PREFACE

WELCOME TO THE FIFTH EDITION OF *C++ Programming: From Problem Analysis to Program Design*. Designed for a first Computer Science (CS1) C++ course, this text provides a breath of fresh air to you and your students. The CS1 course serves as the cornerstone of the Computer Science curriculum. My primary goal is to motivate and excite all CS1 students, regardless of their level. Motivation breeds excitement for learning. Motivation and excitement are critical factors that lead to the success of the programming student. This text is a culmination and development of my classroom notes throughout more than fifty semesters of teaching successful programming to Computer Science students.

> **Warning**: This text can be expected to create a serious reduction in the demand for programming help during your office hours. Other side effects include significantly diminished student dependency on others while learning to program.

C++ Programming: From Problem Analysis to Program Design started as a collection of brief examples, exercises, and lengthy programming examples to supplement the books that were in use at our university. It soon turned into a collection large enough to develop into a text. *The approach taken in this book is, in fact, driven by the students' demand for clarity and readability*. The material was written and rewritten until the students felt comfortable with it. Most of the examples in this book resulted from student interaction in the classroom.

As with any profession, practice is essential. Cooking students practice their recipes. Budding violinists practice their scales. New programmers must practice solving problems and writing code. This is not a C++ cookbook. We do not simply list the C++ syntax followed by an example; we dissect the "why" behind all the concepts. The crucial question of "why?" is answered for every topic when first introduced. This technique offers a bridge to learning C++. Students must understand the "why?" in order to be motivated to learn.

Traditionally, a C++ programming neophyte needed a working knowledge of another programming language. This book assumes no prior programming experience. However, some adequate mathematics background, such as college algebra, is required.

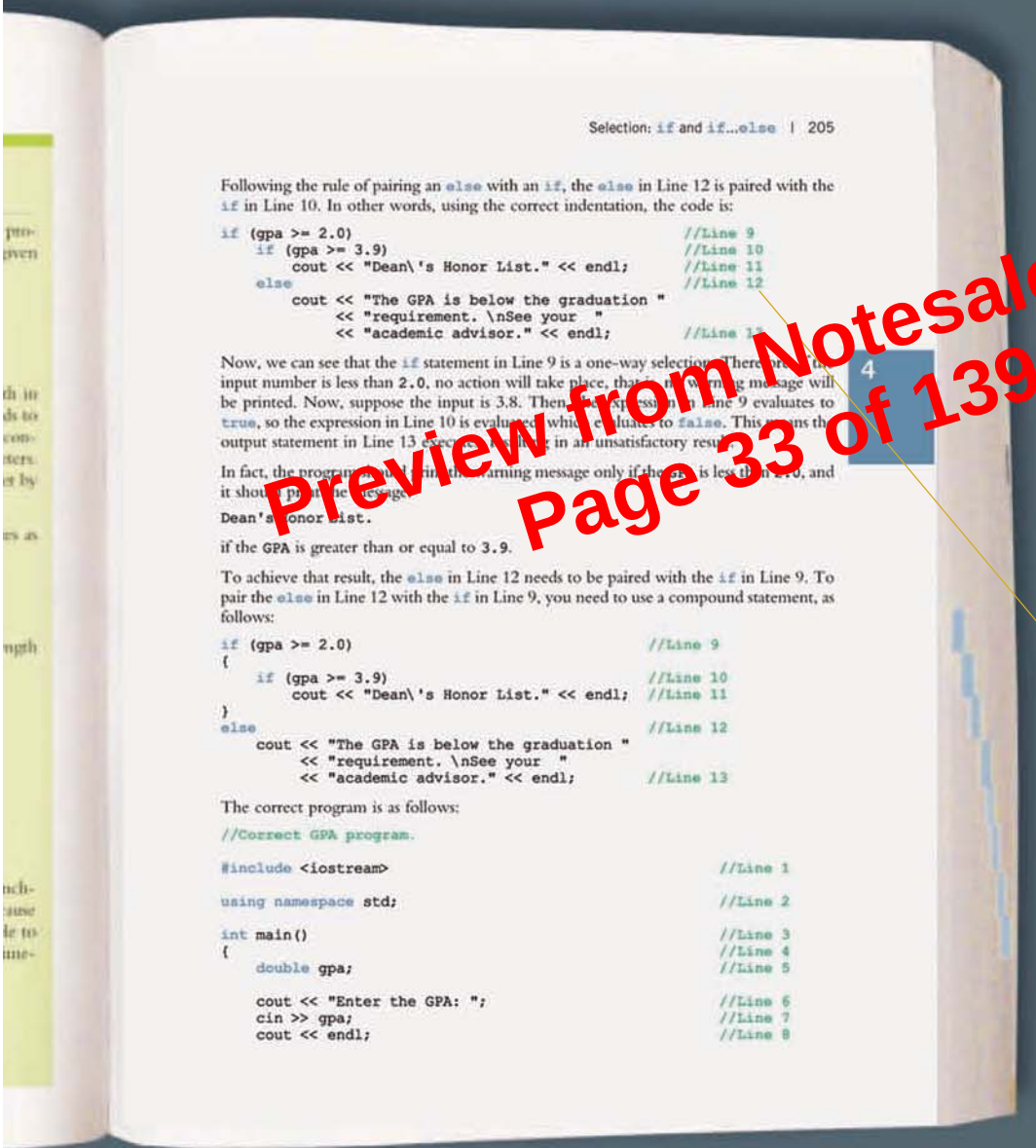
In Figure 1, dotted lines mean the preceding chapter is used in one of the sections of the chapter and is not necessarily a prerequisite for the next chapter. For example, Chapter 9 covers arrays in detail. In Chapters 11 and 12, we show the relationship between arrays and **`struct`**s and arrays and classes, respectively. However, if Chapter 12 is studied before Chapter 9, then the section dealing with arrays in Chapter 12 can be skipped without any discontinuation. This particular section can be studied after studying chapter 9.

