

$$\frac{N^2 + N}{2}$$

Although it would be mathematically correct to use this formula directly in the big-O expression

$$O\left(\frac{N^2 + N}{2}\right)$$



This expression is too complicated.

you would never do so in practice because the formula inside the parentheses is not expressed in the simplest form.

The first step toward simplifying this relationship is to recognize that the formula is actually the sum of two terms, as follows:

$$\frac{N^2}{2} + \frac{N}{2}$$

You then need to consider the contribution of each of these terms to the total formula as N increases in size, which is illustrated by the following table:

N	$\frac{N^2}{2}$	$\frac{N}{2}$	$\frac{N^2 + N}{2}$
10	50	5	55
100	5000	50	5050
1000	500,000	500	500,500
10,000	50,000,000	5000	50,005,000
100,000	5,000,000,000	50,000	5,000,050,000

As N increases, the term involving N^2 quickly dominates the term involving N . As a result, the simplification rule allows you to eliminate the smaller term from the expression. Even so, you would not write that the computational complexity of selection sort is

$$O\left(\frac{N^2}{2}\right)$$



This expression includes a constant factor.

because you can eliminate the constant factor. The simplest expression you can use to indicate the complexity of selection sort is

$$O(N^2)$$

This expression captures the essence of the performance of selection sort. As the size of the problem increases, the running time tends to grow by the square of that increase. Thus, if you double the size of the vector, the running time goes up by a factor of four. If you instead multiply the number of input values by 10, the running time will explode by a factor of 100.

Predicting computational complexity from code structure

How would you determine the computational complexity of the function

```
double Average(Vector<double> & vec) {
    int n = vec.size();
    double total = 0;
    for (int i = 0; i < n; i++) {
        total += vec[i];
    }
    return total / n;
}
```

which computes the average of the elements in a vector? When you call this function, some parts of the code are executed only once, such as the initialization of **total** to 0 and the division operation in the **return** statement. These computations take a certain amount of time, but that time is constant in the sense that it doesn't depend on the size of the vector. Code whose execution time does not depend on the problem size is said to run in **constant time**, which is expressed in big-O notation as $O(1)$.¹

There are, however, other parts of the **Average** function that are executed exactly **n** times, once for each cycle of the **for** loop. These components include the expression **i++** in the **for** loop and the statement

```
total += vec[i];
```

which constitutes the loop body. Although any single execution of this part of the computation takes a fixed amount of time, the fact that these statements are executed **n** times means their total execution time is directly proportional to the vector size. The computational complexity of this part of the **Average** function is $O(N)$, which is commonly called **linear time**.

The total running time for **Average** is therefore the sum of the times required for the constant parts and the linear parts of the algorithm. As the size of the problem increases, however, the constant term becomes less and less relevant. By exploiting the simplification rule that allows you to ignore terms that become insignificant as N gets large, you can assert that the **Average** function as a whole runs in $O(N)$ time.

You could, however, predict this result just by looking at the loop structure of the code. For the most part, the individual expressions and statements—unless they involve function calls that must be accounted separately—run in constant time. What matters in terms of computational complexity is how often those statements are executed. For many programs, you can determine the computational complexity simply by finding the piece of the code that is executed most often and determining how many times it runs as a function of N . In the case of the **Average** function, the body of the loop is executed **n** times. Because no part of the code is executed more often than this, you can predict that the computational complexity will be $O(N)$.

The selection sort function can be analyzed in a similar way. The most frequently executed part of the code is the comparison in the statement

```
if (vec[i] < vec[rh]) rh = i;
```

¹ Some students find the designation $O(1)$ confusing, because the expression inside the parentheses does not depend on N . In fact, this lack of any dependency on N is the whole point of the $O(1)$ notation. As you increase the size of the problem, the time required to execute code whose running time is $O(1)$ increases in exactly the same way that 1 increases; in other words, the running time of the code does not increase at all.

That statement is nested inside two **for** loops whose limits depend on the value of N . The inner loop runs N times as often as the outer loop, which implies that the inner loop body is executed $O(N^2)$ times. Algorithms like selection sort that exhibit $O(N^2)$ performance are said to run in **quadratic time**.

Worst-case versus average-case complexity

In some cases, the running time of an algorithm depends not only on the size of the problem but also on the specific characteristics of the data. For example, consider the function

```
int LinearSearch(int key, Vector<int> & vec) {
    int n = vec.size();
    for (int i = 0; i < n; i++) {
        if (key == vec[i]) return i;
    }
    return -1;
}
```

which returns the first index position in **vec** at which the value **key** appears, or -1 if the value **key** does not appear anywhere in the vector. Because the **for** loop in the implementation executes **n** times, you expect the performance of **LinearSearch**—as its name implies—to be $O(N)$.

On the other hand, some calls to **LinearSearch** can be executed very quickly. Suppose, for example, that the key element you are searching for happens to be in the first position in the vector. In that case, the body of the **for** loop will run only once. If you're lucky enough to search for a value that always occurs at the beginning of the vector, **LinearSearch** will run in constant time.

When you analyze the computational complexity of a program, you're usually not interested in the minimum possible time. In general, computer scientists tend to be concerned about the following two types of complexity analysis:

- *Worst-case complexity.* The most common type of complexity analysis consists of determining the performance of an algorithm in the worst possible case. Such an analysis is useful because it allows you to set an upper bound on the computational complexity. If you analyze for the worst case, you can guarantee that the performance of the algorithm will be at least as good as your analysis indicates. You might sometimes get lucky, but you can be confident that the performance will not get any worse.
- *Average-case complexity.* From a practical point of view, it is often useful to consider how well an algorithm performs if you average its behavior over all possible sets of input data. Particularly if you have no reason to assume that the specific input to your problem is in any way atypical, the average-case analysis provides the best statistical estimate of actual performance. The problem, however, is that average-case analysis is usually much more difficult to carry out and typically requires considerable mathematical sophistication.

The worst case for the **LinearSearch** function occurs when the key is not in the vector at all. When the key is not there, the function must complete all **n** cycles of the **for** loop, which means that its performance is $O(N)$. If the key is known to be in the vector, the **for** loop will be executed about half as many times on average, which implies that average-case performance is also $O(N)$. As you will discover in the section on “The Quicksort algorithm” later in this chapter, the average-case and worst-case performances

of an algorithm sometimes differ in qualitative ways, which means that in practice it is often important to take both performance characteristics into consideration.

A formal definition of big-O

Because understanding big-O notation is critical to modern computer science, it is important to offer a somewhat more formal definition to help you understand why the intuitive model of big-O works and why the suggested simplifications of big-O formulas are in fact justified. In mathematics, big-O notation is used to express the relationship between two functions, in an expression like this:

$$t(N) = O(f(N))$$

The formal meaning of this expression is that $f(N)$ is an approximation of $t(N)$ with the following characteristic: it must be possible to find a constant N_0 and a positive constant C so that for every value of $N \geq N_0$, the following condition holds:

$$t(N) \leq C \times f(N)$$

In other words, as long as N is “large enough,” the function $t(N)$ is always bounded by a constant multiple of the function $f(N)$.

When it is used to express computational complexity, the function $t(N)$ represents the actual running time of the algorithm, which is usually difficult to compute. The function $f(N)$ is a much simpler formula that nonetheless provides a reasonable qualitative estimate for how the running time changes as a function of N , because the condition expressed in the mathematical definition of big-O ensures that the actual running time cannot grow faster than $f(N)$.

To see how the formal definition applies, it is useful to go back to the selection sorting example. Analyzing the loop structure of selection sort showed that the operations in the innermost loop were executed

$$\frac{N^2 + N}{2}$$

times and that the running time was presumably roughly proportional to this formula. When this complexity was expressed in terms of big-O notation, the constants and low-order terms were eliminated, leaving only the assertion that the execution time was $O(N^2)$, which is in fact an assertion that

$$\frac{N^2 + N}{2} = O(N^2)$$

To show that this expression is indeed true under the formal definition of big-O, all you need to do is find constants C and N_0 so that

$$\frac{N^2 + N}{2} \leq C \times N^2$$

for all values of $N \geq N_0$. This particular example is extremely simple. All you need to do to satisfy the constraints is to set the constants C and N_0 both to 1. After all, as long as N is no smaller than 1, you know that $N^2 \geq N$. It must therefore be the case that

$$\frac{N^2 + N}{2} \leq \frac{N^2 + N^2}{2}$$

But the right side of this inequality is simply N^2 , which means that

$$\frac{N^2 + N}{2} \leq N^2$$

for all values of $N \geq 1$, as required by the definition.

You can use a similar argument to show that any polynomial of degree k , which can be expressed in general terms as

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \dots + a_2 N^2 + a_1 N + a_0$$

is $O(N^k)$. To do so, your goal is to find constants C and N_0 so that

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \dots + a_2 N^2 + a_1 N + a_0 \leq C \times N^k$$

for all values of $N \geq N_0$.

As in the preceding example, start by letting N_0 be 1. For all values of $N \geq 1$, each successive power of N is at least as large as its predecessor, so

$$N^k \geq N^{k-1} \geq N^{k-2} \geq \dots \geq N^2 \geq N \geq 1$$

This property in turn implies that

$$\begin{aligned} & a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \dots + a_2 N^2 + a_1 N + a_0 \\ & \leq |a_k| N^k + |a_{k-1}| N^k + |a_{k-2}| N^k + \dots + |a_2| N^k + |a_1| N^k + |a_0| N^k \end{aligned}$$

where the vertical bars surrounding the coefficients on the right side of the equation indicate absolute value. By factoring out N^k , you can simplify the right side of this inequality to

$$(|a_k| + |a_{k-1}| + |a_{k-2}| + \dots + |a_2| + |a_1| + |a_0|) N^k$$

Thus, if you define the constant C to be

$$|a_k| + |a_{k-1}| + |a_{k-2}| + \dots + |a_2| + |a_1| + |a_0|$$

you have established that

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \dots + a_2 N^2 + a_1 N + a_0 \leq C \times N^k$$

This result proves that the entire polynomial is $O(N^k)$.

If all this mathematics scares you, try not to worry. It is much more important for you to understand what big-O means in practice than it is to follow all the steps in the formal derivation.

8.3 Recursion to the rescue

At this point, you know considerably more about complexity analysis than you did when you started the chapter. However, you are no closer to solving the practical problem of how to write a sorting algorithm that is more efficient for large vectors. The selection sort algorithm is clearly not up to the task, because the running time increases in proportion to the square of the input size. The same is true for most sorting algorithms

that process the elements of the vector in a linear order. To develop a better sorting algorithm, you need to adopt a qualitatively different approach.

The power of divide-and-conquer strategies

Oddly enough, the key to finding a better sorting strategy lies in recognizing that the quadratic behavior of algorithms like selection sort has a hidden virtue. The basic characteristic of quadratic complexity is that, as the size of a problem doubles, the running time increases by a factor of four. The reverse, however, is also true. If you divide the size of a quadratic problem by two, you decrease the running time by that same factor of four. This fact suggests that dividing a vector in half and then applying a recursive divide-and-conquer approach might reduce the required sorting time.

To make this idea more concrete, suppose you have a large vector that you need to sort. What happens if you divide the vector into two halves and then use the selection sort algorithm to sort each of those pieces? Because selection sort is quadratic, each of the smaller vectors requires one quarter of the original time. You need to sort both halves, of course, but the total time required to sort the two smaller vectors is still only half the time that would have been required to sort the original vector. If it turns out that sorting two halves of an vector simplifies the problem of sorting the complete vector, you will be able to reduce the total time substantially. More importantly, once you discover how to improve performance at one level, you can use the same algorithm recursively to sort each half.

To determine whether a divide-and-conquer strategy is applicable to the sorting problem, you need to answer the question of whether dividing a vector into two smaller vectors and then sorting each one helps to solve the general problem. As a way to gain some insight into this question, suppose that you start with a vector containing the following eight elements:

vec							
56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

If you divide the vector of eight elements into two vectors of length four and then sort each of those smaller vectors—remember that the recursive leap of faith means you can assume that the recursive calls work correctly—you get the following situation in which each of the smaller vectors is sorted:

v1			
25	37	56	58
0	1	2	3

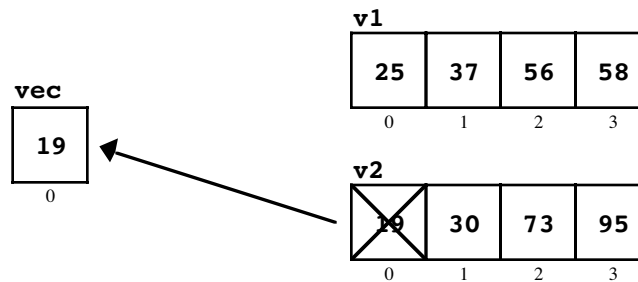
v2			
19	30	73	95
0	1	2	3

How useful is this decomposition? Remember that your goal is to take the values out of these smaller vectors and put them back into the original vector in the correct order. How does having these smaller sorted vectors help you in accomplishing that goal?

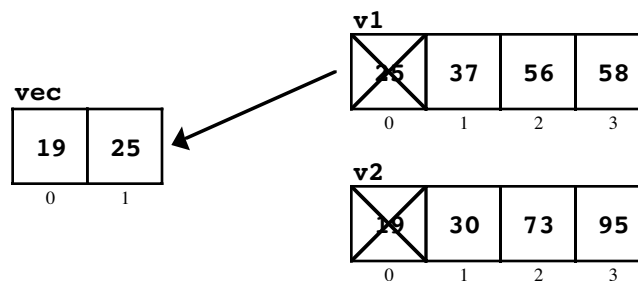
Merging two vectors

As it happens, reconstructing the complete vector from the smaller sorted vectors is a much simpler problem than sorting itself. The required technique, called **merging**,

depends on the fact that the first element in the complete ordering must be either the first element in **v1** or the first element in **v2**, whichever is smaller. In this example, the first element you want in the new vector is the 19 in **v2**. If you add that element to an empty vector **vec** and, in effect, cross it out of **v2**, you get the following configuration:



Once again, the next element can only be the first unused element in one of the two smaller vectors. You compare the 25 from **v1** against the 30 in **v2** and choose the former:



You can easily continue this process of choosing the smaller value from **v1** or **v2** until you have reconstructed the entire vector.

The merge sort algorithm

The merge operation, combined with recursive decomposition, gives rise to a new sorting algorithm called **merge sort**, which you can implement in a straightforward way. The basic idea of the algorithm can be outlined as follows:

1. Check to see if the vector is empty or has only one element. If so, it must already be sorted, and the function can return without doing any work. This condition defines the simple case for the recursion.
2. Divide the vector into two smaller vectors, each of which is half the size of the original.
3. Sort each of the smaller vectors recursively.
4. Clear the original vector so that it is again empty.
5. Merge the two sorted vectors back into the original one.

The code for the merge sort algorithm, shown in Figure 8-2, divides neatly into two functions: **Sort** and **Merge**. The code for **Sort** follows directly from the outline of the algorithm. After checking for the special case, the algorithm divides the original vector into two smaller ones, **v1** and **v2**. As soon as the code for **Sort** has copied all of the elements into either **v1** or **v2**, have been created, the rest of the function sorts these vectors recursively, clears the original vector, and then calls **Merge** to reassemble the complete solution.

Figure 8-2 Implementation of the merge sort algorithm

```
/*
 * Function: Sort
 * -----
 * This function sorts the elements of the vector into
 * increasing numerical order using the merge sort algorithm,
 * which consists of the following steps:
 *
 * 1. Divide the vector into two halves.
 * 2. Sort each of these smaller vectors recursively.
 * 3. Merge the two vectors back into the original one.
 */

void Sort(Vector<int> & vec) {
    int n = vec.size();
    if (n <= 1) return;
    Vector<int> v1;
    Vector<int> v2;
    for (int i = 0; i < n; i++) {
        if (i < n / 2) {
            v1.add(vec[i]);
        } else {
            v2.add(vec[i]);
        }
    }
    Sort(v1);
    Sort(v2);
    vec.clear();
    Merge(vec, v1, v2);
}

/*
 * Function: Merge
 * -----
 * This function merges two sorted vectors (v1 and v2) into the
 * vector vec, which should be empty before this operation.
 * Because the input vectors are sorted, the implementation can
 * always select the first unused element in one of the input
 * vectors to fill the next position.
 */

void Merge(Vector<int> & vec, Vector<int> & v1, Vector<int> & v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    int p1 = 0;
    int p2 = 0;
    while (p1 < n1 && p2 < n2) {
        if (v1[p1] < v2[p2]) {
            vec.add(v1[p1++]);
        } else {
            vec.add(v2[p2++]);
        }
    }
    while (p1 < n1) vec.add(v1[p1++]);
    while (p2 < n2) vec.add(v2[p2++]);
}
```

Most of the work is done by the **Merge** function, which takes the destination vector, along with the smaller vectors **v1** and **v2**. The indices **p1** and **p2** mark the progress through each of the subsidiary vectors. On each cycle of the loop, the function selects an element from **v1** or **v2**—whichever is smaller—and adds that value to the end of **vec**. As soon as the elements in either of the two smaller vector are exhausted, the function can simply copy the elements from the other vector without bothering to test them. In fact, because one of these vectors is already exhausted when the first **while** loop exits, the function can simply copy the rest of each vector to the destination. One of these vectors will be empty, and the corresponding **while** loop will therefore not be executed at all.

The computational complexity of merge sort

You now have an implementation of the **Sort** function based on the strategy of divide-and-conquer. How efficient is it? You can measure its efficiency by sorting vectors of numbers and timing the result, but it is helpful to start by thinking about the algorithm in terms of its computational complexity.

When you call the merge sort implementation of **Sort** on a list of N numbers, the running time can be divided into two components:

1. The amount of time required to execute the operations at the current level of the recursive decomposition
2. The time required to execute the recursive calls

At the top level of the recursive decomposition, the cost of performing the nonrecursive operations is proportional to N . The loop to fill the subsidiary vectors accounts for N cycles, and the call to **Merge** has the effect of refilling the original N positions in the vector. If you add these operations and ignore the constant factor, you discover that the complexity of any single call to **Sort**—not counting the recursive calls within it—requires $O(N)$ operations.

But what about the cost of the recursive operations? To sort an vector of size N , you must recursively sort two vectors of size $N/2$. Each of these operations requires some amount of time. If you apply the same logic, you quickly determine that sorting each of these smaller vectors requires time proportional to $N/2$ at that level, plus whatever time is required by its own recursive calls. The same process then continues until you reach the simple case in which the vectors consist of a single element or no elements at all.

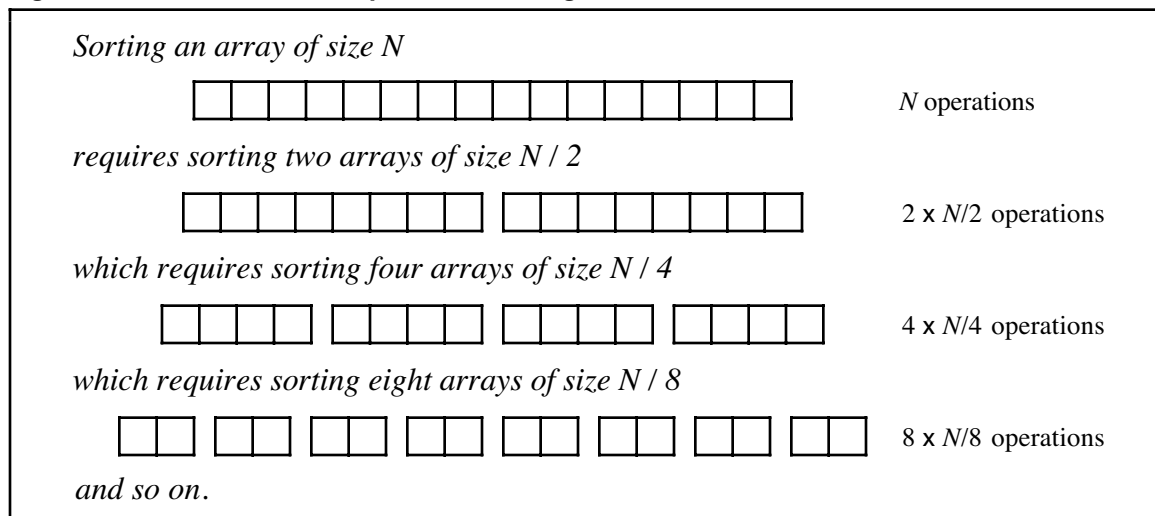
The total time required to solve the problem is the sum of the time required at each level of the recursive decomposition. In general, the decomposition has the structure shown in Figure 8-3. As you move down through the recursive hierarchy, the vectors get smaller, but more numerous. The amount of work done at each level, however, is always directly proportional to N . Determining the total amount of work is therefore a question of finding out how many levels there will be.

At each level of the hierarchy, the value of N is divided by 2. The total number of levels is therefore equal to the number of times you can divide N by 2 before you get down to 1. Rephrasing this problem in mathematical terms, you need to find a value of k such that

$$N = 2^k$$

Solving the equation for k gives

$$k = \log_2 N$$

Figure 8-3 Recursive decomposition of merge sort

Because the number of levels is $\log_2 N$ and the amount of work done at each level is proportional to N , the total amount of work is proportional to $N \log_2 N$.

Unlike other scientific disciplines, in which logarithms are expressed in terms of powers of 10 (common logarithms) or the mathematical constant e (natural logarithms), computer science tends to use **binary logarithms**, which are based on powers of 2. Logarithms computed using different bases differ only by a constant factor, and it is therefore traditional to omit the logarithmic base when you talk about computational complexity. Thus, the computational complexity of merge sort is usually written as

$$O(N \log N)$$

Comparing N^2 and $N \log N$ performance

But how good is $O(N \log N)$? You can compare its performance to that of $O(N^2)$ by looking at the values of these functions for different values of N , as follows:

N	N^2	$N \log N$
10	100	33
100	10,000	664
1000	1,000,000	9965
10,000	100,000,000	132,877

The numbers in both columns grow as N becomes larger, but the N^2 column grows much faster than the $N \log N$ column. Sorting algorithms based on an $N \log N$ algorithm are therefore useful over a much larger range of vector sizes.

It is interesting to verify these results in practice. Because big-O notation discards constant factors, it may be that selection sort is more efficient for some problem sizes. Running the selection and merge sort algorithms on vectors of varying sizes and measuring the actual running times results in the timing data shown in Table 8-2. For 10 items, this implementation of merge sort is more than four times slower than selection sort. At 40 items, selection sort is still faster, but not by very much. By the time you get up to 10,000 items, merge sort is almost 100 times faster than selection sort. On my computer, the selection sort algorithm requires almost a minute and a half to sort 10,000 items while merge sort takes a little under a second. For large vectors, merge sort clearly represents a significant improvement.

Table 8-2 Empirical comparison of selection and merge sorts

N	Selection sort	Merge sort
10	0.12 msec	0.54 msec
20	0.39 msec	1.17 msec
40	1.46 msec	2.54 msec
100	8.72 msec	6.90 msec
200	33.33 msec	14.84 msec
400	135.42 msec	31.25 msec
1000	841.67 msec	84.38 msec
2000	3.35 sec	179.17 msec
4000	13.42 sec	383.33 msec
10,000	83.90 sec	997.67 msec

8.4 Standard complexity classes

In programming, most algorithms fall into one of several common complexity classes. The most important complexity classes are shown in Table 8-3, which gives the common name of the class along with the corresponding big-O expression and a representative algorithm in that class.

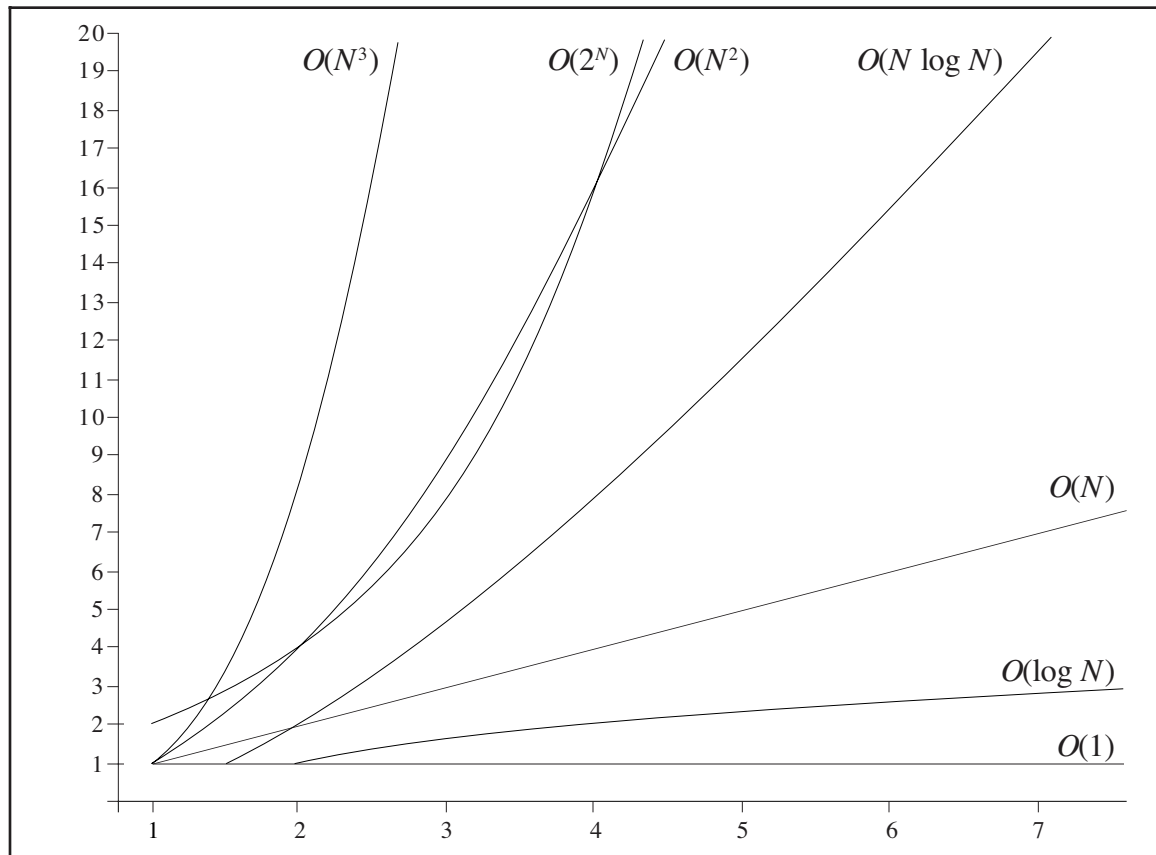
The classes in Table 8-3 are presented in strictly increasing order of complexity. If you have a choice between one algorithm that requires $O(\log N)$ time and another that requires $O(N)$ time, the first will always outperform the second as N grows large. For small values of N , terms that are discounted in the big-O calculation may allow a theoretically less efficient algorithm to do better against one that has a lower computational complexity. On the other hand, as N grows larger, there will always be a point at which the theoretical difference in efficiency becomes the deciding factor.

The differences in efficiency between these classes are in fact profound. You can begin to get a sense of how the different complexity functions stand in relation to one another by looking at the graph in Figure 8-4, which plots these complexity functions on a traditional linear scale. Unfortunately, this graph tells an incomplete and somewhat misleading part of the story, because the values of N are all very small. Complexity analysis, after all, is primarily relevant as the values of N become large. Figure 8-5 shows the same data plotted on a logarithmic scale, which gives you a better sense of how these functions grow.

Algorithms that fall into the constant, linear, quadratic, and cubic complexity classes are all part of a more general family called **polynomial algorithms** that execute in time N^k for some constant k . One of the useful properties of the logarithmic plot shown in Figure 8-5 is that the graph of any function N^k always comes out as a straight line whose slope is proportional to k . From looking at the figure, it is immediately clear that the

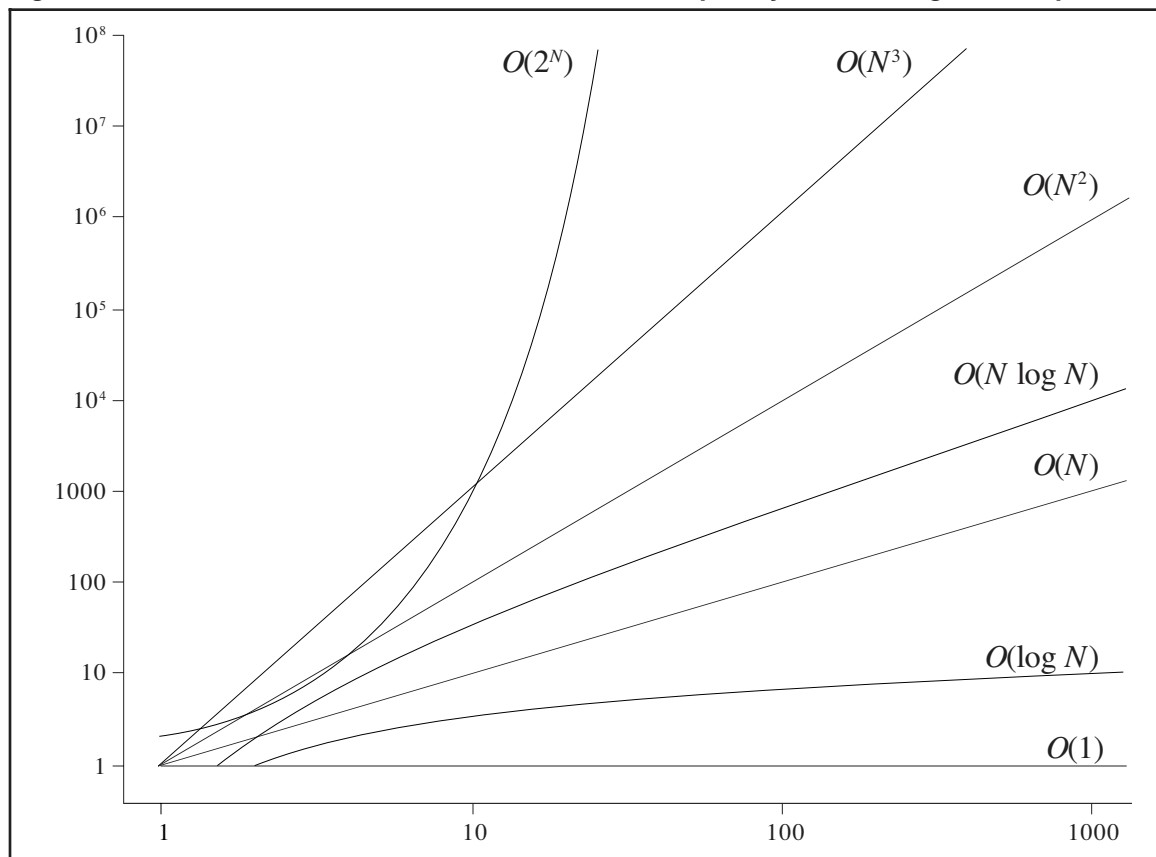
Table 8-3 Standard complexity classes

constant	$O(1)$	Returning the first element in an array or vector
logarithmic	$O(\log N)$	Binary search in a sorted vector
linear	$O(N)$	Linear search in a vector
$N \log N$	$O(N \log N)$	Merge sort
quadratic	$O(N^2)$	Selection sort
cubic	$O(N^3)$	Conventional algorithms for matrix multiplication
exponential	$O(2^N)$	Tower of Hanoi

Figure 8-4 Growth characteristics of the standard complexity classes: linear plot

function N^k —no matter how big k happens to be—will invariably grow more slowly than the exponential function represented by 2^N , which continues to curve upward as the value of N increases. This property has important implications in terms of finding practical algorithms for real-world problems. Even though the selection sort example makes it clear that quadratic algorithms have substantial performance problems for large values of N , algorithms whose complexity is $O(2^N)$ are considerably worse. As a general rule of thumb, computer scientists classify problems that can be solved using algorithms that run in polynomial time as **tractable**, in the sense that they are amenable to implementation on a computer. Problems for which no polynomial time algorithm exists are regarded as **intractable**.

Unfortunately, there are many commercially important problems for which all known algorithms require exponential time. For example, suppose you are a member of the sales force for a business and need to find a way to start at your home office, visit a list of cities, and return home, in a way that minimizes the cost of travel. This problem has become a classic in computer science and is called the **traveling salesman problem**. As far as anyone knows, it is not possible to solve the traveling salesman problem in polynomial time. The best-known approaches all have exponential performance in the worst case and are equivalent in efficiency to generating all possible routings and comparing the cost. In general, the number of possible routes in a connected network of N cities grows in proportion to 2^N , which gives rise to the exponential behavior. On the other hand, no one has been able to prove conclusively that no polynomial-time algorithm for this problem exists. There might be some clever algorithm that makes this problem tractable. There are many open questions as to what types of problems can actually be solved in polynomial time, which makes this topic an exciting area of active research.

Figure 8-5 Growth characteristics of the standard complexity classes: logarithmic plot

8.5 The Quicksort algorithm

Even though the merge sort algorithm presented earlier in this chapter performs well in theory and has a worst-case complexity of $O(N \log N)$, it is not used much in practice. Instead, most sorting programs in use today are based on an algorithm called Quicksort, developed by the British computer scientist C. A. R. (Tony) Hoare.

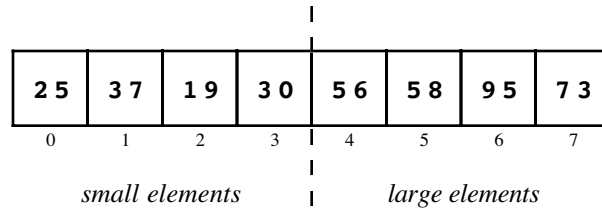
Both Quicksort and merge sort employ a divide-and-conquer strategy. In the merge sort algorithm, the original vector is divided into two halves, each of which is sorted independently. The resulting sorted vectors are then merged together to complete the sort operation for the entire vector. Suppose, however, that you took a different approach to dividing up the vector. What would happen if you started the process by making an initial pass through the vector, changing the positions of the elements so that “small” values came at the beginning of the vector and “large” values came at the end, for some appropriate definition of the words *large* and *small*?

For example, suppose that the original vector you wanted to sort was the following one, presented earlier in the discussion of merge sort:

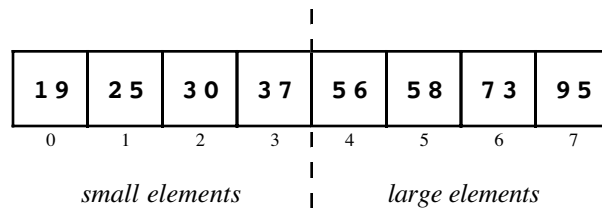
56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

Half of these elements are larger than 50 and half are smaller, so it might make sense to define *small* in this case as being less than 50 and *large* as being 50 or more. If you could

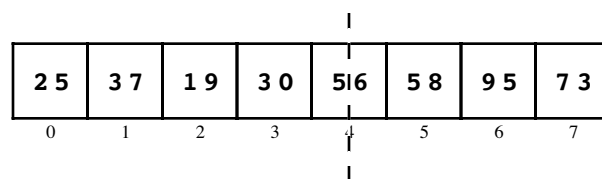
then find a way to rearrange the elements so that all the small elements came at the beginning and all the large ones at the end, you would wind up with an vector that looks something like the following diagram, which shows one of many possible orderings that fit the definition:



When the elements are divided into parts in this fashion, all that remains to be done is to sort each of the parts, using a recursive call to the function that does the sorting. Since all the elements on the left side of the boundary line are smaller than all those on the right, the final result will be a completely sorted vector:



If you could always choose the optimal boundary between the small and large elements on each cycle, this algorithm would divide the vector in half each time and end up demonstrating the same qualitative characteristics as merge sort. In practice, the Quicksort algorithm selects some existing element in the vector and uses that value as the dividing line between the small and large elements. For example, a common approach is to pick the first element, which was 56 in the original vector, and use it as the demarcation point between small and large elements. When the vector is reordered, the boundary is therefore at a particular index position rather than between two positions, as follows:



From this point, the recursive calls must sort the vector between positions 0 and 3 and the vector between positions 5 and 7, leaving index position 4 right where it is.

As in merge sort, the simple case of the Quicksort algorithm is a vector of size 0 or 1, which must already be sorted. The recursive part of the Quicksort algorithm consists of the following steps:

1. *Choose an element to serve as the boundary between the small and large elements.* This element is traditionally called the **pivot**. For the moment, it is sufficient to choose any element for this purpose, and the simplest strategy is to select the first element in the vector.
2. *Rearrange the elements in the vector so that large elements are moved toward the end of the vector and small elements toward the beginning.* More formally, the goal of

this step is to divide the elements around a boundary position so that all elements to the left of the boundary are less than the pivot and all elements to the right are greater than or possibly equal to the pivot.² This processing is called **partitioning** the vector and is discussed in detail in the next section.

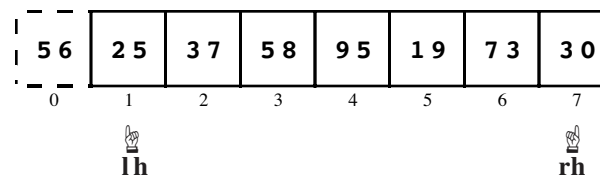
3. *Sort the elements in each of the partial vectors.* Because all elements to the left of the pivot boundary are strictly less than all those to the right, sorting each of the vectors must leave the entire vector in sorted order. Moreover, since the algorithm uses a divide-and-conquer strategy, these smaller vectors can be sorted using a recursive application of Quicksort.

Partitioning the vector

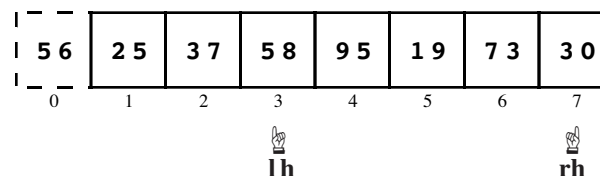
In the partition step of the Quicksort algorithm, the goal is to rearrange the elements so that they are divided into three classes: those that are smaller than the pivot; the pivot element itself, which is situated at the boundary position; and those elements that are at least as large as the pivot. The tricky part about partition is to rearrange the elements without using any extra storage, which is typically done by swapping pairs of elements.

Tony Hoare's original approach to partitioning is fairly easy to explain in English. Because the pivot value has already been selected when you start the partitioning phase of the algorithm, you can tell immediately whether a value is large or small relative to that pivot. To make things easier, let's assume that the pivot value is stored in the initial element position. Hoare's partitioning algorithm proceeds as follows:

1. For the moment, ignore the pivot element at index position 0 and concentrate on the remaining elements. Use two index values, **lh** and **rh**, to record the index positions of the first and last elements in the rest of the vector, as shown:

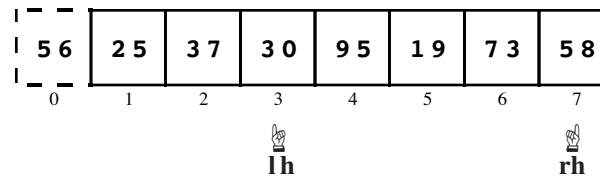


2. Move the **rh** index to the left until it either coincides with **lh** or points to an element containing a value that is small with respect to the pivot. In this example, the value 30 in position 7 is already a small value, so the **rh** index does not need to move.
3. Move the **lh** index to the right until it coincides with **rh** or points to an element containing a value that is larger than or equal to the pivot. In this example, the **lh** index must move to the right until it points to an element larger than 56, which leads to the following configuration:

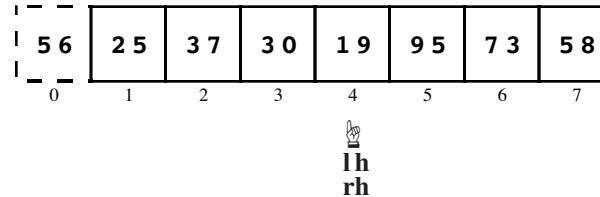


4. If the **lh** and **rh** index values have not yet reached the same position, exchange the elements in those positions in the vector, which leaves it looking like this:

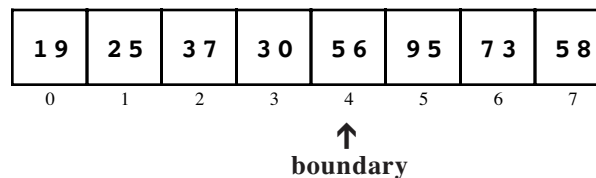
² The subarray to the right of the boundary will contain values equal to the pivot only if the pivot value appears more than once in the array.



5. Repeat steps 2 through 4 until the **lh** and **rh** positions coincide. On the next pass, for example, the exchange operation in step 4 swaps the 19 and the 95. As soon as that happens, the next execution of step 2 moves the **rh** index to the left, where it ends up matching the **lh**, as follows:



6. In most cases, the element at the point where the **lh** and **rh** index positions coincide will be the small value that is furthest to the right in the vector. If that position in fact contains a large value, the pivot must have been the smallest element in the entire vector, so the boundary is at position 0. If that position contains a small value, as it usually does, the only remaining step is to exchange that value in this position with the pivot element at the beginning of the vector, as shown:



Note that this configuration meets the requirements of the partitioning step. The pivot value is at the marked boundary position, with every element to the left being smaller and every element to the right being at least that large.

A simple implementation of **Sort** using the Quicksort algorithm is shown in Figure 8-6.

Figure 8-6 Implementation of Hoare's Quicksort algorithm

```

/*
 * Function: Sort
 * -----
 * This function sorts the elements of the vector into
 * increasing numerical order using the Quicksort algorithm.
 * In this implementation, Sort is a wrapper function that
 * calls Quicksort to do all the work.
 */

void Sort(Vector<int> & vec) {
    Quicksort(vec, 0, vec.size() - 1);
}

```

```

/*
 * Function: Quicksort
 * -----
 * Sorts the elements in the vector between index positions
 * start and finish, inclusive. The Quicksort algorithm begins
 * by "partitioning" the vector so that all elements smaller
 * than a designated pivot element appear to the left of a
 * boundary and all equal or larger values appear to the right.
 * Sorting the subsidiary vectors to the left and right of the
 * boundary ensures that the entire vector is sorted.
 */

void Quicksort(Vector<int> & vec, int start, int finish) {
    if (start >= finish) return;
    int boundary = Partition(vec, start, finish);
    Quicksort(vec, start, boundary - 1);
    Quicksort(vec, boundary + 1, finish);
}

/*
 * Function: Partition
 * -----
 * This function rearranges the elements of the vector so that the
 * small elements are grouped at the left end of the vector and the
 * large elements are grouped at the right end. The distinction
 * between small and large is made by comparing each element to the
 * pivot value, which is initially taken from vec[start]. When the
 * partitioning is done, the function returns a boundary index such
 * that vec[i] < pivot for all i < boundary, vec[i] == pivot
 * for i == boundary, and vec[i] >= pivot for all i > boundary.
 */

int Partition(Vector<int> & vec, int start, int finish) {
    int pivot = vec[start];
    int lh = start + 1;
    int rh = finish;
    while (true) {
        while (lh < rh && vec[rh] >= pivot) rh--;
        while (lh < rh && vec[lh] < pivot) lh++;
        if (lh == rh) break;
        int temp = vec[lh];
        vec[lh] = vec[rh];
        vec[rh] = temp;
    }
    if (vec[lh] >= pivot) return start;
    vec[start] = vec[lh];
    vec[lh] = pivot;
    return lh;
}

```

Analyzing the performance of Quicksort

As you can see from Table 8-4, this implementation of Quicksort tends to run several times faster than the implementation of merge sort given in Figure 8-2, which is one of the reasons why programmers use it more frequently in practice. Moreover, the running times for both algorithms appear to grow in roughly the same way.

Table 8-4 Empirical comparison of merge sort and Quicksort

N	Merge sort	Quicksort
10	0.54 msec	0.10 msec
20	1.17 msec	0.26 msec
40	2.54 msec	0.52 msec
100	6.90 msec	1.76 msec
200	14.84 msec	4.04 msec
400	31.25 msec	8.85 msec
1000	84.38 msec	26.04 msec
2000	179.17 msec	56.25 msec
4000	383.33 msec	129.17 msec
10,000	997.67 msec	341.67 msec

The empirical results presented in Table 8-4, however, obscure an important point. As long as the Quicksort algorithm chooses a pivot that is close to the median value in the vector, the partition step will divide the vector into roughly equal parts. If the algorithm chooses its pivot value poorly, one of the two partial vectors may be much larger than the other, which defeats the purpose of the divide-and-conquer strategy. In a vector with randomly chosen elements, Quicksort tends to perform well, with an average-case complexity of $O(N \log N)$. In the worst case—which paradoxically consists of a vector that is already sorted—the performance degenerates to $O(N^2)$. Despite this inferior behavior in the worst case, Quicksort is so much faster in practice than most other algorithms that it has become the standard choice for general sorting procedures.

There are several strategies you can use to increase the likelihood that the pivot is in fact close to the median value in the vector. One simple approach is to have the Quicksort implementation choose the pivot element at random. Although it is still possible that the random process will choose a poor pivot value, it is unlikely that it would make the same mistake repeatedly at each level of the recursive decomposition. Moreover, there is no distribution of the original vector that is always bad. Given any input, choosing the pivot randomly ensures that the average-case performance for that vector would be $O(N \log N)$. Another possibility, which is explored in more detail in exercise 7, is to select a few values, typically three or five, from the vector and choose the median of those values as the pivot.

You do have to be somewhat careful as you try to improve the algorithm in this way. Picking a good pivot improves performance, but also costs some time. If the algorithm spends more time choosing the pivot than it gets back from making a good choice, you will end up slowing down the implementation rather than speeding it up.

8.6 Mathematical induction

Earlier in the chapter, I asked you to rely on the fact that the sum

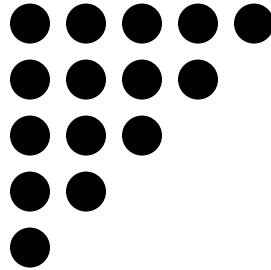
$$N + N-1 + N-2 + \dots + 3 + 2 + 1$$

could be simplified to the more manageable formula

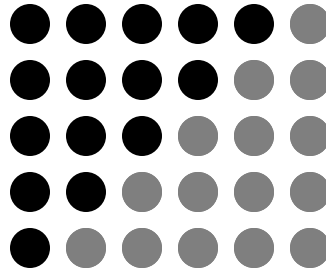
$$\frac{N(N+1)}{2}$$

If you were skeptical about this simplification, how would you go about proving that the simplified formula is indeed correct?

There are, in fact, several different proof techniques you could try. One possibility is to represent the original extended sum in a geometric form. Suppose, for example, that N is 5. If you then represent each term in the summation with a row of dots, those dots form the following triangle:



If you make a copy of this triangle and flip it upside down, the two triangles fit together to form a rectangle, shown here with the lower triangle in gray:



Since the pattern is now rectangular, the total number of dots—both black and gray—is easy to compute. In this picture, there are five rows of six dots each, so the total collection of dots, counting both colors, is 5×6 , or 30. Since the two triangles are identical, exactly half of these dots are black; thus the number of black dots is $30 / 2$, or 15. In the more general case, there are N rows containing $N+1$ dots each, and the number of black dots from the original triangle is therefore

$$\frac{N(N+1)}{2}$$

Proving that a formula is correct in this fashion, however, has some potential drawbacks. For one thing, geometrical arguments presented in this style are not as formal as many computer scientists would like. More to the point, constructing this type of argument requires that you come up with the right geometrical insight, which is different for each problem. It would be better to adopt a more general proof strategy that would apply to many different problems.

The technique that computer scientists generally use to prove propositions like

$$N + N-1 + N-2 + \dots + 3 + 2 + 1 = \frac{N(N+1)}{2}$$

is called **mathematical induction**. Mathematical induction applies when you want to show that a proposition is true for all values of an integer N beginning at some initial starting point. This starting point is called the **basis** of the induction and is typically 0 or 1. The process consists of the following steps:

- *Prove the base case.* The first step is to establish that the proposition holds true when N has the basis value. In most cases, this step is a simple matter of plugging the basis value into a formula and showing that the desired relationship holds.
- *Prove the inductive case.* The second step is to demonstrate that, if you assume the proposition to be true for N , it must also be true for $N+1$.

As an example, here is how you can use mathematical induction to prove the proposition that

$$N + N-1 + N-2 + \dots + 3 + 2 + 1 = \frac{N(N+1)}{2}$$

is indeed true for all N greater than or equal to 1. The first step is to prove the base case, when N is equal to 1. That part is easy. All you have to do is substitute 1 for N in both halves of the formula to determine that

$$1 = \frac{1 \times (1+1)}{2} = \frac{2}{2} = 1$$

To prove the inductive case, you begin by assuming that the proposition

$$N + N-1 + N-2 + \dots + 3 + 2 + 1 = \frac{N(N+1)}{2}$$

is indeed true for N . This assumption is called the **inductive hypothesis**. Your goal is now to verify that the same relationship holds for $N+1$. In other words, what you need to do to establish the truth of the current formula is to show that

$$N+1 + N + N-1 + N-2 + \dots + 3 + 2 + 1 = \frac{(N+1)(N+2)}{2}$$

If you look at the left side of the equation, you should notice that the sequence of terms beginning with N is exactly the same as the left side of your inductive hypothesis. Since you have assumed that the inductive hypothesis is true, you can substitute the equivalent closed-form expression, so that the left side of the proposition you're trying to prove looks like this:

$$N+1 + \frac{N(N+1)}{2}$$

From here on, the rest of the proof is simple algebra:

$$\begin{aligned} N+1 + \frac{N(N+1)}{2} \\ &= \frac{2N+2}{2} + \frac{N^2+N}{2} \\ &= \frac{N^2+3N+2}{2} \\ &= \frac{(N+1)(N+2)}{2} \end{aligned}$$

The last line in this derivation is precisely the result you were looking for and therefore completes the proof.

Many students need time to get used to the idea of mathematical induction. At first glance, the inductive hypothesis seems to be “cheating” in some sense; after all, you get to assume precisely the proposition that you are trying to prove. In fact, the process of mathematical induction is nothing more than an infinite family of proofs, each of which proceeds by the same logic. The base case in a typical example establishes that the proposition is true for $N = 1$. Once you have proved the base case, you can adopt the following chain of reasoning:

Now that I know the proposition is true for $N = 1$, I can prove it is true for $N = 2$.
Now that I know the proposition is true for $N = 2$, I can prove it is true for $N = 3$.
Now that I know the proposition is true for $N = 3$, I can prove it is true for $N = 4$.
Now that I know the proposition is true for $N = 4$, I can prove it is true for $N = 5$.
And so on. . . .

At each step in this process, you could write out a complete proof by applying the logic you used to establish the inductive case. The power of mathematical induction comes from the fact that you don’t actually need to write out the details of each step individually.

In a way, the process of mathematical induction is like the process of recursion viewed from the opposite direction. If you try to explain a typical recursive decomposition in detail, the process usually sounds something like this:

To calculate this function for $N = 5$, I need to know its value for $N = 4$.
To calculate this function for $N = 4$, I need to know its value for $N = 3$.
To calculate this function for $N = 3$, I need to know its value for $N = 2$.
To calculate this function for $N = 2$, I need to know its value for $N = 1$.
The value $N = 1$ represents a simple case, so I can return the result immediately.

Both induction and recursion require you to make a leap of faith. When you write a recursive function, this leap consists of believing that all simpler instances of the function call will work without your paying any attention to the details. Making the inductive hypothesis requires much the same mental discipline. In both cases, you have to restrict your thinking to one level of the solution and not get sidetracked trying to follow the details all the way to the end.

Summary

The most valuable concept to take with you from this chapter is that algorithms for solving a problem can vary widely in their performance characteristics. Choosing an algorithm that has better computational properties can often reduce the time required to solve a problem by many orders of magnitude. The difference in behavior is illustrated dramatically by the tables presented in this chapter that give the actual running times for various sorting algorithms. When sorting an vector of 10,000 integers, for example, the Quicksort algorithm outperforms selection sort by a factor of almost 250; as the vector sizes get larger, the difference in efficiency between these algorithms will become even more pronounced.

Other important points in this chapter include:

- Most algorithmic problems can be characterized by an integer N that represents the size of the problem. For algorithms that operate on large integers, the size of the

integer provides an effective measure of problem size; for algorithms that operate on arrays or vectors, it usually makes sense to define the problem size as the number of elements.

- The most useful qualitative measure of efficiency is *computational complexity*, which is defined as the relationship between problem size and algorithmic performance as the problem size becomes large.
- *Big-O notation* provides an intuitive way of expressing computational complexity because it allows you to highlight the most important aspects of the complexity relationship in the simplest possible form.
- When you use big-O notation, you can simplify the formula by eliminating any term in the formula that becomes insignificant as N becomes large, along with any constant factors.
- You can often predict the computational complexity of a program by looking at the nesting structure of the loops it contains.
- Two useful measures of complexity are *worst-case* and *average-case* analysis. Average-case analysis is usually much more difficult to conduct.
- Divide-and-conquer strategies make it possible to reduce the complexity of sorting algorithms from $O(N^2)$ to $O(N \log N)$, which is a significant reduction.
- Most algorithms fall into one of several common *complexity classes*, which include the *constant*, *logarithmic*, *linear*, $N \log N$, *quadratic*, *cubic*, and *exponential* classes. Algorithms whose complexity class appears earlier in this list are more efficient than those that come afterward when the problems being considered are sufficiently large.
- Problems that can be solved in *polynomial time*, which is defined to be $O(N^k)$ for some constant value k , are considered to be *tractable*. Problems for which no polynomial-time algorithm exists are considered *intractable* because solving such problems requires prohibitive amounts of time even for problems of relatively modest size.
- Because it tends to perform extremely well in practice, most sorting programs are based on the *Quicksort algorithm*, developed by Tony Hoare, even though its worst-case complexity is $O(N^2)$.
- Mathematical induction provides a general technique for proving that a property holds for all values of N greater than or equal to some *base* value. To apply this technique, your first step is to demonstrate that the property holds in the base case. In the second step, you must prove that, if the formula holds for a specific value N , then it must also hold for $N+1$.

Review questions

1. The simplest recursive implementation of the Fibonacci function is considerably less efficient than the iterative version. Does this fact allow you to make any general conclusions about the relative efficiency of recursive and iterative solutions?
2. What is the sorting problem?
3. The implementation of **Sort** shown in Figure 8-1 runs through the code to exchange the values at positions **lh** and **rh** even if these values happen to be the same. If you change the program so that it checks to make sure **lh** and **rh** are different before making the exchange, it is likely to run more slowly than the original algorithm. Why might this be so?
4. Suppose that you are using the selection sort algorithm to sort a vector of 250 values and find that it takes 50 milliseconds to complete the operation. What would you

expect the running time to be if you used the same algorithm to sort a vector of 1000 values on the same machine?

5. What is the closed-form expression that computes the sum of the series

$$N + N-1 + N-2 + \dots + 3 + 2 + 1$$

6. In your own words, define the concept of computational complexity.
7. True or false: Big-O notation was invented as a means to express computational complexity.
8. What are the two rules presented in this chapter for simplifying big-O notation?
9. Is it technically correct to say that selection sort runs in

$$O\left(\frac{N^2 + N}{2}\right)$$

time? What, if anything, is wrong with doing so?

10. Is it technically correct to say that selection sort runs in $O(N^3)$ time? Again, what, if anything, is wrong with doing so?
11. Why is it customary to omit the base of the logarithm in big-O expressions such as $O(N \log N)$?
12. What is the computational complexity of the following function:

```
int Mystery1(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            sum += i * j;
        }
    }
    return sum;
}
```

13. What is the computational complexity of this function:

```
int Mystery2(int n) {
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < i; j++) {
            sum += j * n;
        }
    }
    return sum;
}
```

14. Explain the difference between worst-case and average-case complexity. In general, which of these measures is harder to compute?
15. State the formal definition of big-O.
16. In your own words, explain why the **Merge** function runs in linear time.

17. The last two lines of the **Merge** function are

```
while (p1 < n1) vec.add(v1[p1++]);
while (p2 < n2) vec.add(v2[p2++]);
```

Would it matter if these two lines were reversed? Why or why not?

18. What are the seven complexity classes identified in this chapter as the most common classes encountered in practice?
19. What does the term *polynomial algorithm* mean?
20. What criterion do computer scientists use to differentiate tractable and intractable problems?
21. In the Quicksort algorithm, what conditions must be true at the conclusion of the partitioning step?
22. What are the worst- and average-case complexities for Quicksort?
23. Describe the two steps involved in a proof by mathematical induction.
24. In your own words, describe the relationship between recursion and mathematical induction.

Programming exercises

1. It is easy to write a recursive function

```
double RaiseToPower(double x, int n)
```

that calculates x^n , by relying on the recursive insight that

$$x^n = x \times x^{n-1}$$

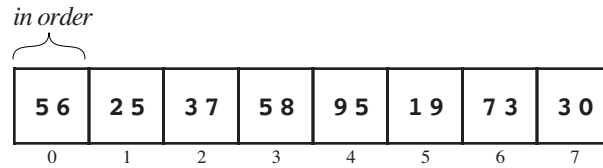
Such a strategy leads to an implementation that runs in linear time. You can, however, adopt a recursive divide-and-conquer strategy which takes advantage of the fact that

$$x^{2^n} = x^n \times x^n$$

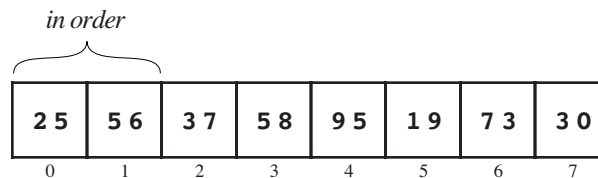
Use this fact to write a recursive version of **RaiseToPower** that runs in $O(\log N)$ time.

2. There are several other sorting algorithms that exhibit the $O(N^2)$ behavior of selection sort. Of these, one of the most important is **insertion sort**, which operates as follows. You go through each element in the vector in turn, as with the selection sort algorithm. At each step in the process, however, the goal is not to find the smallest remaining value and switch it into its correct position, but rather to ensure that the values considered so far are correctly ordered with respect to each other. Although those values may shift as more elements are processed, they form an ordered sequence in and of themselves.

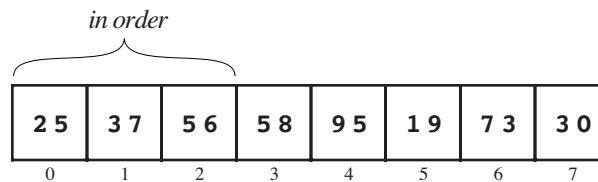
For example, if you consider again the data used in the sorting examples from this chapter, the first cycle of the insertion sort algorithm requires no work, because an vector of one element is always sorted:



On the next cycle, you need to put 25 into the correct position with respect to the elements you have already seen, which means that you need to exchange the 56 and 25 to reach the following configuration:



On the third cycle, you need to find where the value 37 should go. To do so, you need to move backward through the earlier elements—which you know are in order with respect to each other—looking for the position where 37 belongs. As you go, you need to shift each of the larger elements one position to the right, which eventually makes room for the value you’re trying to insert. In this case, the 56 gets shifted by one position, and the 37 winds up in position 1. Thus, the configuration after the third cycle looks like this:

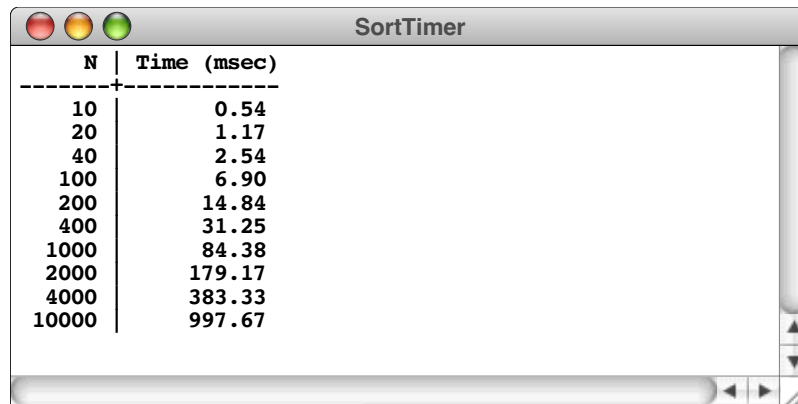


After each cycle, the initial portion of the vector is always sorted, which implies that cycling through all the positions in this way will sort the entire vector.

The insertion sort algorithm is important in practice because it runs in linear time if the vector is already more or less in the correct order. It therefore makes sense to use insertion sort to restore order to a large vector in which only a few elements are out of sequence.

Write an implementation of **Sort** that uses the insertion sort algorithm. Construct an informal argument to show that the worst-case behavior of insertion sort is $O(N^2)$.

3. Suppose you know that all the values in an integer array fall into the range 0 to 9999. Show that it is possible to write a $O(N)$ algorithm to sort arrays with this restriction. Implement your algorithm and evaluate its performance by taking empirical measurements using the strategy outlined in exercise 3. Explain why the performance of the algorithm is so bad for small values of N .
4. Write a function that keeps track of the elapsed time as it executes the **Sort** procedure on a randomly chosen vector. Use that function to write a program that produces a table of the observed running times for a predefined set of sizes, as shown in the following sample run:



N	Time (msec)
10	0.54
20	1.17
40	2.54
100	6.90
200	14.84
400	31.25
1000	84.38
2000	179.17
4000	383.33
10000	997.67

The best way to measure elapsed system time for programs of this sort is to use the ANSI **clock** function, which is exported by the **ctime** interface. The **clock** function takes no arguments and returns the amount of time the processing unit of the computer has used in the execution of the program. The unit of measurement and even the type used to store the result of **clock** differ depending on the type of machine, but you can always convert the system-dependent clock units into seconds by using the following expression:

```
double(clock()) / CLOCKS_PER_SEC
```

If you record the starting and finishing times in the variables **start** and **finish**, you can use the following code to compute the time required by a calculation:

```
double start, finish, elapsed;

start = double(clock()) / CLOCKS_PER_SEC;
. . . Perform some calculation . . .
finish = double(clock()) / CLOCKS_PER_SEC;
elapsed = finish - start;
```

Unfortunately, calculating the time requirements for a program that runs quickly requires some subtlety because there is no guarantee that the system clock unit is precise enough to measure the elapsed time. For example, if you used this strategy to time the process of sorting 10 integers, the odds are good that the time value of **elapsed** at the end of the code fragment would be 0. The reason is that the processing unit on most machines can execute many instructions in the space of a single clock tick—almost certainly enough to get the entire sorting process done for a vector of 10 elements. Because the system’s internal clock may not tick in the interim, the values recorded for **start** and **finish** are likely to be the same.

The best way to get around this problem is to repeat the calculation many times between the two calls to the **clock** function. For example, if you want to determine how long it takes to sort 10 numbers, you can perform the sort-10-numbers experiment 1000 times in a row and then divide the total elapsed time by 1000. This strategy gives you a timing measurement that is much more accurate.

5. The implementations of the various sort algorithms in this chapter are written to sort a **Vector<int>**, because the **Vector** type is safer and more convenient than arrays. Existing software libraries, however, are more likely to define these functions so that they work with arrays, which means that the prototype for the **Sort** function is

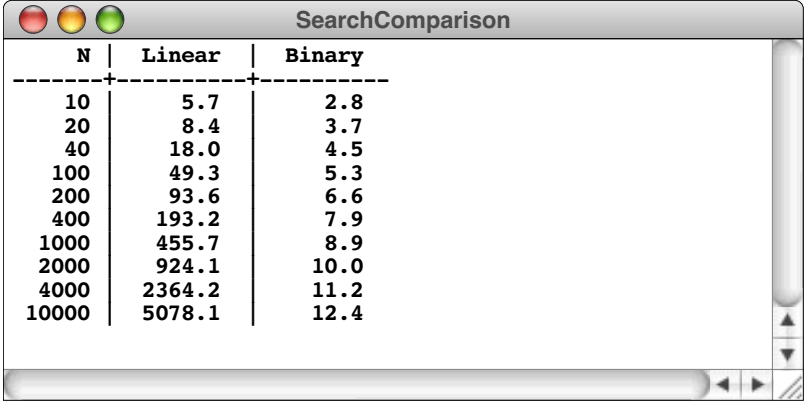
```
void Sort(int array[], int n)
```

where **n** is the effective size of the array.

Revise the implementation of the three sorting algorithms presented in this chapter so that the use arrays rather than the collection classes from Chapter 4.

6. Write a program that generates a table which compares the performance of two algorithms—linear and binary search—when used to find a randomly chosen integer key in a sorted **Vector<int>**. The linear search algorithm simply goes through each element of the vector in turn until it finds the desired one or determines that the key does not appear. The binary search algorithm, which is implemented for string arrays in Figure 5-5, uses a divide-and-conquer strategy by checking the middle element of the vector and then deciding which half of the remaining elements to search.

The table you generate in this problem, rather than computing the time as in exercise 3, should instead calculate the number of comparisons made against elements of the vector. To ensure that the results are not completely random, your program should average the results over several independent trials. A sample run of the program might look like this:



N	Linear	Binary
10	5.7	2.8
20	8.4	3.7
40	18.0	4.5
100	49.3	5.3
200	93.6	6.6
400	193.2	7.9
1000	455.7	8.9
2000	924.1	10.0
4000	2364.2	11.2
10000	5078.1	12.4

7. Change the implementation of the Quicksort algorithm so that, instead of picking the first element in the vector as the pivot, the **Partition** function chooses the median of the first, middle, and last elements.
8. Although $O(N \log N)$ sorting algorithms are clearly more efficient than $O(N^2)$ algorithms for large vectors, the simplicity of quadratic algorithms like selection sort often means that they perform better for small values of N . This fact raises the possibility of developing a strategy that combines the two algorithms, using Quicksort for large vectors but selection sort whenever the vectors become less than some threshold called the **crossover point**. Approaches that combine two different algorithms to exploit the best features of each are called **hybrid strategies**.

Reimplement **Sort** using a hybrid of the Quicksort and selection sort strategies. Experiment with different values of the crossover point below which the implementation chooses to use selection sort, and determine what value gives the best performance. The value of the crossover point depends on the specific timing characteristics of your computer and will change from system to system.

9. Another interesting hybrid strategy for the sorting problem is to start with a recursive Quicksort that simply returns when the size of the vector falls below a certain threshold. When this function returns, the vector is not sorted, but all the elements are relatively close to their final positions. At this point, you can use the insertion

sort algorithm presented in exercise 2 on the entire vector to fix any remaining problems. Because insertion sort runs in linear time on vectors that are mostly sorted, you may be able to save some time using this approach.

10. Suppose you have two functions, f and g , for which $f(N)$ is less than $g(N)$ for all values of N . Use the formal definition of big-O to prove that

$$15f(N) + 6g(N)$$

is $O(g(N))$.

11. Use the formal definition of big-O to prove that N^2 is $O(2^N)$.
12. Use mathematical induction to prove that the following properties hold for all positive values of N .

a) $1 + 3 + 5 + 7 + \dots + 2N-1 = N^2$

b) $1^2 + 2^2 + 3^2 + 4^2 + \dots + N^2 = \frac{N(N+1)(2N+1)}{6}$

c) $1^3 + 2^3 + 3^3 + 4^3 + \dots + N^3 = (1 + 2 + 3 + 4 + \dots + N)^2$

d) $2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^N = 2^{N+1} - 1$

13. Exercise 1 shows that it is possible to compute x^n in $O(\log N)$ time. This fact in turn makes it possible to write an implementation of the function **Fib(n)** that also runs in $O(\log N)$ time, which is much faster than the traditional iterative version. To do so, you need to rely on the somewhat surprising fact that the Fibonacci function is closely related to a value called the **golden ratio**, which has been known since the days of Greek mathematics. The golden ratio, which is usually designated by the Greek letter ϕ , is defined to be the value that satisfies the equation

$$\phi^2 - \phi - 1 = 0$$

Because this is a quadratic equation, it actually has two roots. If you apply the quadratic formula, you will discover that these roots are

$$\phi = \frac{1 + \sqrt{5}}{2}$$

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2}$$

In 1718, the French mathematician Abraham de Moivre discovered that the n^{th} Fibonacci number can be represented in closed form as

$$\frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

Moreover, since $\hat{\phi}^n$ is always very small, the formula can be simplified to

$$\frac{\phi^n}{\sqrt{5}}$$

rounded to the nearest integer.

Use this formula and the **RaiseToPower** function from exercise 1 to write an implementation of **Fib(n)** that runs in $O(\log N)$ time. Once you have verified empirically that the formula seems to work for the first several terms in the sequence, use mathematical induction to prove that the formula

$$\frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

actually computes the n^{th} Fibonacci number.

14. If you're ready for a real algorithmic challenge, write the function

```
int MajorityElement(Vector<int> & vec);
```

that takes a vector of nonnegative integers and returns the **majority element**, which is defined to be a value that occurs in an absolute majority (at least 50 percent plus one) of the element positions. If no majority element exists, the function should return -1 to signal that fact. Your function must also meet the following conditions:

- It must run in $O(N)$ time.
- It must use $O(1)$ additional space. In other words, it may use individual temporary variables but may not allocate any additional array or vector storage. Moreover, this condition rules out recursive solutions, because the space required to store the stack frames would grow with the depth of the recursion.
- It may not change any of the values in the vector.

The hard part about this problem is coming up with the algorithm, not implementing it. Play with some sample vectors and see if you can come up with the right strategy.

Chapter 9

Classes and Objects

You don't understand. I coulda had class.

— Marlon Brando's character in *On the Waterfront*, 1954

Although you have been using classes extensively ever since Chapter 3, you have not yet had the chance to create new classes on your own. The purpose of this chapter is to fill that gap by giving you the tools you need to implement new classes in C++. The chapter begins with a few simple examples, and then moves on to more sophisticated designs.

9.1 A simple example of a class definition

Before diving into the details of defining new classes in C++, it make sense to look at a few simple classes to get an impression of how they work. In certain ways, classes are similar to structures in that they combine independent data values into a coherent whole. It is presumably not too surprising that it is possible to replace any structure definition with a class that does the same thing. Since you already have some intuition about how structures work, classes that duplicate the function of structures seems like a good place to start.

Defining a `Point` class

One of the simpler structure types you have seen so far in this book is the `pointT` structure, which was introduced in Chapter 7 as part of the discussion of solving a maze. The definition of `pointT`, which appears in the `mazelib.h` interface shown in Figure 7-2, looks like this:

```
struct pointT {  
    int x, y;  
};
```

This structure combines an independent *x* and *y* coordinate into a unified structure that you can manipulate as a single value. You can assign one `pointT` value to another, pass it as a parameter to a function, or return it as a result. If you need to, you can select the individual *x* and *y* fields to look at the components individually, but it is often sufficient to work with the point value as a whole.

You can accomplish the same result by replacing `pointT` with the following class:

```
class Point {  
public:  
    int x, y;  
};
```

If you make this change to the `mazelib.h` interface and then replace every occurrence of `pointT` with `Point` in all the source files, the maze application will run precisely as it did before.

As you can see from this example, the fields of a class—which are also called **data members** or **instance variables** when they occur in the context of a class—are declared using much the same syntax as they are for records. The only syntactic difference is that fields within a class are separated into public and private sections to control the visibility of the fields. The keyword `public` introduces the section containing fields that are available to anyone who uses the defining class. A class definition, however, can also include a `private` section; fields that are declared in the `private` section are visible only to the defining class and not to any of its clients.

As the definition of the `Point` class now stands, the *x* and *y* fields are included as part of the `public` section, which makes them visible to visible to clients. In effect, making these instance variables public means that they act exactly like fields in a record. You

can select a public field in an object by using the dot operator. For example, if you have a variable **pt** containing an object that is an instance of the **Point** class, you can select its **x** field by writing

```
pt.x
```

just as if it were still the **pointT** record used in the original implementation.

Declaring public instance variables, however, is generally discouraged in modern approaches to object-oriented programming. Today, the common practice is to make all instance variables private, which means that the client has no direct access to the independent components. Clients instead use methods provided by the class to obtain access to any information the class contains. Keeping implementation details away from the client is likely to foster simplicity, flexibility, and security, as described in the introduction to Chapter 4.

Making the instance variables private is easy enough. All you have to do is change the label that introduces that section of the class from **public** to **private**, as follows:

```
class Point {  
  private:  
    int x, y;  
};
```

The problem is that clients have no way to access the information stored within a **Point** object, which makes it unusable in its current form. Before clients can use a **Point** class that keeps its data private, it is essential to define an appropriate set of methods for working with objects of that class.

In addition to declarations of instance variables, the public and private sections of a class definition include the prototypes for any methods that apply to that class. Method prototypes that appear in the public section of a class are available to clients; such methods are said to be **exported** by the class. By contrast, prototypes that appear in the private section are available only within the implementation.

The most straightforward way to provide access to the information inside a class is to export methods that return the value stored in an instance variable. Such methods are formally called **accessors**, but are referred to colloquially as **getters**. You could, for example, make it possible for clients to read the values of the **x** and **y** fields by exporting two getter methods, one for each component. The class definition that includes the prototypes for the getter methods looks like this:

```
class Point {  
  public:  
    int getX();  
    int getY();  
  
  private:  
    int x, y;  
};
```

As of yet, the code does not specify how the **getX** and **getY** methods are associated with the **x** and **y** instance variables. In C++, the class definition itself supplies only the method prototypes. The implementation of those methods is defined separately, as described in the following section.

Implementing methods in a class

In C++, the methods that are part of a class—both the public methods exported by the class and the private ones that support its implementation—are defined separately from the class specification itself. This approach has the important advantage of separating the specification of the class from its underlying implementation. For the most part, clients need to look only at the class specification and never have to dig deeper into the details.

The separation between the specification and implementation of a class, however, does introduce one small complication into the syntax of the implementation. Suppose that you want to write the definitions of the **getX** and **getY** methods that are exported by the class. It is not enough to define **getX** and **getY** methods because the compiler has no way of knowing which class is exporting those particular methods. In C++, many classes might export a **getX** method. When you supply an implementation for one of them, you also need to identify the class to which that method belongs. In C++, this identification is accomplished by adding the class name as a **qualifier** before the method name, separated by a double colon. The fully qualified name of the **getX** method in the **Point** class is therefore **Point::getX**. Once that minor syntactic wrinkle is out of the way, however, the rest of the implementation is quite simple:

```
int Point::getX() {  
    return x;  
}  
  
int Point::getY() {  
    return y;  
}
```

The methods illustrate an interesting aspect of C++'s rules for determining the scope of variable names. If the name of an instance variable or method appears without specifying the object to which it applies, that field is assumed to apply to the object currently serving as the receiver. Thus, the variable **x** in the **getX** method refers to the instance variable **x** in the **Point** object to which **getX** is being applied.

Although implementing the getter methods for the **Point** class makes it possible to get information out of a **Point** object, the current definition provides no means of getting that data into an object. One approach that is useful in some situations is to export methods that set the values of specific instance variables. Such methods are called **mutators** or, more informally, **setters**. If the **Point** class were to export a **setX** and a **setY** method that allowed the client to change the values of these fields, you could easily replace any application that previously used the **pointT** record with a new version that relies on the new **Point** class. For example, every assignment of the form

```
pt.x = value;
```

would be replaced by a call to

```
pt.setX(value);
```

Similarly, every reference to **pt.y** that wasn't on the left side of an assignment statement would change to **pt.getY()**.

It is, however, a bit unsatisfying to contemplate adding setter methods to a class so soon after deciding that it was important to make its instance variables private. After all, part of the reason for making instance variables private is to ensure that clients don't have unrestricted access to them. Having every instance variable in a class come with a public

getter and setter circumvents those restrictions and therefore eliminates any advantage one might have obtained by making those variables private in the first place. In general, it is considerably safer to allow clients to *read* the values of the instance variables than it is to have clients *change* those values. As a result, setter methods are far less common than getters in object-oriented design.

Many classes, in fact, take the recommendation against allowing change to an even higher level by making it impossible to change the values of any instance variables after an object has been created. Classes designed in this way are said to be **immutable**. For example, the **Point** object as it now stands is immutable because its instance variables are all private and it exports no methods that change the contents of those variables.

Constructors and destructors

When you declare a conventional variable in C++, its value is uninitialized by default. As part of the change to support object-oriented programming, C++ now ensures that declaring a variable used to hold an object value automatically invokes a special method called a **constructor**, which is responsible for initializing the object. A constructor looks very much like an ordinary method definition, but with the following special characteristics:

- The name of the class is used as the constructor name.
- The prototype for the constructor does not specify a return type.
- The body of the constructor does not return a value but instead initializes the object.

As an example, you could declare a constructor in the **Point** class by extending the class definition as follows:

```
class Point {
public:
    Point(int xc, int yc);
    int getX();
    int getY();

private:
    int x, y;
};
```

The implementation of this constructor copies the values of the parameters **xc** and **yc** to the **x** and **y** fields of the **Point** object, like this:

```
Point::Point(int xc, int yc) {
    x = xc;
    y = yc;
}
```

Given this definition, clients can create new objects of type **Point** by calling its constructor with the appropriate arguments. For example, the declaration

```
Point origin(0, 0);
```

initializes a new **Point** object named **origin** to the coordinates (0, 0).

Having the constructor change the values of the instance variables does not violate the condition of immutability, which disallows changes *after* the object is created. Immutable types will typically have a constructor that initializes their state, after which the client simply uses that value as an integral, unchangeable whole.

Classes may also define a **destructor**, which is a method invoked whenever the memory for the object is reclaimed. The destructor handles any cleanup that is necessary when an instance is deleted. The destructor has the same name as the class but preceded by a tilde. The destructor may not take arguments and has no return type, so that the destructor for the **Point** class (if for some reason we decided we needed one) would have the following declaration

```
~Point();
```

Given the definition of the class as it exists at the moment, there isn't any obvious justification for giving **Point** a destructor. By far the most common use for a destructor is to free any memory stored within the object that has been allocated on the heap. If you don't free that memory at the time the object is released, there will be no way for C++ to reclaim it. The situation is different in more modern higher-level languages such as Java. Those languages include a mechanism that automatically searches through memory to find and reclaim heap storage that is no longer accessible. Such a facility is called a **garbage collector**. C++ does not have a garbage collector and therefore forces its programmers to manage memory explicitly.

Fortunately, the ability to define constructors and destructors for classes makes memory management much easier than it would be without those tools. If the only heap memory you use is allocated within the implementation of a class and the destructor can free all that memory when the object disappears, the client need not worry about the details of memory management within the implementation.

The keyword **this**

Conceptually, the constructor for the **Point** class described in the preceding section takes two coordinate values, *x* and *y*. Those variables, however, were named **xc** and **yc** when they appeared in the parameter list. Students are often tempted to use the names **x** and **y** for the parameters and to write the constructor like this:

```
Point::Point(int x, int y) {  
    x = x;  
    y = y;  
}
```



The assignments copy the parameters to themselves.

The problem here is that the instance variables are also named **x** and **y**, which makes these names somewhat ambiguous. Like most other languages, C++ resolves the ambiguity by assuming that a name corresponds to the variable declared in the nearest enclosing scope. The parameter names, which are declared as part of this method, take precedence over the instance variables, which are declared as part of the class. The assignment statements in the buggy version of the constructor end up having no effect at all, because both sides of the equal sign refer to the same variable. Having one variable hide an identically named variable in some larger scope is called **shadowing**.

It isn't always easy or informative to conjure up new names just to avoid the shadowing problem. In many cases—such as this example with the names **x** and **y**—there is one name that seems particularly natural, along with a range of other possibilities that seem cumbersome or arcane. You might argue that **xc** and **yc** stand for *x*-coordinate and *y*-coordinate, but it's a stretch. From the client's point of view, **x** and **y** are precisely the right names for the constructor parameters. From the perspective of the implementer, however, **x** and **y** are the perfect names for the instance variables.

Fortunately, it is possible to please both the client and the implementer at the same time by using the special C++ keyword **this**, which is a pointer to the current object. If

you intend to refer to an instance variable, you can start with the pointer to the current object and use the `->` operator to dereference the pointer and select the instance variable at the same time. The following code uses this keyword to implement the constructor with a single set of names for the parameters and their corresponding instance variables:

```
Point::Point(int x, int y) {
    this->x = x;
    this->y = y;
}
```

Although it would be legal to use `this` in every reference to the current object, C++ programmers tend to use it only when it helps resolve an ambiguity.

9.2 Implementing a specialized version of the **Stack** class

The easiest way to get a more detailed sense of how to write classes is to implement one of the container classes that were introduced in Chapter 4. You already know how those classes work, so you can focus on the implementation without being unsure as to how the class operates from the client's perspective.

In many ways, the simplest container class to implement is the **Stack** class. When you learned about stacks in Chapter 4, you discovered that they came up in several contexts in computer science. The process of calling and returning from functions uses a stack-like discipline because the last function you called is always the last function from which you return—a clear example of the last-in/first-out behavior that defines the stack abstraction. Similarly, you also learned in Chapter 4 that some electronic calculators use a stack to keep track of intermediate results during the calculation. Finally, if you worked through exercise 8 in Chapter 4, you got some insight into a compiler might use a stack to test whether parentheses are balanced in an expression. Each of these stacks has a different base type. The calculator uses a `Stack<double>` while the parenthesis balancing program used a `Stack<char>`. (The function-calling process was defined abstractly and therefore never got around to defining a concrete representation. If you were to simulate the function-calling process, you would need to use something like `Stack<frameT>`, where `frameT` is a structure type that records the contents of a stack frame.) C++'s template mechanism made it very easy to define a single `stack.h` interface that was capable of working with stacks of any of these types.

Now that you've come to the implementation side of things, you need to learn about two different aspects of class design. First, you need to understand how classes and objects work so that you can define your own classes. And if you want to see how the library version of the **Stack** class is implemented, you also need to learn how the C++ `template` keyword operates. As a matter of finding a successful strategy for learning this material, it is wisest to learn these concepts separately rather than all at once. The `template` mechanism is described in detail in Chapter 11, making it possible to focus in this chapter on how classes work before you introduce polymorphism into the equation. The immediate effect of this decision is that the **Stack** class in this chapter will have to choose a particular element type and be done with it. For the purpose of illustration (and because we'll use it in Chapter 10), this section defines the class `CharStack` in which the values pushed on the stack are characters.

Defining the `CharStack` interface

Whenever you define a class for other to use, it makes sense to separate as much as possible the interface for that class from the underlying implementation. In C++, the usual strategy is to put the class definition itself in an interface file with the extension `.h` and the implementation in the corresponding `.c` file. Figure 9-1 shows the `charstack.h`

Figure 9-1 Interface for the CharStack class

```
/*
 * File: charstack.h
 * -----
 * This interface defines the CharStack class, which implements
 * the stack abstraction for characters.
 */

#ifndef _charstack_h
#define _charstack_h

/*
 * Class: CharStack
 * -----
 * This interface defines a class that models a stack of characters.
 * Characters are added and removed only from the top of the stack.
 * The fundamental stack operations are push (add to top) and pop
 * (remove from top).
 */

class CharStack {
public:

/*
 * Constructor: CharStack
 * Usage: CharStack cstk;
 * -----
 * Initializes a new empty stack that can contain characters.
 */

    CharStack();

/*
 * Destructor: ~CharStack
 * Usage: (usually implicit)
 * -----
 * Deallocates storage associated with this cstk. This method is
 * called whenever a CharStack instance variable is deallocated.
 */

    ~CharStack();

/*
 * Method: size
 * Usage: nElems = cstk.size();
 * -----
 * Returns the number of characters in this stack.
 */

    int size();
};
```



```
/*
 * Method: isEmpty
 * Usage: if (cstk.isEmpty()) . . .
 * -----
 * Returns true if this stack contains no characters, and false
 * otherwise.
 */

    bool isEmpty();

/*
 * Method: clear
 * Usage: cstk.clear();
 * -----
 * This method removes all characters from this stack.
 */

    void clear();

/*
 * Method: push
 * Usage: cstk.push(ch);
 * -----
 * Pushes the character ch onto this stack.
 */

    void push(char ch);

/*
 * Method: pop
 * Usage: topChar = cstk.pop();
 * -----
 * Removes the top character from this stack and returns it.
 */

    char pop();

/*
 * Method: peek
 * Usage: topChar = cstk.peek();
 * -----
 * Returns the value of top character from this stack without
 * removing it.  Raises an error if called on an empty stack.
 */

    char peek();

private:

    // Fill this in when the data structure design is complete

};

#endif
```

interface, which includes the definition of the **CharStack** class. The contents of the interface file are primarily comments, which is appropriate given the fact that the intended audience for the interface file is primarily the client. The comments let the client know exactly how each of the methods works, what the arguments need to be, what the result type is, and any special considerations that the client should take into account. This information, moreover, is provided in two forms: the comments and the method prototypes. By contrast, the interface contains no information at all—or at least not yet—about the details of the underlying implementation.

Given the design of C++, however, it is difficult to keep all of these details away from the client. Syntactically, both the **public** and **private** sections of the class must be enclosed within the class body. As things stand now, the **private** section contains none of the implementation details, but that is only because they have yet to be worked out. The **private** section at the moment consists of the comment:

```
// Fill this in when the data structure design is complete
```

In the fullness of time, that section will contain declarations of the private instance variables along with the prototypes for any private methods used by the class.

The fact that C++ interface files typically do contain information that is relevant only to the implementation brings up an interesting question about the meaning of *visibility*. In a technical sense, instance variables that are defined in the **private** section are not visible to the client, because the compiler will prevent the user from using those variables as part of a program. Under the more conventional interpretation of that word, however, everything in the **private** section is visible in the sense that anyone reading the interface file can see that part of the code. This commonsense interpretation of visibility is important as well. Even though the client cannot access a private variable, having all that information right there in the file can create confusion and misimpressions about the behavior of the class. It would be better if this information were hidden a little more effectively.

Given the fact that the compiler has to be able to see the private data, there is probably no foolproof way to prevent a determined client from seeing the contents of the **private** section. Even so, it probably makes sense to avoid sticking all that information in the client's face. To do so, the interface files in this book tend to replace the **private** section with an **#include** line that reads in the private data from another file. For example, in the case of **charstack.h**, the final version of the interface will replace the comment with the line

```
#include "cstkpriv.h"
```

which asks the compiler to read in the contents of the file **cstkpriv.h** at that point in the program. Anyone reading the interface could go off and look at it, but in practice no one bothers. The information you need as a client is in the **public** section, and you have no interest in being bothered with these details.

Representing the stack data

Even though the client may not be interested in the details of the representation, those questions are front and center in the mind of the implementer. If your job is to build the **CharStack** class, you've got to figure out what data structures you can use to hold on to the values stored on the stack and how you can ensure that the last-in/first-out discipline is maintained.

Given that you've been working with the collection classes ever since Chapter 4, the idea that is likely to jump to mind first is that you could store the elements of the stack in a `Vector<char>`. Indeed you could, and you will have a chance to do so in exercise 1. That strategy, however, defeats the primary purpose of this chapter, which is to give you insight into how the classes you've been using work on the inside. The `Vector` class is even more complicated than the `Stack` class, so all you've manage to do is hide the mystery under an even bigger stone. The goal here is to implement the `CharStack` class from scratch, using only the primitive data structures that are built into C++. Vectors are too advanced. To show how things work at a lower level, it is necessary to use arrays.

The simplest low-level approach is to use an array to hold the elements currently on the stack and then to specify a count indicating the number of elements currently in the stack. This strategy suggests the following design for the `private` section of the class:

```
private:
    static const int MAX_STACK_SIZE = 100;

    char elements[MAX_STACK_SIZE];
    int count;
```

As elements are pushed on the stack, you store the new element in the next available space in the array and increment the count. As elements are popped, you decrement the count and return the top element. Since all operations happen at the end of the array, the execution cost of the stack operations is independent of the number of elements already on the stack. Thus, the stack operations `push` and `pop` each run in constant time.

Given this data structure, it is not at all difficult to write the implementation for the `CharStack` class. A complete version of the `charstack.cpp` implementation appears in Figure 9-2.

Figure 9-2 Implementation of the CharStack class

```
/*
 * File: charstack.cpp
 * -----
 * This file implements the CharStack class.
 */

#include "genlib.h"
#include "charstack.h"

/*
 * Implementation notes: CharStack constructor and destructor
 * -----
 * These methods have little work to do because the fixed-size
 * array is allocated on the stack and not in the heap.
 */

CharStack::CharStack() {
    count = 0;
}

CharStack::~CharStack() {
    /* Empty */
}
```

```

/*
 * Implementation notes: size, isEmpty, clear
 * -----
 * These implementations should be self-explanatory.
 */

int CharStack::size() {
    return count;
}

bool CharStack::isEmpty() {
    return count == 0;
}

void CharStack::clear() {
    count = 0;
}

/*
 * Implementation notes: push, pop, peek
 * -----
 * The only subtlety in this implementation is the use of
 * the ++ and -- to select the correct index in the stack
 * and adjust the count variable at the same time.
 */

void CharStack::push(char ch) {
    if (count == MAX_STACK_SIZE) Error("push: Stack is full");
    elements[count++] = ch;
}

char CharStack::pop() {
    if (isEmpty()) Error("pop: Attempting to pop an empty stack");
    return elements[--count];
}

char CharStack::peek() {
    if (isEmpty()) Error("peek: Attempting to peek at an empty
stack");
    return elements[count - 1];
}

```

The advantages of object encapsulation

To beginning programmers, defining a one-line method like `size` sometimes seems wasteful. If the data members had been declared public in the interface, you could eliminate this method entirely and simply have the client write

```
stack.count
```

instead of

```
stack.size()
```

Given that the first form is a bit shorter and avoids the overhead of a function call, it might seem that removing this method would simplify both the implementation and the

client code. This argument, however, is shortsighted and fails to give sufficient weight to the importance of abstraction boundaries to good software engineering.

The expression

stack.count

represents a violation of the **CharStack** abstraction. As a client, you shouldn't know what fields exist in the record that lies underneath the variable **stack**. If clients use the name **count** in their programs, those programs now depend on the specific structure of the **CharStack** representation. If someone wanted to change that implementation in the future—something that the implementer should feel free to do as long as the abstract interface remains unchanged—all the programs that relied on this piece of “forbidden knowledge” would break. By using the

stack.size()

form, you provide a layer of insulation between the client and the implementation. An implementer's decision to change the representation should not affect clients of the **size** method at all, as long as the implementer guarantees to preserve its behavior as advertised in the interface. Change is a fact of life in software engineering, and it is always best to prepare for it as well as you can.

Removing the maximum size limitation

One useful change that you might decide to make in the stack implementation is to remove the arbitrary limitation on the size of the array. The easiest way to do so is to make the array dynamic and reallocate space for a new, larger array whenever space in the old array is exhausted. You therefore need to change the data representation so that it provides a pointer to a dynamic array whose elements are of type **char**. In addition, because the size of the array is no longer constant, the structure will need a variable called **capacity** to keep track of the allocated size. These changes do not affect the **public** section of the class, but does change the **private** section, which is now stored in the **cstkpriv.h** file. Figure 9-3 shows the new contents of that file.

Figure 9-3 Contents of the **cstkpriv.h** file

```
/*
 * File: cstkpriv.h
 * -----
 * This file contains the private data for the CharStack class.
 */

/* Constant defining the initial capacity */
static const int INITIAL_CAPACITY = 100;

/* Data fields required for the stack representation */
char *elements;      /* Dynamic array of characters */
int capacity;        /* Allocated size of that array */
int count;           /* Current count of chars pushed */

/* Prototype for a private helper method to expand the array */
void expandCapacity();
```

This change in representation requires a change in the implementation of several methods, starting with the constructor. When initializing a new **CharStack** object, you now need to allocate initial space to the array. If you assume that the initial stack size is given by the constant **INITIAL_CAPACITY**, the updated definition of the constructor is

```
CharStack::CharStack() {
    elements = new char[INITIAL_CAPACITY];
    capacity = INITIAL_CAPACITY;
    count = 0;
}
```

Once you have made this change, you also need to change the definition of the destructor. The new data representation contains internal storage that is allocated from the heap, which needs to be freed when the stack object itself is being deleted. The new definition is then

```
CharStack::~~CharStack() {
    delete[] elements;
}
```

The other major change is the definition of **push**, which must now expand the stack if necessary. Implementing this change is easiest if you define an auxiliary function **expand** to do the actual expansion. If you adopt this approach, the **push** function looks like this:

```
void CharStack::push(char element) {
    if (count == capacity) expandCapacity();
    elements[count++] = element;
}
```

where **expandCapacity** is given by

```
void CharStack::expandCapacity() {
    capacity *= 2;
    char *array = new char[capacity];
    for (int i = 0; i < count; i++) {
        array[i] = elements[i];
    }
    delete[] elements;
    elements = array;
}
```

Here, the **expandCapacity** function simply doubles the size of the stack. It allocates a new array of the larger size, copies items from the old array into the new one, frees the storage occupied by the old array, and then stores the pointer to the new array in the stack structure.

Making these changes to the implementation requires you to modify the definition of the data members used for the **CharStack** class. Such changes are feasible only if the type is private to the implementation and not part of the public interface. If the data members are publicly exposed, some client will almost certainly write code that depends on a specific detail of those fields. Because this client's program would then be invalidated by the change in representation, the process of upgrading the interface would be much more difficult.

Object copying and the `DISALLOW_COPY` macro

One subtle, but critical, problem that arises from changing the representation from a fixed array to a dynamic one is that object copying will no longer work correctly by default for **CharStack** objects. When assigning one object to another or passing or returning an object by value, a copy of the object is made. Unless its class specifies otherwise, an object is copied using the C++ default, which is a simple member-wise copy of the data members. This is the same default copy operation used for C++ structs. If the data members include pointers, the copied object will end up with pointers copied from the original. This is unlikely to be what you had in mind and can cause surprising results that in turn lead to difficult debugging problems.

As an example, trace through the actions of this code:

```
CharStack first;
first.push('A');
first.push('B');

CharStack second = first;
first.push('C');
second.push('Z');

cout << first.pop() << endl;
```

The first three lines create the **first** object and push the characters 'A' and 'B' onto it. The next line creates the **second** object and assigns it the value of the **first** object. During that assignment, **second** is initialized with a member-by-member copy of **first**. There are three data members in the **CharStack** class: **elements**, which is a pointer to a dynamic array, **count**, which holds the number of elements, and **capacity**, which keeps track of the allocated size of that array. There is nothing tricky about copying the two size fields, but when C++ copies **elements** from one object to another, it copies only the pointer and not the elements in the dynamic array itself. The two objects therefore end up with aliases to the same array—a rather tenuous situation indeed.

After the assignment, the internal state of each stack shows it contains two elements, with values 'A' and 'B'. Now **first** pushes the character 'C'. When **second** pushes the character 'Z', it overwrites that 'C' in the shared array! Thus, when the last line of code prints the value popped from **first**, the resulting 'Z' will come as quite a surprise. Even more troubling is the situation where one of the stacks is deallocated by its destructor and the other stack continues to access the dynamic array that is now deleted.