It is recommended that the first seven chapters be covered sequentially. After covering the first seven chapters, if the reader is interested in learning OOD and OOP early, then Chapter 12 can be studied right after Chapter 7. Chapter 8 can be studied any time after Chapter 7.

After studying the first seven chapters in sequence, some of the approaches are:

1. Study chapters in the sequence: 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19.
2. Study chapters in the sequence: 9, 12, 14, 15, 13, 17, 18, 19, 10, 16.
3. Study chapters in the sequence: 12, 9, 10, 14, 15, 13, 17, 18, 19, 16.
4. Study chapters in the sequence: 12, 9, 14, 15, 13, 17, 18, 19, 10, 16.

Features of the Book



Following the rule of pairing an else with an if, the else in Line 12 is paired with the if in Line 10. In other words, using the correct indentation, the code is:

```
if (gpa >= 2.0)                                    //Line 9
    if (gpa >= 3.9)                                //Line 10
        cout << "Dean\'s Honor List." << endl;      //Line 11
    else                                           //Line 12
        cout << "The GPA is below the graduation "
             << "requirement. \nSee your  "
             << "academic advisor." << endl;       //Line 13
```

Now, we can see that the if statement in Line 9 is a one-way selection. Therefore, if the input number is less than 2.0, no action will take place, that is, the warning message will be printed. Now, suppose the input is 3.8. Then the expression in Line 9 evaluates to true, so the expression in Line 10 is evaluated, which evaluates to false. This means the output statement in Line 13 executes, resulting in an unsatisfactory result.

In fact, the program will print the warning message only if the GPA is less than 2.0, and it should print the message

```
Dean's Honor List.
```

if the GPA is greater than or equal to 3.9.

To achieve that result, the else in Line 12 needs to be paired with the if in Line 9. To pair the else in Line 12 with the if in Line 9, you need to use a compound statement, as follows:

```
if (gpa >= 2.0)                                    //Line 9
{
    if (gpa >= 3.9)                                //Line 10
        cout << "Dean\'s Honor List." << endl;      //Line 11
}
else                                               //Line 12
    cout << "The GPA is below the graduation "
         << "requirement. \nSee your  "
         << "academic advisor." << endl;           //Line 13
```

The correct program is as follows:

```
//Correct GPA program.

#include <iostream>                                //Line 1

using namespace std;                               //Line 2

int main()                                         //Line 3
{                                                  //Line 4
    double gpa;                                    //Line 5

    cout << "Enter the GPA: ";                     //Line 6
    cin >> gpa;                                    //Line 7
    cout << endl;                                  //Line 8
```


Four-color interior design shows accurate C++ code and related comments.

Chapter 2 defined a program as a sequence of statements whose objective is to accomplish some task. The programs you have examined so far were simple and straightforward. To process a program, the computer begins at the first executable statement and executes the statements in order until it comes to the end. In this chapter and Chapter 5, you will learn how to tell the computer that it does not have to follow a simple sequential order of execution. It can also make decisions and repeat certain statements until certain conditions are met.

Control Structures

A computer can process a program in one of the following ways: in sequence; selectively, by making a choice, which is also called a branch; repetitively, by executing a statement over and over, using a structure called a loop; or by calling a function. Figure 4-1 illustrates the first three types of program flow. (In Chapter 7, we will show how function calls work.) The programming examples in Chapters 2 and 3 included simple sequential programs. With such a program, the computer starts at the beginning and follows the statements in order. No choices are made; there is no repetition. Control structures provide alternatives to sequential program execution and are used to alter the sequential flow of execution. The two most common control structures are selection and repetition. In *selection*, the program executes particular statements depending on some condition(s). In *repetition*, the program repeats particular statements a certain number of times based on some condition(s).

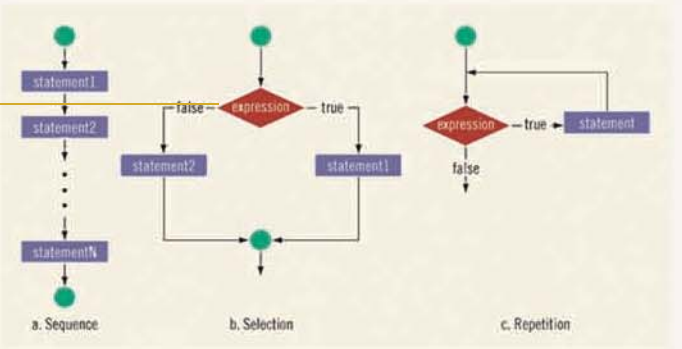


FIGURE 4-1 Flow of execution

More than 300 visual diagrams, both extensive and exhaustive, illustrate difficult concepts.

Processor 2.80 GHz, 1GB RAM, 250 GB HD, VX750 19" Silver Flat CRT Color Monitor" fall into the hardware category; items such as "operating system, games, encyclopedias, and application software" fall into the software category. Let's consider the hardware first.

Hardware

Major hardware components include the central processing unit (CPU); main memory (MM), also called random access memory (RAM); input/output devices; and secondary storage. Some examples of input devices are the keyboard, mouse, and secondary storage. Examples of output devices are the screen, printer, and secondary storage. Let's look at each of these components in greater detail.

Central Processing Unit and Main Memory

The **central processing unit** is the "brain" of the computer and the single most expensive piece of hardware in a computer. The more powerful the CPU, the faster the computer. Arithmetic and logical operations are carried out inside the CPU. Figure 1-1(a) shows some hardware components.

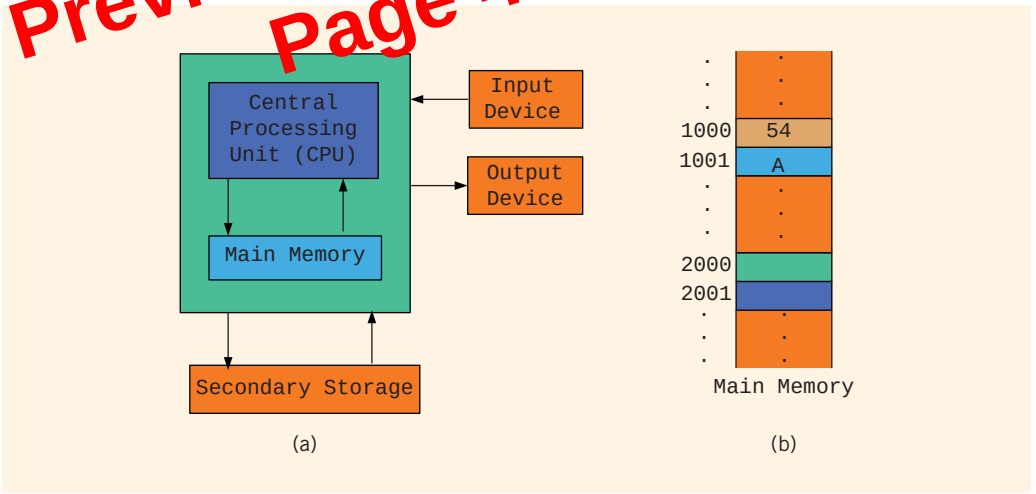


FIGURE 1-1 Hardware components of a computer and main memory

Main memory, or **random access memory**, is connected directly to the CPU. All programs must be loaded into main memory before they can be executed. Similarly, all data must be brought into main memory before a program can manipulate it. When the computer is turned off, everything in main memory is lost.

Main memory is an ordered sequence of cells, called **memory cells**. Each cell has a unique location in main memory, called the **address** of the cell. These addresses help you access the information stored in the cell. Figure 1-1(b) shows main memory with some data.

Today's computers come with main memory consisting of millions to billions of cells. Although Figure 1-1(b) shows data stored in cells, the content of a cell can be either a programming instruction or data. Moreover, this figure shows the data as numbers and letters. However, as explained later in this chapter, main memory stores everything as sequences of `0`s and `1`s. The memory addresses are also expressed as sequences of `0`s and `1`s.

SECONDARY STORAGE

Because programs and data must be stored in main memory before processing and because everything in main memory is lost when the computer is turned off, information stored in main memory must be transferred to some other device for permanent storage. The device that stores information permanently (unless the device becomes unusable or you change the information by rewriting it) is called **secondary storage**. To be able to transfer information from main memory to secondary storage, these components must be directly connected to each other. Examples of secondary storage are hard disks, flash drives, floppy disks, ZIP disks, CD-ROMs, and tapes.

Input/Output Devices

For a computer to perform a useful task, it must be able to take in data and programs and display the results of calculations. The devices that feed data and programs into computers are called **input devices**. The keyboard, mouse, and secondary storage are examples of input devices. The devices that the computer uses to display results are called **output devices**. A monitor, printer, and secondary storage are examples of output devices. Figure 1-2 shows some input and output devices.