When an object has at least one data member of pointer type, the default copy operation is unlikely to be appropriate. Typically more is needed than just copying the pointer. A **deep copy** is one that copies what is being pointed to, instead of just copying the pointer itself. The default assignment that only copies pointers is called a **shallow copy**. There is no distinction when the object has no pointer data members, so up until now, this has not been an issue. However, once you have pointers within an object, relying on the default copy operation will undoubtedly lead to undesired results and difficult bugs. A class can supply its own corrected deep copy operation by implementing the copy constructor and overriding the assignment operator. Learning how to implement these operations correctly involves some details beyond the scope of this course. You can, however, protect yourself against inadvertent copying by disallowing object copying for the class altogether. The **DISALLOW_COPYING** macro is inserted into private section of a class interface and is given one argument, which is the name of the class. Thus, if you wanted to prevent copying of **CharStack** objects, you could add the following line to the **cstkpriv.h** file:

DISALLOW_COPYING(CharStack)

This macro expands to declare the necessary copy operations as private and unimplemented, which prevents making any object copies. Attempting to copy an object of this class, via assignment, parameter passing, or function return, will result in a compiler error about those features being inaccessible. Since objects are typically passed by reference, this limitation is not particularly restricting and avoids a lot of grief that would otherwise result from unintended copying.

9.3 Implementing the **Scanner** class

The final example of a class implementation in this chapter is a simplified version of the **Scanner** class introduced in section 4.7. The implementation given here leaves out the following features of the **Scanner** class as it is distributed in the library:

- The ability to scan tokens from an input stream as well as from a file
- The options that allow the scanner to read numbers, quoted strings, and HTML tags
- The **saveToken** method

You will have the opportunity to implement several of these features in the exercises. Note that the **setSpaceOption** method *is* implemented in this version of the scanner so that you have a model for implementing the other option forms.

The revised **scanner.h** interface is shown in Figure 9-4 so that you can see what methods need to be implemented. Figures 9-5 and 9-6 follow immediately after the interface and show the **scanpriv.h** file containing the private data and the **scanner.cpp** file containing the implementation. Most of the implementation is straightforward. The only method that is at all complicated is **nextToken**, but even its logic follows directly from the definition of what constitutes a token.

Summary

In this chapter, you have been introduced to object-oriented programming and learned how to use classes to define several simple abstract data types. Important points in this chapter include:

- A C++ class definition consists primarily of instance variables and methods, which are collectively called *members*.
- Each class definition is divided into a **public** section that is accessible to the client and a **private** section that is not.
- Instance variables can be specified in the **public** section of a class, which makes them accessible to clients just as if they were fields in a record, but it is far more common to put them in the **private** section and use them to maintain the state of the object.
- Classes ordinarily contain a *constructor*, which is invoked to initialize the fields in an object. Many classes also specify a *destructor*, which is called when the object becomes inaccessible. The most common reason to specify a destructor is when a class needs to free memory that was allocated during its operation.
- The **private** section of a class is not accessible to clients, but the information that is there can easily be seen by clients if it is included directly in the interface file. One strategy for hiding the private information is to put the contents of the **private** section in a separate file that is then read into the compilation using the **#include** directive.
- A C++ class provides an ideal basis for implementing simplified versions of both the **Stack** and **Scanner** classes introduced in Chapter 4.

Figure 9-4 The simplified scanner.h interface

```

/*
 * File: scanner.h
 * -----
 * This file is the interface for a class that facilitates dividing
 * a string into logical units called "tokens", which are either
 *
 * 1. Strings of consecutive letters and digits representing words
 * 2. One-character strings representing punctuation or separators
 *
 * To use this class, you must first create an instance of a
 * Scanner object by declaring
 *
 *     Scanner scanner;
 *
 * You initialize the scanner's input stream by calling
 *
 *     scanner.setInput(str);
 *
 * where str is the string from which tokens should be read.
 * Once you have done so, you can then retrieve the next token
 * by making the following call:
 *
 *     token = scanner.nextTok();
 *
 * To determine whether any tokens remain to be read, you can call
 * the predicate method scanner.hasMoreTokens(). The nextToken
 * method returns the empty string after the last token is read.
 *
 * The following code fragment serves as an idiom for processing
 * each token in the string inputString:
 *
 *     Scanner scanner;
 *     scanner.setInput(inputString);
 *     while (scanner.hasMoreTokens()) {
 *         string token = scanner.nextTok();
 *         . . . process the token . . .
 *     }
 *
 * This version of the Scanner class includes an option for skipping
 * whitespace characters, which is described in the comments for the
 * setSpaceOption method.
 */

#ifndef _scanner_h
#define _scanner_h

#include "genlib.h"

/*
 * Class: Scanner
 * -----
 * This class is used to represent a single instance of a scanner.
 */

class Scanner {

```

```
public:

/*
 * Constructor: Scanner
 * Usage: Scanner scanner;
 * -----
 * The constructor initializes a new scanner object. The scanner
 * starts empty, with no input to scan.
 */

    Scanner();

/*
 * Destructor: ~Scanner
 * Usage: usually implicit
 * -----
 * The destructor deallocates any memory associated with this scanner.
 */

    ~Scanner();

/*
 * Method: setInput
 * Usage: scanner.setInput(str);
 * -----
 * This method configures this scanner to start extracting
 * tokens from the input string str. Any previous input string is
 * discarded.
 */

    void setInput(string str);

/*
 * Method: nextToken
 * Usage: token = scanner.nextToken();
 * -----
 * This method returns the next token from this scanner. If
 * nextToken is called when no tokens are available, it returns the
 * empty string.
 */

    string nextToken();

/*
 * Method: hasMoreTokens
 * Usage: if (scanner.hasMoreTokens()) . . .
 * -----
 * This method returns true as long as there are additional
 * tokens for this scanner to read.
 */

    bool hasMoreTokens();
```

```

/*
 * Methods: setSpaceOption, getSpaceOption
 * Usage: scanner.setSpaceOption(option);
 *         option = scanner.getSpaceOption();
 * -----
 * This method controls whether this scanner
 * ignores whitespace characters or treats them as valid tokens.
 * By default, the nextToken function treats whitespace characters,
 * such as spaces and tabs, just like any other punctuation mark.
 * If, however, you call
 *
 *     scanner.setSpaceOption(Scanner::IgnoreSpaces);
 *
 * the scanner will skip over any white space before reading a
 * token. You can restore the original behavior by calling
 *
 *     scanner.setSpaceOption(Scanner::PreserveSpaces);
 *
 * The getSpaceOption function returns the current setting
 * of this option.
 */

enum spaceOptionT { PreserveSpaces, IgnoreSpaces };

void setSpaceOption(spaceOptionT option);
spaceOptionT getSpaceOption();

private:
#include "scanpriv.h"
};

#endif

```

Figure 9-5 Contents of the scanpriv.h file

```

/*
 * File: scanpriv.h
 * -----
 * This file contains the private data for the simplified version
 * of the Scanner class.
 */

/* Instance variables */

string buffer;           /* The string containing the tokens */
int len;                 /* The buffer length, for efficiency */
int cp;                  /* The current index in the buffer */
spaceOptionT spaceOption; /* Setting of the space option */

/* Private method prototypes */

void skipSpaces();
int scanToEndOfIdentifier();

```

Figure 9-6 The `scanner.cpp` implementation

```
/*
 * File: scanner.cpp
 * -----
 * Implementation for the simplified Scanner class.
 */

#include "genlib.h"
#include "scanner.h"
#include <cctype>
#include <iostream>

/*
 * The details of the representation are inaccessible to the client,
 * but consist of the following fields:
 *
 * buffer          -- String passed to setInput
 * len             -- Length of buffer, saved for efficiency
 * cp              -- Current character position in the buffer
 * spaceOption     -- Setting of the space option extension
 */

Scanner::Scanner() {
    buffer = "";
    spaceOption = PreserveSpaces;
}

Scanner::~Scanner() {
    /* Empty */
}

void Scanner::setInput(string str) {
    buffer = str;
    len = buffer.length();
    cp = 0;
}

/*
 * Implementation notes: nextToken
 * -----
 * The code for nextToken follows from the definition of a token.
 */

string Scanner::nextToken() {
    if (cp == -1) {
        Error("setInput has not been called");
    }
    if (spaceOption == IgnoreSpaces) skipSpaces();
    int start = cp;
    if (start >= len) return "";
    if (isalnum(buffer[cp])) {
        int finish = scanToEndOfIdentifier();
        return buffer.substr(start, finish - start + 1);
    }
    cp++;
    return buffer.substr(start, 1);
}
```

```

bool Scanner::hasMoreTokens() {
    if (cp == -1) {
        Error("setInput has not been called");
    }
    if (spaceOption == IgnoreSpaces) skipSpaces();
    return (cp < len);
}

void Scanner::setSpaceOption(spaceOptionT option) {
    spaceOption = option;
}

Scanner::spaceOptionT Scanner::getSpaceOption() {
    return spaceOption;
}

/* Private functions */

/*
 * Private method: skipSpaces
 * Usage: skipSpaces();
 * -----
 * This function advances the position of the scanner until the
 * current character is not a whitespace character.
 */

void Scanner::skipSpaces() {
    while (cp < len && isspace(buffer[cp])) {
        cp++;
    }
}

/*
 * Private method: scanToEndOfIdentifier
 * Usage: finish = scanToEndOfIdentifier();
 * -----
 * This function advances the position of the scanner until it
 * reaches the end of a sequence of letters or digits that make
 * up an identifier. The return value is the index of the last
 * character in the identifier; the value of the stored index
 * cp is the first character after that.
 */

int Scanner::scanToEndOfIdentifier() {
    while (cp < len && isalnum(buffer[cp])) {
        cp++;
    }
    return cp - 1;
}

```

Review questions

1. Define each of the following terms: *object*, *class*, *data member*, *method*.
2. In a C++ class declaration, what do the keywords **public** and **private** mean?
3. In C++, what does the keyword **this** signify?
4. What does it mean for a class to be *immutable*?
5. What is the syntax for a C++ constructor?
6. Suppose that you have a constructor for a class called **Interval** that contains two private instance variables called **low** and **high**, each of which is of type **double**. Write a constructor for the **Interval** class that takes two arguments, also called **low** and **high**, and initializes the instance variables from these parameters.
7. When is it particularly important to specify a destructor as part of a C++ class?
8. True or false: There can be several different constructors in a class, each with its own argument pattern, but there can be only one destructor.
9. True or false: C++ programs periodically scan through memory to find and reclaim memory space that is no longer accessible.
10. Why is it bad programming practice for a client programmer to use the expression

stack.count

to determine the number of items in a stack? If you use the **CharStack** class interface shown in Figure 9-1, is it possible for the client to make this mistake?

11. What technique was used in this chapter to reduce the likelihood that clients will see the contents of the **private** section of a class?
12. What strategy was used in this chapter to eliminate the limitation on the maximum stack size?
13. What is the default C++ implementation for copying an object? In what situations is the default copy behavior inappropriate?
14. What is the purpose of the **DISALLOW_COPY** macro? How do you use it?
15. Why is it usually better to maintain state information in a class than to keep it in global variables within a module?

Programming exercises

1. Reimplement the **CharStack** class using a **Vector<char>** instead of an array for its internal storage.
2. Using the **CharStack** class as a model, create a **NumStack** class that supports stack operations for data of type **double**. Use this class to reimplement the RPN calculator introduced in Chapter 4.
3. Write a complete definition for a class called **Rational** that implements basic operations on rational numbers, which are represented internally as a pair of **long** integers. Your class should implement the public methods shown in Figure 9-7. All data members should be private to the class.

Figure 9-7 Methods to be implemented for the Rational class

Rational (<i>n</i> , <i>d</i>)	The constructor for the Rational class creates a rational number with <i>n</i> and <i>d</i> as its numerator and denominator, respectively. Your implementation should always reduce the internal fraction to lowest terms and show the sign of the value in the numerator. The denominator of the internal value should always be positive.
<i>r</i> ₁ . add (<i>r</i> ₂) <i>r</i> ₁ . sub (<i>r</i> ₂) <i>r</i> ₁ . mul (<i>r</i> ₂) <i>r</i> ₁ . div (<i>r</i> ₂)	These methods compute a new rational number which is the result of applying the indicated arithmetic operation to the rational numbers <i>r</i> ₁ and <i>r</i> ₂ . The results of these operations are given by the following formulae: $\frac{num_1}{den_1} + \frac{num_2}{den_2} = \frac{num_1 \times den_2 + num_2 \times den_1}{den_1 \times den_2}$ $\frac{num_1}{den_1} - \frac{num_2}{den_2} = \frac{num_1 \times den_2 - num_2 \times den_1}{den_1 \times den_2}$ $\frac{num_1}{den_1} \times \frac{num_2}{den_2} = \frac{num_1 \times num_2}{den_1 \times den_2}$ $\frac{num_1}{den_1} / \frac{num_2}{den_2} = \frac{num_1 \times den_2}{den_1 \times num_2}$
<i>r</i> ₁ . equals (<i>r</i> ₂)	This method returns true if <i>r</i> ₁ and <i>r</i> ₂ are equal.
<i>r</i> ₁ . compareTo (<i>r</i> ₂)	This method returns an integer whose sign reflects the relationship between <i>r</i> ₁ and <i>r</i> ₂ in the manner of comparison functions.
<i>r</i> . toString ()	This method returns a string with the format <i>num/den</i> where <i>num</i> and <i>den</i> are the numerator and denominator of <i>r</i> . If the denominator is 1, the string value should contain only the numerator expressed as an integer.

4. As a test of your implementation of the **Rational** class in exercise 3, reimplement the RPN calculator so that it performs its internal calculations using rational instead of floating-point numbers. For example, your program should be able to produce the following sample run (which demonstrates that rational arithmetic is always exact):

```

Rational RPN Calculator
> 1
> 2
> /
1 / 2
> 1
> 3
> /
1 / 3
> 1
> 6
> /
1 / 6
> +
1 / 2
> +
1
> Q

```

To implement this program, you will need to define a new version of the stack class that stores **Rational** values.

- For certain applications, it is useful to be able to generate a series of names that form a sequential pattern. For example, if you were writing a program to number figures in a paper, having some mechanism to return the sequence of strings "**Figure 1**", "**Figure 2**", "**Figure 3**", and so on, would be very handy. However, you might also need to label points in a geometric diagram, in which case you would want a similar but independent set of labels for points such as "**P0**", "**P1**", "**P2**", and so forth.

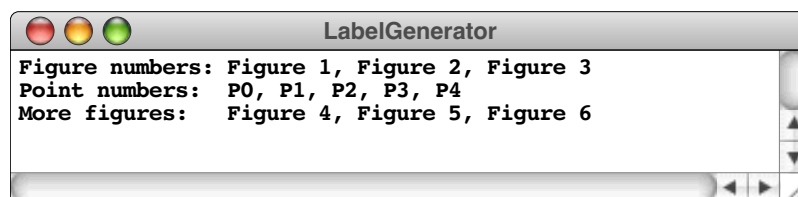
If you think about this problem more generally, the tool you need is a label generator that allows the client to define arbitrary sequences of labels, each of which consists of a prefix string ("**Figure**" or "**P**" for the examples in the preceding paragraph) coupled with an integer used as a sequence number. Because the client may want different sequences to be active simultaneously, it makes sense to define the label generator as an abstract type called **LabelGenerator**. To initialize a new generator, the client provides the prefix string and the initial index as arguments to the **LabelGenerator** constructor. Once the generator has been created, the client can return new labels in the sequence by calling **nextLabel** on the **LabelGenerator**.

As an illustration of how the interface works, the program

```
int main() {
    LabelGenerator figureNumbers("Figure", 1);
    LabelGenerator pointNumbers("P", 0);

    cout << "Figure numbers: ";
    for (int i = 0; i < 3; i++) {
        if (i > 0) cout << ", ";
        cout << figureNumbers.nextLabel();
    }
    cout << endl << "Point numbers: ";
    for (int i = 0; i < 5; i++) {
        if (i > 0) cout << ", ";
        cout << pointNumbers.nextLabel();
    }
    cout << endl << "More figures: ";
    for (int i = 0; i < 3; i++) {
        if (i > 0) cout << ", ";
        cout << figureNumbers.nextLabel();
    }
    cout << endl;
    return 0;
}
```

produces the following sample run:



Write the files **labelgen.h** and **labelgen.cpp** that define and implement this class.

- Add the necessary code to the scanner package to implement the **setStringOption** extension, which allows the scanner to read quoted strings as a single token. By

default, the scanner should continue to treat double-quotation marks just like any other punctuation mark. However, if the client calls

```
scanner.setStringOption(Scanner::ScanQuotesAsStrings);
```

the scanner will instead recognize a quoted string as a single unit. For example, the input line

```
"Hello, world."
```

would be scanned as the following single token:

```
\x("Hello, world.")
```

Note that the quotation marks are preserved as part of the token so that the client can differentiate a string token from other token types.

As an extra challenge, extend the string-recognition code in the scanner so that the scanner correctly interprets the standard escape sequences from ANSI C++, such as `\n` and `\t`. A complete list of these escape sequences appears in Table 1-2.

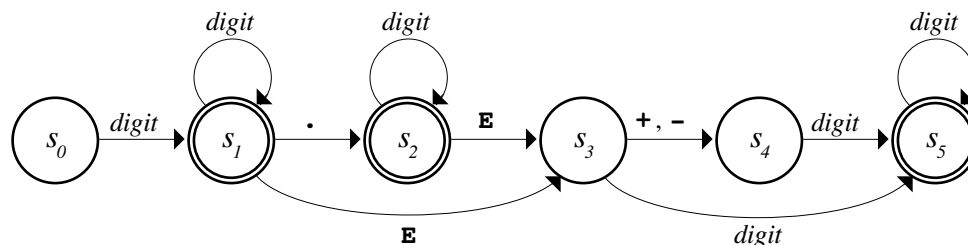
7. Add the necessary code to the scanner package to implement the **setNumberOption** extension, which allows the scanner to read numbers as a single token. Note that this option has two different forms. If the option **ScanNumbersAsIntegers** is set, the package will divide the string **"3.14"** into three tokens, as follows:

```
3 . 14
```

If the **ScanNumbersAsReals** option is set instead, the same string would be scanned as a single token:

```
3.14
```

The process of scanning to the end of an integer is reasonably straightforward and follows the structure of the existing scanner code. The only difference is that a string like **"3rd"** that begins with a digit but also includes letters will be read as one token by the original scanner but as two if the **ScanNumbersAsIntegers** option is set. The hard part is implementing the **ScanNumbersAsReals** option. The specification for a legal real number is given in the comments in Figure 9-8. It is often easier, however, to understand the idea of scanning a number in terms of a structure that computer scientists call a **finite-state machine**, which is usually represented diagrammatically as a collection of circles representing the possible states of the machine. The circles are then connected by a set of labeled arcs that indicate how the process moves from one state to another. A finite-state machine for scanning a real number, for example, looks like this:



You start scanning a number in state s_0 and then follow the labeled arcs for each character in the input until there is no arc that matches the input character. If you

end up in a state marked by a double circle, you have successfully scanned a number. These states are called **final states**.

To illustrate the operation of this finite-state machine, suppose that the scanner string is

3.14-17.5E-9

and that the number option has been set to **ScanNumbersAsReals**. The intended result is that the scanner divide this string into two numbers and an operator as follows:

3.14 **-** **17.5E-9**

When the scanner encounters the digit **3** at the beginning of the input, it starts the numeric scanner in state s_0 . From there, it makes the following state transitions:

State	Input
s_0	3
s_1	.
s_2	1
s_2	4
s_2	-

Because there is no transition from state s_2 labeled with a minus sign and because s_2 is a final state, the process ends at this point, having successfully recognized the numeric string "**3.14**" as a token. The next token is simply the minus sign, which is returned as a single-character string. The final token on the line is scanned by making the following transitions:

State	Input
s_0	1
s_1	7
s_1	.
s_2	5
s_2	E
s_3	-
s_4	9
s_5	<i>none</i>

Chapter 10

Efficiency and Data Representation

*Time granted does not necessarily coincide with time that
can be most fully used.*

— Tillie Olsen, *Silences*, 1965

This chapter brings together two ideas that might at first seem to have little to do with each other: the notion of algorithmic efficiency presented in Chapter 8 and the idea of classes from Chapter 9. Up to now, efficiency has been closely linked with the study of algorithms. If you choose a more efficient algorithm, you can reduce the running time of a program substantially, particularly if the new algorithm is in a different complexity class. In some cases, choosing a different underlying representation for a class can have an equally dramatic effect. To illustrate this idea, this chapter looks at a specific class that can be represented in several different ways and contrasts the efficiency of those representations.

10.1 The concept of an editor buffer

Whenever you create a source file or use a word processor to edit text, you are using a piece of software called an **editor**, which allows you to make changes to a text file. Internally, an editor maintains a sequence of characters, which is usually called a **buffer**. The editor application allows you to make changes to the contents of the buffer, usually by some combination of the following operations:

- Moving the cursor to the point in the text at which editing needs to take place.
- Typing in new text, which is then inserted at the current cursor position.
- Deleting characters using a delete or backspace key.

Modern editors usually provide a highly sophisticated editing environment, complete with such fancy features as using a mouse to position the cursor or commands that search for a particular text string. Moreover, they tend to show the results of all editing operations precisely as they are performed. Editors that display the current contents of the buffer throughout the editing process are called **wysiwyg** (pronounced “wizzy-wig”) editors, which is an acronym for “what you see is what you get.” Such editors are very easy to use, but their advanced features sometimes make it harder to see how an editor works on the inside.

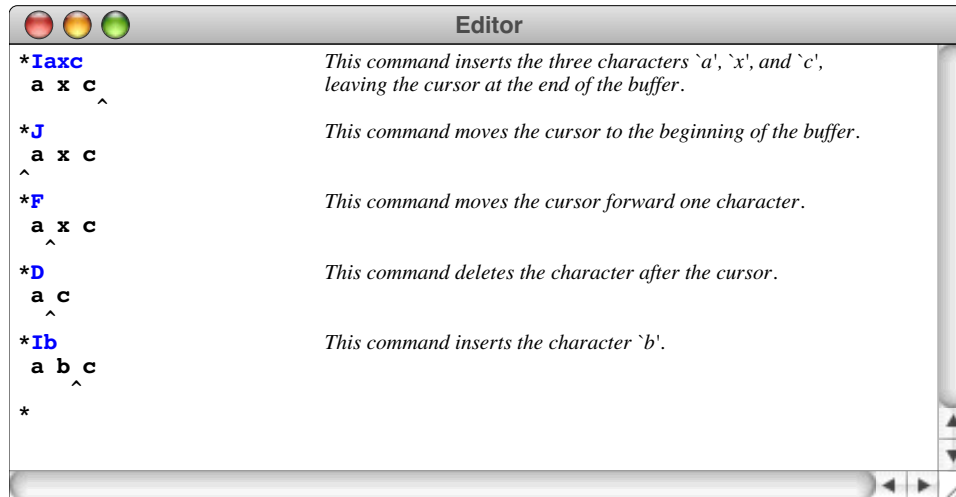
In the early days of computing, editors were much simpler. Lacking access to a mouse or a sophisticated graphics display, editors were designed to respond to commands entered on the keyboard. For example, with a typical keyboard-based editor, you insert new text by typing the command letter **I**, followed by a sequence of characters. Additional commands perform other editing functions, such as moving the cursor around in the buffer. By entering the right combinations of these commands, you can make any desired set of changes.

To make the idea of the editor buffer as concrete as possible, let’s suppose that your task is to build an editor that can execute the six commands shown in Table 10-1. Except for the **I** command, which also takes the characters to be inserted, every editor command consists of a single letter read in on a line.

Table 10-1 Commands implemented by the editor

Command	Operation
F	Moves the editing cursor forward one character position
B	Moves the editing cursor backward one character position
J	Jumps to the beginning of the buffer (before the first character)
E	Moves the cursor to the end of the buffer (after the last character)
Ixxx	Inserts the characters <i>xxx</i> at the current cursor position
D	Deletes the character just after the current cursor position

The following sample run illustrates the operation of the editor, along with annotations that describe each action. In this session, the user first inserts the characters **axc** and then corrects the contents of the buffer to **abc**. The editor program displays the state of the buffer after each command, marking the position of the cursor with a carat symbol (^) on the next line.



10.2 Defining the buffer abstraction

In creating an editor that can execute the commands from Table 10-1, your main task is to design a data structure that maintains the state of the editor buffer. This data structure must keep track of what characters are in the buffer and where the cursor is. It must also be able to update the buffer contents whenever an editing operation is performed. In other words, what you want to do is define a new abstraction that represents the editor buffer.

Even at this early stage, you probably have some ideas about what internal data structures might be appropriate. Because the buffer is clearly an ordered sequence of characters, one seemingly obvious choice is to use a **string** or a **Vector<char>** as the underlying representation. As long as you have these classes available, either would be an appropriate choice. The goal of this chapter, however, is to understand how the choice of representation affects the efficiency of applications. That point is difficult to make if you use higher-level structures like **string** and **Vector** because the inner workings of those classes are not visible to clients. If you choose instead to limit your implementation to the built-in data structures, every operation becomes visible, and it is therefore much easier to determine the relative efficiency of various competing designs. That logic suggests using a character array as the underlying, because array operations have no hidden costs.

Although using an array to represent the buffer is certainly a reasonable approach to the problem, there are other representations that offer interesting possibilities. The fundamental lesson in this chapter—and indeed in much of this book—is that you should not be so quick to choose a particular representation. In the case of the editor buffer, arrays are only one of several options, each of which has certain advantages and disadvantages. After evaluating the tradeoffs, you might decide to use one strategy in a certain set of circumstances and a different strategy in another. At the same time, it is important to note that, no matter what representation you choose, the editor must always be able to perform the same set of commands. Thus, the external behavior of an editor buffer must remain the same, even if the underlying representation changes.

In order to make your interface as flexible as possible, it makes sense to define a new class to represent an editor buffer. The principal advantage of using a class in this context is that doing so allows you to separate the specification of behavior and representation. Because you understand the operations to which it must respond, you already know how an editor buffer behaves. In the **buffer.h** interface, you define the **EditorBuffer** class whose public interface provides the required set of operations, while the data representation is kept private. Clients work entirely with **EditorBuffer** objects through their public interface without any access to the underlying private data representation. That fact, in turn, leaves you free to change that representation without requiring your clients to make any changes in their programs.

The public interface of the **EditorBuffer** class

The public interface consists of prototypes for the methods that implement the primitive operations on an editor buffer. What operations do you need to define? If nothing else, you need functions for each of the six editor commands. As with any class, however, you will also need to define a constructor that initializes a new buffer. Given the class name of **EditorBuffer**, the prototype for the constructor would be

```
EditorBuffer()
```

The class should also provide the destructor

```
~EditorBuffer();
```

which relinquishes its storage.

Once these operators are out of the way, the next step is to define the prototypes for the editing commands. To move the cursor forward, for example, you could define a function like

```
void moveCursorForward();
```

As you design the interface, it is important to keep in mind that you are not concerned with how such an operation is performed or with how the buffer and its cursor are represented. The **moveCursorForward** function is defined entirely in terms of its abstract effect.

The last method you need in order to create a simple editor application is one that displays the contents of the buffer. Because the implementation alone has access to the underlying structure, this operation cannot be implemented by the client and must instead be exported by the interface as a method with the following prototype:

```
void display();
```

Even if an application does not need to display the value of an abstract data type, display functions of this sort are often extremely useful when you are debugging the application. As you search for the problem, you can use the display function to trace the operation of the program as it runs. Thus, it is generally a good idea to include display functions in the interfaces you design.

The public section of the **EditorBuffer** class interface is shown in Figure 10-1. Note that the private section is included from a file called **bufpriv.h** so that its contents don't clutter up the interface and cause confusion for the client.

Figure 10-1 Interface for the editor buffer class

```
/*
 * File: buffer.h
 * -----
 * This file defines the interface for an editor buffer class.
 */

#ifndef _buffer_h
#define _buffer_h

/*
 * Class: EditorBuffer
 * -----
 * This class defines the class used to represent
 * an editor buffer.
 */

class EditorBuffer {
public:

/*
 * Constructor: EditorBuffer
 * Usage: EditorBuffer buffer;
 *        EditorBuffer *bp = new EditorBuffer();
 * -----
 * The constructor initializes a newly allocated buffer
 * to represent an empty buffer.
 */

    EditorBuffer();

/*
 * Destructor: ~EditorBuffer
 * Usage: (usually implicit)
 * -----
 * The destructor frees any storage associated with this buffer.
 */

    ~EditorBuffer();

/*
 * Methods: moveCursorForward, moveCursorBackward
 * Usage: buffer.moveCursorForward();
 *        buffer.moveCursorBackward();
 * -----
 * These functions move the cursor forward or backward one
 * character, respectively. If you call moveCursorForward
 * at the end of the buffer or moveCursorBackward at the
 * beginning, the function call has no effect.
 */

    void moveCursorForward();
    void moveCursorBackward();
};
```

```

/*
 * Methods: moveCursorToStart, moveCursorToEnd
 * Usage: buffer.moveCursorToStart();
 *         buffer.moveCursorToEnd();
 * -----
 * These functions move the cursor to the start or the end of this
 * buffer, respectively.
 */

    void moveCursorToStart();
    void moveCursorToEnd();

/*
 * Method: insertCharacter
 * Usage: buffer.insertCharacter(ch);
 * -----
 * This function inserts the single character ch into this
 * buffer at the current cursor position. The cursor is
 * positioned after the inserted character, which allows
 * for consecutive insertions.
 */

    void insertCharacter(char ch);

/*
 * Method: deleteCharacter
 * Usage: buffer.deleteCharacter();
 * -----
 * This function deletes the character immediately after
 * the cursor. If the cursor is at the end of the buffer,
 * this function has no effect.
 */

    void deleteCharacter();

/*
 * Method: display
 * Usage: buffer.display();
 * -----
 * This function displays the current contents of this buffer
 * on the console.
 */

    void display();

private:
#include "bufpriv.h"    /* Read in the private portion of the class */
};

#endif

```


Coding the editor application

Once you have defined the public interface, you are free to go back and write the editor application, even though you have not yet implemented the buffer class or settled on an appropriate internal representation. When you're writing the editor application, the only important consideration is what each of the operations does. At this level, the implementation details are unimportant.

As long as you limit yourself to the six basic commands in Table 10-1, writing the editor program itself is actually quite easy. All the program has to do is construct a new **EditorBuffer** object and then enter a loop in which it reads a series of editor commands. Whenever the user enters a command, the program simply looks at the first character in the command name and performs the requested operation by calling the appropriate method from the buffer interface. The code for the editor appears in Figure 10-2.

10.3 Implementing the editor using arrays

As noted earlier in the section on “Defining the buffer abstraction,” one of the possible representations for the buffer is an array of characters. Although this design is not the only option for representing the editor buffer, it is nonetheless a useful starting point. After all, the characters in the buffer form an ordered, homogeneous sequence, which is precisely the sort of data that arrays are intended to represent.

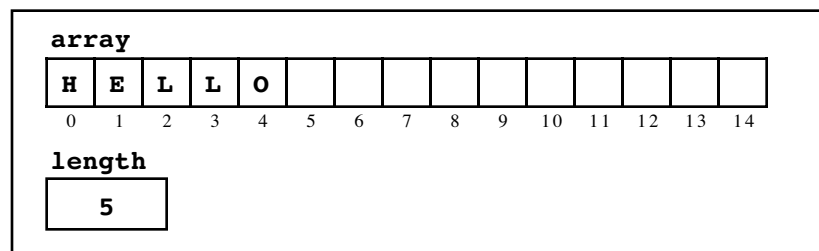
Defining the private data representation

Representing the editor buffer as an array is simple, but not necessarily as simple as it might at first appear. There are a couple of details to consider, and it is worth spending a little time to get them right.

In terms of the characters themselves, the simplest solution is to store the characters from the buffer in consecutive elements of the array. Another design choice that will simplify operations is to allocate the array to a fixed size. This array will often be partially filled and thus the buffer will store the current length in a separate field. Thus, a buffer object containing the characters

H E L L O

would be represented using the following data members:



In addition to these data members, a buffer object must also contain an indication of the current location of the cursor. Because the cursor is essentially a character position, you can represent it by including another integer data member called **cursor** in the **EditorBuffer** class. If you look at the display screen, however, you will realize that the cursor does not actually appear at a character position but instead sits between two characters. Thus, you have to establish a convention relating the index position stored in the **cursor** field to the logical position of the cursor in the buffer. The easiest convention to choose is that the **cursor** field records the index position of the character that follows the cursor. Thus, if the buffer contains

Figure 10-2 Simple implementation of a command-driven editor

```
/*
 * File: editor.cpp
 * -----
 * This program implements a simple character editor, which
 * is used to test the buffer class. The editor reads
 * and executes simple commands entered by the user.
 */

#include "genlib.h"
#include "buffer.h"
#include "simpio.h"
#include <iostream>
#include <cctype>

/* Private function prototypes */

void ExecuteCommand(EditorBuffer & buffer, string line);

/* Main program */

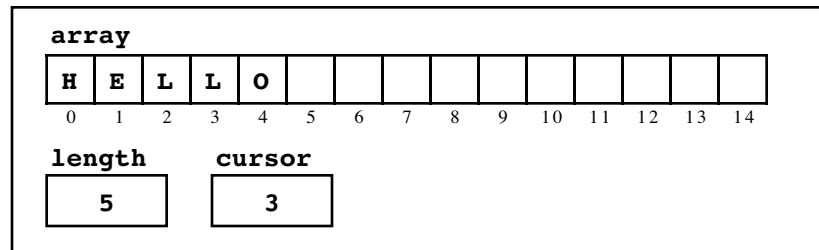
int main() {
    EditorBuffer buffer;
    while (true) {
        cout << "*";
        string cmd = GetLine();
        if (cmd != "") ExecuteCommand(buffer, cmd);
        buffer.display();
    }
    return 0;
}

/*
 * Executes a single editor command.
 */

void ExecuteCommand(EditorBuffer & buffer, string line) {
    switch (toupper(line[0])) {
        case 'I': for (int i = 1; i < line.length(); i++) {
                    buffer.insertCharacter(line[i]);
                }
                break;
        case 'D': buffer.deleteCharacter(); break;
        case 'F': buffer.moveCursorForward(); break;
        case 'B': buffer.moveCursorBackward(); break;
        case 'J': buffer.moveCursorToStart(); break;
        case 'E': buffer.moveCursorToEnd(); break;
        case 'Q': exit(0);
        default: cout << "Illegal command" << endl; break;
    }
}
```

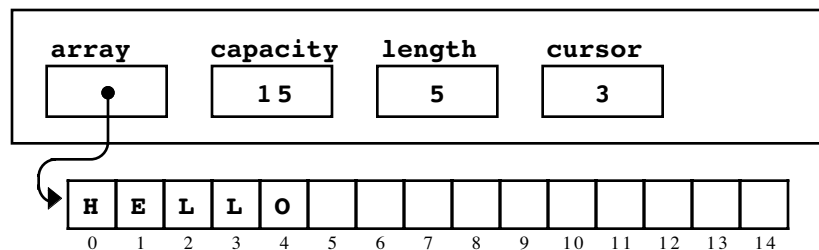
H E L | L O

with the cursor sitting between the two **L** characters, the data members of the buffer object would look something like this:



Using this representation, the cursor is at the beginning of the buffer when the **cursor** field is 0 and at the end of the buffer when the **cursor** field is equal to the **length** field.

The data structure suggested by this diagram, however, is not sufficient to solve the problem. If you were to represent an editor buffer using the precise structure shown in the diagram, your editor could never work with more than 15 characters of text, because the array has only that many elements. Arrays in C++ do not expand dynamically in the way that the **Vector** class does. To create a workable model of the editor buffer, it is essential to redesign the structure so that the storage for **array** is no longer allocated within the structure itself but is instead allocated dynamically on the heap, just as was true for the **CharStack** implementation in Chapter 9. Making this change leads to the following diagram:



As this diagram illustrates, the storage for the array lives outside the structure in a dynamic array whose allocated size is maintained in a field called **capacity**. This design eliminates the character limit because it is possible to allocate new heap space for this array whenever the array runs out of space.

The primary value of creating this data diagram is that it tells you exactly what data members are required in the file **bufpriv.h**, which contains the private section of the **EditorBuffer** class. For the array implementation, all you need are these variables:

```
char *array;
int capacity;
int length;
int cursor;
```

Implementing the buffer operations

Given this design, most of the editor operations are very easy to implement. Each of the four operations that move the cursor can be implemented by assigning a new value to the contents of the **cursor** field. Moving to the beginning of the buffer, for example, requires nothing more than assigning the value 0 to **cursor**; moving it to the end is

simply a matter of copying the **length** field into the **cursor** field. You can see the code for these simple methods in the complete implementation of the **EditorBuffer** class shown in Figure 10-3.

The only operations in Figure 10-3 that require any additional discussion are the constructor, the destructor, and the **insertCharacter** and **deleteCharacter** methods. Because these methods might seem a little tricky, particularly to someone coming upon the for the first time, it is worth including comments in the code that document their operation. The code in Figure 10-3, for example, offers additional documentation for these particular methods in comments labeled “Implementation notes”; the simple methods that implement cursor motion are not documented individually.

The constructor has the responsibility for initializing the instance variables that represent an empty buffer, so the comments for the constructor are a good place to describe those instance variables and what they represent. The destructor is charged with freeing any dynamically allocated storage that was acquired by an object during its lifetime. For the array-based implementation of the **EditorBuffer**, the only dynamically allocated memory is the array used to hold the text. Thus, the code for the destructor consists of the line

```
delete[] array;
```

which deletes the dynamic array storage assigned to **array**.

The **insertCharacter** and **deleteCharacter** methods are interesting because each of them requires shifting characters in the array, either to make room for a character you want to insert or to close up space left by a deleted character. Suppose, for example, that you want to insert the character **x** at the cursor position in the buffer containing

H E L | L O

To do so in the array representation of the buffer, you first need to make sure that there is room in the array. If the **length** field is equal to the **capacity** field, there is no more room in the currently allocated array to accommodate the new character. In that case, it is necessary to expand the array capacity in precisely the same way that the **CharStack** implementation in Chapter 9 expanded its storage.

The extra space in the array, however, is entirely at the end. To insert a character in the middle, you need to make room for that character at the current position of the cursor. The only way to get that space is to shift the remaining characters one position to the right, leaving the buffer structure in the following state:

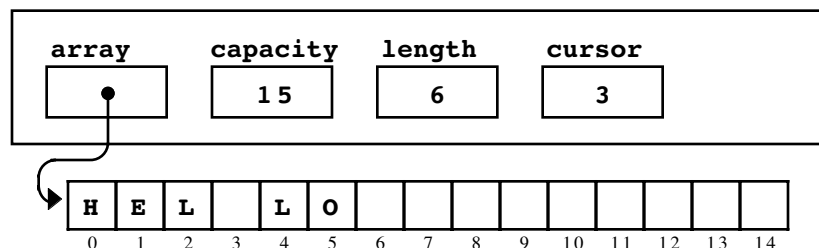


Figure 10-3 Implementation of the editor buffer using an array

```
/*
 * File: arraybuf.cpp
 * -----
 * This file implements the EditorBuffer class using an
 * array to represent the buffer.
 */

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

/* Constants */

const int INITIAL_CAPACITY = 100;

/*
 * Implementation notes: EditorBuffer
 * -----
 * In this representation of the buffer, the characters are stored
 * in a dynamic array along with integers storing the capacity of
 * the buffer (its allocated size), the length of the buffer (its
 * effective size), and the cursor position. The cursor position
 * indicates the index position of the character that follows where
 * the cursor would appear on the screen. The constructor simply
 * initializes that structure.
 */

EditorBuffer::EditorBuffer() {
    capacity = INITIAL_CAPACITY;
    array = new char[capacity];
    length = 0;
    cursor = 0;
}

/*
 * Implementation notes: ~EditorBuffer
 * -----
 * The destructor has to free any memory that is allocated
 * along the way in order to support the guarantee that
 * the editor buffer not leak memory. The only dynamically
 * allocated memory is the dynamic array that stores the text.
 */

EditorBuffer::~EditorBuffer() {
    delete[] array;
}
```

```
/*
 * Implementation notes: moveCursor methods
 * -----
 * The four moveCursor methods simply adjust the value of the
 * cursor instance variable.
 */

void EditorBuffer::moveCursorForward() {
    if (cursor < length) cursor++;
}

void EditorBuffer::moveCursorBackward() {
    if (cursor > 0) cursor--;
}

void EditorBuffer::moveCursorToStart() {
    cursor = 0;
}

void EditorBuffer::moveCursorToEnd() {
    cursor = length;
}

/*
 * Implementation notes: insertCharacter and deleteCharacter
 * -----
 * Each of the functions that inserts or deletes characters
 * must shift all subsequent characters in the array, either
 * to make room for new insertions or to close up space left
 * by deletions.
 */

void EditorBuffer::insertCharacter(char ch) {
    if (length == capacity) expandCapacity();
    for (int i = length; i > cursor; i--) {
        array[i] = array[i - 1];
    }
    array[cursor] = ch;
    length++;
    cursor++;
}

void EditorBuffer::deleteCharacter() {
    if (cursor < length) {
        for (int i = cursor+1; i < length; i++) {
            array[i - 1] = array[i];
        }
        length--;
    }
}
```

```

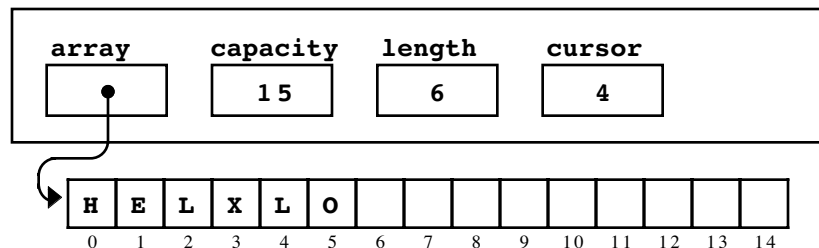
void EditorBuffer::display() {
    for (int i = 0; i < length; i++) {
        cout << ' ' << array[i];
    }
    cout << endl;
    for (int i = 0; i < cursor; i++) {
        cout << " ";
    }
    cout << '^' << endl;
}

/*
 * Implementation notes: expandCapacity
 * -----
 * This private method doubles the size of the array whenever
 * it runs out of space. To do so, it must allocate a new array,
 * copy all the characters from the old array to the new one, and
 * then free the old storage.
 */

void EditorBuffer::expandCapacity() {
    char *oldArray = array;
    capacity *= 2;
    array = new char[capacity];
    for (int i = 0; i < length; i++) {
        array[i] = oldArray[i];
    }
    delete[] oldArray;
}

```

The resulting gap in the array gives you the space you need to insert the **x**, after which the cursor advances so that it follows the newly inserted character, leaving the following configuration:



The **deleteCharacter** operation is similar in that it requires a loop to close the gap left by the deleted character.

Assessing the computational complexity of the array implementation

In order to establish a baseline for comparison with other representations, it is important to consider the computational complexity of the array-based implementation of the editor. As usual, the goal of the complexity analysis is to understand how the execution time required for the editing operations varies qualitatively as a function of the problem size. In the editor example, the problem size is best measured by the number of characters in the buffer. For the editor buffer, you therefore need to determine how each of the editing operations is affected by the number of characters in the buffer.

Let's start with an easy case. How is the efficiency of the **moveCursorForward** operation affected by N , the number of characters? Using the array representation for buffers, the method **moveCursorForward** has the following implementation:

```
void EditorBuffer::moveCursorForward() {
    if (cursor < length) cursor++;
}
```

Even though the function does indeed check the length of the buffer, it doesn't take long to realize that the execution time of the function is independent of the buffer length. This function executes precisely the same operations no matter how long the buffer is: there is one test and, in almost all cases, one increment operation. Because the execution time is independent of N , the **moveCursorForward** operation runs in $O(1)$ time. The same analysis holds for **moveCursorBackward**, **moveCursorToStart**, and **moveCursorToEnd**, none of which involve any operations that depend on the length of the buffer.

But what about **insertCharacter**? In `arraybuf.cpp`, the **insertCharacter** function contains the following **for** loop:

```
for (int i = length; i > cursor; i--) {
    text[i] = text[i - 1];
}
```

If you insert a character at the end of the buffer, this function runs pretty quickly, because there is no need to shift characters to make room for the new one. On the other hand, if you insert a character at the beginning of the buffer, every character in the buffer must be shifted one position rightward in the array. Thus, in the worst case, the running time for **insertCharacter** is proportional to the number of characters in the buffer and is therefore $O(N)$. Because the **deleteCharacter** operation has a similar structure, its complexity is also $O(N)$. These results are summarized in Table 10-2.

The fact that the last two operations in the table require linear time has important performance implications for the editor program. If an editor uses arrays to represent its internal buffer, it will start to run more slowly as the number of characters in the buffer becomes large. Because this problem seems serious, it makes sense to explore other representational possibilities.

10.4 Implementing the editor using stacks

The problem with the array implementation of the editor buffer is that insertions and deletions run slowly when they occur near the beginning of the buffer. When those same operations are applied at the end of the buffer, they run relatively quickly because there is no need to shift the characters in the internal array. This property suggests an approach to making things faster: force all insertions and deletions to occur at the end of the buffer. While this approach is completely impractical from the user's point of view, it does contain the seed of a workable idea.

Table 10-2 Complexity of the editor operations (array representation)

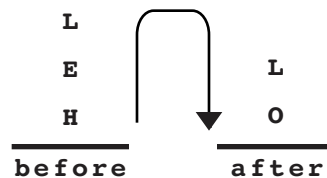
Function	Complexity
moveCursorForward	$O(1)$
moveCursorBackward	$O(1)$
moveCursorToStart	$O(1)$
moveCursorToEnd	$O(1)$
insertCharacter	$O(N)$
deleteCharacter	$O(N)$

The key insight necessary to make insertions and deletions faster is that you can divide the buffer at the cursor boundary and store the characters before and after the cursor in separate structures. Because all changes to the buffer occur at the cursor position, each of those structures behaves like a stack and can be represented by the **Stack** class. The characters that precede the cursor are pushed on one stack so that the beginning of the buffer is at the base and the character just before the pointer is at the top. The characters after the cursor are stored in the opposite direction, with the end of the buffer at the base of the stack and the character just after the pointer at the top.

The best way to illustrate this structure is with a diagram. Suppose that the buffer contains the string

H E L | L O

with the cursor sitting between the two **L** characters as shown. The two-stack representation of the buffer looks like this:



To read the contents of the buffer, it is necessary to read up the characters in the **before** stack and then down the characters in the **after** stack, as indicated by the arrow.

Defining the private data representation for the stack-based buffer

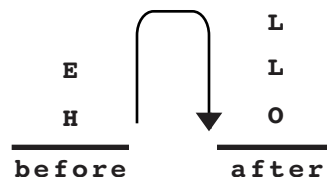
Using this strategy, the data members for a buffer object are simply a pair of stacks, one to hold the character before the cursor and another to hold the ones that come after it. For the stack-based buffer, the **bufpriv.h** file has the following contents:

```
CharStack before;
CharStack after;
```

Note that the cursor is not explicitly represented in this model but is instead simply the boundary between the two stacks.

Implementing the buffer operations

In the stack model, implementing most of the operations for the editor is surprisingly easy. For example, moving backward consists of popping a character from the **before** stack and pushing it back on the **after** stack. Thus, if you were to call **moveCursorBackward** in the position above, the buffer would be left in the following position:



Moving forward is entirely symmetrical. Inserting a character consists of pushing that character on the **before** stack. Deleting a character is accomplished by popping a character from the **after** stack and throwing it away.

This conceptual outline makes it easy to write the code for the stack-based editor, which appears in Figure 10-4. Four commands—**insertCharacter**, **deleteCharacter**, **moveCursorForward**, and **moveCursorBackward**—now contain no loops and therefore clearly run in constant time because the stack operations they call are themselves constant-time operations.

But what about the two remaining operations? Implementing **moveCursorToStart** and **moveCursorToEnd** requires the program to transfer the entire contents from one of the stacks to the other. Given the operations provided by the stack class, the only way to accomplish this operation is to pop values from one stack and push them back on the other stack, one value at a time, until the original stack becomes empty. For example, the **moveCursorToEnd** operation has the following implementation:

```
void EditorBuffer::moveCursorToEnd() {
    while (!after.isEmpty()) {
        before.push(after.pop());
    }
}
```

These implementations have the desired effect, but require $O(N)$ time in the worst case.

Comparing computational complexities

Table 10-3 shows the computational complexity of the editor operations for both the array- and the stack-based implementation of the editor.

Which implementation is better? Without some knowledge of the usage pattern, it is impossible to answer this question. Knowing a little about the way that people use editors, however, suggests that the stack-based strategy might well be more efficient because the slow operations for the array implementation (insertion and deletion) are used much more frequently than the slow operations for the stack implementation (moving the cursor a long distance).

While this tradeoff seems reasonable given the relative frequency of the operations involved, it makes sense to ask whether it is possible to do even better. After all, it is now true that each of the six fundamental editing operations runs in constant time in at least one of the two editor implementations. Insertion is slow given the array implementation but fast when using the stack approach. By contrast, moving to the front of the buffer is fast in the array case but slow in the stack case. None of the operations, however, seems to be *fundamentally* slow, since there is always some implementation which makes that operation fast. Is it possible to develop an implementation in which all the operations are fast? The answer to this question turns out to be “yes,” but discovering the key to the puzzle will require you to learn a new approach to representing ordering relationships in a data structure.

Table 10-3 Relative performance of the array and stack representations

Function	Array	Stack
moveCursorForward	$O(1)$	$O(1)$
moveCursorBackward	$O(1)$	$O(1)$
moveCursorToStart	$O(1)$	$O(N)$
moveCursorToEnd	$O(1)$	$O(N)$
insertCharacter	$O(N)$	$O(1)$
deleteCharacter	$O(N)$	$O(1)$

Figure 10-4 Implementation of the editor buffer using a pair of stacks

```
/*
 * File: stackbuf.cpp
 * -----
 * This file implements the EditorBuffer class using a pair of
 * stacks to represent the buffer. The characters before the
 * cursor are stored in the before stack, and the characters
 * after the cursor are stored in the after stack.
 */

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

/*
 * Implementation notes: EditorBuffer
 * -----
 * In this implementation, the only thing that needs to happen is
 * that the two stacks are initialized to be empty. That happens
 * automatically when the CharStack variables are declared.
 */

EditorBuffer::EditorBuffer() {
    /* Empty */
}

/*
 * Implementation notes: ~EditorBuffer
 * -----
 * The destructor has to free any dynamically allocated memory.
 * In this implementation, however, all the allocation occurs in
 * the CharStack class and will be managed by its destructor.
 */

EditorBuffer::~~EditorBuffer() {
    /* Empty */
}

/*
 * Implementation notes: moveCursor methods
 * -----
 * The four moveCursor methods use push and pop to transfer
 * values between the two stacks.
 */

void EditorBuffer::moveCursorForward() {
    if (!after.isEmpty()) {
        before.push(after.pop());
    }
}

void EditorBuffer::moveCursorBackward() {
    if (!before.isEmpty()) {
        after.push(before.pop());
    }
}
```

```
void EditorBuffer::moveCursorToStart() {
    while (!before.isEmpty()) {
        after.push(before.pop());
    }
}

void EditorBuffer::moveCursorToEnd() {
    while (!after.isEmpty()) {
        before.push(after.pop());
    }
}

/*
 * Implementation notes: insertCharacter and deleteCharacter
 * -----
 * Each of the functions that inserts or deletes characters
 * can do so with a single push or pop operation.
 */

void EditorBuffer::insertCharacter(char ch) {
    before.push(ch);
}

void EditorBuffer::deleteCharacter() {
    if (!after.isEmpty()) {
        after.pop();
    }
}

/*
 * Implementation notes: display()
 * -----
 * The display operator is complicated in the stack-based version,
 * but it is not part of the fundamental operator set.
 */

void EditorBuffer::display() {
    int nBefore = before.size();
    moveCursorToStart();
    while (!after.isEmpty()) {
        cout << ' ' << after.peek();
        moveCursorForward();
    }
    cout << endl;
    moveCursorToStart();
    for (int i = 0; i < nBefore; i++) {
        cout << " ";
        moveCursorForward();
    }
    cout << '^' << endl;
}
```

10.5 Implementing the editor using linked lists

As an initial step toward finding a more efficient representation for the editor buffer, it makes sense to examine why the previous approaches have failed to provide efficient service for certain operations. In the case of the array implementation, the answer is obvious: the problem comes from the fact that you have to move a large number of characters whenever you need to insert some new text near the beginning of the buffer. For example, suppose that you were trying to enter the alphabet and instead typed

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

When you discovered that you'd left out the letter **B**, you would have to shift each of the next 24 characters one position to the right in order to make room for the missing letter. A modern computer could handle this shifting operation relatively quickly as long as the buffer did not grow too long; even so, the delay would eventually become noticeable if the number of characters in the buffer became sufficiently large.

Suppose, however, that you were writing before the invention of modern computers. What would happen if you were instead using a typewriter and had already typed the complete line? To avoid giving the impression that you had slighted one of the letters in the alphabet, you could simply take out a pen and make the following notation:

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

The result is perhaps a trifle inelegant, but nonetheless acceptable in such desperate circumstances.

One advantage of this human editing notation is that it allows you to suspend the rule that says all letters are arranged in sequence in precisely the form in which they appear on the printed page. The carat symbol below the line tells your eyes that, after reading the **A**, you have to then move up, read the **B**, come back down, read the **C**, and then continue with the sequence. It is also important to notice another advantage of using this insertion strategy. No matter how long the line is, all you have to draw is the new character and the carat symbol. Using pencil and paper, insertion runs in constant time.

The concept of a linked list

You can adopt a similar approach in designing a computer-based representation of the editing buffer. In fact, you can even generalize the idea so that, instead of representing the buffer mostly as an array with occasional deviations from the normal sequence, you simply draw an arrow from each letter in the buffer to the letter that follows it. The original buffer contents might then be represented as follows:

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

If you then need to add the character **B** after the character **A**, all you need to do is (1) write the **B** down somewhere, (2) draw an arrow from **B** to the letter to which **A** is pointing (which is currently the **C**) so that you don't lose track of the rest of the string, and (3) change the arrow pointing from the **A** so that it now points to the **B**, like this:

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

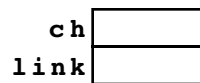
Given the symbols used to draw these diagrams, it is probably not surprising to discover that the principal tool for implementing the new strategy in C++ is the pointer. One of

the great advantages of pointers is that they make it possible for one data object to include a pointer to a second object. You can use this pointer to indicate an ordering relationship, much like the ordering relationship implied by the arrows in the foregoing diagram. Pointers used in this way are often referred to as **links**. When such links are used to create a linearly ordered data structure in which each element points to its successor, that structure is called a **linked list**.

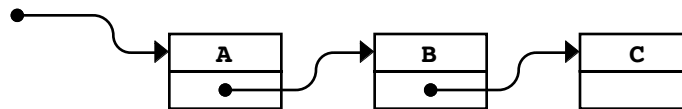
Designing a linked-list data structure

If you want to apply this pointer-based strategy to the design of the editor buffer, what you need is a linked list of characters. You have to associate each character in the buffer with a pointer that indicates the next character in the list. That pointer, however, cannot be a simple character pointer; what you really need is a pointer to the next character/link combination. To make a linked list work, you have to combine in a single record a piece of data relevant to the application (in this case, the character) and a pointer to another such record, which is then used to indicate the internal ordering. This combination of the application data and the pointer becomes the basic building block for the linked list, which is often referred to as a **cell**.

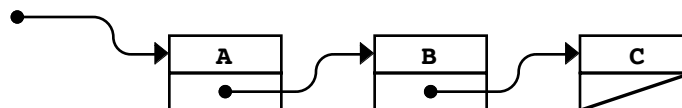
To make the idea of a cell a little easier to visualize, it often helps to start with a structure diagram before moving to an actual definition. In the editor buffer, a cell has two components: a character and link to the following cell. A single cell can therefore be diagrammed as follows:



You can then represent a sequence of characters by linking several of these cells together. For example, the character sequence **ABC** could be represented as a linked list containing the following collection of cells:



If the **c** is the last character in the sequence, you need to indicate that fact by putting a special value in the **link** field of that cell to indicate that there are no additional characters in the list. When programming in C++, it is customary to use the special pointer value **NULL** for this purpose. In list structure diagrams, however, it is common to indicate the **NULL** value with a diagonal line across the box, as follows:



To represent these cell structure diagrams in C++, you need to define an appropriate record type. The **cellT** structure should include a **ch** field for the character and a **link** field that points to the next cell. This definition is a bit unusual in that the **cellT** type is defined in terms of itself and is therefore a **recursive type**. The following type definition correctly represents the conceptual structure of a cell:

```
struct cellT {
    char ch;
    cellT *link;
};
```

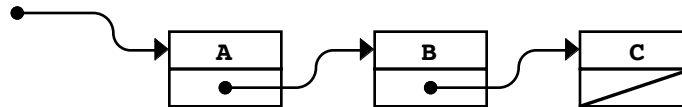
Using a linked list to represent the buffer

You are now in a position to consider the question of how you might use linked lists to represent the editor buffer. One possibility is simply to have the initial pointer in the linked list be the buffer, but this approach ignores the fact that you must also represent the cursor. The need to indicate a current buffer position suggests that the data members for the **EditorBuffer** class should consist of two pointers: one to indicate where the buffer starts and one to indicate the current cursor position. Thus, the contents of the file **bufpriv.h** are

```
struct cellT {
    char ch;
    cellT *link;
};

cellT *start;
cellT *cursor;
```

Given this definition, you probably expect that the linked list representation for a buffer containing the text **ABC** looks like this:



That design seems reasonable until you try to figure out how to insert characters at each possible cursor position. In this diagram, it seems clear that the **start** field should contain the pointer to the initial cell. But what about the **cursor** field? Your hope, of course, is that the **cursor** field can also be a pointer to one of these three cells, depending on the position of the abstract cursor. Unfortunately, there's a bit of a problem. There are four possible positions of the cursor in this buffer:

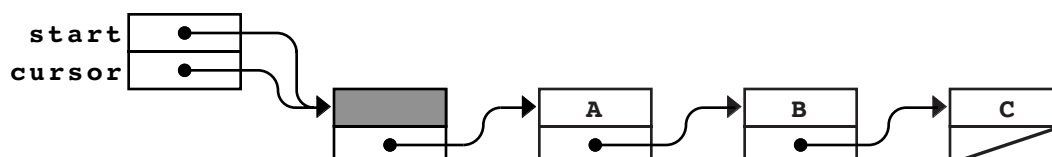
| A B C A | B C A B | C A B C |

On the other hand, there are only three cells to which the **cursor** field might point. Thus, it is not immediately clear how you would be able to represent each of the possible cursor locations.

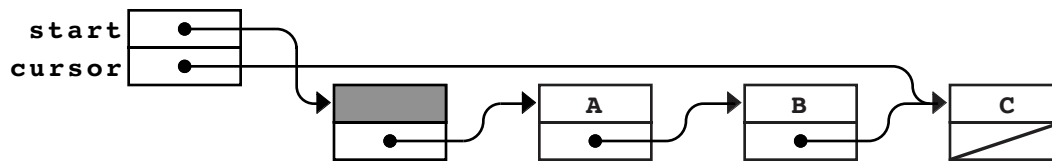
There are many tactical approaches to solving this problem, but the one that usually turns out to be the best is to allocate an extra cell so that the list contains one cell for each possible insertion point. Typically, this cell goes at the beginning of the list and is called a **dummy cell**. The value of the **ch** field in the dummy cell is irrelevant and is indicated in diagrams by filling the value field with a gray background, as follows:



When you use the dummy cell approach, the **cursor** field points to the cell immediately before the logical insertion point. For example, a buffer containing **ABC** with the cursor at the beginning of the buffer would look like this:



Both **start** and **cursor** point to the dummy cell, and insertions will occur immediately after this cell. If the **cursor** field instead indicates the end of the buffer, the diagram looks like this:



Insertion into a linked-list buffer

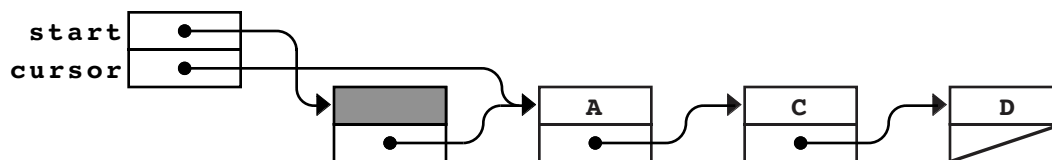
No matter where the cursor is positioned, the insertion operation for a linked list consists of precisely the steps outlined for the conceptual arrow model earlier, written here in the terminology used in the buffer data members:

1. Allocate space for a new cell, and store the pointer to this cell in the temporary variable **cp**.
2. Copy the character to be inserted into the **ch** field of the new cell (**cp**→**ch**).
3. Go to the cell indicated by the **cursor** field of the buffer and copy its link field (**cursor**→**link**) to the **link** field of the new cell (**cp**→**link**). This operation makes sure that you don't lose the characters that lie beyond the current cursor position.
4. Change the **link** field in the cell addressed by the cursor (**cursor**→**link**) so that it points to the new cell.
5. Change the **cursor** field in the buffer so that it also points to the new cell. This operation ensures that the next character will be inserted after this one in repeated insertion operations.

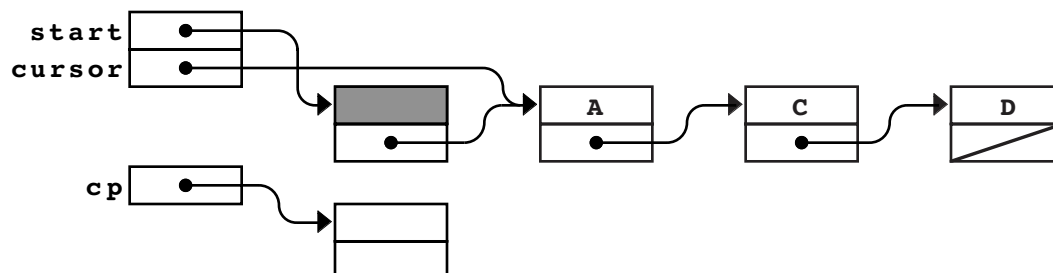
To illustrate this process, suppose that you want to insert the letter **B** into a buffer that currently contains

A | C D

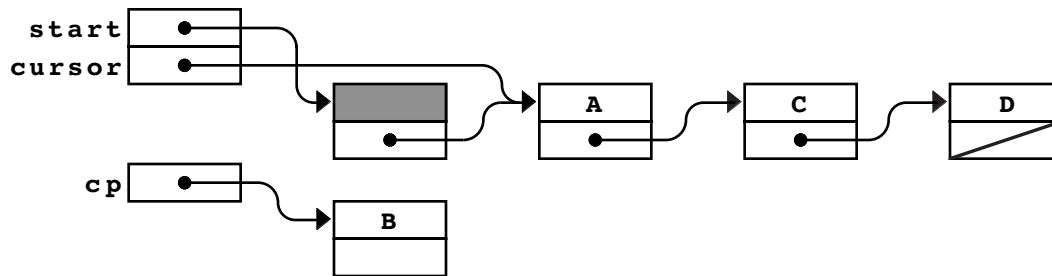
with the cursor between the **A** and the **C** as shown. The situation prior to the insertion looks like this:



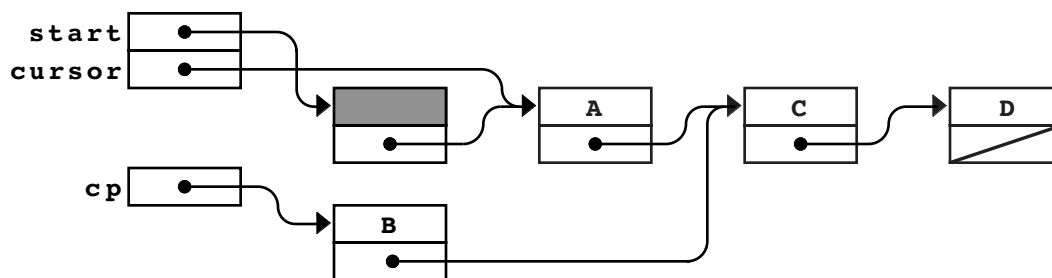
The first step in the insertion strategy is to allocate a new cell and store a pointer to it in the variable **cp**, as shown:



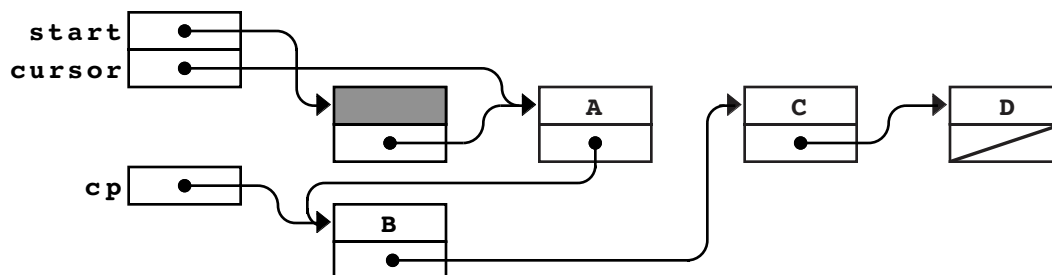
Step 2 is to put the character **B** into the **ch** field of the new cell, which leaves the following configuration:



In step 3, you copy the **link** field from the cell whose address appears in the **cursor** field into the **link** field of the new cell. That **link** field points to the cell containing **C**, so the resulting diagram looks like this:

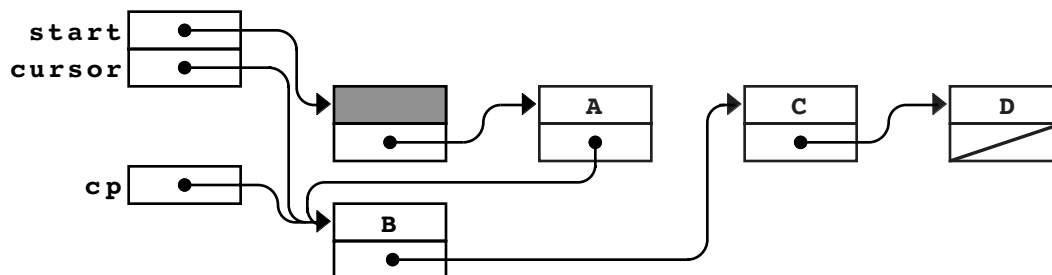


In step 4, you change the **link** field in the current cell addressed by the cursor so that it points to the newly allocated cell, as follows:

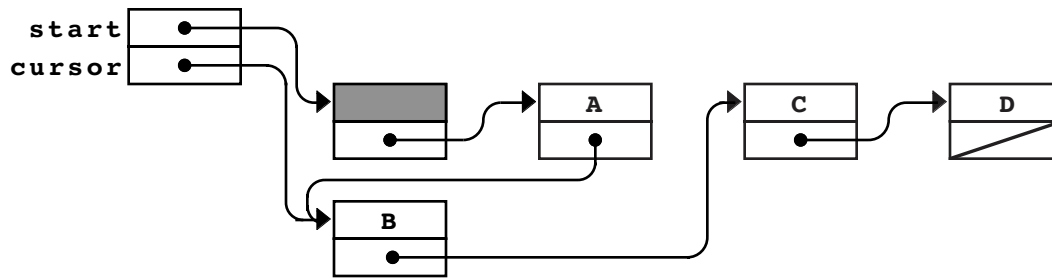


Note that the buffer now has the correct contents. If you follow the arrows from the dummy cell at the beginning of the buffer, you encounter the cells containing **A**, **B**, **C**, and **D**, in order along the path.

The final step consists of changing the **cursor** field in the buffer structure so that it also points to the new cell, which results in the following configuration:



When the program returns from the **insertCharacter** function, the temporary variable **cp** is released, which results in the following final buffer state:



which represents the buffer contents

A B | C D

The following implementation of the **insertCharacter** method is a simple translation into C++ code of the informal steps illustrated in the last several diagrams:

```
void EditorBuffer::insertCharacter(char ch) {
    cellT *cp = new cellT;
    cp->ch = ch;
    cp->link = cursor->link;
    cursor->link = cp;
    cursor = cp;
}
```

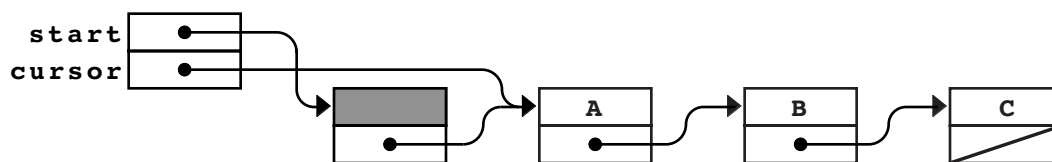
Because there are no loops inside this function, the **insertCharacter** function now runs in constant time.

Deletion in a linked-list buffer

To delete a cell in a linked list, all you have to do is remove it from the pointer chain. Let's assume that the current contents of the buffer are

A B C

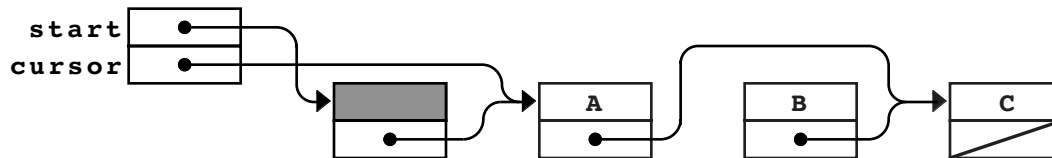
which has the following graphical representation:



Deleting the character after the cursor means that you need to eliminate the cell containing the **B** by changing the **link** field of the cell containing **A** so that it points to the next character further on. To find that character, you need to follow the **link** field from the current cell and continue on to the following **link** field. The necessary statement is therefore

```
cursor->link = cursor->link->link;
```

Executing this statement leaves the buffer in the following state:



Because the cell containing **B** is no longer accessible through the linked-list structure, it is good policy to free its storage by calling **delete**, as shown in the following implementation of **deleteCharacter**:

```
void EditorBuffer::deleteCharacter() {
    if (cursor->link != NULL) {
        cellT *oldcell = cursor->link;
        cursor->link = oldcell->link;
        delete oldcell;
    }
}
```

Note that you need a variable like **oldcell** to hold a copy of the pointer to the cell about to be freed while you adjust the chain pointers. If you do not save this value, there will be no way to refer to that cell when you want to call **delete**.

Cursor motion in the linked-list representation

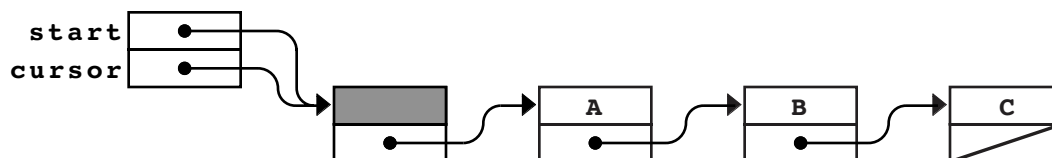
The remaining operations in the **EditorBuffer** class simply move the cursor. How would you go about implementing these operations in the linked-list buffer? It turns out that two of these operations (**moveCursorForward** and **moveCursorToStart**) are easy to perform in the linked-list model. To move the cursor forward, for example, all you have to do is pick up the **link** field from the current cell and make that pointer be the new current cell by storing it in the **cursor** field of the buffer. The statement necessary to accomplish this operation is simply

```
cursor = cursor->link;
```

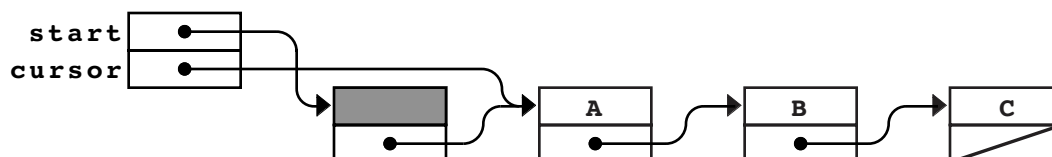
As an example, suppose that the editor buffer contains

| A B C

with the cursor at the beginning as shown. The list structure diagram for the buffer is then



and the result of executing the **moveCursorForward** operation is



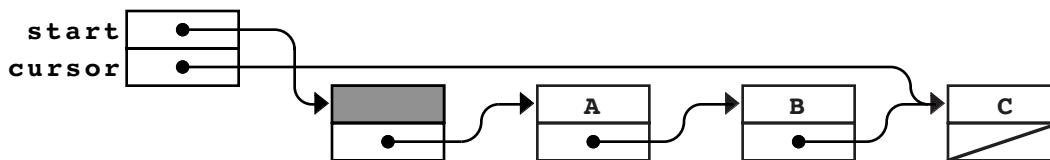
Of course, when you reach the end of the buffer, you can no longer move forward. The implementation of **moveCursorForward** must check for this condition, so the complete function definition looks like this:

```
void EditorBuffer::moveCursorForward() {
    if (cursor->link != NULL) {
        cursor = cursor->link;
    }
}
```

Moving the cursor to the beginning of the buffer is equally easy. No matter where the cursor is, you can always restore it to the beginning of the buffer by copying the **start** field into the **cursor** field. Thus, the implementation of **moveCursorToStart** is simply

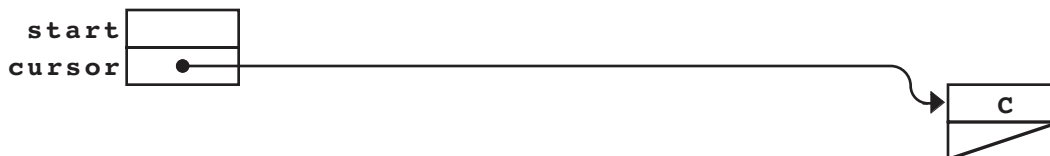
```
void EditorBuffer::moveCursorToStart() {
    cursor = start;
}
```

The operations **moveCursorBackward** and **moveCursorToEnd**, however, are more complicated. Suppose, for example, that the cursor is sitting at the end of a buffer containing the characters **ABC** and that you want to move back one position. In its graphical representation, the buffer looks like this:

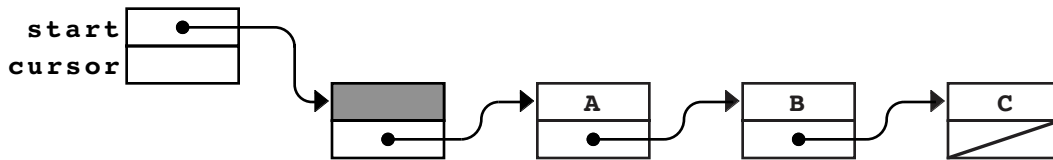


Given the structure of the **EditorBuffer**, there is no constant time strategy for backing up the pointer. The problem is that you have no easy way—given the information you can see—to find out what cell precedes the current one. Pointers allow you to follow a chain from the pointer to the object to which it points, but there is no way to reverse the direction. Given only the address of a cell, it is impossible to find out what cells point to it. With respect to the pointer diagrams, the effect of this restriction is that you can move from the dot at the base of an arrow to the cell to which the arrow points, but you can never go from an arrowhead back to its base.

In the list-structure representation of the buffer, you have to implement every operation in terms of the data that you can see from the buffer structure itself, which contains the **start** and the **cursor** pointers. Looking at just the **cursor** field and following the links that are accessible from that position does not seem promising, because the only cell reachable on that chain is the very last cell in the buffer, as illustrated in the following diagram:



If you instead consider the **start** pointer, the entire buffer is accessible on the linked-list chain:



On the other hand, without looking at the **cursor** field, you no longer have any indication of the current buffer position.

Before you abandon hope, you need to recognize that it is possible in this case to find the cell which precedes the current cell. It is just not possible to do so in constant time. If you start at the beginning of the buffer and follow the links through all its cells, you will eventually hit a cell whose **link** field points to the same cell as the **cursor** field in the **EditorBuffer** itself. This cell must be the preceding cell in the list. Once you find it, you can simply change the **cursor** field in the **EditorBuffer** to point to that cell, which has the effect of moving the cursor backward.

Moving through the values of a linked list, one cell at a time, by following link pointers is a very common operation, which is usually called **traversing** or **walking** the list. To traverse the list representing the buffer, you first declare a pointer variable and initialize it to the beginning of the list. Thus, in this instance, you might write

```
cellT *cp = start;
```

To find the character preceding the cursor, you want to walk down the list as long as **cp**'s **link** field does not match the cursor, moving **cp** from cell to cell by following each **link** field. You might therefore continue the code by adding the following **while** loop:

```
cellT *cp = start;
while (cp->link != cursor) {
    cp = cp->link;
}
```

When the **while** loop exits, **cp** is set to the cell prior to the cursor. As with moving forward, you need to protect this loop against trying to move past the limits of the buffer, so the complete code for **moveCursorBackward** would be

```
void EditorBuffer::moveCursorBackward() {
    cellT *cp = start;
    if (cursor != start) {
        while (cp->link != cursor) {
            cp = cp->link;
        }
        cursor = cp;
    }
}
```

For precisely the same reasons, you can implement **moveCursorToEnd** only by walking through the entire linked list until you detect the **NULL** pointer, as illustrated by the following code:

```
void EditorBuffer::moveCursorToEnd() {
    while (cursor->link != NULL) {
        moveCursorForward();
    }
}
```

Linked-list idioms

Many C++ programmers, however, will not use a **while** in the **moveCursorBackward** implementation to walk through the elements of a list. In C++, whenever you have a repetitive operation in which you can easily specify an initialization, a test to see whether you should continue, and a sequencing operation that moves from one cycle to the next, the iterative construct of choice is the **for** loop, which allows you to put all these ideas together in one place. In the **moveCursorBackward** example, you have all three of these pieces and might therefore have coded the internal loop as follows:

```
for (cp = start; cp->link != cursor; cp = cp->link) {
    /* Empty */
}
```

The first thing to notice about this loop is that the body performs no operations at all. The **for** loop is executed entirely for its effect on the pointer variable **cp**, and there are no other operations to perform. In C++, such situations come up surprisingly often. You can indicate an empty loop body simply by putting a semicolon at the end of the **for** header line; this text will always use a comment to make it easier to see that the loop body has no effect, as in the preceding example.

Because the **for** loop is so useful when working with linked lists, it is important to recognize the standard **for** loop idioms used to manipulate list structure. In C++ programs, such idioms are often as important as those used with arrays. For example, the idiom for performing an operation on every element in an array whose effective size is **n** looks like this:

```
for (int i = 0; i < N; i++) {
    . . . code using a[i] . . .
}
```

For linked lists, the corresponding idiom is

```
for (cellT *cp = list; cp != NULL; cp = cp->link) {
    . . . code using cp . . .
}
```

Completing the buffer implementation

The complete **EditorBuffer** class contains three operations that have yet to be implemented: the constructor, the destructor, and the method **display**. In the constructor, the only wrinkle is that you need to remember the existence of the dummy cell. The code must allocate the dummy cell that is present even in the empty buffer. Once you remember this detail, however, the code is fairly straightforward:

```
EditorBuffer::EditorBuffer() {
    start = cursor = new cellT;
    start->link = NULL;
}
```

The implementation of the destructor is slightly more subtle. When the destructor is called, it is responsible for freeing any memory allocated by the class, which includes every cell in the linked-list chain. Given the earlier discussion of the **for** loop idiom, you might be tempted to code that loop as follows:

```
for (cellT *cp = start; cp != NULL; cp = cp->link) {
    delete cp;
}
```



The problem here is that the code tries to use the **link** pointer inside each block after that block has been freed. Once you call **delete** on a pointer to a record, you are no longer allowed to look inside that record. Doing so is likely to cause errors. To avoid this problem, you need to maintain your position in the list in a separate variable as you free each cell; in essence, you need a place to stand. Thus, the correct code for **~EditorBuffer** is slightly more convoluted and has the following form:

```
EditorBuffer::~EditorBuffer() {
    cellT *cp = start;
    while (cp != NULL) {
        cellT *next = cp->link;
        delete cp;
        cp = next;
    }
}
```

The complete code for the linked-list implementation of the buffer class appears in Figure 10-5.

Computational complexity of the linked-list buffer

From the discussion in the preceding section, it is easy to add another column to the complexity table showing the cost of the fundamental editing operations as a function of the number of characters in the buffer. The new table, which includes the data for all three implementations, appears in Table 10-4.

Unfortunately, the table for the list structure representation still contains two $O(N)$ operations, **moveCursorBackward** and **moveCursorToEnd**. The problem with this representation is that the link pointers impose a directionality on the implementation: moving forward is easy because the pointers move in the forward direction.

Doubly linked lists

The good news is that this problem is quite easy to solve. To get around the problem that the links run only in one direction, all you need to do is make the pointers symmetrical. In addition to having a pointer from each cell that indicates the next one, you can also include a pointer to the previous cell. The resulting structure is called a **doubly linked list**.

Each cell in the doubly linked list has two link fields, a **prev** field that points to the previous cell and a **next** field that points to the next one. For reasons that will become clear when you implement the primitive operations, it simplifies the manipulation of the structure if the **prev** field of the dummy cell points to the end of the buffer and the **next** field of the last cell points back to the dummy cell.

Table 10-4 Relative efficiency of the buffer representations

Function	Array	Stack	List
moveCursorForward	$O(1)$	$O(1)$	$O(1)$
moveCursorBackward	$O(1)$	$O(1)$	$O(N)$
moveCursorToStart	$O(1)$	$O(N)$	$O(1)$
moveCursorToEnd	$O(1)$	$O(N)$	$O(N)$
insertCharacter	$O(N)$	$O(1)$	$O(1)$
deleteCharacter	$O(N)$	$O(1)$	$O(1)$

Figure 10-5 Implementation of the editor buffer using a linked list

```

/*
 * File: listbuf.cpp
 * -----
 * This file implements the EditorBuffer class using a linked
 * list to represent the buffer.
 */

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

/*
 * Implementation notes: EditorBuffer constructor
 * -----
 * This function initializes an empty editor buffer, represented
 * as a linked list. To simplify the link list operation, this
 * implementation adopts the useful programming tactic of
 * keeping an extra "dummy" cell at the beginning of each list,
 * so that the empty buffer has the following representation:
 *
 *      +-----+      +-----+
 *      |   o---+----->|   |
 *      +-----+      /   +-----+
 *      |   o---+---/   | NULL |
 *      +-----+      +-----+
 */

EditorBuffer::EditorBuffer() {
    start = cursor = new cellT;
    start->link = NULL;
}

/*
 * Implementation notes: EditorBuffer destructor
 * -----
 * The destructor must delete every cell in the buffer. Note
 * that the loop structure is not exactly the standard idiom for
 * processing every cell within a linked list, because it is not
 * legal to delete a cell and later look at its link field. To
 * avoid selecting fields in the structure after it has been
 * deallocated, you have to copy the link pointer before calling
 * delete.
 */

EditorBuffer::~EditorBuffer() {
    cellT *cp = start;
    while (cp != NULL) {
        cellT *next = cp->link;
        delete cp;
        cp = next;
    }
}

```



```
/*
 * Implementation notes: cursor movement
 * -----
 * The four functions that move the cursor have different time
 * complexities because the structure of a linked list is
 * asymmetrical with respect to moving backward and forward.
 * Moving forward one cell is simply a matter of picking up the
 * link pointer; moving backward requires a loop until you reach
 * the current cursor position. Similarly, moving to the start
 * of the buffer takes constant time, but finding the end requires
 * an  $O(N)$  loop.
 */

void EditorBuffer::moveCursorForward() {
    if (cursor->link != NULL) {
        cursor = cursor->link;
    }
}

void EditorBuffer::moveCursorBackward() {
    cellT *cp = start;
    if (cursor != start) {
        while (cp->link != cursor) {
            cp = cp->link;
        }
        cursor = cp;
    }
}

void EditorBuffer::moveCursorToStart() {
    cursor = start;
}

void EditorBuffer::moveCursorToEnd() {
    while (cursor->link != NULL) {
        moveCursorForward();
    }
}

/*
 * Implementation notes: insertCharacter
 * -----
 * The primary advantage of the linked list representation for
 * the buffer is that the insertCharacter operation can now be
 * accomplished in constant time. The steps required are:
 *
 * 1. Allocate a new cell and put the new character in it.
 * 2. Copy the pointer indicating the rest of the list into the link.
 * 3. Update the link in the current cell to point to the new one.
 * 4. Move the cursor forward over the inserted character.
 */

void EditorBuffer::insertCharacter(char ch) {
    cellT *cp = new cellT;
    cp->ch = ch;
    cp->link = cursor->link;
    cursor->link = cp;
    cursor = cp;
}
```

```

/*
 * Implementation notes: deleteCharacter
 * -----
 * As with insertCharacter, the list representation makes it
 * possible to implement the deleteCharacter operation in
 * constant time. The necessary steps are:
 *
 * 1. Remove the current cell from the chain by "pointing around it".
 * 2. Free the cell to reclaim the memory.
 */

void EditorBuffer::deleteCharacter() {
    if (cursor->link != NULL) {
        cellT *oldcell = cursor->link;
        cursor->link = oldcell->link;
        delete oldcell;
    }
}

/*
 * Implementation notes: display
 * -----
 * The display method uses the standard for loop idiom to loop
 * through the cells in the linked list. The first loop displays
 * the character; the second marks the cursor position.
 */

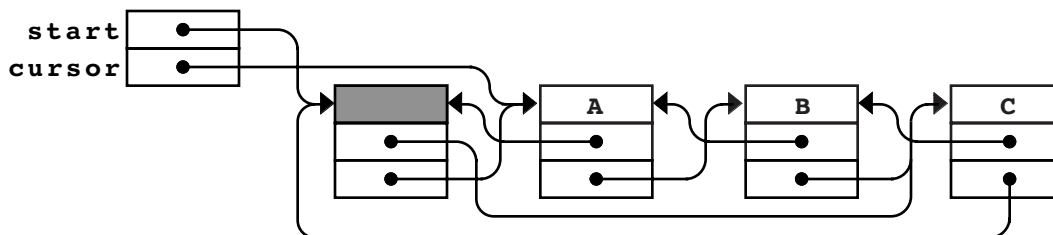
void EditorBuffer::display() {
    for (cellT *cp = start->link; cp != NULL; cp = cp->link) {
        cout << ' ' << cp->ch;
    }
    cout << endl;
    for (cellT *cp = start; cp != cursor; cp = cp->link) {
        cout << " ";
    }
    cout << '^' << endl;
}

```

If you use this design, the doubly linked representation of the buffer containing the text

A | B C

looks like this:



There are quite a few pointers in this diagram, which makes it is easy to get confused. On the other hand, the structure has all the information you need to implement each of the fundamental editing operations in constant time. The actual implementation, however, is left as an exercise so that you can refine your understanding of linked lists.

Time-space tradeoffs

The discovery that you can implement a buffer that has these good computational properties is an important theoretical result. Unfortunately, that result may not in fact be so useful in practice, at least in the context of the editor application. By the time you get around to adding the **prev** field to each cell for the doubly linked list, you will end up using at least nine bytes of memory to represent each character. You may be able to perform editing operations very quickly, but you will use up memory at an extravagant rate. At this point, you face what computer scientists call a **time-space tradeoff**. You can improve the computational efficiency of your algorithm, but waste space in doing so. Wasting this space could matter a lot, if, for example, it meant that the maximum size of the file you could edit on your machine were only a tenth what it would have been if you had chosen the array representation.

When such situations arise in practice, it is usually possible to develop a hybrid strategy that allows you to select a point somewhere in the middle of the time-space tradeoff curve. For example, you could combine the array and linked-list strategies by representing the buffer as a doubly linked list of lines, where each line was represented using the array form. In this case, insertion at the beginning of a line would be a little slower, but only in proportion to the length of the line and not to the length of the entire buffer. On the other hand, this strategy requires link pointers for each line rather than for each character. Since a line typically contains many characters, using this representation would reduce the storage overhead considerably.

Getting the details right on hybrid strategies can be a challenge, but it is important to know that such strategies exist and that there are ways to take advantage of algorithmic time improvements that are not prohibitively expensive in terms of their storage requirements.

Summary

Even though this chapter focused its attention on implementing a class representing an editor buffer, the buffer itself is not the main point. Text buffers that maintain a cursor position are useful in a relatively small number of application domains. The individual techniques used to improve the buffer representation—particularly the concept of a linked list—are fundamental ideas that you will use over and over again.

Important points in this chapter include:

- The strategy used to represent a class can have a significant effect on the computational complexity of its operations.
- Although an array provides a workable representation for an editor buffer, you can improve its performance by using other representation strategies. Using a pair of stacks, for example, reduces the cost of insertion and deletion at the cost of making it harder to move the cursor a long distance.
- You can indicate the order of elements in a sequence by storing a pointer with each value linking it to the one that follows it. In programming, structures designed in this way are called *linked lists*. The pointers that connect one value to the next are called *links*, and the individual records used to store the values and link fields together are called *cells*.
- The conventional way to mark the end of a linked list is to store the pointer constant **NULL** in the link field of the last cell.
- If you are inserting and deleting values from a linked list, it is often convenient to allocate an extra dummy cell at the beginning of the list. The advantage of this

technique is that the existence of the dummy cell reduces the number of special cases you need to consider in your code.

- Insertions and deletions at specified points in a linked list are constant-time operations.
- You can iterate through the cells of a linked list by using the following idiom:

```
for (cellT *cp = list; cp != NULL; cp = cp->link) {
    . . . code using cp . . .
}
```

- Doubly linked lists make it possible to traverse a list efficiently in both directions.
- Linked lists tend to be efficient in execution time but inefficient in their use of memory. In some cases, you may be able to design a hybrid strategy that allows you to combine the execution efficiency of linked lists with the space advantages of arrays.

Review questions

1. True or false: The computational complexity of a program depends only on its algorithmic structure, not on the structures used to represent the data.
2. What does *wysiwyg* stand for?
3. In your own words, describe the purpose of the buffer abstraction used in this chapter.
4. What are the six commands implemented by the editor application? What are the corresponding public members functions in the **EditorBuffer** class?
5. In addition to the methods that correspond to the editor commands, what other public operations are exported by the **EditorBuffer** class?
6. Which editor operations require linear time in the array representation of the editor buffer? What makes those operations slow?
7. Draw a diagram showing the contents of the **before** and **after** stack in the two-stack representation of a buffer that contains the following text, with the cursor positioned as shown:

A B C D | G H I J

8. How is the cursor position indicated in the two-stack representation of the editor buffer?
9. Which editor operations require linear time in the two-stack representation?
10. Define each of the following terms: *cell*, *link*, *linked list*, *dummy cell*.
11. What is the conventional way to mark the end of a linked list?
12. What is meant by a *recursive* data type?
13. What is the purpose of a dummy cell in a linked list?
14. Does the dummy cell go at the beginning or the end of a linked list? Why?
15. What are the five steps required to insert a new character into the linked-list buffer?

16. Draw a diagram showing all the cells in the linked-list representation of a buffer that contains the following text, with the cursor positioned as shown:

H E L | L O

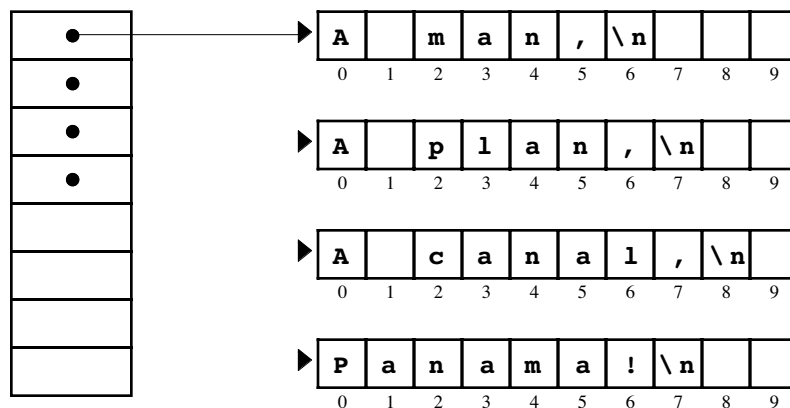
17. Modify the diagram you drew in the preceding exercise to show what happens if you insert the character **x** at the cursor position.
18. What is meant by the phrase *traversing a list*?
19. What is the standard idiom used in C++ to traverse a linked list?
20. Which editor operations require linear time in the linked-list representation of the editor buffer? What makes those operations slow?
21. What modification can you make to the linked-list structure so that all six of the editor operations run in constant time?
22. What is a time-space tradeoff?
23. What is the major drawback to the solution you offered in your answer to question 21? What can you do to improve the situation?

Programming exercises

1. One way to reduce the cost of the array-based implementation is to change the representation from a dynamic array of characters to a dynamic array of lines, each of which is a dynamic array of characters that ends with the newline character, which is written in C++ as `'\n'`. Using this design, for example, the palindromic paragraph

A man,
A plan,
A canal,
Panama!

would look something like this:



Rewrite the **EditorBuffer** class from Figure 10-3 so that it uses this new representation. In doing so, you will need to keep the following points in mind:

- You will need to store length and capacity information for each of the dynamic arrays. This information is not shown in the diagram, and you will have to figure out how best to include it in the data structure.

- The **insertCharacter** and **deleteCharacter** methods need to check whether the character being inserted or deleted is a newline character and modify the buffer structure appropriately.
 - Understanding how best to represent the cursor information will require careful thought. A single integer indicating the index in the overall buffer would not support an efficient implementation.
 - When you complete your implementation, each of the cursor-motion methods should run in $O(1)$ time, just as they do in the original array implementation. The **insertCharacter** and **deleteCharacter** methods should take time proportional to the length of the line for characters other than the newline character.
2. Even though the stacks in the **stackbuf.cpp** implementation of the **EditorBuffer** class (see Figure 10-4) expand dynamically, the amount of character space required in the stacks is likely to be twice as large as that required in the corresponding array implementation. The problem is that each stack must be able to accommodate all the characters in the buffer. Suppose, for example, that you are working with a buffer containing N characters. If you're at the beginning of the buffer, those N characters are in the **after** stack; if you move to the end of the buffer, those N characters move to the **before** stack. As a result, each of the stacks must have a capacity of N characters.