FIGURE 1-2 Some input and output devices

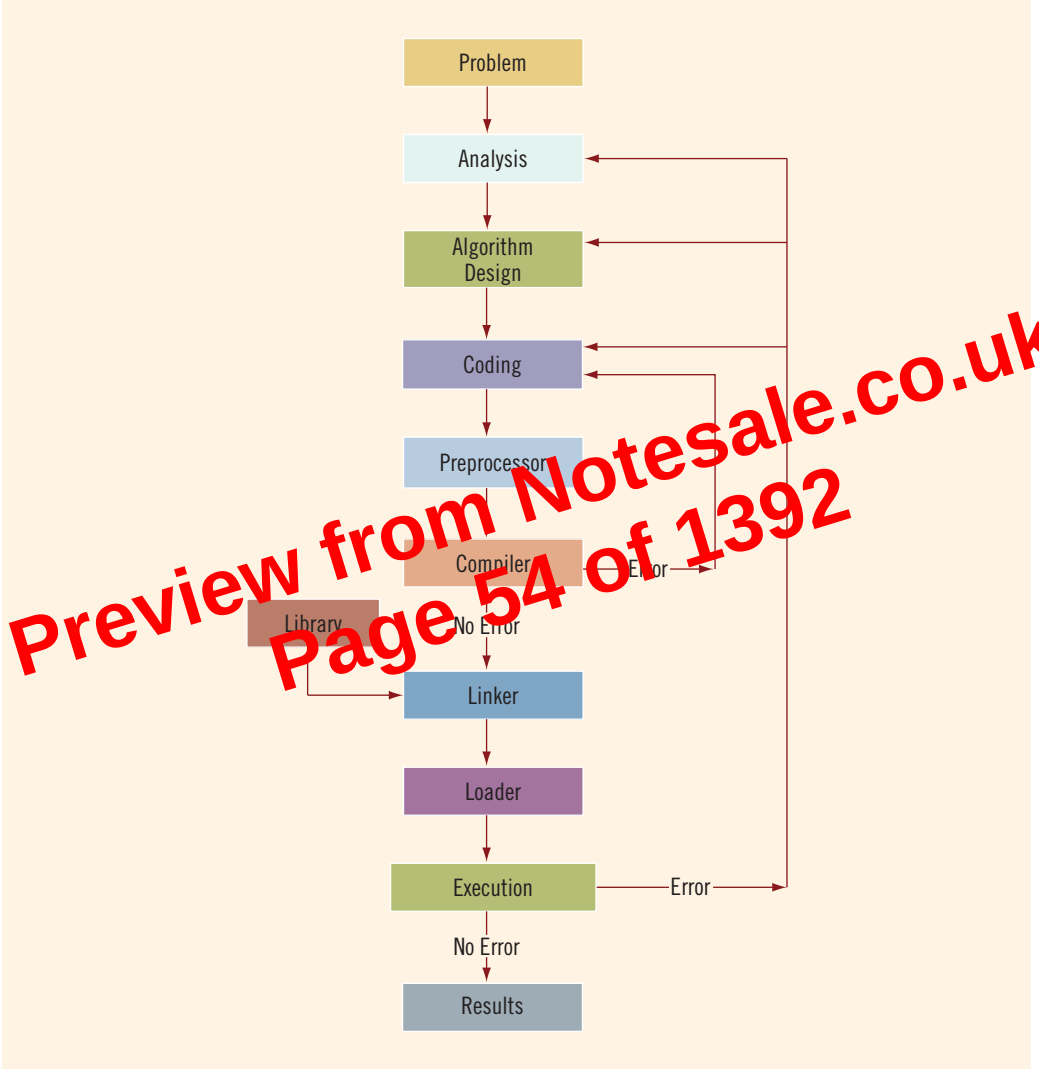


FIGURE 1-4 Problem analysis–coding–execution cycle

To develop a program to solve a problem, you start by analyzing the problem. You then design the algorithm; write the program instructions in a high-level language, or code the program; and enter the program into a computer system.

Analyzing the problem is the first and most important step. This step requires you to do the following:

1. Thoroughly understand the problem.
2. Understand the problem requirements. Requirements can include whether the program requires interaction with the user, whether it manipulates data,

EXAMPLE 1-5

There are 10 students in a class. Each student has taken five tests, and each test is worth 100 points. We want to design an algorithm to calculate the grade for each student, as well as the class average. The grade is assigned as follows: If the average test score is greater than or equal to `90`, the grade is `A`; if the average test score is greater than or equal to `80` and less than `90`, the grade is `B`; if the average test score is greater than or equal to `70` and less than `80`, the grade is `C`; if the average test score is greater than or equal to `60` and less than `70`, the grade is `D`; otherwise, the grade is `F`. Note that the data consists of students' names and their test scores.

This is a problem that can be divided into subproblems as follows: There are five tests, so you design an algorithm to find the average test score. Next, you design an algorithm to determine the grade. The two subproblems are to determine the average test score and to determine the grade.

Let us first design an algorithm to determine the average test score. To find the average test score, add the five test scores and then divide the sum by `5`. Therefore, the algorithm is:

1. Get the five test scores.
2. Add the five test scores. Suppose `sum` stands for the sum of the test scores.
3. Suppose `average` stands for the average test score. Then:

```
average = sum / 5;
```

Next, you design an algorithm to determine the grade. Suppose `grade` stands for the grade assigned to a student. The following algorithm determines the grade:

```
if average is greater than or equal to 90
    grade = A
otherwise
  if average is greater than or equal to 80 and less than 90
      grade = B
otherwise
  if average is greater than or equal to 70 and less than 80
      grade = C
otherwise
  if average is greater than or equal to 60 and less than 70
      grade = D
otherwise
      grade = F
```

You can use the solutions to these subproblems to design the main algorithm as follows: (Suppose `totalAverage` stands for the sum of the averages of each student's test average.)