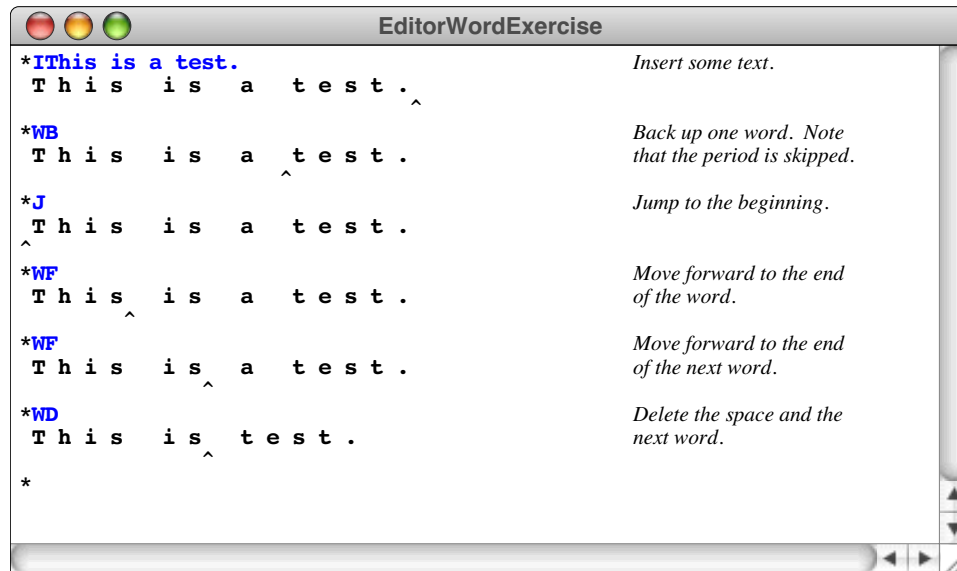
You can reduce the storage requirement in the two-stack implementation of the buffer by storing the two stacks at opposite ends of the same internal array. The **before** stack starts at the beginning of the array, while the **after** stack starts at the end. The two stacks then grow toward each other as indicated by the arrows in the following diagram:



Reimplement the **EditorBuffer** class using this representation (which is, in fact, the design strategy used in many editors today). Make sure that your program continues to have the same computational efficiency as the two-stack implementation in the text and that the buffer space expands dynamically if necessary.

3. Rewrite the editor application given in Figure 10-2 so that the **F**, **B**, and **D** commands take a repetition count specified by a string of digits before the command letter. Thus, the command **17F** would move the cursor forward 17 character positions.
4. Extend the numeric argument facility from exercise 3 so that the numeric argument can be preceded with a negative sign. For the **F** and **B** commands, this facility is not particularly useful because the **-4F** command is the same as **4B**. For the **D** command, however, this extension allows you to delete characters backward using a command like **-3D**, which deletes the three characters before the current cursor position. What changes, if any, do you need to make to the **EditorBuffer** class interface to implement this operation?
5. Extend the editor application so that the **F**, **B**, and **D** commands can be preceded with the letter **w** to indicate word motion. Thus, the command **wF** should move forward to the end of the next word, **wB** should move backward to the beginning of the preceding word, and **wD** should delete characters through the end of the next word. For the purposes of this exercise, a word consists of a consecutive sequence of alphanumeric characters (i.e., letters or digits) and includes any adjacent

nonalphanumeric characters between the cursor and the word. This interpretation is easiest to see in the context of the following example:



To complete this exercise, you will have to extend the **EditorBuffer** class interface in some way. As you design that extension, try to keep in mind the principles of good interface design that were introduced in Chapter 3. After you have designed the interface extension, add the necessary code to **arraybuf.cpp**, **stackbuf.cpp**, and **listbuf.cpp** to implement your changes for each of the representations used in this chapter.

- Most modern editors provide a facility that allows the user to copy a section of the buffer text into an internal storage area and then paste it back in at some other position. For each of the three representations of the buffer given in this chapter, implement the method

```
void EditorBuffer::copy(int count);
```

which stores a copy of the next **count** characters somewhere in the internal structure for the buffer, and the method

```
void EditorBuffer::paste();
```

which inserts those saved characters back into the buffer at the current cursor position. Calling **paste** does not affect the saved text, which means that you can insert multiple copies of the same text by calling **paste** more than once. Test your implementation by adding the commands **C** and **P** to the editor application for the copy and paste operations, respectively. The **C** command should take a numeric argument to specify the number of characters using the technique described in exercise 3.

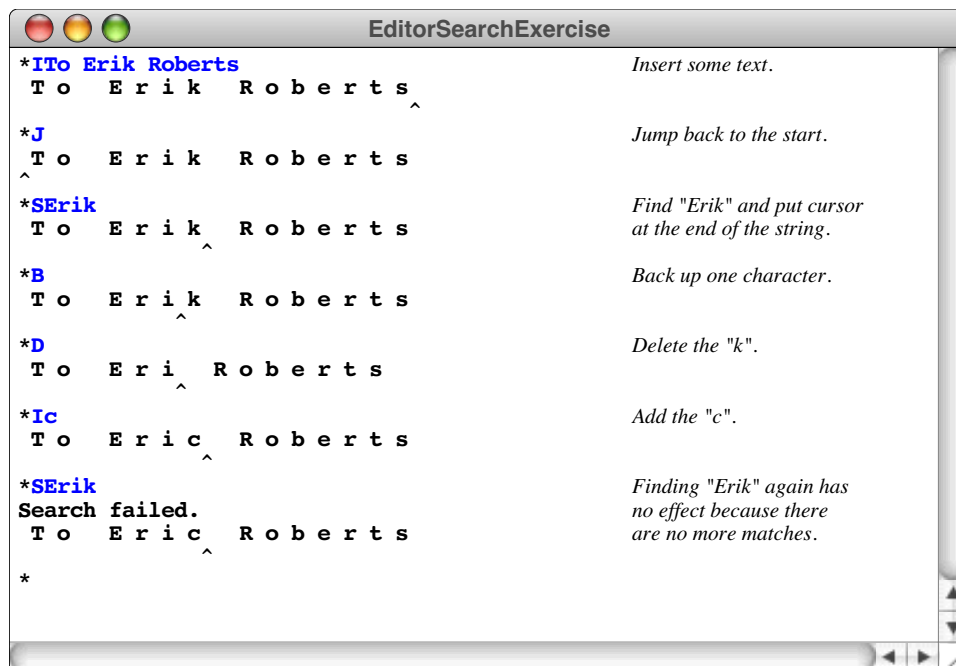
- Editors that support the copy/paste facility described in the preceding exercise usually provide a third operation called *cut* that copies the text from the buffer and then deletes it. Implement a new editor command called **X** that implements the cut operation without making any changes to the **EditorBuffer** class interface beyond those you needed to solve exercise 6.

8. For each of the three representations of the buffer given in this chapter, implement the method

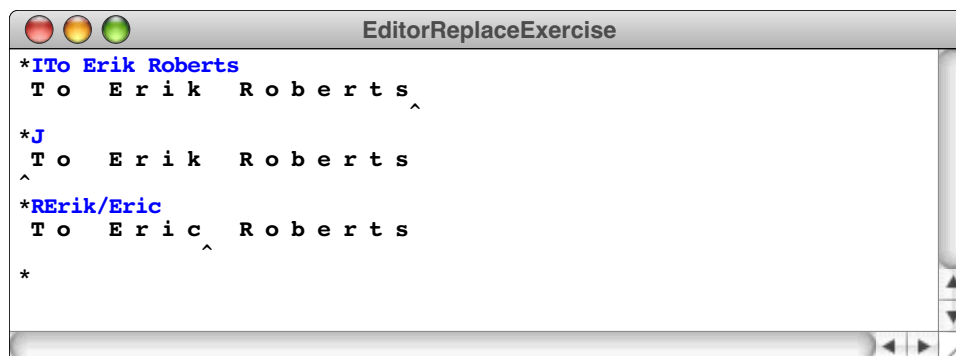
```
bool EditorBuffer::search(string str);
```

When this function is called, it should start searching from the current cursor position, looking for the next occurrence of the string **str**. If it finds it, **search** should leave the cursor after the last character in **str** and return the value **true**. If **str** does not occur between the cursor and the end of the buffer, then **search** should leave the cursor unchanged and return **false**.

To illustrate the operation of **search**, suppose that you have added the **S** command to the **editor.cpp** program so that it calls the **search** function, passing it the rest of the input line. Your program should then be able to match the following sample run:



9. Without making any further changes to the **EditorBuffer** class interface beyond those required for exercise 8, add an **R** command to the editor application that replaces the next occurrence of one string with another, where the two strings are specified after the **R** command separated by a slash, as shown:



10. For each of the three representations of the buffer given in this chapter, implement the method

```
string EditorBuffer::toString();
```

which returns the entire contents of the buffer as a string of characters.

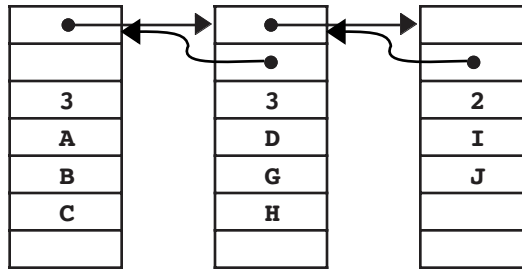
11. The dummy cell strategy described in the text is useful because it reduces the number of special cases in the code. On the other hand, it is not strictly necessary. Write a new implementation of **listbuf.cpp** in which you make the following changes to the design:
- The linked list contains no dummy cell—just a cell for every character.
 - A buffer in which the cursor occurs before the first character is indicated by storing **NULL** in the **cursor** field.
 - Every function that checks the position of the cursor makes a special test for **NULL** and performs whatever special actions are necessary in that case.
12. Implement the **EditorBuffer** class using the strategy described in the section entitled “Doubly linked lists” earlier in this chapter. Be sure to test your implementation as thoroughly as you can. In particular, make sure that you can move the cursor in both directions across parts of the buffer where you have recently made insertions and deletions.
13. The biggest problem with using a doubly linked list to represent the editor buffer is that it is terribly inefficient in terms of space. With two pointers in each cell and only one character, pointers take up 89 percent of the storage, which is likely to represent an unacceptable level of overhead.

The best way around this problem is to combine the array and linked-list models so that the actual structure consists of a doubly linked list of units called *blocks*, where each block contains the following:

- The **prev** and **next** pointers required for the doubly linked list
- The number of characters currently stored in the block
- A fixed-size array capable of holding several characters rather than a single one

By storing several data characters in each block, you reduce the storage overhead, because the pointers take up a smaller fraction of the data. However, since the blocks are of a fixed maximum size, the problem of inserting a new character never requires shifting more than k characters, where k is the **block size** or maximum number of characters per block. Because the block size is a constant, the insertion operation remains $O(1)$. As the block size gets larger, the storage overhead decreases, but the time required to do an insertion increases. In the examples that follow, the block size is assumed to be four characters, although a larger block size would make more sense in practice.

To get a better idea of how this new buffer representation works, consider how you would represent the character data in a block-based buffer. The characters in the buffer are stored in individual blocks, and each block is chained to the blocks that precede and follow it by link pointers. Because the blocks need not be full, there are many possible representations for the contents of a buffer, depending on how many characters appear in each block. For example, the buffer containing the text “**ABCDGHIJ**” might be divided into three blocks, as follows:

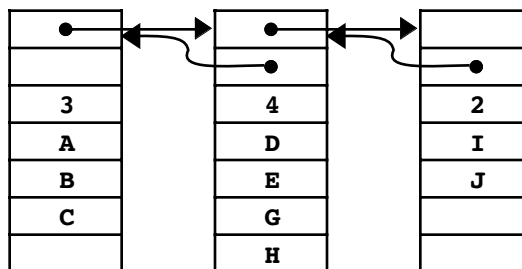


In this diagram, what you see are three blocks, each of which contains a pointer to the next block, a pointer to the previous block, the number of characters currently stored in the block, and four bytes of character storage. The actual definition of the **EditorBuffer** class and the contents of two link fields missing in this diagram (the first backward link and the last forward one) depend on your class representation, which is up to you to design. In particular, your buffer class must include some way to represent the cursor position, which presumably means that the private data members will include a pointer to the current block, as well as an index showing the current position within that block.

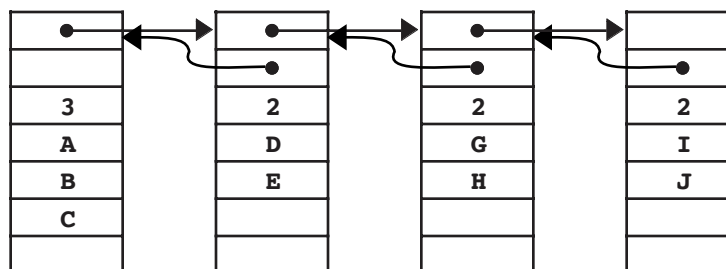
Assume for the moment that the cursor in the previous example follows the **D**, so the buffer contents are

A B C D | G H I J

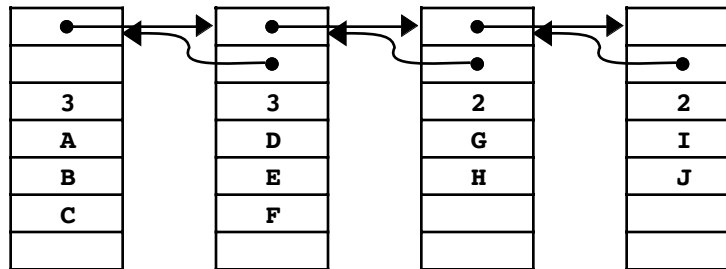
Suppose you want to insert the missing letters, **E** and **F**. Inserting the **E** is a relatively simple matter because the active block has only three characters, leaving room for an extra one. Inserting the **E** into the buffer requires shifting the **G** and the **H** toward the end of the block, but does not require any changes to the pointers linking the blocks themselves. The configuration after inserting the **E** therefore looks like this:



If you now try to insert the missing **F**, however, the problem becomes more complicated. At this point, the current block is full. To make room for the **F**, you need to split the block into two pieces. A simple strategy is to split the block in two, putting half of the characters into each block. After splitting (but before inserting the **F**), the buffer looks like this:



From this point, it is a simple matter to insert the **F** in the second block:



Reimplement the **EditorBuffer** class so that the data representation of the buffer is a linked list of blocks, where each block can hold up to **MAX_BLOCK_SIZE** characters. In writing your implementation, you should be sure to remember the following points:

- You are responsible for designing the data structure for the **EditorBuffer** class. Think hard about the design of your data structure before you start writing the code. Draw pictures. Figure out what the empty buffer looks like. Consider carefully how the data structures change as blocks are split.
- You should choose a strategy for representing the cursor that allows you to represent the possible states of a buffer in a consistent, understandable way. To get a sense of whether your representation works well, make sure that you can answer basic questions about your representation. How does your buffer structure indicate that the cursor is at the beginning of the buffer? What about a cursor at the end? What about a cursor that falls between characters in different blocks? Is there a unique representation for such circumstances, or is there ambiguity in your design? In general, it will help a great deal if you try to simplify your design and avoid introducing lots of special case handling.
- If you have to insert a character into a block that is full, you should divide the block in half before making the insertion. This policy helps ensure that neither of the two resulting blocks starts out being filled, which might immediately require another split when the next character came along.
- If you delete the last character in a block, your program should free the storage associated with that block unless it is the only block in the buffer.
- You should convince yourself that **~EditorBuffer** works, in the sense that all allocated memory is freed and that you never reference data in memory that has been returned to the heap.
- You should explicitly document design decisions in your code.

Chapter 11

Linear Structures

It does not come to me in quite so direct a line as that; it takes a bend or two, but nothing of consequence.

— Jane Austen, *Persuasion*, 1818

The **Stack** and **Queue** classes introduced in Chapter 4 are examples of a general category of abstract data types called **linear structures**, in which the elements are arranged in a linear order. This chapter looks at several representations for these ADTs and considers how the choice of representation affects efficiency.

Because the elements in a linear structure are arranged in an array-like order, using arrays to represent them seems like an obvious choice. Indeed, the **CharStack** class presented in Chapter 9 is implemented using an array as the underlying representation. Arrays, however, are not the only option. Stacks, queues, and vectors can also be implemented using a linked list much like the one used to implement the editor buffer in Chapter 10. By studying the linked-list implementation of these structures, you will increase your understanding not only of how linked lists work, but also of how you can apply them in practical programming contexts.

This chapter has another purpose as well. As noted in Chapter 4, each of these ADTs is a container classes, which means that it can contain data values supplied by the client. To be of maximal utility, a container class should allow the client to store any type of data. The **template** keyword in C++ makes it easy to design a class that can be used for many different types. Such a class is said to be **polymorphic**.

11.1 Reimplementing stacks as a template class

The **CharStack** class in Chapter 9 defines the relevant operations that all stacks require, but is limited because it can only store elements of type **char**. This limitation is unfortunate because the behavior of a stack (or any of the other container classes for that matter) is not dependent on the type of the values being stored. The client-supplied values are pushed onto the stack, stored internally, and then popped as requested. The operations of pushing and popping values are always the same, whether the values being stored on the stack are strings, doubles, or a user-defined type. The same is true for the enqueue and dequeue operations on a queue.

To be useful to a range of clients, the stack and other container classes should be designed to allow any type of data to be stored. C++ includes a facility for defining **templates**, which are structures that take the base type as a parameter. You have been using such templates ever since Chapter 4, but you have not yet seen how to create them.

A template is basically just a pattern. The pattern describes the interface and implementation for a class in terms of a placeholder, which is specified in angle brackets as the template parameter. The placeholder in this case is the particular element type being stored. You create a class from the pattern by filling in that placeholder with the desired element type.

The interface of a class template

Changing a non-template class into a template one involves some simple syntactic changes. For example, if you want to replace the **CharStack** class from Chapter 9 with a general **Stack** template, you begin by replacing the name **CharStack** with **Stack** and then add the following line before the class definition:

```
template <typename ElemType>
```

The **template** keyword indicates that the entire syntactic unit that follows this line—in this case the class definition—is part of a template pattern that can be used for many different values of the **ElemType** parameter. In the class definition that follows, you use the placeholder name **ElemType** wherever you need to refer to the type of element being

stored. Thus, as you convert the **CharStack** class definition to a more general template form, you would replace the appearance of the **char** type in the prototypes for **push**, **pop**, and **peek** with **ElemType**.

You do the same with each of the methods in the implementation file. For example, where the **CharStack** implementation includes the method

```
void CharStack::push(char ch) {
    if (count == capacity) expandCapacity();
    elements[count++] = ch;
}
```

you need to replace that with the following template method:

```
template <typename ElemType>
void Stack<ElemType>::push(ElemType elem) {
    if (count == capacity) expandCapacity();
    elements[count++] = elem;
}
```

Note that the body itself doesn't change in any substantive way; the only difference in the body of the implementation is that the parameter name **ch**, which is no longer appropriate for a general stack implementation, has been replaced with a more generic name.

There is, however, one additional change that you need to make in the class definition. In order for the compiler to create the specific classes that are instances of the general template, it must have access to both the interface and the implementation. To achieve this goal, you could include the implementation directly in the header file, but this would have the undesirable effect of exposing clients to the details of the implementation. A useful strategy for providing those details to the compiler while hiding them from the client is to use the **#include** facility to read in the implementation when it is needed.

You can see this technique in Figure 11-1, which specifies the interface for the generic **Stack** class. Both the private section of the class and the actual implementation are included from separate files named **stackpriv.h** and **stackimpl.cpp**, respectively. The contents of these files appear in Figures 11-2 and 11-3.

11.2 Reimplementing stacks using linked lists

Although arrays are the most common underlying representation for stacks, it is also possible to implement the **Stack** class using linked lists. If you do so, the conceptual representation for the empty stack is simply the **NULL** pointer:

stack NULL

When you push a new element onto the stack, the element is simply added to the front of the linked-list chain. Thus, if you push the element e_1 onto an empty stack, that element is stored in a new cell which becomes the only link in the chain:

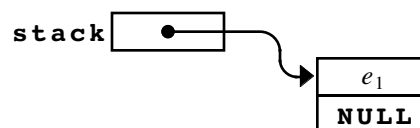


Figure 11-1 Interface for the Stack class

```

/*
 * File: stack.h
 * -----
 * This interface defines a general stack abstraction that uses
 * templates so that it can work with any element type.
 */

#ifndef _stack_h
#define _stack_h

/*
 * Template class: Stack<ElemType>
 * -----
 * This class template models a stack, which is a linear collection
 * of values stacked one on top of the other. Values are added and
 * removed only from the top of the stack. The fundamental stack
 * operations are push (add to top) and pop (remove from top).
 * Because values are added and removed from the same end of the
 * stack, the last value pushed on a stack is the first value that
 * is popped. Stacks therefore operate in a last-in-first-out (LIFO)
 * order. For maximum generality, the Stack class is defined using
 * a template that allows the client to define a stack that contains
 * any type of value, as in Stack<string> or Stack<stateT>.
 */

template <typename ElemType>
class Stack {
public:

/*
 * Constructor: Stack
 * Usage: Stack<int> stack;
 * -----
 * The constructor initializes a new empty stack containing
 * the specified value type.
 */
    Stack();

/*
 * Destructor: ~Stack
 * Usage: (usually implicit)
 * -----
 * The destructor deallocates any heap storage associated
 * with this stack.
 */
    ~Stack();

/*
 * Method: size
 * Usage: nElems = stack.size();
 * -----
 * Returns the number of elements in this stack.
 */
    int size();

```



```

/*
 * Method: isEmpty
 * Usage: if (stack.isEmpty()) . . .
 * -----
 * Returns true if this stack contains no elements, and false
 * otherwise.
 */
    bool isEmpty();

/*
 * Method: clear
 * Usage: stack.clear();
 * -----
 * This method removes all elements from this stack.
 */
    void clear();

/*
 * Method: push
 * Usage: stack.push(elem);
 * -----
 * Pushes the specified element onto this stack.
 */
    void push(ElemType elem);

/*
 * Method: pop
 * Usage: topElem = stack.pop();
 * -----
 * Removes the top element from this stack and returns it.
 * Raises an error if called on an empty stack.
 */
    ElemType pop();

/*
 * Method: peek
 * Usage: topElem = stack.peek();
 * -----
 * Returns the value of top element from this stack without
 * removing it. Raises an error if called on an empty stack.
 */
    ElemType peek();

private:
#include "stackpriv.h"
};

#include "stackimpl.cpp"
#endif

```

Figure 11-2 Private section of the Stack class for the array-based representation

```

/*
 * File: stackpriv.h
 * -----
 * This file contains the private section of the Stack template
 * class. Including this information in a separate file means
 * that clients don't need to look at these details.
 */

/*
 * Implementation notes: Stack data structure
 * -----
 * The elements of the stack are stored in a dynamic array of
 * the specified element type. If the space in the array is ever
 * exhausted, the implementation doubles the array capacity.
 */

/* Constants */
    static const int INITIAL_CAPACITY = 100;

/* Instance variables */
    ElemType *elements;    /* A dynamic array of the elements */
    int capacity;          /* The allocated size of the array */
    int count;             /* The number of elements on the stack */

/* Private method prototypes */
    void expandCapacity();

```

Figure 11-3 Implementation of the Stack class using the array-based representation

```

/*
 * File: stackimpl.cpp
 * -----
 * This file contains the array-based implementation of the
 * Stack class.
 */

#ifdef _stack_h

/*
 * Implementation notes: Stack constructor
 * -----
 * The constructor must allocate the array storage for the stack
 * elements and initialize the fields of the object.
 */

template <typename ElemType>
Stack<ElemType>::Stack() {
    capacity = INITIAL_CAPACITY;
    elements = new ElemType[capacity];
    count = 0;
}

```

```

/*
 * Implementation notes: pop, peek
 * -----
 * These methods must check for an empty stack and report an
 * error if there is no top element.
 */

template <typename ElemType>
ElemType Stack<ElemType>::pop() {
    if (isEmpty()) {
        Error("pop: Attempting to pop an empty stack");
    }
    return elements[--count];
}

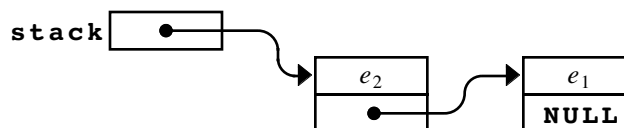
template <typename ElemType>
ElemType Stack<ElemType>::peek() {
    if (isEmpty()) {
        Error("peek: Attempting to peek at an empty stack");
    }
    return elements[count - 1];
}

/*
 * Implementation notes: expandCapacity
 * -----
 * This private method doubles the capacity of the elements array
 * whenever it runs out of space. To do so, it must allocate a new
 * array, copy all the elements from the old array to the new one,
 * and free the old storage.
 */

template <typename ElemType>
void Stack<ElemType>::expandCapacity() {
    capacity *= 2;
    ElemType *oldElements = elements;
    elements = new ElemType[capacity];
    for (int i = 0; i < count; i++) {
        elements[i] = oldElements[i];
    }
    delete[] oldElements;
}

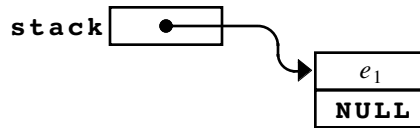
```

Pushing a new element onto the stack adds that element at the beginning of the chain. The steps involved are the same as those required to insert a character into a linked-list buffer. You first allocate a new cell, then enter the data, and, finally, update the link pointers so that the new cell becomes the first element in the chain. Thus, if you push the element e_2 on the stack, you get the following configuration:



In the linked-list representation, the **pop** operation consists of removing the first cell in the chain and returning the value stored there. Thus, a **pop** operation from the stack

shown in the preceding diagram returns e_2 and restores the previous state of the stack, as follows:



The data structure for the **Stack** class now consists of a single instance variable used to store the start of the linked list representing the stack. The private section of the **Stack** therefore consists of the structure definition for the **cellT** type, the pointer to the start of the list, and an integer that holds the number of elements so that the implementation doesn't have to count them each time. The contents of the revised **stackpriv.h** file are shown in Figure 11-4.

Once you have defined the data structure, you can then move on to reimplement the **Stack** class methods so that they operate on the new data representation. The first function that needs to change is the constructor, which must initialize the **start** field to **NULL** and the **count** field to 0, like this:

```
template <typename ElemType>
Stack<ElemType>::Stack() {
    start = NULL;
    count = 0;
}
```

The remaining methods are equally straightforward. The revised implementation of the **push** method must allocate a new cell and then add it to the front of the list. The **pop** method removes the first cell from the list and returns the value that cell contained. Since these operations only change a few pointers at the start of the list, both can be performed in constant time. The complete implementation of the **Stack** class using linked lists is shown in Figure 11-5.

Figure 11-4 Private section of the Stack class for the list-based representation

```
/*
 * File: stackpriv.h
 * -----
 * This file contains the private section for the list-based
 * implementation of the Stack class. Including this section
 * in a separate file means that clients don't need to look
 * at these details.
 */

/* Type for linked list cell */

struct cellT {
    ElemType data;
    cellT *link;
};

/* Instance variables */

cellT *list;          /* Beginning of the list of elements */
int count;            /* Number of elements in the stack */
```

Figure 11-5 Implementation of the Stack class using the list-based representation

```

/*
 * File: stackimpl.cpp
 * -----
 * This file contains the list-based implementation of the
 * Stack class.
 */

#ifdef _stack_h

/*
 * Implementation notes: Stack constructor
 * -----
 * The constructor must create an empty linked list and then
 * initialize the fields of the object.
 */

template <typename ElemType>
Stack<ElemType>::Stack() {
    list = NULL;
    count = 0;
}

/*
 * Implementation notes: ~Stack destructor
 * -----
 * The destructor frees any memory allocated by the implementation.
 * Freeing this memory guarantees that the stack abstraction
 * will not "leak memory" in the process of running an
 * application. Because clear frees each element it processes,
 * this implementation of the destructor uses that method.
 */

template <typename ElemType>
Stack<ElemType>::~~Stack() {
    clear();
}

/*
 * Implementation notes: size, isEmpty, clear
 * -----
 * These implementations should be self-explanatory.
 */

template <typename ElemType>
int Stack<ElemType>::size() {
    return count;
}

template <typename ElemType>
bool Stack<ElemType>::isEmpty() {
    return count == 0;
}

template <typename ElemType>
void Stack<ElemType>::clear() {
    while (count > 0) {
        pop();
    }
}

}

```

```
/*
 * Implementation notes: push
 * -----
 * This method chains a new element onto the list
 * where it becomes the top of the stack.
 */

template <typename ElemType>
void Stack<ElemType>::push(ElemType elem) {
    cellT *cell = new cellT;
    cell->data = elem;
    cell->link = list;
    list = cell;
    count++;
}

/*
 * Implementation notes: pop, peek
 * -----
 * These methods must check for an empty stack and report an
 * error if there is no top element. The pop method must free
 * the cell to ensure that the implementation does not waste
 * heap storage as it executes.
 */

template <typename ElemType>
ElemType Stack<ElemType>::pop() {
    if (isEmpty()) {
        Error("pop: Attempting to pop an empty stack");
    }
    cellT *cell = list;
    ElemType result = cell->data;
    list = list->link;
    count--;
    delete cell;
    return result;
}

template <typename ElemType>
ElemType Stack<ElemType>::peek() {
    if (isEmpty()) {
        Error("peek: Attempting to peek at an empty stack");
    }
    return list->data;
}

#endif
```

11.3 Implementing queues

As you know from Chapter 4, stacks and queues are very similar structures. The only difference between them is in the order in which elements are processed. A stack uses a last-in/first-out (LIFO) discipline in which the last item pushed is always the first item popped. A queue adopts a first-in/first-out (FIFO) model that more closely resembles a waiting line. The interfaces for stacks and queues are also extremely similar. As you can see from the public section of the `queue.h` interface in Figure 11-6, the only things that have changed from the `stack.h` interface in Figure 11-1 are the names of two methods (**push** is now **enqueue** and **pop** is **dequeue**) and the comments that describe how elements in each structure are ordered.

Given the conceptual similarity of these structures and their interfaces, it is probably not surprising that both stacks and queues can be implemented using either an array-based or list-based strategy. With each of these models, however, the implementation of a queue has subtleties that don't arise in the case of a stack. These differences arise from the fact that all the operations on a stack occur at the same end of the internal data structure. In a queue, the **enqueue** operation happens at one end, and the **dequeue** operation happens at the other.

An array-based implementation of queues

In light of the fact that actions in a queue are no longer confined to one end of an array, you need two indices to keep track of the head and tail positions in the queue. The private instance variables therefore look like this:

```
ElemType *elements;  /* A dynamic array of the elements */
int capacity;        /* The allocated size of the array */
int head;            /* The index of the head of the queue */
int tail;            /* The index of the tail of the queue */
```

In this representation, the **head** field holds the index of the next element to come out of the queue, and the **tail** field holds the index of the next free slot. In an empty queue, it is clear that the **tail** field should be 0 to indicate the initial position in the array, but what about the **head** field? For convenience, the traditional strategy is to set the **head** field to 0 as well. When queues are defined in this way, having the **head** and **tail** fields be equal indicates that the queue is empty.

Given this representation strategy, the **Queue** constructor looks like this:

```
template <typename ElemType>
Queue<ElemType>::Queue() {
    head = tail = 0;
}
```

Although it is tempting to think that the **enqueue** and **dequeue** methods will look almost exactly like their **push** and **pop** counterparts in the **Stack** class, you will run into several problems if you simply try to copy the existing code. As is often the case in programming, it makes more sense to use diagrams to make sure you understand exactly how the queue should operate before you start writing the code.

To get a sense of how this representation of a queue works, imagine that the queue represents a waiting line, similar to one in the simulation from Chapter 4. From time to time, a new customer arrives and is added to the queue. Customers waiting in line are periodically served at the head end of the queue, after which they leave the waiting line entirely. How does the queue data structure respond to each of these operations?

Figure 11-6 Interface for the Queue class

```

/*
 * File: queue.h
 * -----
 * This interface defines a general queue abstraction that uses
 * templates so that it can work with any element type.
 */

#ifndef _queue_h
#define _queue_h

/*
 * Template class: Queue<ElemType>
 * -----
 * This class template models a queue, which is a linear collection
 * of values that resemble a waiting line. Values are added at
 * one end of the queue and removed from the other. The fundamental
 * operations are enqueue (add to the tail of the queue) and dequeue
 * (remove from the head of the queue). Because a queue preserves
 * the order of the elements, the first value enqueued is the first
 * value dequeued. Queues therefore operate in a first-in-first-out
 * (FIFO) order. For maximum generality, the Queue class is defined
 * using a template that allows the client to define a queue that
 * contains any type of value, as in Queue<string> or Queue<taskT>.
 */

template <typename ElemType>
class Queue {
public:
    /*
     * Constructor: Queue
     * Usage: Queue<int> queue;
     * -----
     * The constructor initializes a new empty queue containing
     * the specified value type.
     */
    Queue();

    /*
     * Destructor: ~Queue
     * Usage: (usually implicit)
     * -----
     * The destructor deallocates any heap storage associated
     * with this queue.
     */
    ~Queue();

    /*
     * Method: size
     * Usage: nElems = queue.size();
     * -----
     * Returns the number of elements in this queue.
     */
    int size();

```



```
/*
 * Method: isEmpty
 * Usage: if (queue.isEmpty()) . . .
 * -----
 * Returns true if this queue contains no elements, and false
 * otherwise.
 */

    bool isEmpty();

/*
 * Method: clear
 * Usage: queue.clear();
 * -----
 * This method removes all elements from this queue.
 */

    void clear();

/*
 * Method: enqueue
 * Usage: queue.enqueue(elem);
 * -----
 * Adds the specified element to the end of this queue.
 */

    void enqueue(ElemType elem);

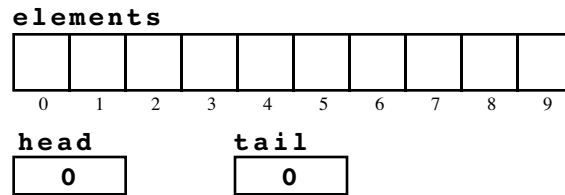
/*
 * Method: dequeue
 * Usage: first = queue.dequeue();
 * -----
 * Removes the first element from this queue and returns it.
 * Raises an error if called on an empty queue.
 */

    ElemType dequeue();

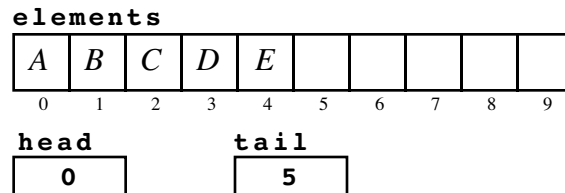
/*
 * Method: peek
 * Usage: topElem = queue.peek();
 * -----
 * Returns the value of first element from this queue without
 * removing it.  Raises an error if called on an empty queue.
 */

    ElemType peek();
#include "queuepriv.h"
};
#include "queueimpl.cpp"
#endif
```

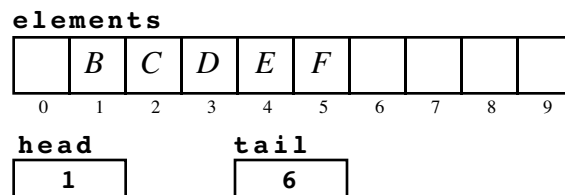
Assuming that the queue is empty at the beginning, its internal structure looks like this:



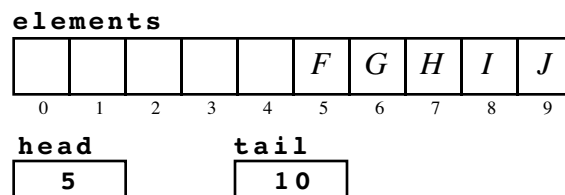
Suppose now that five customers arrive, indicated by the letters *A* through *E*. Those customers are enqueued in order, which gives rise to the following configuration:



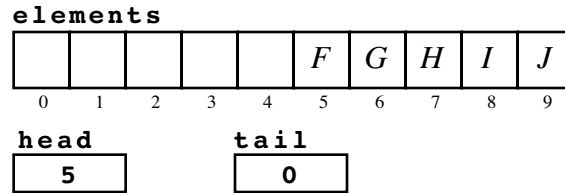
The value 0 in the **head** field indicates that the first customer in the queue is stored in position 0 of the array; the value 5 in **tail** indicates that the next customer will be placed in position 5. So far, so good. At this point, suppose that you alternately serve a customer at the beginning of the queue and then add a new customer to the end. For example, customer *A* is dequeued and customer *F* arrives, which leads to the following situation:



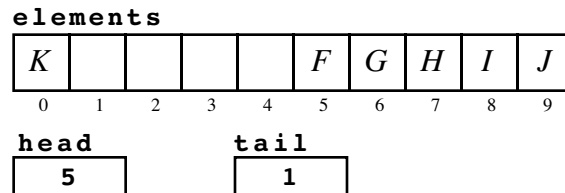
Imagine that you continue to serve one customer just before the next customer arrives and that this trend continues until customer *J* arrives. The internal structure of the queue then looks like this:



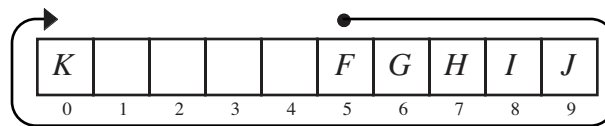
At this point, you've got a bit of a problem. There are only five customers in the queue, but you have used up all the available space. The **tail** field is pointing beyond the end of the array. On the other hand, you now have unused space at the beginning of the array. Thus, instead of incrementing **tail** so that it indicates the nonexistent position 10, you can “wrap around” from the end of the array back to position 0, as follows:



From this position, you have space to enqueue customer *K* in position 0, which leads to the following configuration:

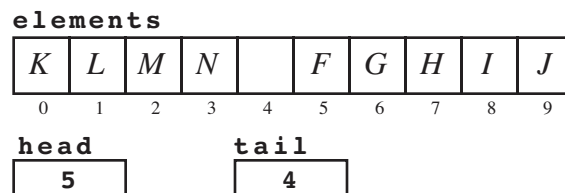


If you allow the elements in the queue to wrap around from the end of the array to the beginning, the active elements always extend from the **head** index up to the position immediately preceding the **tail** index, as illustrated in this diagram:

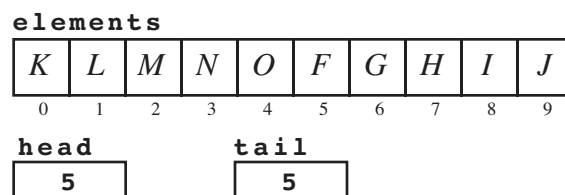


Because the ends of the array act as if they were joined together, programmers call this representation a **ring buffer**.

The only remaining issue you need to consider before you can write the code for **enqueue** and **dequeue** is how to check whether the queue is completely full. Testing for a full queue is trickier than you might expect. To get a sense of where complications might arise, suppose that three more customers arrive before any additional customers are served. If you enqueue the customers *L*, *M*, and *N*, the data structure looks like this:



At this point, it appears as if there is one extra space. What happens, though, if customer *O* arrives at this moment? If you followed the logic of the earlier enqueue operations, you would end up in the following configuration:



This queue looks empty.

The queue array is now completely full. Unfortunately, whenever the **head** and **tail** fields have the same value, as they do in this diagram, the queue is considered to be empty. There is no way to tell from the contents of the queue structure itself which of the two conditions—empty or full—actually applies, because the data values look the same in each case. Although you can fix this problem by adopting a different definition for the empty queue and writing some special-case code, the traditional approach is to limit the number of elements in the queue to one less than the number of elements in the array, and to expand the capacity whenever that slightly lower limit is reached.

The code for the array implementation of the **Queue** class template is shown in Figure 11-7. It is important to observe that the code does not explicitly test the array indices to see whether they wrap around from the end of the array to the beginning. Instead, the code makes use of the **%** operator to compute the correct index automatically. The technique of using remainders to reduce the result of a computation to a small, cyclical range of integers is an important mathematical technique called **modular arithmetic**.

Figure 11-7 Implementation of the **Queue** class using the array-based representation

```

/*
 * File: queueimpl.cpp
 * -----
 * This file contains the array-based implementation of the
 * Queue class.
 */

#ifdef _queue_h

/*
 * Implementation notes: Queue constructor
 * -----
 * The constructor must allocate the array storage for the queue
 * elements and initialize the fields of the object.
 */

template <typename ElemType>
Queue<ElemType>::Queue() {
    capacity = INITIAL_CAPACITY;
    elements = new ElemType[capacity];
    head = 0;
    tail = 0;
}

/*
 * Implementation notes: ~Queue destructor
 * -----
 * The destructor frees any memory that is allocated by the
 * implementation. Freeing this memory guarantees the client
 * that the queue abstraction will not "leak memory" in the
 * process of running an application.
 */

template <typename ElemType>
Queue<ElemType>::~Queue() {
    delete[] elements;
}

```

```
/*
 * Implementation notes: size
 * -----
 * The size of the queue can be calculated from the head and tail
 * indices by using modular arithmetic.
 */

template <typename ElemType>
int Queue<ElemType>::size() {
    return (tail + capacity - head) % capacity;
}

/*
 * Implementation notes: isEmpty
 * -----
 * The queue is empty whenever the head and tail pointers are
 * equal. Note that this interpretation means that the queue
 * cannot be allowed to fill the capacity entirely and must
 * always leave one unused space.
 */

template <typename ElemType>
bool Queue<ElemType>::isEmpty() {
    return head == tail;
}

/*
 * Implementation notes: clear
 * -----
 * The clear method need not take account of where in the
 * ring buffer any existing data is stored and can simply
 * set the head and tail index back to the beginning.
 */

template <typename ElemType>
void Queue<ElemType>::clear() {
    head = tail = 0;
}

/*
 * Implementation notes: enqueue
 * -----
 * This method must first check to see whether there is
 * enough room for the element and expand the array storage
 * if necessary. Because it is otherwise impossible to
 * differentiate the case when a queue is empty from when
 * it is completely full, this implementation expands the
 * queue when the size is one less than the capacity.
 */

template <typename ElemType>
void Queue<ElemType>::enqueue(ElemType elem) {
    if (size() == capacity - 1) expandCapacity();
    elements[tail] = elem;
    tail = (tail + 1) % capacity;
}
```

```

/*
 * Implementation notes: dequeue, peek
 * -----
 * These methods must check for an empty queue and report an
 * error if there is no first element.
 */

template <typename ElemType>
ElemType Queue<ElemType>::dequeue() {
    if (isEmpty()) {
        Error("dequeue: Attempting to dequeue an empty queue");
    }
    ElemType result = elements[head];
    head = (head + 1) % capacity;
    return result;
}

template <typename ElemType>
ElemType Queue<ElemType>::peek() {
    if (isEmpty()) {
        Error("peek: Attempting to peek at an empty queue");
    }
    return elements[head];
}

/*
 * Implementation notes: expandCapacity
 * -----
 * This private method doubles the capacity of the elements array
 * whenever it runs out of space. To do so, it must allocate a new
 * array, copy all the elements from the old array to the new one,
 * and free the old storage. Note that this implementation also
 * shifts all the elements back to the beginning of the array.
 */

template <typename ElemType>
void Queue<ElemType>::expandCapacity() {
    int count = size();
    capacity *= 2;
    ElemType *oldElements = elements;
    elements = new ElemType[capacity];
    for (int i = 0; i < count; i++) {
        elements[i] = oldElements[(head + i) % capacity];
    }
    head = 0;
    tail = count;
    delete[] oldElements;
}

```

The one file in the array-based queue definition that has not yet been specified is the **queuepriv.h** file that contains the private data for this class. Although you already know the instance variables it contains, it is worth showing the contents of the file as an illustration of the sort of comments that go into the private section. This section defines the underlying data representation and is therefore logically part of the implementation. The primary audience for any comments included in the **queuepriv.h** file is that set of programmers who may need to maintain this code. Those comments should include anything special or complex about the representation, as illustrated in Figure 11-8.

Figure 11-8 Private section of the queue class for the array-based representation

```
/*
 * File: queuepriv.h
 * -----
 * This file contains the private section of the Queue template
 * class. Including this information in a separate file means
 * that clients don't need to look at these details.
 */

/*
 * Implementation notes: Queue data structure
 * -----
 * The array-based queue stores the elements in successive index
 * positions in an array, just as a stack does. What makes the
 * queue structure more complex is the need to avoid shifting
 * elements as the queue expands and contracts. In the array
 * model, this goal is achieved by keeping track of both the
 * head and tail indices. The tail index increases by one each
 * time an element is enqueued, and the head index increases by
 * one each time an element is dequeued. Each index therefore
 * marches toward the end of the allocated array and will
 * eventually reach the end. Rather than allocate new memory,
 * this implementation lets each index wrap around back to the
 * beginning as if the ends of the array of elements were joined
 * to form a circle. This representation is called a ring buffer.
 *
 * The elements of the queue are stored in a dynamic array of
 * the specified element type. If the space in the array is ever
 * exhausted, the implementation doubles the array capacity.
 * Note that the queue capacity is reached when there is still
 * one unused element in the array. If the queue is allowed to
 * fill completely, the head and tail indices will have the same
 * value, and the queue will appear empty.
 */

/* Constants */

    static const int INITIAL_CAPACITY = 100;

/* Instance variables */

    ElemType *elements; /* A dynamic array of the elements */
    int capacity;        /* The allocated size of the array */
    int head;            /* The index of the head of the queue */
    int tail;            /* The index of the tail of the queue */

/* Private method prototypes */

    void expandCapacity();
```

Linked-list representation of queues

The queue class also has a simple representation using list structure. To illustrate the basic approach, the elements of the queue are stored in a list beginning at the head of the queue and ending at the tail. To allow both **enqueue** and **dequeue** to run in constant time, the **Queue** object must keep a pointer to both ends of the queue. The private instance variables are therefore defined as shown in the revised version of **queuepriv.h** shown in Figure 11-9. The data diagram drawn in characters in **queuepriv.h** is likely to convey more information to the implementer than the surrounding text. Such diagrams are difficult to produce, but they offer enormous value to the reader.

Given a modern word processor and a drawing program, it is possible to produce much more detailed diagrams than you can make using ASCII characters alone. If you are

Figure 11-9 Private section of the Queue class for the list-based representation

```

/*
 * File: queuepriv.h
 * -----
 * This file contains the private section for the list-based
 * implementation of the Queue class. Including this section
 * in a separate file means that clients don't need to look
 * at these details.
 */

/*
 * Implementation notes: Queue data structure
 * -----
 * The list-based queue uses a linked list to store the elements
 * of the queue. To ensure that adding a new element to the tail
 * of the queue is fast, the data structure maintains a pointer
 * to the last cell in the queue as well as the first. If the
 * queue is empty, the tail pointer is always set to be NULL.
 *
 * The following diagram illustrates the structure of a queue
 * containing two elements, A and B.
 *
 *      +-----+      +-----+      +-----+
 * head |  o---+----->|  A  | +---==>|  B  |
 *      +-----+      +-----+      +-----+
 * tail |  o---+-----+  |  o---+-----+  |  NULL
 *      +-----+      +-----+      +-----+
 *
 *                               +-----+
 *                               |
 *                               +-----+
 */

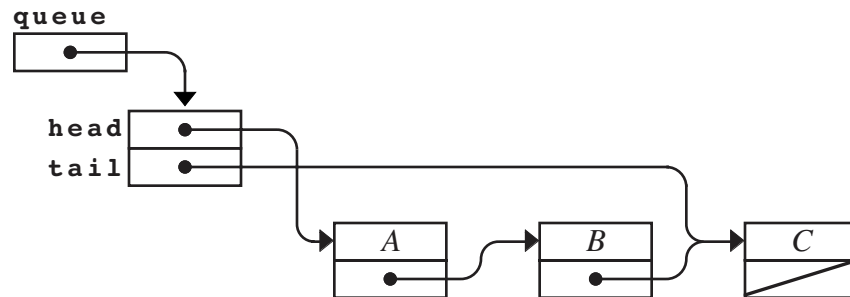
/* Type for linked list cell */
struct cellT {
    ElemType data;
    cellT *link;
};

/* Instance variables */

cellT *head;      /* Pointer to the cell at the head */
cellT *tail;      /* Pointer to the cell at the tail */
int count;        /* Number of elements in the queue */

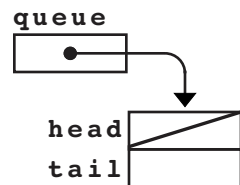
```


designing data structures for a large and complex system, it probably makes sense to create these diagrams and include them as part of the extended documentation of a package, ideally on a web page. Here, for example, is a somewhat more readable picture of a queue containing the customers *A*, *B*, and *C*:

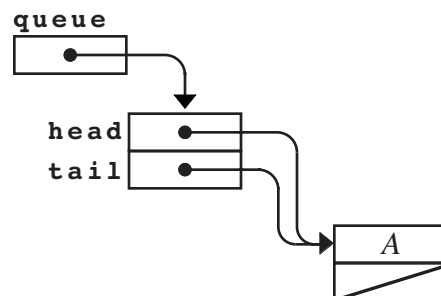


The code for the linked-list implementation of queues appears in Figure 11-6. On the whole, the code is reasonably straightforward, particularly if you use the linked-list implementation of stacks as a model. The diagram of the internal structure provides the essential insights you need to understand how to implement each of the queue operations. The **enqueue** operation, for example, adds a new cell after the one marked by the **tail** pointer and then updates the **tail** pointer so that it continues to indicate the end of the list. The **dequeue** operation consists of removing the cell addressed by the **head** pointer and returning the value in that cell.

The only place where the implementation gets tricky is in the representation of the empty queue. The most straightforward approach is to indicate an empty queue by storing **NULL** in the **head** pointer, as follows:

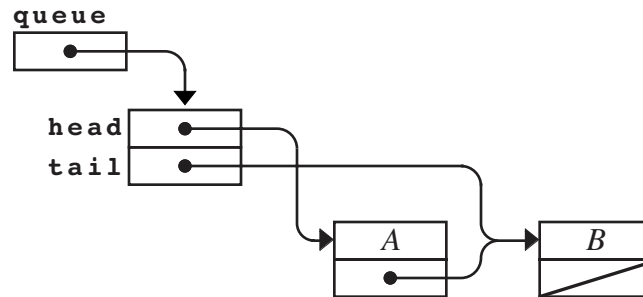


The **enqueue** implementation must check for the empty queue as a special case. If the **head** pointer is **NULL**, **enqueue** must set both the **head** and **tail** pointers so that they point to the cell containing the new element. Thus, if you were to enqueue the customer *A* into an empty queue, the internal structure of the pointers at the end of the **enqueue** operation would look like this:



If you make another call to **enqueue**, the **head** pointer is no longer **NULL**, which means that the implementation no longer has to perform the special-case action for the empty queue. Instead, the **enqueue** implementation uses the **tail** pointer to find the end of the

linked-list chain and adds the new cell at that point. For example, if you enqueue the customer *B* after customer *A*, the resulting structure looks like this:



A complete implementation of the list-based queue structure appears in Figure 11-10.

Figure 11-10 Implementation of the `Queue` class using the list-based representation

```

/*
 * File: queueimpl.cpp
 * -----
 * This file contains the list-based implementation of the
 * Queue class.
 */

#ifdef _queue_h

/*
 * Implementation notes: Queue constructor
 * -----
 * The constructor must create an empty linked list and then
 * initialize the fields of the object.
 */

template <typename ElemType>
Queue<ElemType>::Queue() {
    head = tail = NULL;
    count = 0;
}

/*
 * Implementation notes: ~Queue destructor
 * -----
 * The destructor frees any memory that is allocated by the
 * implementation. Freeing this memory guarantees the client
 * that the queue abstraction will not "leak memory" in the
 * process of running an application. Because clear frees
 * each element it processes, this implementation of the
 * destructor simply calls that method.
 */

template <typename ElemType>
Queue<ElemType>::~~Queue() {
    clear();
}

```

```
/*
 * Implementation notes: size
 * -----
 * In order to return the size in constant time, it is necessary
 * to store the count in the data structure and keep it updated
 * on each call to enqueue and dequeue.
 */

template <typename ElemType>
int Queue<ElemType>::size() {
    return count;
}

/*
 * Implementation notes: isEmpty
 * -----
 * This code uses the traditional head == tail test for an empty
 * stack; testing the value of count would work just as well.
 */

template <typename ElemType>
bool Queue<ElemType>::isEmpty() {
    return head != tail;
}

/*
 * Implementation notes: clear
 * -----
 * This code calls dequeue to make sure the cells are freed.
 */

template <typename ElemType>
void Queue<ElemType>::clear() {
    while (count > 0) {
        dequeue();
    }
}

/*
 * Implementation notes: enqueue
 * -----
 * This method allocates a new list cell and chains it in
 * at the tail of the queue. If the queue is currently empty,
 * the new cell must also become the head pointer in the queue.
 */

template <typename ElemType>
void Queue<ElemType>::enqueue(ElemType elem) {
    cellT *cell = new cellT;
    cell->data = elem;
    cell->link = NULL;
    if (head == NULL) {
        head = cell;
    } else {
        tail->link = cell;
    }
    tail = cell;
    count++;
}
```

```

/*
 * Implementation notes: dequeue, peek
 * -----
 * These methods must check for an empty queue and report an
 * error if there is no first element. The dequeue method
 * must also check for the case in which the queue becomes
 * empty and set both the head and tail pointers to NULL.
 */

template <typename ElemType>
ElemType Queue<ElemType>::dequeue() {
    if (isEmpty()) {
        Error("dequeue: Attempting to dequeue an empty queue");
    }
    cellT *cell = head;
    ElemType result = cell->data;
    head = cell->link;
    if (head == NULL) tail = NULL;
    count--;
    delete cell;
    return result;
}

template <typename ElemType>
ElemType Queue<ElemType>::peek() {
    if (isEmpty()) {
        Error("peek: Attempting to peek at an empty queue");
    }
    return head->data;
}

#endif

```

11.4 Implementing vectors

The **Vector** class introduced in Chapter 4 is another example of a linear structure. In many respects, the implementation of the **Vector** class is similar to that for the stack and queue abstractions you have already seen in this chapter. As with those structures, you can implement vectors using a variety of internal representations that vary in their efficiency, although arrays are the most common choice. Moreover, because vectors must expand dynamically, an implementation of the **Vector** class must be able to extend the capacity of its underlying array, just as the array-based implementations of stacks and queues do. Implementing the **Vector** class also raises a few new issues that did not arise in the stack and queue structures:

- The **Vector** class allows the client to insert and remove elements at any index position.
- The **Vector** class supports selection using square brackets, just as arrays do.
- The **Vector** class exports an iterator for cycling through the elements.

The implementation issues that arise in implementing these features are described in individual sections that follow the listing of the **vector.h** interface, which appears as Figure 11-11, beginning on the next page. That figure is followed immediately by Figure 11-12, which show the contents of the private section of the **Vector** class. The implementation of the parameterized **Vector** class appears in Figure 11-13, which begins on page 409.

Figure 11-11 Public interface for the Vector class

```

/*
 * File: vector.h
 * -----
 * This interface file contains the Vector class template, an
 * efficient, safer, convenient replacement for the built-in array.
 */

#ifndef _vector_h
#define _vector_h

#include "genlib.h"

/*
 * Class: Vector
 * -----
 * This interface defines a class template that stores a homogeneous
 * indexed collection. The basic operations are similar to those
 * in the built-in array type, with the added features of dynamic
 * memory management, bounds-checking on indexes, and convenient
 * insert/remove operations. Like an array, but better! For
 * maximum generality, the Vector is supplied as a class template.
 * The client specializes the vector to hold values of a specific
 * type, such as Vector<int> or Vector<studentT>.
 */

template <typename ElemType>
class Vector {
public:
    /*
     * Constructor: Vector
     * Usage: Vector<int> vec;
     * -----
     * The constructor initializes a new empty vector.
     */
    Vector();

    /*
     * Destructor: ~Vector
     * Usage: (usually implicit)
     * -----
     * Frees the storage associated with this vector.
     */
    ~Vector();

    /*
     * Method: size
     * Usage: nElems = vec.size();
     * -----
     * Returns the number of elements in this vector.
     */
    int size();

```

```
/*
 * Method: isEmpty
 * Usage: if (vec.isEmpty())...
 * -----
 * Returns true if this vector contains no elements, false otherwise.
 */
    bool isEmpty();

/*
 * Method: clear
 * Usage: vec.clear();
 * -----
 * Removes all elements from this vector.
 */
    void clear();

/*
 * Method: getAt
 * Usage: val = vec.getAt(3);
 * -----
 * Returns the element at the specified index in this vector.
 * Elements are indexed starting with 0. A call to vec.getAt(0)
 * returns the first element; vec.getAt(vec.size()-1) returns the
 * last. Raises an error if index is not in the range [0, size()-1].
 */
    ElemType getAt(int index);

/*
 * Method: setAt
 * Usage: vec.setAt(3, value);
 * -----
 * Replaces the element at the specified index in this vector with
 * a new value. The previous value at that index is overwritten.
 * Raises an error if index is not in the range [0, size()-1].
 */
    void setAt(int index, ElemType value);

/*
 * Method: insertAt
 * Usage: vec.insertAt(0, value);
 * -----
 * Inserts the element into this vector before the specified index,
 * shifting all subsequent elements one index higher. A call to
 * vec.insertAt(0, val) inserts a new element at the beginning;
 * vec.insertAt(vec.size(), val) adds a new element to the end.
 * Raises an error if index is outside the range [0, size()].
 */
    void insertAt(int index, ElemType elem);
```

```

/*
 * Method: removeAt
 * Usage: vec.removeAt(3);
 * -----
 * Removes the element at the specified index from this vector,
 * shifting all subsequent elements one index lower. A call to
 * vec.removeAt(0) removes the first element, while a call to
 * vec.removeAt(vec.size()-1), removes the last. Raises an error
 * if index is outside the range [0, size()-1].
 */

void removeAt(int index);

/*
 * Method: add
 * Usage: vec.add(value);
 * -----
 * Adds an element to the end of this vector.
 */

void add(ElemType elem);

/*
 * Method: operator[]
 * Usage: vec[0] = vec[1];
 * -----
 * Overloads [] to select elements from this vector. This extension
 * allows the client to use traditional array notation to get/set
 * individual elements. Returns a reference to the element to
 * allow in-place modification of values. Raises an error if the
 * index is outside the range [0, size()-1].
 */

ElemType & operator[](int index);

/*
 * Nested class: Vector<ElemType>::Iterator
 * -----
 * This code defines a nested class within the Vector template that
 * provides iterator access to the Vector contents. The Vector
 * and Iterator classes must declare each other as "friends" so
 * that they have access to the private variables.
 */

class Iterator {
public:
    Iterator();
    bool hasNext();
    ElemType next();

private:
    Vector *vp;
    int index;

    Iterator(Vector *vp);
    friend class Vector;
};
friend class Iterator;

```

```

/*
 * Method: iterator
 * Usage: iter = vec.iterator();
 * -----
 * Creates an iterator that allows the client to iterate through
 * the elements in this vector in index order.
 *
 * The idiomatic code for accessing elements using an iterator is
 * to create the iterator from the collection and then enter a loop
 * that calls next() while hasNext() is true, like this:
 *
 *     Vector<int>::Iterator iter = vec.iterator();
 *     while (iter.hasNext()) {
 *         int elem = iter.next();
 *         . . .
 *     }
 */
    Iterator iterator();

private:
#include "vecpriv.h"
};
#include "vecimpl.cpp"
#endif

```

Figure 11-12 Private section of the vector.h interface

```

/*
 * File: vecpriv.h
 * -----
 * This file contains the private section of the vector.h interface.
 */

/*
 * Implementation notes: Vector data structure
 * -----
 * The elements of the Vector are stored in a dynamic array of
 * the specified element type. If the space in the array is ever
 * exhausted, the implementation doubles the array capacity.
 */

/* Constants */
    static const int INITIAL_CAPACITY = 100;

/* Instance variables */
    ElemType *elements; /* A dynamic array of the elements */
    int capacity;        /* The allocated size of the array */
    int count;           /* The number of elements in use */

/* Private method prototypes */
    void expandCapacity();

```


Figure 11-13 Private implementation of the Vector class

```
/*
 * File: vecimpl.cpp
 * -----
 * This file contains the implementation of the vector.h interface.
 * Because of the way C++ compiles templates, this code must be
 * available to the compiler when it reads the header file.
 */

#ifdef _vector_h

/*
 * Implementation notes: Vector constructor and destructor
 * -----
 * The constructor allocates storage for the dynamic array
 * and initializes the other fields of the object. The
 * destructor frees the memory used for the array.
 */

template <typename ElemType>
Vector<ElemType>::Vector() {
    capacity = INITIAL_CAPACITY;
    count = 0;
    elements = new ElemType[capacity];
}

template <typename ElemType>
Vector<ElemType>::~~Vector() {
    delete[] elements;
}

/*
 * Implementation notes: Vector methods
 * -----
 * The basic Vector methods are straightforward and should require
 * no detailed documentation.
 */

template <typename ElemType>
inline int Vector<ElemType>::size() {
    return count;
}

template <typename ElemType>
bool Vector<ElemType>::isEmpty() {
    return count == 0;
}

template <typename ElemType>
void Vector<ElemType>::clear() {
    delete[] elements;
    capacity = INITIAL_CAPACITY;
    count = 0;
    elements = new ElemType[capacity];
}
```

```
template <typename ElemType>
ElemType Vector<ElemType>::getAt(int index) {
    if (index < 0 || index >= count) {
        Error("getAt: index out of range");
    }
    return elements[index];
}

template <typename ElemType>
void Vector<ElemType>::setAt(int index, ElemType elem) {
    if (index < 0 || index >= count) {
        Error("setAt: index out of range");
    }
    elements[index] = elem;
}

/*
 * Implementation notes: insertAt, removeAt, add
 * -----
 * These methods must shift the existing elements in the array to
 * make room for a new element or to close up the space left by a
 * deleted one.
 */

template <typename ElemType>
void Vector<ElemType>::insertAt(int index, ElemType elem) {
    if (count == capacity) expandCapacity();
    if (index < 0 || index > count) {
        Error("insertAt: index out of range");
    }
    for (int i = count; i > index; i--) {
        elements[i] = elements[i - 1];
    }
    elements[index] = elem;
    count++;
}

template <typename ElemType>
void Vector<ElemType>::removeAt(int index) {
    if (index < 0 || index >= count) {
        Error("removeAt: index out of range");
    }
    for (int i = index; i < count - 1; i++) {
        elements[i] = elements[i + 1];
    }
    count--;
}

template <typename ElemType>
void Vector<ElemType>::add(ElemType elem) {
    insertAt(count, elem);
}
```

```

/*
 * Implementation notes: Vector selection
 * -----
 * The following code implements traditional array selection using
 * square brackets for the index. The name of the method is
 * indicated by specifying the C++ keyword "operator" followed by
 * the operator symbol. To ensure that this operator returns an
 * assignable value, this method uses an & to return the result
 * by reference.
 */

template <typename ElemType>
ElemType & Vector<ElemType>::operator[](int index) {
    if (index < 0 || index >= count) {
        Error("Vector selection index out of range");
    }
    return elements[index];
}

/*
 * Vector::Iterator class implementation
 * -----
 * The Iterator for Vector maintains a pointer to the original
 * Vector and an index into that vector that identifies the next
 * element to return.
 */

template <typename ElemType>
Vector<ElemType>::Iterator::Iterator() {
    vp = NULL;
}

template <typename ElemType>
typename Vector<ElemType>::Iterator Vector<ElemType>::iterator() {
    return Iterator(this);
}

template <typename ElemType>
Vector<ElemType>::Iterator::Iterator(Vector *vp) {
    this->vp = vp;
    index = 0;
}

template <typename ElemType>
bool Vector<ElemType>::Iterator::hasNext() {
    if (vp == NULL) Error("hasNext called on uninitialized iterator");
    return index < vp->count;
}

template <typename ElemType>
ElemType Vector<ElemType>::Iterator::next() {
    if (vp == NULL) Error("next called on uninitialized iterator");
    if (!hasNext()) Error("next: No more elements");
    return vp->getAt(index++);
}

```

```

/*
 * Implementation notes: expandCapacity
 * -----
 * This private method doubles the capacity of the elements array
 * whenever it runs out of space. To do so, it must allocate a new
 * array, copy all the elements from the old array to the new one,
 * and free the old storage.
 */

template <typename ElemType>
void Vector<ElemType>::expandCapacity() {
    capacity *= 2;
    ElemType *oldElements = elements;
    elements = new ElemType[capacity];
    for (int i = 0; i < count; i++) {
        elements[i] = oldElements[i];
    }
    delete[] oldElements;
}

#endif

```

Supporting insertion and deletion at arbitrary index positions

The ability to insert and remove elements at arbitrary index positions is not particularly difficult to implement, particularly given that you have already seen the necessary techniques in the discussion of the editor buffer in Chapter 10. Just as in the case of the array-based editor buffer, inserting a new element into a **Vector** requires shifting all subsequent elements forward in the array. Similarly, removing an element requires shifting the remaining elements backward to close up the hole. The code to shift the elements appears in the **insertAt** and **removeAt** methods on page 410. Each of these operations can require $O(N)$ time in the worst case, which occurs when the insertion and deletion occurs near the beginning of the array.

Implementing selection brackets

One of the most powerful features of C++ is that it allows clients to redefine the operators for a particular class. To do so, all you need to do is define a method that implements the new behavior, where the name of that method is replaced by the keyword **operator** followed by the operator symbol. Most of the time, the operator symbol is exactly the one you use in writing programs, so that you could, for example, redefine the behavior of the **+** operator for some class by defining the **operator+** method. For selection, C++ uses the method name **operator[]** to suggest that the brackets are paired, even though the brackets are not in fact adjacent in the code.

The code to implement bracket selection is reasonably straightforward once you know how to write the method name. The code for **operator[]** looks exactly like the code for **get** except for the header line and the text of the error message:

```

template <typename ElemType>
ElemType & Vector<ElemType>::operator[](int index) {
    if (index < 0 || index >= count) {
        Error("Vector selection index out of range");
    }
    return elements[index];
}

```

The code for the **operator[]** method is, however, a bit more subtle than it first appears. The header line for this method introduces a new feature of C++ that you have not yet seen, although you will certainly have some insight into how it works from your understanding of call by reference. In this method, the return type is marked with an **&** to indicate **return by reference**, which means that the result of this function is in fact shared with the memory address found at **elements[index]**. Thus, if you assign a value to the result of the selection operator, that assignment updates the value in that array position, just as you would hope it would.

To complete the tasks of implementing the selection operator, you also need to add the prototype for **operator=** to the **map.h** interface, where it will appear as the single line

```
ValueType & operator[](string key);
```

Implementing iterators

The most important extension in the **Vector** class is the inclusion of an iterator facility that allows clients to step through the elements of the vector in order. The iterator class itself is defined within the **Vector** class and is therefore an example of a **nested class** definition. When you use a vector iterator, you need to qualify the class name with the complete designation of the class in which it appears, including the template parameters. The use of this form of qualification is illustrated in the conventional idiom for declaring an iterator, which typically looks something like this:

```
Vector<int>::Iterator iter = vec.iterator();
```

The full name of the iterator class is **Vector<int>::Iterator**, which indicates that this particular version of the iterator belongs to the **Vector<int>** class.

Except for the fact that it is nested inside another class, the definition of the **Iterator** class doesn't look particularly different from the classes you have used all along:

```
class Iterator {
public:
    Iterator();
    bool hasNext();
    ElemType next();

private:
    Vector *vp;
    int index;

    Iterator(Vector *vp);
    friend class Vector;
};
friend class Iterator;
```

The public section of the class declares a constructor and two methods, which are precisely the **hasNext** and **next** methods you would expect an iterator to export. The private section declares two instance variables, a pointer to a **Vector** and an integer. Without looking at the implementation, you won't necessarily know exactly what these variables are used for, but they seem reasonable enough from this perspective.

The more interesting parts of the class are the private constructor

```
Iterator(Vector *vp);
```

which takes a pointer to an existing vector and the two **friend** clauses, one of which appears inside the **Iterator** class itself, and one that appears outside of the **Iterator** class but inside the **Vector** class. When you design several classes that interact closely, it is often useful for each of those classes to manipulate private variables in the other without necessarily opening up access to those variables to other parts of the program. In C++, you can accomplish that goal by having one class declare another as a **friend**. Once you designate another class as your friend, that class can see your private variables. In this example, both **Iterator** and **Vector** declare each other as friends. Thus, code in the implementation of **Iterator** can refer to the private variables of a **Vector** object, and the code in **Vector** can invoke the private constructor inside the **Iterator** class.

It is important to note that friend access must be granted and cannot simply be asserted by one class over another. The **Iterator** class tells the compiler that the **Vector** class is its friend, and the **Vector** class does the same thing in the other direction. It is not possible for **Iterator** to claim friendship with **Vector** without **Vector** participating in the process. The friend relationship is therefore similar to what one sees in social networks like Facebook, where all friendship requests require approval.

The strategy for creating an iterator has two components. First, the iterator must store a pointer to the collection so that it has access to its elements. In this example, the instance variable **vp** in the **Iterator** points to the **Vector** object that created it. The creation occurs in the **iterator** method of the **Vector** class, which invokes the private constructor, passing in its own address by using the keyword **this**. Second, the iterator must keep track of enough private state to know where it is in the sequence of elements. For a **Vector** iterator, the only information needed is the index of the current element, which is stored in the variable **index**. Each call to **next** returns the current element, but increments **index** along the way so that the next call returns the next value. The **hasNext** method simply checks to see whether the value in **index** is still less than the vector size.

It is important to note that changes to the contents of a vector can invalidate the order of elements returned by an iterator. In general, iterators make sense only if the structure is static. You can provide some protection against such problems by trying to detect changes in the structure as you cycle through it, as described in exercise 8.

Summary

In this chapter, you have learned how to use the C++ template mechanism for generic container classes. A template allows you to define the class in terms of a type placeholder that can be specialized to a particular client data type. You have also had the chance to see a list-based implementation of the **Stack** class, a list- and an array-based implementation of **Queue**, and an array-based implementation of **Vector**.

Important points in this chapter include:

- Templates are used to define generic container classes.
- Stacks can be implemented using a linked-list structure in addition to the more traditional array-based representation.
- The array-based implementation of queues is somewhat more complex than its stack counterpart. The traditional implementation uses a structure called a *ring buffer*, in which the elements logically wrap around from the end of the array to the beginning. Modular arithmetic makes it easy to implement the ring buffer concept.
- In the ring-buffer implementation used in this chapter, a queue is considered empty when its head and tail indices are the same. This representation strategy means that the maximum capacity of the queue is one element less than the allocated size of the array.

Attempting to fill all the elements in the array makes a full queue indistinguishable from an empty one.

- Queues can also be represented using a singly linked list marked by two pointers, one to the head of the queue and another to the tail.
- Vectors can easily be represented using dynamic arrays. Inserting new elements and removing existing ones requires shifting data in the array, which means that these operations typically require $O(N)$ time.
- You can redefine operators for a class by defining methods whose name consists of the keyword **operator** followed by the operator symbol. In particular, you can redefine selection by defining the **operator[]** method.
- Implementing an iterator requires keeping track of a pointer to the collection along with enough information to cycle through the elements.

Review questions

1. When designing a generic container, what advantages does a C++ template offer?
2. When specializing a class template for use as a client, how do you specify what type should be used to fill in the template placeholder?
3. Draw a linked-list diagram of the stack **myStack** after the following operations are performed:

```
Stack<char> myStack;
myStack.push( 'A' );
myStack.push( 'B' );
myStack.push( 'C' );
```

4. What are the expanded forms of the acronyms LIFO and FIFO? Which of these disciplines pertains to the queue abstraction?
5. What are the names of the fundamental queue operations?
6. If you use an array to store the underlying elements in a queue, what are the **Queue** class private instance variables?
7. What is a ring buffer? How does the ring-buffer concept apply to queues?
8. How can you tell if an array-based queue is empty? How can you tell if it has reached its capacity?
9. Assuming that **INITIAL_CAPACITY** has the artificially small value 3, draw a diagram showing the underlying representation of the array-based queue **myQueue** after the following sequence of operations:

```
Queue<char> myQueue;
myQueue.enqueue( 'A' );
myQueue.enqueue( 'B' );
myQueue.enqueue( 'C' );
myQueue.dequeue();
myQueue.dequeue();
myQueue.enqueue( 'D' );
myQueue.enqueue( 'E' );
myQueue.dequeue();
myQueue.enqueue( 'F' );
```

10. Explain how modular arithmetic is useful in the array-based implementation of queues.
11. Describe what is wrong with the following implementation of **size** for the array-based representation of queues:

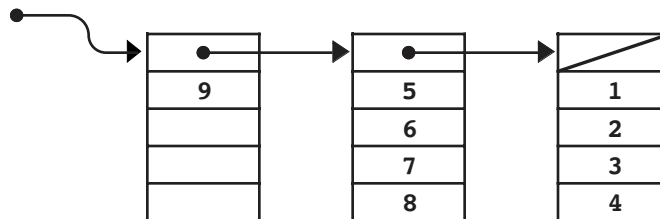
```
template <typename ElemType>
int Queue<ElemType>::size() {
    return (tail - head) % QueueArraySize;
}
```



12. Draw a diagram showing the internal structure of a linked-list queue after the computer finishes the set of operations in question 9.
13. How can you tell if a linked-list queue is empty?
14. What is the purpose of the **operator** keyword?
15. What is the name of the method you need to override if you want to define bracket selection for a class?
16. True or false: A class named **MyClass** can gain access to the private variables of a class named **YourClass** by including a line designating **YourClass** as a friend.
17. What is meant by the term *nested class*?
18. What are the two pieces of information that any **Iterator** class must contain?

Programming exercises

1. One of the principal reasons that stacks are usually implemented using arrays is that linked lists impose a significant memory overhead to store the pointers. You can, however, reduce this cost by adopting a hybrid approach similar to the one described in Chapter 10, exercise 13. The idea is to represent the stack as a linked list of blocks, each of which contains a fixed array of elements. Whenever a stack block is exhausted, a new block can be added to the front of a chain in the data structure to open up the necessary additional space. For example, if there were four elements per block, a stack into which the integers 1 through 9 had been pushed in numerical order would look like this:



Write a new implementation of the **Stack** class that uses this design. Note that the private instance variables are not shown in the diagram and are left for you to design.

2. Because the ring-buffer implementation of queues makes it impossible to tell the difference between an empty queue and one that is completely full, the capacity of the queue is one less than the allocated size of the array. You can avoid this restriction by changing the internal representation so that the concrete structure of the queue keeps track of the number of elements in the queue instead of the index of

the tail element. Given the index of the head element and the number of data values in the queue, you can easily calculate the tail index, which means that you don't need to store this value explicitly. Rewrite the array-based queue representation so that it uses this representation.

3. In exercise 9 from Chapter 4, you had the opportunity to write a function

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

that reverses the elements in the queue, working entirely from the client side. If you are the designer of a class, however, you could add this facility to the **queue.h** interface and export it as one of its methods. For both the array- and list-based implementations of the queue, make all the changes necessary to export the method

```
void reverse();
```

that reverses the elements in the queue. In both cases, write the functions so that they use the original memory cells and do not allocate any additional storage.

4. In the queue abstraction presented in this chapter, new items are always added at the end of the queue and wait their turn in line. For some programming applications, it is useful to extend the simple queue abstraction into a **priority queue**, in which the order of the items is determined by a numeric priority value. When an item is enqueued in a priority queue, it is inserted in the list ahead of any lower priority items. If two items in a queue have the same priority, they are processed in the standard first-in/first-out order.

Extend the linked-list implementation of queues so that it supports priority queues. To do so, you need to add a new version of the **enqueue** function to the interface that is overloaded to take an additional argument, as follows:

```
void enqueue(ElemType element, double priority);
```

The parameter **element** is the same as for the traditional versions of **enqueue**; the **priority** argument is a numeric value representing the priority. As in conventional English usage, smaller integers correspond to higher priorities, so that priority 1 comes before priority 2, and so forth.

Keep in mind that you are implementing an extension to an existing interface. Clients who do not use the new version of **enqueue** should not need to make any changes in their code.

5. Reimplement the **Vector** class presented in this chapter so that it uses a linked list as its underlying representation. What operations are slower using this model? What operations are faster? How might you design the data structure so that the **add** method always executes in constant time?
6. Reimplement the **Vector** class so that its underlying representation uses two stacks in the style of the stack-based editor buffer introduced in Chapter 10. The gap between the two stacks should always be the last point at which an insertion or deletion was made, which means that executing a series of insertions or deletions at the same index position will run in constant time.
7. Use the techniques from the **Vector** implementation in section 11.4 to implement the **Grid** class, with the exception of bracket selection, which is much trickier to code for a two-dimensional structure. The interface for the **Grid** class appears in Appendix A.

8. One way to guard against the problem of iterating through a collection that is simultaneously being modified is to check against that possibility as the iterator proceeds. The usual strategy is to declare an integer variable called a **timestamp**, which is initially set to 0 when the collection is created and then incremented by one each time the contents of the collection change. If you record this timestamp in the iterator structure when you create it, you can then check on each cycle to see whether it still matches the value in the collection object. If not, you can report an error so that the client programmer can find and fix the problem.

Chapter 12

Implementing Maps

*Yea, from the table of my memory
I'll wipe away all trivial fond records*

— Shakespeare, *Hamlet*, 1602

One of the most useful data structures introduced in Chapter 4 was the **Map** class, which provides an association between keys and values. The primary goal of this chapter is to show you how maps can be implemented extremely efficiently using a particularly clever representation called a *hash table*. Before doing so, however, it makes sense to start with a less efficient implementation that is not nearly so clever just to make sure that you understand what is involved in implementing the **map.h** interface. The following section defines an array-based implementation for the **Map** class. The rest of the chapter then looks at various strategies for improving on that simple design.

12.1 An array-based implementation of the map interface

Figure 12-1 shows a slightly simplified implementation of the **map.h** interface, which leaves out three features of the library version of the interface: deep copying, selection using square brackets, and the ability to iterate over the keys in a map. Even in its current form, however, the interface is quite useful, and it makes sense to investigate possible implementations of the fundamental operations before extending the interface.

When you are trying to learn how a particular data structure operates, it is often helpful to start with a specific example, understand how that example works, and then generalize from that example to get a sense of how the abstraction works as a whole. Suppose, for example, that you have been asked to implement (as you will be in exercise 1 at the end of this chapter) a program that translates Roman numerals into integers. As part of that program, you will need some way of encoding the following translation table:

I	→	1
V	→	5
X	→	10
L	→	50
C	→	100
D	→	500
M	→	1000

Figure 12-1 Preliminary version of the **map.h** interface

```

/*
 * File: map.h
 * -----
 * This interface exports a slightly simplified version of the Map
 * template class.
 */

#ifndef _map_h
#define _map_h

#include "genlib.h"

/*
 * Class: Map
 * -----
 * This interface defines a class template that stores a collection
 * of key-value pairs. The keys are always strings, but the values
 * can be of any type. This interface defines the value type using
 * the template facility in C++, which makes it possible to specify
 * the value type in angle brackets, as in Map<int> or Map<string>.
 */

```

```
template <typename ValueType>
class Map {

public:

    /*
    * Constructor: Map
    * Usage: Map<int> map;
    * -----
    * The constructor initializes a new empty map.
    */

    Map();

    /*
    * Destructor: ~Map
    * Usage: delete mp;
    * -----
    * The destructor frees any heap storage associated with this map.
    */

    ~Map();

    /*
    * Method: size
    * Usage: nEntries = map.size();
    * -----
    * This method returns the number of entries in this map.
    */

    int size();

    /*
    * Method: isEmpty
    * Usage: if (map.isEmpty())...
    * -----
    * This method returns true if this map contains no entries,
    * false otherwise.
    */

    bool isEmpty();

    /*
    * Method: clear
    * Usage: map.clear();
    * -----
    * This method removes all entries from this map.
    */

    void clear();
```

```
/*
 * Method: put
 * Usage: map.put(key, value);
 * -----
 * This method associates key with value in this map. Any value
 * previously associated with this key is replaced by the new one.
 */

    void put(string key, ValueType value);

/*
 * Method: get
 * Usage: value = map.get(key);
 * -----
 * If key is found in this map, this method returns the associated
 * value. If key is not found, the get method raises an error.
 * Clients can use the containsKey method to verify the presence
 * of a key in the map before attempting to get its value.
 */

    ValueType get(string key);

/*
 * Method: containsKey
 * Usage: if (map.containsKey(key))...
 * -----
 * Returns true if there is an entry for key in this map,
 * false otherwise.
 */

    bool containsKey(string key);

/*
 * Method: remove
 * Usage: map.remove(key);
 * -----
 * This method removes any entry for key from this map.
 * If there is no entry for the key, the map is unchanged.
 */

    void remove(string key);

private:
#include "mappriv.h"
};
#include "mapimpl.cpp"
#endif
```

Given your experience with the classes in Chapter 4, the idea of using a map should spring immediately to mind whenever you see a translation table that maps strings to some other value. To set up such a map, you would need the following code:

```
Map<int> romanNumerals;
romanNumerals.put("I", 1);
romanNumerals.put("V", 5);
romanNumerals.put("X", 10);
romanNumerals.put("L", 50);
romanNumerals.put("C", 100);
romanNumerals.put("D", 500);
romanNumerals.put("M", 1000);
```

The simplest strategy for representing this data structure is to store each key/value pair in an array. As with most of the implementations you've seen since Chapter 9, that array needs to be dynamic so that it can expand if the number of keys grows beyond the initial allocation. The `mappriv.h` file for the array-based implementation of the `Map` class appears in Figure 12-2, which contains the structure definitions and instance variables necessary to represent this information. The code for the array-based representation of the `Map` class appears in Figure 12-3.

Figure 12-2 Contents of the private section of `map.h` for the array-based representation

```
/*
 * File: mappriv.h
 * -----
 * This file contains the private section of the map.h interface
 * for the array-based map.
 */

/*
 * Type: keyValuePairT
 * -----
 * This type represents a key-value pair. This implementation of
 * the Map class stores these entries in an array.
 */

struct keyValuePairT {
    string key;
    ValueType value;
};

/* Constants */

static const int INITIAL_CAPACITY = 100;

/* Instance variables */

keyValuePairT *array; /* A dynamic array of key/value pairs */
int capacity; /* The allocated size of the array */
int count; /* The current number of entries */

/* Private function prototypes */

int findKey(string key);
void expandCapacity();
```

Figure 12-3 The array-based implementation of the Map class

```

/*
 * File: mapimpl.cpp
 * -----
 * This file implements the map interface using the array-based
 * representation. Most of these implementations are too short
 * to require additional comments.
 */

#ifdef _map_h

template <typename ValueType>
Map<ValueType>::Map() {
    capacity = INITIAL_CAPACITY;
    array = new keyValuePairT[capacity];
    count = 0;
}

template <typename ValueType>
Map<ValueType>::~~Map() {
    delete[] array;
}

template <typename ValueType>
int Map<ValueType>::size() {
    return count;
}

template <typename ValueType>
bool Map<ValueType>::isEmpty() {
    return (count == 0);
}

template <typename ValueType>
void Map<ValueType>::clear() {
    count = 0;
}

/*
 * Implementation notes: put
 * -----
 * The put method begins by calling findKey to searches for an
 * existing key. If that key is found, put stores the value in the
 * corresponding key/value pair. If not, put adds a new key/value
 * pair to the array, expanding the capacity if necessary.
 */

template <typename ValueType>
void Map<ValueType>::put(string key, ValueType value) {
    int index = findKey(key);
    if (index == -1) {
        if (count == capacity) expandCapacity();
        index = count++;
        array[index].key = key;
    }
    array[index].value = value;
}

```



```
/*
 * Implementation notes: get
 * -----
 * The get method calls findKey to search for the specified key.
 * If the key is found, get returns the value from that key/value
 * pair. If not, get reports an error.
 */

template <typename ValueType>
ValueType Map<ValueType>::get(string key) {
    int index = findKey(key);
    if (index == -1) {
        Error("Attempt to get value for key that is not in the map.");
    }
    return array[index].value;
}

/*
 * Implementation notes: containsKey
 * -----
 * This method simply checks the result of the private findKey
 * method, which does all the work.
 */

template <typename ValueType>
bool Map<ValueType>::containsKey(string key) {
    return (findKey(key) != -1);
}

/*
 * Implementation notes: remove
 * -----
 * The code for remove saves a little time by copying the
 * key/value pair from the last entry into this cell. Note
 * that there is no reason to check whether the deleted item
 * is the last element. If it is, the copy is harmless, and
 * that key/value pair will no longer be part of the active
 * region of the array. Note also that count can't be zero
 * if findKey has found a match.
 */

template <typename ValueType>
void Map<ValueType>::remove(string key) {
    int index = findKey(key);
    if (index != -1) {
        array[index] = array[--count];
    }
}
```

```

/* Private methods */

/*
 * Private method: findKey
 * Usage: int index = findKey(key);
 * -----
 * This method searches through all the keys in the map searching
 * for a cell that contains the specified key. If it finds one,
 * it returns the index of that element in the array. If no
 * such key exists, findKey returns -1.
 */

template <typename ValueType>
int Map<ValueType>::findKey(string key) {
    for (int i = 0; i < count; i++) {
        if (array[i].key == key) return i;
    }
    return -1;
}

/*
 * Implementation notes: expandCapacity
 * -----
 * This private method doubles the capacity of the array whenever
 * it runs out of space. To do so, it must allocate a new array,
 * copy all the elements from the old array to the new one, and
 * free the old storage.
 */

template <typename ValueType>
void Map<ValueType>::expandCapacity() {
    int count = size();
    capacity *= 2;
    keyValuePairT *oldArray = array;
    array = new keyValuePairT[capacity];
    for (int i = 0; i < count; i++) {
        array[i] = oldArray[i];
    }
    delete[] oldArray;
}

#endif

```

For the most part, the code in Figure 12-3 is similar to the implementations you have seen for the other data structures; the only difference is that the **Map** class needs to search the array to find a particular key. Because that operation occurs in each of the **get**, **put**, **remove**, and **containsKey** operations, it makes sense to code it as a private method that makes it possible to share the code among these methods. In the simple array-based implementation, **findKey** scans the elements of the array until it finds a matching key, in which case it returns the index of that key/value pair in the array. If no such key exists, **findKey** returns **-1**, which the caller can then respond to in the appropriate way.

Unfortunately, the simple array-based implementation of maps is not very efficient. In the worst case—which occurs when a key does not appear in the map—both **put** and **get** require $O(N)$ time because the code must search through every entry. If you kept the array in sorted order, you could use binary search to reduce the complexity of **get** to $O(\log N)$, but **put** would still require $O(N)$ time.

12.2 The advantage of knowing where to look

The array-based implementation of maps from the preceding section operates in $O(N)$ time with respect to the number of keys. While that time constant might be tolerable in an application like the roman-numeral translation in which N has the value 7, it will quickly become a problem as maps grow larger. In practice, maps are used heavily in applications and can consume a considerable fraction of the total execution time. As a result, efficiency is a particularly important concern. It would be far better to choose an internal representation that offered, at least on average, a constant-time implementation for both **put** and **get**.

To get a sense of how you might accomplish that kind of efficiency improvement, think again about the problem of translating Roman numerals to their decimal equivalent. Even though the translation table

I	→	1
V	→	5
X	→	10
L	→	50
C	→	100
D	→	500
M	→	1000

might suggest using a map, there are other ways to accomplish the same goal. In this table, the keys are letters of the alphabet and therefore have an internal ASCII code. What would happen if you entered these values in an integer array with 256 elements in which each element contains one of the following values:

- The integer 0 if the character corresponding to the index value is legal in a Roman numeral. For example, the element at index position 65 (the ASCII code for 'A') would be 0 because **A** cannot appear in a Roman numeral.
- The value of that character as a Roman numeral constituent. For example, the element at index position 88 (the ASCII code for 'x') would be 10 because **x** has the value 10 in Roman numeral.

You could initialize such an array using the following code:

```
int romanLookupTable[256];
for (int i = 0; i < 256; i++) {
    romanLookupTable[i] = 0;
}
romanLookupTable['I'] = 1;
romanLookupTable['V'] = 5;
romanLookupTable['X'] = 10;
romanLookupTable['L'] = 50;
romanLookupTable['C'] = 100;
romanLookupTable['D'] = 500;
romanLookupTable['M'] = 1000;
```

After executing this code, the array **romanLookupTable** would look like this:

...										...									
	0	0	100	500	0	0	0	0	1	0	0	50	1000		5	0	10		
	65	66	67	68	69	70	71	72	73	74	75	76	77		86	87	88		
	'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	'I'	'J'	'K'	'L'	'M'		'V'	'W'	'X'		

With this data structure, you can look up the value of any of the Roman numeral constituents with a simple $O(1)$ array reference. Moreover, you can also add new key/value pairs—subject to the restriction that the keys are single characters—in constant time. For example, if you want your Roman numeral translator to operate equally well with both upper- and lowercase letters, all you need to do is set the values in the array elements corresponding to the letters **i**, **v**, **x**, **l**, **c**, **d**, and **m**, like this:

```
romanLookupTable['i'] = 1;
romanLookupTable['v'] = 5;
romanLookupTable['x'] = 10;
romanLookupTable['l'] = 50;
romanLookupTable['c'] = 100;
romanLookupTable['d'] = 500;
romanLookupTable['m'] = 1000;
```

The reason that this redesign makes it possible to operate in constant time is that there is no longer any searching involved because you know exactly where to look. In this case, you decide where to look based on the ASCII code of the character, but the fundamental idea is that the key somehow directs you immediately to the location in which the value is stored.

If you think about this idea for a bit, you'll realize that you do the same thing when you are looking up words in a dictionary. If you were to apply the array-based map strategy to the dictionary-lookup problem, you would start at the first entry, go on to the second, and then the third, until you found the word. No one in their right mind would apply this algorithm in a real dictionary of any significant size. But it is also unlikely that you would apply the $O(\log N)$ binary search algorithm, which consists of opening the dictionary exactly at the middle, deciding whether the word you're searching for appears in the first or second half, and then repeatedly applying this algorithm to smaller and smaller parts of the dictionary. In all likelihood, you would take advantage of the fact that most dictionaries have thumb tabs along the side that indicate where the entries for each letter appear. You look for words starting with *A* in the *A* section, words starting with *B* in the *B* section, and so on.

You can use the same strategy to implement the map abstraction. In the map, each key begins with some character value, although that character is not necessarily a letter. If you wanted to simulate the strategy of using thumb tabs for every possible first character, you could divide the map into 256 independent lists of key/value pairs—one for each starting character. When **put** or **get** is presented with a key, the code can choose the appropriate list on the basis of the first character in the key. If the characters used to form keys were uniformly distributed, this strategy would reduce the average search time by a factor of 256.

Unfortunately, keys in a map—like words in a dictionary—are not uniformly distributed. In the dictionary case, for example, many more words begin with *C* than with *X*. When you use a map in an application, it is likely that most of the 256 characters will never appear at all. As a result, some of the lists will remain empty, while others become quite long. The increase in efficiency you get by applying the first-character strategy therefore depends on how common the first character in the key happens to be.

On the other hand, there is no reason that you have to use the first character of the key as a hint for finding the desired key/value pair; that's just the closest analogue for what you do with a dictionary. All you really need is some strategy that uses the key to find the location of the value. That idea is most elegantly implemented using a technique called *hashing*, which is described in the following section.

12.3 Hashing

As suggested by the discussion in the preceding section, the key to improving the efficiency of the map implementation is to come up with some way of using the key to determine, at least fairly closely, where to look for the corresponding value. Choosing almost any obvious property of the key, such as its first character or even its first two characters, runs into the problem that keys are not equally distributed with respect to that property. Given that you are using a computer, however, there is no reason to require that the property you use to locate the key has to be something easy for a *human* to figure out. To maintain the efficiency of the implementation, the only thing that matters is whether that property is easy for a *computer* to figure out. Since computers are much better at computation than humans are, the idea that a property can be calculated algorithmically opens a much wider range of possibilities.

The computational strategy called **hashing** operates as follows:

1. Select a function f that transforms a key into an integer value. That value is called the **hash code** of that key, and the function that computes it is called, naturally enough, a **hash function**. A map that uses this strategy is conventionally called a **hash table**.
2. Use the hash code for a key as the starting point as you search for a matching key in the table. You might, for example, use the hash code value as an index into an array of lists, each of which holds all the key/value pairs that correspond to that hash code. To find an exact match, all you need to do is search through the list of key/value pairs in that list. As long as the hash function always returns the same value for any particular key, you know that the value, if it exists, must be in that list. Suppose, for example, that you call **put** on a key whose hash code is 17. According to the basic hashing algorithm, the **put** method must store that key and its value on list #17. If you later call **get** with that same key, the hash code you get will be the same, which means that the desired key/value pair must be on list #17, if it exists at all.

Implementing the hash table strategy

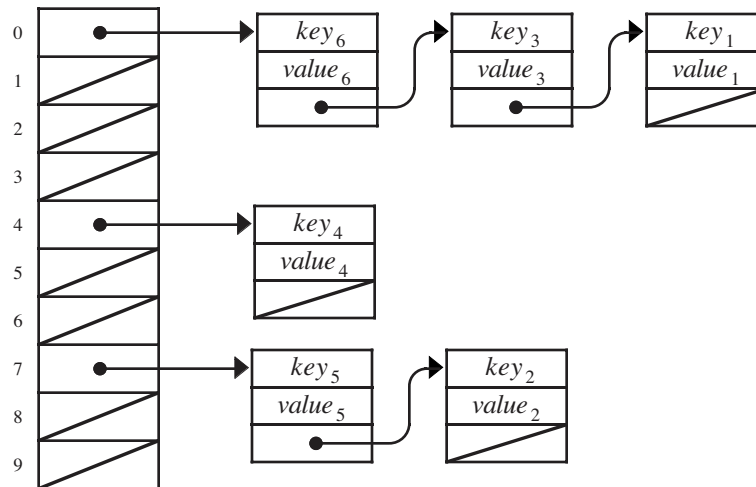
The first step toward implementing a hash table is to design the data structure. As the preceding section suggests, each hash code serves as an index into an array of linked lists. Each list is traditionally called a **bucket**. Whenever you call **put** or **get**, you select the appropriate bucket by applying the hash function to the key. That function gives you an integer, which is likely to be larger than the number of buckets you have in your hash table. You can, however, convert an arbitrarily large nonnegative hash code into a bucket number by dividing the hash code by the number of buckets and taking the remainder. Thus, if the number of buckets is stored in the variable **nBuckets** and the method **hash** computes the hash code for a given key, you can use the following line to compute the bucket number:

```
int bucket = hash(key) % nBuckets;
```

A bucket number represents an index into an array, each of whose elements is a pointer to the first cell in a list of key/value pairs. Colloquially, computer scientists say that a key **hashes to a bucket** if the hash function applied to the key returns that bucket number. Thus, the common property that links all the keys in a single linked list is that they all hash to the same bucket. Having two or more different keys hash to the same bucket is called **collision**.

To help you visualize the representation of a hash table, the following diagram shows a table with six key/value pairs distributed across a map with 10 buckets. In the diagram,

three entries (key_1 , key_3 , and key_6) hash to bucket #0, one (key_4) hashes to bucket #4, and two (key_2 and key_5) hash to bucket #7:



For the hash table implementation of the `map.h` interface, the private section of the `Map` class must contain the instance variables and type definitions necessary to represent this data structure. The hash table version of `mappriv.h` appears in Figure 12-4.

Figure 12-4 Contents of the private section of `map.h` for the hash table representation

```

/*
 * File: mappriv.h
 * -----
 * This file contains the private section of the Map template
 * class. Including this information in a separate file means
 * that clients don't need to look at these details.
 */

/* Constants */

static const int INITIAL_SIZE = 101;

/* Type for a linked list cell */

struct cellT {
    string key;
    ValueType value;
    cellT *link;
};

/* Instance variables */

cellT **buckets; /* A dynamic array of the buckets */
int nBuckets;    /* Allocated size of the buckets array */
int nEntries;    /* The number of entries in the map */

/* Private method prototypes */

int hash(string s);
cellT *findCell(cellT *chain, string key);
void deleteChain(cellT *chain);
  
```

The definition of `cellT` in Figure 12-4 looks very much like the type definitions used for linked list cells in earlier chapters, except that each cell includes both a key and a value along with the link. The private data of a `Map` object is an array of buckets, each of which is a pointer to the first cell in its chain.

The code for the hash table implementation of `map.h` appears in Figure 12-5. Assuming that you understand linked lists at the level of the earlier examples, the code in Figure 12-5 should be reasonably straightforward. Most of the list manipulation happens in the private `findCell` method, which searches for the key in the list associated with a particular bucket chain. This method is used in `get`, `put`, and `containsKey` to avoid duplicating the common code.

It is interesting to note that even though the code for `remove` involves pretty much the same iteration through a list, that method does use its own version of the code. As the implementation notes make clear, the reason for this design decision is that `remove` needs to keep track of the cell *before* the deleted one so that it can keep the pointer chain intact.

Figure 12-5 The hash table implementation of the `Map` class

```

/*
 * File: mapimpl.cpp
 * -----
 * This file implements the map.h interface. Because of the
 * way C++ templates are defined, it must be included as part
 * of the map.h header file.
 */

#ifdef _map_h

/*
 * Implementation notes: Map constructor
 * -----
 * The constructor allocates the array of buckets and initializes
 * each bucket to the empty list.
 */

template <typename ValueType>
Map<ValueType>::Map() {
    nBuckets = INITIAL_SIZE;
    buckets = new cellT *[nBuckets];
    for (int i = 0; i < nBuckets; i++) {
        buckets[i] = NULL;
    }
}

/*
 * Implementation notes: ~Map destructor
 * -----
 * The destructor must deallocate every cell (which it can do by
 * calling clear) and then free the dynamic bucket array.
 */

template <typename ValueType>
Map<ValueType>::~~Map() {
    clear();
    delete[] buckets;
}

```

```
/*
 * Implementation notes: size, isEmpty
 * -----
 * These methods can each be implemented in a single line
 * because the size is stored in the nEntries instance variable.
 */

template <typename ValueType>
int Map<ValueType>::size() {
    return nEntries;
}

template <typename ValueType>
bool Map<ValueType>::isEmpty() {
    return nEntries;
}

/*
 * Implementation notes: clear
 * -----
 * This method calls the recursive deleteChain method for each
 * bucket chain.
 */

template <typename ValueType>
void Map<ValueType>::clear() {
    for (int i = 0; i < nBuckets; i++) {
        deleteChain(buckets[i]);
    }
    nEntries = 0;
}

/*
 * Implementation notes: put
 * -----
 * This method first looks to see whether the key already
 * exists in the map by calling the findCell method. If one
 * exists, this method simply changes the value; if not, the
 * implementation adds a new cell to the beginning of the chain.
 */

template <typename ValueType>
void Map<ValueType>::put(string key, ValueType value) {
    int index = hash(key) % nBuckets;
    cellT *cell = findCell(buckets[index], key);
    if (cell == NULL) {
        cell = new cellT;
        cell->key = key;
        cell->link = buckets[index];
        buckets[index] = cell;
        nEntries++;
    }
    cell->value = value;
}
```



```

/*
 * Implementation notes: get, containsKey
 * -----
 * These methods uses findCell to find the key in the map, which is
 * where all the real work happens.
 */

template <typename ValueType>
ValueType Map<ValueType>::get(string key) {
    cellT *cell = findCell(buckets[hash(key) % nBuckets], key);
    if (cell == NULL) {
        Error("Attempt to get value for key that is not in the map.");
    }
    return cell->value;
}

template <typename ValueType>
bool Map<ValueType>::containsKey(string key) {
    return findCell(buckets[hash(key) % nBuckets], key) != NULL;
}

/*
 * Implementation notes: remove
 * -----
 * The remove method cannot use the findCell method as it
 * stands because it needs a pointer to the previous entry.
 * Because that code is used only in this method, the loop
 * through the cells in a chain is reimplemented here and
 * therefore does not add any cost to the get/put operations.
 */

template <typename ValueType>
void Map<ValueType>::remove(string key) {
    int index = hash(key) % nBuckets;
    cellT *prev = NULL;
    cellT *cp = buckets[index];
    while (cp != NULL && cp->key != key) {
        prev = cp;
        cp = cp->link;
    }
    if (cp != NULL) {
        if (prev == NULL) {
            buckets[index] = cp->link;
        } else {
            prev->link = cp->link;
        }
        delete cp;
        nEntries--;
    }
}

```

```

/* Private methods */

/*
 * Implementation notes: hash
 * Usage: bucket = hash(key);
 * -----
 * This function takes the key and uses it to derive a hash code,
 * which is a nonnegative integer. The hash code is computed
 * using a method called linear congruence.
 */

template <typename ValueType>
int Map<ValueType>::hash(string s) {
    const long MULTIPLIER = -1664117991L;
    unsigned long hashcode = 0;
    for (int i = 0; i < s.length(); i++) {
        hashcode = hashcode * MULTIPLIER + s[i];
    }
    return hashcode & ((unsigned) -1 >> 1);
}

/*
 * Implementation notes: findCell
 * Usage: cell = findCell(chain, key);
 * -----
 * This function finds a cell in the chain that matches key.
 * If a match is found, findCell returns a pointer to that cell;
 * if not, findCell returns NULL.
 */

template <typename ValueType>
typename Map<ValueType>::cellT *Map<ValueType>
    ::findCell(cellT *chain, string key) {
    for (cellT *cp = chain; cp != NULL; cp = cp->link) {
        if (cp->key == key) return cp;
    }
    return NULL;
}

/*
 * Private method: deleteChain
 * -----
 * This method deletes all of the cells in a bucket chain.
 * It operates recursively by freeing the rest of the chain
 * and the freeing the current cell.
 */

template <typename ValueType>
void Map<ValueType>::deleteChain(cellT *chain) {
    if (chain != NULL) {
        deleteChain(chain->link);
        delete chain;
    }
}

```

Although it would be possible to change the definition of `findCell` so that it kept track of this information, doing so would have the effect of slowing down `put` and `get`, which tend to be much more common operations.

Choosing a hash function

The one method in Figure 12-5 that clearly demands further explanation is `hash`, which is, after all, the method that gives this technique its name. As it appears in Figure 12-5, the `hash` function has the following implementation, which is certainly rather cryptic:

```
template <typename ValueType>
int Map<ValueType>::hash(string s) {
    const long MULTIPLIER = -1664117991L;
    unsigned long hashcode = 0;
    for (int i = 0; i < s.length(); i++) {
        hashcode = hashcode * MULTIPLIER + s[i];
    }
    return int(hashcode & (unsigned(-1) >> 1));
}
```

Although there are many different strategies for writing hash functions, the code for `hash` shown here is typical of the functions most often used in commercial practice. The code iterates through each character in the key, updating an integer value stored in the local variable `hashcode`, which is for some reason declared as an `unsigned long` rather than an integer. On each loop cycle, the `hash` function multiplies the previous value of `hashcode` by a mysterious constant called `MULTIPLIER` and then adds the ASCII value of the current character. At the end of the loop, the result is not simply the value of `hashcode` but instead computed by means of the rather odd-looking expression

```
int(hashcode & (unsigned(-1) >> 1))
```

Given the amount of confusing code present in such a short function, you should feel perfectly justified in deciding that the details of the `hash` function are not important enough to understand in detail. The point of all the complexity is to ensure that the result of the `hash` function are as unpredictable as possible given a particular set of keys. The details as to how it does so, while interesting in its own right as a theoretical question, are not of immediate concern to clients of the `Map` class. What is important is knowing that `hash` guarantees that its result will be a nonnegative integer.

To see why the design of the `hash` function might have an effect on efficiency, consider what might happen if you used the following, much simpler implementation:

```
template <typename ValueType>
int Map<ValueType>::hash(string s) {
    int hashcode = 0;
    for (int i = 0; i < s.length(); i++) {
        hashcode += s[i];
    }
    return hashcode;
}
```



This implementation is far more understandable. All it does is add up the ASCII codes for all the characters in the string, which will be a nonnegative integer unless the string is hugely long. Even in spite of the fact that long strings might cause integer overflow and result in negative results (which justifies the inclusion of the bug symbol), coding `hash` in this way is much more likely to cause collisions in the table if the keys happen to fall into certain patterns. The strategy of adding the ASCII values means that any keys whose

letters are permutations of each other would collide. Thus, **cat** and **act** would hash to the same bucket. So would the keys **a3**, **b2**, and **c1**. If you were using this hash table in the context of a compiler, variable names that fit such patterns would all end up hashing to the same bucket.

At the cost of making the code for the **hash** function more obscure, you can reduce the likelihood that similar keys will collide. Figuring out how to design such a function, however, requires some experience and a more advanced knowledge of computer science theory. The strategy used in Figure 12-5 is closely related to the technique used in a typical random number generator like the ANSI function **rand**. In both the hashing algorithms and the random number generator, the arithmetical properties of the calculation make the results harder to predict. In the hash table, the consequence of this unpredictability is that keys chosen by a programmer are unlikely to exhibit any higher level of collision than one would expect by random chance.

Even though careful choice of a hash function can reduce the number of collisions and thereby improve performance, it is important to recognize that the *correctness* of the algorithm is not affected by the collision rate. The only requirement is that the hash function deliver a nonnegative integer. If it does, the map implementation will still work even if the hash function always returned 0. In that case, every key would end up in the chain attached to bucket #0. Programs that used such a hash function would run slowly because every key would be linked into the same chain, but they would nonetheless continue to give the correct results.

Determining the number of buckets

Although the design of the hash function is important, it is clear that the likelihood of collision also depends on the number of buckets. If the number is small, collisions occur more frequently. In particular, if there are more entries in the hash table than buckets, collisions are inevitable. Collisions affect the performance of the hash table strategy because they force **put** and **get** to search through longer chains. As the hash table fills up, the number of collisions rises, which in turn reduces the performance of the hash table.

Remember that the goal of using a hash table is to implement a map so that the **put** and **get** methods run in constant time, at least in the average case. To achieve this goal, it is important that the linked-list chains emerging from each bucket remain fairly short. Thus, you want to make sure that the number of buckets is large enough to keep the chains relatively modest in length. If the hash function does a good job of distributing the keys evenly among the buckets, the average length of each bucket chain is given by the formula

$$\lambda = \frac{N_{\text{entries}}}{N_{\text{buckets}}}$$

For example, if the total number of entries in the table is three times the number of buckets, the average chain will contain three entries, which in turn means that three string comparisons will be required, on average, to find a key. The value λ is called the **load factor** of the hash table.

For good performance, you want to make sure that the value of λ remains relatively small. On the other hand, choosing a large value for **nBuckets** means that there are lots of empty buckets in the hash table array, which wastes a certain amount of space. Hash tables represent a good example of a time-space tradeoff, a concept introduced in Chapter 10. By increasing the amount of space, you can improve performance.

Of course, it may be difficult to choose a value of **nBuckets** that works well for all clients. If a client keeps entering more and more entries into a map, the performance will eventually decline. If you want to restore good performance in such a case, one approach is to allow the implementation to increase the number of buckets dynamically. For example, you can design the implementation so that it allocates a larger hash table if the load factor in the table ever reaches a certain threshold. Unfortunately, if you increase the number of buckets, the bucket numbers all change, which means that the code to expand the table must reenter every key from the old table into the new one. This process is called **rehashing**. Although rehashing can be time-consuming, it is performed infrequently and therefore has minimal impact on the overall running time of the application. Rehashing is unnecessary for most application domains and is not included in the implementations of hashing used in this text.

Using the **typename** keyword

While you have previously seen the keyword **typename** used to declare type parameters when defining a template (as in **template <typename ValueType>**), there is also another important use of this keyword in C++. If you closely examine the implementation of the **Map** class template in Figure 12-5, you will note that the declaration of the return type for the **findCell** helper method also uses the keyword **typename**. In this case, the keyword **typename** informs the compiler that the next token is, in fact, a type.

While it may seem that this information should be quite clear to the compiler, there are cases when dealing with templates where it is impossible for the compiler to determine if a particular identifier actually refers to a type instead of, say, a variable. The general rule is that **typename** is required for a type that is defined *inside* a class template when the reference is made *outside* of the implementation of that class template. Note that **typename** is only used for types declared within a template; without a template there is no ambiguity, and thus, no need for resolution.

The return type of the **findCell** method is **Map<ValueType>::cellT***. Since the type **cellT** is defined inside the **Map** template and you are referring to this type outside of the scope of the **Map** implementation, you must use the keyword **typename** to denote that **Map<ValueType>::cellT** is a type. It might seem that this usage is not outside the scope of the implementation of the **Map** template since you are declaring the return type of a **Map** method. C++, however, considers the return type to be outside the scope of the implementation, and you are therefore required to qualify the name of the type by using the full name **Map<ValueType>::cellT** rather than just **cellT**. You must also precede the return type with the keyword **typename** because this type is defined within a template.

Within the body of a template method, you can refer directly to the types defined within the template without the qualifier **Map<ValueType>::** or the **typename** keyword. For example, within the **get** method, the local variable **cp** is declared to be of type **cellT***. Because it is within the template implementation, this usage does not require the fully qualified type name **Map<ValueType>::cellT** or the keyword **typename**.

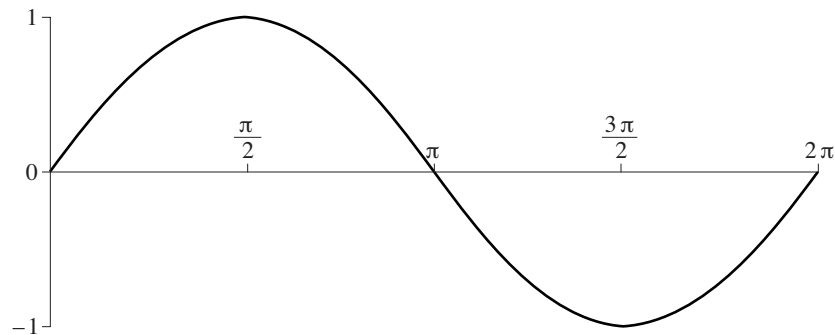
Even though this rule governing the use of **typename** is part of the C++ language standard, some compilers are not strict about enforcing this requirement. Some compilers will allow you to omit the **typename** keyword where it is required or allow you to insert it where it is not needed. You should not depend on the whim of such a compiler as it will make your code less portable if you try to compile it with a compiler that is not so lenient.

12.4 Functions as data

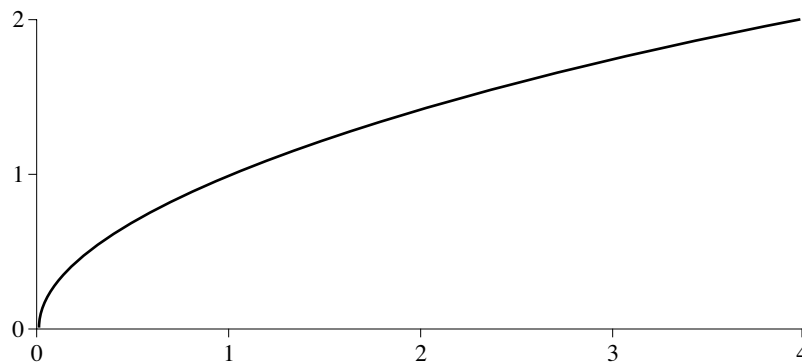
In the programming you have done up to this point, the concepts of functions and data structures have remained quite separate. Functions provide the means for representing an algorithm; data structures allow you to organize the information to which those algorithms are applied. Functions have only been part of the algorithmic structure, not part of the data structure. Being able to use functions as data values, however, often makes it much easier to design effective interfaces because doing so allows clients to specify operations as well as data. The next few sections offer several examples of how functions can be used as data that illustrate the importance of this technique.

A general plotting function

One of the easiest ways to illustrate the notion of functions as data is to design a simple plotting package that allows clients to plot graphs of mathematical functions. Suppose, for example, that you want to write a general function that plots the value of a function $f(x)$ for values of x between two specified limits. For example, if f is the trigonometric sine function and the limits are 0 and 2π , you get a graph that looks like this:



If f were the square root function plotted for values of x from 0 to 4, the shape of the resulting graph would instead look like this:



Note that the values in these two graphs are different on both the x and y axes. To produce these plots, the prototype of a general plotting procedure must take values representing the limits of the x value and the corresponding limits on the y value; these limits are called the **domain** and the **range** of the function, respectively. From a programming perspective, however, the interesting question is whether you can design a plotting procedure that allows you to supply the function itself as an argument. For example, assuming that **PI** is defined to be the mathematical constant π , you would like to be able to call

```
Plot(sin, 0, 2 * PI, -1, 1);
```

to produce the first sample graph and

```
Plot(sqrt, 0, 4, 0, 2);
```

to produce the second.

What would the prototype for **Plot** look like? The four arguments indicating the domain and range are easy to declare, but what is the type of the first argument? The implementation of **Plot** clearly needs to know something about the function it is given. You would not expect just any old function to work. For example, you would have no idea what to expect if you called

```
Plot(GetLine, 0, 2 * PI, -1, 1);
```



even though **GetLine** is a perfectly legal C++ function. The problem, of course, is that **GetLine** is a string function. For **Plot** to make sense, its first argument must be a function that takes a real number (presumably a **double**) and returns one as well. Thus, you can say that the first argument to **Plot** must be an instance chosen from the general class of functions that map one **double** into another **double**.

Declaring pointers to functions and function typedefs

To make functions fit more comfortably into the existing data structure facilities, the designers of C++ took advantage of the fact that the code for a function is stored somewhere in memory and can therefore be identified by the address of its first instruction. Thus, it makes sense to define pointers to functions, using a syntactic form that turns out to be compatible with all other C++ declarations, even though it appears a bit odd at first. If you want, for example, to declare a variable **fn** to be a pointer to a function taking and returning a **double**, you can write

```
double (*fn)(double);
```

It's important to remember the parentheses around ***fn** in the declaration of a function pointer. The alternative

```
double *fn(double);
```



declares **fn** as a function returning a pointer to a **double**.

In many cases, it makes more sense to use **typedef** to define the space of acceptable functions and then to define individual variables within that space. Thus, to define the name **doubleFnT** to indicate the type of pointers to functions that take one **double** and return a **double**, you would write the following

```
typedef double (*doubleFnT)(double);
```

This definition means that you could write the prototype for **Plot** as follows:

```
void Plot(doubleFnT fn, double minX, double maxX,  
          double minY, double maxY);
```

Implementing `Plot`

Once you have defined the prototype, you can write a simple implementation of `Plot` using the graphics library presented in Chapter 6. The implementation, which appears in Figure 12-6, assumes that the graphics window has been initialized in screen coordinates with the origin in the upper left corner, as is conventional with modern graphics libraries.

As simple as it is, the `Plot` provides a compelling illustration of the utility of functions as data. In the implementation, the parameter `fn` is a variable whose value is logically a function supplied by the caller. It can be `sin`, `sqrt`, or a user-defined function as long as its prototype matches the `doubleFnT` type, which consists of all functions taking and

Figure 12-5 The hash table implementation of the `Map` class

```

/*
 * Function: Plot
 * Usage: Plot(fn, minX, maxX, minY, maxY);
 * -----
 * This method plots the specified function (which must map
 * one double to another double) on the screen. The remaining
 * arguments indicate the domain and range of the function, which
 * are transformed so that they fill the dimensions of the
 * graphics window.
 */

void Plot(doubleFnT fn, double minX, double maxX,
          double minY, double maxY) {
    double width = GetWindowWidth();
    double height = GetWindowHeight();
    double nSteps = (int) width;
    double dx = (maxX - minX) / nSteps;
    for (int i = 0; i < nSteps; i++) {
        double x = minX + i * dx;
        double y = fn(x);
        if (y > maxY) y = maxY + 1;
        if (y < minY) y = minY - 1;
        double sx = (x - minX) / (maxX - minX) * width;
        double sy = height - (y - minY) / (maxY - minY) * height;
        if (i == 0) {
            MovePen(sx, sy);
        } else {
            DrawLineTo(sx, sy);
        }
    }
}

/*
 * Function: DrawLineTo
 * Usage: DrawLineTo(x, y);
 * -----
 * This function draws a line from the current point to (x, y).
 * In many applications, this function is more convenient than
 * DrawLine, which specifies relative motion using dx and dy.
 */

void DrawLineTo(double x, double y) {
    DrawLine(x - GetCurrentX(), y - GetCurrentY());
}

```


returning a **double**. Inside the implementation, calls to **fn** are interpreted as calls to the function whose address is stored in the variable **fn**, which is declared as a function pointer. Thus, these calls end up invoking the function that the caller specified as the first argument to **Plot**.

A generic sorting function

The various implementations of sorting functions presented in Chapter 8 make it clear that sorting is a complex problem that requires considerable care to implement efficiently. Because sorting is so important—and because its performance depends so heavily on its implementation—it would make sense to design a high-efficiency, general-purpose sorting function. Such a generic function would allow clients to sort arrays of any type without having to worry about the details of writing a sorting procedure that is both efficient and correct.

The details of the various sorting algorithms are discussed in detail in Chapter 8. The issue here is how a sorting function can be written in a way that allows the element type of the array being sorted to vary from call to call. The solution depends on function pointers for its generality and would not be possible in C++ without them.

In principle, sorting can be applied to data values of any ordered type. The structure of the sorting algorithm does not depend on the base type of the array. As long as the implementation can compare two values and move elements from one position in the array to another, it should be possible to use the same basic code to sort an array of any type. On the other hand, writing such a function is tricky because C++ requires function prototypes to declare the types of its parameters. How should a generic sorting routine declare the type of the array being sorted?

When you are defining the prototype for **Sort**, what you would like is something that looks like this:

```
void Sort(any array[], int n);
```



Unfortunately, no type **any** exists in C++. However, the template facility you used earlier to implement flexible container classes can also be used for generic functions. All you need to do is introduce the function with a template marker that names the placeholder being used for the type. Within the function header and body you refer to the array element type using the placeholder name. For example, here is a function template that implements the selection sort algorithm for arrays of any type:

```
template <typename Type>
void Sort(Type array[], int n) {
    for (int i = 0; i < n; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[minIndex]) minIndex = j;
        }
        Type temp = array[i];
        array[i] = array[minIndex];
        array[minIndex] = temp;
    }
}
```

A function template serves as a pattern from which versions specialized for particular types can be created on demand. A client can specify the desired element type for **Sort** by adding a qualifier to the name. Thus, **Sort<int>** refers to a version of the **Sort** function template specialized to operate on integer arrays. However, it is not always necessary to include an explicit qualifier. When a client makes a call to a function template, the compiler infers the type of the placeholder if it can. For example, if the client makes a call to just **Sort** without qualification, the base type of the first argument is assumed as the placeholder type. In the client code below, the first call passes an array of **string** elements, so the compiler will assume a version of **Sort** where the placeholder type has been filled in with **string**. For the second call, the compiler uses a second version of **Sort** in which the placeholder has been replaced with **int**.

```
string names[5] = {"Owen", "Stuart", "Claire", "Mike", "Nick"};
int numbers[8] = {5, 8, -34, 15, 12, 22, 28, 1};

Sort(names, 5);
Sort(numbers, 8);
```

Even with the template placeholder, the **Sort** function is still not completely general. The code in the body of the function template assumes that it will be able to compare two elements in the array using the operator **<** to determine the order between the two. For primitive types, this operator has an established meaning. The designers of the **string** class, moreover, made sure that the **<** operator also works on strings by having it compare the ASCII values of the characters in order from left to right. If, however, you call **Sort** with an array of records, the compiler will create a version of the **Sort** function that will attempt to apply **<** to two of those structures. That version of the template will fail to compile since arbitrary structures cannot be compared using the **<** operator. The fields within that record have to be compared against each other in some application-specific way. The bottom line is that the **Sort** function can perform the correct comparison only if the client tells it how. The best way for the client to communicate this information is by passing a comparison function to **Sort**. You can add an additional argument that allows for a client-supplied comparison function.

What is the type of a comparison function? Typically, a comparison functions will take two arguments representing the values to be compared and return an integer. The conventional interpretation of the integer result is that the return value is less than 0 when the first argument is less than the second, equal to 0 when the arguments are equal, and greater than 0 when the first argument is greater than the second. What are the types for the values? If the array elements being sorted are strings, an appropriate comparison function compares two strings. If the array elements are **studentT** records, the comparison function compares two **studentTs**. When adding this third parameter to the **Sort** procedure, you must describe its prototype in terms of the template placeholder:

```
template <typename Type>
void Sort(Type array[], int n, int (*cmp)(Type, Type)) {
    for (int i = 0; i < n; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (cmp(array[j], array[minIndex]) < 0) minIndex = j;
        }
        Type temp = array[i];
        array[i] = array[minIndex];
        array[minIndex] = temp;
    }
}
```

The client-supplied comparison function makes it possible for the client to specify how to compare values of the specific type stored in the array. For example, to compare two **pointT** structures, you could supply a function like the following:

```
int PointCompare(pointT p1, pointT p2) {
    if (p1.x < p2.x) return -1;
    if (p1.x > p2.x) return 1;
    if (p1.y < p2.y) return -1;
    if (p1.y > p2.y) return 1;
    return 0;
}
```

To sort an array of **pointT** structures, you could then call

```
Sort(points, nPoints, PointCompare);
```

One advantage of the **Sort** function template that uses the client-supplied comparison function is that it allows the client to overrule the default behavior of the < operator. If you want to sort an array of integers into descending order or to sort an array of strings by length, you could do so by controlling the comparison function. If you define a function

```
int LengthCompare(string s1, string s2) {
    return s1.length() - s2.length();
}
```

you can use it to sort the array of strings by length instead of alphabetically, as follows:

```
Sort(names, 5, LengthCompare);
```

One small nicety lost by adding the third argument to the **Sort** function template is that it always requires a comparison function. The convenience of using the built-in relational operations for those types that don't require a specialized comparison function has been lost. You can restore this convenience by providing the client with a version of the comparison function in the required format that performs an application of the built-in relational operations. Such a function would be useful for the built-in types. Rather than creating multiple functions—one to compare integers, another to compare strings, and so on—you can define this convenience function as a function template. This allows the client to specialize it as needed for the types to be compared.

Figure 12-7 shows the **cmpfn.h** interface that supplies such a function template. If you supply **OperatorCmp** as the default argument to the third parameter to **Sort**, you can leave out the comparison function argument as long as you are satisfied with the default as shown in the prototype below allows a client who wishes to use the default relational ordering can leave off the third argument when calling **Sort**. The prototype for **Sort** would then look like this

```
template <typename Type>
void Sort(Type array[], int n,
         int (*cmp)(Type, Type) = OperatorCmp);
```

Updating the **Sort** prototype in this way means that you can once again sort the arrays of integers and strings by calling

```
Sort(names, 5);
Sort(numbers, 8);
```

but still supply a comparison function when sorting points.

Figure 12-7 The `cmpfn.h` interface

```

/*
 * File: cmpfn.h
 * -----
 * This interface exports a comparison function template.
 */

#ifndef _cmpfn_h
#define _cmpfn_h

/*
 * Function template: OperatorCmp
 * Usage:  int sign = OperatorCmp(val1, val2);
 * -----
 * This function template is a generic function to
 * compare two values using the built-in == and < operators.
 * It is supplied as a convenience for those situations
 * where a comparison function is required, and the type
 * has a built-in ordering that you would like to use.
 */

template <typename Type>
int OperatorCmp(Type one, Type two) {
    if (one == two) return 0;
    if (one < two) return -1;
    return 1;
}

#endif

```

Functions that are passed by clients to an implementation, like the comparison function in this example, are called **callback functions** because the implementation uses these functions to make calls back to the client. Because callback functions make it possible for the client to specify operations along with data, they play an important role in modern programming methodology and are an integral part of the implementation of object-oriented languages.

12.5 Mapping functions

Callback functions make it possible to achieve the effect of a map iterator, even if they do not get us all the way to the **Iterator** class itself. Iterators—or something very much like them—are essential to many applications. For example, it would be impossible to complete the word frequency program from Figure 4-9 back in Chapter 4. That program, if you recall, used a `Map<int>` to keep track of the frequency of each word in an input text, such as the following lines from *Macbeth*:

macbeth.txt

```

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

```

If you tried to implement the same program using the version of the `map.h` interface as it has been developed in this chapter, you would be able to compute all of the necessary counts. The only problem is that you couldn't report those counts to anyone. The facility you are missing is the iterator mechanism, which made it possible to write the function

```

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;
    }
}

```

Without the iterator, you have no way to implement this operation.

It is easy to describe what you need to implement this function. In pseudocode, the **DisplayWordCounts** function has the following form:

```

void DisplayWordCounts(Map<int> & wordCounts) {
    for (each entry in the map) {
        Display the key and its corresponding count.
    }
}

```

If you think about the italicized parts of this pseudocode, it quickly becomes clear that the task of displaying the map must be shared by both the client and the implementation. Only the client knows how to display the key and the count, which means that the body of the loop is the client's responsibility. On the other hand, only the implementation knows what keys exist in the map. Thus, the process of stepping through each entry in the map must be driven from the implementation side.

One strategy that makes it possible to manage this shared responsibility between the client and implementation is to have the **Map** class interface export a function that allows the client to execute a callback operation on every key/value pair in the map. Such a function is called a **mapping function**. For example, if the **Map** class were to include a public method called **mapAll** that took a callback function as an argument, all you would have to do to display the entries is to write a function **DisplayEntry** to display the value of a single entry and then use **mapAll** to apply **DisplayEntry** to the map as a whole.

Mapping over entries in a map

The first step toward defining a function like **mapAll** is to decide on the structure of the callback function. If you are writing a callback function to process a single entry in the map, you would assume that its arguments consist of a key and the corresponding value. Given a **Map<int>**, you would expect the prototype for **DisplayEntry** to look like this:

```
void DisplayEntry(string key, int value);
```

By generalizing from the prototype for **DisplayEntry**, it is easy to define a type that represents the type of functions you can use with **mapAll**. Each such function must take a key of type **string** and a value of type **ValueType** and return no value. Understanding the structure of the callback function makes it possible to write the interface description for **mapAll**, which appears in Figure 12-8.

Once you have the definition of **mapAll**, it is then easy to complete the implementation of the word frequency program by writing the functions **DisplayWordFrequencies** and **DisplayEntry**, as follows:

```

void DisplayWordFrequencies(Map<int> & map) {
    map.mapAll(DisplayEntry);
}

```

Figure 12-8 A simple mapping function for the Map class

```

/* Method: mapAll
 * Usage: map.mapAll(fn);
 * -----
 * This method goes through every entry in the map and calls
 * the function fn, passing it the following arguments:
 * the current key and its associated value.
 */

void mapAll(void (*fn)(string, ValueType));

```

```

void DisplayEntry(string key, int value) {
    cout << left << setw(15) << key
         << right << setw(5) << value << endl;
}

```

DisplayWordFrequencies just calls **mapAll**, which in turn calls **DisplayEntry** on each key/value pair. **DisplayEntry** displays the key along with the integer count in a field of the appropriate size.

Implementing mapAll

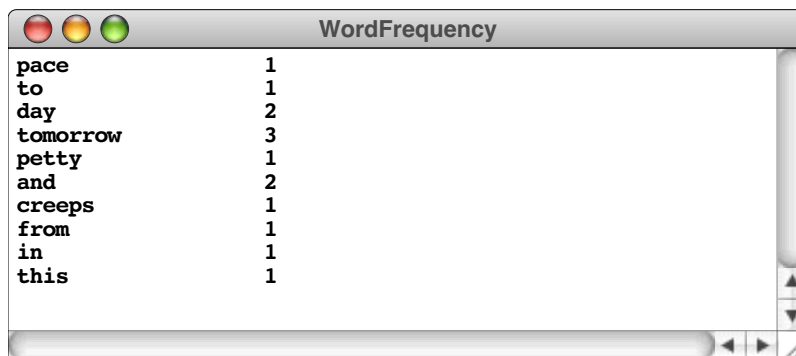
The only remaining task is to add the code for **mapAll** to the **mapimpl.cpp** file. Because the hash table consists of an array of linked lists, mapping over the entries in the table is simply a matter of going through each of the buckets in the array and then iterating through every entry in the corresponding chain, as illustrated by the following implementation:

```

template <typename ValueType>
void Map<ValueType>::mapAll(mapFnT fn) {
    for (int i = 0; i < N_BUCKETS; i++) {
        for (cellT *cp = buckets[i]; cp != NULL; cp = cp->link) {
            fn(cp->key, cp->value);
        }
    }
}

```

Putting all the code together leaves you with an implementation of the word frequency program that does pretty much everything you want. If you run the program in its current form on the **macbeth.txt** data file, the tabulation looks like this:



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

All the counts are correct. There is, however, still this nagging problem that the words are not displayed in any recognizable order. The **mapAll** function has simply cycled through the entries as they appear in the internal structure, which means that the order depends on how the keys are organized into the different buckets.

The fact that the keys are not processed in order is not the fault of **mapAll**, which is behaving exactly as it should. Trying to recode **mapAll** so that the keys were always delivered in alphabetical order would make the function more complex and less efficient. Because the keys are jumbled in the hash table, putting them back in order would require **mapAll** to sort the entries, which requires at least $O(N \log N)$ time. Because many clients of **mapAll** may not care about the order of the keys, it is better to assign the responsibility of sorting the keys to any clients that need to do so.

Passing client information to a callback function

Even if **mapAll** is behaving as it should, there will still be occasions when you would like to be able to sort the output of a program like the one that computes word frequencies. The simplest expedient might be to write the data to an output file instead of to the console. If you did that, you could simply run the unsorted output of the current program through one of many utility programs that sort data files. In this example, what you would like to do is replace **DisplayWordFrequencies** with a new method

```
void DumpWordFrequencies(Map<int> & map, string filename);
```

that writes the same information to a data file.

Unfortunately, writing **DumpWordFrequencies** is not quite as easy as it might seem. In order to write data to an output file, you have to open the file using a statement like

```
outfile.open(filename.c_str());
```

The problem arises when you try to write the callback function. You can't simply write a function **DumpEntry** like this:

```
void DumpEntry(string key, int value) {  
    outfile << key << " " << value << endl;  
}
```



The crux of the problem is that you can't use the variable **outfile** in **DumpEntry** because it is declared as a local variable in **DumpWordFrequencies**. When the compiler reads the function **DumpEntry**, it regards the variable **outfile** as undeclared. Moreover, since **DumpWordFrequencies** calls **mapAll**, which in turn calls **DumpEntry**, the value of **outfile** must be passed across the interface boundary to **mapAll** and then back again to **DumpEntry**.

The easiest way to communicate this information is to extend the mapping operation to add a third parameter to the callback function. The type of that parameter needs to be flexible so it is declared as a template. This addition of a **clientData** argument allows the client to pass an arbitrary value through **mapAll** to the callback function. The prototype for the **mapAll** function that supports this **clientData** argument is:

```
template<typename ClientDataType>  
void mapAll(void (*fn)(string, ValueType, ClientDataType &),  
            ClientDataType & data);
```

Using the `clientData` parameter allows you to implement `DumpWordFrequencies` and `DumpEntry`, as follows:

```
void DumpWordFrequencies(Map<int> & map, string filename) {
    ofstream outfile;
    outfile.open(filename.c_str());
    if (outfile.fail()) Error("Can't open output file");
    map.mapAll(DumpEntry, outfile);
    outfile.close();
}

void DumpEntry(string key, int value, ofstream & outfile) {
    outfile << key << " " << value << endl;
}
```

The `clientData` parameter also makes it possible to retain information between independent invocations to a callback function. If you need, for example, to update a variable every time the callback function is executed, you can pass that variable using the `clientData` parameter. Each call then has access to the information in that variable.

A note on function types and methods

In some cases, you may be tempted to define your callback function as a method of a class. For example, in the case above, you may consider defining `DisplayEntry` as a method of the `Map` class template rather than as a standalone function. The problem with taking such an approach is a subtle, but important, detail of C++ class members. Recall from Chapter 9 that each method has the pointer `this` available to it, which is a pointer to the object that the method was invoked on. You may have wondered about how the `this` pointer is communicated to the function. Behind the scenes, the `this` pointer is passed as a hidden extra parameter when invoking a method. As a result, the true prototype of a method includes that extra parameter which will cause it to not match the prototype for the callback function expected by `mapAll`. The simplest solution is to define callback function as free functions, outside of any class.

Summary

The focus of this chapter has been the implementation of the `Map` class. As part of the discussion of maps, the chapter also introduced the concept of using functions as data—a powerful general technique that has important applications in the map domain.

Important points in this chapter include:

- Maps can be implemented very efficiently using a strategy called *hashing*, in which keys are converted to an array index by a function that tends to distribute the keys uniformly throughout the array. As long as your implementation permits the table space to expand with the number of entries, hashing allows both `put` and `get` to operate in constant time on average.
- The detailed design of a hash function is subtle and requires mathematical analysis to achieve optimum performance. Even so, any hash function that delivers integer values in the specified range always produces correct results.
- The `put` and `get` functions alone are not sufficient for many applications of maps. The client often needs the ability to perform some operation for all the keys defined in a map. This chapter presents two techniques—mapping functions and iterators—that provide this capability.
- C++ makes it possible to use a pointer to the code for a function as a data value. This facility makes it possible for the client of an interface to pass information to the

implementation about how a particular data value behaves. The usual approach is for the client to supply a *callback function* that can then be invoked on the implementation side of the interface boundary.

- A common use of callback functions in practice is in the context of *mapping functions*, which allow clients to invoke an operation on all internal elements of a container.

Review questions

1. Describe the relationship between dictionaries and maps.
2. What does the term *key* mean in the context of a map?
3. Name the two operations that define the characteristic behavior of a map.
4. What types are used in the **Map** class interface to represent keys and values?
5. What reasons justify committing to a key type of **string** while the value type is left open as a template placeholder?
6. What simple strategy could you use if you wanted to use integers as keys, given the definition of the **Map** class interface used in this chapter?
7. Why does the **get** method raise an error instead of returning a sentinel when retrieving the value for a non-existent key?
8. True or false: If you represent a map using an array sorted by keys, the binary search algorithm allows you to implement both **put** and **get** in $O(\log N)$ time.
9. Describe the dictionary analogue to the hashing algorithm.
10. What disadvantages would you expect from using the ASCII value of the first character in a key as its hash code?
11. What is meant by the term *bucket* in the implementation of a hash table? What constitutes a collision?
12. Explain the operation of the **findCell** function in the **map.cpp** implementation given in Figure 12-5.
13. The **hash** function that appears in the text has an internal structure similar to that of a random-number generator. If you took that similarity too literally, however, you might be tempted to write the following **hash** function:

```
int hash(string s, int nBuckets) {
    return RandomInteger(0, nBuckets - 1);
}
```



Why would this approach fail?

14. What time-space tradeoff arises in the implementation of a hash table?
15. What is meant by the term *rehashing*?
16. What advantages are cited in the chapter for regarding functions as part of the data structure?

17. Explain in English the effect of the following definition:

```
typedef string (*stringFnT)(string);
```

Give an example of a library function that fits this function type.

18. What **typedef** line would you use to define **proct** as the class of functions that take no arguments and return no results?
19. Describe the parameters used by the **Sort** function described in the section “A generic sorting function”. Why is each of those parameters necessary?
20. What advantage does the version of the **Sort** function that uses a client-supplied comparator have over one that always uses the built-in **<** operator?
21. Define the terms *callback function* and *mapping function*.
22. What is one limitation of the context of a callback function?

Programming exercises

1. Write a main program that uses the Roman numeral table described in section 12.1 to translate Roman numerals into their Arabic equivalents. You compute the value of a Roman numeral by adding up the values corresponding to each letter, subject to one exception: If the value of a letter is less than the letter that follows it, that value should be subtracted from the total instead of added. For example, the string

MCMLXIX

corresponds to

$$1000 - 100 + 1000 + 50 + 10 - 1 + 10$$

or 1969. The **c** and the **i** are subtracted rather than added because they are followed by a letter with a larger value.

2. Modify the code in Figure 12-3 so that **put** always keeps the keys in sorted order in the array. Change the implementation of the private **findKey** method so that it uses binary search to find the key in $O(\log N)$ time.
3. Evaluate the performance of the hashing algorithm by writing a procedure called **DisplayHashTableStatistics** that counts the length of each hash chain and displays the mean and standard deviation of those lengths. The mean is equivalent to the traditional average. The standard deviation is a measure of the how much the individual values tend to differ from the mean. The formula for calculating the standard deviation is

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (\text{len}_{avg} - \text{len}_i)^2}{N}}$$

where N is the number of buckets, len_i is the length of bucket chain i , and len_{avg} is the average chain length. If the hash function is working well, the standard deviation should be relatively small in comparison to the mean, particularly as the number of entries increases.

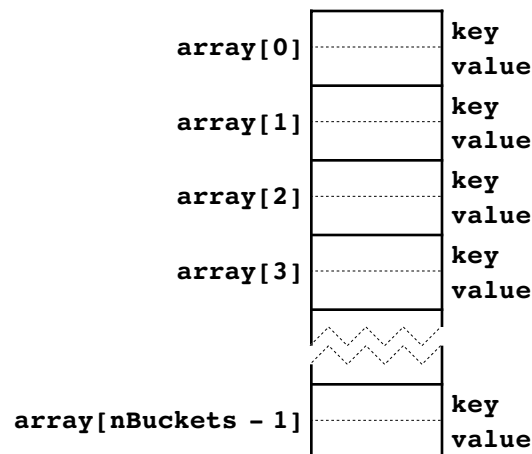
4. In certain applications, it is useful to extend the `map.h` interface so that you can insert a temporary definition for a particular key, hiding away any previous value associated with that key. Later in the program, you can remove the temporary definition, restoring the next most recent one. For example, you could use such a mechanism to capture the effect of local variables, which come into existence when a function is called and disappear again when the function returns.

Implement such a facility by adding the method

```
void insert(string key, ValueType value);
```

to the `Map` class. Because the `put` and `get` functions always find the first entry in the chain, you can ensure that `insert` hides the previous definitions simply by adding each new entry at the beginning of the list for a particular hash bucket. Moreover, as long as the implementation of `remove` function deletes the first occurrence of an entry from its hash chain, you can use `remove` to remove the most recently inserted definition for a key.

5. Extend the implementation of the hash table from Figure 12-5 so that the internal table can expand dynamically. Your implementation should keep track of the load factor for the hash table and perform a rehashing operation if the load factor exceeds the limit indicated by a constant `REHASH_THRESHOLD` defined in the private section.
6. Although the bucket-chaining approach used in the text is extremely effective in practice, other strategies exist for resolving collisions in hash tables. In the early days of computing—when memories were small enough that the cost of introducing extra pointers was taken seriously—hash tables often used a more memory-efficient strategy called **open addressing**, in which the key/value pairs are stored directly in the array, like this:



For example, if a key hashes to bucket #3, the open-addressing implementation of a hash table tries to put that key and its value directly into the entry at `array[3]`.

The problem with this approach is that `array[3]` may already be assigned to another key that hashes to the same bucket. The simplest approach to dealing with collisions of this sort is to store each new key in the first free cell at or after its expected hash position. Thus, if a key hashes to bucket #3, the `put` and `get` functions first try to find or insert that key in `array[3]`. If that entry is filled with a different key, however, these functions move on to try `array[4]`, continuing the process until they find an empty entry or an entry with a matching key. As in the ring-buffer implementation of queues in Chapter 11, if the index advances past the

end of the array, it should wrap back to the beginning. This strategy for resolving collisions is called **linear probing**.

Reimplement the **Map** class using open addressing with linear probing. Make sure your function generates an error if the client tries to enter a new key into a table that is already full. The biggest challenge is to make sure your implementation supports the **remove** method.

7. As noted at the beginning of this chapter, the **map.h** interface in Figure 12-1 was deliberately simplified by eliminating deep copying, selection using square brackets, and the ability to iterate over the keys in a map. Although deep copying is beyond the scope of this text, you have enough background from Chapter 11 to implement the other two features. Rewrite the various files associated with the **Map** class to implement bracket selection and an **Iterator** class.
8. When you design a class, it often makes sense to include a function that creates a copy of an existing object by duplicating its internal data. For example, in the **Map** class interface, it would be useful to include a method

```
void copyEntriesInto(Map<ValueType> & copy);
```

that copies all the key/value pairs from the receiver into the copy parameter. Thus, client code that wanted to make a snapshot of the contents of a map named **varTable** could use the statement

```
varTable.copyEntriesInto(saveTable);
```

Later on, the client could restore the previous contents of the table by executing the following code:

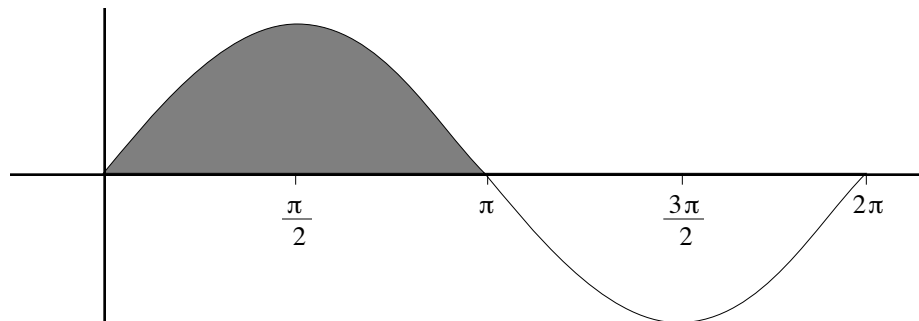
```
saveTable.copyEntriesInto(varTable);
```

Implement the **copyEntriesInto** function as an extension to the hash table code in Figure 12-5.

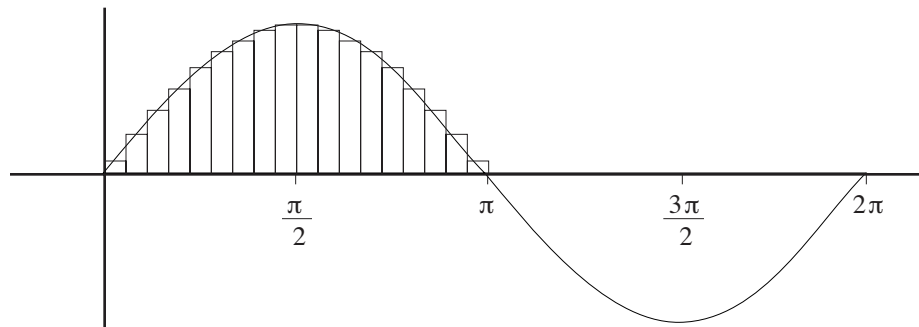
9. Write an implementation of the Quicksort algorithm, which is coded specifically for integers in Figure 8-6, so that its prototype corresponds to the one given in this chapter:

```
template<typename Type>  
void Sort(Type array[], int n,  
         int (*cmp)(Type, Type) = OperatorCmp);
```

10. In calculus, the **definite integral** of a function is defined to be the area bounded horizontally by two specified limits and vertically by the x -axis and the value of the function. For example, the definite integral of **sin** in the range 0 to π is the area of the shaded region in the following diagram:



You can compute an approximation to this area by adding up the area of small rectangles of a fixed width, where the height is given by the value of the function at the midpoint of the rectangle:



Design the prototype (including any associated type definitions) and write the implementation for a function **Integrate** that calculates the definite integral by summing the areas of a set of rectangles. For example, to calculate the area of the shaded region in the earlier example, the client would write

```
double value = Integrate(sin, 0, 3.1415926, 18);
```

where the last argument is the number of rectangles into which the area gets divided; the larger this value, the more accurate the approximation.

Note that any region that falls below the x -axis is treated as negative area. Thus, if you computed the definite integral of **sin** from 0 to 2π , the result would be 0 because the areas above and below the x -axis cancel each other out.

11. Another useful generic operation is a general implementation of the binary search algorithm with the following prototype:

```
template<typename Type>
int BinarySearch(Type key, Type array[], int n,
                 int (*cmp)(Type, Type) = OperatorCmp);
```

The **key** argument is the value for which the function is searching, and the remaining arguments are equivalent to those used in **Sort**. The function returns the index of an array element containing the value **key**, assuming that the value appears in the array at all. If not, **BSearch** returns -1 .

12. The easiest way to ensure that the words generated by the **WordFrequency** program appear in alphabetical order is to enter all the words in a **Lexicon**, which guarantees alphabetical order whenever it iterates over its contents. That strategy is very easy to implement if you have access to iterators or the **foreach** structure, but a bit trickier if you use only mapping functions, which are exported by both the **Map** and **Lexicon** classes. Modify the code for **WordFrequency** so that it generates an alphabetical listing without using iterators or **foreach**.
13. Function pointers make it possible to design a more general map interface that does not require keys to be of type **string**. If keys are represented instead as a second template parameter, the client must tell the implementation how to compare and hash individual keys by supplying pointers to appropriate functions.

Modify the map package so that the map is parametrized by two template parameters, the key and value types, and the constructor requires two additional arguments:

```
Map(cmpFnT cmpFn, hashFnT hashFn);
```

which are client-supplied functions to compare two keys and derive a hash code. Designing a suitable definition for the type **hashFnT** is part of the exercise.

14. Use the **mapAll** method to implement a function **LongestKey** that takes a **Map<int>** and returns the longest key in that map.

Chapter 13

Trees

*I like trees because they seem more resigned to the way
they have to live than other things do.*

— Willa Cather, *O Pioneers!*, 1913

As you have seen in several earlier chapters, linked lists make it possible to represent an ordered collection of values without using arrays. The link pointers associated with each cell form a linear chain that defines the underlying order. Although linked lists require more memory space than arrays and are less efficient for operations such as selecting a value at a particular index position, they have the advantage that insertion and deletion operations can be performed in constant time.

The use of pointers to define the ordering relationship among a set of values is considerably more powerful than the linked-list example suggests and is by no means limited to creating linear structures. In this chapter, you will learn about a data structure that uses pointers to model hierarchical relationships. That structure is called a **tree**, which is defined to be a collection of individual entries called **nodes** for which the following properties hold:

- As long as the tree contains any nodes at all, there is a specific node called the **root** that forms the top of a hierarchy.
- Every other node is connected to the root by a unique line of descent.

Tree-structured hierarchies occur in many contexts outside of computer science. The most familiar example is the family tree, which is discussed in the next section. Other examples include

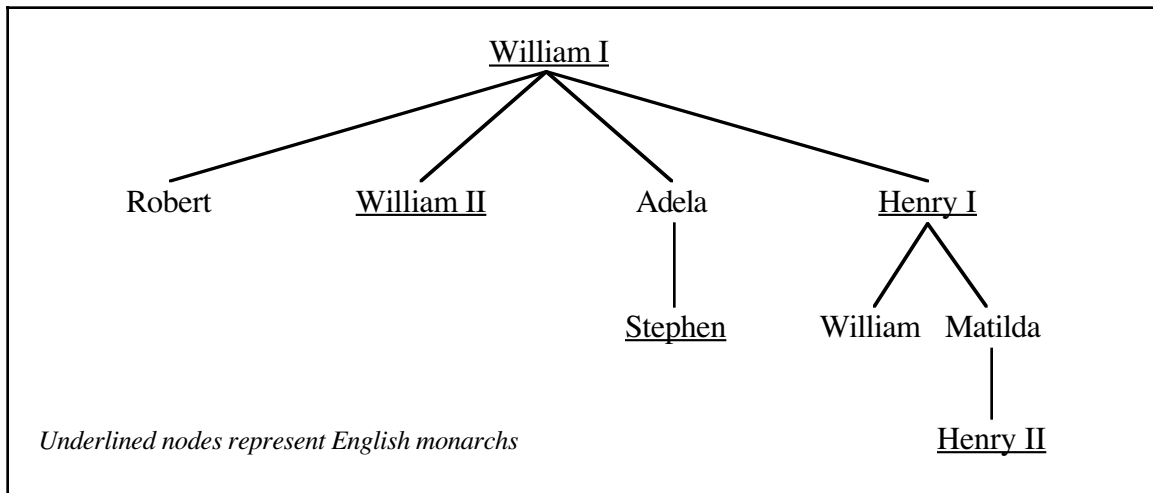
- *Game trees.* The game trees introduced in the section on “The minimax strategy” in Chapter 7 have a branching pattern that is typical of trees. The current position is the root of the tree; the branches lead to positions that might occur later in the game.
- *Biological classifications.* The classification system for living organisms, which was developed in the eighteenth century by the Swedish botanist Carolus Linnaeus, is structured as a tree. The root of the tree is all living things. From there, the classification system branches to form separate kingdoms, of which animals and plants are the most familiar. From there, the hierarchy continues down through several additional levels until it defines an individual species.
- *Organization charts.* Many businesses are structured so that each employee reports to a single supervisor, forming a tree that extends up to the company president, who represents the root.
- *Directory hierarchies.* On most modern computers, files are stored in directories that form a tree. There is a top-level directory that represents the root, which can contain files along with other directories. Those directories may contain subdirectories, which gives rise to the hierarchical structure representative of trees.

13.1 Family trees

Family trees provide a convenient way to represent the lines of descent from a single individual through a series of generations. For example, the diagram in Figure 13-1 shows the family tree of the House of Normandy, which ruled England after the accession of William I at the Battle of Hastings in 1066. The structure of the diagram fits the definition of a tree given in the preceding section. William I is the root of the tree, and all other individuals in the chart are connected to William I through a unique line of descent.

Terminology used to describe trees

The family tree in Figure 13-1 makes it easy to introduce the terminology computer scientists use to describe tree structures. Each node in a tree may have several **children**, but only a single **parent** in the tree. In the context of trees, the words **ancestor** and

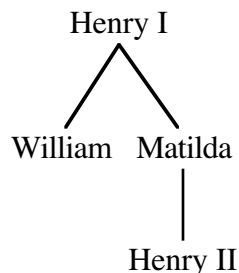
Figure 13-1 Family tree for the House of Normandy

descendant have exactly the same meaning as they do in English. The line of descent through Henry I and Matilda shows that Henry II is a descendant of William I, which in turn implies that William I is an ancestor of Henry II. Similarly, the term **siblings** is used to refer to two nodes that share the same parent, such as Robert and Adela.

Although most of the terms used to describe trees come directly from the family-tree analogue, others—like the word *root*—come from the botanical metaphor instead. At the opposite end of the tree from the root, there are nodes that have no children, which are called **leaves**. Nodes that are neither the root nor a leaf are called **interior** nodes. For example, in Figure 13-1, Robert, William II, Stephen, William, and Henry II represent leaf nodes; Adela, Henry I, and Matilda represent interior nodes. The **height** of a tree is defined to be the length of the longest path from the root to a leaf. Thus, the height of the tree shown in Figure 13-1 is 4, because there are four nodes on the path from William I to Henry II, which is longer than any other path from the root.

The recursive nature of a tree

One of the most important things to notice about any tree is that the same branching pattern occurs at every level of the decomposition. If you take any node in a tree together with all its descendants, the result fits the definition of a tree. For example, if you extract the portion of Figure 13-1 beginning at Henry I, you get the following tree:



A tree formed by extracting a node and its descendants from an existing tree is called a **subtree** of the original one. The tree in this diagram, for example, is the subtree rooted at Henry I.

The fact that each node in a tree can be considered the root of its own subtree underscores the recursive nature of tree structures. If you think about trees from a

recursive perspective, a tree is simply a node and a set—possibly empty in the case of a leaf node—of attached subtrees. The recursive character of trees is fundamental to their underlying representation as well as to most algorithms that operate on trees.

Representing family trees in C++

In order to represent any type of tree in C++, you need some way to model the hierarchical relationships among the data values. In most cases, the easiest way to represent the parent/child relationship is to include a pointer in the parent that points to the child. If you use this strategy, each node is a structure that contains—in addition to other data specific to the node itself—pointers to each of its children. In general, it works well to define a node as the structure itself and to define a tree as a pointer to that structure. This definition is mutually recursive even in its English conception because of the following relationship:

- Trees are pointers to nodes.
- Nodes are structures that contain trees.

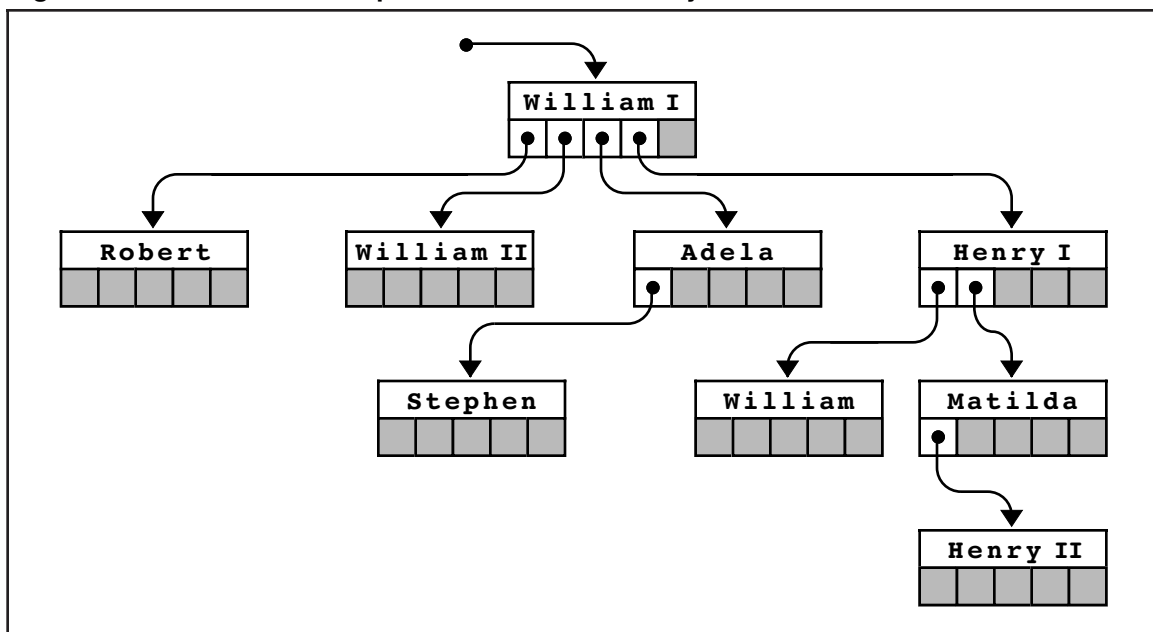
How would you use this recursive insight to design a structure suitable for storing the data in a family tree such as the one shown in Figure 13-1? Each node consists of a name of a person and a set of pointers to its children. If you store the child pointers in a vector, a node has the following form as a C++ structure:

```
struct familyTreeNodeT {
    string name;
    Vector<familyTreeNodeT *> children;
};
```

A family tree is simply a pointer to one of these nodes.

A diagram showing the internal representation of the royal family tree appears in Figure 13-2. To keep the figure neat and orderly, Figure 13-2 represents the children as if they were stored in a five-element array; in fact, the **children** field is a vector that grows to

Figure 13-2 Pointer-based representation of the family tree



accommodate any number of children. You will have a chance to explore other strategies for storing the children, such as keeping them in a linked list rather than a vector, in the exercises at the end of this chapter.

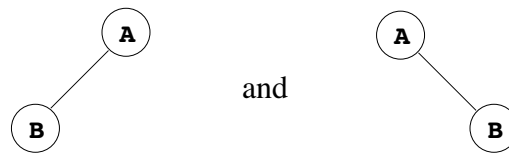
13.2 Binary search trees

Although it is possible to illustrate tree algorithms using family trees, it is more effective to do so in a simpler environment that has more direct application to programming. The family-tree example provides a useful framework for introducing the terminology used to describe trees, but suffers in practice from the complication that each node can have an arbitrary number of children. In many programming contexts, it is reasonable to restrict the number of children to make the resulting trees easier to implement.

One of the most important subclasses of trees—which has many practical applications—is a **binary tree**, which is defined to be a tree in which the following additional properties hold:

- Each node in the tree has at most two children.
- Every node except the root is designated as either a *left child* or a *right child* of its parent.

The second condition emphasizes the fact that child nodes in a binary tree are ordered with respect to their parents. For example, the binary trees



are different trees, even though they consist of the same nodes. In both cases, the node labeled **B** is a child of the root node labeled **A**, but it is a left child in the first tree and a right child in the second.

The fact that the nodes in a binary tree have a defined geometrical relationship makes it convenient to represent ordered collections of data using binary trees. The most common application uses a special class of binary tree called a **binary search tree**, which is defined by the following properties:

1. Every node contains—possibly in addition to other data—a special value called a *key* that defines the order of the nodes.
2. Key values are *unique*, in the sense that no key can appear more than once in the tree.
3. At every node in the tree, the key value must be greater than all the keys in the subtree rooted at its left child and less than all the keys in the subtree rooted at its right child.

Although this definition is formally correct, it almost certainly seems confusing at first glance. To make sense of the definition and begin to understand why constructing a tree that meets these conditions might be useful, it helps to go back and look at a specific problem for which binary search trees represent a potential solution strategy.

The underlying motivation for using binary search trees

In Chapter 12, one of the strategies proposed for representing maps—before the hashing algorithm made other options seem far less attractive—was to store the key/value pairs in

an array. This strategy has a useful computational property: if you keep the keys in sorted order, you can write an implementation of **get** that runs in $O(\log N)$ time. All you need to do is employ the binary search algorithm, which was introduced in Chapter 5. Unfortunately, the array representation does not offer any equally efficient way to code the **put** function. Although **put** can use binary search to determine where any new key fits into the array, maintaining the sorted order requires $O(N)$ time because each subsequent array element must be shifted to make room for the new entry.

This problem brings to mind a similar situation that arose in Chapter 11. When arrays were used to represent the editor buffer, inserting a new character was a linear-time operation. In that case, the solution was to replace the array with a linked list. Is it possible that a similar strategy would improve the performance of **put** for the map? After all, inserting a new element into a linked list—as long as you have a pointer to the cell prior to the insertion point—is a constant-time operation.

The trouble with linked lists is that they do not support the binary search algorithm in any efficient way. Binary search depends on being able to find the middle element in constant time. In an array, finding the middle element is easy. In a linked list, the only way to do so is to iterate through all the link pointers in the first half of the list.

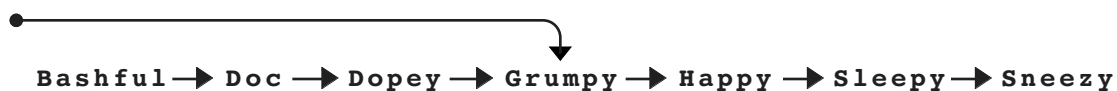
To get a more concrete sense of why linked lists have this limitation, suppose that you have a linked list containing the following seven elements:



The elements in this list appear in lexicographic order, which is the order imposed by their internal character codes.

Given a linked list of this sort, you can easily find the first element, because the initial pointer gives you its address. From there, you can follow the link pointer to find the second element. On the other hand, there is no easy way to locate the element that occurs halfway through the sequence. To do so, you have to walk through each chain pointer, counting up to $N/2$. This operation requires linear time, which completely negates the efficiency advantage of binary search. If binary search is to offer any improvement in efficiency, the data structure must enable you to find the middle element quickly.

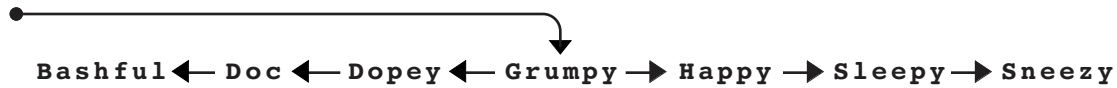
Although it might at first seem silly, it is useful to consider what happens if you simply point at the middle of the list instead of the beginning:



In this diagram, you have no problem at all finding the middle element. It's immediately accessible through the list pointer. The problem, however, is that you've thrown away the first half of the list. The pointers in the structure provide access to **Grumpy** and any name that follows it in the chain, but there is no longer any way to reach **Bashful**, **Doc**, and **Dopey**.

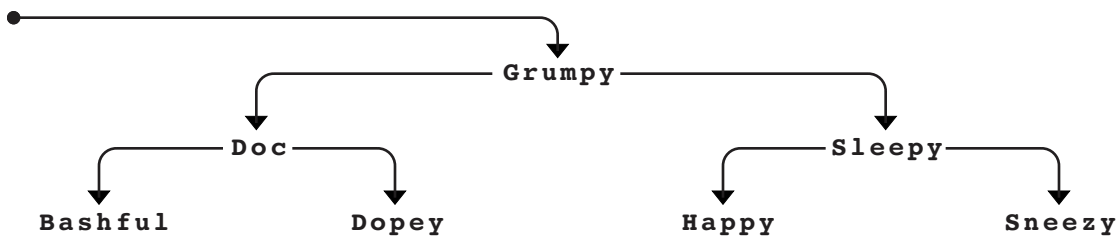
If you think about the situation from **Grumpy**'s point of view, the general outline of the solution becomes clear. What you need is to have two chains emanating from the **Grumpy** cell: one that consists of the cells whose names precede **Grumpy** and another for the cells

whose names follow **Grumpy** in the alphabet. In the conceptual diagram, all you need to do is reverse the arrows:



Each of the strings is now accessible, and you can easily divide the entire list in half.

At this point, you need to apply the same strategy recursively. The binary search algorithm requires you to find the middle of not only the original list but its sublists as well. You therefore need to restructure the lists that precede and follow **Grumpy**, using the same decomposition strategy. Every cell points in two directions: to the midpoint of the list that precedes it and to the midpoint of the list that follows it. Applying this process transforms the original list into the following binary tree:



The most important feature about this particular style of binary tree is that it is ordered. For any particular node in the tree, the string it contains must follow all the strings in the subtree descending to the left and precede all strings in the subtree to the right. In this example, **Grumpy** comes after **Doc**, **Bashful**, and **Dopey** but before **Sleepy**, **Happy**, and **Sneezy**. The same rule, however, applies at each level, so the node containing **Doc** comes after the **Bashful** node but before the **Dopey** node. The formal definition of a binary search tree, which appears at the end of the preceding section, simply ensures that every node in the tree obeys this ordering rule.

Finding nodes in a binary search tree

The fundamental advantage of a binary search tree is that you can use the binary search algorithm to find a particular node. Suppose, for example, that you are looking for the node containing the string **Happy** in the tree diagram shown at the end of the preceding section. The first step is to compare **Happy** with **Grumpy**, which appears at the root of the tree. Since **Happy** comes after **Grumpy** in lexicographic order, you know that the **Happy** node, if it exists, must be in the right subtree. The next step, therefore, is to compare **Happy** and **Sleepy**. In this case, **Happy** comes before **Sleepy** and must therefore be in the left subtree of this node. That subtree consists of a single node, which contains the correct name.

Because trees are recursive structures, it is easy to code the search algorithm in its recursive form. For concreteness, let's suppose that the type definition for **nodeT** looks like this:

```

struct nodeT {
    string key;
    nodeT *left, *right;
};
  
```

Given this definition, you can easily write a function **FindNode** that implements the binary search algorithm, as follows:

```
nodeT *FindNode(nodeT *t, string key) {
    if (t == NULL) return NULL;
    if (key == t->key) return t;
    if (key < t->key) {
        return FindNode(t->left, key);
    } else {
        return FindNode(t->right, key);
    }
}
```

If the tree is empty, the desired node is clearly not there, and **FindNode** returns the value **NULL** as a sentinel indicating that the key cannot be found. If the tree is not equal to **NULL**, the implementation checks to see whether the desired key matches the one in the current node. If so, **FindNode** returns a pointer to the current node. If the keys do not match, **FindNode** proceeds recursively, looking in either the left or right subtree depending on the result of the key comparison.

Inserting new nodes in a binary search tree

The next question to consider is how to create a binary search tree in the first place. The simplest approach is to begin with an empty tree and then call an **InsertNode** function to insert new keys into the tree, one at a time. As each new key is inserted, it is important to maintain the ordering relationship among the nodes of the tree. To make sure the **FindNode** function continues to work, the code for **InsertNode** must use binary search to identify the correct insertion point.

As with **FindNode**, the code for **InsertNode** can proceed recursively beginning at the root of the tree. At each node, **InsertNode** must compare the new key to the key in the current node. If the new key precedes the existing one, the new key belongs in the left subtree. Conversely, if the new key follows the one in the current node, it belongs in the right subtree. Eventually, the process will encounter a **NULL** subtree that represents the point in the tree where the new node needs to be added. At this point, the **InsertNode** implementation must replace the **NULL** pointer with a new node initialized to contain a copy of the key.

The code for **InsertNode**, however, is a bit tricky. The difficulty comes from the fact that **InsertNode** must be able to change the value of the binary search tree by adding a new node. Since the function needs to change the values of the argument, it must be passed by reference. Instead of taking a **nodeT *** as its argument the way **FindNode** does, **InsertNode** must instead take a **nodeT * &**. The prototype for **InsertNode** therefore looks like this:

```
void InsertNode(nodeT * &t, string key);
```

Once you understand the prototype for the **InsertNode** function, writing the code is not particularly hard. A complete implementation of **InsertNode** appears in Figure 13-3. If **t** is **NULL**, **InsertNode** creates a new node, initializes its fields, and then replaces the **NULL** pointer in the existing structure with a pointer to the new node. If **t** is not **NULL**, **InsertNode** compares the new key with the one stored at the root of the tree **t**. If the keys match, the key is already in the tree and no further operations are required. If not, **InsertNode** uses the result of the comparison to determine whether to insert the key in the left or the right subtree and then makes the appropriate recursive call.

Figure 13-3 Standard algorithm for inserting a node into a binary search tree

```

/*
 * Inserts the specified key at the appropriate location in the
 * binary search tree rooted at t. Note that t must be passed
 * by reference, since it is possible to change the root.
 */

void InsertNode(nodeT * & t, string key) {
    if (t == NULL) {
        t = new nodeT;
        t->key = key;
        t->left = t->right = NULL;
        return;
    }
    if (key == t->key) return;
    if (key < t->key) {
        InsertNode(t->left, key);
    } else {
        InsertNode(t->right, key);
    }
}

```

Because the code for **InsertNode** seems complicated until you've seen it work, it makes sense to go through the process of inserting a few keys in some detail. Suppose, for example, that you have declared and initialized an empty tree as follows:

```
nodeT *dwarfTree = NULL;
```

These statements create a local variable **dwarfTree** that lives at some address in memory, as illustrated by the following diagram:



The actual address depends on the structure of the computer's memory system and the way the compiler allocates addresses. In this example, each memory word has been assigned an arbitrary address to make it easier to follow the operation of the code.

What happens if you call

```
InsertNode(dwarfTree, "Grumpy");
```

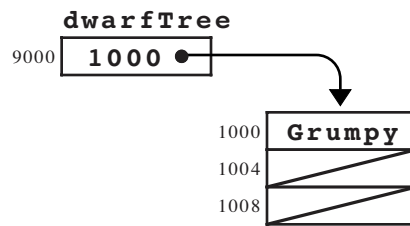
starting with this initial configuration in which **dwarfTree** is empty? In the frame for **InsertNode**, the variable **t** is a reference parameter, aliased to the variable **dwarfTree** stored at address 9000. The first step in the code checks if **t** is set to **NULL**, which is true in this case, so it executes the body of the **if** statement

```

if (t == NULL) {
    t = new nodeT;
    t->key = key;
    t->left = t->right = NULL;
    return;
}

```

which has the effect of creating a new node, initializing it to hold the key **Grumpy**, and then storing the address of the new node into the reference parameter **t**. When the function returns, the tree looks like this:

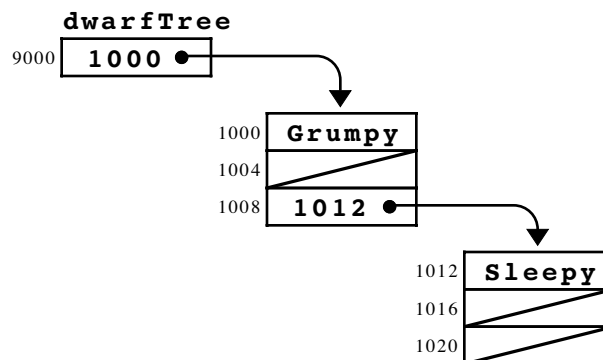


This structure correctly represents the binary search tree with a single node containing **Grumpy**.

What happens if you then use **InsertNode** to insert **Sleepy** into the tree? As before, the initial call generates a stack frame in which the reference parameter **t** is aliased to **dwarfTree**. This time, however, the value of the tree **t** is no longer **NULL**. **dwarfTree** now contains the address of the node containing **Grumpy**. Because **Sleepy** comes after **Grumpy** in the lexicographical order, the code for **InsertNode** continues with the following recursive call:

```
InsertNode(t->right, key);
```

At this point, the recursive call looks much like the insertion of **Grumpy** into the original empty tree. The only difference is that the reference parameter **t** now refers to a field within an existing node. The effect of the recursive call is therefore to store a pointer to a new node containing **Sleepy** in the **right** field of the root node, which gives rise to the following configuration:

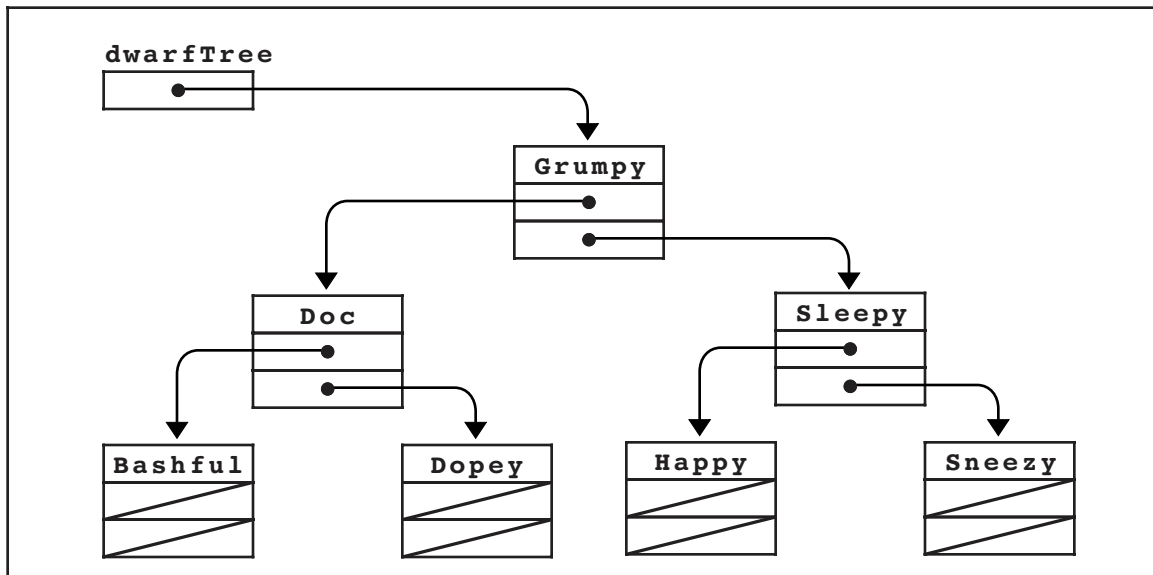


Additional calls to **InsertNode** will create additional nodes and insert them into the structure in a way that preserves the ordering constraint required for binary search trees. For example, if you insert the names of the five remaining dwarves in the order **Doc**, **Bashful**, **Dopey**, **Happy**, and **Sneezy**, you end up with the binary search tree shown in Figure 13-4.

Tree traversals

The structure of a binary search tree makes it easy to go through the nodes of the tree in the order specified by the keys. For example, you can use the following function to display the keys in a binary search tree in lexicographic order:

Figure 13-4 Binary search tree for the seven dwarves



```

void DisplayTree(nodeT *t) {
    if (t != NULL) {
        DisplayTree(t->left);
        cout << t->key << endl;
        DisplayTree(t->right);
    }
}

```

Thus, if you call **DisplayTree** on the tree shown in Figure 13-4, you get the following output:



At each recursive level, **DisplayTree** checks to see whether the tree is empty. If it is, **DisplayTree** has no work to do. If not, the ordering of the recursive calls ensures that the output appears in the correct order. The first recursive call displays the keys that precede the current node, all of which must appear in the left subtree. Displaying the nodes in the left subtree before the current one therefore maintains the correct order. Similarly, it is important to display the key from the current node before making the last recursive call, which displays the keys that occur later in the ASCII sequence and therefore appear in the right subtree.

The process of going through the nodes of a tree and performing some operation at each node is called **traversing** or **walking** the tree. In many cases, you will want to traverse a tree in the order imposed by the keys, as in the **DisplayTree** example. This approach, which consists of processing the current node between the recursive calls to the left and right subtrees, is called an **inorder traversal**. There are, however, two other

types of tree traversals that occur frequently in the context of binary trees, which are called **preorder** and **postorder** traversals. In the preorder traversal, the current node is processed before either of its subtrees; in the postorder traversal, the subtrees are processed first, followed by the current node. These traversal styles are illustrated by the functions **PreOrderWalk** and **PostOrderWalk**, which appear in Figure 13-5. The sample output associated with each function shows the result of applying that function to the balanced binary search tree in Figure 13-4.

13.3 Balanced trees

Although the recursive strategy used to implement **InsertNode** guarantees that the nodes are organized as a legal binary search tree, the structure of the tree depends on the order in which the nodes are inserted. The tree in Figure 13-4, for example, was generated by inserting the names of the dwarves in the following order:

Grumpy, Sleepy, Doc, Bashful, Dopey, Happy, Sneezy

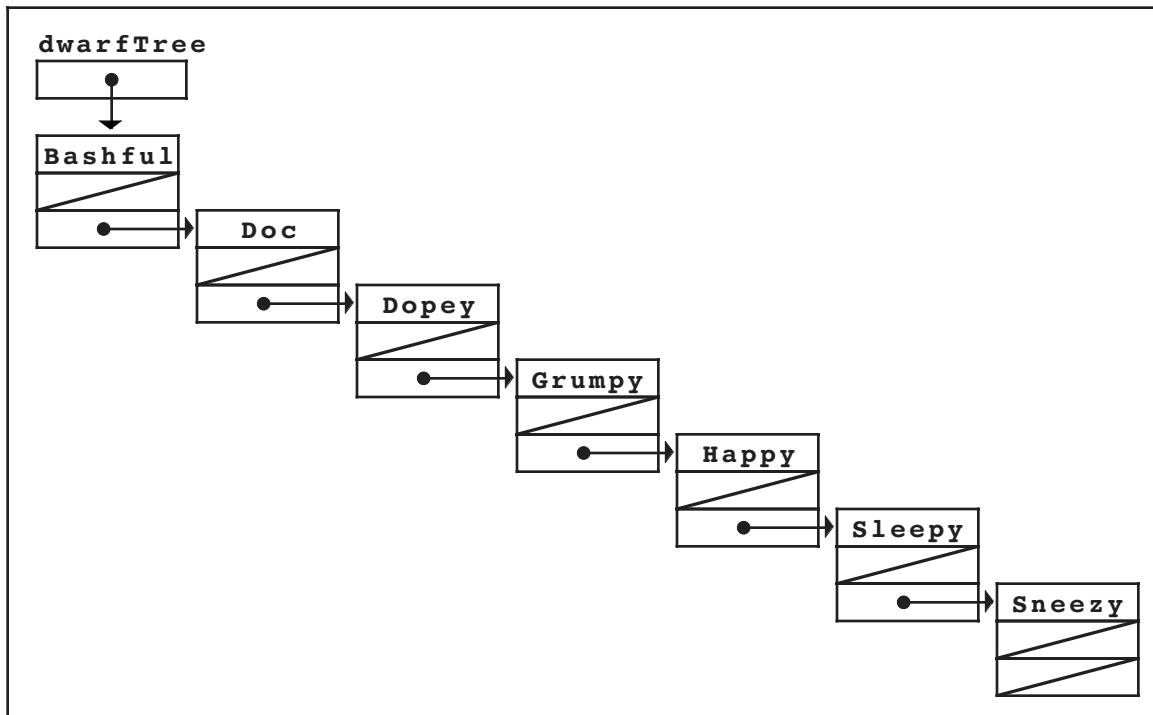
Suppose that you had instead entered the names of the dwarves in alphabetical order. The first call to **InsertNode** would insert **Bashful** at the root of the tree. Subsequent calls would insert **Doc** after **Bashful**, **Dopey** after **Doc**, and so on, appending each new node to the **right** chain of the previously one.

The resulting figure, which is shown in Figure 13-6, looks more like a linked list than a tree. Nonetheless, the tree in Figure 13-6 maintains the property that the key field in any node follows all the keys in its left subtree and precedes all the keys in its right subtree. It therefore fits the definition of a binary search tree, so the **FindNode** function will operate correctly. The running time of the **FindNode** algorithm, however, is proportional to the height of the tree, which means that the structure of the tree can have a significant impact on the algorithmic performance. If a binary search tree is shaped like the one shown in Figure 13-4, the time required to find a key in the tree will be $O(\log N)$. On the other hand, if the tree is shaped like the one in Figure 13-6, the running time will deteriorate to $O(N)$.

The binary search algorithm used to implement **FindNode** achieves its ideal performance only if the left and right subtrees have roughly the same height at each level

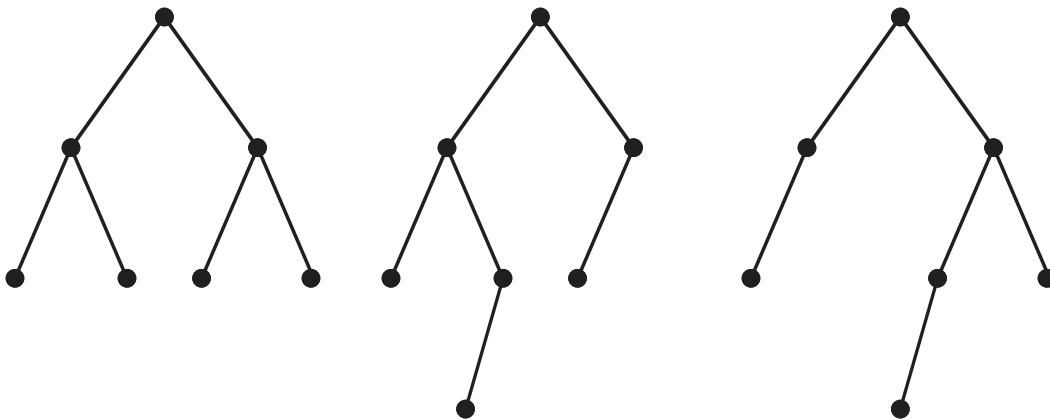
Figure 13-5 Preorder and postorder traversals of a binary search tree

<pre>void PreOrderWalk(nodeT *t) { if (t != NULL) { cout << t->key << endl; PreOrderWalk(t->left); PreOrderWalk(t->right); } }</pre> Grumpy Doc Bashful Dopey Sleepy Happy Sneezy	<pre>void PostOrderWalk(nodeT *t) { if (t != NULL) { PostOrderWalk(t->left); PostOrderWalk(t->right); cout << t->key << endl; } }</pre> Bashful Dopey Doc Happy Sneezy Sleepy Grumpy

Figure 13-6 Unbalanced tree that results if the names are inserted in lexicographic order

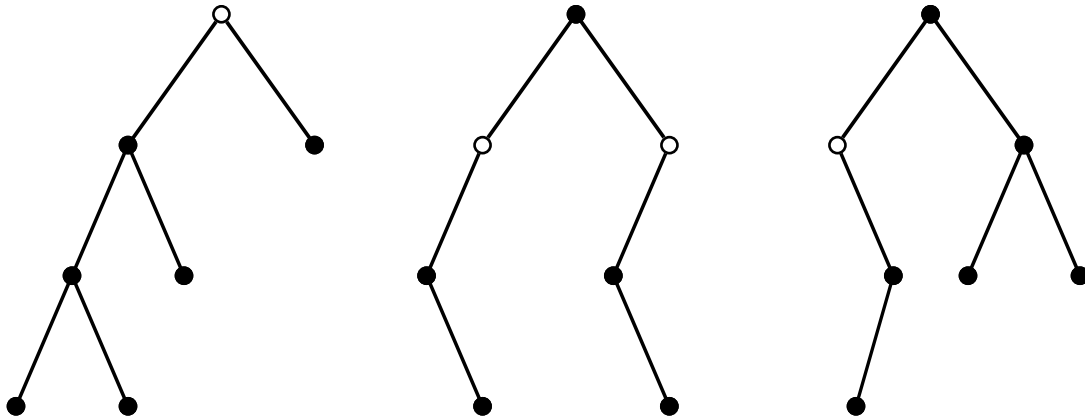
of the tree. Trees in which this property holds—such as the tree in Figure 13-4—are said to be **balanced**. More formally, a binary tree is defined to be balanced if, at each node, the height of the left and right subtrees differ by at most one.

To illustrate this definition of a balanced binary tree, each of the following diagrams shows a balanced arrangement of a tree with seven nodes:



The tree diagram on the left is optimally balanced in the sense that the heights of the two subtrees at each node are always equal. Such an arrangement is possible, however, only if the number of nodes is one less than a power of two. If the number of nodes does not meet this condition, there will be some point in the tree where the heights of the subtrees differ to some extent. By allowing the heights of the subtrees to differ by one, the definition of a balanced tree provides some flexibility in the structure of a tree without adversely affecting its computational performance.

The diagrams at the top of the next page represent unbalanced trees.



In each diagram, the nodes at which the balanced-tree definition fails are shown as open circles. In the leftmost tree, for example, the left subtree of the root node has height 3 while the right subtree has height 1.

Tree-balancing strategies

Binary search trees are useful in practice only if it is possible to avoid the worst-case behavior associated with unbalanced trees. As trees become unbalanced, the **FindNode** and **InsertNode** operations become linear in their running time. If the performance of binary trees deteriorates to $O(N)$, you might as well use a sorted array to store the values. With a sorted array, it requires $O(\log N)$ time to implement **FindNode** and $O(N)$ time to implement **InsertNode**. From a computational perspective, the performance of the array-based algorithms is therefore superior to that of unbalanced trees, even though the array implementation is considerably easier to write.

What makes binary search trees useful as a programming tool is the fact that you can keep them balanced as you build them. The basic idea is to extend the implementation of **InsertNode** so that it keeps track of whether the tree is balanced while inserting new nodes. If the tree ever becomes out of balance, **InsertNode** must rearrange the nodes in the tree so that the balance is restored without disturbing the ordering relationships that make the tree a binary search tree. Assuming that it is possible to rearrange a tree in time proportional to its height, both **FindNode** and **InsertNode** can be implemented in $O(\log N)$ time.

Algorithms for maintaining balance in a binary tree have been studied extensively in computer science. The algorithms used today to implement balanced binary trees are quite sophisticated and have benefited enormously from theoretical research. Most of these algorithms, however, are difficult to explain without reviewing mathematical results beyond the scope of this text. To demonstrate that such algorithms are indeed possible, the next few sections present one of the first tree-balancing algorithms, which was published in 1962 by the Russian mathematicians Georgii Adel'son-Vel'skii and Evgenii Landis and has since been known by the initials AVL. Although the AVL algorithm has been largely replaced in practice by more modern approaches, it has the advantage of being considerably easier to explain. Moreover, the operations used to implement the basic strategy reappear in many other algorithms, which makes it a good model for more modern techniques.

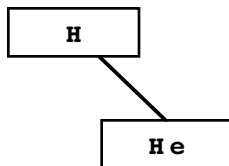
Illustrating the AVL idea

Before you attempt to understand the implementation of the AVL algorithm in detail, it helps to follow through the process of inserting nodes into a binary search tree to see

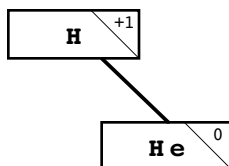
what can go wrong and, if possible, what steps you can take to fix any problems that arise. Let's imagine that you want to create a binary search tree in which the nodes contain the symbols for the chemical elements. For example, the first six elements are

H (Hydrogen)
He (Helium)
Li (Lithium)
Be (Beryllium)
B (Boron)
C (Carbon)

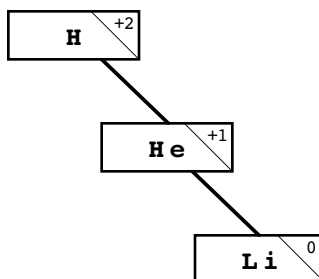
What happens if you insert the chemical symbols for these elements in the indicated order, which is how these elements appear in the periodic table? The first insertion is easy because the tree is initially empty. The node containing the symbol **H** becomes the root of the tree. If you call **InsertNode** on the symbol **He**, the new node will be added after the node containing **H**, because **He** comes after **H** in lexicographic order. Thus, the first two nodes in the tree are arranged like this:



To keep track of whether the tree is balanced, the AVL algorithm associates an integer with each node, which is simply the height of the right subtree minus the height of the left subtree. This value is called the **balance factor** of the node. In the simple tree that contains the symbols for the first two elements, the balance factors, which are shown here in the upper right corner of each node, look like this:

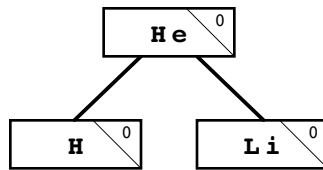


So far, the tree is balanced because none of the nodes has a balance factor whose absolute value is greater than 1. That situation changes, however, when you add the next element. If you follow the standard insertion algorithm, adding **Li** results in the following configuration:



Here, the root node is out of balance because its right subtree has height 2 and its left subtree has height 0, which differ by more than one.

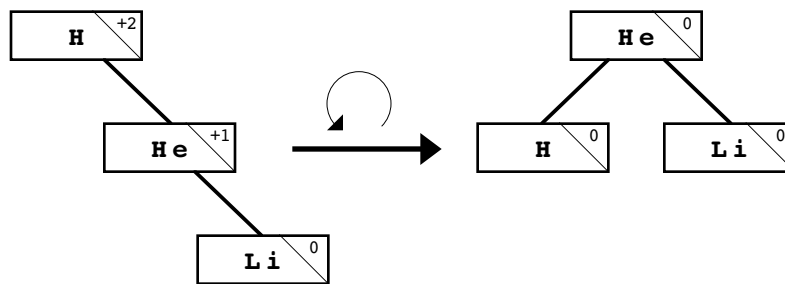
To fix the imbalance, you need to restructure the tree. For this set of nodes, there is only one balanced configuration in which the nodes are correctly ordered with respect to each other. That tree has **He** at the root, with **H** and **Li** in the left and right subtrees, as follows:



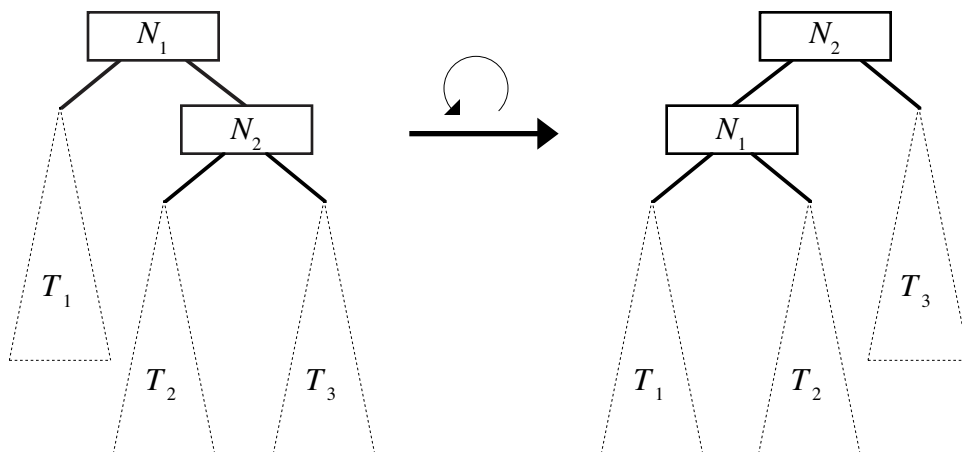
This tree is once again balanced, but an important question remains: how do you know what operations to perform in order to restore the balance in a tree?

Single rotations

The fundamental insight behind the AVL strategy is that you can always restore balance to a tree by a simple rearrangement of the nodes. If you think about what steps were necessary to correct the imbalance in the preceding example, it is clear that the **He** node moves upward to become the root while **H** moves downward to become its child. To a certain extent, the transformation has the characteristic of rotating the **H** and **He** nodes one position to the left, like this:

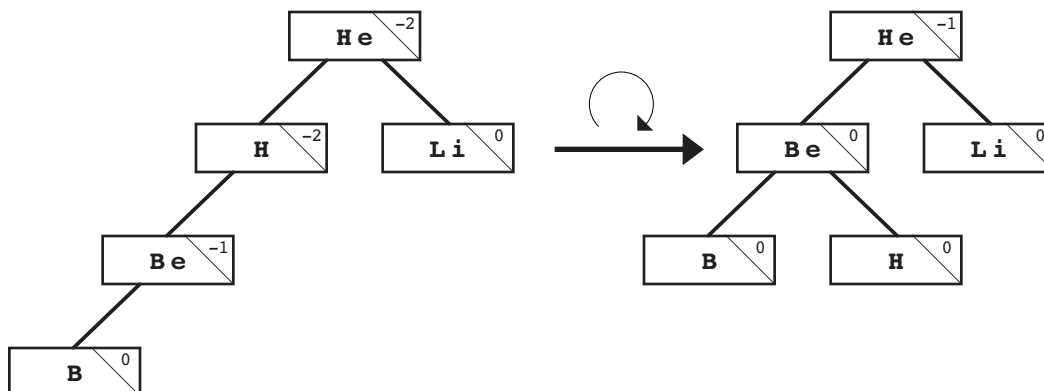


In general, you can always perform this type of rotation operation on any two nodes in a binary search tree without invalidating the relative ordering of the nodes, even if the nodes have subtrees descending from them. In a tree with additional nodes, the basic operation looks like this:

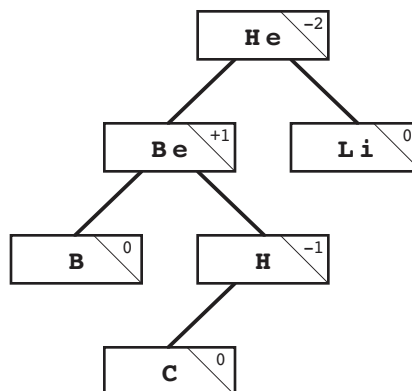


Note that the T_2 subtree, which would otherwise be orphaned by this process, must be reattached to N_1 after N_1 and N_2 change positions.