1. `totalAverage = 0;`
2. *Repeat* the following steps for each student in the class:
   a. Get student's name.
   b. Use the algorithm as discussed above to find the average test score.

c. Use the algorithm as discussed above to find the grade.
d. Update `totalAverage` by adding the current student's average test score.

3. Determine the class average as follows:

```
classAverage = totalAverage / 10
```

A programming exercise in Chapter 7 asks you to write a C++ program to determine the average test score and grade for each student in a class.

Programming Methodologies

Two popular approaches to programming design are the structured approach and the object-oriented approach, which are outlined below.

Structured Programming

Dividing a problem into smaller subproblems is called **structured design**. Each subproblem is then analyzed, and a solution is obtained to solve the subproblem. The solutions to all of the subproblems are then combined to solve the overall problem. This process of implementing a structured design is called **structured programming**. The structured-design approach is also known as **top-down design**, **bottom-up design**, **stepwise refinement**, and **modular programming**.

Object-Oriented Programming

Object-oriented design (OOD) is a widely used programming methodology. In OOD, the first step in the problem-solving process is to identify the components called objects, which form the basis of the solution, and to determine how these objects interact with one another. For example, suppose you want to write a program that automates the video rental process for a local video store. The two main objects in this problem are the video and the customer.

After identifying the objects, the next step is to specify for each object the relevant data and possible operations to be performed on that data. For example, for a video object, the *data* might include:

- movie name
- starring actors
- producer
- production company
- number of copies in stock

Some of the *operations* on a video object might include:

- checking the name of the movie
- reducing the number of copies in stock by one after a copy is rented
- incrementing the number of copies in stock by one after a customer returns a particular video

This illustrates that each object consists of data and operations on that data. An object combines data and operations on the data into a single unit. In OOD, the final program is a collection of interacting objects. A programming language that implements OOD is called an **object-oriented programming (OOP)** language. You will learn about the many advantages of OOD in later chapters.

Because an object consists of data and operations on that data, before you can design and use objects, you need to learn how to represent data in computer memory, how to manipulate data, and how to implement operations. In Chapter 2, you will learn the basic data types of C++ and discover how to represent and manipulate data in computer memory. Chapter 3 discusses how to input data into a C++ program and output the results generated by a C++ program.

To create operations, you write algorithms and implement them in a programming language. Because a data element in a complex program usually has many operations, to separate operations from each other and to use them effectively and in a convenient manner, you use functions to implement algorithms. After a brief introduction in Chapters 2 and 3, you will learn the details of functions in Chapters 6 and 7. Certain algorithms require that a program make decisions, a process called selection. Other algorithms might require certain statements to be repeated until certain conditions are met, a process called repetition. Still other algorithms might require both selection and repetition. You will learn about selection and repetition mechanisms, called control structures, in Chapters 4 and 5. Also, in Chapter 9, using a mechanism called an array, you will learn how to manipulate data when data items are of the same type, such as items in a list of sales figures.

Finally, to work with objects, you need to know how to combine data and operations on the data into a single unit. In C++, the mechanism that allows you to combine data and operations on the data into a single unit is called a class. You will learn how classes work, how to work with classes, and how to create classes in the chapter Classes and Data Abstraction (later in this book).

As you can see, you need to learn quite a few things before working with the OOD methodology. To make this learning easier and more effective, this book purposely divides control structures into two chapters (4 and 5) and user-defined functions into two chapters (6 and 7).

For some problems, the structured approach to program design will be very effective. Other problems will be better addressed by OOD. For example, if a problem requires manipulating sets of numbers with mathematical functions, you might use the structured design approach and outline the steps required to obtain the solution. The C++ library supplies a wealth of functions that you can use effectively to manipulate numbers. On the other hand, if you want to write a program that would make a candy machine operational, the OOD approach is more effective. C++ was designed especially to implement OOD. Furthermore, *OOD works well and is used in conjunction with structured design.*