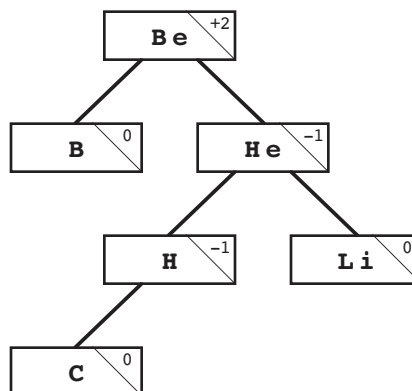
The two nodes involved in the rotation operation are called the **axis** of the rotation. In the example consisting of the elements **H**, **He**, and **Li**, the rotation was performed around the **H-He** axis. Because this operation moves nodes to the left, the operation illustrated by this diagram is called a **left rotation**. If a tree is out of balance in the opposite direction, you can apply a symmetric operation called a **right rotation**, in which all the operations are simply reversed. For example, the symbols for the next two elements—**Be** and **B**—each get added at the left edge of the tree. To rebalance the tree, you must perform a right rotation around the **Be-H** axis, as illustrated in the following diagram:



Unfortunately, simple rotation operations are not always sufficient to restore balance to a tree. Consider, for example, what happens when you add **c** to the tree. Before you perform any balancing operations, the tree looks like this:



The **He** node at the root of the tree is out of balance. If you try to correct the imbalance by rotating the tree to the right around the **Be-He** axis, you get the following tree:



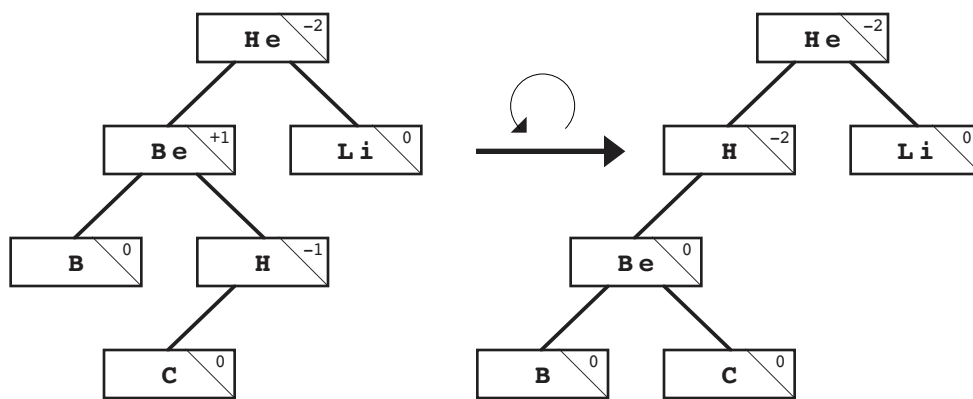
The tree is still unbalanced.

After the rotation, the tree is just as unbalanced as it was before. The only difference is that the root node is now unbalanced in the opposite direction.

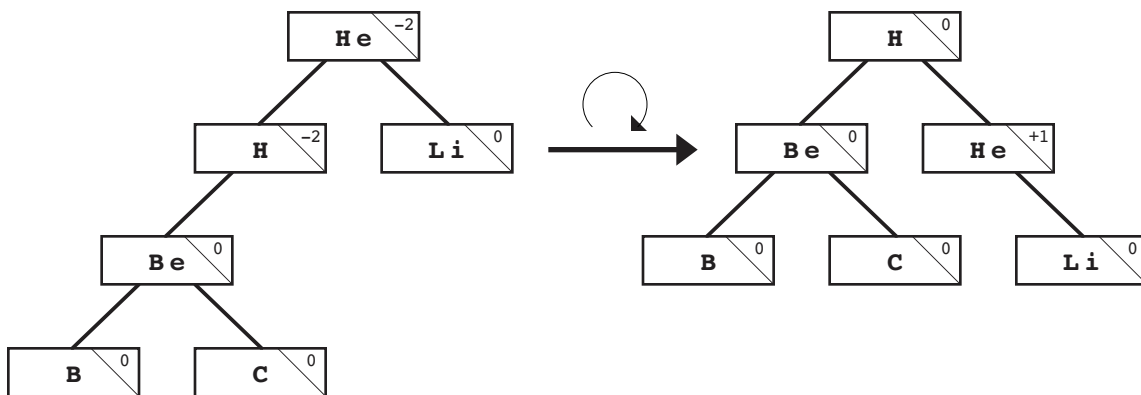
Double rotations

The problem in this last example arises because the nodes involved in the rotation have balance factors with opposite signs. When this situation occurs, a single rotation is not enough. To fix the problem, you need to make two rotations. Before rotating the out-of-balance node, you rotate its child in the opposite direction. Rotating the child gives the balance factors in the parent and child the same sign, which means that the following rotation will succeed. This pair of operations is called a **double rotation**.

As an illustration of the double-rotation operation, consider the preceding unbalanced tree of elements just after adding the symbol **c**. The first step is to rotate the tree to the left around the **Be-H** axis, like this:



The resulting tree is still out of balance at the root node, but the **H** and **He** nodes now have balance factors that share the same sign. In this configuration, a single rotation to the right around the **H-He** axis restores balance to the tree, as follows:



In their paper describing these trees, Adel'son-Vel'skii and Landis demonstrated the following properties of their tree-balancing algorithm:

- If you insert a new node into an AVL tree, you can always restore its balance by performing at most one operation, which is either a single or a double rotation.
- After you complete the rotation operation, the height of the subtree at the axis of rotation is always the same as it was before inserting the new node. This property ensures that none of the balance factors change at any higher levels of the tree.

Implementing the AVL algorithm

Although the process involves quite a few details, implementing **InsertNode** for AVL trees is not as difficult as you might imagine. The first change you need to make is to include a new field in the node structure that allows you to keep track of the balance factor, as follows:

```
struct nodeT {
    string key;
    nodeT *left, *right;
    int bf;
};
```

The code for **InsertNode** itself appears in Figure 13-7. As you can see from the code, **InsertNode** is implemented as a wrapper to a function **InsertAVL**, which at first glance seems to have the same prototype. The parameters to the two functions are indeed the same. The only difference is that **InsertAVL** returns an integer value that represents the change in the height of the tree after inserting the node. This return value, which will always be 0 or 1, makes it easy to fix the structure of the tree as the code makes its way back through the level of recursive calls. The simple cases are

1. Adding a node in place of a previously **NULL** tree, which increases the height by one
2. Encountering an existing node containing the key, which leaves the height unchanged

In the recursive cases, the code first adds the new node to the appropriate subtree, keeping track of the change in height in the local variable **delta**. If the height of the subtree to which the insertion was made has not changed, then the balance factor in the current node must also remain the same. If, however, the subtree increased in height, there are three possibilities:

1. *That subtree was previously shorter than the other subtree in this node.* In this case, inserting the new node actually makes the tree more balanced than it was previously. The balance factor of the current node becomes 0, and the height of the subtree rooted there remains the same as before.
2. *The two subtrees in the current node were previously the same size.* In this case, increasing the size of one of the subtrees makes the current node slightly out of balance, but not to the point that any corrective action is required. The balance factor becomes -1 or $+1$, as appropriate, and the function returns 1 to show that the height of the subtree rooted at this node has increased.
3. *The subtree that grew taller was already taller than the other subtree.* When this situation occurs, the tree has become seriously out of balance, because one subtree is now two nodes higher than the other. At this point, the code must execute the appropriate rotation operations to correct the imbalance. If the balance factors in the current node and the root of the subtree that expanded have the same sign, a single rotation is sufficient. If not, the code must perform a double rotation. After performing the rotations, the code must correct the balance factors in the nodes whose positions have changed. The effect of the single and double rotation operations on the balance factors in the node is shown in Figure 13-8.

Using the code for the AVL algorithm shown in Figure 13-7 ensures that the binary search tree remains in balance as new nodes are added. As a result, both **FindNode** and **InsertNode** will run in $O(\log N)$ time. Even without the AVL extension, however, the code will continue to work. The advantage of the AVL strategy is that it guarantees good performance, at some cost in the complexity of the code.

Figure 13-7 Code to insert a node into an AVL tree

```

/*
 * Function: InsertNode
 * Usage: InsertNode(t, key);
 * -----
 * This function calls InsertAVL and discards the result.
 */

void InsertNode(nodeT * & t, string key) {
    InsertAVL(t, key);
}

/*
 * Function: InsertAVL
 * Usage: delta = InsertAVL(t, key);
 * -----
 * This function enters the key into the tree whose is passed by
 * reference as the first argument. The return value is the change
 * in depth in the tree, which is used to correct the balance
 * factors in ancestor nodes.
 */

int InsertAVL(nodeT * & t, string key) {
    if (t == NULL) {
        t = new nodeT;
        t->key = key;
        t->bf = 0;
        t->left = t->right = NULL;
        return +1;
    }
    if (key == t->key) return 0;
    if (key < t->key) {
        int delta = InsertAVL(t->left, key);
        if (delta == 0) return 0;
        switch (t->bf) {
            case +1: t->bf = 0; return 0;
            case 0: t->bf = -1; return +1;
            case -1: FixLeftImbalance(t); return 0;
        }
    } else {
        int delta = InsertAVL(t->right, key);
        if (delta == 0) return 0;
        switch (t->bf) {
            case -1: t->bf = 0; return 0;
            case 0: t->bf = +1; return +1;
            case +1: FixRightImbalance(t); return 0;
        }
    }
}

```

```

/*
 * Function: FixLeftImbalance
 * Usage: FixLeftImbalance(t);
 * -----
 * This function is called when a node has been found that
 * is out of balance with the longer subtree on the left.
 * Depending on the balance factor of the left child, the
 * code performs a single or double rotation.
 */

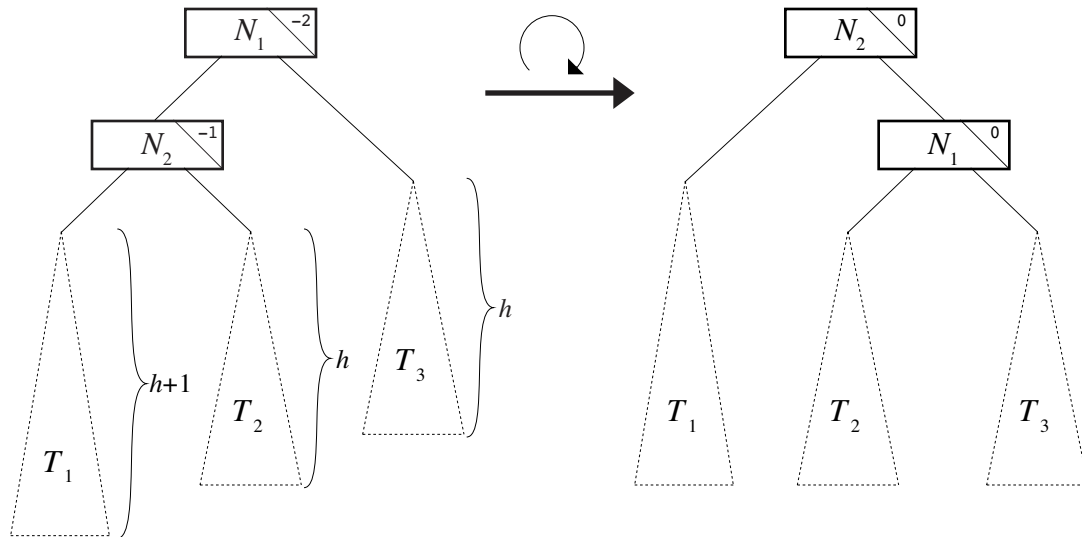
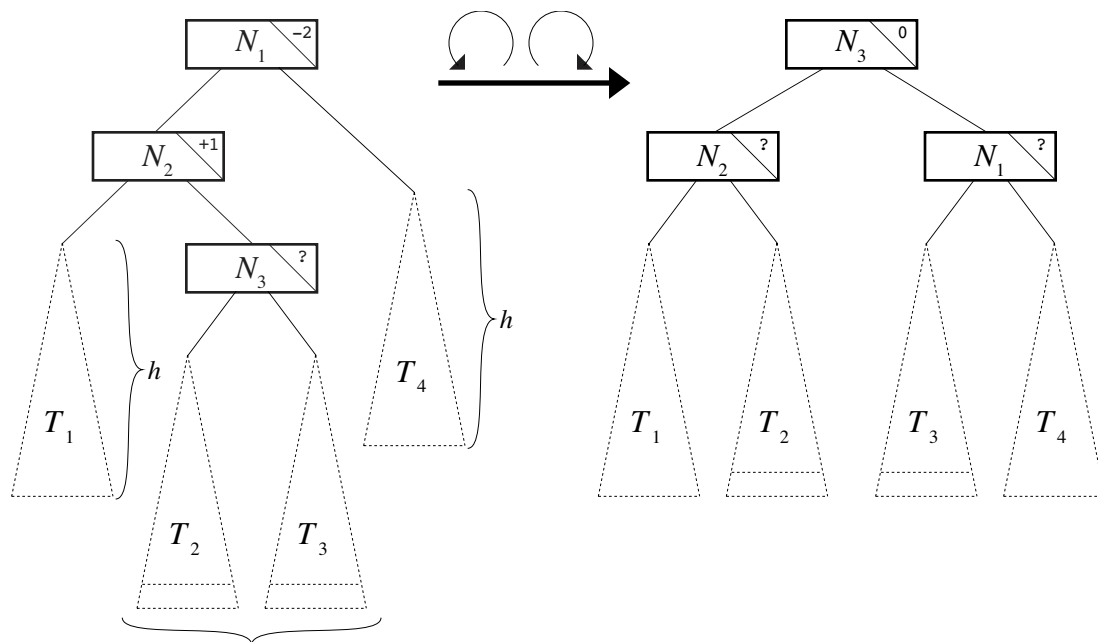
void FixLeftImbalance(nodeT * & t) {
    nodeT *child = t->left;
    if (child->bf != t->bf) {
        int oldBF = child->right->bf;
        RotateLeft(t->left);
        RotateRight(t);
        t->bf = 0;
        switch (oldBF) {
            case -1: t->left->bf = 0; t->right->bf = +1; break;
            case 0: t->left->bf = t->right->bf = 0; break;
            case +1: t->left->bf = -1; t->right->bf = 0; break;
        }
    } else {
        RotateRight(t);
        t->right->bf = t->bf = 0;
    }
}

/*
 * Function: RotateLeft
 * Usage: RotateLeft(t);
 * -----
 * This function performs a single left rotation of the tree
 * that is passed by reference. The balance factors
 * are unchanged by this function and must be corrected at a
 * higher level of the algorithm.
 */

void RotateLeft(nodeT * & t) {
    nodeT *child = t->right;
    t->right = child->left;
    child->left = t;
    t = child;
}

/* FixRightImbalance and RotateRight are defined similarly */

```

Figure 13-8 The effect of rotation operations on balance factors**Single rotation:****Double rotation:**

Unless both T_2 and T_3 are empty ($h = 0$), one will have height h and the other height $h-1$

The new balance factors in the N_1 and N_2 nodes depend on the relative heights of the subtrees T_2 and T_3

13.4 Defining a general interface for binary search trees

The code in these last several sections has given you an idea of how binary search trees work. Up to this point, however, the focus has been entirely on the implementation level. What would you do if you wanted to build an application program that used a binary search tree? As things stand, you would almost certainly have to change the definition of the `nodeT` type to include the data required by your application. Such a change would require you to edit the source code for the implementation, which violates the basic principles of interface-based design. As a client, you should never have to edit the implementation code. In general, you should not need to know the details of the implementation at all.

Since Chapter 9, this text has used the public and private sections of a class to separate the client and the implementation. If you want to make binary search trees usable as a general tool, the ideal approach is to define a `BST` class that allows clients to invoke the basic operations without having to understand the underlying detail. That interface, moreover, needs to be as general as possible to offer the client maximum flexibility. In particular, the following features would certainly make the `BST` class more useful:

- *The class should allow the client to define the structure of the data in a node.* The binary search trees you've seen so far have included no data fields except the key itself. In most cases, clients want to work with nodes that contain additional data fields as well.
- *The keys should not be limited to strings.* Although the implementations in the preceding sections have used strings as keys, there is no reason that a general package would need to impose this constraint. To maintain the proper order in a binary tree, all the implementation needs to know is how to compare two keys. As long as the client provides a comparison function, it should be possible to use any type as a key.
- *It should be possible to remove nodes as well as to insert them.* Some clients—particularly including the set package introduced in Chapter 15—need to be able to remove entries from a binary search tree. Removing a node requires some care, but is easy enough to specify in the implementation.
- *The details of any balancing algorithm should lie entirely on the implementation side of the abstraction boundary.* The interface itself should not reveal what strategy, if any, the implementation uses to keep the tree in balance. Making the process of balancing the tree private to the implementation allows you to substitute new algorithms that perform more effectively than the AVL strategy without forcing clients to change their code.

Figure 13-9 defines an interface that includes each of these features. To understand why the interface looks the way it does, it is important to consider some of the issues that arise in its design and implementation. The sections that follow review these issues.

Allowing the client to define the node data

As it stands, the code for the binary search tree algorithms introduced earlier in this chapter defines a node structure in which the key is the only data field. As a client, you almost certainly want to include other information as well. How would you go about incorporating this additional information into the structure of a binary search tree? If you think back to how the container classes introduced in the earlier chapters have enabled clients to store arbitrary types within them, the answer to this question is quite obvious. As the implementer, you define the `BST` as a class template with a template parameter for the type of the data. As a client, you define a record type that contains the information you need, and then use that type to fill in the placeholder when creating a new `BST`.

Figure 13-9 Interface for a binary search tree class template

```

/*
 * File: bst.h
 * -----
 * This file provides an interface for a general binary search
 * tree class template.
 */

#define _bst_h

#include "cmpfn.h"
#include "disallowcopy.h"

/*
 * Class: BST
 * -----
 * This interface defines a class template for a binary search tree.
 * For maximum generality, the BST is supplied as a class template.
 * The data type is set by the client. The client specializes the
 * tree to hold a specific type, e.g. BST<int> or BST<studentT>.
 * The one requirement on the type is that the client must supply a
 * a comparison function that compares two elements (or be willing
 * to use the default comparison function that relies on < and ==).
 */

template <typename ElemType>
class BST {

    public:

    /*
     * Constructor: BST
     * Usage: BST<int> bst;
     *        BST<song> songs(CompareSong)
     *        BST<string> *bp = new BST<string>;
     * -----
     * The constructor initializes a new empty binary search tree.
     * The one argument is a comparison function, which is called
     * to compare data values. This argument is optional, if not
     * given, OperatorCmp from cmpfn.h is used, which applies the
     * built-in operator < to its operands. If the behavior of <
     * on your type is defined and sufficient, you do not need to
     * supply your own comparison function.
     */

        BST(int (*cmpFn)(ElemType one, ElemType two) = OperatorCmp);

    /*
     * Destructor: ~BST
     * Usage: delete bp;
     * -----
     * This function deallocates the storage for a tree.
     */

        ~BST();

```

```

/*
 * Method: find
 * Usage:  if (bst.find(key) != NULL) . . .
 * -----
 * This method applies the binary search algorithm to find a key
 * in this tree. The argument is the key you're looking for. If
 * a node matching key appears in the tree, find returns a pointer
 * to the data in that node; otherwise, find returns NULL.
 */

ElemType *find(ElemType key);

/*
 * Method: add
 * Usage: bst.add(elem);
 * -----
 * This method adds a new node to this tree. The elem argument
 * is compared with the data in existing nodes to find the proper
 * position. If a node with the same value already exists, the
 * contents are overwritten with the new copy, and the add method
 * returns false. If no matching node is found, a new node is
 * allocated and added to the tree, and the method returns true.
 */

bool add(ElemType elem);

/*
 * Method: remove
 * Usage: bst.remove(key);
 * -----
 * This method removes a node in this tree that matches the
 * specified key. If a node matching key is found, the node
 * is removed from the tree and true is returned. If no match
 * is found, no changes are made and false is returned.
 */

bool remove(ElemType key);

/*
 * Method: mapAll
 * Usage: bst.mapAll(PrintToFile, outputStream);
 * -----
 * This method iterates through the binary search tree and
 * calls the function fn once for each element, passing the
 * element and the client's data. That data can be of whatever
 * type is needed for the client's callback. The order of calls
 * is determined by an InOrder walk of the tree.
 */

template <typename ClientElemType>
void mapAll(void (*fn)(ElemType elem, ClientElemType &data),
            ClientElemType &data);

private:
#include "bstpriv.h"
}
#include "bstimpl.cpp"

```

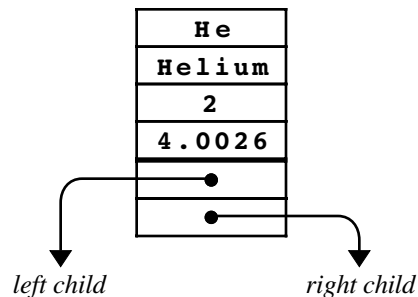
For example, let's go back to the idea of inserting the symbols for the chemical elements into a binary search tree. It is hard to imagine why anyone would want to create a tree that contained *only* the symbols for the elements, even though the symbol—being unique—makes a perfectly reasonable key. If you were writing an application that needed to know something about the elements, you would almost certainly want to store additional information. For example, in addition to the symbol for the element, you might want to store its name, atomic number, and atomic weight, which suggests the following data structure for the node:

```
struct elementT {
    string symbol;
    string name;
    int atomicNumber;
    double atomicWeight;
};
```

Thus, from the client's view, the data for the element helium would look like this:

He
Helium
2
4.0026

The client would create an object of `BST<elementT>` which indicates each node in the tree will store the client's record plus the additional node information, such the pointers to its left and right children. A node in the tree might look like this:



The heavy line in the diagram divides the client data from the implementation data. Everything above that line belongs to the client. Everything below it—which might also include other fields necessary to keep the tree in balance—belongs to the implementation.

Generalizing the types used for keys

Allowing the client to use keys of any type is not particularly difficult. For the most part, the basic strategy is to have the client supply a comparison function. By storing a pointer to the client's comparison function as a data member for the tree as a whole, the implementation can invoke that function whenever it needs to compare keys. The comparison function is passed as an argument when constructing a new BST object. For each new data type to be stored in a BST, the client will have to supply the appropriate comparison function. For example, the `BST<elementT>` described above would require a function for comparing two `elementT` structs. The key field within an element is its symbol, so the `CompareElements` function below returns the result of comparing the symbol fields within the `elementT` structures.


```

int CompareElements(elementT one, elementT two) {
    if (one.symbol == two.symbol) return 0;
    return (one.symbol < two.symbol) ? -1 : 1;
};

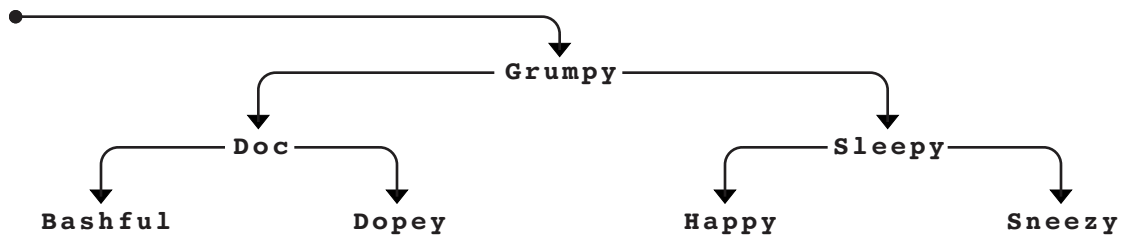
```

The comparison function argument to the constructor is optional. If not supplied, the default comparison function is used. The default comparison function was introduced for the general sorting routine in Chapter 12, it simply compares two values using the built-in operator `<`. If the BST is storing types such as `int` or `string` that can be compared using `<`, this default function can be used.

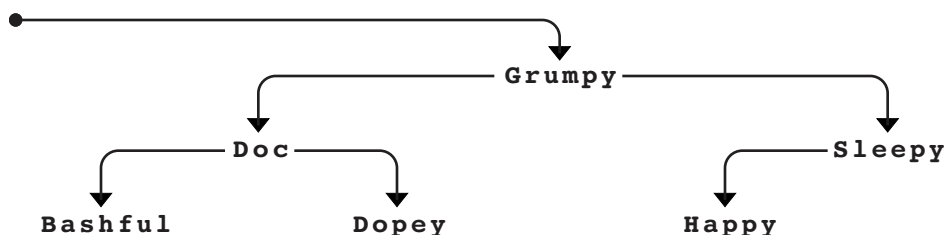
Removing nodes

The operation of removing a node from a binary search tree is not hard to define in the **BST** class interface. The interesting issues that arise in removing a node are primarily the concern of the implementation. Finding the node to be removed requires the same binary-search strategy as finding a node. Once you find the appropriate node, however, you have to remove it from the tree without violating the ordering relationship that defines a binary search tree. Depending on where the node to be removed appears in the tree, removing it can get rather tricky.

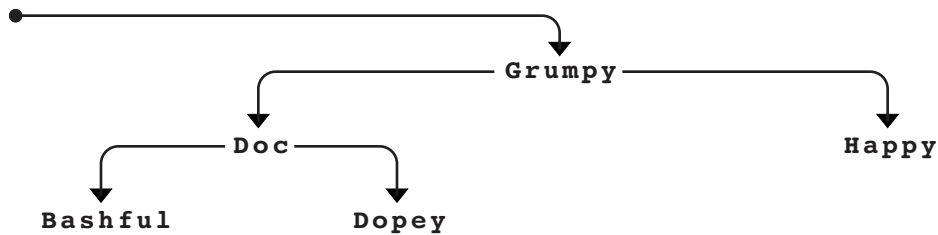
To get a sense of the problem, suppose that you are working with a binary search tree whose nodes have the following structure:



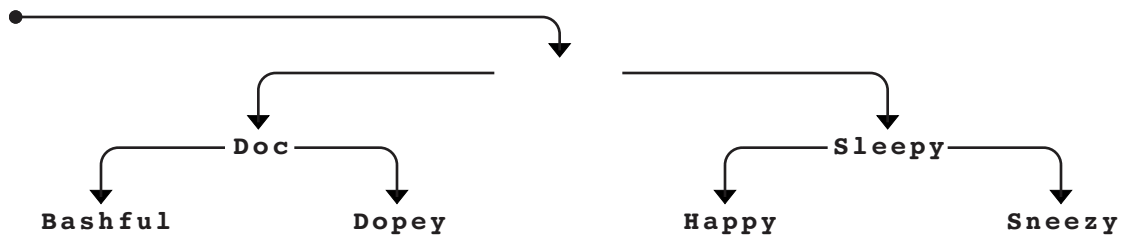
Removing **Sneezzy** (for creating an unhealthy work environment) is easy. All you have to do is replace the pointer to the **Sneezzy** node with a **NULL** pointer, which produces the following tree:



Starting from this configuration, it is also relatively easy to remove **Sleepy** (who has trouble staying awake on the job). If either child of the node you want to remove is **NULL**, all you have to do is replace it with its non-**NULL** child, like this:

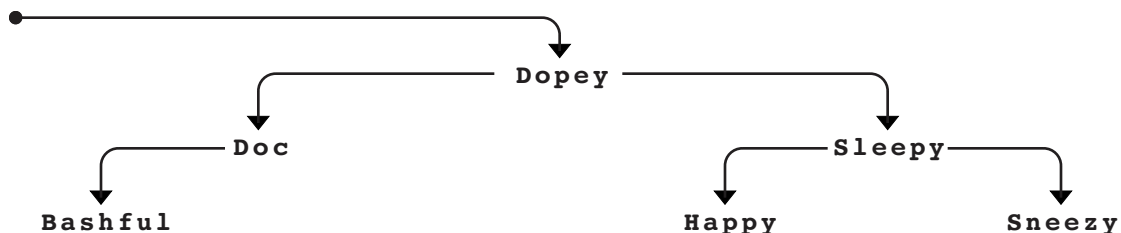


The problem arises if you try to remove a node with both a left and a right child. Suppose, for example, that you instead want to remove **Grumpy** (for failure to whistle while working) from the original tree containing all seven dwarves. If you simply remove the **Grumpy** node, you're left with two partial search trees, one rooted at **Doc** and one rooted at **Sleepy**, as follows:



How can you patch things back together so that you have a valid binary search tree?

At this point, what you would like to do is find a node that can be inserted into the empty space left behind by the removal of the **Grumpy** node. To ensure that the resulting tree remains a binary search tree, there are only two nodes you can use: the rightmost node in the left subtree or the leftmost node in the right subtree. These two nodes work equally well, and you can write the removal algorithm using either one. For example, if you choose the rightmost node in the left subtree, you get the **Dopey** node, which is guaranteed to be larger than anything else in the left subtree but smaller than the values in the right subtree. To complete the removal, all you have to do is replace the **Dopey** node with its left child—which may be **NULL**, as it is in this example—and then move the **Dopey** node into the deleted spot. The resulting picture looks like this:



Implementing the binary search tree package

The code for the generalized binary search tree package, including the details of the removal algorithm described in the preceding section, appears in Figure 13-10 and 13-11. This implementation does not include the AVL extension that keeps the nodes balanced. Allowing the tree to become out of balance increases the running time of the algorithm but does not compromise its correctness. You will have a chance to implement the self-balancing features in the exercises.

Figure 13-10 Private data for the BST class

```

/*
 * File: bstpriv.h
 * -----
 * This file contains the private section for the BST class.
 */

/* Type for tree node */
struct nodeT {
    ElemType data;
    nodeT *left, *right;
};

/* Instance variables */

nodeT *root;                /* Pointer to root node */
int (*cmpFn)(ElemType, ElemType); /* Comparison function */

/* Private function prototypes */

nodeT *recFindNode(nodeT *t, ElemType key);
bool recAddNode(nodeT * & t, ElemType key);
bool recRemoveNode(nodeT * & t, ElemType key);
void removeTargetNode(nodeT * & t);
void recFreeTree(nodeT *t);

template <typename ClientType>
void recMapAll(nodeT *t, void (*fn)(ElemType, ClientType &),
               ClientType &data);

```

Figure 13-11 Implementation of the BST class

```

/*
 * File: bstimpl.cpp
 * -----
 * This file implements the bst.h interface, which provides a
 * general implementation of binary search trees.
 */

/*
 * Implementation notes for the constructor
 * -----
 * The constructor sets root to NULL to indicate an empty tree.
 * It also stores the client-supplied comparison function for
 * later use; this value defaults to OperatorCmp if no value
 * is supplied.
 */

template <typename ElemType>
BST<ElemType>::BST(int (*cmpFn)(ElemType, ElemType)) {
    root = NULL;
    this->cmpFn = cmpFn;
}

```

```

/*
 * Implementation notes for the destructor
 * -----
 * The destructor must delete all the nodes in the tree. To
 * do so, it calls recFreeNode, which does a recursive postorder
 * walk on the tree, deleting all nodes in the subtrees before
 * deleting the current node.
 */

template <typename ElemType>
BST<ElemType>::~~BST() {
    recFreeTree(root);
}

template <typename ElemType>
void BST<ElemType>::recFreeTree(nodeT *t) {
    if (t != NULL) {
        recFreeTree(t->left);
        recFreeTree(t->right);
        delete t;
    }
}

/*
 * Implementation notes: find, recFindNode
 * -----
 * The find function simply calls recFindNode to do the work. The
 * recursive function takes the current node along with the original
 * argument. If found, it returns a pointer to the matching data.
 */

template <typename ElemType>
ElemType *BST<ElemType>::find(ElemType key) {
    nodeT *found = recFindNode(root, key);
    if (found != NULL) {
        return &found->data;
    }
    return NULL;
}

template <typename ElemType>
typename BST<ElemType>::nodeT *
    BST<ElemType>::recFindNode(nodeT *t, ElemType key) {
    if (t == NULL) return NULL;
    int sign = cmpFn(key, t->data);
    if (sign == 0) return t;
    if (sign < 0) {
        return recFindNode(t->left, key);
    } else {
        return recFindNode(t->right, key);
    }
}

```

```

/*
 * Implementation notes: add, recAddNode
 * -----
 * The add function is implemented as a simple wrapper to recAddNode,
 * which does all the work. The recAddNode function takes an extra
 * argument, which is a reference to the root of the current subtree.
 */

template <typename ElemType>
bool BST<ElemType>::add(ElemType data) {
    return recAddNode(root, data);
}

template <typename ElemType>
bool BST<ElemType>::recAddNode(nodeT * & t, ElemType data) {
    if (t == NULL) {
        t = new nodeT;
        t->data = data;
        t->left = t->right = NULL;
        return true;
    }
    int sign = cmpFn(data, t->data);
    if (sign == 0) {
        t->data = data;
        return false;
    } else if (sign < 0) {
        return recAddNode(t->left, data);
    } else {
        return recAddNode(t->right, data);
    }
}

/*
 * Implementation notes: remove, recRemoveNode
 * -----
 * The first step in removing a node is to find it using binary
 * search, which is performed by these two functions. If the
 * node is found, removeTargetNode does the actual deletion.
 */

template <typename ElemType>
bool BST<ElemType>::remove(ElemType data) {
    return recRemoveNode(root, data);
}

template <typename ElemType>
bool BST<ElemType>::recRemoveNode(nodeT * & t, ElemType data) {
    if (t == NULL) return false;
    int sign = cmpFn(data, t->data);
    if (sign == 0) {
        removeTargetNode(t);
        return true;
    } else if (sign < 0) {
        return recRemoveNode(t->left, data);
    } else {
        return recRemoveNode(t->right, data);
    }
}

```

```

/*
 * Implementation notes: removeTargetNode
 * -----
 * This function removes the node which is passed by reference as t.
 * The easy case occurs when either of the children is NULL: all
 * you need to do is replace the node with its non-NULL child.
 * If both children are non-NULL, this code finds the rightmost
 * descendent of the left child; this node may not be a leaf, but
 * will have no right child. Its left child replaces it in the
 * tree, after which the replacement node is moved to the position
 * occupied by the target node.
 */

template <typename ElemType>
void BST<ElemType>::removeTargetNode(nodeT * & t) {
    nodeT *toDelete = t;
    if (t->left == NULL) {
        t = t->right;
    } else if (t->right == NULL) {
        t = t->left;
    } else {
        nodeT *newRoot = t->left;
        nodeT *parent = t;
        while (newRoot->right != NULL) {
            parent = newRoot;
            newRoot = newRoot->right;
        }
        if (parent != t) {
            parent->right = newRoot->left;
            newRoot->left = t->left;
        }
        newRoot->right = t->right;
        t = newRoot;
    }
    delete toDelete;
}

/* Implementation of the mapping functions */

template <typename ElemType>
template <typename ClientType>
void BST<ElemType>::mapAll(void (*fn)(ElemType, ClientType &),
                          ClientType &data) {
    recMapAll(root, fn, data);
}

template <typename ElemType>
template <typename ClientType>
void BST<ElemType>::recMapAll(nodeT *t,
                             void (*fn)(ElemType, ClientType &),
                             ClientType &data) {
    if (t != NULL) {
        recMapAll(t->left, fn, data);
        fn(t->data, data);
        recMapAll(t->right, fn, data);
    }
}

```

Implementing the `map.h` interface using binary trees

Once you have defined and implemented a class template for binary search trees, you can use the class as part of other applications. For example, you can easily use the **BST** class to reimplement the **Map** class from Chapter 12. If you think about the problem in terms of what the map needs—as opposed to the details of the binary search tree itself—an entry consists of a key and a value. You would define the type **pairT** as this pair and change the data members of the **Map** class to a **BST** containing such pairs. Here is the private section of the **Map** class with these changes:

```
private:

    struct pairT {
        string key;
        ValueType value;
    };

    BST<pairT> bst;

    /* private helper function to compare two pairTs */

    static int ComparePairByKey(pairT one, pairT two);
```

When you have access to the **BST** class template, writing the code to implement the **Map** constructor, **put**, and **get** methods becomes a simple task. The code for these functions appears in Figure 13-12.

Using the `static` keyword

If you examine the implementation of the **Map** class template closely, you will note that the declaration of the **ComparePairByKey** method in the class interface is marked with

Figure 13-12 Implementation of maps using binary search trees

```
/*
 * File: bstmap.cpp
 * -----
 * This file implements the Map class layered on top of BSTs.
 */

template <typename ValueType>
Map<ValueType>::Map() : bst(ComparePairByKey) {
    /* Empty */
}

template <typename ValueType>
Map<ValueType>::~~Map() {
    /* Empty */
}

template <typename ValueType>
void Map<ValueType>::put(string key, ValueType value) {
    pairT pair;
    pair.key = key;
    pair.value = value;
    bst.add(pair);
}
```

```

template <typename ValueType>
ValueType Map<ValueType>::get(string key) {
    pairT pair, *found;
    pair.key = key;
    if ((found = bst.find(pair)) != NULL) {
        return found->value;
    }
    Error("getValue called on non-existent key");
}

/*
 * This static function is used to compare two pairs. It
 * ignores the value fields and just returns the ordering
 * of the two key fields. This function must be declared static
 * so that is not a method of the class and thus
 * is not expected to be invoked on a receiver object.
 */

template <typename ValueType>
int Map<ValueType>::ComparePairByKey(pairT one, pairT two) {
    if (one.key == two.key) return 0;
    return (one.key < two.key) ? -1 : 1;
}

```

the keyword **static**. Within a class interface, the **static** keyword is used to identify members that are shared across the class, and not specific to a particular object or instance. You have previously used the **static** keyword when declaring class constants.

The distinction between static and non-static functions of a class is subtle, but important. By default, a method is assumed to operate on a particular object. When you call such a method, you identify which object is being acted upon by specifying the receiver in the call, as in

```
obj.performAction()
```

By declaring **ComparePairByKey** as **static**, the function is modified so that it becomes associated with the class itself and is not be invoked on a specific receiver object. This function must be declared **static** in order to be compatible with the type of comparison function used by the **BST** class. The **BST** class assumes the comparison callback has the form of a free function that takes two arguments, the two values being compared, and does not invoke the callback on a receiver object.

Any **static** methods you declare are still considered part of the class implementation, and thus have access to the private internals, such as the definition of the **Map** class **pairT** type. However, in the body of a **static** method, there is no reference to **this** (because there is no receiving object) and thus no access to data members.

Summary

In this chapter, you have been introduced to the concept of *trees*, which are hierarchical collections of nodes that obey the following properties:

- There is a single node at the top that forms the root of the hierarchy.
- Every node in the tree is connected to the root by a unique line of descent.

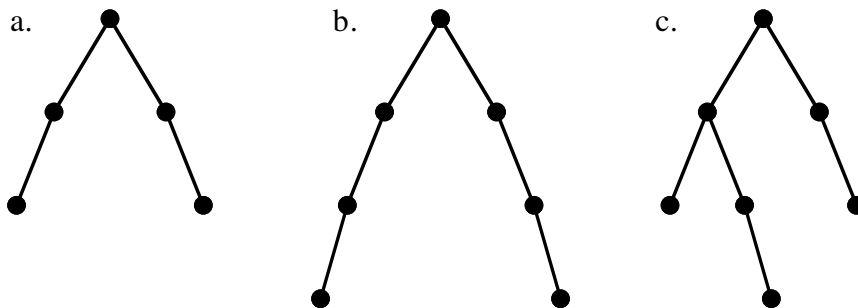
Important points in this chapter include:

- Many of the terms used to describe trees, such as *parent*, *child*, *ancestor*, *descendant*, and *sibling*, come directly from family trees. Other terms, including *root* and *leaf*, are derived from trees in nature. These metaphors make the terminology used for trees easy to understand because the words have the same interpretation in computer science as they do in these more familiar contexts.
- Trees have a well-defined recursive structure because every node in a tree is the root of a subtree. Thus, a tree consists of a node together with its set of children, each of which is a tree. This recursive structure is reflected in the underlying representation for trees, which are defined as a pointer to a **nodeT**, and the type **nodeT** is defined as a record containing values of type pointer to a **nodeT**.
- Binary trees are a subclass of trees in which nodes have at most two children and every node except the root is designated as either a left child or a right child of its parent.
- If a binary tree is organized so that every node in the tree contains a key field that follows all the keys in its left subtree and precedes all the keys in its right subtree, that tree is called a *binary search tree*. As its name implies, the structure of a binary search tree permits the use of the binary search algorithm, which makes it possible to find individual keys more efficiently. Because the keys are ordered, it is always possible to determine whether the key you're searching for appears in the left or right subtree of any particular node.
- Using recursion makes it easy to step through the nodes in a binary search tree, which is called *traversing* or *walking* the tree. There are several types of traversals, depending on the order in which the nodes are processed. If the key in each node is processed before the recursive calls to process the subtrees, the result is a *preorder* traversal. Processing each node after both recursive calls gives rise to a *postorder* traversal. Processing the current node between the two recursive calls represents an *inorder* traversal. In a binary search tree, the inorder traversal has the useful property that the keys are processed in order.
- Depending on the order in which nodes are inserted, given the same set of keys, binary search trees can have radically different structures. If the branches of the tree differ substantially in height, the tree is said to be unbalanced, which reduces its efficiency. By using techniques such as the AVL algorithm described in this chapter, you can keep a tree in balance as new nodes are added.
- It is possible to design an interface for binary search trees that allows the client to control the data of the individual nodes by using C++ templates. The **BST** class template that appears in Figure 13-9 exports a flexible implementation of the binary search tree structure that can be used in a wide variety of applications.

Review questions

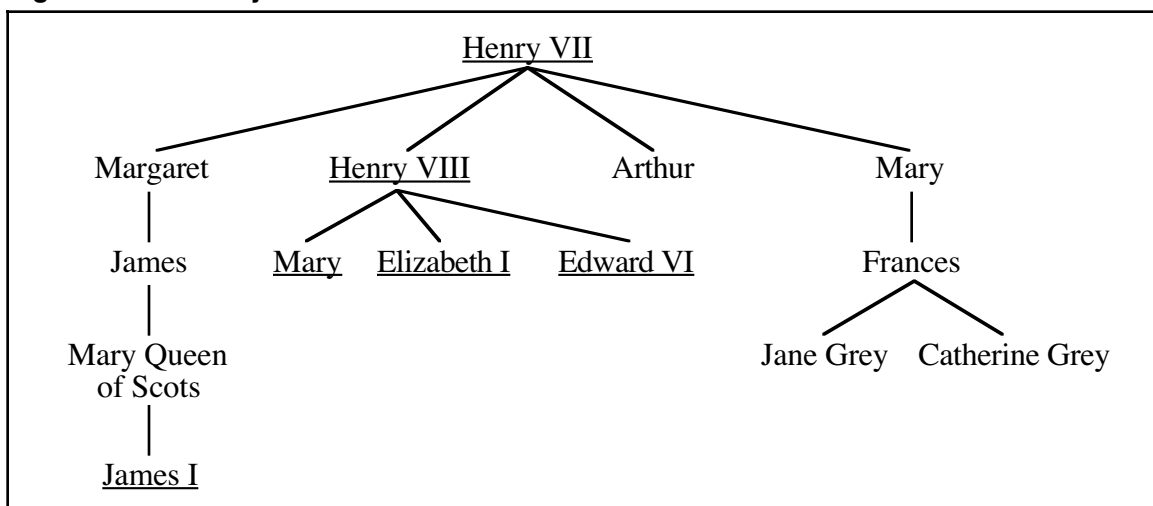
1. What two conditions must be satisfied for a collection of nodes to be a tree?
2. Give at least four real-world examples that involve tree structures.
3. Define the terms *parent*, *child*, *ancestor*, *descendant*, and *sibling* as they apply to trees.
4. The family tree for the House of Tudor, which ruled England in Shakespeare's time, is shown in Figure 13-13. Identify the root, leaf, and interior nodes. What is the height of this tree?
5. What is it about trees that makes them recursive?

6. Diagram the internal structure of the tree shown in Figure 13-13 when it is represented as a **familyNodeT**.
7. What is the defining property of a binary search tree?
8. Why are different type declarations used for the first argument in **FindNode** and **InsertNode**?
9. In *The Hobbit* by J. R. R. Tolkien, 13 dwarves arrive at the house of Bilbo Baggins in the following order: **Dwalin**, **Balin**, **Kili**, **Fili**, **Dori**, **Nori**, **Ori**, **Oin**, **Gloin**, **Bifur**, **Bofur**, **Bombur**, and **Thorin**. Diagram the binary search tree that results from inserting the names of these dwarves into an empty tree.
10. Given the tree you created in the preceding question, what key comparisons are made if you call **FindNode** on the name **Bombur**?
11. Write down the preorder, inorder, and postorder traversals of the binary search tree you created for question 9.
12. One of the three standard traversal orders—preorder, inorder, or postorder—does not depend on the order in which the nodes are inserted into the tree. Which one is it?
13. What does it mean for a binary tree to be balanced?
14. For each of the following tree structures, indicate whether the tree is balanced:

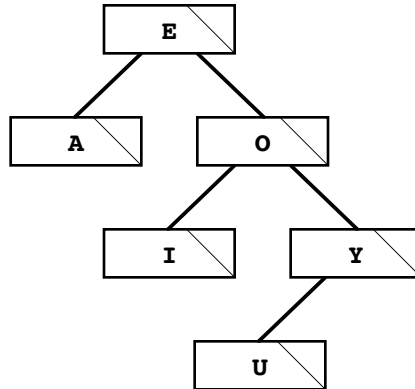


For any tree structure that is out of balance, indicate which nodes are out of balance.

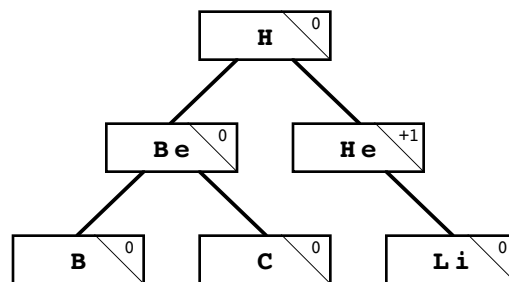
Figure 13-13 Family tree for the House of Tudor



15. True or false: If a binary search tree becomes unbalanced, the algorithms used in the functions **FindNode** and **InsertNode** will fail to work correctly.
16. How do you calculate the balance factor of a node?
17. Fill in the balance factors for each node in the following binary search tree:



18. If you use the AVL balancing strategy, what rotation operation must you apply to the tree in the preceding question to restore its balanced configuration? What is the structure of the resulting tree, including the updated balance factors?
19. True or false: When you insert a new node into a balanced binary tree, you can always correct any resulting imbalance by performing one operation, which will be either a single or a double rotation.
20. As shown in the section on “Illustrating the AVL idea,” inserting the symbols for the first six elements into an AVL tree results in the following configuration:



Show what happens to the tree as you add the next six element symbols:

N (Nitrogen)
O (Oxygen)
F (Fluorine)
Ne (Neon)
Na (Sodium)
Mg (Magnesium)

21. Describe in detail what happens when the **add** method is called.
22. What strategy does the text suggest to avoid having a binary search tree become disconnected if you remove an interior node?

Programming exercises

1. Working from the definition of **familyNodeT** given in the section entitled “Representing family trees in C++,” write a function

```
familyNodeT *ReadFamilyTree(string filename);
```

that reads in a family tree from a data file whose name is supplied as the argument to the call. The first line of the file should contain a name corresponding to the root of the tree. All subsequent lines in the data file should have the following form:

child:parent

where *child* is the name of the new individual being entered and *parent* is the name of that child’s parent, which must appear earlier in the data file. For example, if the file **normandy.dat** contains the lines

```
William I
Robert:William I
William II:William I
Adela:William I
Henry I:William I
Stephan:Adela
William:Henry I
Matilda:Henry I
Henry II:Matilda
```

calling **ReadFamilyTree("normandy.dat")** should return the family-tree structure shown in Figure 13-2.

2. Write a function

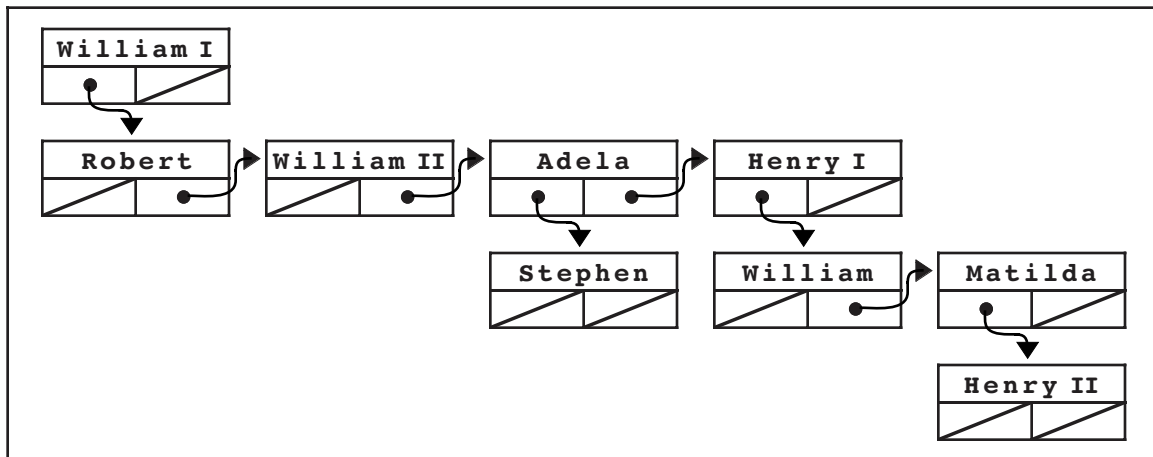
```
void DisplayFamilyTree(familyNodeT *tree);
```

that displays all the individuals in a family tree. To record the hierarchy of the tree, the output of your program should indent each generation so that the name of each child appears two spaces to the right of the corresponding parent, as shown in the following sample run:



3. As defined in the chapter, the **familyNodeT** structure uses a vector to store the children. Another possibility is to include an extra pointer in these nodes that will allow them to form a linked list of the children. Thus, in this design, each node in the tree needs to contain only two pointers: one to its eldest child and one to its next younger sibling. Using this representation, the House of Normandy appears as shown in Figure 13-14. In each node, the pointer on the left always points down to a child; the pointer on the right indicates the next sibling in the same generation. Thus, the eldest child of William I is Robert, which you obtain by following the link at the

Figure 13-14 House of Normandy using a linked list of siblings



left of the diagram. The remaining children are linked together through the link cells shown at the right of the node diagram. The chain of children ends at Henry I, which has the value **NULL** in its next-sibling link.

Using the linked design illustrated in this diagram, write new definitions of **familyNodeT**, **ReadFamilyTree**, and **DisplayFamilyTree**.

- In exercise 3, the changes you made to **familyNodeT** forced you to rewrite the functions—specifically **ReadFamilyTree** and **DisplayFamilyTree**—that depend on that representation. If the family tree were instead represented as a class that maintained its interface despite any changes in representation, you could avoid much of this recoding. Such an interface appears in Figure 13-15. Write the corresponding implementation using a vector to store the list of children.

Note that the class exported by the **famtree.h** interface corresponds to an individual node rather than to the tree as a whole. From each node, you can find the parent using **getParent** and the children using **getChildren**.

- Using the **famtree.h** interface defined in the preceding exercise, write a function

```
FamilyTreeNode *FindCommonAncestor(FamilyTreeNode *p1,
                                   FamilyTreeNode *p2);
```

that returns the closest ancestor shared by **p1** and **p2**.

- Write a function

```
int Height(nodeT *tree);
```

that takes a binary search tree—using the definition of **nodeT** from section 13.2—and returns its height.

- Write a function

```
bool IsBalanced(nodeT *tree);
```

that determines whether a given tree is balanced according to the definition in the section on “Balanced trees.” To solve this problem, all you really need to do is translate the definition of a balanced tree more or less directly into code. If you do so, however, the resulting implementation is likely to be relatively inefficient because it has to make several passes over the tree. The real challenge in this

Figure 13-15 Interface for a class that supports the representation of family trees

```

class FamilyTreeNode {
public:
    /*
     * Constructor: FamilyTreeNode
     * Usage: FamilyTreeNode *person = new FamilyTreeNode(name);
     * -----
     * This function constructs a new FamilyTreeNode with the specified
     * name. The newly constructed entry has no children, but clients
     * can add children by calling the addChild method.
     */
    FamilyTreeNode(string name);

    /*
     * Method: getName
     * Usage: string name = person->getName();
     * -----
     * Returns the name of the person.
     */
    string getName();

    /*
     * Method: addChild
     * Usage: person->addChild(child);
     * -----
     * Adds child to the end of the list of children for person, and
     * makes person the parent of child.
     */
    void addChild(FamilyTreeNode *child);

    /*
     * Method: getParent
     * Usage: FamilyTreeNode *parent = person->getParent();
     * -----
     * Returns the parent of the specified person.
     */
    FamilyTreeNode *getParent();

    /*
     * Method: getChildren
     * Usage: Vector<FamilyTreeNode *> children = person->getChildren();
     * -----
     * Returns a vector of the children of the specified person.
     * Note that this vector is a copy of the one in the node, so
     * that the client cannot change the tree by adding or removing
     * children from this vector.
     */
    Vector<FamilyTreeNode *> getChildren();

    /* Whatever private section you need */
}

```

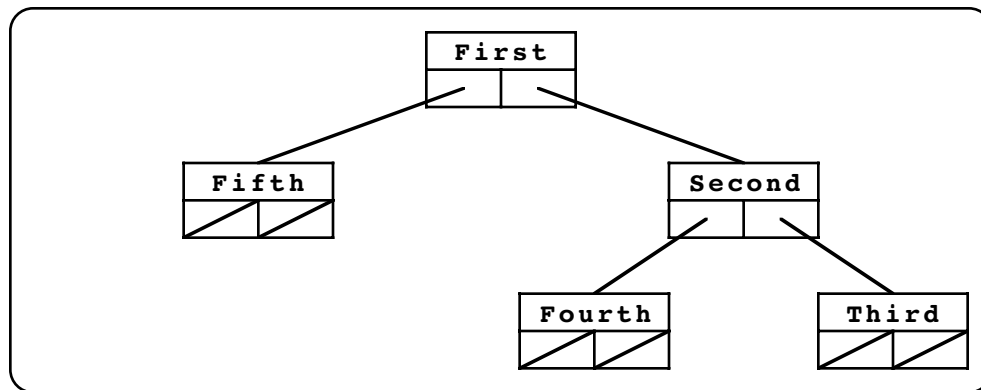
problem is to implement the **IsBalanced** function so that it determines the result without looking at any node more than once.

8. Write a function

```
bool HasBinarySearchProperty(nodeT *tree);
```

that takes a tree and determines whether it maintains the fundamental property that defines a binary search tree: that the key in each node follows every key in its left subtree and precedes every key in its right subtree.

9. Write a test program for the **BST** class that uses the graphics library described in section 6.3 to display the structure of the tree. For example, if you insert the keys **First**, **Second**, **Third**, **Fourth**, and **Fifth** into a binary search tree without balancing, your program should display the following diagram in the graphics window:



Including the keys as part of the node diagram will require you to use the extended version of the graphics library interface, **extgraph.h**, which is available for many systems as part of the Addison-Wesley software archive. Even without it, you can construct the line drawing for the nodes from the simple commands available in the simpler **graphics.h** interface.

10. Extend the implementation of the **BST** class template so that it uses the AVL algorithm to keep the tree balanced as new nodes are inserted. The algorithm for balanced insertion is coded in Figure 13-7. Your task in this problem is simply to integrate this algorithm into the more general implementation of binary search trees given in Figure 13-11.
11. Integrating the AVL algorithm for inserting a node into the **bst.cpp** implementation only solves part of the balancing problem for the generalized **BST** class. Because the **BST** class interface also exports a function to remove a node from the tree, the complete implementation of the package must also rebalance the tree when a node is removed. The structure of the algorithm to rebalance after removal is quite similar to that for insertion. Removing a node either may have no effect on the height of a tree or may shorten it by one. If a tree gets shorter, the balance factor in its parent node changes. If the parent node becomes out of balance, it is possible to rebalance the tree at that point by performing either a single or a double rotation.

Revise the implementation of the **remove** method so that it keeps the underlying AVL tree balanced. Think carefully about the various cases that can arise and make sure that your implementation handles each of these cases correctly.

12. From a practical standpoint, the AVL algorithm is too aggressive. Because it requires that the heights of the subtrees at each node never differ by more than one, the AVL algorithm spends quite a bit of time performing rotation operations to correct imbalances that occur as new nodes are inserted. If you allow trees to become somewhat more unbalanced—but still keep the subtrees relatively similar—you can reduce the balancing overhead significantly.

One of the most popular techniques for managing binary search trees is called *red-black trees*. The name comes from the fact that every node in the tree is assigned a color, either red or black. A binary search tree is a legal red-black tree if all three of the following properties hold:

1. The root node is black.
2. The parent of every red node is black.
3. Every path from the root to a leaf contains the same number of black nodes.

These properties ensure that the longest path from the root to a leaf can never be more than twice the length of the shortest path. Given the rules, you know that every such path has the same number of black nodes, which means that the shortest possible path is composed entirely of black nodes, and the longest has black and red nodes alternating down the chain. Although this condition is less strict than the definition of a balanced tree used in the AVL algorithm, it is sufficient to guarantee that the operations of finding and inserting new nodes both run in logarithmic time.

The key to making red-black trees work is finding an insertion algorithm that allows you to add new nodes while maintaining the conditions that define red-black trees. The algorithm has much in common with the AVL algorithm and uses the same rotation operations. The first step is to insert the new node using the standard insertion algorithm with no balancing. The new node always replaces a **NULL** entry at some point in the tree. If the node is the first node entered into the tree, it becomes the root and is therefore colored black. In all other cases, the new node must initially be colored red to avoid violating the rule that every path from the root to a leaf must contain the same number of black nodes.

As long as the parent of the new node is black, the tree as a whole remains a legal red-black tree. The problem arises if the parent node is also red, which means that the tree violates the second condition, which requires that every red node have a black parent. In this case, you need to restructure the tree to restore the red-black condition. Depending on the relationship of the red-red pair to the remaining nodes in the tree, you can eliminate the problem by performing one of the following operations:

1. A single rotation, coupled with a recoloring that leaves the top node black.
2. A double rotation, coupled with a recoloring that leaves the top node black.
3. A simple change in node colors that leaves the top node red and may therefore require further restructuring at a higher level in the tree.

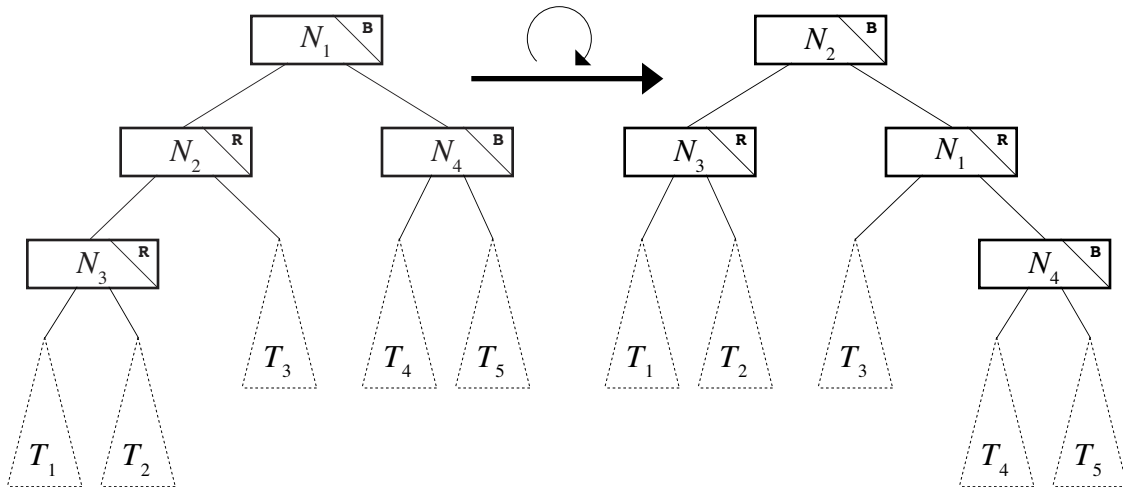
These three operations are illustrated in Figure 13-16. The diagram shows only the cases in which the imbalance occurs on the left side. Imbalances on the right side are treated symmetrically.

Change the implementation of the **BST** class template in Figures 13-10 and 13-11 so that it uses red-black trees to maintain balance.

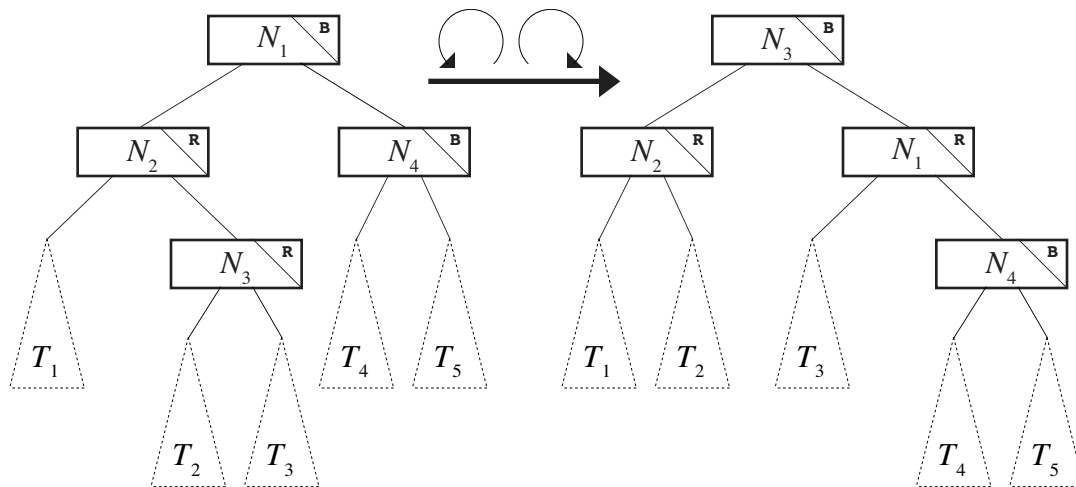
13. Complete the BST-based implementation of the **Map** class, which appears in a partial form in Figure 13-12. The missing methods are **mapAll** and **remove**.

Figure 13-16 Balancing operations on red-black trees

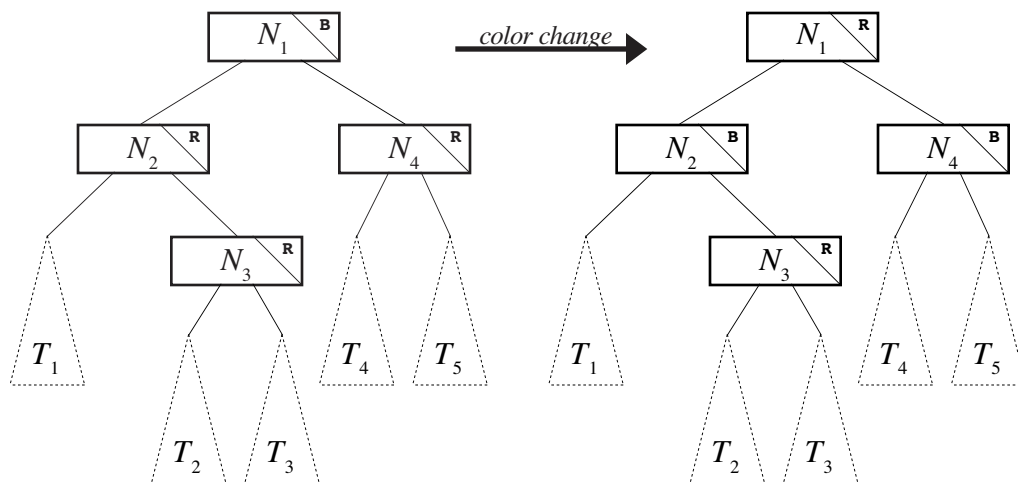
Case 1: N_4 is black (or nonexistent); N_1 and N_2 are out of balance in the same direction



Case 2: N_4 is black (or nonexistent); N_1 and N_2 are out of balance in opposite directions



Case 3: N_4 is red; the relative balance of N_1 and N_2 does not matter



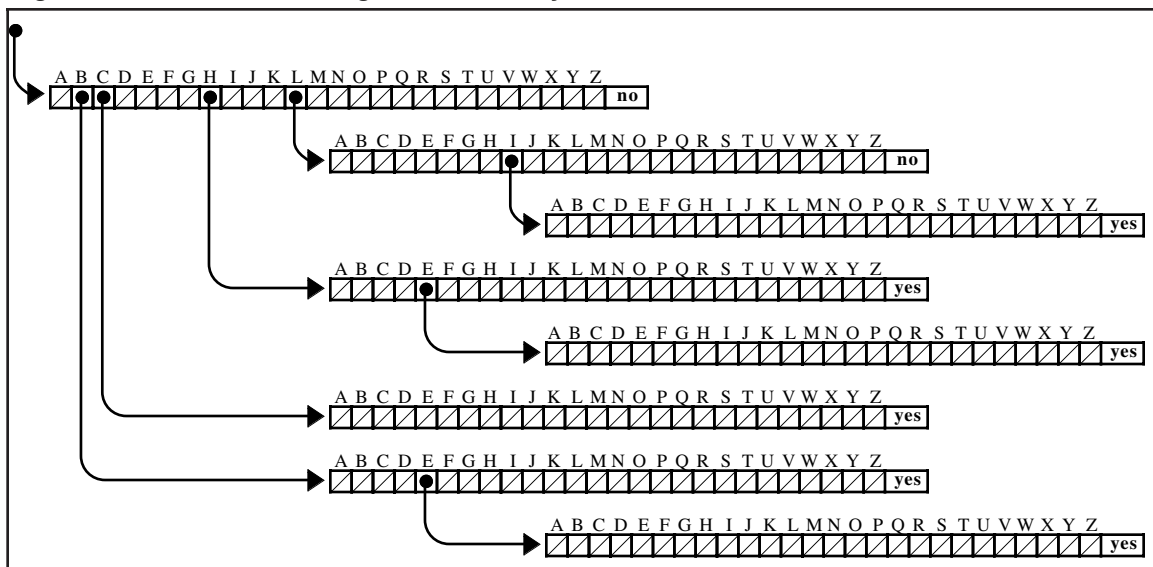
14. Trees have many applications beyond those listed in this chapter. For example, trees can be used to implement a lexicon, which was introduced in Chapter 4. The resulting structure, first developed by Edward Fredkin in 1960, is called a **trie**. (Over time, the pronunciation of this word has evolved to the point that it is now pronounced like *try*, even though the name comes from the central letters of *retrieval*.) The trie-based implementation of a lexicon, while somewhat inefficient in its use of space, makes it possible to determine whether a word is in the lexicon much more quickly than you can using a hash table.

At one level, a trie is simply a tree in which each node branches in as many as 26 ways, one for each possible letter of the alphabet. When you use a trie to represent a lexicon, the words are stored implicitly in the structure of the tree and represented as a succession of links moving downward from the root. The root of the tree corresponds to the empty string, and each successive level of the tree corresponds to the subset of the entire word list formed by adding one more letter to the string represented by its parent. For example, the *A* link descending from the root leads to the subtree containing all the words beginning with *A*, the *B* link from that node leads to the subtree containing all the words beginning with *AB*, and so forth. Each node is also marked with a flag indicating whether the substring that ends at that particular point is a legitimate word.

The structure of a trie is much easier to understand by example than by definition. Figure 13-17 shows a trie containing the symbols for the first six elements—**H**, **He**, **Li**, **Be**, **B**, and **C**. The root of the tree corresponds to the empty string, which is not a legal symbol, as indicated by the designation **no** in the field at the extreme right end of the structure. The link labeled **B** from the node at the root of the trie descends to a node corresponding to the string "**B**". The rightmost field of this node contains **yes**, which indicates that the string "**B**" is a complete symbol in its own right. From this node, the link labeled **E** leads to a new node, which indicates that the string "**BE**" is a legal symbol as well. The **NULL** pointers in the trie indicate that no legal symbols appear in the subtree beginning with that substring and therefore make it possible to terminate the search process.

Remplement the **Lexicon** class that uses a trie as its internal representation. Your implementation should be able to read text files but not the binary ones.

Figure 13-17 Trie containing the element symbols **H**, **He**, **Li**, **Be**, **B**, and **C**



Chapter 14

Expression Trees

*“What’s twice eleven?” I said to Pooh.
 (“Twice what?” said Pooh to Me.)
 “I think it ought to be twenty-two.”
 “Just what I think myself,” said Pooh.*

— A. A. Milne, “Us Two,” *Now We Are Six*, 1927

Chapter 13 focused on binary search trees because they provide a simple context for explaining how trees work. Trees occur in many other programming contexts as well. In particular, trees often show up in the implementation of compilers because they are ideal for representing the hierarchical structure of a program. By exploring this topic in some detail, you will learn quite a bit, not only about trees, but also about the compilation process itself. Understanding how compilers work removes some of the mystery surrounding programming and makes it easier to understand the process as a whole.

Unfortunately, designing a complete compiler is far too complex to serve as a useful illustration. Typical commercial compilers require many person-years of programming, much of which is beyond the scope of this text. Even so, it is possible to give you a sense of how they work—and, in particular, of how trees fit into the process—by making the following simplifications:

- *Having you build an interpreter instead of a compiler.* As described in the section on “What is C++?” in Chapter 1, a compiler translates a program into machine-language instructions that the computer can then execute directly. Although it has much in common with a compiler, an **interpreter** never actually translates the source code into machine language but simply performs the operations necessary to achieve the effect of the compiled program. Interpreters are generally easier to write, but have the disadvantage that interpreted programs tend to run much more slowly than their compiled counterparts.
- *Focusing only on the problem of evaluating arithmetic expressions.* A full-scale language translator for a modern programming language—whether a compiler or an interpreter—must be able to process control statements, function calls, type definitions, and many other language constructs. Most of the fundamental techniques used in language translation, however, are illustrated in the seemingly simple task of translating arithmetic expressions. For the purpose of this chapter, arithmetic expressions will be limited to constants and variables combined using the operators `+`, `-`, `*`, `/`, and `=` (assignment). As in C++, parentheses may be used to define the order of operations, which is otherwise determined by applying precedence rules.
- *Limiting the types used in expressions to integers.* Modern programming languages like C++ allow expressions to manipulate data of many different types. In this chapter, all data values are assumed to be of type `int`, which simplifies the structure of the interpreter considerably.

14.1 Overview of the interpreter

The goal of this chapter is to show you how to design a program that accepts arithmetic expressions from the user and then displays the results of evaluating those expressions. The basic operation of the interpreter is therefore to execute the following steps repeatedly as part of a loop in the main program:

1. Read in an expression from the user and translate it into an appropriate internal form.
2. Evaluate the expression to produce an integer result.
3. Print the result of the evaluation on the console.

This iterated process is characteristic of interpreters and is called a **read-eval-print loop**.

At this level of abstraction, the code for the read-eval-print interpreter is extremely simple. Although the final version of the program will include a little more code than is shown here, the following main program captures the essence of the interpreter:

```

int main() {
    while (true) {
        Expression *exp = ReadExp();
        int value = exp->eval();
        cout << value << endl;
        delete exp;
    }
    return 0;
}

```

As you can see, the idealized structure of the main program is simply a loop that calls functions to accomplish each phase of the read-eval-print loop. In this formulation, the task of reading an expression is indicated by a call to the function **ReadExp**, which will be replaced in subsequent versions of the interpreter program with a somewhat longer sequence of statements. Conceptually, **ReadExp** is responsible for reading an expression from the user and converting it into its internal representation, which takes the form of a pointer to an **Expression** object. The task of evaluating the expression falls to the **eval** method, which returns the integer you get if you apply all the operators in the expression in the appropriate order. The **state** parameter is used to maintain the overall state of the interpreter and includes such information as the values of any variables. The print phase of the read-eval-print loop is simply a matter of writing the value to **cout**.

At this point, you don't yet have any detailed sense of what the **Expression** class is or how it is represented. From its declaration, you know that the variable **exp** is a pointer to an **Expression** object, which means that any methods pertaining to the object to which **exp** points will use the **->** operator rather than the **.** operator that appears when objects are used directly. Also, you can infer from the code that the **Expression** class has a method called **eval**, even though you don't know the details of that operation. That, of course, is how it should be. As a client of the expression package, you are less concerned with how expressions are implemented than you are with how to use them. As a client, you need to think of the **Expression** class as an abstract data type. The underlying details become important only when you have to understand the implementation.

The operation of the **ReadExp** function consists of the three following steps:

1. *Input.* The input phase consists of reading in a line of text from the user, which can be accomplished with a simple call to the **GetLine** function from **simpio.h**.
2. *Lexical analysis.* The lexical analysis phase consists of dividing the input line into individual units called *tokens*, each of which represents a single logical entity, such as an integer constant, an operator, or a variable name. Fortunately, all the facilities required to implement lexical analysis are provided by the **Scanner** class introduced in Chapter 4.
3. *Parsing.* Once the line has been broken down into its component tokens, the parsing phase consists of determining whether the individual tokens represent a legal expression and, if so, what the structure of that expression is. To do so, the parser must determine how to construct a valid parse tree from the individual tokens in the input.

It would be easy enough to implement **ReadExp** as a single function that combines each of these steps. In many applications, however, it makes more sense to keep the individual phases separate because doing so gives you more flexibility in designing the interpreter structure. The full implementation of the interpreter therefore includes explicit code for each of the three phases, as you can see in Figure 14-1.

Figure 14-1 Main module for the interpreter

```

/*
 * File: interp.cpp
 * -----
 * This program simulates the top level of a programming
 * language interpreter. The program reads an expression,
 * evaluates the expression, and displays the result.
 */

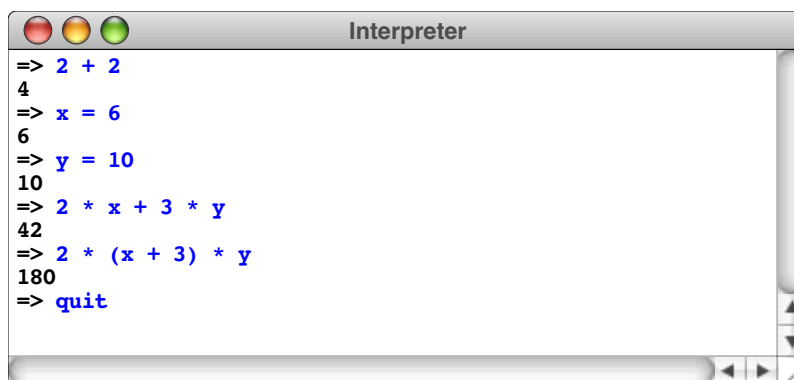
#include "genlib.h"
#include "simpio.h"
#include "exp.h"
#include "evalstate.h"
#include "parser.h"
#include "scanner.h"
#include <iostream>

/* Main program */

int main() {
    EvalState state;
    Scanner scanner;
    scanner.setSpaceOption(Scanner::IgnoreSpaces);
    while (true) {
        cout << "=> ";
        string line = GetLine();
        if (line == "quit") break;
        scanner.setInput(line);
        Expression *exp = ParseExp(scanner);
        int value = exp->eval(state);
        cout << value << endl;
        delete exp;
    }
    return 0;
}

```

The following sample run illustrates the operation of the interpreter as implemented in Figure 14-1:



```

Interpreter
=> 2 + 2
4
=> x = 6
6
=> y = 10
10
=> 2 * x + 3 * y
42
=> 2 * (x + 3) * y
180
=> quit

```

As the sample run makes clear, the interpreter allows assignment to variables and adheres to C++'s precedence conventions by evaluating multiplication before addition.

Implementing variables and assignment in the interpreter explains another difference between the simple read-eval-print loop and the more detailed implementation in Figure 14-1. In the final version of the code, the `eval` method in the `Expression` class takes a parameter called `state`, which is an object of type `EvalState`. This parameter contains—possibly along with additional information—a **symbol table**, which maps each variable name to its corresponding value. Because variables maintain their value across many calls to `eval`, the symbol table information must be passed as a parameter.

Although the `Map` class from Chapter 4 provides an appropriate tool for implementing a symbol table, using a `Map` to convey this information is probably not the ideal design. What this code does instead is define a new class called `EvalState` that keeps track of all information that needs to be preserved as the interpreter runs. In the simplest versions of the interpreter, an `EvalState` object might contain no information besides the `Map` used to implement the symbol table, as shown in Figures 14-2 and 14-3. If it later becomes necessary to extend the interpreter so that it keeps track of additional information, having the `EvalState` class provides a place to add that data without being forced to change the overall structure of the program.

Figure 14-2 The interface for the `EvalState` class

```
/*
 * File: evalstate.h
 * -----
 * This interface exports a class called EvalState, which
 * keeps track of additional information required by the
 * evaluator, most notably the values of variables.
 */

#ifndef _evalstate_h
#define _evalstate_h

#include "genlib.h"
#include "map.h"

/*
 * Class: EvalState
 * -----
 * This class is passed by reference through the recursive levels
 * of the evaluator and contains information from the evaluation
 * environment that the evaluator may need to know. The only
 * such information implemented here is a symbol table that maps
 * variable names into their values.
 */

class EvalState {

public:

    /*
     * Constructor: EvalState
     * Usage: EvalState state;
     * -----
     * Creates a new EvalState object with no variable bindings.
     */

    EvalState();
```

```
/*
 * Destructor: ~EvalState
 * Usage: usually implicit
 * -----
 * Frees all heap storage associated with this object.
 */
~EvalState();

/*
 * Method: setValue
 * Usage: state.setValue(var, value);
 * -----
 * Sets the value associated with the specified var.
 */
void setValue(string var, int value);

/*
 * Method: getValue
 * Usage: int value = state.getValue(var);
 * -----
 * Returns the value associated with the specified variable.
 */
int getValue(string var);

/*
 * Method: isDefined
 * Usage: if (state.isDefined(var)) . . .
 * -----
 * Returns true if the specified variable is defined.
 */
bool isDefined(string var);

private:

/*
 * Implementation notes: private data
 * -----
 * The only private data required for this version of EvalState
 * is a map that stores the mapping of variable names to their
 * integer values.
 */
Map<int> symbolTable;
};

#endif
```


Figure 14-3 Implementation of the `EvalState` class

```

/*
 * File: evalstate.cpp
 * -----
 * This file implements the EvalState class, which maintains all
 * state information required by the interpreter. In this version,
 * the only required state is a symbol table that keeps track of
 * the value of identifiers. The methods in this class are simple
 * enough that they need no individual documentation.
 */

#include "genlib.h"
#include "evalstate.h"
#include "map.h"

/* Implementation of the EvalState class */

EvalState::EvalState() {
    /* Implemented automatically by Map constructor */
}

EvalState::~EvalState() {
    /* Implemented automatically by Map destructor */
}

void EvalState::setValue(string var, int value) {
    symbolTable.put(var, value);
}

int EvalState::getValue(string var) {
    return symbolTable.get(var);
}

bool EvalState::isDefined(string var) {
    return symbolTable.containsKey(var);
}

```

Although the code for the main program is quite straightforward, you still have some unfinished business. First, you need to think about exactly what expressions are and how to represent them. Then, you have to implement the `ParseExp` function. Because each of these problems involves some subtlety, completing the interpreter will take up the remainder of the chapter.

14.2 Understanding the abstract structure of expressions

Your first task in completing the interpreter is to understand the concept of an expression and how that concept can be represented as an object. As is often the case when you are thinking about a programming abstraction, it makes sense to begin with the insights you have acquired about expressions from your experience as a C++ programmer. For example, you know that the lines

```

0
2 * 11
3 * (a + b + c)
x = x + 1

```

represent legal expressions in C++. At the same time, you also know that the lines

```
2 * (x - y
17 k
```

are not expressions; the first has unbalanced parentheses, and the second is missing an operator. An important part of understanding expressions is articulating what constitutes an expression so that you can differentiate legal expressions from malformed ones.

A recursive definition of expressions

As it happens, the best way to define the structure of a legal expression is to adopt a recursive perspective. A sequence of symbols is an expression if it has one of the following forms:

1. An integer constant
2. A variable name
3. An expression enclosed in parentheses
4. A sequence of two expressions separated by an operator

The first two possibilities represent the simple cases for the recursive definition. The remaining possibilities, however, define an expression in terms of simpler ones.

To see how you might apply this recursive definition, consider the following sequence of symbols:

```
y = 3 * (x + 1)
```

Does this sequence constitute an expression? You know from experience that the answer is yes, but you can use the recursive definition of an expression to justify that answer. The integer constants 3 and 1 are expressions according to rule #1. Similarly, the variable names **x** and **y** are expressions as specified by rule #2. Thus, you already know that the expressions marked by the symbol *exp* in the following diagram are expressions, as defined by the simple-case rules:

```

exp      exp      exp      exp
|        |        |        |
y  =  3  *  (  x  +  1  )

```

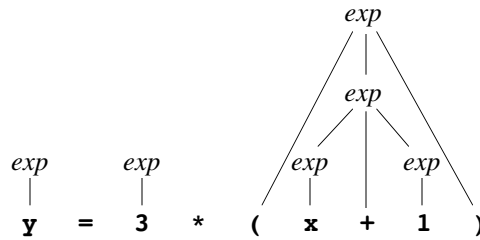
At this point, you can start to apply the recursive rules. Given that **x** and **1** are both expressions, you can tell that the string of symbols **x + 1** is an expression by applying rule #4, because it consists of two expressions separated by an operator. You can record this observation in the diagram by adding a new expression marker tied to the parts of the expression that match the rule, as shown:

```

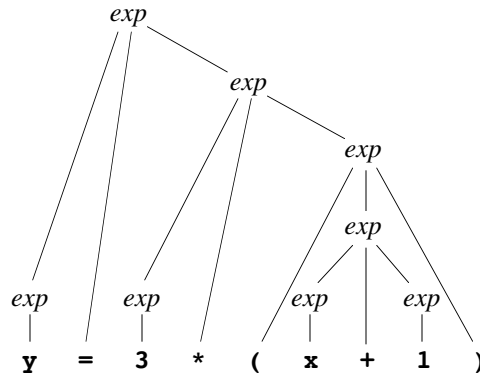
exp      exp      exp
|        |        |
y  =  3  *  (  x  +  1  )
              |
              exp
            /  |  \
           exp + exp
           |   |   |
          x   +   1

```

The parenthesized quantity can now be identified as an expression according to rule #3, which results in the following diagram:



By applying rule #4 two more times to take care of the remaining operators, you can show that the entire set of characters is indeed an expression, as follows:



As you can see, this diagram forms a tree. A tree that demonstrates how a sequence of input symbols fits the syntactic rules of a programming language is called a **parse tree**.

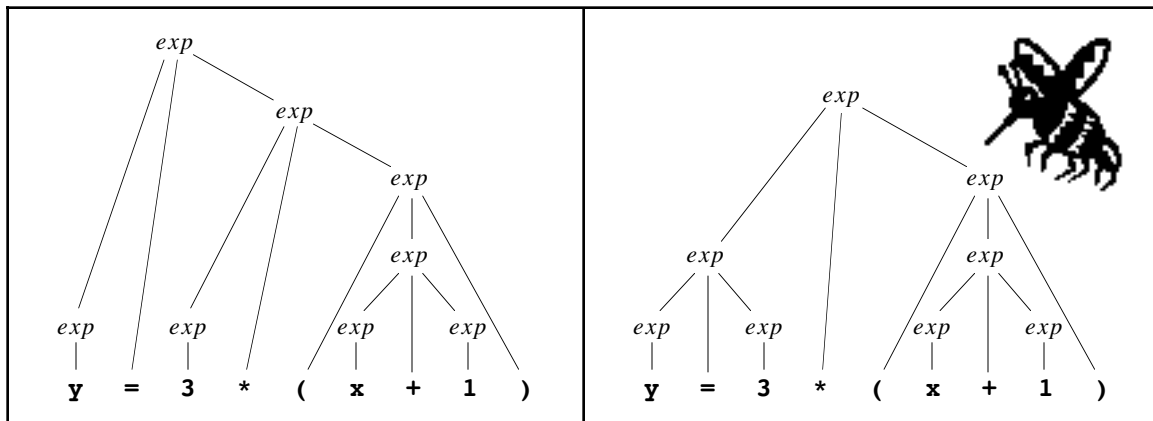
Ambiguity

Generating a parse tree from a sequence of symbols requires a certain amount of caution. Given the four rules for expressions outlined in the preceding section, you can form more than one parse tree for the expression

$$y = 3 * (x + 1)$$

Although the tree structure shown at the end of the last section presumably represents what the programmer intended, it is just as valid to argue that $y = 3$ is an expression according to rule #4, and that the entire expression therefore consists of the expression $y = 3$, followed by a multiplication sign, followed by the expression $(x + 1)$. This argument ultimately reaches the same conclusion about whether the input line represents an expression, but generates a different parse tree. Both parse trees are shown in Figure 14-4. The parse tree on the left is the one generated in the last section and corresponds to what a C++ programmer means by that expression. The parse tree on the right represents a legal application of the expression rules but reflects an incorrect ordering of the operations, given C++'s rules of precedence.

The problem with the second parse tree is that it ignores the mathematical rule specifying that multiplication is performed before assignment. The recursive definition of an expression indicates only that a sequence of two expressions separated by an operator is an expression; it says nothing about the relative precedence of the different operators and therefore admits both the intended and unintended interpretations. Because it allows multiple interpretations of the same string, the informal definition of expression given in the preceding section is said to be **ambiguous**. To resolve the ambiguity, the parsing algorithm must include some mechanism for determining the order in which operators are applied.

Figure 14-4. Intended parse tree and a legal but incorrect alternative

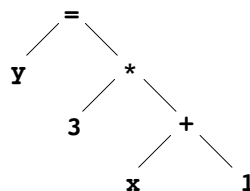
The question of how to resolve the ambiguity in an expression during the parsing phase is discussed in the section on “Parsing an expression” later in this chapter. At the moment, the point of introducing parse trees is to provide some insight into how you might represent an expression as a data structure. To this end, it is extremely important to make the following observation about the parse trees in Figure 14-4: the trees themselves are not ambiguous. The structure of each parse tree explicitly represents the structure of the expression. The ambiguity exists only in deciding how to generate the parse tree from the original string of constants, variables, and operators. Once you have the correct parse tree, its structure contains everything you need to understand the order in which the operators need to be applied.

Expression trees

In fact, parse trees contain more information than you need in the evaluation phase. Parentheses are useful in determining how to generate the parse tree but play no role in the evaluation of an expression once its structure is known. If your concern is simply to find the value of an expression, you do not need to include parentheses within the structure. This observation allows you to simplify a complete parse tree into an abstract structure called an **expression tree** that is more appropriate to the evaluation phase. In the expression tree, nodes in the parse tree that represent parenthesized subexpressions are eliminated. Moreover, it is convenient to drop the *exp* labels from the tree and instead mark each node in the tree with the appropriate operator symbol. For example, the intended interpretation of the expression

$$y = 3 * (x + 1)$$

corresponds to the following expression tree:



The structure of an expression tree is similar in many ways to the binary search tree from Chapter 13, but there are also some important differences. In the binary search tree, every node had the same structure. In an expression tree, there are three different types of nodes, as follows:

1. *Integer nodes* represent integer constants, such as 3 and 1 in the example tree.
2. *Identifier nodes* represent the names of variables and are presumably represented internally by a string.
3. *Compound nodes* represent the application of an operator to two operands, each of which is an arbitrary expression tree.

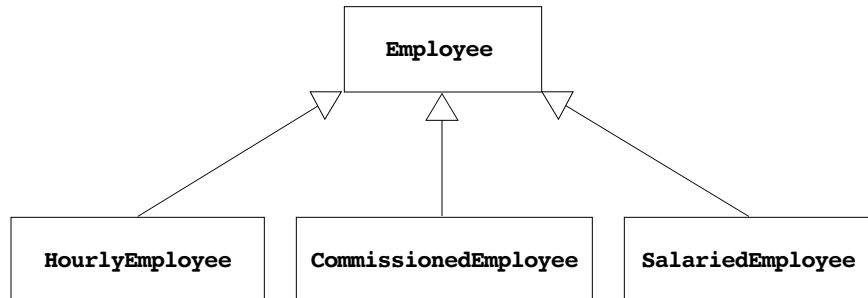
Each of these node types corresponds to one of the rules in the recursive formulation of an expression. The definition of the **Expression** class itself must make it possible for clients to work with expression nodes of all three types. Similarly, the underlying implementation must somehow make it possible for different expression types to coexist within the tree.

To represent such a structure, you need to define a representation for expressions that allows them to have different structures depending on their type. An integer expression, for example, must include the value of the integer as part of its internal structure. An identifier expression must include the name of the identifier. A compound expression must include the operator along with the left and right subexpressions. Defining a single abstract type that allows expressions to take on these different underlying structures requires you to learn a new aspect of C++’s type system, which is introduced in the next section. Once these preliminaries are out of the way, section 14.4 will return to the problem of representing expressions.

14.3 Class hierarchies and inheritance

Object-oriented languages like C++ and Java allow you define hierarchical relationships among classes. Whenever you have a class that provides some of the functionality you need for a particular application, you can define new classes that are derived from the original class, but which specialize its behavior in some way. The derived classes are known as **subclasses** of the original class, which in turn becomes the **superclass** for each of its subclasses.

As an example, suppose that you have been charged with designing an object-oriented payroll system for a company. You might begin by defining a general class called **Employee**, which encapsulates the information about an individual worker along with methods that implement operations required for the payroll system. These operations could include simple methods like **getName**, which returns the name of an employee, along with more complicated methods like **computePay**, which calculates the pay for an employee based on data stored within each **Employee** object. In many companies, however, employees fall into several different classes that are similar in certain respects but different in others. For example, a company might have hourly employees, commissioned employees, and salaried employees on the same payroll. In such companies, it might make sense to define subclasses for each employee category as illustrated by the following diagram:



Each of the classes **HourlyEmployee**, **CommissionedEmployee**, and **SalariedEmployee** is a subclass of the more general **Employee** class, which acts as their common superclass.

By default, each subclass **inherits** the behavior its superclass, which means that the methods and internal data structure of the superclass are also available to its subclasses. In cases in which the behavior of a subclass needs to differ from its superclass, the designer of the subclass can define entirely new methods for that subclass or **override** existing methods with modified ones. In the payroll example, all three subclasses will presumably inherit the **getName** method from the **Employee** superclass. All employees, after all, have a name. On the other hand, it probably makes sense to write separate **computePay** methods for each subclass, because the computation is likely to be different in each case.

The relationship between the subclasses and the superclass goes beyond just the convenience of allowing the common implementation to be shared between the classes. The subclass has an **is-a** relationship with the superclass; that is, a **SalariedEmployee** is an **Employee**. This means that in any context where an **Employee** object is used, a **SalariedEmployee** can be substituted instead. Anything that a client can do with an **Employee** object (*i.e.*, using the features available in the public interface) can be equivalently done to a **SalariedEmployee** object. This powerful **subtyping** relationship makes it possible for client code to be written in terms of the generic **Employee** object type and any specialization required for a specific kind of employee is handled by the subclass implementation.

14.4 Defining an inheritance hierarchy for expressions

As noted earlier, there are three different types of expressions that can make up an expression tree. An integer expression requires different storage and is evaluated differently than an identifier or compound expression. Yet all three of these types of expressions need to be able to coexist within an expression tree and need to behave similarly from an abstract perspective.

An inheritance hierarchy is an appropriate way to represent the different types of expression trees. At the top of the hierarchy will be the **Expression** class that specifies the features that will be common to each of the expression types. The **Expression** class has three subclasses, one for each expression type. The definitions for all four of these classes—the high-level **Expression** class and the lower-level subclasses **ConstantExp**, **IdentifierExp**, and **CompoundExp**—are all included as part of the **exp.h** interface.

As is typical for a class hierarchy, many of the most common methods are defined at the level of the **Expression** class but implemented individually in each of the subclasses. Every **Expression** object implements three methods:

1. The **eval** method determines the value of the expression, which is always an integer in this implementation. For constant expressions, the value is simply the value of the constant stored in the node. For identifier expressions, the value is determined by looking up the identifier name in a symbol table and returning the corresponding value. For compound expressions, the value must be computed by recursively evaluating the subexpressions and then applying the appropriate operator.
2. The **toString** method converts an expression into a string that makes the structure explicit by adding parentheses around every subexpression, even if those parentheses are not required. Although the **toString** method is not used in the interpreter, it is very useful to have around, particularly during debugging. If you are unsure whether an expression has the correct form, you can use **toString** to verify its structure.

3. The **type** method makes it possible to determine the type of an existing expression. The return value is one of the enumeration constants defined for the type **expTypeT**: **ConstantType**, **IdentifierType**, and **CompoundType**. Being able to check the type of an expression—in conjunction with the getter methods in each of the expression subclasses—makes it possible to create new expressions from existing ones.

In the **Expression** class itself, each of these methods is declared using the C++ keyword **virtual**. The **virtual** keyword informs the compiler that this method can be overridden by a subclass. The **virtual** keyword ensures that the method is invoked using the dynamic run-time type of the object instead of relying on the static compile-time type. For example, consider this code fragment

```
Expression *exp = ParseExp(scanner);
int value = exp->eval(state);
```

In the above code, **exp** has a compile-time type of “pointer to **Expression**.” In reality, however, the variable **exp** points to a particular expression subclass, which means that its type might really be “pointer to **ConstantExp**” or “pointer to **CompoundExp**.” When you invoke **eval**, you want to use the overridden version that is defined for the specific subclass. By tagging the **eval** method with the **virtual** keyword, you indicate that the method should be chosen based on the dynamic type of the object. This **dynamic dispatch** is typically the right option for any class for which you create subclasses.

All classes that are subtypes of **Expression**—integers, identifiers, and compound nodes—are able to evaluate themselves using the **eval** method. What the **Expression** superclass does is declare the common prototype for that **eval** method so that clients can call it for any type of expression. At the same time, it isn’t possible to evaluate an expression unless you know what type of expression it is. You can evaluate an integer or an identifier easily enough, but you can’t evaluate a generic expression without more information. Therefore the method **eval** in the **Expression** class is indicated as **pure virtual**, which means the superclass provides no default implementation. Instead, each subclass must supply an implementation that is appropriate for that node type.

If you think about this problem, you’ll soon realize that the **Expression** class is somewhat different from the other classes in this hierarchy. You can’t have an **Expression** object that is not also a member of one of its subclasses. It never makes sense to construct an **Expression** object in its own right. Whenever you want to create an expression, you simply construct an object of the appropriate subclass. Classes, like **Expression**, that are never constructed are called **abstract classes**. In C++, you indicate that a class is abstract by including at least one pure virtual method in the class interface.

Defining the interface for the Expression subclasses

In C++, the inheritance relationship for a class is declared in the class header like this

```
class ConstantExp : public Expression {
```

This class header declares the new class **ConstantExp** to be a public subclass of the **Expression** class. Being a public subclass means that all of the public features of the **Expression** class are inherited and public in the **ConstantExp** class. This establishes the subtyping relationship that an **ConstantExp** is an **Expression**, which means an **ConstantExp** object can be substituted wherever an **Expression** object is expected.

Each concrete **Expression** subclass must provide the implementation for the two pure virtual methods declared in the superclass: **toString** and **eval**. Each expression

subclass, whether it be an integer constant, an identifier, or a compound expression, will have its own specific way of implementing these methods, but must provide that functionality using the exact prototype specified by the superclass.