11. The most basic language of a computer is a sequence of 0s and 1s called machine language. Every computer directly understands its own machine language.
12. A bit is a binary digit, 0 or 1.
13. A byte is a sequence of eight bits.
14. A sequence of 0s and 1s is referred to as a binary code or a binary number.
15. One kilobyte (KB) is $2^{10} = 1024$ bytes; one megabyte (MB) is $2^{20} = 1{,}048{,}576$ bytes; one gigabyte (GB) is $2^{30} = 1{,}073{,}741{,}824$ bytes; one terabyte (TB) is $2^{40} = 1{,}099{,}511{,}627{,}776$ bytes; one petabyte (PB) is $2^{50} = 1{,}125{,}899{,}906{,}842{,}624$ bytes; one exabyte (EB) is $2^{60} = 1{,}152{,}921{,}504{,}606{,}846{,}976$ bytes; and one zettabyte (ZB) is $2^{70} = 1{,}180{,}591{,}620{,}717{,}411{,}303{,}424$ bytes.
16. Assembly language uses easy-to-remember instructions called mnemonics.
17. Assemblers are programs that translate a program written in assembly language into machine language.
18. Compilers are programs that translate a program written in a high-level language into machine code called object code.
19. A linker links the object code with other programs provided by the integrated development environment (IDE) and used in the program to produce executable code.
20. Typically, six steps are needed to execute a C++ program: edit, preprocess, compile, link, load, and execute.
21. A loader transfers executable code into main memory.
22. An algorithm is a step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.
23. The problem-solving process has three steps: analyze the problem and design an algorithm, implement the algorithm in a programming language, and maintain the program.
24. Programs written using the structured design approach are easier to understand, easier to test and debug, and easier to modify.
25. In structured design, a problem is divided into smaller subproblems. Each subproblem is solved, and the solutions to all of the subproblems are then combined to solve the problem.
26. In object-oriented design (OOD), a program is a collection of interacting objects.
27. An object consists of data and operations on that data.
28. The ANSI/ISO Standard C++ syntax was approved in mid-1998.

EXERCISES

1. Mark the following statements as true or false.
   a. The first device known to carry out calculations was the Pascaline.
   b. Modern-day computers can accept spoken-word instructions but cannot imitate human reasoning.

NOTE

In C++, you must declare all identifiers before you can use them. If you refer to an identifier without declaring it, the compiler will generate an error message (syntax error), indicating that the identifier is not declared. Therefore, to use either a named constant or a variable, you must first declare it.

Now that data types, variables, and constants have been defined and discussed, it is possible to offer a formal definition of simple data types. A data type is called **simple** if the variable or named constant of that type can store only one value at a time. For example, if `x` is an `int` variable, at a given time, only one value can be stored in `x`.

Putting Data into Variables

Now that you know how to declare variables, the next question is: How do you put data into those variables? In C++, you can place data into a variable in two ways:

1. Use C++'s assignment statement.
2. Use input (read) statements.

Assignment Statement

The assignment statement takes the following form:

```
variable = expression;
```

In an assignment statement, the value of the `expression` should match the data type of the `variable`. The expression on the right side is evaluated, and its value is assigned to the variable (and thus to a memory location) on the left side.

A variable is said to be **initialized** the first time a value is placed in the variable.

In C++, `=` is called the **assignment operator**.

EXAMPLE 2-13

Suppose you have the following variable declarations:

```
int num1, num2;
double sale;
char first;
string str;
```

Now consider the following assignment statements:

```
num1 = 4;
num2 = 4 * 5 - 11;
sale = 0.02 * 1000;
first = 'D';
str = "It is a sunny day.";
```

	Values of the Variables			Explanation
Before Statement 1	? num1	? num2	? num3	
After Statement 1	18 num1	? num2	? num3	
After Statement 2	45 num1	? num2	? num3	`num1 + 27 = 18 + 27 = 45.` This value is assigned to `num1`, which replaces the old value of `num1`.
After Statement 3	45 num1	45 num2	? num3	Copy the value of `num1` into `num2`.
After Statement 4	45 num1	45 num2	9 num3	`num2 / 5 = 45 / 5 = 9.` This value is assigned to `num3`. So `num3` = 9.
After Statement 5	45 num1	45 num2	2 num3	`num3 / 4 = 9 / 4 = 2.` This value is assigned to `num3`, which replaces the old value of `num3`.

Thus, after the execution of the statement in Line 5, `num1 = 45`, `num2 = 45`, and `num3 = 2`.

Tracing values through a sequence, called a **walk-through**, is a valuable tool to learn and practice. Try it in the sequence above. You will learn more about how to walk through a sequence of C++ statements later in this chapter.

NOTE

Suppose that `x`, `y`, and `z` are `int` variables. The following is a legal statement in C++:

```
x = y = z;
```

In this statement, first the value of `z` is assigned to `y`, and then the new value of `y` is assigned to `x`. Because the assignment operator, `=`, is evaluated from right to left, the **associativity** of the **assignment operator** is said to be from right to left.

Saving and Using the Value of an Expression

Now that you know how to declare variables and put data into them, you can learn how to save the value of an expression. You can then use this value in a later expression without using the expression itself, thereby answering the question raised earlier in this chapter. To save the value of an expression and use it in a later expression, do the following:

1. Declare a variable of the appropriate data type. For example, if the result of the expression is an integer, declare an `int` variable.

During data manipulation, the computer takes the value stored in particular cells and performs a calculation. If you declare a variable and do not store a value in it, the memory cell still has a value—usually the value of the setting of the bits from their last use—and you have no way to know what this value is.

If you only declare a variable and do not instruct the computer to put data into the variable, the value of that variable is garbage. However, the computer does not warn us, regards whatever values are in memory as legitimate, and performs calculations using those values in memory. Using a variable in an expression without initializing it produces erroneous results. To avoid these pitfalls, C++ allows you to initialize variables while they are being declared. For example, consider the following C++ statements in which variables are first declared and then initialized:

```
int first, second;
char ch;
double x;

first = 13;
second = 10;
ch = ' ';
x = 12.6;
```


You can declare and initialize these variables at the same time using the following C++ statements:

```
int first = 13, second = 10;
char ch = ' ';
double x = 12.6;
```

The first C++ statement declares two `int` variables, `first` and `second`, and stores `13` in `first` and `10` in `second`. The meaning of the other statements is similar.