Each subclass also declares its own constructor that depends on the expression type. To construct an integer expression, for example, you need to know the value of the integer constant. To construct a compound expression, you need to specify the operator along with the left and right subexpressions.

Figure 14-5 shows the interface for the **Expression** abstract superclass and its three subclasses, along with a simple definition of an **EvalState** class that supports binding variable names to integer values. All **Expression** objects are immutable, which means that any **Expression** object, once created, will never change. Although clients are free to embed existing expressions in larger ones, the interface offers no facilities for changing the components of any existing expression. Using an immutable type to represent expressions helps enforce the separation between the implementation of the **Expression** class and its clients. Because those clients are prohibited from making changes in the underlying representation, they are unable to change the internal structure in a way that violates the requirements for expression trees.

As written, the **Expression** classes export constructors, string conversion, and evaluation functions. There are, however, other operations on expressions that you might at first think belong in this interface. For example, the main program for the interpreter calls the function **ParseExp**, which is in some sense part of the behavior of the expression type. This observation raises the question of whether the **exp.h** interface should export that function as well.

Although **ParseExp** must be defined somewhere in the code, exporting it through the **exp.h** interface may not be the best design strategy. In a full-scale interpreter, the parser requires a significant amount of code—enough to warrant making this phase a complete module in its own right. In the stripped-down version of the interpreter presented in this chapter, the code is much smaller. Even so, it makes sense to partition the phases of the interpreter into separate modules for the following reasons:

1. *The resulting modular decomposition resembles more closely the structure you would tend to encounter in practice.* Full-scale interpreters are divided into separate modules; following this convention even in our restricted example clarifies how the pieces fit together.
2. *The program will be easier to maintain as you add features.* Getting the module structure right early in the implementation of a large system makes it easier for that system to evolve smoothly over time. If you start with a single module and later discover that the program is growing too large, it usually takes more work to separate the modules than it would have earlier in the program evolution.
3. *Using separate module for the parser makes it easier to substitute new implementations.* One of the principal advantages of using a modular design is that doing so makes it easier to substitute one implementation of an interface for another. For example, the section on “Parsing” later in this chapter defines two different implementations of the **ParseExp** function. If **ParseExp** is exported by the **exp.h** interface, it is more difficult to substitute a new implementation than it would be if **ParseExp** were exported from a separate module.

For these reasons, the **exp.h** interface exports only the types needed to represent expressions, along with the constructor and evaluation functions. The **ParseExp** function is exported by a separate interface called **parser.h**.

Figure 14-5 The exp.h interface

```

/*
 * File: exp.h
 * -----
 * This interface defines a class hierarchy for expressions,
 * which allows the client to represent and manipulate simple
 * binary expression trees.
 */

#ifndef _exp_h
#define _exp_h

#include "genlib.h"
#include "evalstate.h"
#include "map.h"

/*
 * Type: expTypeT
 * -----
 * This enumerated type is used to differentiate the three
 * different expression types: ConstantType, IdentifierType,
 * and CompoundType.
 */

enum expTypeT { ConstantType, IdentifierType, CompoundType };

/*
 * Class: Expression
 * -----
 * This class is used to represent a node in an expression tree.
 * Expression is an example of an abstract class, which defines
 * the structure and behavior of a set of classes but has no
 * objects of its own. Any object must be one of the three
 * concrete subclasses of Expression:
 *
 * 1. ConstantExp -- an integer constant
 * 2. IdentifierExp -- a string representing an identifier
 * 3. CompoundExp -- two expressions combined by an operator
 *
 * The Expression class defines the interface common to all
 * Expression objects; each subclass provides its own specific
 * implementation of the common interface.
 *
 * Note on syntax: Each of the virtual methods in the Expression
 * class is marked with the designation = 0 on the prototype line.
 * This notation is used in C++ to indicate that this method is
 * purely virtual and will always be supplied by the subclass.
 */

```

```

class Expression {
public:

    /*
     * Constructor: Expression
     * -----
     * The base class constructor is empty. Each subclass must provide
     * its own constructor.
     */

    Expression();

    /*
     * Destructor: ~Expression
     * Usage: delete exp;
     * -----
     * The destructor deallocates the storage for this expression.
     * It must be declared virtual to ensure that the correct subclass
     * destructor is called when deleting an expression.
     */

    virtual ~Expression();

    /*
     * Method: eval
     * Usage: result = exp->eval(state);
     * -----
     * This method evaluates this expression and returns its value in
     * the context of the specified EvalState object.
     */

    virtual int eval(EvalState & state) = 0;

    /*
     * Method: toString
     * Usage: str = exp->toString();
     * -----
     * This method returns a string representation of this expression.
     */

    virtual string toString() = 0;

    /*
     * Method: type
     * Usage: type = exp->type();
     * -----
     * This method returns the type of the expression, which must be one
     * of the constants ConstantType, IdentifierType, or CompoundType.
     */

    virtual expTypeT type() = 0;
};

```

```
/*
 * Class: ConstantExp
 * -----
 * This subclass represents a constant integer expression.
 */

class ConstantExp: public Expression {
public:

    /*
     * Constructor: ConstantExp
     * Usage: Expression *exp = new ConstantExp(10);
     * -----
     * The constructor initializes a new integer constant expression
     * to the given value.
     */

    ConstantExp(int val);

    /*
     * Prototypes for the virtual methods
     * -----
     * These method have the same prototypes as those in the Expression
     * base class and don't require additional documentation.
     */

    virtual int eval(EvalState & state);
    virtual string toString();
    virtual expTypeT type();

    /*
     * Method: getValue
     * Usage: value = ((ConstantExp *) exp)->getValue();
     * -----
     * This method returns the value field without calling eval and
     * can be applied only to an object known to be a ConstantExp.
     */

    int getValue();

private:

    int value;
};
```

```
/*
 * Class: IdentifierExp
 * -----
 * This subclass represents a expression corresponding to a variable.
 */

class IdentifierExp : public Expression {
public:

    /*
     * Constructor: IdentifierExp
     * Usage: Expression *exp = new IdentifierExp("count");
     * -----
     * The constructor initializes a new identifier expression
     * for the variable named by name.
     */

    IdentifierExp(string name);

    /*
     * Prototypes for the virtual methods
     * -----
     * These method have the same prototypes as those in the Expression
     * base class and don't require additional documentation.
     */

    virtual int eval(EvalState & state);
    virtual string toString();
    virtual expTypeT type();

    /*
     * Method: getName
     * Usage: name = ((IdentifierExp *) exp)->getName();
     * -----
     * This method returns the name field of the identifier node and
     * can be applied only to an object known to be an IdentifierExp.
     */

    string getName();

private:

    string name;
};
```

```

/*
 * Class: CompoundExp
 * -----
 * This subclass represents a compound expression consisting of
 * two subexpressions joined by an operator.
 */

class CompoundExp: public Expression {

public:

/*
 * Constructor: CompoundExp
 * Usage: Expression *exp = new CompoundExp('+', e1, e2);
 * -----
 * The constructor initializes a new compound expression
 * which is composed of the operator (op) and the left and
 * right subexpression (lhs and rhs).
 */

    CompoundExp(char op, Expression *lhs, Expression *rhs);

/*
 * Prototypes for the virtual methods
 * -----
 * These method have the same prototypes as those in the Expression
 * base class and don't require additional documentation.
 */

    virtual ~CompoundExp();
    virtual int eval(EvalState & state);
    virtual string toString();
    virtual expTypeT type();

/*
 * Methods: getOp, getLHS, getRHS
 * Usage: op = ((CompoundExp *) exp)->getOp();
 *         lhs = ((CompoundExp *) exp)->getLHS();
 *         rhs = ((CompoundExp *) exp)->getRHS();
 * -----
 * These methods return the components of a compound node and can
 * be applied only to an object known to be a CompoundExp.
 */

    char getOp();
    Expression *getLHS();
    Expression *getRHS();

private:

    char op;
    Expression *lhs, *rhs;

};

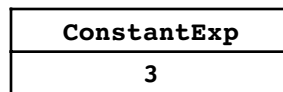
#endif

```

14.5 Implementing the expression subclasses

The abstract **Expression** superclass declares no data members. This design makes sense because no data values are common to all node types. Each specific subclass has its own unique storage requirements—an integer node needs to store an integer constant, a compound node stores pointers to its subexpressions, and so on. Each subclass declares those specific data members that are required for its particular expression type.

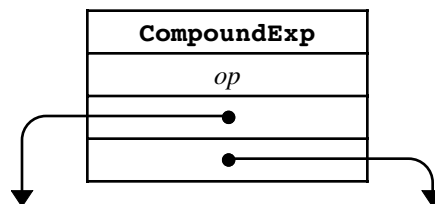
To reinforce your understanding of how **Expression** objects are stored, you can visualize how the concrete structure is represented inside the computer's memory. The representation of an **Expression** object depends on its specific subclass. You can diagram the structure of an expression tree by considering the three classes independently. An **ConstantExp** object simply stores an integer value, shown here as it would exist for the integer 3:



An **IdentifierExp** object stores a string representing a variable name, as illustrated here for the variable **x**:



In the case of a **CompoundExp** object, it stores the binary operator along with two pointers which indicate the left and right subexpressions:



Because compound nodes contain subexpressions that can themselves be compound nodes, expression trees can grow to an arbitrary level of complexity. Figure 14-6 illustrates the internal data structure for the expression

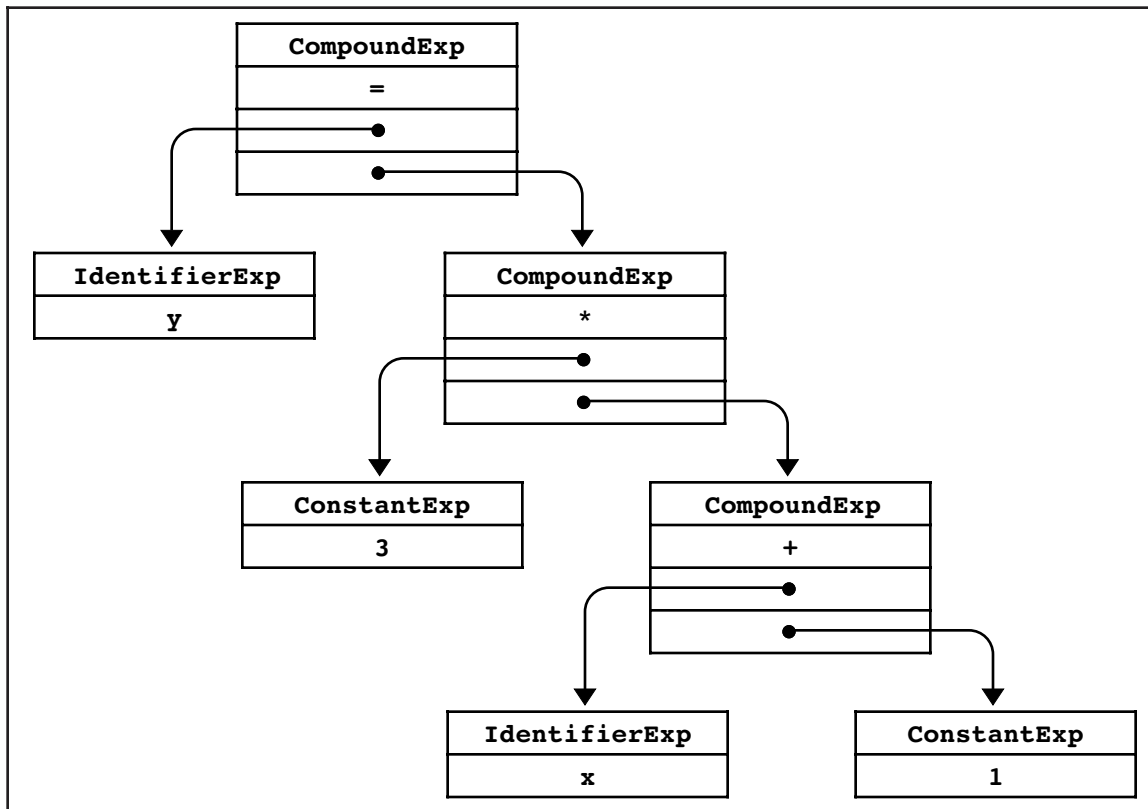
$$y = 3 * (x + 1)$$

which includes three operators and therefore requires three compound nodes. Although the parentheses do not appear explicitly in the expression tree, its structure correctly reflects the desired order of operations.

Implementing the methods

The methods in the expression classes are quite easy to implement. Each subclass provides a constructor that takes in appropriate arguments and initializes the data members. The implementation of the **toString** method uses the information from the data members to return a string representation of the expression.

The only remaining task to implement the evaluation method. Each subclass has its own strategy for evaluating an expression. Integer expressions are the easiest. The value

Figure 14-6 Representation of the expression $y = 3 * (x + 1)$ 

of an expression of an integer expression is simply the value of the integer stored in that node. Thus, the **ConstantExp** **eval** method looks like

```
int ConstantExp::eval(EvalState & state) {
    return value;
}
```

Note that even though an **ConstantExp** does not use the parameter **state**, it is required in the prototype for **eval** method so that it exactly matches the prototype given in the **Expression** superclass.

The next case to consider is that of identifiers. To evaluate an identifier expression, you look up the variable in the variable table and return the associated value as shown:

```
int IdentifierExp::eval(EvalState & state) {
    if (!varTable.containsKey(id)) Error(id + " is undefined");
    return state.getValue(id);
}
```

The last case you need to consider is that of compound expressions. A compound expression consists of an operator and two subexpressions, but you must differentiate two subcases: the arithmetic operators (+, −, *, and /) and the assignment operator (=). For the arithmetic operators, all you have to do is evaluate the left and right subexpressions recursively and then apply the appropriate operation. For assignment, you need to evaluate the right-hand side and then store that value into the variable table for the identifier on the left-hand side.

The full implementation of the **Expression** class hierarchy appears in Figure 14-7.

Figure 14-7 The exp.cpp implementation

```

/*
 * File: exp.cpp
 * -----
 * This file implements the Expression class and its subclasses.
 */

#include "genlib.h"
#include "exp.h"
#include "evalstate.h"

/* Implementation of the base Expression class. */
Expression::Expression() { /* Empty */ }
Expression::~Expression() { /* Empty */ }

/* Implementation of the ConstantExp subclass. */
ConstantExp::ConstantExp(int value) {
    this->value = value;
}

int ConstantExp::eval(EvalState & state) {
    return value;
}

string ConstantExp::toString() {
    return IntegerToString(value);
}

expTypeT ConstantExp::type() {
    return ConstantType;
}

int ConstantExp::getValue() {
    return value;
}

/* Implementation of the IdentifierExp subclass. */
IdentifierExp::IdentifierExp(string name) {
    this->name = name;
}

int IdentifierExp::eval(EvalState & state) {
    if (!state.isDefined(name)) Error(name + " is undefined");
    return state.getValue(name);
}

string IdentifierExp::toString() {
    return name;
}

expTypeT IdentifierExp::type() {
    return IdentifierType;
}

string IdentifierExp::getName() {
    return name;
}

```



```

/*
 * Implementation of the CompoundExp subclass. For this subclass,
 * the implementation must include explicit code for evaluating each
 * of the operators.
 */

CompoundExp::CompoundExp(char op, Expression *lhs, Expression *rhs) {
    this->op = op;
    this->lhs = lhs;
    this->rhs = rhs;
}

CompoundExp::~CompoundExp() {
    delete lhs;
    delete rhs;
}

int CompoundExp::eval(EvalState & state) {
    if (op == '=') {
        if (lhs->type() != IdentifierType) {
            Error("Illegal variable in assignment");
        }
        int val = rhs->eval(state);
        state.setValue(((IdentifierExp *) lhs)->getName(), val);
        return val;
    }
    int left = lhs->eval(state);
    int right = rhs->eval(state);
    switch (op) {
        case '+': return left + right;
        case '-': return left - right;
        case '*': return left * right;
        case '/': return left / right;
    }
    Error("Illegal operator in expression");
    return 0; /* To avoid the warning message */
}

string CompoundExp::toString() {
    return '(' + lhs->toString() + ' ' + op + ' ' +
        rhs->toString() + ')';
}

expTypeT CompoundExp::type() {
    return CompoundType;
}

char CompoundExp::getOp() {
    return op;
}

Expression *CompoundExp::getLHS() {
    return lhs;
}

Expression *CompoundExp::getRHS() {
    return rhs;
}

```

14.6 Parsing an expression

The problem of building the appropriate parse tree from a stream of tokens is not an easy one. To a large extent, the underlying theory necessary to build a efficient parser lies beyond the scope of this text. Even so, it is possible to make some headway on the problem and solve it for the limited case of arithmetic expressions.

Parsing and grammars

In the early days of programming languages, programmers implemented the parsing phase of a compiler without thinking very hard about the nature of the process. As a result, early parsing programs were difficult to write and even harder to debug. In the 1960s, however, computer scientists studied the problem of parsing from a more theoretical perspective, which simplified it greatly. Today, a computer scientist who has taken a course on compilers can write a parser for a programming language with very little work. In fact, most parsers can be generated automatically from a simple specification of the language for which they are intended. In the field of computer science, parsing is one of the areas in which it is easiest to see the profound impact of theory on practice. Without the theoretical work necessary to simplify the problem, programming languages would have made far less headway than they have.

The essential theoretical insight necessary to simplify parsing is actually borrowed from linguistics. Like human languages, programming languages have rules of syntax that define the grammatical structure of the language. Moreover, because programming languages are much more regular in structure than human languages, it is usually easy to describe the syntactic structure of a programming language in a precise form called a **grammar**. In the context of a programming language, a grammar consists of a set of rules that show how a particular language construct can be derived from simpler ones.

If you start with the English rules for expression formation, it is not hard to write down a grammar for the simple expressions used in this chapter. Partly because it simplifies things a little in the parser, it helps to incorporate the notion of a term into the parser as any single unit that can appear as an operand to a larger expression. For example, constants and variables are clearly terms. Moreover, an expression in parentheses acts as a single unit and can therefore also be regarded as a term. Thus, a term is one of the following possibilities:

- An integer constant
- A variable
- An expression in parentheses

An expression is then either of the following:

- A term
- Two expressions separated by an operator

This informal definition can be translated directly into the following grammar, presented in what programmers call **BNF**, which stands for Backus-Naur form after its inventors John Backus and Peter Naur:

$$\begin{array}{ll} E \rightarrow T & T \rightarrow \textit{integer} \\ E \rightarrow E \textit{ op } E & T \rightarrow \textit{identifier} \\ & T \rightarrow (E) \end{array}$$

In the grammar, uppercase letters like E and T are called **nonterminal symbols** and stand for an abstract linguistic class, such as an expression or a term. The specific punctuation

marks and the italicized words represent the **terminal symbols**, which are those that appear in the token stream. Explicit terminal symbols, such as the parentheses in the last rule, must appear in the input exactly as written. The italicized words represent placeholders for tokens that fit their general description. Thus, the notation *integer* stands for any string of digits returned by the scanner as a token. Each terminal corresponds to exactly one token in the scanner stream. Nonterminals typically correspond to an entire sequence of tokens.

Like the informal rules for defining expressions presented in the section on “A recursive definition of expressions” earlier in the chapter, grammars can be used to generate parse trees. Just like those rules, this grammar is ambiguous as written and can generate several different parse trees for the same sequence of tokens. Once again, the problem is that the grammar does not incorporate any knowledge of the precedence of the operators and is therefore not immediately useful in constructing a parser.

Parsing without precedence

Before considering how it might be possible to add precedence to the grammar, it helps to think about circumventing this problem in a simpler way. What if there were no precedence in the language? Would that make parsing easier? Throwing away precedence is not as crazy an idea as it might seem. In the 1960s, Ken Iverson designed a language called APL (which is an abbreviation for *A Programming Language*), which is still in use today. Instead of using standard rules of precedence, APL operators all have equal precedence and are executed in strictly right-to-left order. Thus, the expression

$$2 * x + y$$

is interpreted in APL as if it had been written

$$2 * (x + y)$$

which is exactly the opposite of the conventional mathematical interpretation. To recover the conventional meaning, you would have to write

$$(2 * x) + y$$

in APL. This style of precedence is called **Iversonian precedence** after its inventor.

The problem of parsing turns out to be much easier for languages that use Iversonian precedence, mostly because, in them, the grammar for expressions can be written in a form that is both unambiguous and simple to parse:

$$\begin{array}{ll} E \rightarrow T & T \rightarrow \textit{integer} \\ E \rightarrow T \textit{ op } E & T \rightarrow \textit{identifier} \\ & T \rightarrow (E) \end{array}$$

This grammar is almost the same as the ambiguous grammar presented in the preceding section. The only difference is the rule

$$E \rightarrow T \textit{ op } E$$

which specifies that the left-hand operand to any operator must be a simple term.

Writing a parser based on the Iversonian expression grammar requires little more than a direct translation of the grammar into code. For each of the nonterminal symbols, you write a function that follows the structure of the grammar. For example, the task of reading an expression is assigned to a function called **ReadE**, whose structure follows the rules for expressions. To parse either of the two expression forms, the **ReadE** function

must first call the function **ReadT** to read a term and then check to see whether the next token is an operator. If it is, **ReadE** calls itself recursively to read the expression following the operator and creates a compound expression node from the parts. If the token is not an operator, **ReadE** calls **saveToken** to put that token back in the input being scanned where it will be read at a higher level of the recursive structure. In much the same way, the **ReadT** function implements the rules in the grammar that define a term. The code for **ReadT** begins by reading a token and determining whether it represents an integer, an identifier, or a parenthesized expression. If it does, **ReadT** returns the corresponding expression. If the token does not correspond to any of these possibilities, the expression is illegal.

Parsers that are structured as a collection of functions that call themselves recursively in a fashion guided by a grammar are called **recursive-descent parsers**. A complete implementation of a recursive-descent parser for expressions with Iversonian precedence appears in Figure 14-8. The real work is done by the mutually recursive functions **ReadE** and **ReadT**. The **ParseExp** function itself simply calls **ReadE** to read the expression and then checks to see that there are no extra tokens on the input line.

If all of this seems like magic, you should go through each step in the operation of the parser on a simple expression of your own choosing. As with many recursive functions, the code for the parser is simple even though the effect is profound.

Figure 14-8 An implementation of **ParseExp** using Iversonian precedence

```

/*
 * Implementation notes: ParseExp
 * -----
 * This function reads an expression and then checks for extra tokens.
 */

Expression *ParseExp(Scanner & scanner) {
    Expression *exp = ReadE(scanner);
    if (scanner.hasMoreTokens()) {
        Error("ParseExp found extra token: " + scanner.nextTok());
    }
    return exp;
}

/*
 * Implementation notes: ReadE
 * Usage: exp = ReadE(scanner);
 * -----
 * This function reads the next expression from the scanner by
 * matching the input to one of the following grammatical rules:
 *
 *      E  ->  T
 *      E  ->  T op E
 *
 * Both right-hand sides start with T, so the code can begin by
 * calling ReadT. If the next token is an operator, the code
 * creates a compound expression from the term, the operator,
 * and the expression after the operator.
 */

```

```

Expression *ReadE(Scanner & scanner) {
    Expression *exp = ReadT(scanner);
    string token = scanner.nextToken();
    if (IsOperator(token)) {
        Expression *rhs = ReadE(scanner);
        exp = new CompoundExp(token[0], exp, rhs);
    } else {
        scanner.saveToken(token);
    }
    return exp;
}

/*
 * Function: ReadT
 * Usage: exp = ReadT(scanner);
 * -----
 * This function reads a single term from the scanner by matching
 * the input to one of the following grammatical rules:
 *
 *      T -> integer
 *      T -> identifier
 *      T -> ( E )
 */

Expression *ReadT(Scanner & scanner) {
    Expression *exp;
    string token = scanner.nextToken();
    if (isdigit(token[0])) {
        exp = new ConstantExp(StringToInteger(token));
    } else if (isalpha(token[0])) {
        exp = new IdentifierExp(token);
    } else if (token == "(") {
        exp = ReadE(scanner);
        if (scanner.nextToken() != ")") {
            Error("Unbalanced parentheses in expression");
        }
    } else {
        Error("Illegal term in expression");
    }
    return exp;
}

/*
 * Function: IsOperator
 * Usage: if (IsOperator(token)) . . .
 * -----
 * This function returns true if the token is a legal operator.
 */

bool IsOperator(string token) {
    if (token.length() != 1) return false;
    switch (token[0]) {
        case '+': case '-': case '*': case '/': case '=':
            return true;
        default:
            return false;
    }
}

```

Adding precedence to the parser

Although the Iversonian parser presented in the preceding section illustrates the techniques of recursive-descent parsing, it probably seems somewhat unsatisfying because the expressions it parses do not behave in the way you have come to expect from your experience with languages like C++. Unfortunately, developing a simple, unambiguous grammar for traditional expressions is beyond the scope of this text. Nevertheless, you can incorporate the notion of precedence explicitly into the design of the recursive-descent parser.

As a start, you can define the precedence of the operators by writing a function **Precedence** that returns a numeric precedence for each of the legal operators, as follows:

```
int Precedence(string token) {
    if (StringLength(token) > 1) return 0;
    switch (token[0]) {
        case '=': return 1;
        case '+': case '-': return 2;
        case '*': case '/': return 3;
        default: return 0;
    }
}
```

Given the ability to determine the relative precedence of the operators, you can extend the definition of **ReadE** so that it takes the current precedence level as an argument. As long as the precedence of the operators it encounters is greater than the prevailing precedence, **ReadE** can create the appropriate compound expression node and then loop back to check the next operator. When **ReadE** encounters the end of the input or an operator whose precedence is less than or equal to the precedence at this level, it simply returns to the next higher level, where the prevailing precedence is lower. Each recursive call to **ReadE** passes the precedence of the operator it has just encountered to ensure that each operator is applied at the appropriate time. The code for the revised **ReadE** function therefore looks like this:

```
Expression *ReadE(Scanner & scanner, int prec) {
    Expression *exp = ReadT(scanner);
    string token;
    while (true) {
        token = scanner.nextToken();
        int newPrec = Precedence(token);
        if (newPrec <= prec) break;
        Expression *rhs = ReadE(scanner, newPrec);
        exp = new CompoundExp(token[0], exp, rhs);
    }
    scanner.saveToken(token);
    return exp;
}
```

The code for the other functions in the parser is largely unaffected, although this change means you can simplify **isOperator** so that it checks if the precedence is nonzero.

Even though the current application does not call any functions in this module other than **ParseExp**, the **parser.h** interface shown in Figure 14-9 also exports the functions **ReadE**, **ReadT**, and **Precedence**. These functions are quite useful in some applications, including several of the exercises at the end of this chapter.

Figure 14-9 The parser.h interface

```

/*
 * File: parser.h
 * -----
 * This file acts as the interface to the parser module, which
 * exports functions to parse expressions from a scanner.
 */

#ifndef _parser_h
#define _parser_h

#include "exp.h"
#include "scanner.h"

/*
 * Function: ParseExp
 * Usage: exp = ParseExp(scanner);
 * -----
 * This function parses an expression by reading tokens from
 * the scanner, which must be provided by the client.
 */

Expression *ParseExp(Scanner & scanner);

/*
 * Functions: ReadE, ReadT
 * Usage: exp = ReadE(scanner, prec);
 *        exp = ReadT(scanner);
 * -----
 * These functions provide low-level entry points to the parser,
 * which are useful to clients who need to use expression
 * parsing in large applications. ReadE(scanner, prec) returns
 * the next expression involving operators whose precedence
 * is at least prec; ReadT returns the next individual term.
 */

Expression *ReadE(Scanner & scanner, int prec = 0);
Expression *ReadT(Scanner & scanner);

/*
 * Function: Precedence
 * Usage: prec = Precedence(token);
 * -----
 * This function returns the precedence of the specified operator
 * token. If the token is not an operator, Precedence returns 0.
 */

int Precedence(string token);

#endif

```

Summary

In this chapter, you have taken your first steps toward understanding how compilers translate programs into an executable form by considering how to represent arithmetic expressions. Important points in the chapter include:

- The conventional tools for implementing programming languages fall into two classes: compilers and interpreters. Compilers translate source code into a set of instructions that can be executed directly by the hardware. Interpreters do not actually produce machine-executable code but instead achieve the same effect by executing the operations directly, as the source program is translated.
- A typical interpreter system operates by repeatedly reading an expression from the user, evaluating it, and displaying the result. This approach is called a *read-eval-print loop*.
- Expressions have a fundamentally recursive structure. There are simple expressions, which consist of constants and variable names. More complex expressions are created by combining simpler subexpressions into larger units, forming a hierarchical structure that can easily be represented as a tree.
- If you define expressions in their most straightforward recursive form, those involving multiple operators may be ambiguous in the sense that you can come up with several interpretations that are consistent with the basic form. Despite the ambiguity of the expression itself, the trees for the different interpretations are distinct, which means that ambiguity is a property of the written form of the expression and not its internal representation.
- It is easy to define a class to represent nodes in expression trees. The corresponding implementation, however, must support multiple representations to account for the fact that there are several different kinds of expressions. C++ inheritance can be used to define several subclasses that encompass the various possibilities.
- The process of reading an expression from the user can be divided into the phases of *input*, *lexical analysis*, and *parsing*. The input phase is the simplest and consists of reading a string from the user. Lexical analysis involves breaking a string into component tokens in the way that the scanner abstraction in Chapter 4 does. Parsing consists of translating the collection of tokens returned from the lexical analysis phase into its internal representation, following a set of syntactic rules called a *grammar*.
- For many grammars, it is possible to solve the parsing problem using a strategy called *recursive descent*. In a recursive-descent parser, the rules of the grammar are encoded as a set of mutually recursive functions.
- Once parsed, expression trees can be manipulated recursively in much the same way as the trees in Chapter 13. In the context of the interpreter, one of the most important operations is evaluating an expression tree, which consists of walking the tree recursively to determine its value.

Review questions

1. What is the difference between an interpreter and a compiler?
2. What is a read-eval-print loop?
3. What are the three phases involved in reading an expression?
4. State the recursive definition for an arithmetic expression as given in this chapter.

5. Identify which of the following lines constitutes an expression according to the definition used in this chapter:
 - a. `((0))`
 - b. `2x + 3y`
 - c. `x - (y * (x / y))`
 - d. `-y`
 - e. `x = (y = 2 * x - 3 * y)`
 - f. `10 - 9 + 8 / 7 * 6 - 5 + 4 * 3 / 2 - 1`
6. For each of the legal expressions in the preceding question, draw a parse tree that reflects the standard precedence assumptions of mathematics.
7. Of the legal expressions in question 5, which ones are ambiguous with respect to the simple recursive definition of expressions?
8. What are the differences between parse trees and expression trees?
9. What are the three types of expressions that can occur in an expression tree?
10. True or false: The methods in the `exp.h` interface do not work with `Expression` objects directly but instead use pointers to `Expression` objects.
11. What are the public methods of the `Expression` class?
12. Why is `ParseExp` not exported directly from `exp.h`?
13. Describe the relationship between a superclass and a subclass.
14. Define the term override.
15. What is a pure virtual method? Why is such a construct useful?
16. What is an abstract class? Is it possible for an abstract class to have public methods?
17. Using Figure 14-6 as a model, draw a complete structure diagram for the following expression:

$$y = (x + 1) / (x - 2)$$
18. Why are grammars useful in translating programming languages?
19. What do the letters in *BNF* stand for?
20. In a grammar, what is the difference between a terminal and a nonterminal symbol?
21. What is the value of the following expression if parsed using Iversonian precedence:

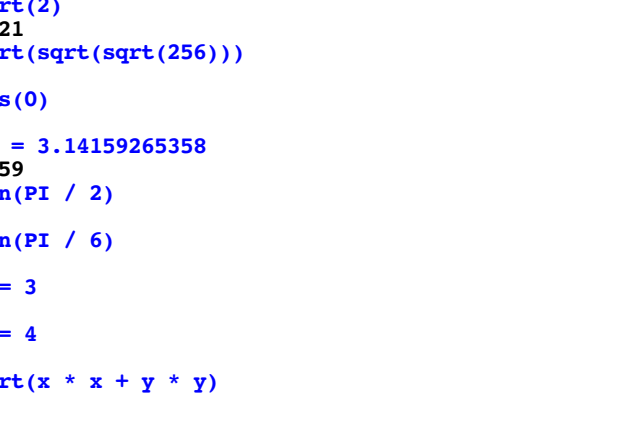
$$1 + 4 * 3 / 2 - 1$$
22. What is the value of the expression in the preceding question if parsed using standard mathematical precedence?
23. What is a recursive-descent parser?
24. What is the significance of the second argument to the `ReadE` function in the precedence-based implementation of the parser?

25. If you look at the definition of **ReadT** in Figure 14-8, you will see that the function body does not contain any calls to **ReadT**. Is **ReadT** a recursive function?
26. Why is the = operator handled specially in the method **CompoundExp::eval**?

Programming exercises

1. Make all the necessary changes to the interpreter program to add the operator `%`, which returns the remainder of its arguments as in C++. The precedence of `%` is the same as that for `*` and `/`.
2. Make the changes you would need to have the interpreter work with values of type `double` instead of type `int`.
3. Once you have finished implementing the change from `int` to `double` described in the previous exercise, extend the interpreter so that it supports simple mathematical functions, each of which takes an argument of type `double` and returns a `double` result. This change requires several extensions to the existing framework, including the following:
 - The expression structure defined in `exp.h` will need a new expression subtype that corresponds to a function call on a single argument.
 - The parser module will need to include a new grammatical rule for expressions that represents a function call with a single argument.
 - The evaluator implemented in `exp.cpp` will need to have some mechanism to apply the appropriate mathematical function given its name.

Your implementation should allow functions to be combined and nested just as they can be in a programming language. For example, if your interpreter defines the functions `sqrt`, `sin`, and `cos` your program should be able to produce the following sample run:



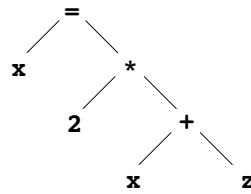
```
Interpreter
=> sqrt(2)
1.41421
=> sqrt(sqrt(sqrt(256)))
2
=> cos(0)
1
=> PI = 3.14159265358
3.14159
=> sin(PI / 2)
1
=> sin(PI / 6)
0.5
=> x = 3
3
=> y = 4
4
=> sqrt(x * x + y * y)
5
```

4. In mathematics, there are several common procedures that require you to replace all instances of a variable in a formula with some other variable. Working entirely as a client of the `exp.h` interface, write a function

```
Expression *ChangeVariable(Expression *exp,
```

```
string oldName, string newName);
```

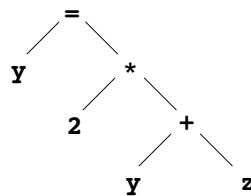
that returns a new expression which is the same as **exp** except that every occurrence of the identifier **oldName** is replaced with **newName**. For example, if **exp** is the expression



calling

```
Expression *newExp = ChangeVariable(exp, "x", "y");
```

will assign the following expression tree to **newExp**:

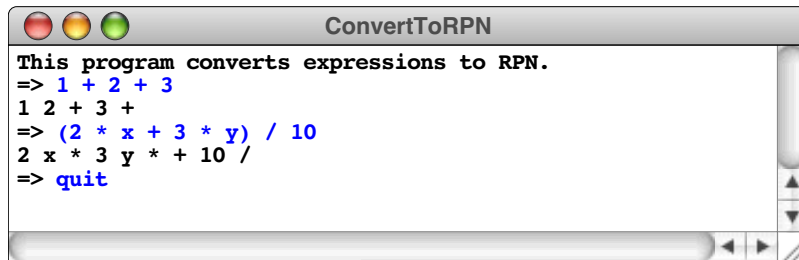


5. Write a function

```
bool ExpMatch(expressionT e1, expressionT e2);
```

that returns **true** if **e1** and **e2** are matching expressions, which means that they have exactly the same structure, the same operators, the same constants, and the same identifier names in the same order. If there are any differences at any level of the expression tree, your function should return **false**.

6. In the expression interpreter designed in the chapter, every operator is a binary operator in the sense that it takes two operands, one on each side. Most programming languages also allow unary operators, which take a single operand, which usually follows the operator. The most common example is the unary minus operation, as in the expression **-x**. Make whatever changes you need to make in the expression interpreter to add the unary minus operator.
7. Write a program that reads expressions from the user in their standard mathematical form and then writes out those same expressions using reverse Polish notation, in which the operators follow the operands to which they apply. (Reverse Polish notation, or RPN, was introduced in the discussion of the calculator in Chapter 4.) Your program should be able to duplicate this sample run:



8. Although the interpreter program that appears in this chapter is considerably easier to implement than a complete compiler, it is possible to get a sense of how a compiler works by defining one for a simplified computer system called a *stack machine*. A stack machine performs operations on an internal stack, which is maintained by the hardware, in much the same fashion as the calculator described in Chapter 4. For the purposes of this problem, you should assume that the stack machine can execute the following operations:

LOAD #<i>n</i>	Pushes the constant <i>n</i> on the stack.
LOAD <i>var</i>	Pushes the value of the variable <i>var</i> on the stack.
STORE <i>var</i>	Stores the top stack value in <i>var</i> without actually popping it.
DISPLAY	Pops the stack and displays the result.
ADD	These instructions pop the top two values from the stack and apply the indicated operation, pushing the final result back on the stack. The top value is the right operand, the next one down is the left.
SUB	
MUL	
DIV	

Write a function

```
void Compile(istream & in, ostream & out);
```

that reads expressions from **in** and writes to **out** a sequence of instructions for the stack-machine that have the same effect as evaluating each of the expressions in the input file and displaying their result. For example, if the file opened as **in** contains

```
x = 7
y = 5
2 * x + 3 * y
```

calling **Compile(in, out)** should write the following code to **out**:

```
LOAD #7
STORE x
DISPLAY
LOAD #5
STORE y
DISPLAY
LOAD #2
LOAD x
MUL
LOAD #3
LOAD y
MUL
ADD
DISPLAY
```

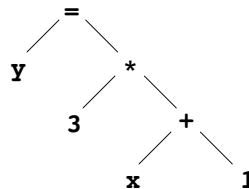
9. After it parses an expression, a commercial compiler typically looks for ways to simplify that expression so that it can be computed more efficiently. This process is called **optimization**. One common technique used in the optimization process is **constant folding**, which consists of identifying subexpressions that are composed entirely of constants and replacing them with their value. For example, if a compiler encountered the expression

```
days = 24 * 60 * 60 * sec
```

there would be no point in generating code to perform the first two multiplications when the program was executed. The value of the subexpression $24 * 60 * 60$ is constant and might as well be replaced by its value (86400) before the compiler actually starts to generate code.

Write a function **FoldConstants(exp)** that takes an expression pointer and returns a pointer to new expression in which any subexpressions that are entirely composed of constants are replaced by the computed value.

10. The process of turning the internal representation of an expression back into its text form is generally called **unparsing** the expression. Write a function **Unparse(exp)** that displays the expression **exp** on the screen in its standard mathematical form. Parentheses should be included in the output only if they are required by the precedence rules. Thus, the expression represented by the tree



should be unparsed as

y = 3 * (x + 1)

11. *Note: If you have not studied calculus, you should skip this exercise and the one that follows.* Using tree structures to represent expressions makes it possible to perform sophisticated mathematical operations by transforming the structure of the tree. For example, it is not very hard to write a function that differentiates an expression by applying the standard rules from calculus that allow you to express the derivative of a complex expression in terms of the derivatives of its parts. The most common rules for differentiating an expression involving the standard arithmetic operators are shown in Figure 14-10.

Write a recursive function **Differentiate(exp, var)** that uses the rules from Figure 14-10 to find the derivative of the expression **exp** with respect to the variable **var**. The result of the **Differentiate** function is an **Expression** that can be used in any context in which such values are legal. For example, you could evaluate it, unparse it, or pass it to **Differentiate** to calculate the second derivative of the original expression.

Figure 14-10 Standard formulas for differentiation

$x' = 1$	where:
$c' = 0$	x is the variable used as the basis for the differentiation
$(u + v)' = u' + v'$	c is a constant or variable that does not depend on x
$(u - v)' = u' - v'$	u and v are arbitrary expressions
$(uv)' = uv' + vu'$	n is an integer constant
$(u / v)' = \frac{uv' - vu'}{v^2}$	
$(u^n)' = nu^{n-1}u'$	

12. If you implement the **Differentiate** function from the preceding exercise in the most straightforward way, the expressions that you get back will be quite complex, even though they are mathematically correct. For example, if you apply the differentiation rules from Figure 14-10 to the expression

$$x^2 + 2x - 3$$

the result is

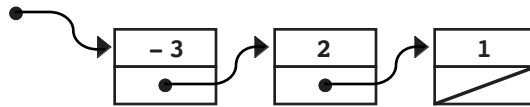
$$2 \times x^1 \times 1 + 2 \times 1 + x \times 0 - 0$$

To transform this expression into the more familiar

$$2x + 2$$

requires you to make several mathematical simplifications. For example, because you know that multiplying any value by 0 always gives back 0, you can eliminate the $x \times 0$ term from the expression.

One approach to simplifying expressions that turns out to be useful in the context of differentiating an arithmetic expression is to convert expressions into an equivalent polynomial form. Polynomials involving powers of x , for example, can be represented as a list of coefficients in which the first element in the list is the constant term, the next element is the coefficient of x , the next is the coefficient of x^2 , and so on. For example, the expression $x^2 + 2x - 3$, would be represented in polynomial form as follows:



Write a function

```
Expression *Simplify(Expression *exp)
```

that attempts to simplify an expression by converting it to a polynomial and then converting the result back into an expression. If the expression is too complex to convert to a simple polynomial, your function should return the original expression unchanged.

Chapter 15

Sets

Dear me, what a wonderfully mixed set!

— George Eliot, *Middlemarch*, 1871

Although much of Chapter 14 is concerned with the practical problem of implementing an expression parser, there is another lesson you should take from that chapter, which is that theory can have a profound effect on practice. In the case of the expression parser, the theory consists of using a formal grammar to define the syntactic structure of a programming language. Such situations arise often in computer science, which has strong theoretical foundations that have direct application to practical problems. Because theory is so central to computer science as a discipline, it is useful to learn about theory and practical techniques together.

In this chapter, you will learn about sets, which are central to both the theory and practice of computer science. The next section begins with an informal presentation of the underlying mathematical theory of sets. The rest of the chapter then turns to the more practical concern of how to implement sets as an abstract data type.

15.1 Sets as a mathematical abstraction

In all likelihood, you have already encountered sets at some point in your study of mathematics. Although the definition is not entirely precise, it is best to think of a **set** as an unordered collection of distinct elements. For example, the days of the week form a set of seven elements that can be written down as follows:

{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}

The individual elements are written in this order only because it is conventional. If you wrote these same names down in some other order, you would still have the same set. A set, however, never has multiple copies of the same element.

The set of weekdays is a **finite set** because it contains a finite number of elements. In mathematics, there are also **infinite sets**, such as the set of all integers. In a computer system, sets are usually finite, even if they correspond to infinite sets in mathematics. For example, the set of integers that a computer can represent in a variable of type `int` is finite because the hardware imposes a limit on the range of integer values.

To illustrate the fundamental operations on sets, it is important to have a few sets to use as a foundation. In keeping with mathematical convention, this text uses the following symbols to refer to the indicated sets:

- $\emptyset$ The **empty set**, which contains no elements
- Z** The set of all integers
- N** The set of **natural numbers**, which consists of the nonnegative integers
- R** The set of all real numbers

Following mathematical convention, this text uses uppercase letters to refer to sets. Sets whose membership is defined—like **N**, **Z**, and **R**—are denoted using boldface letters. Names that refer to some unspecified set are written using italic letters, such as *S* and *T*.

Membership

The fundamental property that defines a set is that of **membership**, which has the same intuitive meaning in mathematics that it does in English. Mathematicians indicate membership symbolically using the notation $x \in S$, which indicates that the value x is an element of the set S . For example, given the sets defined in the preceding section, the following statements are true:

$$17 \in \mathbf{N}$$

$$-4 \in \mathbf{Z}$$

$$\pi \in \mathbf{R}$$

Conversely, the notation $x \notin S$ indicates that x is *not* an element of S . For example, $-4 \notin \mathbf{N}$, because the set of natural numbers does not include the negative integers.

The membership of a set is typically specified in one of the two following ways:

- *Enumeration.* Defining a set by enumeration is simply a matter of listing its elements. By convention, the elements in the list are enclosed in curly braces and separated by commas. For example, the set **D** of single-digit natural numbers can be defined by enumeration as follows:

$$\mathbf{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

- *Rule.* You can also define a set by specifying a rule that distinguishes the members of that set. In most cases, the rule is expressed in two parts: a larger set that provides the potential candidates and some conditional expression that identifies the elements that should be selected for inclusion. For example, the set **D** from the preceding example can also be defined like this:

$$\mathbf{D} = \{x \mid x \in \mathbf{N} \text{ and } x < 10\}$$

If you read this definition aloud, it comes out sounding like this: “**D** is defined to be the set of all elements x such that x is a natural number and x is less than 10.”

Set operations

Mathematical set theory defines several operations on sets, of which the following are the most important:

- *Union.* The union of two sets, which is written as $A \cup B$, is the set of all elements belonging to the set A , the set B , or both.

$$\begin{aligned}\{1, 3, 5, 7, 9\} \cup \{2, 4, 6, 8\} &= \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ \{1, 2, 4, 8\} \cup \{2, 3, 5, 7\} &= \{1, 2, 3, 4, 5, 7, 8\} \\ \{2, 3\} \cup \{1, 2, 3, 4\} &= \{1, 2, 3, 4\}\end{aligned}$$

- *Intersection.* The intersection of two sets is written as $A \cap B$ and consists of the elements belonging to both A and B .

$$\begin{aligned}\{1, 3, 5, 7, 9\} \cap \{2, 4, 6, 8\} &= \emptyset \\ \{1, 2, 4, 8\} \cap \{2, 3, 5, 7\} &= \{2\} \\ \{2, 3\} \cap \{1, 2, 3, 4\} &= \{2, 3\}\end{aligned}$$

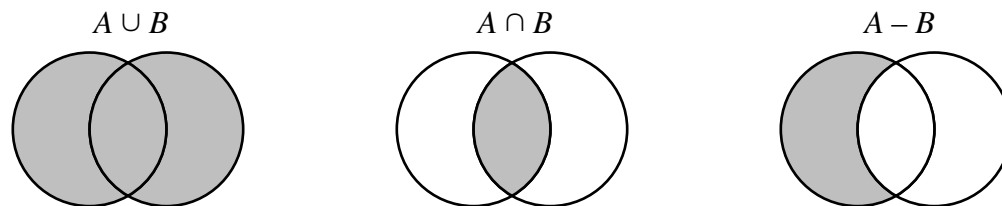
- *Set difference.* The difference of two sets is written as $A - B$ and consists of the elements belonging to A except for those that are also contained in B .

$$\begin{aligned}\{1, 3, 5, 7, 9\} - \{2, 4, 6, 8\} &= \{1, 3, 5, 7, 9\} \\ \{1, 2, 4, 8\} - \{2, 3, 5, 7\} &= \{1, 4, 8\} \\ \{2, 3\} - \{1, 2, 3, 4\} &= \emptyset\end{aligned}$$

In addition to set-producing operations like union and intersection, the mathematical theory of sets also defines several operations that determine whether some property holds between two sets. Operations that test a particular property are the mathematical equivalent of predicate functions and are usually called **relations**. The most important relations on sets are the following:

- *Equality.* The sets A and B are equal if they have the same elements. The equality relation for sets is indicated by the standard equal sign used to denote equality in other mathematical contexts. Thus, the notation $A = B$ indicates that the sets A and B contain the same elements.
- *Subset.* The subset relation is written as $A \subseteq B$ and is true if all the elements of A are also elements of B . For example, the set $\{2, 3, 5, 7\}$ is a subset of the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Similarly, the set $\mathbf{N}$ of natural numbers is a subset of the set $\mathbf{Z}$ of integers. From the definition, it is clear that every set is a subset of itself. Mathematicians use the notation $A \subset B$ to indicate that A is a **proper subset** of B , which means that the subset relation holds but that the sets are not equal.

Set operations are often illustrated by drawing **Venn diagrams**, which are named for the British logician John Venn (1834–1923). In a Venn diagram, the individual sets are represented as geometric figures that may overlap to indicate regions in which they share elements. For example, the results of the set operations union, intersection, and set difference are indicated by the shaded regions in the following Venn diagrams:



Identities on sets

One of the useful bits of knowledge you can derive from mathematical set theory is that the union, intersection, and difference operations are related to each other in various ways. These relationships are usually expressed as **identities**, which are rules indicating that two expressions are invariably equal. In this text, identities are written in the form

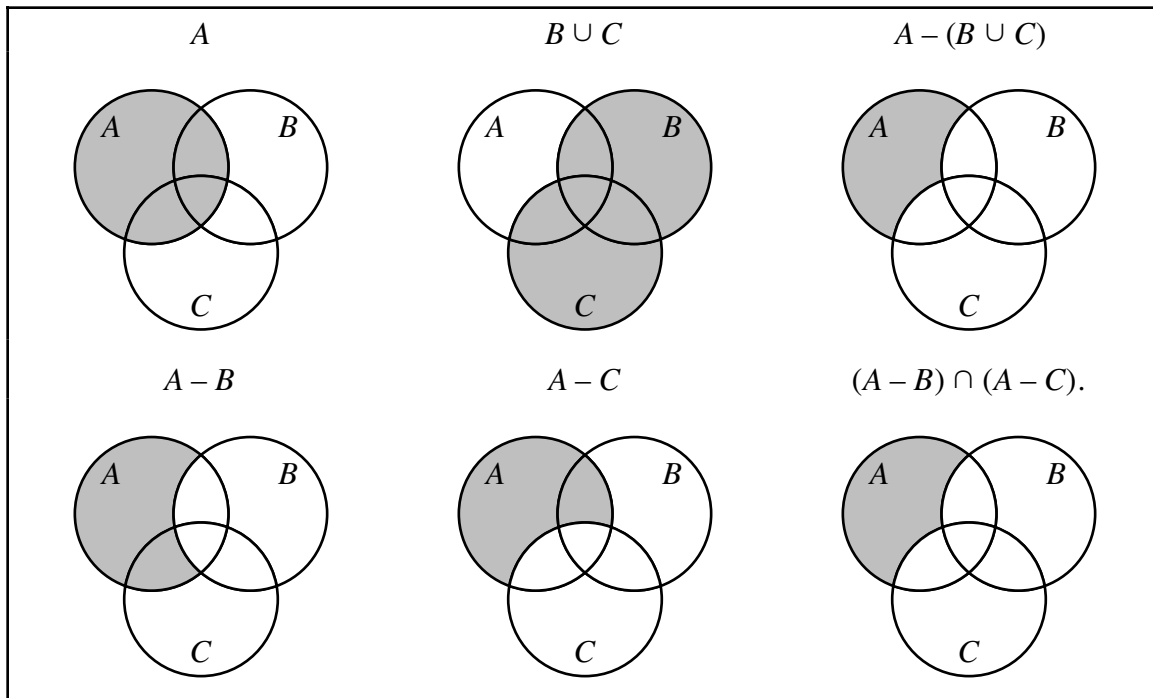
$$lhs \equiv rhs$$

which means that the set expressions lhs and rhs are always the same and can therefore be substituted for one another. The most common set identities are shown in Table 15-1.

You can get a sense of how these identities work by drawing Venn diagrams to represent individual stages in the computation. Figure 15-1, for example, verifies the first

Table 15-1 Fundamental identities on sets

$S \cup S \equiv S$ $S \cap S \equiv S$	<i>Idempotence</i>
$A \cap (A \cup B) \equiv A$ $A \cup (A \cap B) \equiv A$	<i>Absorption</i>
$A \cup B \equiv B \cup A$ $A \cap B \equiv B \cap A$	<i>Commutative laws</i>
$A \cup (B \cup C) \equiv (A \cup B) \cup C$ $A \cap (B \cap C) \equiv (A \cap B) \cap C$	<i>Associative laws</i>
$A \cap (B \cup C) \equiv (A \cap B) \cup (A \cap C)$ $A \cup (B \cap C) \equiv (A \cup B) \cap (A \cup C)$	<i>Distributive laws</i>
$A - (B \cup C) \equiv (A - B) \cap (A - C)$ $A - (B \cap C) \equiv (A - B) \cup (A - C)$	<i>DeMorgan's laws</i>

Figure 15-1 Illustration of DeMorgan's law using Venn diagrams

of DeMorgan's laws listed in Table 15-1. The shaded areas represent the value of each subexpression in the identity. The fact that the figures at the right have the same shaded region demonstrates that the set $A - (B \cup C)$ is the same as the set $(A - B) \cap (A - C)$.

What may still be unclear, however, is why you as a programmer might ever need to learn rules that at first seem so complex and arcane. Mathematical techniques are important to computer science for several reasons. For one thing, theoretical knowledge is useful in its own right because it deepens your understanding of the foundations of computing. At the same time, this type of theoretical knowledge often has direct application to programming practice. By relying on data structures whose mathematical properties are well established, you can use the theoretical underpinnings of those structures to your advantage. For example, if you write a program that uses sets as an abstract type, you may be able to simplify your code by applying one of the standard set identities shown in Table 15-1. The justification for making that simplification comes from the abstract theory of sets. Choosing to use sets as a programming abstraction, as opposed to designing some less formal structure of your own, makes it easier for you to apply theory to practice.

15.2 Designing a set interface

If you want to use sets as an abstract type in an application, you need to design an interface that exports the standard set operations in a way that clients will find easy to use. As with the abstract interfaces presented in earlier chapters, the `set.h` interface presumably exports a `Set` class to represent sets, and that class includes the methods needed to manipulate them. From your experience with interface design and your knowledge of how sets behave, you should expect that the `Set` class interface allows clients to perform the following operations:

- *Allocation and deallocation.* The **Set** class interface must make it possible to create new sets. The usual approach is to export a constructor that allocates an empty set and a destructor that allows you to free the storage associated with a set.
- *Adding and removing elements.* Once you have used the constructor to create a new set, you need to be able to add elements to it so that you can create a set containing the elements you need. For symmetry, and because the operation is required for many set algorithms, it is also convenient to be able to remove elements from a set. Thus, the **Set** class interface should include functions for adding and removing individual elements from an existing set.
- *Testing membership.* There must be a function corresponding to the membership operator $\in$ that allows you to determine whether a set contains a particular element.
- *Determining the number of elements.* In many applications, it helps to be able to determine the number of elements in a set, which mathematicians sometimes call its **cardinality**. Moreover, it often makes sense to have the **Set** class include a method **isEmpty** that makes it easy for clients to check for an empty set.
- *High-level set operations.* The **Set** class interface should export functions like **unionWith**, **intersect**, **subtract**, **equals**, and **isSubsetOf** that correspond to the fundamental mathematical operations on sets.
- *Iteration.* Given a set, it must be possible to step through the elements in the set, applying an operation as you go. To provide clients with this capability, the interface can use any of the strategies presented in Chapter 12 for maps. One approach, for example, would be to have the **Set** class export a mapping operation that invokes a client-supplied callback function on each element of a set. Another strategy is to use the iterator facility so that clients can iterate over elements in a set in exactly the same way that those clients can iterate over keys in a map. The **Set** class defined in this chapter adopts the iterator approach.

Defining the element type

Before writing out the complete interface, you need to resolve a very important issue: what type should you use to represent the elements in a set? As the designer of the interface, you need to make sure that your interface gives programmers the freedom to define sets containing the values they want to use. The element type may be strings in some situations, integers in others, and sets or other abstract types in a third. As you have seen several times before, the best solution to supporting this kind of flexibility is to define **Set** as a class template, leaving the specific element type as a template placeholder to be filled in by the client.

Writing the set interface

If you adopt the template strategy in the preceding section, you can easily write the public interface of **Set** class such as the one shown in Figure 15-2. Note that the constructor requires the client to specify a comparison function so the implementation can compare individual elements. As with the **BST** class from Chapter 13, the comparison function is an optional argument. If not given, the **Set** will use the default comparison function from **cmpfn.h** that compares two values using the built-in relational operators.

Character sets

One of the most common element types for sets is the built-in type **char**, which proves useful in a number of applications. For example, if you need to test whether a particular character is a legal operator in the context of the expression parser from Chapter 14, you can use the set package to construct a set of the legal operators and then use the **Set** method **contains** to determine whether the character in question is a member of that set.

Figure 15-2 Interface for a general set class template

```

/*
 * File: set.h
 * -----
 * This interface exports the Set class template, a generic
 * pattern for a set of data values. The data values can be of
 * any type, specialized through a template placeholder. The one
 * requirement on the type is that the client must supply a
 * a comparison function that compares two elements (or be willing
 * to use the default comparison function that relies on < and ==).
 */

#ifndef _set_h
#define _set_h

#include "cmpfn.h"
#include "bst.h"

/*
 * Class: Set
 * -----
 * This class is used to represent a set of values.
 */

template <typename ElemType>
class Set {
public:

/*
 * Constructor: Set
 * Usage: Set<int> set;
 *        Set<student> students(CompareStudentsById);
 *        Set<string> *sp = new Set<string>;
 * -----
 * The constructor initializes an empty set. The optional argument
 * to the constructor is a function pointer that is applied to
 * two elements to determine relative ordering. The comparison
 * function should return 0 if the two elements are equal, negative
 * result if first is "less than" second, and positive result if
 * first is "greater than" second. If no argument is supplied, the
 * OperatorCmp template is used as a default, which applies
 * < and == to the elements to determine ordering.
 */

        Set(int (*cmpFn)(ElemType, ElemType) = OperatorCmp);

/*
 * Destructor: ~Set
 * Usage: delete sp;
 * -----
 * The destructor frees the storage associated with set.
 */

        ~Set();

```

```
/*
 * Method: size
 * Usage: n = set.size();
 * -----
 * This function returns the number of elements in this set.
 */
    int size();

/*
 * Method: isEmpty
 * Usage: if (set.isEmpty()) . . .
 * -----
 * This function returns true if this set has no elements.
 */
    bool isEmpty();

/*
 * Method: add
 * Usage: set.add(element);
 * -----
 * This function adds a new element to an existing set.
 */
    void add(ElemType element);

/*
 * Method: remove
 * Usage: set.remove(element);
 * -----
 * This functions removes the element from the set, if it exists.
 */
    void remove(ElemType element);

/*
 * Method: contains
 * Usage: if (set.contains(element)) . . .
 * -----
 * This function returns true if the element is in the set.
 */
    bool contains(ElemType element);

/*
 * Functions: equals, isSubsetOf
 * Usage: if (s1.equals(s2)) . . .
 *         if (s1.isSubsetOf(s2)) . . .
 * -----
 * These predicate functions implement the equality and subset
 * relations on sets, respectively.  s1.equals(s2) returns
 * true if s1 and s2 have the same elements.  s1.isSubsetOf(s2)
 * returns true if all elements of s1 are also elements of s2.
 */
    bool equals(Set & otherSet);
    bool isSubsetOf(Set & otherSet);
```

```

/*
 * Methods: unionWith, intersect, subtract
 * Usage: s1.union(s2);
 *         s1.intersect(s2);
 *         s1.subtract(s2);
 * -----
 * These methods modify the receiver set as follows:
 * s1.unionWith(s2); Adds all elements from s2 to this set.
 * s1.intersect(s2); Removes any element not in s2 from this set.
 * s1.subtract(s2);  Removes all element in s2 from this set.
 */

    void unionWith(Set & otherSet);
    void intersect(Set & otherSet);
    void subtract(Set & otherSet);

/*
 * Method: iterator
 * Usage: iter = set.iterator();
 * -----
 * This method creates a new iterator that iterates
 * through the elements in this set. The elements are
 * accessed in sorted order, as defined by the comparison
 * callback function.
 */

    Iterator iterator();

private:

#include "setpriv.h"

};

#include "setimpl.cpp"

#endif

```

You can also use character sets as the basis for an implementation of the ANSI `<cctype>` interface, which allows you to determine whether a character falls into a particular class. To implement the functions using sets, the first step is to create several sets of characters, which correspond to the various predicate functions exported by the interface. For example, the following function initializes five sets of characters corresponding to the `<cctype>` functions `isdigit`, `islower`, `isupper`, `isalpha`, and `isalnum`, respectively:

```

void InitCTypeSets() {
    AssignSetFromString(digitSet, "0123456789");
    AssignSetFromString(lowerSet, "abcdefghijklmnopqrstuvwxyz");
    AssignSetFromString(upperSet, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    alphaSet = lowerSet;
    alphaSet.unionWith(upperSet);
    alnumSet = alphaSet;
    alnumSet.unionWith(digitSet);
}

```

```

void AssignSetFromString(Set<char> & set, string str) {
    for (int i = 0; i < str.length(); i++) {
        set.add(str[i]);
    }
}

```

Once these sets have been defined, the implementation of functions like **isdigit** become simple membership tests, as follows:

```

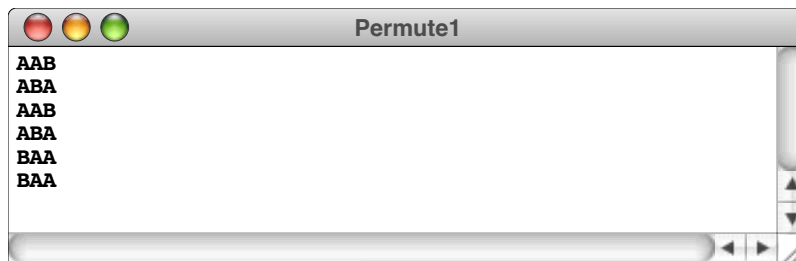
bool isdigit(int ch) {
    return digitSet.contains(ch);
}

```

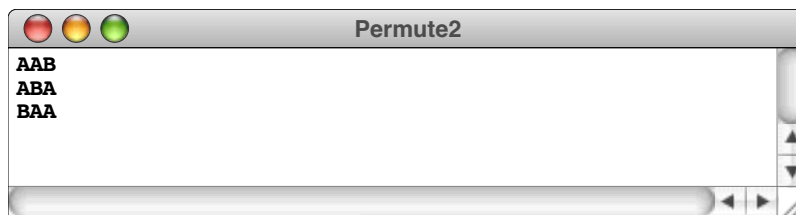
The other functions from **<cctype>** can be defined in much the same way.

Using sets to avoid duplication

To get a sense of how you might use sets containing other types, it helps to go back to the program from Chapter 6 that generates permutations. The simple recursive implementation of **ListPermutations** does not work well if there are repeated characters in the input string. For example, if you use the program as written to generate the permutations of the string "AAB", you get the following output:



Each of the permutations appears twice somewhere in the list. It would be better if the program listed each unique permutation only once, so the output looks like this instead:



You can write a program to produce this output by using sets to keep track of the permutations you have already encountered. Instead of displaying each permutation as you compute it, all you have to do is add it to a set. When the recursive decomposition is complete, you can simply go through that set, displaying each element in turn. Because a set contains only one copy of each element, the duplicates are automatically eliminated.

In the context of the permutation program, the elements of the set are strings. The revised code for **ListPermutations** and **RecursivePermute** appears in Figure 15-3.

15.3 Implementing the set class

Writing an implementation for the **Set** class interface is considerably easier than you might expect, mostly because you can layer the implementation on top of the binary search tree facility from Chapter 13. If you think about the operations that the **BST** class

Figure 15-3 Implementation of ListPermutations that works with repeated characters

```

/*
 * Function: ListPermutations
 * Usage: ListPermutations(str);
 * -----
 * This function lists all permutations of the characters in the
 * string str. If the same string is generated more than once
 * in the course of the algorithm, each of those permutations is
 * listed only once.
 */

void ListPermutations(string str) {
    Set<string> set;
    RecursivePermute("", str, set);
    Set<string>::Iterator iter = set.iterator();
    while (iter.hasNext()) {
        cout << iter.next() << endl;
    }
}

/*
 * Function: RecursivePermute
 * Usage: RecursivePermute(pre, rest, set);
 * -----
 * This function implements the recursive permutation algorithm,
 * adding each permutation to the set as it goes.
 */

void RecursivePermute(string prefix, string rest, Set<string> & set) {
    if (rest == "") {
        set.add(prefix);
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string newPrefix = prefix + rest[i];
            string newRest = rest.substr(0, i) + rest.substr(i+1);
            RecursivePermute(newPrefix, newRest, set);
        }
    }
}

```

exports, you will quickly discover that they correspond closely to the operations you need for sets. Testing for membership in a set is comparable to finding a node in a binary search tree. Similarly, the insertion and deletion operations behave in much the same way for both structures, because keys in a binary search tree—like elements in a set—are unique in the sense that a given key will never appear more than once. Thus, if you use a binary search tree to store the elements of a set, you can use the **BST** methods **find**, **add**, and **remove** to implement the corresponding set operations. The only definition required for the **setpriv.h** file is therefore:

```
BST<ElemType> bst;
```

Because most of the work necessary to implement sets is left to the **BST** class, the implementation of the set package itself is quite straightforward. The complete code for the set class appears in Figure 15-4.

Figure 15-4 Implementation of the set class based on binary search trees

```
/*
 * File: setimpl.cpp
 * -----
 * This file implements the set class template defined in set.h.
 */

#ifdef _set_h

template <typename ElemType>
Set<ElemType>::Set(int (*cmp)(ElemType, ElemType)) : bst(cmp) {
    /* Empty */
}

template <typename ElemType>
Set<ElemType>::~~Set() {
    /* Empty */
}

template <typename ElemType>
int Set<ElemType>::size() {
    return bst.size();
}

template <typename ElemType>
bool Set<ElemType>::isEmpty() {
    return bst.isEmpty();
}

template <typename ElemType>
void Set<ElemType>::add(ElemType element) {
    bst.add(element);
}

template <typename ElemType>
void Set<ElemType>::remove(ElemType element) {
    bst.remove(element);
}

template <typename ElemType>
bool Set<ElemType>::contains(ElemType element) {
    return bst.find(element) != NULL;
}

}
```

```

/*
 * Implementation notes: Set operations
 * -----
 * The functions isSubsetOf, unionWith, intersect, and subtract
 * are similar in structure. Each one uses an iterator to walk over
 * the appropriate set.
 */

template <typename ElemType>
bool Set<ElemType>::equals(Set & otherSet) {
    return isSubsetOf(otherSet) && otherSet.isSubsetOf(*this);
}

template <typename ElemType>
bool Set<ElemType>::isSubsetOf(Set & otherSet) {
    Iterator iter = iterator();
    while (iter.hasNext()) {
        if (!otherSet.contains(iter.next())) return false;
    }
    return true;
}

template <typename ElemType>
typename Set<ElemType>::Iterator Set<ElemType>::iterator() {
    return bst.iterator();
}

template <typename ElemType>
void Set<ElemType>::unionWith(Set & otherSet) {
    Iterator iter = otherSet.iterator();
    while (iter.hasNext()) {
        add(iter.next());
    }
}

template <typename ElemType>
void Set<ElemType>::intersect(Set & otherSet) {
    Iterator iter = iterator();
    while (iter.hasNext()) {
        ElemType elem = iter.next();
        if (!otherSet.contains(elem)) {
            remove(elem);
        }
    }
}

template <typename ElemType>
void Set<ElemType>::subtract(Set & otherSet) {
    Iterator iter = otherSet.iterator();
    while (iter.hasNext()) {
        remove(iter.next());
    }
}

#endif

```

It is also useful to note that the higher-level set operations such as **unionWith** and **subtract** are written using the iterator facility described in the interface. In pseudocode, these functions look like this:

```
void Set<ElemType>::unionWith(Set & otherSet) {
    for (each element x in otherSet) {
        Add x to this.
    }
}

void Set<ElemType>::subtract(Set & otherSet) {
    for (each element x in otherSet) {
        Remove x from this.
    }
}
```

You can easily translate the **for** statements in the pseudocode formulation into a **while** loop that uses the iterator structure. For example, the loop in the **unionWith** function ends up looking like this in the finished code:

```
Iterator iter = otherSet.iterator();
while (iter.hasNext()) {
    add(iter.next());
}
```

15.4 Enhancing the efficiency of integer sets

The implementation of the set abstraction shown in Figure 15-4 does not include any optimizations for integer sets. Even so, it is important to know that there are more efficient strategies for representing integer sets as long as you can restrict the elements of the set to integers in some predefined range. In many applications, sets naturally obey this constraint. For example, if you are using integer sets to represent sets of characters, the values will never be outside the range [0, 255]. If the element type is a typical enumeration type, the set of legal values will be restricted to an even smaller range.

Characteristic vectors

Suppose for the moment that you are working with a set whose elements will always lie between 0 and **RANGE_SIZE** – 1, where **RANGE_SIZE** is a constant that specifies the size of the range to which element values are restricted. You can represent such sets efficiently by using an array of Boolean values. The value at index position *k* in the array indicates whether the integer *k* is in the set. For example, if **elements[4]** has the value **true**, then 4 is in the set represented by the Boolean array **elements**. Similarly, if **elements[5]** is **false**, then 5 is not an element of that set. Boolean arrays in which the elements indicate whether the corresponding index is a member of some set are called **characteristic vectors**. The following examples illustrate how the characteristic-vector strategy can be used to represent the indicated sets, assuming that **RANGE_SIZE** has the value 10:

$\emptyset$									
F	F	F	F	F	F	F	F	F	F
0	1	2	3	4	5	6	7	8	9

{1, 3, 5, 7, 9}									
F	T	F	T	F	T	F	T	F	T
0	1	2	3	4	5	6	7	8	9

x

0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0

 $\sim x$

1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1

In programming, applying the $\sim$ operation is called **taking the complement** of its operand.

The operators $\ll$ and $\gg$ shift the bits in their left operand the number of positions specified by their right operand. The only difference between the two operations is the direction in which the shifting occurs. The $\ll$ operator shifts bits to the left; the $\gg$ operator shifts them to the right. Thus, the expression $x \ll 1$ produces a new value in which every bit in the value of x is shifted one position to the left, as follows:

x

0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0

 $x \ll 1$

0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0

Similarly, the expression $y \gg 2$ produces a value in which the bits in y have been shifted two positions to the right, like this:

y

0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1

 $y \gg 2$

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

As long as the value being shifted is unsigned, bits that are shifted past the end of the word disappear and are replaced on the opposite end by 0 bits. If the value being shifted is signed, the behavior of the shift operators depends on the underlying characteristics of the hardware. For this reason, it is good practice to restrict your use of the shift operators to unsigned values, thereby increasing the portability of your code.

Bit patterns such as those used in the preceding examples correspond to integers, which are represented internally in their binary form. The value shown for the variable x , for example, represents the integer 42, which you can determine by adding up the place values of the digits, each of which accounts for twice as much as the bit on its right.

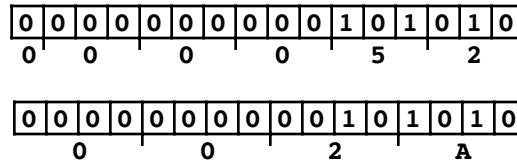
x

0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0

↑	×	1	=	0
↑	×	2	=	2
↑	×	4	=	0
↑	×	8	=	8
↑	×	16	=	0
↑	×	32	=	32
				42

When you work with bit patterns, however, it is best not to regard them as base-10 integers, because it is hard to recognize the underlying sequence of bits when given an integer in its decimal form. In programming, it is much more common to express bit patterns in **octal** (base 8) or **hexadecimal** (base 16). The advantage of using either of these bases is that doing so allows you to convert integers directly into their underlying bit patterns one digit at a time. Because both 8 and 16 are powers of two, you can translate an octal or a hexadecimal integer into its binary form by replacing each digit in the integer with the appropriate bit pattern, as shown in Table 15-3.

In C++, you can write integer constants in either octal or hexadecimal notation. To specify an octal constant, all you do is begin the number with the digit **0**. To specify a hexadecimal constant in C++, you begin with the prefix **0x** followed by the digits that make up the number. Since hexadecimal notation requires 16 digits, the digits corresponding to the numbers 10 through 15 are indicated using the letters **A** through **F**. For example, the decimal integer 42 can be written as either the octal constant **052** or the hexadecimal constant **0x2A**, as illustrated by the following diagrams:



Implementing characteristic vectors using the bitwise operators

The bitwise operators introduced in the preceding section make it possible to implement operations on characteristic vectors in a highly efficient way. If you want to test the state of an individual bit in a characteristic vector, all you have to do is create a value that has a 1 bit in the desired position and 0 bits everywhere else. Such a value is called a **mask** because you can use it to hide all the other bits in the word. If you apply the **&** operator to the word in the characteristic vector that contains the bit you're trying to find and the mask that corresponds to the correct bit position, all the other bits in that word will be stripped away, leaving you with a value that reflects the state of the desired bit.

To make this strategy more concrete, it helps to define the structure of a characteristic vector in more detail. You can define the type **cVectorT** as an array of machine words interpreted as a sequence of bits.

```
struct cVectorT {
    unsigned long words[CVEC_WORDS];
};
```

where **CVEC_WORDS** is a constant defined as follows:

```
const int BITS_PER_BYTE = 8;
const int BITS_PER_LONG = (BITS_PER_BYTE * sizeof(long));
const int CVEC_WORDS = ((RANGE_SIZE + BITS_PER_LONG - 1) /
    BITS_PER_LONG);
```

Given this structure, you can test a specific bit in a characteristic vector using the function **CVectorTest**, which has the following implementation:

Table 15-3 Bit patterns for the octal and hexadecimal digits

Octal		Hexadecimal			
0	000	0	0000	8	1000
1	001	1	0001	9	1001
2	010	2	0010	A	1010
3	011	3	0011	B	1011
4	100	4	0100	C	1100
5	101	5	0101	D	1101
6	110	6	0110	E	1110
7	111	7	0111	F	1111

particular bit in a word by applying the logical OR operation to the old value of that word and a mask containing the desired bit. You can clear a bit by applying the logical AND operation to the old value of the word and the complement of the mask. These operations are illustrated by the following definitions of the functions **CVectorSet** and **CVectorClear**:

```
void CVectorSet(cVectorT & cv, int k) {
    if (k < 0 || k >= RANGE_SIZE) {
        Error("CVectorSet: Value is out of range");
    }
    cv.words[k / BITS_PER_LONG] |= BitMask(k);
}

void CVectorClear(cVectorT & cv, int k) {
    if (k < 0 || k >= RANGE_SIZE) {
        Error("CVectorClear: Value is out of range");
    }
    cv.words[k / BITS_PER_LONG] &= ~BitMask(k);
}
```

Implementing the high-level set operations

Packing characteristic vectors into the bits in a word certainly saves a large amount of space. As it happens, this same strategy also improves the efficiency of the high-level set operations like **unionWith**, **intersect**, and **subtract**. The trick is to compute each word in the new characteristic vector using a single application of the appropriate bitwise operator.

As an example, the union of two sets consists of all elements that belong to either of its arguments. If you translate this idea into the realm of characteristic vectors, it is easy to see that any word in the characteristic vector of the set $A \cup B$ can be computed by applying the logical OR operation to the corresponding words in the characteristic vectors for those sets. The result of the logical OR operation has a 1 bit in those positions in which either of its operands has a 1 bit, which is exactly what you want to compute the union.

This approach is illustrated by the function **CVectorUnion**, which creates a new characteristic vector whose value corresponds to the union of the characteristic vectors passed in as arguments. The code for **CVectorUnion** looks like this:

```
cVectorT CVectorUnion(cVectorT cv1, cVectorT cv2) {
    cVectorT result;
    for (int i = 0; i < CVEC_WORDS; i++) {
        result.words[i] = cv1.words[i] | cv2.words[i];
    }
    return result;
}
```

The functions **CVectorIntersection** and **CVectorDifference** can be implemented similarly; the details are left to you as a programming exercise.

Using a hybrid implementation

You can apply the characteristic vector approach only to sets whose elements fall into a limited range. On the other hand, there is no reason that clients of a general set package necessarily have to be aware of this distinction. If you are implementing the set package, you can adopt a hybrid approach that uses characteristic vectors as long as the values stay

Table 15-4 Mathematical notations for sets

Empty set	$\emptyset$	The set containing no elements
Membership	$x \in S$	True if x is an element of S
Nonmembership	$x \notin S$	True if x is not an element of S
Equality	$A = B$	True if A and B contain exactly the same elements
Subset	$A \subseteq B$	True if all elements in A are also in B
Proper subset	$A \subset B$	True if A is a subset of B but the sets are not equal
Union	$A \cup B$	The set of elements in either A , B , or both
Intersection	$A \cap B$	The set of elements in both A and B
Set difference	$A - B$	The set of elements in A that are not also in B

in range. As soon as the client adds an element to the set that is outside the range allowed by the characteristic-vector representation, the implementation automatically converts the set to use the more general form offered by binary search trees. As a result, clients who only use integers in the restricted range get the enhanced performance associated with the characteristic vector strategy. On the other hand, clients who need to define sets containing integers outside the optimal range can still use the same interface.

Summary

In this chapter, you have learned about sets, which are important to computer science as both a theoretical and a practical abstraction. The fact that sets have a well-developed mathematical foundation—far from making them too abstract to be useful—increases their utility as a programming tool. Because of that theoretical foundation, you can count on sets to exhibit certain properties and obey specific rules. By coding your algorithms in terms of sets, you can build on that same theoretical base and construct systems whose behavior is easier to predict and understand.

Important points in this chapter include:

- A set is an unordered collection of distinct elements. The set operations used in this book appear in Table 15-4, along with their mathematical symbols.
- The interactions among the various set operators are often easier to understand if you keep in mind certain identities that indicate that two set expressions are invariably equal. Using these identities can also improve your programming practice, because they provide you with tools to simplify set operations appearing in your code.
- The set class is straightforward to implement because much of it can be layered on top of the `bst.h` interface from Chapter 13, which defines a class template for binary search trees.
- To implement most of the algorithms that apply to sets, it must be possible to step through the elements of a set one at a time. One of the best ways to provide this capability is to define an iterator that works for sets in the same way that the iterator facility in Chapter 12 works for maps.
- Sets of integers can be implemented very efficiently using arrays of Boolean data called *characteristic vectors*. If you use the bitwise operators provided by C++, you can pack characteristic vectors into a small number of machine words and perform such set operations as union and intersection on many elements of the vector at a time.

Review questions

1. True or false: The elements of a set are unordered, so the set $\{3, 2, 1\}$ and the set $\{1, 2, 3\}$ represent the same set.
2. True or false: A set can contain multiple copies of the same element.
3. What sets are denoted by each of the following symbols: $\emptyset$, $\mathbf{Z}$, $\mathbf{N}$, and $\mathbf{R}$?
4. What do the symbols $\in$ and $\notin$ mean?
5. Use an enumeration to specify the elements of the following set:

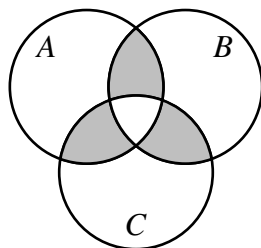
$$\{x \mid x \in \mathbf{N} \text{ and } x \leq 100 \text{ and } \sqrt{x} \in \mathbf{N}\}$$

6. Write a rule-based definition for the following set:

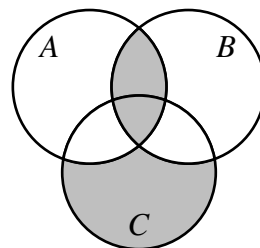
$$\{0, 9, 18, 27, 36, 45, 54, 63, 72, 81\}$$
7. What are the mathematical symbols for the operations union, intersection, and set difference?
8. Evaluate the following set expressions:
 - a. $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \cup \{\mathbf{a}, \mathbf{c}, \mathbf{e}\}$
 - b. $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \cap \{\mathbf{a}, \mathbf{c}, \mathbf{e}\}$
 - c. $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\} - \{\mathbf{a}, \mathbf{c}, \mathbf{e}\}$
 - d. $(\{\mathbf{a}, \mathbf{b}, \mathbf{c}\} - \{\mathbf{a}, \mathbf{c}, \mathbf{e}\}) \cup (\{\mathbf{a}, \mathbf{b}, \mathbf{c}\} - \{\mathbf{a}, \mathbf{c}, \mathbf{e}\})$
9. What is the difference between a subset and a proper subset?
10. Give an example of an infinite set that is a proper subset of some other infinite set.
11. For each of the following set operations, draw Venn diagrams whose shaded regions illustrate the contents of the specified set expression:
 - a. $A \cup (B \cap C)$
 - b. $(A - C) \cap (B - C)$
 - c. $(A - B) \cup (B - A)$
 - d. $(A \cup B) - (A \cap B)$

12. Write set expressions that describe the shaded region in each of the following Venn diagrams:

a.



b.



13. Draw Venn diagrams illustrating each of the identities in Table 15-1.

14. What is the cardinality of a set?
15. Why is the **Set** designed as a class template?
16. What argument must be passed to the **Set** constructor?
17. How is the **Set** class used in the implementation of the **ListPermutations** function that appears in Figure 15-3?
18. The general implementation of the **Set** class uses a data structure from an earlier chapter to represent the elements of a set. What is that structure? What properties make that structure useful for this purpose?
19. Once you have created a **Set** object, how can you step through the elements of that set?
20. Why is it useful to use a similar iterator facility for maps, binary search trees, and sets?
21. What is a characteristic vector?
22. What restrictions must be placed on a set in order to use characteristic vectors as an implementation strategy?
23. Assuming that **RANGE_SIZE** has the value 10, diagram the characteristic vectors for the following sets:
 - a. $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
 - b. $\{5\}$
24. What set is represented by the following characteristic vector:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

By consulting the ASCII chart in Table 1-1, identify the function in **<cctype>** to which this set corresponds.

25. In the diagrams used to represent characteristic vectors (such as the one in the preceding exercise), the type **unsigned long** is shown as taking 32 bits. Suppose that you are using a machine in which this type is represented using 64 bits instead. Does the code given in the chapter continue to work? Why or why not?
26. Suppose that the variables **x** and **y** are of type **unsigned** and contain the following bit patterns:

x 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 1

y 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

Expressing your answer as a sequence of bits, compute the value of each of the following expressions:

a. $x \& y$	f. $x \& \sim y$
b. $x y$	g. $\sim x \& \sim y$
c. $x \wedge y$	h. $y \gg 4$
d. $x \wedge x$	i. $x \ll 3$
e. $\sim x$	j. $(x \gg 8) \& y$

27. Express the values of x and y from the preceding exercise as constants using both octal and hexadecimal notation.
28. Suppose that the variables x and $mask$ are both declared to be of type **unsigned**, and that the value of $mask$ contains a single 1 bit in some position. What expressions would you use to accomplish each of the following operations:
- Test the bit in x corresponding to the bit in $mask$ to see whether it is nonzero.
 - Set the bit in x corresponding to the bit in $mask$.
 - Clear the bit in x corresponding to the bit in $mask$.
 - Complement the bit in x corresponding to the bit in $mask$.
29. Write an expression that constructs a mask of type **unsigned** in which there is a single 1 bit in bit position k , where bits are numbered from 0 starting at the right end of the word. For example, if k is 2, the expression should generate the following mask:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Programming exercises

1. In order to write test programs for the set package, it is useful to have some facility for performing input and output operations. On the input side, for example, you would like to have a function

```
Set<int> GetIntSet();
```

that returns a set of integers entered by the user. On the output side, it would be nice to have a function

```
void PrintIntSet(Set<int> & set);
```

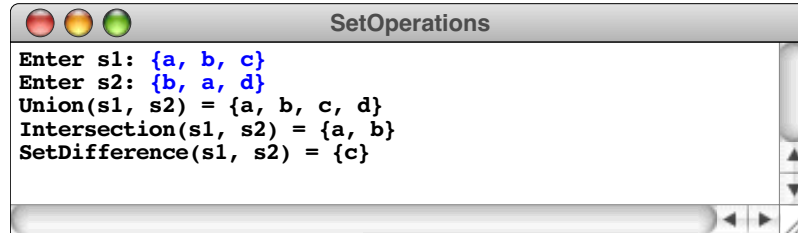
that displays the contents of the specified integer set on the screen.

Implement the functions **GetIntSet** and **PrintIntSet**. For each of these functions, the set should appear in its traditional form, with the elements enclosed in curly braces and separated by commas. The empty set should be represented as an empty pair of curly braces. Your implementation of **GetIntSet** should ignore the spacing of the input, which is easy to do if you use the **Scanner** class with the **IgnoreSpaces** option.

2. Using the preceding exercise as a model, implement the functions

```
Set<string> GetStringSet();
void PrintStringSet(Set<string> & set);
```

which perform input and output operations for sets of strings. For **GetStringSet**, the input values should be individual string tokens as defined by the scanner abstraction. Use these functions to write a simple test program that reads in two sets of strings and then displays their union, intersection, and difference, as shown in this sample run:



3. Write a function

```
Set<int> PrimeSet(int max)
```

that returns a set of the prime numbers between 2 and **max**. A number N is prime if it has exactly two divisors, which are always 1 and the number N itself. Checking for primality, however, doesn't require you to try every possible divisor. The only numbers you need to check are the prime numbers between 2 and the square root of N . As it tests whether a number is prime, your code should make use of the fact that all potential factors must be in the set of primes you are constructing.

4. The implementation of the general **Set** class presented in this chapter uses binary search trees to ensure that the set operations are executed with reasonable efficiency. If efficiency is not a relevant concern, you can implement the same abstraction much more easily by representing a set as a linked list of its elements, sorted by the appropriate comparison function. Implement the **Set** class using this representation strategy.
5. Write a function **SetCompare(s1, s2)** that compares the sets **s1** and **s2** according to the following rules:
 - In order to be comparable, **s1** and **s2** must have the same element type.
 - If two sets have a different number of elements, the smaller set is always considered to be less than the larger one. Thus, if **s1** is the set {2, 3} and **s2** is the set {0, 1, 2}, calling **SetCompare(s1, s2)** should return -1.
 - If two sets have the same number of elements, **SetCompare** should compare elements in the two sets in the order specified by their comparison function, which is also the order in which iteration occurs. If the elements ever differ, the set containing the smaller of the two values is considered to be less than the set that has the larger value. For example, if **s1** is the set {1, 3, 4} and **s2** is the set {1, 2, 3}, calling **SetCompare(s1, s2)** should return +1, because the integer 3 in **s1** is larger than the corresponding element in **s2**, which is the integer 2.
 - If **s1** and **s2** have exactly the same elements, **SetCompare** should return 0.
6. Because the elements of a set can be any values for which a comparison function exists, you can use the set package to create sets containing other sets as elements. Write a function template

```
template <typename ElemType>  
Set<Set<ElemType> > PowerSet(Set<ElemType> &s);
```

that returns the **power set** of the set **s**, which is defined as the set of all subsets of **s**. For example, if **s** is the set {**a**, **b**, **c**}, calling **PowerSet(s)** should return the following set:

{{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}

7. Write a function template

```
template <typename ElemType>
void AddArrayToSet(Set<ElemType> &set, ElemType arr[], int n);
```

that adds the first **n** elements in **arr** to the specified set. Such a function is extremely useful in initializing a set whose elements are known at compile time, as illustrated by the following code, which initializes **articleSet** to contain the strings "a", "an", and "the":

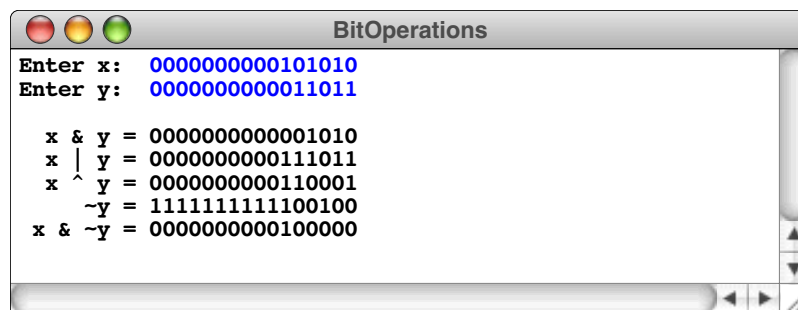
```
string articleArray[] = { "a", "an", "the" };
int nArticles = sizeof articleArray / sizeof articleArray[0];
Set<string> articleSet;

AddArrayToSet(articleSet, articleArray, nArticles);
```

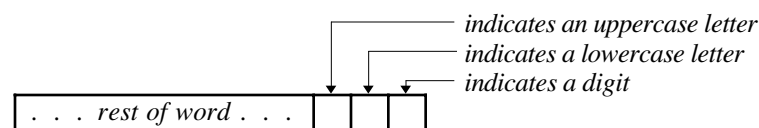
8. Write a program that implements the following procedure:

- Read in two strings, each of which represents a sequence of bits. These strings must consist only of the characters **0** and **1** and must be exactly 16 characters long.
- Convert each of these strings into a value of type **unsigned** with the same internal pattern of bits. Assume that the variables used to store the converted result are named **x** and **y**.
- Display the value of each of the following expressions as a sequence of 16 bits: **x & y**, **x | y**, **x ^ y**, **~y**, **x & ~y**.

The operation of this program is illustrated by the following sample run:



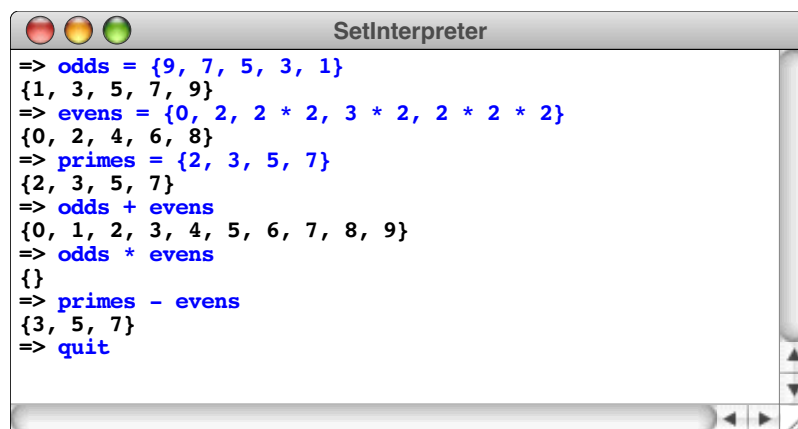
9. On most computer systems, the ANSI **<cctype>** interface introduced in Chapter 3 is implemented using the bitwise operators. The strategy is to use specific bit positions in a word to indicate properties that a character might have. For example, imagine that the three bits at the right end of a word are used to indicate whether a character is a digit, a lowercase letter, or an uppercase letter, as shown in this diagram:



If you create an array consisting of 256 of these words—one for each character—you can implement the functions from **<cctype>** so that each function requires selecting the appropriate element of the array selection, applying one of the bitwise operators, and testing the result.

Use this strategy to implement a simplified version of the **<cctype>** interface that exports the functions **isdigit**, **islower**, **isupper**, **isalpha**, and **isalnum**. In your implementation, it is important to make sure that the code for **isalpha** and **isalnum** requires no more operations than the other three functions do.

10. The function **CVectorUnion** described in the section on “Implementing the high-level set operations” shows how to use the bitwise operators as the basis for a highly efficient implementation of the union operation for sets. Using **CVectorUnion** as a model, implement the functions **CVectorIntersection** and **CVectorDifference**.
11. Extend the implementation of the set package so that it uses the strategy described in the section on “Using a hybrid implementation.” The basic idea is that the constructor creates a set whose internal representation uses a characteristic vector consisting of 256 bits. As long as no elements are added that lie outside the range [0, 255], the implementation can continue to use that representation. However, if the client attempts to add an element to an integer set that is outside this range, the implementation of **add** must convert the internal representation to use the more general binary search tree form. Except for the fact that doing so may take some time, the operation of converting the internal representation of the set from one form to another should be invisible to the client.
12. Extend the expression interpreter from Chapter 14 so that it supports sets of integers as a separate data type. When they are used with sets as arguments, the operators **+**, *****, and **-** should compute the union, intersection, and set difference, respectively. Sets are specified in the traditional way, which means that you need to extend the grammar used by the parser to support braces that enclose a comma-separated list of expressions. A sample run of the program might look like this:



```

=> odds = {9, 7, 5, 3, 1}
{1, 3, 5, 7, 9}
=> evens = {0, 2, 2 * 2, 3 * 2, 2 * 2 * 2}
{0, 2, 4, 6, 8}
=> primes = {2, 3, 5, 7}
{2, 3, 5, 7}
=> odds + evens
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
=> odds * evens
{}
=> primes - evens
{3, 5, 7}
=> quit

```

Note that the computation involving integers is still legal and can be used in any expression context, including the values used to specify the elements of a set.

Chapter 16

Graphs

*So I draw the world together link by link:
Yea, from Delos up to Limerick and back!*

— Rudyard Kipling, “The Song of
the Banjo,” from *Verses*, 1894

Many structures in the real world consist of a set of values connected by a set of links. Such a structure is called a **graph**. Common examples of graphs include cities connected by highways, computers connected by network links, and courses in a college curriculum connected by prerequisites. Programmers typically refer to the individual elements—such as the cities, computers, and courses—as **nodes** and the interconnections—the highways, network connections, and prerequisites—as **arcs**, although mathematicians tend to use the terms **vertex** and **edge** instead.

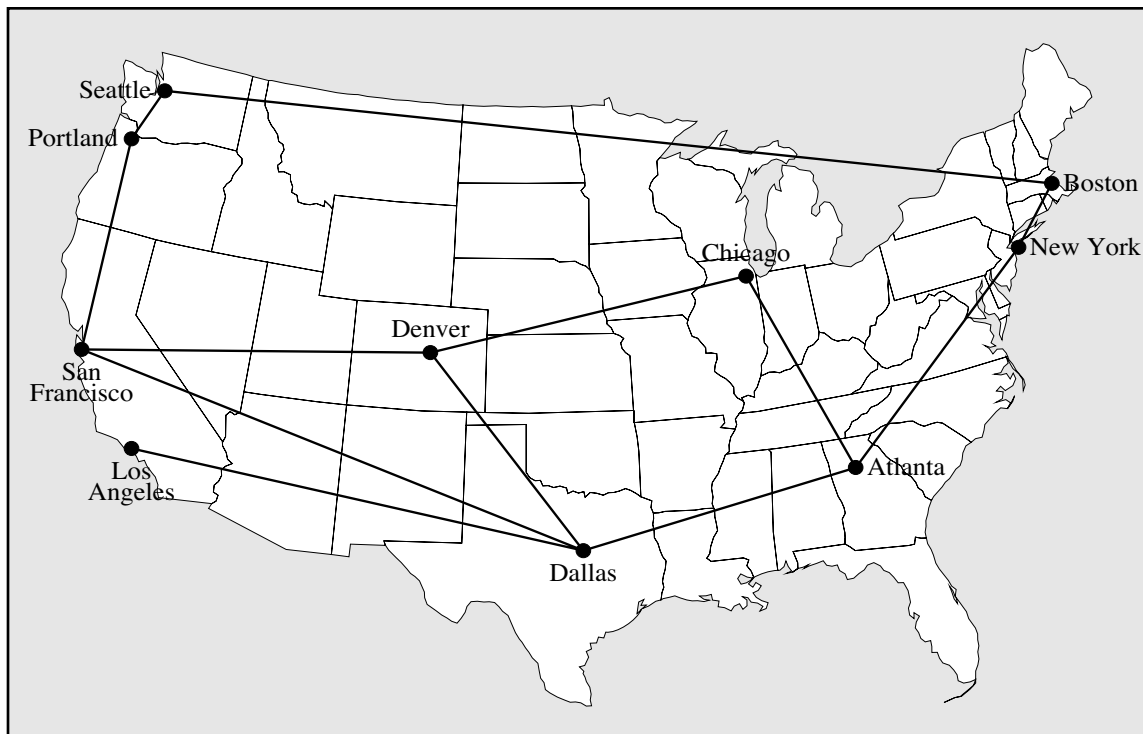
Because they consist of nodes connected by a set of links, graphs are clearly similar to trees, which were introduced in Chapter 13. In fact, the only difference is that there are fewer restrictions on the structure of the connections in a graph than there are in a tree. The arcs in a graph, for example, often form cyclical patterns. In a tree, cyclical patterns are illegal because of the requirement that every node must be linked to the root by a unique line of descent. Because trees have restrictions that do not apply to graphs, graphs are a more general type that includes trees as a subset. Thus, every tree is a graph, but there are some graphs that are not trees.