In reality, not all variables are initialized during declaration. It is the nature of the program or the programmer's choice that dictates which variables should be initialized during declaration. The key point is that all variables must be initialized before they are used.

Input (Read) Statement

Previously, you learned how to put data into variables using the assignment statement. In this section, you will learn how to put data into variables from the *standard input device*, using C++'s input (or read) statements.

NOTE In most cases, the standard input device is the keyboard.

When the computer gets the data from the keyboard, the user is said to be acting interactively.

```
    cin >> firstName >> lastName;                    //Line 6
    cin >> age >> weight;                            //Line 7

    cout << "Name: " << firstName << " "
         << lastName << endl;                        //Line 8

    cout << "Age: " << age << endl;                  //Line 9
    cout << "Weight: " << weight << endl;            //Line 10

    return 0;                                        //Line 11
}
```


Sample Run: In this sample run, the user input is shaded.

```
Enter first name, last name, age, and weight, separated by spaces.
Sheila Mann 23 120.5
Name: Sheila Mann
Age: 23
Weight: 120.5
```

The preceding program works as follows: The statements in Lines 1 to 4 declare the variables `firstName` and `lastName` of type `string`, `age` of type `int`, and `weight` of type `double`. The statement in Line 5 is an output statement and tells the user what to do. (Such output statements are called prompt lines.) As shown in the sample run, the input to the program is:

```
Sheila Mann 23 120.5
```

The statement in Line 6 first reads and stores the string `Sheila` into the variable `firstName` and then skips the space after `Sheila` and reads and stores the string `Mann` into the variable `lastName`. Next, the statement in Line 7 first skips the blank after `Mann` and reads and stores `23` into the variable `age` and then skips the blank after `23` and reads and stores `120.5` into the variable `weight`.

The statements in Lines 8, 9, and 10 produce the third, fourth, and fifth lines of the sample run.

NOTE

During programming execution, if more than one value is entered in a line, these values must be separated by at least one blank or tab. Alternately, one value per line can be entered.

Variable Initialization

Remember, there are two ways to initialize a variable: by using the assignment statement and by using a read statement. Consider the following declaration:

```
int feet;
int inches;
```

Next, we show the values of the variables after the execution of each statement.

After St.	Values of the Variables	Explanation
1	4 ? ? ? ? firstNum secondNum z ch name	Store 4 into firstNum.
2	4 14 ? ? ? firstNum secondNum z ch name	2 * firstNum + 6 = 2 * 4 + 6 = 14. Store 14 into secondNum.
3	4 14 2.5 ? ? firstNum secondNum z ch name	(firstNum + 1) / 2.0 = (4 + 1) / 2.0 = 5 / 2.0 = 2.5. Store 2.5 into z.
4	4 14 2.5 A ? firstNum secondNum z ch name	Store 'A' into ch.
5	4 8 2.5 A ? firstNum secondNum z ch name	Read a number from the keyboard (which is 8) and store it into secondNum. This statement replaces the old value of secondNum with this new value.
6	4 8 16.3 A ? firstNum secondNum z ch name	Read a number from the keyboard (which is 16.3) and store this number into z. This statement replaces the old value of z with this new value.
7	32 8 16.3 A ? firstNum secondNum z ch name	2 * secondNum + static_cast<int>(z) = 2 * 8 + static_cast<int> (16.3) = 16 + 16 = 32. Store 32 into firstNum. This statement replaces the old value of firstNum with this new value.
8	32 8 16.3 A Jenny firstNum secondNum z ch name	Read the next input, Jenny, from the keyboard and store it into name.
9	32 9 16.3 A Jenny firstNum secondNum z ch name	secondNum + 1 = 8 + 1 = 9. Store 9 into secondNum.
10	32 9 16.3 D Jenny firstNum secondNum z ch name	Read the next input from the keyboard (which is D) and store it into ch. This statement replaces the old value of ch with the new value.

```
    cout << 29 / 4 << endl;                 //Line  3
    cout << 3.0 / 2 << endl;                //Line  4
    cout << "Hello there.\n";                //Line  5
    cout << 7 << endl;                      //Line  6
    cout << 3 + 5 << endl;                  //Line  7
    cout << "3 + 5";                        //Line  8
    cout << endl;                           //Line  9
    cout << a << endl;                      //Line  10
    cout << "a" << endl;                    //Line  11
    cout << (a + 5) * 6 << endl;            //Line  12
    cout << 2 * b << endl;                  //Line  13

    return 0;
}
```

In the following output, the column marked "Output of Statement at" and the line numbers are not part of the output. The line numbers are shown in this column to make it easy to see which output corresponds to which statement.

```
                                        Output of Statement at
7                                       Line 3
1.5                                     Line 4
Hello there.                            Line 5
7                                       Line 6
8                                       Line 7
3 + 5                                   Line 8
65                                      Line 10
a                                       Line 11
420                                     Line 12
156                                     Line 13
```


For the most part, the output is straightforward. Look at the output of the statements in Lines 7, 8, 9, and 10. The statement in Line 7 outputs the result of `3 + 5`, which is `8`, and moves the insertion point to the beginning of the next line. The statement in Line 8 outputs the string `3 + 5`. Note that the statement in Line 8 consists only of the string `3 + 5`. Therefore, after printing `3 + 5`, the insertion point stays positioned after `5`; it does not move to the beginning of the next line.