In this chapter, you will learn about graphs from both a practical and a theoretical perspective. Learning to work with graphs as a programming tool is useful because they come up in a surprising number of contexts. Mastering the theory is extremely valuable as well, because doing so often makes it possible to find much more efficient solutions to problems with considerable practical importance.

16.1 The structure of a graph

The easiest way to get a sense of the structure of a graph is to consider a simple example. Suppose that you work for a small airline that serves 10 major cities in the United States with the routes shown in Figure 16-1. The labeled circles represent cities—Atlanta,

Figure 16-1 Route map for a small airline serving 10 cities



The diagrams in this text represent directed graphs only if the arcs include an arrow indicating their direction. If the arrows are missing—as they are in the airline graph in Figure 16-1—you can assume the graph is undirected.

Arcs in a directed graph are specified using the notation *start*→*finish*, where *start* and *finish* are the nodes on each side of the directed arc. Thus, the triangular route shown in the preceding diagram consists of the following arcs:

San Francisco→**Dallas**
Dallas→**Denver**
Denver→**San Francisco**

Arcs in an undirected graph are sometimes denoted using a double-headed arrow (↔). Although this notation is easy enough to read, you don't really need a separate symbol for an undirected arc, because you can always represent it as a pair of directed arcs. For example, if a graph contains a bidirectional arc **Portland**↔**Seattle**, you can represent the fact that each city is connected to the other by including both **Portland**→**Seattle** and **Seattle**→**Portland** in the set of arcs. If you use this approach, you can represent the undirected airline graph in Figure 16-1 as a directed graph with the following mathematical structure:

$V = \{ \text{Atlanta, Boston, Chicago, Dallas, Denver, Los Angeles, New York, Portland, San Francisco, Seattle} \}$
 $E = \{ \text{Atlanta} \rightarrow \text{Chicago, Atlanta} \rightarrow \text{Dallas, Atlanta} \rightarrow \text{New York, Boston} \rightarrow \text{New York, Boston} \rightarrow \text{Seattle, Chicago} \rightarrow \text{Atlanta, Chicago} \rightarrow \text{Denver, Dallas} \rightarrow \text{Atlanta, Dallas} \rightarrow \text{Denver, Dallas} \rightarrow \text{Los Angeles, Dallas} \rightarrow \text{San Francisco, Denver} \rightarrow \text{Chicago, Denver} \rightarrow \text{Dallas, Denver} \rightarrow \text{San Francisco, Los Angeles} \rightarrow \text{Dallas, New York} \rightarrow \text{Atlanta, New York} \rightarrow \text{Boston, Portland} \rightarrow \text{San Francisco, Portland} \rightarrow \text{Seattle, San Francisco} \rightarrow \text{Dallas, San Francisco} \rightarrow \text{Denver, San Francisco} \rightarrow \text{Portland, Seattle} \rightarrow \text{Boston, Seattle} \rightarrow \text{Portland} \}$

Because it is always possible to simulate undirected graphs using directed ones, most graph packages—including the one introduced in this chapter—define a single graph type that supports directed graphs. If you want to define an undirected graph, all you have to do is create two arcs for every connection, one in each direction.

Paths and cycles

The arcs in a graph represent direct connections, which correspond to nonstop flights in the airline example. The fact that there is no explicit arc **San Francisco**→**New York** in the example graph does not mean that you cannot travel between those cities on this airline. If you wanted to fly from San Francisco to New York, you could use any of the following routes:

San Francisco→**Dallas**→**Atlanta**→**New York**
San Francisco→**Denver**→**Chicago**→**Atlanta**→**New York**
San Francisco→**Portland**→**Seattle**→**Boston**→**New York**

A sequence of arcs that allow you to move from one node to another is called a **path**. A path that begins and ends at the same node, such as the path

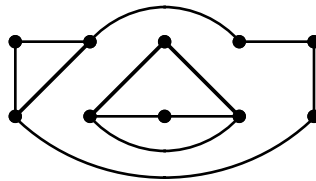
Dallas→**Atlanta**→**Chicago**→**Denver**→**Dallas**

is called a **cycle**. A **simple path** is a path that contains no duplicated nodes. Similarly, a **simple cycle** is a cycle that has no duplicated nodes other than the common node that appears at both the beginning and the end.

Nodes in a graph that are connected directly by an arc are called **neighbors**. If you count the number of neighbors for a particular node, that number is called the **degree** of that node. In the airline graph, for example, **Dallas** has degree 4 because it has direct connections to four cities: **Atlanta**, **Denver**, **Los Angeles**, and **San Francisco**. By contrast, **Los Angeles**, has degree 1 because it connects only to **Dallas**.

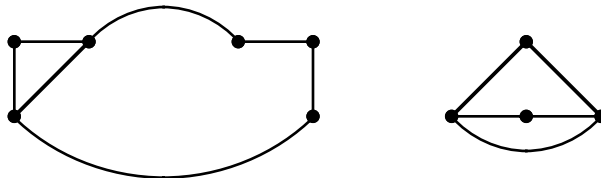
Connectivity

An undirected graph is **connected** if there is a path from each node to every other node. For example, the airline graph in Figure 16-1 is connected according to this rule. The definition of a graph, however, does not require that all nodes be connected in a single unit. For example, the graph

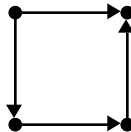


is an example of an unconnected graph, because no path links the cluster of four nodes in the interior of the diagram to any of the other nodes.

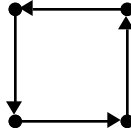
Given any undirected graph, you can always decompose it into a unique set of subgraphs in which each subgraph is connected, but no arcs lead from one subgraph to another. These subgraphs are called the **connected components** of the graph. The connected components of the preceding graph diagram look like this:



For directed graphs, the concept of connectivity is somewhat more complicated. If a directed graph contains a path connecting every pair of nodes, the graph is **strongly connected**. A directed graph is **weakly connected** if eliminating the directions on the arcs creates a connected graph. For example, the graph



is not strongly connected because you cannot travel from the node on the lower right to the node on the upper left moving only in the directions specified by the arcs. On the other hand, it is weakly connected because the undirected graph formed by eliminating the arrows is a connected graph. If you reverse the direction of the top arc, the resulting graph



is strongly connected.

16.2 Implementation strategies for graphs

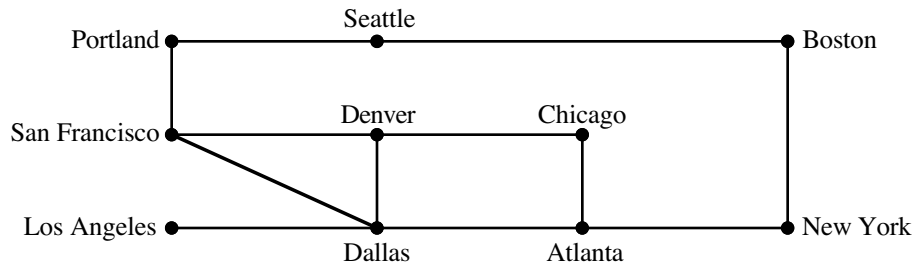
Like most abstract structures, graphs can be implemented in several different ways. The primary feature that differentiates these implementations is the strategy used to represent connections between nodes. In practice, the most common strategies are:

- Storing the connections for each node in an *adjacency list*.
- Storing the connections for the entire graph in an *adjacency matrix*.
- Storing the connections for each node as a *set of arcs*.

These representation strategies are described in greater detail in the sections that follow.

Representing connections using an adjacency list

The simplest way to represent connections in a graph is to store within the data structure for each node a list of the nodes to which it is connected. This structure is called an **adjacency list**. For example, in the now-familiar airline graph



the adjacency lists for each node look like this:

```

AdjacencyList(Atlanta)      = (Chicago, Dallas, New York)
AdjacencyList(Boston)      = (New York, Seattle)
AdjacencyList(Chicago)     = (Atlanta, Denver)
AdjacencyList(Dallas)      = (Atlanta, Denver, Los Angeles)
AdjacencyList(Denver)      = (Chicago, Dallas, San Francisco)
AdjacencyList(Los Angeles) = (Dallas)
AdjacencyList(New York)    = (Atlanta, Boston)
AdjacencyList(Portland)    = (San Francisco, Seattle)
AdjacencyList(San Francisco) = (Dallas, Denver, Portland)
AdjacencyList(Seattle)     = (Boston, Portland)
    
```

Representing connections using an adjacency matrix

Although lists provide a convenient way to represent the connections in a graph, they can be inefficient when an operation requires searching through the list of arcs associated with a node. For example, if you use the adjacency list representation, determining whether two nodes are connected requires $O(D)$ time, where D represents the degree of the originating node. If the nodes in a graph all have a small number of neighbors, the cost of searching through this list is small. If, however, the nodes in a graph tend to have a large number of neighbors, this cost becomes more significant.

If efficiency becomes a concern, you can reduce the cost of checking for connections to constant time by representing the arcs in a two-dimensional array called an **adjacency matrix** that shows which nodes are connected. The adjacency matrix for the airline graph looks like this:

	Atlanta	Boston	Chicago	Dallas	Denver	Los Angeles	New York	Portland	San Francisco	Seattle
Atlanta			x	x			x			
Boston							x			x
Chicago	x				x					
Dallas	x				x	x			x	
Denver			x	x					x	
Los Angeles				x						
New York	x	x								
Portland									x	x
San Francisco				x	x			x		
Seattle		x						x		

For an undirected graph of this sort, the adjacency matrix is **symmetric**, which means that the entries match when they are reflected across the main diagonal, which is shown in the figure as a dotted line.

To use the adjacency matrix approach, you must associate each node with an index number that specifies the column or row number in that table corresponding to that node. As part of the concrete structure for the graph, the implementation needs to allocate a two-dimensional grid with one row and one column for each node in the graph. The elements of the array are Boolean values. If the entry in `matrix[start][finish]` is **true**, there is an arc $start \rightarrow finish$ in the graph.

In terms of execution time, using an adjacency matrix is considerably faster than using an adjacency list. On the other hand, a matrix requires $O(N^2)$ storage space, where N is the number of nodes. For most graphs, the adjacency list representation tends to be more efficient in terms of space, although this is not necessarily the case. In the adjacency list representation, each node has a list of connections, which, in the worst case, will be $D_{\max}$ entries long, where $D_{\max}$ is the maximum degree of any node in the graph, which is therefore the maximum number of arcs emanating from a single node. The space cost for adjacency lists is therefore $O(N \times D_{\max})$. If most of the nodes are connected to each other, $D_{\max}$ will be relatively close to N , which means that the cost of representing connections is comparable for the two approaches. If, on the other hand, the graph contains many nodes but relatively few interconnections, the adjacency list representation can save considerable space.

Although the dividing line is never precisely defined, graphs for which the value of $D_{\max}$ is small in comparison to N are said to be **sparse**. Graphs in which $D_{\max}$ is comparable to N are considered **dense**. Often, the algorithms and representation strategies you use for graphs depend on whether you expect those graphs to be sparse or dense. The analysis in the preceding paragraph, for example, shows that the list representation is likely to be more appropriate for sparse graphs; if you are working with dense graphs, the matrix representation may well be a better choice.

Representing connections using a set of arcs

The motivation behind the third strategy for representing connections in a graph comes from the mathematical formulation of a graph as a set of nodes coupled with a set of arcs. If you were content to store no more information with each node other than its name, you could define a graph as a pair of sets, as follows:

```
struct graphT {  
    Set<string> nodes;  
    Set<string> arcs;  
};
```

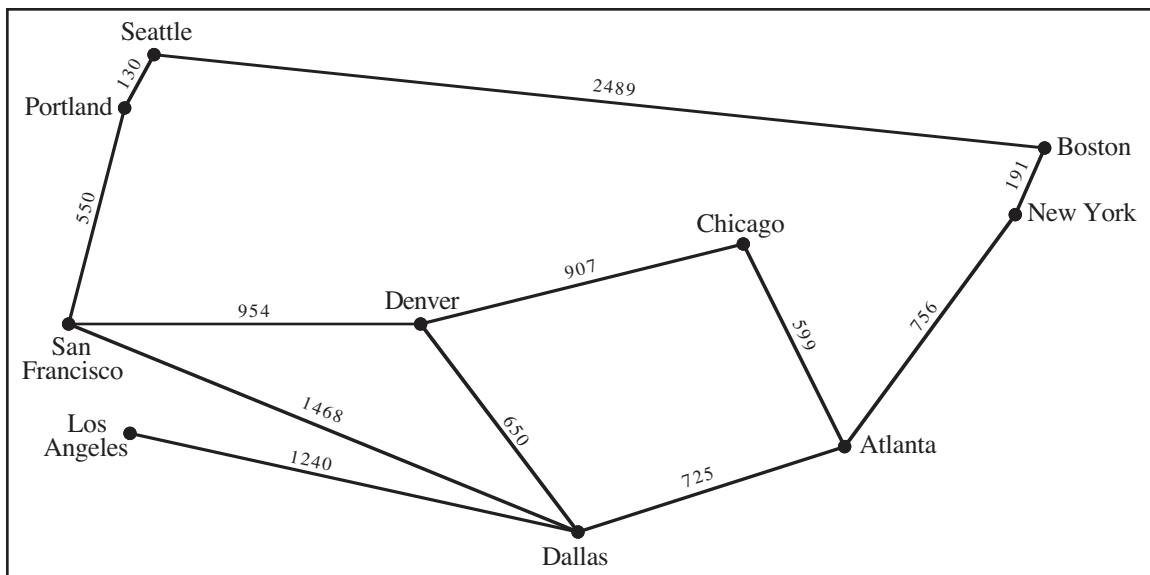
The set of nodes contains the names of every node in the graph. The set of arcs contains pairs of node names connected in some way that makes it easy to separate the node names representing the beginning and end of each arc. For example, you could adopt the convention that the keys in the **arcs** set consist of the names of the two cities separated by the characters **->**. Thus, to add the connection **Chicago→Atlanta** to the arcs for a graph **g**, you would use a statement like this:

```
g.arcs.add("Chicago->Atlanta");
```

The primary advantages of this representation are its conceptual simplicity and the fact that it mirrors so precisely the mathematical definition. It does, however, have certain important limitations. For one thing, finding the neighbors for any particular node require iterating through all the nodes in the entire graph. For another, most applications need to associate additional data with the individual nodes and arcs. As an example, many graph algorithms assign a numeric value to each of the arcs that represents either its distance or the cost of traveling along that route. Figure 16-2, for example, labels each arc in the airline graph with the distance between the endpoints. You could use this information to implement a frequent-flier program that assigns points to travelers based on the number of miles they fly.

Fortunately, neither of these problems is particularly difficult to solve. The most important change is to generalize the definition of **graphT** so that it uses new data types to represent nodes and arcs. In addition, the desire to streamline the process of iterating

Figure 16-2 Route map with the corresponding mileage associated with each arc



through the neighbors of a node means that each node must contain a set of the arcs that connect it to other nodes.

Given the fact that C++ is an object-oriented language, you would expect that graphs, nodes, and arcs would be represented as objects, with a new class definition for each level of the hierarchy. That design is certainly appropriate for this problem, and one possible implementation along these lines appears in section 16.5. The following section, however, introduces each of these types as structures rather than classes. There are two reasons behind that decision. First, using structures results in a simpler implementation that makes it easier to focus on the high-level operations rather than the details of object representation. Second, low-level structures are frequently used in practice because the contexts in which graphs arise vary so widely that it is hard to fit those applications into a common framework. For that reason, it often makes sense to add the relevant parts of the graph abstraction to the underlying implementation of some other data structure. If you do so, the code will probably resemble the structure-based implementation more closely than the object-based implementation presented later in the chapter.

16.3 Designing a low-level graph abstraction

The goal of this section is to design the data structures for a low-level graph package in which the three levels of the hierarchy—the graph as a whole, the individual nodes, and the arcs that connect those nodes—are represented using C++ structure types. In keeping with the naming convention for types introduced in Chapter 2, the most natural names for these types are **graphT**, **nodeT**, and **arcT**. At a minimum, those types will contain the following information:

- The **graphT** structure must specify the set of nodes in the graph and the complete set of arcs. Given that nodes have names, it is useful if the **graphT** structure also includes a map that allows clients to translate from names into the corresponding node structure.
- Each **nodeT** must specify the name of the node and the set of arcs leaving that node.
- Each **arcT** must specify the nodes connected by that arc along with the cost of traversing that arc in completing some path.

This informal description of the data types provides almost enough information to write the structure definitions, but there is one important consideration that needs to be included as part of the design process. The **graphT** structure conceptually “contains” **nodeT** values not only as part of its set of nodes but also as components of the elements in the set of arcs. Similarly, **arcT** values appear in two places because both the **graphT** and **nodeT** structure specify a set of arcs. In each case, the nodes and arcs that appear in different parts of these structures must be identical whenever they refer to the same entity in the abstract structure. For example, the **nodeT** corresponding to **Atlanta** must be the *same nodeT* no matter whether it appears in the top-level set of nodes or in one of the internal arcs. They cannot simply be copies of one another, because in that case changes to one copy would not be reflected in the others.

The critical implication of this observation is that the sets and structures used to represent the graph cannot contain **nodeT** and **arcT** values directly. The need to share common structures means that all of the internal references to these structures must specify *pointers* to **nodeT** and **arcT** values. The sets in the **graphT** structure, therefore, must use **nodeT *** and **arcT *** as their element type and not the underlying structure types themselves. The same is true for the set of arcs in the **nodeT** structure and the references to nodes in the **arcT** structure. Figure 16-3 shows the structure of a low-level graph interface that takes these considerations into account.

Figure 16-3 The low-level graph.h interface

```

/*
 * File: graph.h
 * -----
 * This file defines the interface for a simple graph package that
 * uses the structure types graphT, nodeT, and arcT.
 */

#ifndef _graph_h
#define _graph_h

#include "map.h"
#include "set.h"

struct nodeT;      /* Forward references to these two types so */
struct arcT;       /* that the C++ compiler can recognize them. */

/*
 * Type: graphT
 * -----
 * This type represents a graph and consists of two sets -- a set
 * of nodes and a set of arcs -- along with a map that creates an
 * association between names and nodes.
 */

struct graphT {
    Set<nodeT *> nodes;
    Set<arcT *> arcs;
    Map<nodeT *> nodeMap;
};

/*
 * Type: nodeT
 * -----
 * This type represents an individual node and consists of the
 * name of the node and the set of arcs from this node.
 */

struct nodeT {
    string name;
    Set<arcT *> arcs;
};

/*
 * Type: arcT
 * -----
 * This type represents an individual arc and consists of pointers
 * to the endpoints, along with the cost of traversing the arc.
 */

struct arcT {
    nodeT *start;
    nodeT *finish;
    double cost;
};

#endif

```

As the definitions in Figure 16-3 make clear, all of the data that defines the structure of the graph is stored in the form of sets whose elements are either nodes or arcs. These sets, moreover, are represented using the parameterized **Set** class from Chapter 15. One advantage of doing so is that the data structure closely parallels the mathematical formulation of a graph, which is defined in terms of sets. Building the **graph.h** interface on top of the **set.h** framework also has significant advantages in terms of simplifying the implementation.

When you define one abstraction in terms of another—as in the current proposal to define graphs in terms of sets—the resulting abstractions are said to be **layered**. Layered abstractions have a number of advantages. For one thing, they are usually easy to implement because much of the work can be relegated to the existing, lower-level interface. For example, defining the **graph.h** interface in terms of sets eliminates the need to define a separate iteration facility for graphs, because sets already support iteration. Thus, if you wanted to iterate over the nodes in a graph whose address was stored in the pointer variable **g**, all you would need to write is

```
foreach (nodeT *node in g->nodes) {  
    code to process an individual node  
}
```

In addition to simplifying the process of iteration, defining graphs in terms of sets makes it possible to apply higher-level set operations like union and intersection. Theoretical computer scientists often formulate graph algorithms in terms of these operations, and having them available to clients often makes those algorithms easier to code.

Using the low-level **graph.h** interface

Unlike the interface files you’ve seen so far in this book, the **graph.h** file—as it currently stands—introduces three structure types but no classes or methods. As a result, it does not require an implementation, so there is no need for a **graph.cpp** file as yet. The fact that the interface does not provide a suite of methods for working with graphs, nodes, and arcs means that clients have to define their own tools to create the required data structure. For example, the code in Figure 16-4 defines a method to create the airline graph from Figure 16-2, along with several helper methods that make it easier to create the graph structure.

Once you have created a graph, you can write code to work with the underlying graph data by using set operations. For example, the following code displays a list of all the cities served by the airline, along with the list of cities one can reach via a direct flight:

```
int main() {  
    graphT *airline = CreateAirlineGraph();  
    foreach (nodeT *node in airline->nodes) {  
        cout << node->name << " -> ";  
        bool commaNeeded = false;  
        foreach (arcT *arc in node->arcs) {  
            if (commaNeeded) cout << ", ";  
            cout << arc->finish->name;  
            commaNeeded = true;  
        }  
        cout << endl;  
    }  
    return 0;  
}
```

Figure 16-4 Code to initialize the airline graph

```

graphT *CreateAirlineGraph() {
    graphT *g = new graphT;
    AddCity(g, "Atlanta");
    AddCity(g, "Boston");
    AddCity(g, "Chicago");
    AddCity(g, "Dallas");
    AddCity(g, "Denver");
    AddCity(g, "Los Angeles");
    AddCity(g, "New York");
    AddCity(g, "Portland");
    AddCity(g, "San Francisco");
    AddCity(g, "Seattle");
    AddFlight(g, "Atlanta", "Chicago", 599);
    AddFlight(g, "Atlanta", "Dallas", 725);
    AddFlight(g, "Atlanta", "New York", 756);
    AddFlight(g, "Boston", "New York", 191);
    AddFlight(g, "Boston", "Seattle", 2489);
    AddFlight(g, "Chicago", "Denver", 907);
    AddFlight(g, "Dallas", "Denver", 650);
    AddFlight(g, "Dallas", "Los Angeles", 1240);
    AddFlight(g, "Dallas", "San Francisco", 1468);
    AddFlight(g, "Denver", "San Francisco", 954);
    AddFlight(g, "Portland", "Seattle", 130);
    AddFlight(g, "Portland", "San Francisco", 550);
    return g;
}

/* Adds a node for the specified city */

void AddCity(graphT *g, string name) {
    nodeT *node = new nodeT;
    node->name = name;
    g->nodes.add(node);
    g->nodeMap[name] = node;
}

/* Adds an arc in each direction between the cities c1 and c2. */

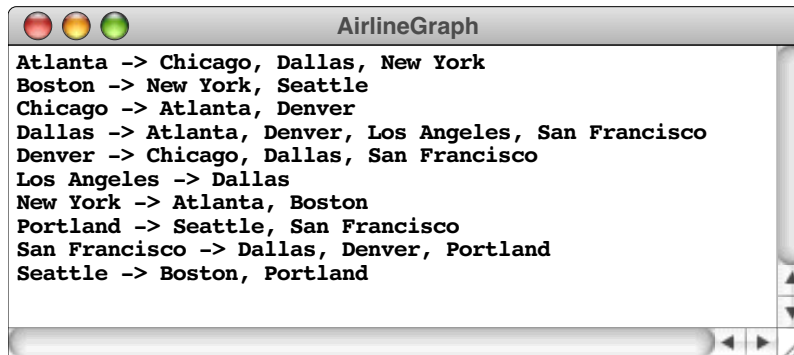
void AddFlight(graphT *g, string c1, string c2, int miles) {
    nodeT *n1 = g->nodeMap[c1];
    nodeT *n2 = g->nodeMap[c2];
    AddArc(g, n1, n2, miles);
    AddArc(g, n2, n1, miles);
}

/* Adds a new arc to the graph g that connects n1 to n2. */

void AddArc(graphT *g, nodeT *n1, nodeT *n2, double cost) {
    arcT *arc = new arcT;
    arc->start = n1;
    arc->finish = n2;
    arc->cost = cost;
    g->arcs.add(arc);
    n1->arcs.add(arc);
}

```

Given the code to initialize the airline graph from Figure 16-4, the output from this program looks like this:



If you think carefully about the output shown in this sample run, the fact that the output is neatly presented in alphabetical order might come as a surprise. The sets on which the graph is built are, at least in their mathematical form, unordered collections. The **Set** class that implements that underlying mathematical abstraction allows clients to specify a comparison function that defines the ordering relationship for the binary search tree used in the set implementation. In this example, however, no comparison function appears, which means that the **nodeT** and **arcT** pointers appear in their default order, which is defined by the memory addresses at which those structures appear. Since memory addresses bear no relation to the city names, the alphabetical ordering of the output is something of a mystery.

The reason for this apparently mysterious behavior is that most C++ runtime systems allocate heap memory in the order that requests come in. The initialization code in Figure 16-4 creates the cities and connection information in alphabetical order, which means that the node for the second city appears at a higher memory address than the node for the first. Thus, the fact that the output is so nicely ordered is actually something of a coincidence. As you will see in section 16.5, it is possible to extend the graph structure to ensure that these sets are ordered alphabetically by name.

16.4 Graph traversals

As you saw in the preceding example, it is easy to cycle through the nodes in a graph, as long as you are content to process the nodes in the order imposed by the set abstraction. Many graph algorithms, however, require you to process the nodes in an order that takes the connections into account. Such algorithms typically start at some node and then advance from node to node by moving along the arcs, performing some operation on each node. The precise nature of the operation depends on the algorithm, but the process of performing that operation—whatever it is—is called **visiting** the node. The process of visiting each node in a graph by moving along its arcs is called **traversing** the graph.

In Chapter 13, you learned that several traversal strategies exist for trees, of which the most important are preorder, postorder, and inorder traversals. Like trees, graphs also support more than one traversal strategy. For graphs, the two fundamental traversal algorithms are called *depth-first search* and *breadth-first search*, which are described in the next two sections.

To make the mechanics of the algorithms easier to understand, the implementations of depth- and breadth-first search assume that the client has supplied a function called **Visit** that takes care of whatever processing is required for each individual node. The goal of a traversal is therefore to call **Visit** once—and only once—on every node in an order

determined by the connections. Because graphs often have many different paths that lead to the same node, ensuring that the traversal algorithm does not visit the same node many times requires additional bookkeeping to keep track of which nodes have already been visited. To do so, the implementations in the next two sections define a set of nodes called **visited** to keep track of the nodes that have already been processed.

Depth-first search

The depth-first strategy for traversing a graph is similar to the preorder traversal of trees and has the same recursive structure. The only additional complication is that graphs—unlike trees—can contain cycles. If you don't check to make sure that nodes are not processed many times during the traversal, the recursive process can go on forever as the algorithm proceeds.

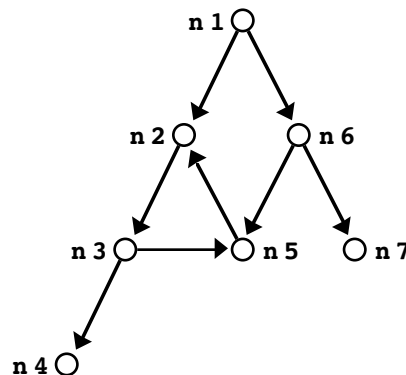
The algorithm to perform depth-first search starting at a particular node looks like this:

```
void DepthFirstSearch(nodeT *node) {
    Set<nodeT *> visited;
    RecDepthFirstSearch(node, visited);
}

void RecDepthFirstSearch(nodeT *node, Set<nodeT *> & visited) {
    if (visited.contains(node)) return;
    Visit(node);
    visited.add(node);
    foreach (arcT *arc in node->arcs) {
        RecDepthFirstSearch(arc->finish, visited);
    }
}
```

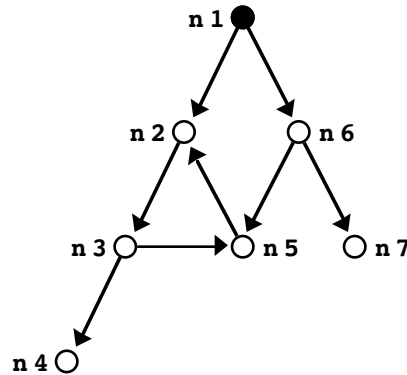
The **DepthFirstSearch** function itself is a wrapper function whose only purpose is to introduce the **visited** set used to keep track of nodes that have already been processed. **RecDepthFirstSearch** visits the current node and then calls itself recursively for each node directly accessible from the current one.

The depth-first strategy is most easily understood by tracing its operation in the context of a simple example, such as the following directed graph:

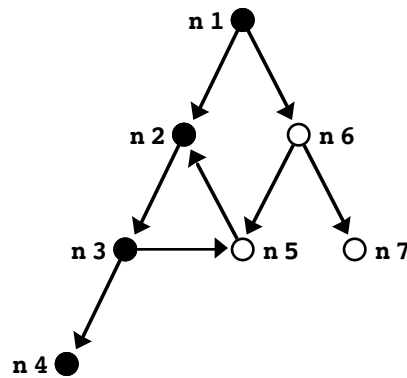


The open circles in the diagram indicate nodes that have not yet been visited. When the depth-first traversal algorithm calls **visit** to process a node, the diagram records this fact by using a filled circle for that node.

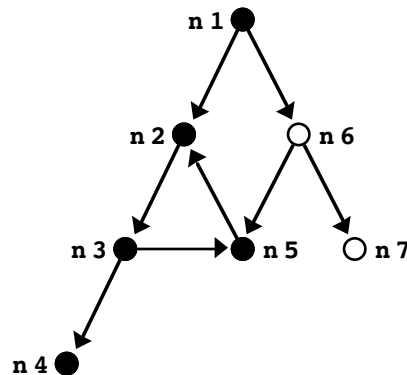
Suppose that the process begins by calling **DepthFirstSearch(n1)**. The first step is to visit **n1**, which leads to the following configuration:



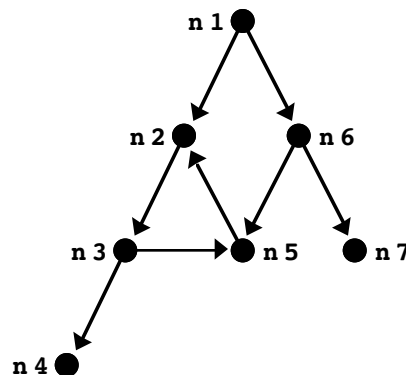
At this point, the **RecDepthFirstSearch** function enters a loop that cycles through the arcs emanating from **n1**. For each arc, the implementation calls itself recursively on the destination node. Because each level of recursive call follows the same recursive pattern, the overall effect of the depth-first strategy is to explore a single path in the graph as far as possible before backtracking to complete the exploration of paths at higher levels. As a result, assuming that connections are processed in alphabetical order by node name, the recursive calls will begin by exploring the left-hand branch to its end, as follows:



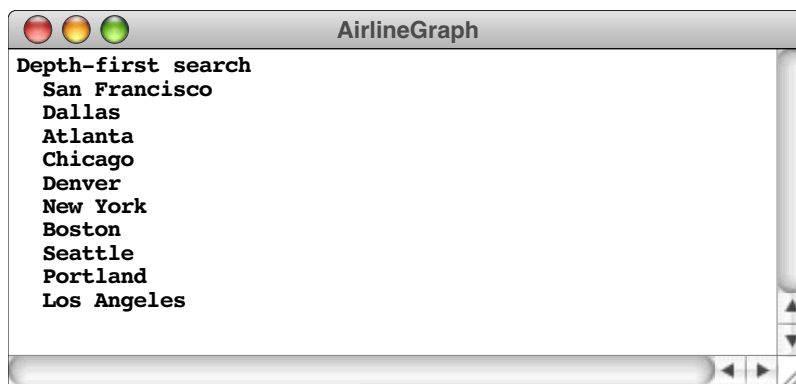
Because there are no arcs leading out of **n4**, the recursive descent through the graph ends at this point and returns to the previous level, at which it was processing **n3**. The recursive call to **RecDepthFirstSearch** then visits every node accessible along that path. The process, however, must stop if a node has already been visited, to guard against the possibility of infinite loops. For example, when the code follows the arcs leaving **n5**, it calls **RecDepthFirstSearch** on **n2**. Because this node is in the **visited** set, the call to **RecDepthFirstSearch** returns immediately to the following configuration:



From this point, the execution walks backward through the execution stack all the way to the **n1** node. Calling **DepthFirstSearch** on the only remaining path from the **n1** node marks **n6** and **n7**, at which point the traversal is complete, as follows:



As another example, suppose that you wanted to apply depth-first search to the airline graph from Figure 16-1 starting from San Francisco. Assuming that the arcs from each city are processed in alphabetical order by destination node, the depth-first strategy will visit the cities in the order shown in the following sample run:



If you think about the depth-first algorithm in relation to other algorithms you've seen, you will realize that its operation is exactly the same as that of the maze-solving algorithm in Chapter 7. In that algorithm, it was also necessary to mark squares along the path to avoid cycling forever around a loop in the maze. That marking process is analogous to the **visited** set in the depth-first search implementation.

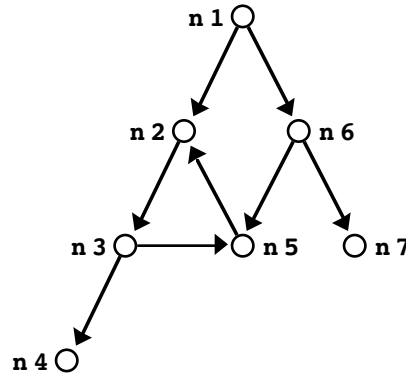
Breadth-first search

Although depth-first search has many important uses, the strategy has drawbacks that make it inappropriate for certain applications. The biggest problem with the depth-first approach is that it explores an entire path beginning at one neighbor before it ever goes back and looks at the other nearby neighbors. If you were trying to discover the shortest path between two nodes in a large graph, using depth-first search would take you all the way to the far reaches of the graph, even if your destination were one step away along a different path.

The breadth-first search algorithm gets around this problem by visiting each node in an order determined by how close it is to the starting node, measured in terms of the number of arcs along the shortest possible path. When you measure distance by counting arcs, each arc constitutes one **hop**. Thus, the essence of breadth-first search is that you visit

the starting node first, then the nodes that are one hop away, followed by the nodes two hops away, and so on.

To get a more concrete sense of this algorithm, suppose that you wanted to apply a breadth-first traversal to the graph used for the depth-first search illustration:



If you start at node **n1**, the breadth-first strategy visits the nodes in the following order:

- The initial node: **n1**
- All nodes one hop away: **n2** and **n6**
- All nodes two hops away: **n3**, **n5**, and **n7**
- All nodes three hops away: **n4**

Note that **n5** is three hops away from **n1** along the path that begins with **n2**, but only two hops away along the path beginning with **n6**. Thus, it is important to make sure that the breadth-first algorithm considers the possibility of multiple paths to the same node and visits them in order of the minimum hop count from the original node.

The easiest way to code the breadth-first algorithm is to use a queue of unprocessed nodes. At each step in the process, you enqueue the neighbors of the current node. Because the queue is processed in order, all nodes that are one hop away from the starting node will appear earlier in the queue than nodes that are two hops away, and so forth.

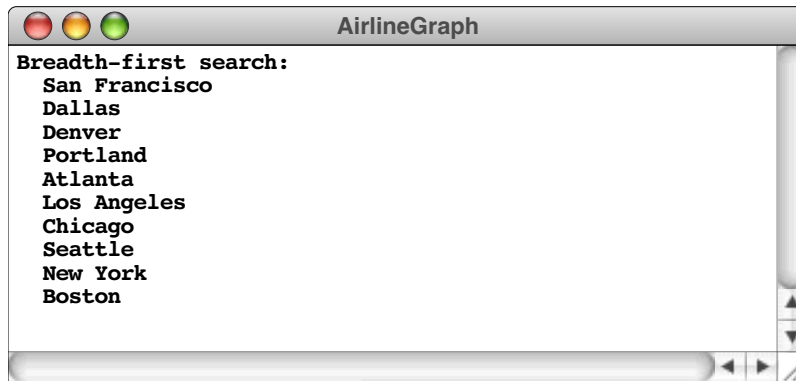
The most straightforward—but not necessarily the most efficient, as you will have a chance to discover in exercise 7—implementation of this strategy looks like this:

```

void BreadthFirstSearch(nodeT *node) {
    Set<nodeT *> visited;
    Queue<nodeT *> queue;
    queue.enqueue(node);
    while (!queue.isEmpty()) {
        node = queue.dequeue();
        if (!visited.contains(node)) {
            Visit(node);
            visited.add(node);
            foreach (arcT *arc in node->arcs) {
                queue.enqueue(arc->finish);
            }
        }
    }
}

```

Once again, it is useful to trace this algorithm on the airline graph. If you start in San Francisco and enqueue nodes in alphabetic order, a breadth-first search looks like this:



16.5 Defining a Graph class

As it appears in Figure 16-3, the `graph.h` interface leaves much to be desired. In particular, the existing version of the interface uses low-level structures to represent a graph and consequently takes no advantage of the object-oriented features of C++. The use of structures, moreover, means that there are no methods associated with graphs, forcing clients to develop their own tools.

The sections that follow outline two possible strategies for replacing the low-level graph package with a more sophisticated object-oriented design. The first makes more extensive use of the object-oriented features of C++, but is somewhat difficult to use. The second outlines an intermediate-level strategy that introduces a **Graph** class, but then allows clients to use either structures or classes for the nodes and arcs. This design offers greater flexibility and, at the same time, simplifies the design of client code.

Using classes for graphs, nodes, and arcs

The most straightforward way to apply object-oriented design to the `graph.h` interface is simply to replace each of the low-level structures with a class. Under this strategy, the interface would export a **Graph** class in the place of the `graphT` structure, a **Node** class in place of `nodeT`, and an **Arc** class in place of `arcT`. The private fields in each of those classes would presumably match those in the corresponding structure. Clients, however, would gain access to that fields through method calls instead of through direct references.

Although this design is workable, it turns out to be cumbersome in practice. To understand why, it is important to note that graphs are somewhat different from the more familiar container classes such as arrays, grids, stacks, queues, and sets are. These more conventional collections contain values of some client-defined type. In this text, for example, you have seen programs that declare variables of type `Stack<double>` and `Set<string>`. The type within the angle brackets specifies the element type and is used as a template parameter for the class definition. The element type, however, plays no role in the implementation of the collection class itself.

For graphs, the situation is different. The elements of a graph are the nodes and arcs. Those structures, however, are an integral part of the graph and contain information required to maintain the overall data structure. Nodes, for example, need to keep track of the set of arcs to other nodes; arcs need to keep track of their endpoints. At the same time, clients may want to associate additional data with each node or arc, depending on the application. Thus, nodes and arcs are hybrid structures that contain data required by the client along with data required by the implementation.

When applications require hybrid structures of this kind, the usual strategy is to use subclassing. Under this model, the graph package itself defines base classes for **Node** and **Arc** that contain the information necessary to represent the graph structure. Clients needing to include additional data would do so by extending these base classes to create new subclasses with additional fields and methods.

Suppose, for example, that you needed to include flight numbers as part of the data represented in the airline graph. A flight number is not relevant to the graph structure itself but instead makes sense only in the context of a particular application. To accommodate this information, you could define an extension to the **Arc** class like this:

```
class Flight : public Arc {
public:
    int getFlightNumber();
    void setFlightNumber(int number);
private:
    int flightNumber;
};
```

This definition defines a new class called **Flight** that inherits all the public methods from the **Arc** class but that also exports a getter and a setter method for an integer field containing the flight number. The implementation of the client application for the airline can then use this new data field without affecting the ability of a **Flight** object to act as an arc in a graph, because that behavior is inherited from the superclass. Similarly, you might need to define a **City** class that extends **Node** by adding some information that makes sense only in the airline context, such as the three-letter code for that city's airport. Your application would then work with pointers to **City** and **Flight** objects, secure in the knowledge that these were also valid pointers to the underlying **Node** and **Arc** objects.

Although this design has some advantages in terms of protecting the data in the graph abstraction from manipulation by the client, it adds complexity to the client code, mostly because the use of subclasses introduces the need for type casting. Even though they are as yet unspecified, you can be sure that the methods exported by the classes in **graph.h** are defined in terms of the base classes **Node** and **Arc**. The **graph.h** interface itself can't know anything about the **City** and **Flight** subclasses, because those are in the domain of a single client. Whenever one of the graph methods returns to a **Node *** or an **Arc ***, the application will need to cast that pointer to a **City *** or **Flight ***, as appropriate. Having to include all these type casts in the program introduces enough complexity to obscure some of the elegance of the underlying algorithms.

Adopting an intermediate strategy

Fortunately, it is possible to adopt a slightly less aggressive strategy that takes advantage of the power of object-oriented design while retaining the simplicity of the low-level, structure-based approach. The conventional collection classes use template parameters to specify their element type. The basic idea behind this design is to export a parameterized **Graph** class that lets the client choose structure types for the nodes and arcs. Those structure types, however, cannot be chosen arbitrarily but instead must adhere to a few basic rules that allow them to function correctly in the context of a graph. These required fields are essentially the ones that appear in the low-level structures. Thus, the type the client chooses to represent a node must contain

- A public string field called **name** that specifies the unique name of the node
- A public field called **arcs** that specifies the set of arcs that begin at this node

Similarly, the type chosen to represent an arc must contain

- Public field called **start** and **finish** that indicate the endpoints of the arc
- A public field called **cost** that indicates the cost of traversing this arc (this field is not actually required by the interface itself, but is needed for several of the extensions described in the exercises)

Each of these structures types, moreover, must define its internal fields in terms of the client data structures. Thus, the element type for set of arcs contained in a node must be pointers to the client's arc type. In precisely the same way, the two node pointers in the arc structure must be declared as pointers to the client's node type.

A simple example will prove helpful in clarifying these rules. Drawing on the discussion from the previous section, suppose that your goal is to define an airline graph in which the nodes are cities containing an airport code and the arcs are flights containing a flight number. Using this design, you could define two new structure types as follows:

```
struct cityT {
    string name;
    Set<flightT *> arcs;
    string airportCode;
};

struct flightT {
    cityT *start;
    cityT *finish;
    double cost;
    int flightNumber;
};
```

The airline graph itself would then be declared using a template class like this:

```
Graph<cityT,flightT> airline;
```

Given that the template version of the **Graph** class now has access to the node and arc types used by the clients, it can itself ensure that the types are correctly represented throughout the structure, eliminating the need for type casts.

Figure 16-5 shows the interface for a graph package that uses this intermediate strategy of defining **Graph** as a class with two template parameters, one that specifies the type used for nodes and one that specifies the type used for arcs. The interface is followed immediately by Figures 16-6 and 16-7, which supply the private section of the class and the template class implementation, respectively.

For the most part, the code in Figures 16-5 through 16-7 is straightforward, especially if you understand the low-level version of the interface. A few points, however, are worth special mention:

- The new **graph.h** interface includes all the definitions from the preliminary version. This strategy increases stability because clients of the original version will not need to make any changes to their code. In addition, the inclusion of definitions for **nodeT** and **arcT** allow clients to use these types if they have no need to define their own.
- The implementation includes two methods—**NodeCompare** and **ArcCompare**—that compare nodes and arcs alphabetically by node name. These methods, which use the **template** keyword to ensure that they operate on any client-supplied types, are then used as the comparison function for the sets in the graph and in the individual nodes.

Figure 16-5 The extended version of the graph.h interface

```

/*
 * File: graph.h
 * -----
 * Extended interface for the graph package. This interface exports
 * both a low-level graphT type and a parameterized Graph class.
 */

#ifndef _graph_h
#define _graph_h

#include "set.h"
#include "map.h"

struct nodeT;      /* Forward references to these two types so */
struct arcT;       /* that the C++ compiler can recognize them. */

/*
 * Type: graphT
 * -----
 * This type represents a graph and consists of two sets -- a set
 * of nodes and a set of arcs -- along with a map that creates an
 * association between names and nodes.
 */

struct graphT {
    Set<nodeT *> nodes;
    Set<arcT *> arcs;
    Map<nodeT *> nodeMap;
};

/*
 * Type: nodeT
 * -----
 * This type is the supplied type for a node in a graph. Clients
 * may substitute their own type, as described in the notes for
 * the Graph class.
 */

struct nodeT {
    string name;
    Set<arcT *> arcs;
};

/*
 * Type: arcT
 * -----
 * This type is the supplied type for an arc in a graph. Clients
 * may substitute their own type, as described in the notes for
 * the Graph class.
 */

struct arcT {
    nodeT *start;
    nodeT *finish;
    double cost;
};

```

```

/*
 * Class: Graph<NodeType,ArcType>
 * -----
 * This class represents a graph with the specified node and arc
 * types. The NodeType and ArcType parameters indicate the record
 * or object types used for nodes and arcs, respectively. These
 * types can contain any fields or methods required by the client,
 * but must also contain the following public fields required by
 * the Graph package itself:
 *
 * The NodeType definition must include:
 *   - A string field called name
 *   - A Set<ArcType *> field called arcs
 *
 * The ArcType definition must include:
 *   - A NodeType * field called start
 *   - A NodeType * field called finish
 *   - A double field called cost
 */

template <typename NodeType,typename ArcType>
class Graph {

public:

/*
 * Constructor: Graph
 * Usage: Graph<NodeType,ArcType> g;
 * -----
 * Declares a new Graph object named g.
 */
    Graph();

/*
 * Destructor: ~Graph
 * Usage: (usually implicit)
 * -----
 * Frees the internal storage allocated for the nodes and
 * arcs in the graph.
 */
    ~Graph();

/*
 * Method: clear
 * Usage: g.clear();
 * -----
 * Frees the storage for all nodes and arcs in the graph and
 * reinitializes the graph to be empty.
 */
    void clear();

```



```

/*
 * Method: addNode
 * Usage: g.addNode(name);
 *         g.addNode(node);
 * -----
 * Adds a node to the graph. The first version of this method
 * creates a new node of the appropriate type and initializes its
 * fields; the second assumes that the client has already created
 * the node and simply adds it to the graph. Both versions of this
 * method return a pointer to the node in case the client needs to
 * capture this value.
 */
NodeType *addNode(string name);
NodeType *addNode(NodeType *node);

/*
 * Method: addArc
 * Usage: g.addArc(s1, s2);
 *         g.addArc(n1, n2);
 *         g.addArc(arc);
 * -----
 * Adds an arc to the graph. The endpoints of the arc can be
 * specified either as strings indicating the names of the nodes
 * or as pointers to the node structures. Alternatively, the
 * client can create the arc structure explicitly and pass that
 * pointer to the addArc method. All three of these versions
 * return a pointer to the arc in case the client needs to
 * capture this value.
 */
ArcType *addArc(string s1, string s2);
ArcType *addArc(NodeType *n1, NodeType *n2);
ArcType *addArc(ArcType *arc);

/*
 * Method: isConnected
 * Usage: if (g.isConnected(n1, n2)) . . .
 *         if (g.isConnected(s1, s2)) . . .
 * -----
 * Returns true if the graph contains an arc from n1 to n2. As
 * in the addArc method, nodes can be specified either as node
 * pointers or by name.
 */
bool isConnected(NodeType *n1, NodeType *n2);
bool isConnected(string s1, string s2);

/*
 * Method: getNode
 * Usage: NodeType *node = g.getNode(name);
 * -----
 * Looks up a node in the name table attached to the graph and
 * returns a pointer to that node. If no node with the specified
 * name exists, getNode returns NULL.
 */
NodeType *getNode(string name);

```

```

/*
 * Method: getNodeSet
 * Usage: foreach (NodeType *node in g.getNodeSet()) . . .
 * -----
 * Returns the set of all nodes in the graph. This method returns
 * a reference to the set rather than a copy of the set, which makes
 * it possible to iterate over its elements.
 */
Set<NodeType *> & getNodeSet();

/*
 * Method: getArcSet
 * Usage: foreach (ArcType *arc in g.getArcSet()) . . .
 *        foreach (ArcType *arc in g.getArcSet(node)) . . .
 * -----
 * Returns the set of all arcs in the graph or, in the second form,
 * the arcs that start at the specified node. This method returns
 * a reference to the set rather than a copy of the set, which makes
 * it possible to iterate over its elements.
 */
Set<ArcType *> & getArcSet();
Set<ArcType *> & getArcSet(NodeType *node);

private:

#include "graphpriv.h"

};

#include "graphimpl.cpp"

#endif

```

Figure 16-6 The contents of the private section of the Graph class

```

/*
 * File: graphpriv.h
 * -----
 * This file defines the private data for the Graph class.
 */

/*
 * Implementation notes: Data structure
 * -----
 * The Graph class is defined -- as it traditionally is in
 * mathematics -- as a set of nodes and a set of arcs. These
 * structures, in turn, are implemented using the Set class.
 * The element type for each set is a pointer to a structure
 * chosen by the client, which is specified as one of the
 * parameters to the class template.
 */

/* Instance variables */
Set<NodeType *> nodes;
Set<ArcType *> arcs;
Map<NodeType *> nodeMap;

```

Figure 16-7 The private implementation of the Graph class

```

/*
 * File: graphimpl.cpp
 * -----
 * This file provides the private implementation for the Graph
 * class. As is typical for layered abstractions built on top
 * of other classes, the implementations of these methods tend
 * to be one or two lines long and require no detailed commentary.
 */

#ifdef _graph_h

#include "genlib.h"

/*
 * Implementation notes: NodeCompare, ArcCompare
 * -----
 * These generic functions compare nodes and arcs by comparing
 * the node names alphabetically.
 */

template <typename NodeType>
int NodeCompare(NodeType *n1, NodeType *n2) {
    if (n1->name == n2->name) return 0;
    return (n1->name < n2->name) ? -1 : +1;
}

template <typename NodeType, typename ArcType>
int ArcCompare(ArcType *a1, ArcType *a2) {
    NodeType *n1 = a1->start;
    NodeType *n2 = a2->start;
    int cmp = NodeCompare(n1, n2);
    if (cmp != 0) return cmp;
    n1 = a1->finish;
    n2 = a2->finish;
    return NodeCompare(n1, n2);
}

/*
 * Implementation notes: Constructor
 * -----
 * The constructor requires no code in its function body, but
 * does use the initialization list to set up the nodes and
 * arc sets with the appropriate comparison functions.
 */

template <typename NodeType, typename ArcType>
Graph<NodeType, ArcType>::Graph() : nodes(NodeCompare<NodeType>),
                                   arcs(ArcCompare<NodeType, ArcType>) {
    /* Empty */
}

template <typename NodeType, typename ArcType>
Graph<NodeType, ArcType>::~~Graph() {
    clear();
}

```

```

template <typename NodeType,typename ArcType>
void Graph<NodeType,ArcType>::clear() {
    foreach (NodeType *node in nodes) {
        delete node;
    }
    foreach (ArcType *arc in arcs) {
        delete arc;
    }
    arcs.clear();
    nodes.clear();
    nodeMap.clear();
}

template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::addNode(NodeType *node) {
    nodes.add(node);
    nodeMap[node->name] = node;
    return node;
}

template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::addNode(string name) {
    NodeType *node = new NodeType();
    node->arcs = Set<ArcType *>(ArcCompare<NodeType,ArcType>);
    node->name = name;
    return addNode(node);
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(ArcType *arc) {
    arc->start->arcs.add(arc);
    arcs.add(arc);
    return arc;
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(NodeType *n1, NodeType *n2) {
    ArcType *arc = new ArcType();
    arc->start = n1;
    arc->finish = n2;
    arc->cost = 1;
    return addArc(arc);
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(string s1, string s2) {
    return addArc(getNode(s1), getNode(s2));
}

template <typename NodeType,typename ArcType>
bool Graph<NodeType,ArcType>::isConnected(NodeType *n1, NodeType *n2) {
    foreach (ArcType *arc in n1->arcs) {
        if (arc->finish == n2) return true;
    }
    return false;
}

```

```

template <typename NodeType,typename ArcType>
bool Graph<NodeType,ArcType>::isConnected(string s1, string s2) {
    return isConnected(getNode(s1), getNode(s2));
}

template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::getNode(string name) {
    if (nodeMap.containsKey(name)) return nodeMap.get(name);
    return NULL;
}

template <typename NodeType,typename ArcType>
Set<NodeType *> & Graph<NodeType,ArcType>::getNodeSet() {
    return nodes;
}

template <typename NodeType,typename ArcType>
Set<ArcType *> & Graph<NodeType,ArcType>::getArcSet() {
    return arcs;
}

template <typename NodeType,typename ArcType>
Set<ArcType *> & Graph<NodeType,ArcType>::getArcSet(NodeType *node) {
    return node->arcs;
}

#endif

```

16.6 Finding minimum paths

Because graphs arise in many applications areas that have commercial importance, a considerable amount of research has been invested in developing effective algorithms for solving graph-related problems. Of these problems, one of the most interesting is that of finding a path in a graph from one node to another that has the smallest possible cost when evaluated according to some metric. This metric need not be economic. Although you might be interested in finding the cheapest path between two nodes for certain applications, you can use the same algorithm to find a path with the shortest overall distance, the smallest number of hops, or the least travel time.

As a concrete example, suppose that you want to find the path from San Francisco to Boston that has the shortest total distance, as computed by the mileage values shown on the arcs in Figure 16-2. Is it better to go through Portland and Seattle, or should you instead go through Dallas, Atlanta, and New York? Or is there perhaps some less obvious route that is shorter still?

With graphs as simple as the route map of this tiny airline, it is easy to compute the answer just by adding up the length of the arcs along all possible paths. As the graph grows larger, however, this approach can become unworkable. In general, the number of paths between two nodes in a graph grows in an exponential fashion, which means that the running time of the explore-all-paths approach is $O(2^N)$. As you know from the discussion of computational complexity in Chapter 8, problems whose solutions require exponential running time are considered to be intractable. If you want to find the shortest path through a graph in a reasonable amount of time, it is essential to use a more efficient algorithm.

The most commonly used algorithm for finding shortest paths was discovered by Edsger Dijkstra in 1959. Dijkstra’s algorithm for finding shortest paths is a particular example of a class of algorithms called **greedy algorithms**, in which you find the overall answer by making a series of locally optimal decisions. Greedy algorithms do not work for every problem, but are quite useful in solving the problem of finding the shortest path.

At its essence, the core of Dijkstra’s algorithm for finding the shortest path—or, more generally, the path whose arcs have the minimum total cost—can be expressed as follows: explore all paths from the starting node in order of increasing total path cost until you encounter a path that takes you to your destination. This path must be the best one, because you have already explored all paths beginning at the starting node that have a lower cost. In the context of the specific problem of finding the shortest path, Dijkstra’s algorithm can be implemented as shown in Figure 16-8.

The code for **FindShortestPath** makes more sense if you think carefully about the data structures it uses. The implementation declares three local variables, as follows:

- The variable **path** keeps track of the minimum path and consists of a vector of arcs. The first arc in the vector will start at the origin and proceed to the first intermediate stop. Each subsequent path begins where the preceding one left off, continuing on in this way until the final arc ends at the destination. If there is no path between the requested nodes, **FindShortestPath** indicates that fact by returning an empty vector.
- The variable **queue** is a queue of paths, ordered so that paths in the queue are sorted in order of increasing cost. This queue therefore differs from the first-in/first-out discipline of traditional queues and is instead a **priority queue**, in which the client can specify a priority value for each element in the queue. As in conventional English usage, smaller priority numbers come first in the queue, so that elements with priority 1 are entered into the queue ahead of elements with priority 2. This code for **FindShortestPath** assumes that this feature has been added to the **queue.h** interface as suggested in Chapter 10, exercise 4. The new version of the **Queue** class exported by this interface exports an overloaded version of **enqueue** with the prototype

```
void enqueue(ElemType element, double priority);
```

Calling this version of **enqueue** does not simply insert the new element at the end of the queue, but instead inserts it into the list of existing elements according to the priority. By entering each path into the queue using its total distance as a priority value, each call to **dequeue** returns the shortest path remaining in the queue.

- The variable **fixed** is a map that associates each city name with the minimum distance to that city, as soon as that distance becomes known. Whenever a path is dequeued from the priority queue, you know the path must indicate the shortest route to the node at the end of that path, unless you have already found a shorter path ending at that node. Thus, whenever you dequeue a path from the priority queue, you can note that its distance is now known by storing the minimum distance in the map **fixed**.

The operation of **FindShortestPath** is illustrated in Figure 16-9, which shows the steps involved in computing the shortest path from San Francisco to Boston in the airline graph from Figure 16-2. The complete trace of the process illustrates the following properties of the algorithm, which are important to keep in mind:

- *Paths are explored in order of the total distance rather than the number of hops.* Thus, the connections beginning with **San Francisco**→**Portland**→**Seattle** are explored before those of either **San Francisco**→**Denver** or **San Francisco**→**Dallas**, because the total distance is shorter.

Figure 16-8 Dijkstra's algorithm for finding the shortest path

```

/*
 * Function: FindShortestPath
 * Usage: Vector<arcT *> path = FindShortestPath(start, finish);
 * -----
 * Uses Dijkstra's algorithm to find the shortest path from the
 * start to the finish node. The path is returned as a vector of
 * arc pointers. If no path exists, the result is an empty vector.
 */

Vector<arcT *> FindShortestPath(nodeT *start, nodeT *finish) {
    Vector<arcT *> path;
    Queue< Vector<arcT *> > queue;
    Map<double> fixed;
    while (start != finish) {
        if (!fixed.containsKey(start->name)) {
            fixed.put(start->name, TotalPathDistance(path));
            foreach (arcT *arc in start->arcs) {
                if (!fixed.containsKey(arc->finish->name)) {
                    Vector<arcT *> newPath = path;
                    newPath.add(arc);
                    queue.enqueue(newPath, TotalPathDistance(newPath));
                }
            }
        }
        if (queue.isEmpty()) return Vector<arcT *>();
        path = queue.dequeue();
        start = path[path.size() - 1]->finish;
    }
    return path;
}

/*
 * Function: TotalPathDistance
 * Usage: double distance = TotalPathDistance(path);
 * -----
 * Returns the total distance along the path.
 */

double TotalPathDistance(Vector<arcT *> path) {
    double distance = 0;
    foreach (arcT *arc in path) {
        distance += arc->cost;
    }
    return distance;
}

```

Figure 16-9 Steps involved in finding the shortest path from San Francisco to Boston

```

Fix the distance to San Francisco at 0
Process the arcs out of San Francisco (Dallas, Denver, Portland)
    Enqueue the path: San Francisco → Dallas (1468)
    Enqueue the path: San Francisco → Denver (954)
    Enqueue the path: San Francisco → Portland (550)
Dequeue the shortest path: San Francisco → Portland (550)
Fix the distance to Portland at 550
Process the arcs out of Portland (San Francisco, Seattle)
    Ignore San Francisco because its distance is known
    Enqueue the path: San Francisco → Portland → Seattle (680)
Dequeue the shortest path: San Francisco → Portland → Seattle (680)
Fix the distance to Seattle at 680
Process the arcs out of Seattle (Boston, Portland)
    Enqueue the path: San Francisco → Portland → Seattle → Boston (3169)
    Ignore Portland because its distance is known
Dequeue the shortest path: San Francisco → Denver (954)
Fix the distance to Denver at 954
Process the arcs out of Denver (Chicago, Dallas, San Francisco)
    Ignore San Francisco because its distance is known
    Enqueue the path: San Francisco → Denver → Chicago (1861)
    Enqueue the path: San Francisco → Denver → Dallas (1604)
Dequeue the shortest path: San Francisco → Dallas (1468)
Fix the distance to Dallas at 1468
Process the arcs out of Dallas (Atlanta, Denver, Los Angeles, San Francisco)
    Ignore Denver and San Francisco because their distances are known
    Enqueue the path: San Francisco → Dallas → Atlanta (2193)
    Enqueue the path: San Francisco → Dallas → Los Angeles (2708)
Dequeue the shortest path: San Francisco → Denver → Dallas (1604)
Ignore Dallas because its distance is known
Dequeue the shortest path: San Francisco → Denver → Chicago (1861)
Fix the distance to Chicago at 1861
Process the arcs out of Chicago (Atlanta, Denver)
    Ignore Denver because its distance is known
    Enqueue the path: San Francisco → Denver → Chicago → Atlanta (2460)
Dequeue the shortest path: San Francisco → Dallas → Atlanta (2193)
Fix the distance to Atlanta at 2193
Process the arcs out of Atlanta (Chicago, Dallas, New York)
    Ignore Chicago and Dallas because their distances are known
    Enqueue the path: San Francisco → Dallas → Atlanta → New York (2949)
Dequeue the shortest path: San Francisco → Denver → Chicago → Atlanta (2460)
Ignore Atlanta because its distance is known
Dequeue the shortest path: San Francisco → Dallas → Los Angeles (2708)
Fix the distance to Los Angeles at 2708
Process the arcs out of Los Angeles (Dallas)
    Ignore Dallas because its distance is known
Dequeue the shortest path: San Francisco → Dallas → Atlanta → New York (2949)
Fix the distance to New York at 2949
Process the arcs out of New York (Atlanta, Boston)
    Ignore Atlanta because its distance is known
    Enqueue the path: San Francisco → Dallas → Atlanta → New York → Boston (3140)
Dequeue the shortest path: San Francisco → Dallas → Atlanta → New York → Boston (3140)
    
```

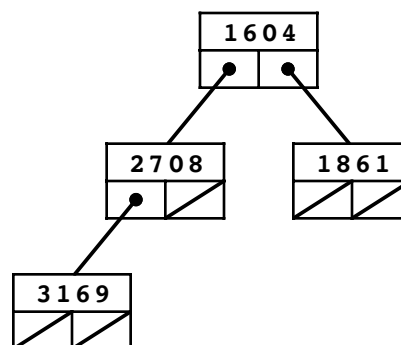

- *The distance to a node is fixed when a path is dequeued, not when it is enqueued.* The first path to Boston stored in the priority queue is the one that goes through Portland and Seattle, which is not the shortest available path. The total distance along the path **San Francisco**→**Portland**→**Seattle**→**Boston** is 3169. Because the minimum distance is actually only 3140, the **San Francisco**→**Portland**→**Seattle**→**Boston** path is still in the priority queue when the algorithm finishes its operation.
- *The arcs from each node are scanned at most once.* The inner loop of the algorithm is executed only when the distance to that node is fixed, which happens only once for each node. As a result, the total number of cycles executed within the inner loop is the product of the number of nodes and the maximum number of arcs leading from a node. A complete analysis of Dijkstra's algorithm is beyond the scope of this text, but the running time—after making an important enhancement to the efficiency of the implementation described in the following section—is $O(M \log N)$, where N is the number of nodes and M is either N or the number of arcs, whichever is larger.

16.7 An efficient implementation of priority queues

As is often the case, the performance of Dijkstra's algorithm depends to a large extent on how well its underlying operations are implemented. For example, if you can improve the efficiency of the priority queue implementation, Dijkstra's algorithm will run more quickly, because it depends on the priority queue package. If you implement priority queues using the strategy described in exercise 4 from Chapter 10, the **PriorityEnqueue** function requires $O(N)$ time. You can improve the performance of the priority queue package to $O(\log N)$ by using a data structure called a **partially ordered tree**, which is a binary tree in which the following two properties hold:

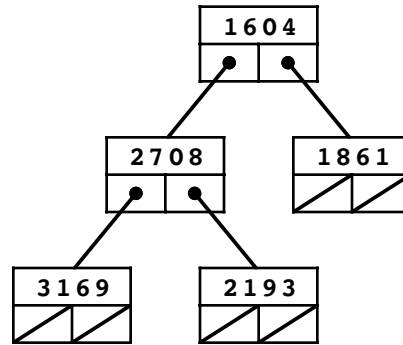
1. The nodes of the tree are arranged in a pattern as close to that of a completely symmetrical tree as possible. Thus, the number of nodes along any path in the tree can never differ by more than one. Moreover, the bottom level must be filled in a strictly left-to-right order.
2. Each node contains a key that is always less than or equal to the key in its children. Thus, the smallest key in the tree is always at the root.

As an example, the following diagram shows a partially ordered tree with four nodes, each of which contains a numeric key:

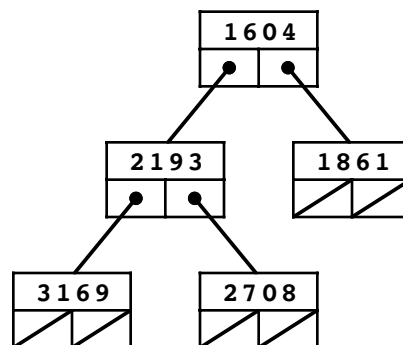


The second level of the tree is completely filled, and the third level is being filled from left to right, as required by the first property of partially ordered trees. The second property holds because the key in each node is always less than the keys in its children.

Suppose that you want to add a node with the key 2193. It is clear where the new node goes. The requirement that the lowest level of the tree be filled from left to right dictates that the new node be added at the following position:

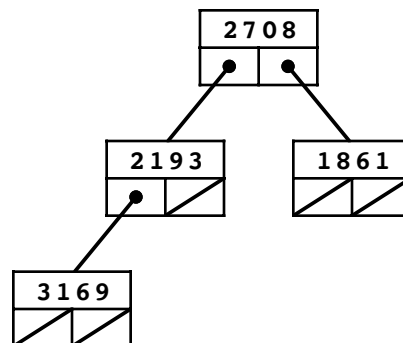


This diagram, however, violates the second property of partially ordered trees, because the key 2193 is smaller than the 2708 in its parent. To fix the problem, you begin by exchanging the keys in those nodes, like this:

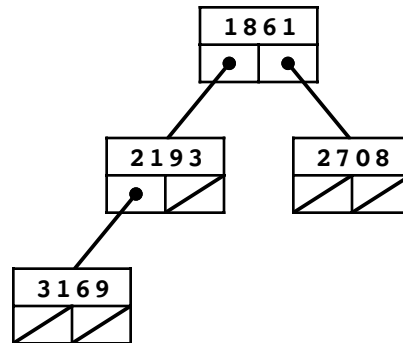


In general, it is possible that the newly inserted key would have to be exchanged with its parent in a cascading sequence of changes that proceed up through the levels of the tree. In this specific case, the process of exchanging keys stops here because 2193 is greater than 1604. In any event, the structure of the tree guarantees that the total number of such exchanges will never require more than $O(\log N)$ time.

The structure of the partially ordered tree means that the smallest value in the tree is always at the root. Removing the root node, however, takes a little more work because you have to arrange for the node that actually disappears to be the rightmost node in the bottom level. The standard approach is to replace the key in the root with the key in the node to be deleted and then swap keys down the tree until the ordering property is restored. If you wanted, for example, to delete the root node from the preceding tree diagram, the first step would be to replace the key in the root node with the 2708 in the rightmost node from the lowest level, as follows:



Then, because the nodes of the tree no longer have correctly ordered keys, you need to exchange the key 2708 with the smaller of the two keys in its children, like this:

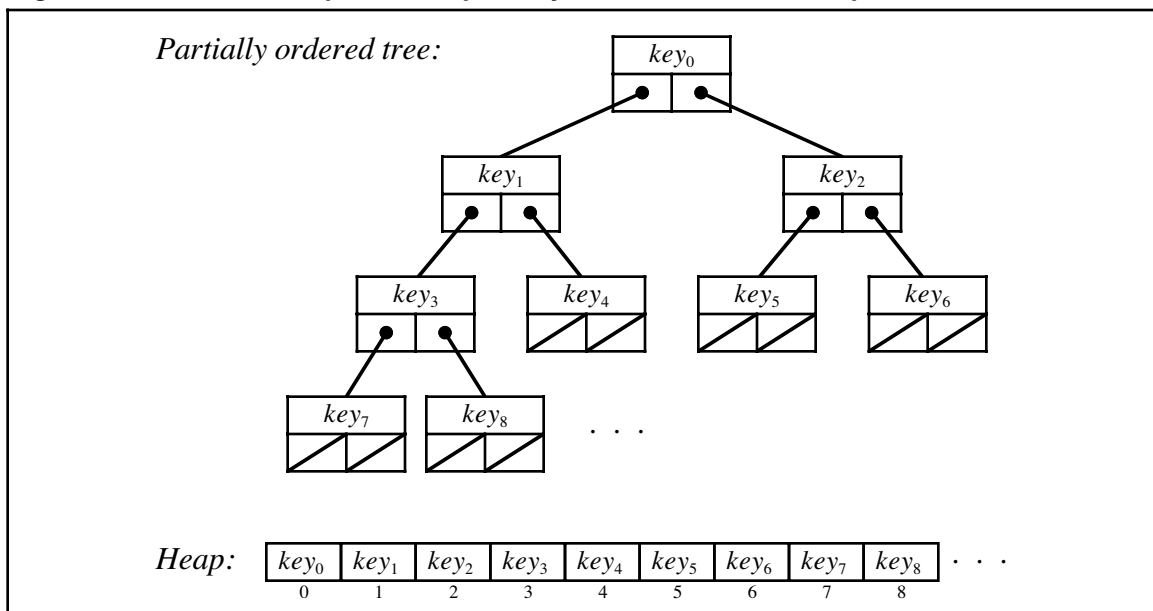


Although a single interchange is enough to restore the ordering property of the tree in this example, the general process of finding the correct position for the key that was moved into the root position may require you to swap that element through each of the levels in the tree. Like insertion, deleting the smallest key requires $O(\log N)$ time.

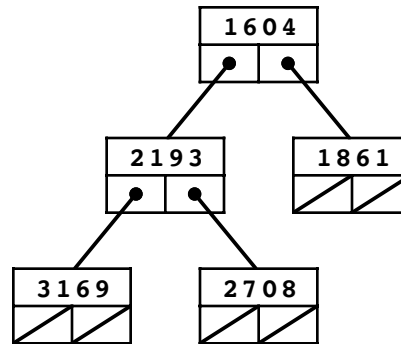
The operations that define the partially ordered tree are precisely the ones you need to implement priority queues. The **enqueue** operation consists of inserting a new node into the partially ordered tree. The **dequeue** operation consists of removing the lowest value. Thus, if you use partially ordered trees as the underlying representation, you can implement the priority queue package so that it runs in $O(\log N)$ time.

Although you can implement partially ordered trees using a pointer-based structure, priority queues are usually implemented using an array-based structure called a **heap**, which simulates the operation of a partially ordered tree. (The terminology is confusing at first, because the heap data structure bears no relationship to the pool of unused memory available for dynamic allocation, which is also referred to by the word *heap*.) The implementation strategy used in a heap depends on the property that the nodes in a partially ordered tree of size N can be stored in the first N elements of an array simply by counting off the nodes, level by level, left to right, as illustrated in Figure 16-10.

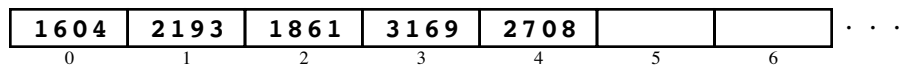
Figure 16-10 Relationship between partially ordered trees and heaps



As an example, the partially ordered tree



can be represented as the following heap:



The heap organization makes it simple to implement tree operations, because parent and child nodes always appear at an easily computed position. For example, given a node at index position n , you can find the indices of its parent and children using the following expressions:

ParentIndex (n)	<i>is always given by</i>	$(n - 1) / 2$
LeftChildIndex (n)	<i>is always given by</i>	$2 * n + 1$
RightChildIndex (n)	<i>is always given by</i>	$2 * n + 2$

The division operator in the calculation of **ParentIndex** is the standard integer division operator from C++. Thus, the parent of the node at index position 8 in the array appears at position 3 in the array, because the result of evaluating the expression $(8 - 1) / 2$ is 3.

Implementing the heap-based priority queue is an excellent exercise that will sharpen your programming skills and give you more experience working with many of the data structures you have seen in this text. You will have the opportunity to do so in exercise 12 at the end of this chapter.

Summary

This chapter has introduced you to the idea of a graph, which is defined as a set of nodes linked together by a set of arcs that connect individual pairs of nodes. Like sets, graphs are not only important as a theoretical abstraction, but also as a tool for solving practical problems that arise in many application domains. For example, graph algorithms are useful in studying the properties of connected structures ranging from the Internet to large-scale transportation systems.

Important points in this chapter include:

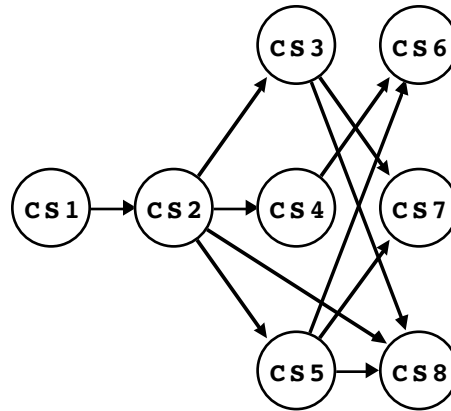
- Graphs may be either directed or undirected. The arcs in a directed graph run in one direction only, so the existence of an arc $n_1 \rightarrow n_2$ does not imply the existence of an arc $n_2 \rightarrow n_1$. You can represent undirected graphs by using directed graphs in which each connected pair of nodes is linked with two arcs, one in each direction.
- You can adopt any of several strategies to represent the connections in a graph. One common approach is to construct an adjacency list, in which the data structure for each

node contains a list of the connected nodes. You can also use an adjacency matrix, which stores the connections in a two-dimensional array of Boolean values. The rows and columns of the matrix are indexed by the nodes in the graph; if two nodes are connected in the graph, the corresponding entry in the matrix contains the value **true**.

- The **graph.h** interface can be implemented easily by layering it on top of the set package. Although it is possible to define such an interface using either a low-level, structure-based approach or a high-level, entirely object-oriented style, it is better to adopt an intermediate approach that defines a **Graph** class but leaves the client free to define the structures used for nodes and arcs.
- The two most important traversal orders for a graph are depth-first search and breadth-first search. The depth-first algorithm chooses one arc from the starting node and then recursively explores all paths beginning with that arc until no additional nodes remain. Only at that point does the algorithm return to explore other arcs from the original node. The breadth-first algorithm explores nodes in order of their distance from the original node, measured in terms of the number of arcs along the shortest path. After processing the initial node, breadth-first search processes all the neighbors of that node before moving on to nodes that are two hops away.
- You can find the minimum-cost path between two nodes in a graph by using Dijkstra's algorithm, which is vastly more efficient than the exponential strategy of comparing the cost of all possible paths. Dijkstra's algorithm is an example of a larger class of algorithms called *greedy algorithms*, which select the locally best option at any decision point.
- Priority queues—which are an essential component of Dijkstra's algorithm—can be implemented efficiently using a data structure called a *heap*, which is based on a special class of binary tree called a *partially ordered tree*. If you use this representation, both the **enqueue** and **dequeue** operations run in $O(\log N)$ time.

Review questions

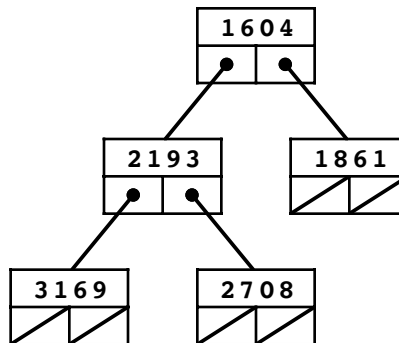
1. What is a graph?
2. True or false: Trees are a subset of graphs, which form a more general class.
3. What is the difference between a directed and an undirected graph?
4. If you are using a graph package that supports only directed graphs, how can you represent an undirected graph?
5. Define the following terms as they apply to graphs: *path*, *cycle*, *simple path*, *simple cycle*.
6. What is relationship between the terms *neighbor* and *degree*?
7. What is the difference between a strongly connected and a weakly connected graph?
8. True or false: The term *weakly connected* has no practical relevance to undirected graphs because all such graphs are automatically strongly connected if they are connected at all.
9. What terms do mathematicians typically use in place of the words *node* and *arc*?
10. Suppose that the computer science offerings at some university consisted of eight courses with the following prerequisite structure:



Using the mathematical formulation for graphs described in this chapter, define this graph as a pair of sets.

11. Draw a diagram showing the adjacency list representation of the graph in the preceding question.
12. Given the prerequisite graph shown in question 10, what are the contents of corresponding adjacency matrix?
13. What is the difference between a sparse and a dense graph?
14. If you were asked to choose the underlying representation of a graph for a particular application, what factors would you consider in deciding whether to use adjacency lists or adjacency matrices in the implementation?
15. Why is it unnecessary to implement a separate iterator facility for the graph package?
16. Why do the sets used in either version of the **graph.h** interface use pointers to arcs and nodes as their element types?
17. What are the two fundamental traversal strategies for graphs?
18. Write down both the depth-first and the breadth-first traversal of the airline graph in Figure 16-1, starting from Atlanta. Assume that iteration over nodes and arcs always occurs in alphabetical order.
19. What problem does this chapter cite as the most significant problem with including class definitions for **Node** and **Arc** in the **graph.h** interface?
20. What rules does the **graph.h** interface impose on the client-defined types used to represent nodes and arcs?
21. What is a greedy algorithm?
22. Explain the operation of Dijkstra's algorithm for finding minimum-cost paths.
23. Show the contents of the priority queue at each step of the trace of Dijkstra's algorithm shown in Figure 16-9.
24. Using Figure 16-9 as a model, trace the execution of Dijkstra's algorithm to find the shortest path from Portland to Atlanta.

25. Suppose that you are working with a partially ordered tree that contains the following data:



Show the state of the partially ordered tree after inserting a node with the key 1521.

26. What is the relationship between heaps and partially ordered trees?

Programming exercises

1. Using the low-level, structure-based version of the **graph.h** interface, design and implement a function

```
graphT *ReadGraph(ifstream & infile);
```

that reads a text description of a graph from **infile** and returns a pointer to a newly allocated **graphT** that contains the corresponding graph structure. The file, which must already be open for input, consists of lines that can be in any of these three forms:

x	Defines a node with name x
$x \rightarrow y$	Defines the directional arc $x \rightarrow y$
$x \leftrightarrow y$	Defines the arcs $x \rightarrow y$ and $y \rightarrow x$

The names x and y are arbitrary strings that do not contain either of the character sequences $\rightarrow$ or $\leftrightarrow$. The single arrow ($\rightarrow$) defines an arc in one direction only; the double arrow ($\leftrightarrow$) defines arcs in both directions and is therefore useful for describing an undirected graph.

Either of the two connection formats may be followed optionally by a number in parentheses representing the cost of that arc. If no parenthesized value appears, the cost of the arc should be initialized to 1. The definition of the graph ends with a blank line or the end of the file.

New nodes are defined whenever a new name appears in the data file. Thus, if every node is connected to some other node, it is sufficient to include only the arcs in the data file because defining an arc automatically defines the nodes at its endpoints. If you need to represent a graph containing isolated nodes, you must specify the names of those nodes on separate lines somewhere in the data file.

When reading in an arc description, your implementation should discard leading and trailing spaces from the node names, but retain any internal spaces. The line

```
San Francisco <-> Denver (954)
```

should therefore define nodes with the names "**San Francisco**" and "**Denver**", and then create connections between the two nodes in each direction, initializing both arcs to have a cost of 954.

As an example, calling **ReadGraph** on the following data file would produce the airline graph that appears in the chapter as Figure 16-2:

```
Atlanta <--> Chicago (599)
Atlanta <--> Dallas (725)
Atlanta <--> New York (756)
Boston <--> New York (191)
Boston <--> Seattle (2489)
Chicago <--> Denver (907)
Dallas <--> Denver (650)
Dallas <--> Los Angeles (1240)
Dallas <--> San Francisco (1468)
Denver <--> San Francisco (954)
Portland <--> Seattle (130)
Portland <--> San Francisco (550)
```

2. Write the counterpart function

```
void WriteGraph(graphT *g, ofstream & outfile);
```

that writes a text description of a graph to the specified output file. You may assume that the data field in each node of the graph contains its name, just as if **ReadGraph** had created the graph. The output of the **WriteGraph** function must be readable using **ReadGraph**.

3. Extend the **graph.h** interface from Figure 16-5 so that the **Graph** class exports the methods

```
void read(ifstream & infile);
void write(ofstream & outfile);
```

These methods should read and write graph data for the template version of the **Graph** class. The format of the data files is described in the two preceding exercises.

4. Eliminate the recursion from the implementation of **DepthFirstSearch** by using a stack to store the unexplored nodes. At the beginning of the algorithm, you simply push the starting node on the stack. Then, until the stack is empty, you repeat the following operations:
 1. Pop the topmost node from the stack.
 2. Visit that node.
 3. Push its neighbors on the stack
5. Take your solution to the preceding exercise and replace the stack with a queue. Describe the traversal order implemented by the resulting code.
6. The **DepthFirstSearch** and **BreadthFirstSearch** traversal functions given in the chapter are written to emphasize the structure of the underlying algorithms. If you wanted to include these traversal strategies as part of the graph package, you would need to reimplement the functions so that they no longer depended on a client-supplied **visit** function. One approach is to implement these two algorithms by adding the following methods to **graph.h**:

```
void mapDFS(void (*fn)(NodeType *), nodeADT start);
void mapBFS(void (*fn)(NodeType *), nodeADT start);
```

In each case, the functions should call **fn(node)** for every node reachable from **start** in the specified traversal order.

7. The implementation of breadth-first search given in the chapter generates the correct traversal but ends up adding a large number of unnecessary paths to the queue. The problem is that the code adds new paths to the queue even when the final node in the chain has already been visited, which means that it will simply be ignored whenever that path is removed from the queue. You can fix this problem simply by adding an additional check, as follows:

```
void BreadthFirstSearch(nodeT *node) {
    Set<nodeT *> visited;
    Queue<nodeT *> queue;
    queue.enqueue(node);
    while (!queue.isEmpty()) {
        node = queue.dequeue();
        if (!visited.contains(node)) {
            Visit(node);
            visited.add(node);
            foreach (arcT *arc in node->arcs) {
                if (!visited(arc->finish)) {
                    queue.enqueue(arc->finish);
                }
            }
        }
    }
}
```

Write a program to test assess the relative efficiency of the implementations with and without this test. Your program should read in several large graphs that vary in their average degree and then run each of these algorithms starting at random nodes in each graph. Your program should keep track of both the average queue length during the execution of the algorithm and the total running time necessary to visit each of the nodes.

7. Write a function

```
bool PathExists(nodeT *n1, nodeT *n2);
```

that returns **true** if there is a path in the graph between the nodes **n1** and **n2**. Implement this function by using depth-first search to traverse the graph from **n1**; if you encounter **n2** along the way, then a path exists.

8. Write a function

```
int HopCount(nodeT *n1, nodeT *n2);
```

that returns the number of hops in the shortest path between the nodes **n1** and **n2**. If **n1** and **n2** are the same node, **HopCount** should return 0; if no path exists, **HopCount** should return -1. This function is easily implemented using breadth-first search.

9. Although the section entitled “Finding minimum paths” includes an implementation of Dijkstra’s algorithm, there is no surrounding infrastructure to turn that algorithm into an application. Create one by writing a C++ program that performs the following operations:

- Reads in a graph from a file.
- Allows the user to enter the names of two cities.
- Uses Dijkstra’s algorithm to find and display the minimum path.

The program will be easier to write if you break it down into the following phases:

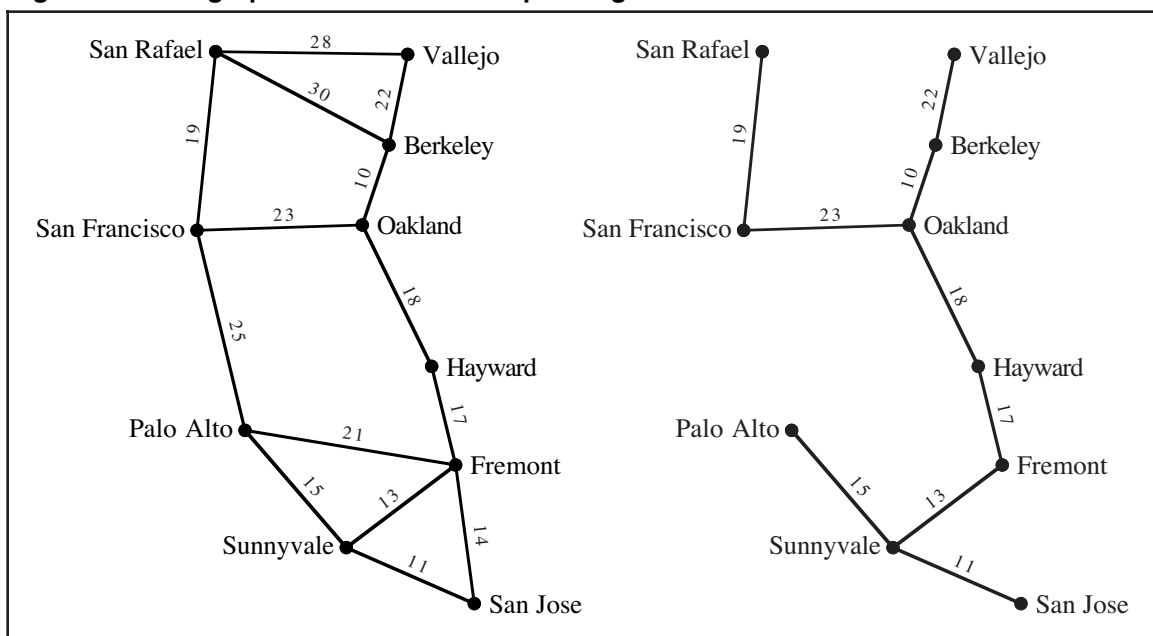
1. Complete the implementation of the **read** method described in exercises 1 and 3 so that you read the airline data from a file.
 2. Implement the priority queue extensions to the queue package, which are described in Chapter 10, exercise 4.
 3. Write a main program that draws together the independent pieces of your solution into a coherent whole.
10. Although Dijkstra's algorithm for finding minimum-cost paths has considerable practical importance, there are numerous other graph algorithms that have comparable commercial significance. In many cases, finding a minimum-cost path between two specific nodes is not as important as minimizing the cost of a network as a whole.

As an example, suppose that you are working for a company that is building a new cable system that connects 10 large cities in the San Francisco Bay area. Your preliminary research has provided you with cost estimates for laying new cable lines along a variety of possible routes. Those routes and their associated costs are shown in the graph on the left side of Figure 16-11. Your job is to find the cheapest way to lay new cables so that all the cities are connected through some path.

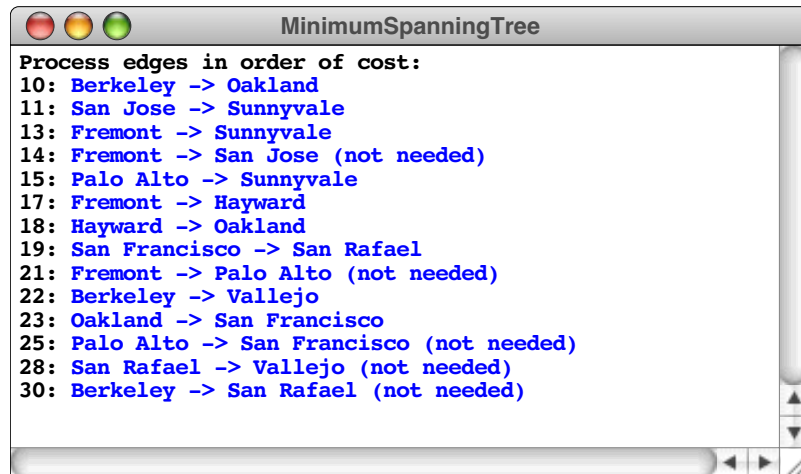
To minimize the cost, one of the things you need to avoid is laying a cable that forms a cycle in the graph. Such a cable would be unnecessary, because the cities it connects are already linked by some other path. If your goal is to find a set of arcs that connects the nodes of a graph at a minimum cost, you might as well leave such edges out. The remaining graph, given that it has no cycles, forms a tree. A tree that links all the nodes of a graph is called a **spanning tree**. The spanning tree in which the total cost associated with the arcs is as small as possible is called a **minimum spanning tree**. The cable-network problem described earlier in this exercise is therefore equivalent to finding the minimum spanning tree of the graph, which is shown in the right side of Figure 16-11.

There are many algorithms in the literature for finding a minimum spanning tree.

Figure 16-11 A graph and its minimum spanning tree



Of these, one of the simplest was devised by Joseph Kruskal in 1956. In Kruskal's algorithm, all you do is consider the arcs in the graph in order of increasing cost. If the nodes at the endpoints of the arc are unconnected, then you include this arc as part of the spanning tree. If, however, the nodes are already connected by a path, you ignore this arc entirely. The steps in the construction of the minimum spanning tree for the graph in Figure 16-11 are shown in the following sample run:



Write a function

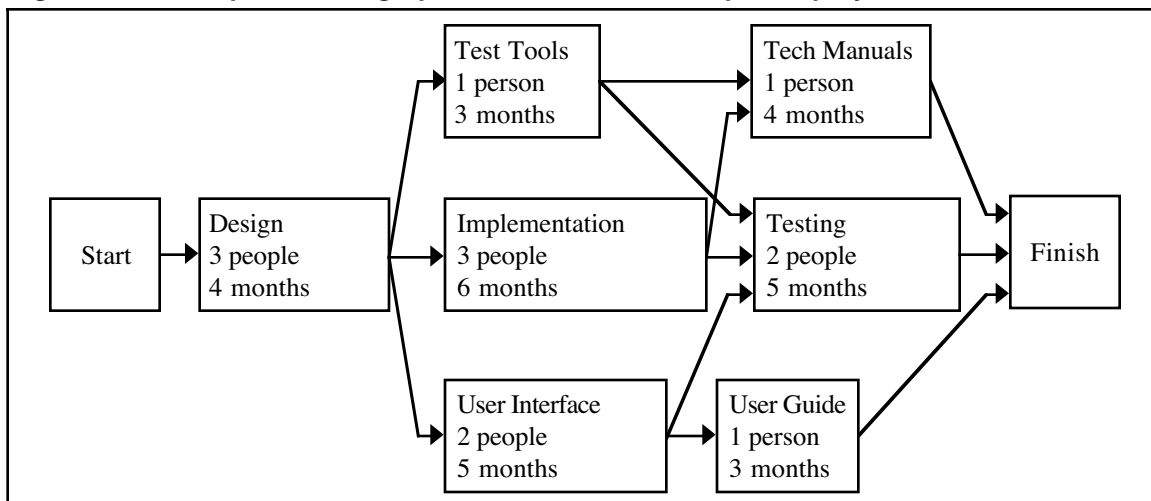
```
Graph<nodeT,arcT> MinimumSpanningTree(Graph<nodeT,arcT> & g);
```

that implements Kruskal's algorithm to find the minimum spanning tree. The function should return a new graph whose nodes match those in the original graph, but which includes only the arcs that are part of the minimum spanning tree.

11. *Work expands to fill the time available.*
—C. Northcote Parkinson, *Parkinson's Law*, 1957

Computers are often used in business applications to manage the scheduling of complex projects in which there are ordering relationships among the individual tasks. For example, a large software project might include the tasks illustrated in the graph in Figure 16-12:

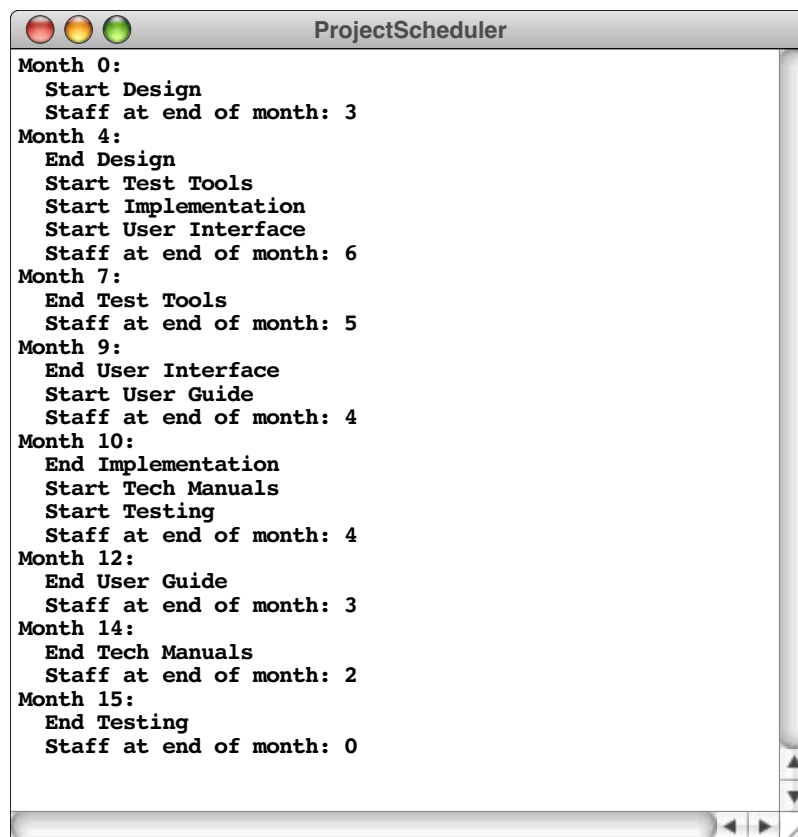
Figure 16-12 Simplified task graph for a software development project



The nodes in this graph represent individual tasks; the arcs represent dependencies. The structure of the project graph tells you, for example, that it is impossible to start the **Implementation** task before the **Design** task is complete. Similarly, before you begin the **Testing** task, you must first have completed each of the following tasks: **Test Tools**, **Implementation**, and **User Interface** tasks. On the other hand, the **Test Tools**, **Implementation**, and **User Interface** tasks can proceed in parallel. Moreover, the **User Guide** task can begin before the **Implementation** task is complete, because it depends only on the **User Interface** task.

Each project contains two special nodes—**Start** and **Finish**—that mark the beginning and end of the project. Except for these special nodes, every task node is associated with three pieces of information: the name of the task, the number of employees required, and the duration.

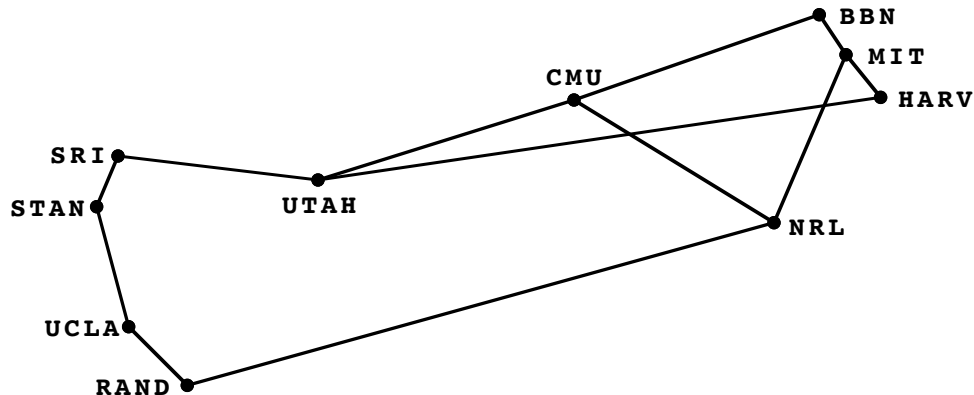
Write a program that uses the graph package to represent the type of data contained in project graphs such as the one shown in Figure 16-12. The first step in the process is to write the code necessary to read the data for a project graph from a file. To do so, you will have to design both the format of the data file and the data blocks needed to associate task data with the nodes in the graph. Once you have read in the project file, the next step is to write a program that generates a schedule for the project, showing when each task begins and ends, along with the total staffing level required at that time. For example, given the project graph from Figure 16-12, your program should produce a schedule that looks like this:



12. Use the algorithms described in section 16.7 to implement a new priority queue class that uses a heap instead of a linked list as its underlying representation. In doing so, it is necessary to define a distinct type for priority queues because the heap data

structure does not guarantee first-in/first-out behavior for items with equal priority. Thus, your heap-based implementation should define a new class called **PQueue** so that it is still possible to use the traditional queue mechanism.

13. Graph algorithms are often well suited to distributed implementations in which processing is performed at each node in the graph. In particular, such algorithms are used to find optimal transmission routes in a computer network. As an example, the following graph shows the first 10 nodes in the ARPANET—the network created by the Advanced Research Projects Agency (ARPA) of the U.S. Department of Defense—which was the forerunner of today’s much more sophisticated Internet:



Each node in the early ARPANET consisted of a small computer called an **Interface Message Processor**, or **IMP**. As part of the network operation, each IMP sent messages to its neighbors indicating the number of hops from that node to every other node, to the extent that the IMP possessed that information. By monitoring the messages coming in, each IMP could quickly develop useful routing information about the network as a whole.

To make this idea more concrete, imagine that every IMP maintains an array in which each index position corresponds to one of the nodes. When things are up and running, the array in the Stanford IMP (**STAN**) should have the following contents:

4	3	3	4	3	2	1	0	1	2
BBN	CMU	HARV	MIT	NRL	RAND	SRI	STAN	UCLA	UTAH

The interesting question, however, is not so much the contents of the array as it is how the network computes and maintains these counts. When a node is restarted, it has no knowledge of the complete network. In fact, the only information the Stanford node can determine on its own is that its own entry is 0 hops away. Thus, at start-up time, the array in the **STAN** node looks like this:

?	?	?	?	?	?	?	0	?	?
BBN	CMU	HARV	MIT	NRL	RAND	SRI	STAN	UCLA	UTAH

The routing algorithm then proceeds by letting each node forward its routing array to its neighbors. The Stanford IMP, for example, sends its array off to SRI and UCLA. It also receives similar messages from its neighbors. If the IMP at UCLA has just started up as well, it might send a message containing the array

?	?	?	?	?	?	?	?	0	?
BBN	CMU	HARV	MIT	NRL	RAND	SRI	STAN	UCLA	UTAH

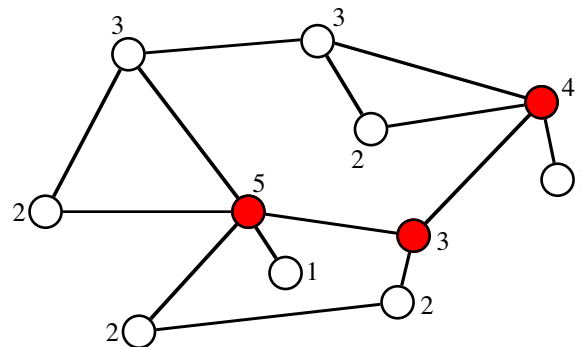
This message provides the Stanford node with some interesting information. If its neighbor can get to UCLA in 0 hops, then the Stanford node can get there in 1. As a result, the Stanford node can update its own routing array as follows:

?	?	?	?	?	?	?	0	1	?
BBN	CMU	HARV	MIT	NRL	RAND	SRI	STAN	UCLA	UTAH

In general, whenever any node gets a routing array from its neighbor, all it has to do is go through each of the known entries in the incoming array and replace the corresponding entry in its own array with the incoming value plus one, unless its own entry is already smaller. In a very short time, the routing arrays throughout the entire network will have the correct information.

Write a program that uses the graph package to simulate the calculations of this routing algorithm on a network of nodes.

14. A **dominating set** of a graph is a subset of the nodes such that those nodes along with their immediate neighbors constitute all graph nodes. That is, every node in the graph is either in the dominating set or is a neighbor of a node in the dominating set. In the graph diagrammed below—in which each node is labeled with the number of neighbors to facilitate tracing the algorithm—the filled-in nodes constitute a dominating set for the graph. Other dominating sets are also possible.



Ideally, you would like to be able to find the smallest possible dominating set, but that is known to be a computationally difficult task—too expensive for most graphs. The following algorithm usually finds a relatively small dominating set, even though it does not always produce the optimal result:

1. Start with an empty set S .
2. Consider each graph node in order of decreasing degree. In other words, you want to start with the node that has the most neighbors and then work down through the nodes with fewer neighbors. If two or more nodes have the same degree, you can process them in any order.
3. If the node you chose in step 2 is not redundant, add it to S . A node is *redundant* if it and all of its neighbors are neighbors of some node already in S .
4. Continue until S dominates the entire graph

Write a template function

```
template <NodeType, ArcType>
Set<NodeType *> FindDominatingSet (Graph<NodeType, ArcType> & g);
```

that uses this algorithm to find a small dominating set for the graph g .

Appendix A

Library Interfaces

This appendix contains full listings for the interfaces used in this text. Given that this appendix is likely to be used primarily as a reference, it makes sense to list the interfaces in alphabetical order rather than any logical grouping. The interfaces and their page numbers are:

bst.h	608
cmpfn.h	611
extgraph.h	612
genlib.h	622
graph.h	623
graphics.h	627
grid.h	630
lexicon.h	634
map.h	638
queue.h	642
random.h	644
scanner.h	646
set.h	652
simpio.h	656
sound.h	657
stack.h	658
strutils.h	660
vector.h	662

```
/*
 * File: bst.h
 * -----
 * This interface file contains the BST class template, an
 * implementation of a general binary search tree.
 */

#ifndef _bst_h
#define _bst_h

#include "genlib.h"
#include "cmpfn.h"
#include "stack.h"
#include "foreach.h"

/*
 * Class: BST
 * -----
 * This interface defines a class template for a binary search tree.
 * For maximum generality, the BST is supplied as a class template.
 * The element type is set by the client. The client specializes
 * the tree for specific type, e.g. BST<int> or BST<studentT>.
 * The one requirement on the element type is that the client must
 * supply a comparison fn that compares two elements (or be willing
 * to use the default comparison function that relies on < and ==).
 */

template <typename ElemType>
class BST {
public:

    class Iterator;          /* Forward reference */

    /*
     * Constructor: BST
     * Usage: BST<int> bst;
     *         BST<song> songs(CompareSong)
     *         BST<string> *bp = new BST<string>;
     * -----
     * The constructor initializes a new empty binary search tree.
     * The one argument is a comparison function, which is called
     * to compare data values. This argument is optional, if not
     * given, the OperatorCmp function from cmpfn.h is used, which
     * applies the built-in operator < to its operands. If the
     * behavior of < on your ElemType is defined and sufficient,
     * you do not need to supply your own comparison function.
     */

        BST(int (*cmpFn)(ElemType one, ElemType two) = OperatorCmp);

    /*
     * Destructor: ~BST
     * Usage: delete bp;
     * -----
     * The destructor deallocates storage for this tree.
     */

        ~BST();
};
```



```
/*
 * Method: size
 * Usage: count = bst.size();
 * -----
 * Returns the number of elements in this tree.
 */
int size();

/*
 * Method: isEmpty
 * Usage: if (bst.isEmpty())...
 * -----
 * Returns true if this tree contains no elements, false otherwise.
 */
bool isEmpty();

/*
 * Method: find
 * Usage:  if (bst.find(key) != NULL) . . .
 * -----
 * Applies the binary search algorithm to find a particular key
 * in this tree.  If key appears in the tree, find returns a
 * pointer to the data in that node; otherwise, find returns NULL.
 */
ElemType *find(ElemType key);

/*
 * Method: add
 * Usage: bst.add(elem);
 * -----
 * Adds a new node to this tree.  If a node with the same value
 * already exists, the contents are overwritten with the new copy.
 * The add method returns true if a new copy is allocated.
 */
bool add(ElemType elem);

/*
 * Method: remove
 * Usage: bst.remove(key);
 * -----
 * Removes a node in this tree that matches the specified key.
 * If no match is found, no changes are made.  The remove method
 * returns true if the removal actually occurs.
 */
bool remove(ElemType key);

/*
 * Method: clear
 * Usage: bst.clear();
 * -----
 * Removes all elements from this tree.
 */
void clear();
```

```
/*
 * Method: mapAll
 * Usage: bst.mapAll(Print);
 *        bst.mapAll(PrintToFile, outputStream);
 * -----
 * Iterates through this tree and calls the function fn once for
 * each element. The order is determined by an InOrder walk of
 * the tree. The second form allows the client to pass data of
 * any type to the callback function.
 */

void mapAll(void (*fn)(ElemType elem));

template <typename ClientDataType>
void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
            ClientDataType & data);

/*
 * Method: iterator
 * Usage: iter = bst.iterator();
 * -----
 * Creates an iterator that allows the client to iterate through
 * the elements in this binary search tree. The order of elements
 * produced by the iterator is that of an InOrder walk of the tree.
 *
 * The idiomatic code for accessing elements using an iterator is
 * to create the iterator from the collection and then enter a loop
 * that calls next() while hasNext() is true, like this:
 *
 *     BST<string>::Iterator iter = bst.iterator();
 *     while (iter.hasNext()) {
 *         string key = iter.next();
 *         . . .
 *     }
 *
 * This pattern can be abbreviated as follows:
 *
 *     foreach (string key in bst) {
 *         . . .
 *     }
 */

Iterator iterator();

private:
#include "private/bst.h"
};
#include "private/bst.cpp"
#endif
```

```
/*
 * File: cmpfn.h
 * -----
 * This interface exports a comparison function template.
 */

#ifndef _cmpfn_h
#define _cmpfn_h

/*
 * Function template: OperatorCmp
 * Usage:  int sign = OperatorCmp(v1, v2);
 * -----
 * This function template is a generic function to compare two
 * values using the built-in == and < operators. It is supplied
 * as a convenience for those situations where a comparison
 * function is required, and the type has a built-in ordering
 * that you would like to use.
 */

template <typename Type>
int OperatorCmp(Type v1, Type v2) {
    if (v1 == v2) return 0;
    if (v1 < v2) return -1;
    return 1;
}

#endif
```

```
/*
 * File: extgraph.h
 * -----
 * This interface is the extended graphics interface. It
 * includes all the facilities in graphics.h, plus several
 * additional functions designed to support more sophisticated,
 * interactive graphics.
 */

#ifndef _extgraph_h
#define _extgraph_h

#include "genlib.h"

/* Exported functions */

/* Section 1 -- Basic functions from graphics.h */
#include "graphics.h"

/* Section 2 -- Elliptical arcs */

/*
 * Function: DrawEllipticalArc
 * Usage: DrawEllipticalArc(rx, ry, start, sweep);
 * -----
 * This procedure draws an elliptical arc. It is exactly the
 * same in its operation as DrawArc in the graphics.h interface,
 * except that the radius is different along the two axes.
 */

void DrawEllipticalArc(double rx, double ry,
                      double start, double sweep);

/* Section 3 -- Graphical regions */

/*
 * Functions: StartFilledRegion, EndFilledRegion
 * Usage: StartFilledRegion(density);
 *        . . . other calls . . .
 *        EndFilledRegion();
 * -----
 * These calls make it possible to draw filled shapes on the
 * display. After calling StartFilledRegion, any calls to
 * DrawLine and DrawArc are used to create a shape definition and
 * do not appear on the screen until EndFilledRegion is called.
 * The lines and arcs must be consecutive, in the sense that each
 * new element must start where the last one ended. MovePen calls
 * may occur at the beginning or the end of the region, but not in
 * the interior. When EndFilledRegion is called, the entire region
 * appears on the screen, with its interior filled in. The density
 * parameter is a number between 0 and 1 and indicates how the dot
 * density to be used for the fill pattern. If density is 1, the
 * shape will be filled in a solid color; if it is 0, the fill will
 * be invisible. In between, the implementation will use a dot
 * pattern that colors some of the screen dots but not others.
 */

void StartFilledRegion(double density = 1.0);
void EndFilledRegion();
```

```
/* Section 4 -- String functions */

/*
 * Function: DrawTextString
 * Usage: DrawTextString(text);
 * -----
 * This function displays the string text at the current point
 * in the current font and size. The current point is updated
 * so that the next DrawTextString command would continue from
 * the next character position. The string may not include the
 * newline character.
 */

void DrawTextString(string text);

/*
 * Function: TextStringWidth
 * Usage: w = TextStringWidth(text);
 * -----
 * This function returns the width of the text string if displayed
 * at the current font and size.
 */

double TextStringWidth(string text);

/*
 * Function: SetFont
 * Usage: SetFont(font);
 * -----
 * This function sets a new font according to the font string,
 * which is case-independent. Different systems support different
 * fonts, although common ones like "Times" and "Courier" are often
 * supported. Initially, the font is set to "Default" which is
 * always supported, although the underlying font is system
 * dependent. If the font name is unrecognized, no error is
 * generated, and the font remains unchanged. If you need to
 * detect this condition, you can call GetFont to see if the
 * change took effect. By not generating an error in this case,
 * programs become more portable.
 */

void SetFont(string font);

/*
 * Function: GetFont
 * Usage: font = GetFont();
 * -----
 * This function returns the current font name as a string.
 */

string GetFont();
```

```
/*
 * Function: SetPointSize
 * Usage: SetPointSize(size);
 * -----
 * This function sets a new point size.  If the point size is
 * not supported for a particular font, the closest existing
 * size is selected.
 */

void SetPointSize(int size);

/*
 * Function: GetPointSize
 * Usage: size = GetPointSize();
 * -----
 * This function returns the current point size.
 */

int GetPointSize();

/*
 * Text style constants
 * -----
 * The constants Bold and Italic are used in the SetStyle
 * command to specify the desired text style.  They may also
 * be used in combination by adding these constants together,
 * as in Bold + Italic.  The constant Normal indicates the
 * default style.
 */

#define Normal 0
#define Bold 1
#define Italic 2

/*
 * Function: SetStyle
 * Usage: SetStyle(style);
 * -----
 * This function establishes the current style properties
 * for text based on the parameter style, which is an integer
 * representing the sum of any of the text style constants.
 */

void SetStyle(int style);

/*
 * Function: GetStyle
 * Usage: style = GetStyle();
 * -----
 * This function returns the current style.
 */

int GetStyle();
```

```
/*
 * Functions: GetFontAscent, GetFontDescent, GetFontHeight
 * Usage: ascent = GetFontAscent();
 *        descent = GetFontDescent();
 *        height = GetFontHeight();
 * -----
 * These functions return properties of the current font that are
 * used to calculate how to position text vertically on the page.
 * The ascent of a font is the distance from the baseline to the
 * top of the largest character; the descent is the maximum
 * distance any character extends below the baseline. The height
 * is the total distance between two lines of text, including the
 * interline space (which is called leading).
 *
 * Examples:
 *   To change the value of y so that it indicates the next text
 *   line, you need to execute
 *
 *       y -= GetFontHeight();
 *
 *   To center text vertically around the coordinate y, you need
 *   to start the pen at
 *
 *       y - GetFontAscent() / 2
 */

double GetFontAscent();
double GetFontDescent();
double GetFontHeight();

/* Section 5 -- Mouse support */

/*
 * Functions: GetMouseX, GetMouseY
 * Usage: x = GetMouseX();
 *        y = GetMouseY();
 * -----
 * These functions return the x and y coordinates of the mouse,
 * respectively. The coordinate values are real numbers measured
 * in inches from the origin and therefore match the drawing
 * coordinates.
 */

double GetMouseX();
double GetMouseY();

/*
 * Functions: MouseButtonIsDown
 * Usage: if (MouseButtonIsDown()) . . .
 * -----
 * This function returns true if the mouse button is currently
 * down. For maximum compatibility among implementations, the
 * mouse is assumed to have one button. If the mouse has more
 * than one button, this function returns true if any button
 * is down.
 */

bool MouseButtonIsDown();
```

```
/*
 * Functions: WaitForMouseDown, WaitForMouseUp
 * Usage: WaitForMouseDown();
 *         WaitForMouseUp();
 * -----
 * The WaitForMouseDown function waits until the mouse button
 * is pressed and then returns. WaitForMouseUp waits for the
 * button to be released.
 */

void WaitForMouseDown();
void WaitForMouseUp();

/* Section 6 -- Color support */

/*
 * Function: SetPenColor
 * Usage: SetPenColor(color);
 * -----
 * This function sets the color of the pen used for any drawing,
 * including lines, text, and filled regions. The color is a
 * string, which will ordinarily be one of the following
 * predefined color names:
 *
 *     Black, Dark Gray, Gray, Light Gray, White,
 *     Red, Yellow, Green, Cyan, Blue, Magenta
 *
 * The first line corresponds to standard gray scales and the
 * second to the primary and secondary colors of light. The
 * built-in set is limited to these colors because they are
 * likely to be the same on all hardware devices. For finer
 * color control, you can use the DefineColor function to
 * create new color names as well.
 */

void SetPenColor(string color);

/*
 * Function: SetPenColorRGB
 * Usage: SetPenColorRGB(1.0, 0.5, 0.0);
 * -----
 * This function sets the color of the pen used for any drawing,
 * including lines, text, and filled regions. The color is
 * specified by supplying intensity levels for the colors red, green,
 * and blue, which are the primary colors of light. The
 * color values are provided as real numbers between 0 and 1,
 * indicating the intensity of that color. This version is
 * useful when you don't need to keep the color around under a
 * defined name (for that, use DefineColor below).
 */

void SetPenColorRGB(double red, double green, double blue);
```



```
/*
 * Function: GetPenColor
 * Usage: color = GetPenColor();
 * -----
 * This function returns the current pen color as a string.
 */

string GetPenColor();

/*
 * Function: DefineColor
 * Usage: DefineColor(name, red, green, blue);
 * -----
 * This function allows the client to define a new color name
 * by supplying intensity levels for the colors red, green,
 * and blue, which are the primary colors of light. The
 * color values are provided as real numbers between 0 and 1,
 * indicating the intensity of that color. For example,
 * the predefined color Magenta has full intensity red and
 * blue but no green and is therefore defined as:
 *
 *     DefineColor("Magenta", 1, 0, 1);
 *
 * DefineColor allows you to create intermediate colors on
 * many displays, although the results vary significantly
 * depending on the hardware. For example, the following
 * usually gives a reasonable approximation of brown:
 *
 *     DefineColor("Brown", .35, .20, .05);
 */

void DefineColor(string name,
                 double red, double green, double blue);

/* Section 7 -- Pictures */

/*
 * Function: DrawNamedPicture
 * Usage: DrawNamedPicture("bird.jpg");
 * -----
 * This function looks for a image file with the specified
 * name in the "Pictures" subdirectory of the project. It
 * displays this image in the graphics window so that the
 * lower left corner of the image appears at the current point.
 * The function generates an error if the named picture cannot
 * be found. Note that, although the interface presented here
 * is the same for all systems, the format used for the resource
 * itself may not be supported across all platforms. The ones
 * that are guaranteed to work are bitmaps (.bmp), GIFs (.gif),
 * and JPEGs (.jpg). Others may work, but those are the only
 * ones that are guaranteed.
 */

void DrawNamedPicture(string name);
```

```
/*
 * Functions: GetPictureWidth, GetPictureHeight
 * Usage: w = GetPictureWidth("ball.gif");
 *        h = GetPictureHeight("ball.gif");
 * -----
 * These functions return the width and height of the named
 * picture, as described in the comments for DrawNamedPicture.
 */

double GetPictureWidth(string name);
double GetPictureHeight(string name);

/* Section 8 -- Miscellaneous functions */

/*
 * Function: SetCoordinateSystem
 * Usage: SetCoordinateSystem("cartesian");
 *        SetCoordinateSystem("screen");
 * -----
 * Sets the coordinate system used by the graphics library.
 * The argument to SetCoordinateSystem is either the string
 * "cartesian", which specifies a classical coordinate system
 * in which the origin is in the lower left and coordinates are
 * measured in inches, or "screen", which specifies a more
 * contemporary screen-based coordinate system in which the
 * origin is in the upper left and coordinates are measured
 * in pixels. The strings are considered without regard to
 * case. Unlike the other functions in the library, this call
 * must be made before the call to InitGraphics.
 */

void SetCoordinateSystem(string system);

/*
 * Function: GetCoordinateSystem
 * Usage: mode = GetCoordinateSystem();
 * -----
 * Returns the coordinate system in effect for the graphics
 * library, which is either "cartesian" or "screen".
 */

string GetCoordinateSystem();
```

```
/*
 * Function: SetEraseMode
 * Usage: SetEraseMode(true);
 *        SetEraseMode(false);
 * -----
 * The SetEraseMode function sets the value of the internal
 * erasing flag. Setting this flag is similar to setting the
 * color to "White" in its effect but does not affect the
 * current color setting. When erase mode is set to false,
 * normal drawing is restored, using the current color.
 */

void SetEraseMode(bool mode);

/*
 * Function: GetEraseMode
 * Usage: mode = GetEraseMode();
 * -----
 * This function returns the current state of the erase mode flag.
 */

bool GetEraseMode();

/*
 * Function: SetWindowTitle
 * Usage: SetWindowTitle(title);
 * -----
 * This function sets the title of the graphics window, if such
 * an operation is possible on the display. If it is not possible
 * for a particular implementation, the call is simply ignored.
 * This function may be called prior to the InitGraphics call to
 * set the initial name of the window.
 */

void SetWindowTitle(string title);

/*
 * Function: GetWindowTitle
 * Usage: title = GetWindowTitle();
 * -----
 * This function returns the title of the graphics window. If the
 * implementation does not support titles, this call returns the
 * empty string.
 */

string GetWindowTitle();

/*
 * Function: UpdateDisplay
 * Usage: UpdateDisplay();
 * -----
 * This function initiates an immediate update of the graphics
 * window and is necessary for animation. Ordinarily, the
 * graphics window is only updated when the program waits for
 * user input.
 */

void UpdateDisplay();
```

```
/*
 * Function: Pause
 * Usage: Pause(seconds);
 * -----
 * The Pause function updates the graphics window and then
 * pauses for the indicated number of seconds. This function
 * is useful for animation where the motion would otherwise
 * be too fast.
 */

void Pause(double seconds);

/*
 * Function: ExitGraphics
 * Usage: ExitGraphics();
 * -----
 * The ExitGraphics function closes the graphics window and
 * exits from the application without waiting for any additional
 * user interaction.
 */

void ExitGraphics();

/*
 * Functions: SaveGraphicsState, RestoreGraphicsState
 * Usage: SaveGraphicsState();
 *         . . . graphical operations . . .
 *         RestoreGraphicsState();
 * -----
 * The SaveGraphicsState function saves the current graphics
 * state (the current pen position, the font, the point size,
 * and the erase mode flag) internally, so that they can be
 * restored by the next RestoreGraphicsState call. These two
 * functions must be used in pairs but may be nested to any depth.
 */

void SaveGraphicsState();
void RestoreGraphicsState();

/*
 * Functions: GetFullScreenWidth, GetFullScreenHeight
 * Usage: width = GetFullScreenWidth();
 *        height = GetFullScreenHeight();
 * -----
 * These functions return the height and width of the entire
 * display screen, not the graphics window. Their only
 * significant use is for applications that need to adjust
 * the size of the graphics window based on available screen
 * space. These functions may be called before InitGraphics
 * has been called.
 */

double GetFullScreenWidth();
double GetFullScreenHeight();
```

```
/*
 * Functions: SetWindowSize
 * Usage: SetWindowSize(width, height);
 * -----
 * This function sets the window size to the indicated dimensions,
 * if possible. This function should be called before the graphics
 * window is created by InitGraphics. Attempts to change the size
 * of an existing window are ignored by most implementations.
 * This function should be used sparingly because it reduces the
 * portability of applications, particularly if the client
 * requests more space than is available on the screen.
 */

void SetWindowSize(double width, double height);

/*
 * Functions: GetXResolution, GetYResolution
 * Usage: xres = GetXResolution();
 *        yres = GetYResolution();
 * -----
 * These functions return the number of pixels per inch along
 * each of the coordinate directions and are useful for applications
 * in which it is important for short distances to be represented
 * uniformly in terms of dot spacing. Even though the x and y
 * resolutions are the same for most displays, clients should
 * not rely on this property.
 *
 * Note: Lines in the graphics library are one pixel unit wide and
 * have a length that is always one pixel longer than you might
 * expect. For example, the function call
 *
 *     DrawLine(2 / GetXResolution(), 0);
 *
 * draws a line from the current point to the point two pixels
 * further right, which results in a line of three pixels.
 */

double GetXResolution();
double GetYResolution();

#endif
```

```
/*
 * File: genlib.h
 * -----
 * This header file is included in all programs written for
 * Stanford's C++ courses and provides a few common definitions.
 * Note that this header has a "using namespace std" clause.
 * If a file includes the genlib.h header, it can then use
 * features from the std namespace without qualifying by scope.
 */

#ifndef _genlib_h
#define _genlib_h

#include <string>
#include <exception>
using namespace std;

/*
 * Class: ErrorException
 * -----
 * This exception is raised by calls to the Error function, which
 * makes it possible for clients to respond to error conditions
 * without having their programs simply exit.
 */

class ErrorException : public exception {
public:
    ErrorException(string msg);
    virtual ~ErrorException() throw ();
    virtual string getMessage();
private:
    string msg;
};

/*
 * Function: Error
 * Usage: Error(msg);
 * -----
 * Error is used to signal an error condition in a program. It
 * first attempts to throw an ErrorException. If that fails,
 * it outputs the error message string to the cerr stream and
 * then exits the program with a status code indicating failure.
 */

void Error(string str);

/*
 * Function macro: main
 * -----
 * The purpose of this macro definition is to rename the student
 * main to Main in order to allow a custom main defined in our
 * libraries to configure the application before passing control
 * back to the student program.
 */

#define main Main

#endif
```

```
/*
 * File: graph.h
 * -----
 * Extended interface for the graph package. This interface exports
 * both a low-level graphT type and a parameterized Graph class.
 */

#ifndef _graph_h
#define _graph_h

#include "set.h"
#include "map.h"

struct nodeT;      /* Forward references to these two types so */
struct arcT;       /* that the C++ compiler can recognize them. */

/*
 * Type: graphT
 * -----
 * This type represents a graph and consists of two sets -- a set
 * of nodes and a set of arcs -- along with a map that creates an
 * association between names and nodes.
 */

struct graphT {
    Set<nodeT *> nodes;
    Set<arcT *> arcs;
    Map<nodeT *> nodeMap;
};

/*
 * Type: nodeT
 * -----
 * This type is the supplied type for a node in a graph. Clients
 * may substitute their own type, as described in the notes for
 * the Graph class.
 */

struct nodeT {
    string name;
    Set<arcT *> arcs;
};

/*
 * Type: arcT
 * -----
 * This type is the supplied type for an arc in a graph. Clients
 * may substitute their own type, as described in the notes for
 * the Graph class.
 */

struct arcT {
    nodeT *start;
    nodeT *finish;
    double cost;
};
```

```
/*
 * Class: Graph<NodeType,ArcType>
 * -----
 * This class represents a graph with the specified node and arc
 * types. The NodeType and ArcType parameters indicate the record
 * or object types used for nodes and arcs, respectively. These
 * types can contain any fields or methods required by the client,
 * but must also contain the following public fields required by
 * the Graph package itself:
 *
 * The NodeType definition must include:
 *   - A string field called name
 *   - A Set<ArcType *> field called arcs
 *
 * The ArcType definition must include:
 *   - A NodeType * field called start
 *   - A NodeType * field called finish
 *   - A double field called cost
 */

template <typename NodeType,typename ArcType>
class Graph {

public:

/*
 * Constructor: Graph
 * Usage: Graph<NodeType,ArcType> g;
 * -----
 * Declares a new Graph object named g.
 */
    Graph();

/*
 * Destructor: ~Graph
 * Usage: (usually implicit)
 * -----
 * Frees the internal storage allocated for the nodes and
 * arcs in the graph.
 */
    ~Graph();

/*
 * Method: clear
 * Usage: g.clear();
 * -----
 * Frees the storage for all nodes and arcs in the graph and
 * reinitializes the graph to be empty.
 */
    void clear();
}
```



```
/*
 * Method: addNode
 * Usage: g.addNode(name);
 *         g.addNode(node);
 * -----
 * Adds a node to the graph. The first version of this method
 * creates a new node of the appropriate type and initializes its
 * fields; the second assumes that the client has already created
 * the node and simply adds it to the graph. Both versions of this
 * method return a pointer to the node in case the client needs to
 * capture this value.
 */

NodeType *addNode(string name);
NodeType *addNode(NodeType *node);

/*
 * Method: addArc
 * Usage: g.addArc(s1, s2);
 *         g.addArc(n1, n2);
 *         g.addArc(arc);
 * -----
 * Adds an arc to the graph. The endpoints of the arc can be
 * specified either as strings indicating the names of the nodes
 * or as pointers to the node structures. Alternatively, the
 * client can create the arc structure explicitly and pass that
 * pointer to the addArc method. All three of these versions
 * return a pointer to the arc in case the client needs to
 * capture this value.
 */

ArcType *addArc(string s1, string s2);
ArcType *addArc(NodeType *n1, NodeType *n2);
ArcType *addArc(ArcType *arc);

/*
 * Method: isConnected
 * Usage: if (g.isConnected(n1, n2)) . . .
 *         if (g.isConnected(s1, s2)) . . .
 * -----
 * Returns true if the graph contains an arc from n1 to n2. As
 * in the addArc method, nodes can be specified either as node
 * pointers or by name.
 */

bool isConnected(NodeType *n1, NodeType *n2);
bool isConnected(string s1, string s2);

/*
 * Method: getNode
 * Usage: NodeType *node = g.getNode(name);
 * -----
 * Looks up a node in the name table attached to the graph and
 * returns a pointer to that node. If no node with the specified
 * name exists, getNode returns NULL.
 */

NodeType *getNode(string name);
```

```
/*
 * Method: getNodeSet
 * Usage: foreach (NodeType *node in g.getNodeSet()) . . .
 * -----
 * Returns the set of all nodes in the graph. This method returns
 * a reference to the set rather than a copy of the set, which makes
 * it possible to iterate over its elements.
 */

Set<NodeType *> & getNodeSet();

/*
 * Method: getArcSet
 * Usage: foreach (ArcType *arc in g.getArcSet()) . . .
 *        foreach (ArcType *arc in g.getArcSet(node)) . . .
 * -----
 * Returns the set of all arcs in the graph or, in the second form,
 * the arcs that start at the specified node. This method returns
 * a reference to the set rather than a copy of the set, which makes
 * it possible to iterate over its elements.
 */

Set<ArcType *> & getArcSet();
Set<ArcType *> & getArcSet(NodeType *node);

private:

#include "private/graph.h"

};

#include "private/graph.cpp"

#endif
```

```
/*
 * File: graphics.h
 * -----
 * This interface provides access to a simple library of
 * functions that make it possible to draw lines and arcs
 * on the screen. This interface presents a portable
 * abstraction that can be used with a variety of window
 * systems implemented on different hardware platforms.
 */

#ifndef _graphics_h
#define _graphics_h

/*
 * Overview
 * -----
 * This library provides several functions for drawing lines
 * and circular arcs in a region of the screen that is
 * defined as the "graphics window." Once drawn, these
 * lines and arcs stay in their position, which means that
 * the package can only be used for static pictures and not
 * for animation.
 *
 * Individual points within the window are specified by
 * giving their x and y coordinates. These coordinates are
 * real numbers measured in inches, with the origin in the
 * lower left corner, as it is in traditional mathematics.
 *
 * The calls available in the package are listed below. More
 * complete descriptions are included with each function
 * description.
 *
 *   InitGraphics();
 *   MovePen(x, y);
 *   DrawLine(dx, dy);
 *   DrawArc(r, start, sweep);
 *   width = GetWindowWidth();
 *   height = GetWindowHeight();
 *   x = GetCurrentX();
 *   y = GetCurrentY();
 */

/*
 * Function: InitGraphics
 * Usage: InitGraphics();
 * -----
 * This procedure creates the graphics window on the screen.
 * The call to InitGraphics must precede any calls to other
 * functions in this package and must also precede any printf
 * output. In most cases, the InitGraphics call is the first
 * statement in the function main.
 */

void InitGraphics();
```

```
/*
 * Function: MovePen
 * Usage: MovePen(x, y);
 * -----
 * This procedure moves the current point to the position (x, y),
 * without drawing a line. The model is that of the pen being
 * lifted off the graphics window surface and then moved to its
 * new position.
 */

void MovePen(double x, double y);

/*
 * Function: DrawLine
 * Usage: DrawLine(dx, dy);
 * -----
 * This procedure draws a line extending from the current point by
 * moving the pen dx inches in the x direction and dy inches in the
 * y direction. The final position becomes the new current point.
 */

void DrawLine(double dx, double dy);

/*
 * Function: DrawArc
 * Usage: DrawArc(r, start, sweep);
 * -----
 * This procedure draws a circular arc, which always begins at
 * the current point. The arc itself has radius r, and starts at
 * the angle specified by the parameter start, relative to the
 * center of the circle. This angle is measured in degrees
 * counterclockwise from the 3 o'clock position along the x-axis,
 * as in traditional mathematics. For example, if start is 0,
 * the arc begins at the 3 o'clock position; if start is 90,
 * the arc begins at the 12 o'clock position; and so on. The
 * fraction of the circle drawn is specified by the parameter
 * sweep, which is also measured in degrees. If sweep is 360,
 * DrawArc draws a complete circle; if sweep is 90, it draws a
 * quarter of a circle. If the value of sweep is positive, the
 * arc is drawn counterclockwise from the current point. If
 * sweep is negative, the arc is drawn clockwise from the current
 * point. The current point at the end of the DrawArc operation
 * is the final position of the pen along the arc.
 *
 * Examples:
 *   DrawArc(r, 0, 360)    Draws a circle to the left of the
 *                          current point.
 *   DrawArc(r, 90, 180)   Draws the left half of a semicircle
 *                          starting from the 12 o'clock position.
 *   DrawArc(r, 0, 90)     Draws a quarter circle from the 3
 *                          o'clock to the 12 o'clock position.
 *   DrawArc(r, 0, -90)    Draws a quarter circle from the 3
 *                          o'clock to the 6 o'clock position.
 *   DrawArc(r, -90, -90)  Draws a quarter circle from the 6
 *                          o'clock to the 9 o'clock position.
 */

void DrawArc(double r, double start, double sweep);
```

```
/*
 * Functions: GetWindowWidth, GetWindowHeight
 * Usage: width = GetWindowWidth();
 *         height = GetWindowHeight();
 * -----
 * These functions return the width and height of the graphics
 * window, in inches.
 */

double GetWindowWidth();
double GetWindowHeight();

/*
 * Functions: GetCurrentX, GetCurrentY
 * Usage: x = GetCurrentX();
 *        y = GetCurrentY();
 * -----
 * These functions return the current x and y positions.
 */

double GetCurrentX();
double GetCurrentY();

#endif
```

```
/*
 * File: grid.h
 * -----
 * This interface defines a class template to store a homogeneous
 * 2-D indexed collection. The basic operations are similar to
 * those defined for built-in multidimensional arrays with the
 * added features of dynamic resizing, deep copying, and
 * bounds-checking on indexes.
 *
 * Here is some sample code showing use of a Grid of strings:
 *
 *   Grid<string> g(4, 6);  <= creates 4x6 grid, each string empty
 *
 *   g[0][2] = "blue";      <= can set elemes using [] or setAt
 *   g.setAt(2, 3, "red");
 *
 *   for (int row = 0; row < g.numRows(); row++) {
 *       for (int col = 0; col < g.numCols(); col++) {
 *           cout << g[row][col] << " ";
 *       }
 *   }
 *
 * Access to individual elements in the grid is provided via the
 * setAt/getAt methods, as well as an overloaded operator[].
 * Either refers to cells by row/col location; indexes start at 0
 * in each dimension.
 */

#ifndef _grid_h
#define _grid_h

#include "genlib.h"
#include "strutils.h"
#include "foreach.h"

/*
 * Class: Grid
 * -----
 * The class Grid stores an indexed, two-dimensional collection.
 * It is supplied as a class template. The client specializes
 * the grid to hold values of a specific type. Because the class
 * is defined with private data members, clients have no access
 * to the underlying data members and can only manipulate a grid
 * object through its public methods.
 */

template <typename ElemType>
class Grid {
public:
    /* Forward references */
    class GridRow;
    class Iterator;
```

```
/*
 * Constructor: Grid
 * Usage: Grid<bool> grid;
 * -----
 * Initializes a new empty grid with 0 rows and 0 cols. Clients
 * that use this form of the constructor will need to call resize
 * to set the dimensions of the grid.
 */
    Grid();

/*
 * Constructor: Grid
 * Usage: Grid<bool> grid(10, 20);
 * -----
 * Initializes a new grid with the specified number of rows and
 * columns. Each element in the grid is assigned a value equal
 * to the default for that element type (e.g., for strings it would
 * be empty string, for ints, the default value is uninitialized).
 * Raises an error if numRows or numCols is negative.
 */
    Grid(int numRows, int numCols);

/*
 * Destructor: ~Grid
 * Usage: (usually implicit)
 * -----
 * Frees the storage associated with this grid.
 */
    ~Grid();

/*
 * Methods: numRows, numCols
 * Usage: numRows = grid.numRows();
 * -----
 * These methods returns the number of rows or columns in this grid.
 */
    int numRows();
    int numCols();

/*
 * Method: resize
 * Usage: grid.resize(5, 10);
 * -----
 * Sets the number of rows and columns in this grid to the specified
 * values. Any previous grid contents are discarded. Each element
 * in the resized grid has value equal to the default for that
 * element type. Raises an error if numRows or numCols is negative.
 */
    void resize(int numRows, int numCols);
```

```
/*
 * Method: getAt
 * Usage: val = grid.getAt(3, 10);
 * -----
 * Returns the element at specified row/col location in this grid.
 * Locations are indexed starting from 0.  Raises an error if row
 * is outside the range [0, numRows()-1] or if col is outside
 * the range [0, numCols()-1].
 */

ElemType getAt(int row, int col);

/*
 * Method: setAt
 * Usage: grid.setAt(3, 10, value);
 * -----
 * Replaces the element at the specified row/col location in
 * this grid with a new value.  The previous value at that
 * location is overwritten with the new value.  Raises an error
 * if row is outside the range [0, numRows()-1] or if col is
 * outside the range [0, numCols()-1].
 */

void setAt(int row, int col, ElemType value);

/*
 * Method: inBounds
 * Usage: if (grid.inBounds(row, col);
 * -----
 * Returns true if the specified row and column position are inside
 * the bounds of the grid.
 */

bool inBounds(int row, int col);

/*
 * Method: operator[]
 * Usage: grid[0][0] = grid[1][1];
 * -----
 * Overloads [] to select elements from this grid.  This extension
 * allows the client to use traditional array notation to get/set
 * individual elements.  Returns a reference to the element to
 * allow in-place modification of values.  Raises an error if row
 * is outside the range [0, numRows()-1] or if col is outside
 * the range [0, numCols()-1].
 */

GridRow operator[](int row);
```



```
/*
 * Method: mapAll
 * Usage: grid.mapAll(Print);
 *        grid.mapAll(PrintToFile, outputStream);
 * -----
 * Iterates through the grid and calls the function fn once for
 * each element. The elements are processed by moving across each
 * row and then continuing on to the next row until the entire grid
 * has been processed. This is called row-major order. The second
 * form of this method allows the client to pass data of any type
 * to the callback function.
 */

void mapAll(void (*fn)(ElemType elem));

template <typename ClientDataType>
void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
            ClientDataType & data);

/*
 * Method: iterator
 * Usage: iter = grid.iterator();
 * -----
 * Creates an iterator that allows the client to iterate through
 * the elements in this grid in row-major order.
 *
 * The idiomatic code for accessing elements using an iterator is
 * to create the iterator from the collection and then enter a loop
 * that calls next() while hasNext() is true, like this:
 *
 *     Grid<int>::Iterator iter = grid.iterator();
 *     while (iter.hasNext()) {
 *         int elem = iter.next();
 *         . . .
 *     }
 *
 * This pattern can be abbreviated as follows:
 *
 *     foreach (int elem in grid) {
 *         . . .
 *     }
 */

Iterator iterator();

private:
#include "private/grid.h"
};
#include "private/grid.cpp"
#endif
```

```
/*
 * File: lexicon.h
 * -----
 * Defines the Lexicon class, which implements a word list.
 */

#ifndef _lexicon_h
#define _lexicon_h

#include "genlib.h"
#include "foreach.h"
#include "set.h"
#include "stack.h"

/*
 * Class: Lexicon
 * -----
 * This class is used to represent a lexicon, or word list. The
 * main difference between the lexicon abstraction and a map or
 * dictionary container is that the lexicon does not provide any
 * mechanism for storing definitions of words. The lexicon
 * contains only words, with no associated information. The
 * lexicon supports very efficient lookup operations for words
 * and prefixes. You can add words to a lexicon one at a time
 * or initialize a lexicon with words read from a file.
 *
 * The following example illustrates the use of a Lexicon object:
 *
 *     Lexicon lex("lexicon.dat");
 *     lex.add("doughnut");
 *     if (lex.containsPrefix("fru")) . . .
 *     if (lex.containsWord("ball")) . . .
 */

#include <cctype>

class Lexicon {
public:
    /* Forward references */
    class Iterator;

    /*
     * Constructor: Lexicon
     * Usage: Lexicon lex;
     * -----
     * This version of the constructor initializes a new empty lexicon.
     */
    Lexicon();
```

```
/*
 * Constructor: Lexicon
 * Usage: Lexicon lex("lexicon.dat");
 * -----
 * The version of the constructor initializes a lexicon by reading
 * the words read from the specified file. The file is expected
 * to be either a special binary format that represents a saved
 * lexicon or a plain text file of words, one word per line. The
 * file must be in the same folder as the program to be found. If
 * the file doesn't exist or is malformed, the constructor calls
 * Error to report the failure.
 */

Lexicon(string filename);

/*
 * Destructor: ~Lexicon
 * Usage: (usually implicit)
 * -----
 * Frees any storage associated with the lexicon.
 */

~Lexicon();

/*
 * Method: size
 * Usage: count = lex.size();
 * -----
 * Returns the number of words contained in this lexicon.
 */

int size();

/*
 * Method: isEmpty
 * Usage: if (lex.isEmpty()) . . .
 * -----
 * Returns true if this lexicon contains no words, false otherwise.
 */

bool isEmpty();

/*
 * Method: clear
 * Usage: lex.clear();
 * -----
 * Removes all words from this lexicon.
 */

void clear();
```

```
/*
 * Method: add
 * Usage: lex.add("computer");
 * -----
 * Adds the specified word to this lexicon.
 */
void add(string word);

/*
 * Method: addWordsFromFile
 * Usage: lex.addWordsFromFile("words.txt");
 * -----
 * Reads the specified file and adds all of its words to this
 * lexicon. The file is expected to be either a special
 * binary format that represents a saved lexicon or a plain
 * text file of words, one word per line. The file must be in
 * the same folder as the program to be found. If the file doesn't
 * exist or is malformed, addWordsFromFile calls Error to report
 * the failure.
 */
void addWordsFromFile(string filename);

/*
 * Method: containsWord
 * Usage: if (lex.containsWord("happy")) . . .
 * -----
 * Returns true if the word is contained in this lexicon, false
 * otherwise. Words are considered case-insensitively, "zoo"
 * is the same as "ZOO" or "zoo".
 */
bool containsWord(string word);

/*
 * Method: containsPrefix
 * Usage: lex.containsPrefix("mo");
 * -----
 * Returns true if any words in this lexicon begin with prefix,
 * false otherwise. A word is defined to be a prefix of itself
 * and the empty string is a prefix of everything. Prefixes are
 * considered case-insensitively, "mo" is a prefix of "MONKEY"
 * or "Monday".
 */
bool containsPrefix(string prefix);
```

```
/*
 * Method: mapAll
 * Usage: lexicon.mapAll(Print);
 *         lexicon.mapAll(PrintToFile, outputStream);
 * -----
 * Iterates through the lexicon and calls the function fn once
 * for each word. The words are processed in alphabetical order.
 * The second form of this method allows the client to pass data
 * of any type to the callback function.
 */

void mapAll(void (*fn)(string word));

template <typename ClientDataType>
void mapAll(void (*fn)(string word, ClientDataType & data),
            ClientDataType & data);

/*
 * Method: iterator
 * Usage: iter = lexicon.iterator();
 * -----
 * Creates an iterator that allows the client to iterate through
 * the elements in this lexicon in alphabetical order.
 *
 * The idiomatic code for accessing elements using an iterator is
 * to create the iterator from the collection and then enter a loop
 * that calls next() while hasNext() is true, like this:
 *
 *     Lexicon::Iterator iter = lexicon.iterator();
 *     while (iter.hasNext()) {
 *         string word = iter.next();
 *         . . .
 *     }
 *
 * This pattern can be abbreviated as follows:
 *
 *     foreach (string word in lexicon) {
 *         . . .
 *     }
 */

Iterator iterator();

private:
#include "private/lexicon.h"
};
#include "private/lexicon.cpp"
#endif
```

```
/*
 * File: map.h
 * -----
 * This interface file contains the Map class template, a
 * collection for efficiently storing key-value pairs.
 */

#ifndef _map_h
#define _map_h

#include "genlib.h"
#include "vector.h"
#include "foreach.h"

/*
 * Class: Map
 * -----
 * This interface defines a class template that stores a collection
 * of key-value pairs. For maximum generality, the Map is supplied
 * as a class template. The keys are always of type string, but
 * the value type is set by the client. The client specializes the
 * map to hold values of a specific type by specifying a template
 * parameter, as in Map<int> or Map<studentT>.
 */

template <typename ValueType>
class Map {

public:

    /* Forward references */
        class Iterator;

    /*
     * Constructor: Map
     * Usage: Map<int> map;
     *        Map<int> map(500);
     * -----
     * The constructor initializes a new empty map. The optional
     * argument is a hint about the expected number of entries that
     * this map will hold, which allows the map to configure itself
     * for efficiency at that size. The explicit keyword is used to
     * prevent accidental construction of a Map from an integer.
     * Raises an error if sizeHint is negative.
     */
        explicit Map(int sizeHint = 101);

    /*
     * Destructor: ~Map
     * Usage: (usually implicit)
     * -----
     * Frees storage associated with this map.
     */
        ~Map();
}
```

```
/*
 * Method: size
 * Usage: nEntries = map.size();
 * -----
 * Returns the number of entries in this map.
 */
int size();

/*
 * Method: isEmpty
 * Usage: if (map.isEmpty())...
 * -----
 * Returns true if this map contains no entries, false otherwise.
 */
bool isEmpty();

/*
 * Method: clear
 * Usage: map.clear();
 * -----
 * Removes all entries from this map.
 */
void clear();

/*
 * Method: put
 * Usage: map.put(key, value);
 * -----
 * Creates an association between key with value in this map. Any
 * previous value associated with key is replaced by this new entry.
 */
void put(string key, ValueType value);

/*
 * Method: get
 * Usage: value = map.get(key);
 * -----
 * Returns the value associated with key in this map, if any. If
 * key is not found, raises an error. The containsKey method can
 * be used to verify the presence of a key in the map before
 * attempting to get its value.
 */
ValueType get(string key);
```

```
/*
 * Method: containsKey
 * Usage: if (map.containsKey(key))...
 * -----
 * Returns true if there is an entry for key in this map,
 * false otherwise.
 */
    bool containsKey(string key);

/*
 * Method: remove
 * Usage: map.remove(key);
 * -----
 * Removes any entry for key from this map.  If there is no entry
 * for the key, the map is unchanged.
 */
    void remove(string key);

/*
 * Method: operator[]
 * Usage: map[key] = newValue;
 * -----
 * This method overloads [] to support array-based syntax for maps.
 * The argument inside the brackets is the key.  If the key is
 * already present in the map, this function returns a reference
 * to its associated value.  If key does not exist in the map,
 * a new entry for the key is added.  The value for the newly
 * entered key is set to the default for the value type.  Because
 * this method returns the value by reference, clients can assign
 * a value to the selection expression to modify the value.
 */
    ValueType & operator[](string key);

/*
 * Method: mapAll
 * Usage: map.mapAll(Print);
 *        map.mapAll(PrintToFile, outputStream);
 * -----
 * Iterates through the map and calls the function fn once, passing
 * in both the key and the value.  The order of processing is not
 * specified and will typically appear random to the client.  The
 * second form of this method allows the client to pass data of any
 * type to the callback function.
 */
    void mapAll(void (*fn)(key word, ValueType val));

    template <typename ClientDataType>
    void mapAll(void (*fn)(string, ValueType, ClientDataType &),
                ClientDataType & data);
```



```
/*
 * Method: iterator
 * Usage: iter = map.iterator();
 * -----
 * Creates an iterator that allows the client to iterate through
 * the keys in this map in an unspecified order.
 *
 * The idiomatic code for accessing elements using an iterator is
 * to create the iterator from the collection and then enter a loop
 * that calls next() while hasNext() is true, like this:
 *
 *     Map<int>::Iterator iter = map.iterator();
 *     while (iter.hasNext()) {
 *         string key = iter.next();
 *         . . .
 *     }
 *
 * This pattern can be abbreviated as follows:
 *
 *     foreach (string key in map) {
 *         . . .
 *     }
 */

    Iterator iterator();

private:
#include "private/map.h"
};
#include "private/map.cpp"
#endif
```

```
/*
 * File: queue.h
 * -----
 * This interface file contains the Queue class template,
 * which provides a linear FIFO collection.
 */

#ifndef _queue_h
#define _queue_h

#include "genlib.h"

/*
 * Class: Queue
 * -----
 * This interface defines a class that models a queue, or waiting
 * line. It is a linear collection managed in first-in/first-out
 * order. Values are added to the end and removed from the front.
 * The queue operations are enqueue (add to the end) and dequeue
 * (remove from the front).
 *
 * For maximum generality, the Queue is supplied as a class
 * template. The client specializes the queue to hold values of
 * a specific type, such as Queue<customerT> or Queue<string>.
 */

template <typename ElemType>
class Queue {
public:

/*
 * Constructor: Queue
 * Usage: Queue<int> queue;
 * -----
 * Initializes a new empty queue.
 */
    Queue();

/*
 * Destructor: ~Queue
 * Usage: (usually implicit)
 * -----
 * Frees the storage associated with this queue.
 */
    ~Queue();

/*
 * Method: size
 * Usage: nElems = queue.size();
 * -----
 * Returns the number of elements in this queue.
 */
    int size();
};
```

```
/*
 * Method: isEmpty
 * Usage: if (queue.isEmpty())...
 * -----
 * Returns true if this queue contains no elements, false otherwise.
 */
    bool isEmpty();

/*
 * Method: clear
 * Usage: queue.clear();
 * -----
 * Removes all elements from this queue.
 */
    void clear();

/*
 * Method: enqueue
 * Usage: queue.enqueue(element);
 * -----
 * Adds element to the end of this queue.
 */
    void enqueue(ElemType elem);

/*
 * Method: dequeue
 * Usage: first = queue.dequeue();
 * -----
 * Removes the front element from this queue and returns it.
 * This function raises an error if called on an empty queue.
 */
    ElemType dequeue();

/*
 * Method: peek
 * Usage: first = queue.peek();
 * -----
 * Returns the value of front element in this queue, without
 * removing it. This function raises an error if called on
 * an empty queue.
 */
    ElemType peek();

private:
#include "private/queue.h"

};

#include "private/queue.cpp"

#endif
```

```
/*
 * File: random.h
 * -----
 * This interface provides several functions for generating
 * pseudo-random numbers.
 */

#ifndef _random_h
#define _random_h

/*
 * Functions: Randomize
 * Usage: Randomize();
 * -----
 * This function sets the random seed so that the random sequence
 * is unpredictable. If this function is not called, the other
 * functions will return the same values on each run. During the
 * debugging phase, it is best not to call this function, so that
 * program behavior is repeatable.
 */

void Randomize();

/*
 * Function: RandomInteger
 * Usage: n = RandomInteger(low, high);
 * -----
 * This function returns a random integer in the range low to high,
 * inclusive.
 */

int RandomInteger(int low, int high);

/*
 * Function: RandomReal
 * Usage: d = RandomReal(low, high);
 * -----
 * This function returns a random real number in the half-open
 * interval [low .. high), meaning that the result is always
 * greater than or equal to low but strictly less than high.
 */

double RandomReal(double low, double high);

/*
 * Function: RandomChance
 * Usage: if (RandomChance(p)) . . .
 * -----
 * The RandomChance function returns true with the probability
 * indicated by p, which should be a floating-point number between
 * 0 (meaning never) and 1 (meaning always). For example, calling
 * RandomChance(.30) returns true 30 percent of the time.
 */

bool RandomChance(double p);
```



```
/*
 * Function: SetRandomSeed
 * Usage: SetRandomSeed(seed);
 * -----
 * This function sets the internal random number seed to the
 * specified value. Clients can use this function to set a
 * specific starting point for the pseudorandom sequence.
 */

void SetRandomSeed(int seed);

#endif
```

```
/*
 * File: scanner.h
 * -----
 * This file is the interface for a class that facilitates dividing
 * a string into logical units called "tokens", which are either
 *
 * 1. Strings of consecutive letters and digits representing words
 * 2. One-character strings representing punctuation or separators
 *
 * To use this class, you must first create an instance of a
 * Scanner object by declaring
 *
 *     Scanner scanner;
 *
 * You initialize the scanner's input in one of two ways. First,
 * you can read input from a string by calling
 *
 *     scanner.setInput(str);
 *
 * where str is the string from which tokens should be read.
 * Alternatively, you can retrieve tokens from an input file
 * by calling
 *
 *     scanner.setInput(infile);
 *
 * where infile is an open istream object. In either case, you
 * can then retrieve the next token by making the following call:
 *
 *     token = scanner.nextTok();
 *
 * To determine whether any tokens remain to be read, you can call
 * the predicate method scanner.hasMoreTokens(). The nextTok
 * method returns the empty string after the last token is read.
 *
 * The following code fragment serves as an idiom for processing
 * each token in the string inputString:
 *
 *     Scanner scanner;
 *     scanner.setInput(inputString);
 *     while (scanner.hasMoreTokens()) {
 *         string token = scanner.nextTok();
 *         . . . process the token . . .
 *     }
 *
 * Note that it is possible for clients to have more than one scanner
 * active at the same time. For instance
 *
 *     Scanner scannerX, scannerY;
 *
 * creates two independent scanner objects. The client specifies a
 * particular object name before the "." in a method call to
 * identify which particular instance of the scanner is to be used.
 *
 * The Scanner class also exports the methods saveToken,
 * setSpaceOption, setStringOption, setNumberOption, and
 * setBracketOption, which are documented later in the interface.
 */
```

```
#ifndef _scanner_h
#define _scanner_h

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

/*
 * Class: Scanner
 * -----
 * This class is used to represent a single instance of a scanner.
 */

class Scanner {
public:

/*
 * Constructor: Scanner
 * Usage: Scanner scanner;
 *        Scanner *sp = new Scanner;
 * -----
 * Initializes a new scanner object. The scanner starts empty,
 * with no input to scan.
 */
    Scanner();

/*
 * Destructor: ~Scanner
 * Usage: (usually implicit)
 * -----
 * Frees the storage associated with this scanner.
 */
    ~Scanner();

/*
 * Method: setInput
 * Usage: scanner.setInput(str);
 *        scanner.setInput(infile);
 * -----
 * Configures the scanner to start extracting tokens either from
 * the string or the input stream, as specified. Any previous
 * input source is discarded.
 */
    void setInput(string str);
    void setInput(istream & infile);

/*
 * Method: nextToken
 * Usage: token = scanner.nextToken();
 * -----
 * Returns the next token from this scanner. If no more tokens
 * are available, nextToken returns the empty string.
 */
    string nextToken();
};
```

```
/*
 * Method: hasMoreTokens
 * Usage: if (scanner.hasMoreTokens()) . . .
 * -----
 * Returns true as long as there are additional tokens for this
 * scanner to read.
 */

bool hasMoreTokens();

/*
 * Method: saveToken
 * Usage: scanner.saveToken(token);
 * -----
 * Saves the specified token into the scanner's private data
 * structures so that the next time nextToken is called, the
 * scanner will return the saved token without reading any
 * additional characters from the token stream.
 */

void saveToken(string token);

/*
 * Methods: setSpaceOption, getSpaceOption
 * Usage: scanner.setSpaceOption(option);
 *         option = scanner.getSpaceOption();
 * -----
 * The setSpaceOption method controls whether this scanner
 * ignores whitespace characters or treats them as valid tokens.
 * By default, the nextToken function treats whitespace characters,
 * such as spaces and tabs, just like any other punctuation mark.
 * If, however, you call
 *
 *     scanner.setSpaceOption(Scanner::IgnoreSpaces);
 *
 * the scanner will skip over any white space before reading a
 * token. You can restore the original behavior by calling
 *
 *     scanner.setSpaceOption(Scanner::PreserveSpaces);
 *
 * The getSpaceOption function returns the current setting
 * of this option.
 */

enum spaceOptionT { PreserveSpaces, IgnoreSpaces };

void setSpaceOption(spaceOptionT option);
spaceOptionT getSpaceOption();
```



```
/*
 * Methods: setNumberOption, getNumberOption
 * Usage: scanner.setNumberOption(option);
 *         option = scanner.getNumberOption();
 * -----
 * The setNumberOption method controls whether this scanner treats
 * numeric values specially. The default behavior for a scanner is
 * to treat digits as equivalent to letters. If you call
 *
 *     scanner.setNumberOption(Scanner::ScanNumbersAsIntegers);
 *
 * a token beginning with a digit will end at the first nondigit.
 * (Note that digits can still be scanned as part of a token as in
 * the token "x1".) If you call
 *
 *     scanner.setNumberOption(Scanner::ScanNumbersAsReals);
 *
 * the scanner will return the longest token string that represents
 * a real number, if the next character to be scanned is a digit.
 * The format for a real number is a sequence of digit characters
 * that may include at most one decimal point, optionally followed
 * by the letter 'E' in either upper- or lowercase, an optional sign,
 * and an exponent. You can restore the default behavior by calling
 *
 *     scanner.setNumberOption(Scanner::ScanNumbersAsLetters);
 *
 * Even if the number options are enabled, nextToken always returns
 * its result as a string, which means that you need to call
 * StringToInteger or StringToReal to convert the token to a number.
 */

enum numberOptionT {
    ScanNumbersAsLetters,
    ScanNumbersAsIntegers,
    ScanNumbersAsReals
};

void setNumberOption(numberOptionT option);
numberOptionT getNumberOption();
```

```
/*
 * Methods: setStringOption, getStringOption
 * Usage: scanner.setStringOption(option);
 *         option = scanner.getStringOption();
 * -----
 * The setStringOption method controls how this scanner treats
 * double quotation marks in the input. The default behavior
 * for a scanner is to treat quotes just like any other punctuation
 * character. If, however, you call
 *
 *     scanner.setStringOption(Scanner::ScanQuotesAsStrings);
 *
 * a token beginning with a quotation mark will be scanned up to
 * the closing quotation mark. The quotation marks are returned
 * as part of the scanned token so that clients can differentiate
 * strings from other token types. The original behavior can be
 * restored by calling
 *
 *     scanner.setStringOption(Scanner::ScanQuotesAsPunctuation);
 *
 * When scanning a string, the scanner recognizes the standard
 * escape sequences from ANSI C/C++, such as \n and \t.
 */

enum stringOptionT {
    ScanQuotesAsPunctuation,
    ScanQuotesAsStrings
};

void setStringOption(stringOptionT option);
stringOptionT getStringOption();
```

```
/*
 * Methods: setBracketOption, getBracketOption
 * Usage: scanner.setBracketOption(option);
 *         option = scanner.getBracketOption();
 * -----
 * The setBracketOption method controls how this scanner treats
 * angle brackets in the input. The default behavior for a
 * scanner is to treat brackets just like any other punctuation
 * character. If, however, you call
 *
 *     scanner.setBracketOption(Scanner::ScanBracketsAsTag);
 *
 * a token beginning with a open < bracket will be scanned up to
 * the closing > bracket. The brackets are returned as part of the
 * scanned token so that clients can differentiate tags from other
 * token types. The original behavior can be restored by calling
 *
 *     scanner.setBracketOption(Scanner::ScanBracketsAsPunctuation);
 */

enum bracketOptionT {
    ScanBracketsAsPunctuation,
    ScanBracketsAsTag
};

void setBracketOption(bracketOptionT option);
bracketOptionT getBracketOption();

private:

#include "private/scanner.h"

};

#endif
```

```
/*
 * File: set.h
 * -----
 * This interface file contains the Set class template, a
 * collection for efficiently storing a set of distinct elements.
 */

#ifndef _set_h
#define _set_h

#include "cmpfn.h"
#include "bst.h"
#include "vector.h"
#include "foreach.h"

/*
 * Class: Set
 * -----
 * This interface defines a class template that stores a collection
 * of distinct elements, using a sorted relation on the elements to
 * provide efficient management of the collection. For maximum
 * generality, the Set is supplied as a class template. The element
 * type is determined by the client. The client configures the set
 * to hold values of a specific type, such as Set<int> or
 * Set<studentT>. The one requirement on the element type is that
 * the client must supply a comparison function that compares two
 * elements (or be willing to use the default comparison function
 * that uses the built-on operators < and ==).
 */

template <typename ElemType>
class Set {

public:

    /* Forward references */
    class Iterator;

    /*
     * Constructor: Set
     * Usage: Set<int> set;
     *        Set<student> students(CompareStudentsById);
     * -----
     * The constructor initializes an empty set. The optional
     * argument is a function pointer that is applied to two
     * elements to determine their relative ordering. The
     * comparison function should return 0 if the two elements
     * are equal, a negative result if first is "less than" second,
     * and a positive result if first is "greater than" second. If
     * no argument is supplied, the OperatorCmp template is used as
     * a default, which applies the built-in < and == to the
     * elements to determine ordering.
     */

    Set(int (*cmpFn)(ElemType, ElemType) = OperatorCmp);
};
```

```
/*
 * Destructor: ~Set
 * Usage: (usually implicit)
 * -----
 * Frees the storage associated with set.
 */
~Set();

/*
 * Method: size
 * Usage: count = set.size();
 * -----
 * Returns the number of elements in this set.
 */
int size();

/*
 * Method: isEmpty
 * Usage: if (set.isEmpty())...
 * -----
 * Returns true if this set contains no elements, false otherwise.
 */
bool isEmpty();

/*
 * Method: clear
 * Usage: set.clear();
 * -----
 * Removes all elements from this set.
 */
void clear();

/*
 * Method: add
 * Usage: set.add(elem);
 * -----
 * Adds an element to this set.
 */
void add(ElemType elem);

/*
 * Method: remove
 * Usage: set.remove(elem);
 * -----
 * Removes an element from this set. If the element was not
 * contained in the set, the set is unchanged.
 */
void remove(ElemType elem);
```

```
/*
 * Method: contains
 * Usage: if (set.contains(elem)) . . .
 * -----
 * Returns true if the element in this set, false otherwise.
 */
bool contains(ElemType elem);

/*
 * Method: find
 * Usage: eptr = set.find(elem);
 * -----
 * Searches for the element in this set and returns a pointer to
 * that value, if it exists. If the element is not contained in
 * this set, this method returns NULL.
 */
ElemType *find(ElemType elem);

/*
 * Method: equals
 * Usage: if (set.equals(set2)) . . .
 * -----
 * Returns true if this set and set2 contain exactly the same
 * elements, false otherwise.
 */
bool equals(Set & set2);

/*
 * Method: isSubsetOf
 * Usage: if (set.isSubsetOf(set2)) . . .
 * -----
 * Returns true if all elements in this set are contained in set2.
 * Note that the subset relation holds if the sets are equal.
 */
bool isSubsetOf(Set & set2);

/*
 * Methods: unionWith, intersectWith, subtract
 * Usage: set.unionWith(set2);
 *         set.intersectWith(set2);
 *         set.subtract(set2);
 * -----
 * These methods modify the receiver set as follows:
 *
 * set.unionWith(set2);      Adds all elements from set2.
 * set.intersectWith(set2);  Removes all elements not in set2.
 * set.subtract(set2);       Removes all elements in set2.
 */
void unionWith(Set & set2);
void intersectWith(Set & set2);
void subtract(Set & set2);
```

```
/*
 * Method: mapAll
 * Usage: set.mapAll(Print);
 *        set.mapAll(PrintToFile, outputStream);
 * -----
 * Iterates through this set and calls the function fn once for
 * each element. The order is determined by the set's comparison
 * function. The second form allows the client to pass data of
 * any type to the callback function.
 */

void mapAll(void (*fn)(ElemType elem));

template <typename ClientDataType>
void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
            ClientDataType & data);

/*
 * Method: iterator
 * Usage: iter = set.iterator();
 * -----
 * Creates an iterator that allows the client to iterate through
 * the elements in this set. The elements are returned in the
 * order determined by the comparison function.
 *
 * The idiomatic code for accessing elements using an iterator is
 * to create the iterator from the collection and then enter a loop
 * that calls next() while hasNext() is true, like this:
 *
 *     Set<int>::Iterator iter = set.iterator();
 *     while (iter.hasNext()) {
 *         int value = iter.next();
 *         . . .
 *     }
 *
 * This pattern can be abbreviated as follows:
 *
 *     foreach (int value in set) {
 *         . . .
 *     }
 */

Iterator iterator();

private:
#include "private/set.h"
};
#include "private/set.cpp"
#endif
```

```
/*
 * File: simpio.h
 * -----
 * This interface provides access to a simple package of
 * functions that simplify the reading of console input.
 */

#ifndef _simpio_h
#define _simpio_h

#include "genlib.h"

/*
 * Function: GetInteger
 * Usage: n = GetInteger();
 * -----
 * GetInteger reads a line of text from standard input and scans
 * it as an integer. The integer value is returned. If an
 * integer cannot be scanned or if more characters follow the
 * number, the user is given a chance to retry.
 */

int GetInteger();

/*
 * Function: GetLong
 * Usage: n = GetLong();
 * -----
 * GetLong reads a line of text from standard input and scans
 * it into a long integer. The long is returned. If the
 * number cannot be scanned or if extra characters follow it,
 * the user is given a chance to retry.
 */

long GetLong();

/*
 * Function: GetReal
 * Usage: x = GetReal();
 * -----
 * GetReal reads a line of text from standard input and scans
 * it as a double. If the number cannot be scanned or if extra
 * characters follow after the number ends, the user is given
 * a chance to reenter the value.
 */

double GetReal();

/*
 * Function: GetLine
 * Usage: s = GetLine();
 * -----
 * GetLine reads a line of text from standard input and returns
 * the line as a string. The newline character that terminates
 * the input is not stored as part of the string that is returned.
 */

string GetLine();

#endif
```



```
/*
 * File: sound.h
 * -----
 * This interface defines a function for playing a named
 * sound file.
 */

#ifndef _sound_h
#define _sound_h

#include "genlib.h"

/*
 * Function: PlayNamedSound
 * Usage: PlayNamedSound("beep.wav");
 * -----
 * This function looks for a sound file with the given name
 * in the "Sounds" subdirectory of the project.
 * If a matching sound file is located, it is loaded and played.
 * The function generates an error if the file cannot be
 * found or the sound facility is unimplemented for the platform.
 */

void PlayNamedSound(string name);

/*
 * Function: SetSoundOn
 * Usage: SetSoundOn(false);
 * -----
 * This function enables/disables the playing of sound.
 * If you call the function passing false, it will disable sound
 * and any subsequent calls to PlayNamedSound will be ignored.
 * If you call the function passing true, it will enable sound
 * and any subsequent call to PlayNamedSound will operate normally.
 * The default is true.
 */

void SetSoundOn(bool on);

#endif
```

```
/*
 * File: stack.h
 * -----
 * This interface file contains the Stack class template,
 * which provides a linear LIFO collection.
 */

#ifndef _stack_h
#define _stack_h

#include "vector.h"

/*
 * Class: Stack
 * -----
 * This interface defines a class template that models a "stack":
 * that is, a linear collection of values stacked one on top of
 * the other. Values added and removed only from the top of the
 * stack. The fundamental stack operations are push (add to top)
 * and pop (remove from top). A stack is said to operate in
 * last-in/first-out (LIFO) order.
 *
 * For maximum generality, the Stack is supplied as a class
 * template. The client specializes the stack to hold values
 * of a specific type, such as Stack<string> or Stack<stateT>d.
 */

template <typename ElemType>
class Stack {
public:

/*
 * Constructor: Stack
 * Usage: Stack<int> stack;
 * -----
 * The constructor initializes a new empty stack.
 */
    Stack();

/*
 * Destructor: ~Stack
 * Usage: (usually implicit)
 * -----
 * Frees the storage associated with this stack.
 */
    ~Stack();

/*
 * Method: size
 * Usage: nElems = stack.size();
 * -----
 * Returns the number of elements in this stack.
 */
    int size();
};
```

```
/*
 * Method: isEmpty
 * Usage: if (stack.isEmpty())...
 * -----
 * Returns true if this stack contains no elements, false otherwise.
 */
    bool isEmpty();

/*
 * Method: clear
 * Usage: stack.clear();
 * -----
 * This method removes all elements from this stack.
 */
    void clear();

/*
 * Method: push
 * Usage: stack.push(elem);
 * -----
 * Pushes the specified element onto this stack, so that it becomes
 * the top element.
 */
    void push(ElemType elem);

/*
 * Method: pop
 * Usage: top = stack.pop();
 * -----
 * Removes the top element from this stack and returns it. This
 * method raises an error if called on an empty stack.
 */
    ElemType pop();

/*
 * Method: peek
 * Usage: top = stack.peek();
 * -----
 * Returns the value of top element from this stack, without
 * removing it. Raises an error if called on an empty stack.
 */
    ElemType peek();

private:
#include "private/stack.h"

};

#include "private/stack.cpp"

#endif
```

```
/*
 * File: strutils.h
 * -----
 * The strutils.h file defines some useful helper functions
 * not included by the C++ string library. These were taken
 * from Eric Roberts's original strlib from his text
 * _The Art and Science of C_.
 */

#ifndef _strutils_h
#define _strutils_h

#include "genlib.h"

/*
 * Function: IntegerToString
 * Usage: s = IntegerToString(n);
 * -----
 * This function converts an integer into the corresponding
 * string of digits. For example, IntegerToString(123)
 * returns "123" as a string.
 */
string IntegerToString(int num);

/*
 * Function: RealToString
 * Usage: s = RealToString(d);
 * -----
 * This function converts a floating-point number into the
 * corresponding string form. For example, calling
 * RealToString(23.45) returns "23.45".
 */
string RealToString(double num);

/*
 * Function: StringToInteger
 * Usage: n = StringToInteger(s);
 * -----
 * This function converts a string of digits into an integer.
 * If the string is not a legal integer or contains extraneous
 * characters, StringToInteger signals an error condition.
 */
int StringToInteger(string str);

/*
 * Function: StringToReal
 * Usage: d = StringToReal(s);
 * -----
 * This function converts a string representing a real number
 * into its corresponding value. If the string is not a
 * legal floating-point number or if it contains extraneous
 * characters, StringToReal signals an error condition.
 */
double StringToReal(string str);
```

```
/*
 * Function: ConvertToLowerCase
 * Usage: s = ConvertToLowerCase(s);
 * -----
 * This function returns a new string with all
 * alphabetic characters converted to lower case.
 */

string ConvertToLowerCase(string s);

/*
 * Function: ConvertToUpperCase
 * Usage: s = ConvertToUpperCase(s);
 * -----
 * This function returns a new string with all
 * alphabetic characters converted to upper case.
 */

string ConvertToUpperCase(string s);

#endif
```

```
/*
 * File: vector.h
 * -----
 * This interface file contains the Vector class template, an
 * efficient, safer, convenient replacement for the built-in array.
 */

#ifndef _vector_h
#define _vector_h

#include "genlib.h"
#include "strutils.h"
#include "foreach.h"

/*
 * Class: Vector
 * -----
 * This interface defines a class template that stores a homogeneous
 * indexed collection. The basic operations are similar to those
 * in the built-in array type, with the added features of dynamic
 * memory management, bounds-checking on indexes, and convenient
 * insert/remove operations. Like an array, but better! For
 * maximum generality, the Vector is supplied as a class template.
 * The client specializes the vector to hold values of a specific
 * type, such as Vector<int> or Vector<studentT>.
 */

template <typename ElemType>
class Vector {

public:

    /* Forward references */
    class Iterator;

    /*
     * Constructor: Vector
     * Usage: Vector<int> vec;
     *        Vector<student> dormlist(200);
     * -----
     * The constructor initializes a new empty vector. The optional
     * argument is a hint about the expected number of elements that
     * this vector will hold, which allows vector to configure itself
     * for that capacity during initialization. If not specified,
     * it is initialized with default capacity and grows as elements
     * are added. Note that capacity does not mean size: a newly
     * constructed vector always has size = 0. A large starting
     * capacity allows you to add that many elements without requiring
     * any internal reallocation. The explicit keyword is required to
     * avoid accidental construction of a vector from an int.
     */

    explicit Vector(int sizeHint = 0);
```

```
/*
 * Destructor: ~Vector
 * Usage: (usually implicit)
 * -----
 * Frees the storage associated with this vector.
 */
~Vector();

/*
 * Method: size
 * Usage: nElems = vec.size();
 * -----
 * Returns the number of elements in this vector.
 */
int size();

/*
 * Method: isEmpty
 * Usage: if (vec.isEmpty())...
 * -----
 * Returns true if this vector contains no elements, false otherwise.
 */
bool isEmpty();

/*
 * Method: clear
 * Usage: vec.clear();
 * -----
 * Removes all elements from this vector.
 */
void clear();

/*
 * Method: getAt
 * Usage: val = vec.getAt(3);
 * -----
 * Returns the element at the specified index in this vector.
 * Elements are indexed starting with 0. A call to vec.getAt(0)
 * returns the first element; vec.getAt(vec.size()-1) returns the
 * last. Raises an error if index is not in the range [0, size()-1].
 */
ElemType getAt(int index);

/*
 * Method: setAt
 * Usage: vec.setAt(3, value);
 * -----
 * Replaces the element at the specified index in this vector with
 * a new value. The previous value at that index is overwritten.
 * Raises an error if index is not in the range [0, size()-1].
 */
void setAt(int index, ElemType value);
```

```
/*
 * Method: operator[]
 * Usage: vec[0] = vec[1];
 * -----
 * Overloads [] to select elements from this vector. This extension
 * allows the client to use traditional array notation to get/set
 * individual elements. Returns a reference to the element to
 * allow in-place modification of values. Raises an error if the
 * index is outside the range [0, size()-1].
 */

ElemType & operator[](int index);

/*
 * Method: add
 * Usage: vec.add(value);
 * -----
 * Adds an element to the end of this vector.
 */

void add(ElemType elem);

/*
 * Method: insertAt
 * Usage: vec.insertAt(0, value);
 * -----
 * Inserts the element into this vector before the specified index,
 * shifting all subsequent elements one index higher. A call to
 * vec.insertAt(0, val) inserts a new element at the beginning;
 * vec.insertAt(vec.size(), val) adds a new element to the end.
 * Raises an error if index is outside the range [0, size()].
 */

void insertAt(int index, ElemType elem);

/*
 * Method: removeAt
 * Usage: vec.removeAt(3);
 * -----
 * Removes the element at the specified index from this vector,
 * shifting all subsequent elements one index lower. A call to
 * vec.removeAt(0) removes the first element, while a call to
 * vec.removeAt(vec.size()-1), removes the last. Raises an error
 * if index is outside the range [0, size()-1].
 */

void removeAt(int index);
```



```

/*
 * Method: mapAll
 * Usage: vec.mapAll(Print);
 *        vec.mapAll(PrintToFile, outputStream);
 * -----
 * Iterates through the vector and calls the function fn once for
 * each element. The elements are processed in index order. The
 * second form of this method allows the client to pass data of
 * any type to the callback function.
 */

void mapAll(void (*fn)(ElemType elem));

template <typename ClientDataType>
void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
            ClientDataType & data);

/*
 * Method: iterator
 * Usage: iter = vec.iterator();
 * -----
 * Creates an iterator that allows the client to iterate through
 * the elements in this vector in index order.
 *
 * The idiomatic code for accessing elements using an iterator is
 * to create the iterator from the collection and then enter a loop
 * that calls next() while hasNext() is true, like this:
 *
 *     Vector<int>::Iterator iter = vec.iterator();
 *     while (iter.hasNext()) {
 *         int elem = iter.next();
 *         . . .
 *     }
 *
 * This pattern can be abbreviated as follows:
 *
 *     foreach (int elem in vec) {
 *         . . .
 *     }
 */

Iterator iterator();

private:
#include "private/vector.h"
};
#include "private/vector.cpp"
#endif

```


Index

! operator	23	at method	101
% operator	18	AT&T Bell Laboratories	3
& operator	53	atan function	112
&& operator	23	atan2 function	112
* operator	53	atomic type	11
*= operator	21	Austen, Jane	381
++ operator	21	automatic allocation	71
+= operator	21	Average program	285
-- operator	21	axis	214, 463
-= operator	21		
-> operator	71	Bachmann, Paul	283
. operator	69	backtracking algorithm	236
/= operator	21	Backus, John	2, 514
:: specification	316	Backus-Naur form	514
<< operator	8, 107, 111	balance factor	461
>> operator	111	balanced tree	459
? : operator	23	base type	52, 126
operator	23	basis	302
		big-O notation	283
abs function	112	binary logarithm	293
absolute coordinates	214	binary operator	17
absorption	528	binary search	190
abstract class	503	binary search tree	451
abstract data type	124	binary tree	451
abstraction	87	bit	49
accessor	315	bitwise operator	540
add method	127, 152	block	24
additive sequence	186	block size	377
addlist.cpp	30	BNF	514
address	49	boilerplate	89
addWordsFromFile	152	Boole, George	14
AdjacentPoint method	243	Boolean data	14
ADT	124	bounds checking	125
allocated size	59	Brando, Marlon	313
allocation	66	Braque, Georges	217
alphanumeric	112	break statement	26, 29
ambiguity	499	bst.h interface	598
anagram	211	bucket	421
analysis of algorithms	278	buffer	340
ancestor	448	buffer.h interface	343
APL	515	byte	49
argument	32		
array	46, 64	calculator	135
array element	56	call by reference	35
array parameter	59	callback function	436
array size	56	calling a function	32
ASCII	13	cardinality	530
assignment statement	9, 20	Carmichael, Stokely	1
associative array	151	Carroll, Lewis	2, 105
associative law	528	Cather, Willa	447
associativity	17	ceil function	112

- cell 358
- cellT** type 358
- cerr** stream 107
- characteristic vector 538
- charstack.h** interface 320
- checkout.cpp** program 143
- chess 274
- children 448
- cin** stream 107
- clear method 106, 111, 127, 135, 139, 147, 152
- clearing a bit 543
- client 86
- close** method 107, 111
- closing a file 107
- cmpfn.h** interface 436, 601
- CollectContributions** 175
- collection 126
- collision 421
- colorT** type 47
- combinations 199
- comment 5
- commutative law 528
- compiler 4
- complement 541
- complexity class 294
- compound statement 24
- computational complexity .. 282
- conditional execution 25
- conditional operator 23
- constant folding 524
- constant time 285
- constructor 126, 317
- container class 126
- containsKey** method 147
- containsPrefix** method ... 152
- containsWord** method 152
- control expression 26
- ConvertToLowerCase** 103
- ConvertToUpperCase** 103
- coordinate 214
- CopyFile** 109
- CopyRemovingComments** 110
- cos** function 112
- cout** stream 8, 107
- craps.cpp** file 93
- crossover point 310
- cstkpriv.h** file 325
- Cubism 217
- cycle 28
- cyclic cipher 118
- Dahl, Ole-Johan 3
- data members 314
- data structure 46, 124
- data type 11
- de Moivre, Abraham 311
- decomposition 35
- decrement operator 21
- deep copy 327
- default constructor 132
- definite integral 444
- delete** operator 72
- deleteCharacter** method 344
- DeMorgan's law 528
- dequeue** method 139
- dereferencing a pointer 53
- descendant 448
- destructor 318
- Dickens, Charles 67, 231
- digital root 198
- DirectionName** 49
- directionT** type 46
- DISALLOW_COPYING** 327
- discrete time 141
- Disney, Walt 169
- display 344
- DisplayTree** function 457
- distributive law 528
- divide-and-conquer 175, 289
- domain 11, 430
- dot operator 69
- double** type 13
- double rotation 464
- doubly linked list 367
- Doyle, Sir Arthur Conan 45
- DrawArc** function 215
- DrawFractalLine** 223
- DrawLine** function 215
- DrawLineTo** function 432
- DrawPolarLine** 222
- dummy cell 359
- Dürer, Albrecht 167
- dynamic allocation 71
- dynamic array 73
- dynamic dispatch 503
- editor 340
- editor.cpp** program 346
- EditorBuffer** class 343
- EditorBuffer** constructor 343, 366
- effective size 59
- element 56, 134
- element type 56
- Eliot, George 525
- embedded assignment 20
- empty set 526

- empty vector 126
- endl** format specification ... 8
- Engels, Friedrich 277
- EnglishWords.dat** file 151
- enqueue** method 139
- enumeration 46, 527
- enumeration constant 46
- enumeration type 46
- EOF** constant 109
- Eratosthenes 78
- Error** function 26
- escape sequence 14
- Euclid 197
- eval** method 502
- EvaluatePosition** function 257
- executable file 4
- exp** function 112
- exporting a method 315
- expression 16
- expression tree 500
- extending an interface 89
- extgraph.h** interface 602
-
- fabs** function 112
- Fact** function 176
- factorial 176
- fail** method 106, 108, 111
- Fib** function 184, 186
- fib.cpp** program 184
- Fibonacci sequence 182
- Fibonacci, Leonardo 181
- field 67
- FIFO 139
- file 105
- find method 101
- findcell** method 426
- FindGoodMove** function 247
- findKey** method 418
- FindNode** function 454
- finite set 526
- finite-state machine 337
- FlipCoin** function 91
- float** type 13
- floating-point 13
- floor** function 112
- fmod** function 112
- for** statement 30
- Fortran 2
- fractal 220
- Fredkin, Edward 490
- free** function 100
- friend** keyword 414
- fstream** interface 105
- function 7, 32
- function domain 430
- function pointer 431
- function prototype 7
- function range 430
-
- garbage collection 72
- Gauss, Karl Friedrich 40
- genlib.h** interface 612
- get** method 108, 111, 147
- getAt** method 127, 132
- getBracketOption** 157
- GetCurrentX** function 215
- GetCurrentY** function 215
- GetInteger** function 15
- GetLine** function 15
- getline** method 110
- GetLong** function 15
- getNumberOption** 157
- GetReal** function 15
- getSpaceOption** 157
- getStringOption** 157
- getters 315
- GetWindowHeight** 215
- GetWindowWidth** 215
- global variable 11
- Goldberg, Adele 3
- golden ratio 311
- grammar 514
- graph.h** interface 613
- graphics window 214
- graphics.h** interface 215, 617
- Gray code 229
- greatest common divisor 197
- Grid** class 131
- grid.h** interface 620
- gymjudge.cpp** program 61
-
- half-open interval 91
- Hamilton, Charles 1
- hash code 421
- hash function 421, 427
- hash** method 426
- hash table 421
- hashing 421
- hasMoreTokens** method 157
- head 139
- header file 5, 86
- heap 71
- height of a tree 449
- hexadecimal 12
- higher-level language 2
- histogram 79
- Hoare, C. A. R. 296
- holism 192

- Holmes, Sherlock 45
- hybrid strategy 310
- IATA 148
- idempotence 528
- identifier 10
- identity 528
- identity matrix 64
- if** statement 25
- ifstream** type 106
- immutable class 317
- implementation 86
- increment operator 21
- index 57
- inductive hypothesis 303
- infinite set 526
- inheritance 502
- InitGraphics** function 215
- initializer 10
- inorder traversal 457
- insert** method 101
- insertAt** method 127
- insertCharacter** method 344
- insertion sort 307
- InsertNode** function 455
- instance variables 314
- Instant Insanity 276
- int** type 12
- IntegerToString** 104
- interface 86
- interface entry 87
- interior node 449
- interpreter 492
- intersection 527
- iomanip** interface 5, 15
- iostream** interface 5, 105
- isalnum** function 112
- isalpha** function 112
- IsBadPosition** 247
- isdigit** function 112
- isEmpty** method 127, 135, 139, 147, 152
- IsEven** function 192
- islower** function 112
- IsOdd** function 192
- IsPalindrome** function 188
- isprint** function 112
- ispunct** function 112
- isspace** function 112
- isupper** function 112
- IsVowel** function 28
- iteration order 159
- iterative 176
- iterative statement 28
- Iterator** class 413
- iterator 156, 413
- Iverson, Ken 515
- Iversonian precedence 515
- Jabberwocky 105
- James, William 173
- key 146, 451
- keyword 11
- knight's tour 274
- Koch snowflake 220
- Koch, Helge von 220
- koch.cpp** program 223
- Kuhn, Thomas 3
- leaf node 449
- left** format specification 16
- left-associative 17
- Leibniz, Gottfried 42
- length** method 101
- LengthCompare** function 435
- letter-substitution cipher 118
- lexical analysis 493
- lexicographic order 190, 452
- Lexicon** class 151
- lexicon.h** interface 624
- library 4
- LIFO 134
- linear probing 444
- linear search 190
- linear structures 382
- linear time 285
- link 358
- linked list 358
- Linnaeus, Carolus 448
- ListPermutations** 213
- load factor 428
- local variable 11
- log** function 112
- logarithm 293
- logical and 23
- logical not 23
- logical operator 23
- logical or 23
- long** type 12
- longdouble** type 13
- loop 28
- loop-and-a-half problem 29
- Lucas, Edouard 202
- lvalue 52
- machine language 2
- magic square 167

- majority element 312
- Map** class 146, 412
- map 146
- map.h** interface 412, 628
- mapAll** method 438
- mapimpl.cpp** file 416, 423
- mapping function 437
- mappriv.h** file 415, 422
- mask 542
- mathematical induction 302
- matrix 63
- mazelib.h** interface 240
- mean 59
- member 67
- member function 100
- membership 526
- memory allocation 66
- Merge** function 291
- merge sort algorithm 290
- merging 289
- message 100
- method 100
- metric 193
- Milne, A. A. 491
- mnemonic 227
- model 140
- modular arithmetic 396
- Mondrian, Piet 217
- mondrian.cpp** program 219
- monthT** type 47
- Morse code 170
- Morse, Samuel F. B. 170
- moveCursorBackward** 343
- moveCursorForward** 343
- moveCursorToEnd** 344
- moveCursorToStart** 344
- MovePen** function 215
- MoveSingleDisk** function 207
- MoveTower** function 207
- msort.cpp** program 291
- multidimensional array 62
- multiple assignment 20
- mutator 316
- mutual recursion 190
- Myhrhaug, Björn 3

- N-Queens problem 275
- natural number 191, 526
- Naur, Peter 514
- nested class 413
- new** operator 72
- nextToken** method 157
- nim 245, 276
- nim.cpp** program 248

- node 448
- nonterminal symbol 514
- nonterminating recursion 194
- NULL** constant 56
- null pointer 56
- numCols** method 132
- numRows** method 132
- Nygaard, Kristen 3

- object file 4
- object-oriented paradigm 3
- octal 12
- ofstream** type 106
- Olsen, Tillie 339
- open addressing 443
- open method 111
- opening a file 106
- operator 16, 107
- OperatorCmp** function 435
- operator** keyword 412
- optimization 524
- origin 214
- override 502

- package 87
- palindrome 118, 171, 187
- paradigm shift 3
- parameter 33
- parameterized classes 126
- parent 448
- parse tree 499
- parsing 493
- Partition function 300
- Parville, Henri De 202
- Pascal's Triangle 199
- Pascal, Blaise 199
- peek** method 135, 139
- perfect number 41
- permutation 211
- Picasso, Pablo 217
- pixel 273
- Plot** function 432
- plotting a function 430
- ply 254
- pocket calculator 135
- Point** class 314
- PointCompare** function 435
- pointer 46, 51, 64
- pointer assignment 55
- Poisson distribution 142
- polar coordinates 222
- polymorphic class 382
- polynomial algorithm 294
- pop** method 135

- portability 12
- postfix 22
- postorder traversal 458
- pow** function 112
- power set 550
- powertab.cpp** program 6
- precedence 16
- predicate function 33
- prefix 22
- preorder traversal 458
- prime factorization 41
- prime number 78
- priority queue 417
- procedural paradigm 3
- procedure 33
- prompt 15
- proper subset 528
- pseudo-random number 91
- pure virtual method 503
- push** method 135
- put** method 109, 111, 147

- qsort.cpp** program 299
- quadeq.cpp** program 36
- quadratic equation 35
- quadratic time 286
- qualifier 316
- Queue** class 136
- queue 139
- queue.h** interface 392, 632
- queueimpl.cpp** file 396, 402
- queuepriv.h** file 399, 400
- Quicksort algorithm 296
- Quicksort** function 300

- RaiseIntToPower** 197
- RAM 49
- random.cpp** file 97
- random.h** interface 90, 634
- Randomize** function 96
- RAND_MAX** constant 94
- range 430
- Rational** class 334
- rational number 334
- read-eval-print loop 492
- real number 526
- RealToString** function 104
- receiver 100
- record 46
- record selection 69
- recurrence relation 183
- recursion 174
- recursive decomposition 175
- recursive descent 516
- recursive leap of faith 181
- recursive paradigm 175
- recursive type 358
- RecursivePermute** 213
- red-black tree 488
- reductionism 192
- rehashing 429
- relation 527
- relational operator 22
- relative coordinates 214
- remainder operator (%) 18
- remove** method 147
- removeAt** method 127
- replace** method 101
- reserved word 11
- resize** method 132
- return by reference 413
- return** statement 33
- returning a value 32, 413
- reverse Polish notation 135
- revfile.cpp** program 130
- right** format specification 16
- right-associative 17
- right-hand rule 236
- RightFrom** function 48
- ring buffer 395
- Ritchie, Dennis 3
- Robson, David 3
- Roman numeral 412, 442
- root 448
- RPN 135
- rpncalc.cpp** program 137

- saveToken** method 157
- scalar type 48
- Scanner** class 154, 328
- scanner.h** interface 636
- Scrabble 211
- searching 189
- seed 95
- selection 57, 69
- selection sort algorithm 279
- semicolon 25
- sentinel 29
- set 526
- set difference 527
- set equality 528
- set.h** interface 642
- setAt** method 127, 132
- setBracketOption** 157
- setInput** method 157
- setNumberOption** 157
- setprecision** 16
- setSpaceOption** 157

- setStringOption** 157
- setter 316
- setting a bit 543
- setw** format specification... 16
- shadowing 318
- Shakespeare, William 85, 411
- shallow copy 327
- Shelley, Mary 201
- short** type 12
- short-circuit evaluation 23
- shorthand assignment 20
- sibling 449
- sieve of Eratosthenes 79
- simpio.h** interface 108, 646
- simple case 175
- simple statement 24
- SIMULA 3
- simulation 140
- sin** function 112
- sine function 430
- size** method 127, 135, 139, 147, 152
- sizeof** operator 51
- Smalltalk 3
- SolveMaze** function 242
- Sort** function 279, 291, 299, 433
- sound.h** interface 647
- source file 4
- sqrt** function 112
- srand** function 96
- Stack** class 133
- stack 133
- stack frame 33
- stack.h** interface 384, 648
- stackimpl.cpp** file 386
- stackpriv.h** file 386
- Standard Template Library 124
- standard stream 107
- statement terminator 25
- static allocation 71
- stepwise refinement 35
- Stoppard, Tom 116
- stream 107
- string 14
- string literal 99
- string** class 98
- StringToInteger** function 104
- StringToReal** function 104
- Stroustrup, Bjarne 3
- strutils.h** interface 103, 650
- subclass 501
- subset 528
- substr** method 101
- subtree 449
- subtyping 502
- Sudoku 167
- SumIntegerArray** 66
- superclass 501
- switch** statement 26
- symbol table 146, 495
- tail 139
- template 68, 126, 382
- term 16
- terminal symbol 515
- termination condition 28
- text file 105
- this** keyword 318
- tic-tac-toe 132, 257
- tictac.cpp** program 258
- time** function 96
- time-space tradeoff 371
- timestamp 418
- token 154, 493
- tolower** function 112
- top-down design 37
- toString** method 502
- toupper** function 112
- Tower of Hanoi 202
- traveling salesman 295
- traversing a list 365
- traversing a tree 457
- tree 448
- trie 490
- truncation 18
- truth table 23
- type cast 19
- type** method 503
- typedef** keyword 431
- typename** keyword 429
- unary operator 17
- unset** method 111
- union 527
- unparsing 525
- unsigned** type 12
- value assignment 55
- variable 9
- vecimpl.cpp** file 409
- vecpriv.h** file 408
- Vector** class 125, 404
- vector.h** interface 405, 652
- Venn diagram 528
- Venn, John 528
- virtual** keyword 503

walking a list	365
walking a tree	457
Weil, Simone	235
while statement	28
whitespace character	112
Woolf, Virginia	123
word	49
wordfreq.cpp program	161
wrapper	186
wysiwyg	340
 Xerox PARC	 3