The output statement in Line 9 contains only the manipulator `endl`, which moves the insertion point to the beginning of the next line. Therefore, when the statement in Line 10 executes, the output starts at the beginning of the line. Note that in this output, the column "Output of Statement at" does not contain Line 9. This is due to the fact that the statement in Line 9 does not produce any printable output. It simply moves the insertion point to the beginning of the next line. Next, the statement in Line 10 outputs the value of `a`, which is `65`. The manipulator `endl` then moves the insertion point to the beginning of the next line.

only tells the user to input a number, but also informs the user that the number should be between `1` and `10`.

Documentation

The programs that you write should be clear not only to you, but also to anyone else. Therefore, you must properly document your programs. A well-documented program is easier to understand and modify, even a long time after you originally wrote it. You use comments to document programs. Comments should appear in a program to explain the purpose of the program, identify who wrote it, and explain the purpose of particular statements.

Form and Style

You might be thinking that C++ has too many rules. However, in practice, the rules give C++ a great degree of freedom. For example, consider the following two ways of declaring variables:

```
int feet, inches;
double x, y;
```

and:

```
int feet,inches;double x,y;
```

The computer would have no difficulty understanding either of these formats, but the first form is easier to read and follow. Of course, the omission of a single comma or semicolon in either format may lead to all sorts of strange error messages.

What about blank spaces? Where are they significant and where are they meaningless? Consider the following two statements:

```
int a,b,c;
```

and:

```
int    a,    b,    c;
```

Both of these declarations mean the same thing. Here, the blanks between the identifiers in the second statement are meaningless. On the other hand, consider the following statement:

```
inta,b,c;
```

This statement contains a syntax error. The lack of a blank between `int` and the identifier `a` changes the reserved word `int` and the identifier `a` into a new identifier, `inta`.

The clarity of the rules of syntax and semantics frees you to adopt formats that are pleasing to you and easier to understand.

```
int main ()
{
        //Declare variables
    int feet, inches;
    int totalInches;
    double centimeter;

        //Statements: Step 1 - Step 7
    cout << "Enter two integers, one for feet and "
         << "one for inches: ";                          //Step 1
    cin >> feet >> inches;                               //Step 2
    cout << endl;
    cout << "The numbers you entered are " << feet
         << " for feet and " << inches
         << " for inches. " << endl;                     //Step 3

    totalInches = INCHES_PER_FOOT * feet + inches;       //Step 4

    cout << "The total number of inches = "
         << totalInches << endl;                         //Step 5

    centimeter = CENTIMETERS_PER_INCH * totalInches;     //Step 6

    cout << "The number of centimeters = "
         << centimeter << endl;                          //Step 7

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter two integers, one for feet, one for inches: 15 7

The numbers you entered are 15 for feet and 7 for inches.
The total number of inches = 187
The number of centimeters = 474.98
```


PROGRAMMING EXAMPLE: Make Change

Write a program that takes as input any change expressed in cents. It should then compute the number of half-dollars, quarters, dimes, nickels, and pennies to be returned, returning as many half-dollars as possible, then quarters, dimes, nickels, and pennies, in that order. For example, 483 cents should be returned as 9 half-dollars, 1 quarter, 1 nickel, and 3 pennies.

Input Change in cents.

Output Equivalent change in half-dollars, quarters, dimes, nickels, and pennies.

23. The modulus operator, `%`, takes only integer operands.
24. Arithmetic expressions are evaluated using the precedence rules and the associativity of the arithmetic operators.
25. All operands in an integral expression, or integer expression, are integers, and all operands in a floating-point expression are decimal numbers.
26. A mixed expression is an expression that consists of both integers and decimal numbers.
27. When evaluating an operator in an expression, an integer is converted to a floating-point number, with a decimal part of `0`, only if the operator has mixed operands.
28. You can use the cast operator to explicitly convert values from one data type to another.
29. A string is a sequence of zero or more characters.
30. Strings in C++ are enclosed in double quotation marks.
31. A string containing no characters is called a null or empty string.
32. Every character in a string has a relative position in the string. The position of the first character is 0, the position of the second character is 1, and so on.
33. The length of a string is the number of characters in it.
34. During program execution, the contents of a named constant cannot be changed.
35. A named constant is declared by using the reserved word `const`.
36. A named constant is initialized when it is declared.
37. All variables must be declared before they can be used.
38. C++ does not automatically initialize variables.
39. Every variable has a name, a value, a data type, and a size.
40. When a new value is assigned to a variable, the old value is lost.
41. Only an assignment statement or an input (read) statement can change the value of a variable.
42. In C++, `>>` is called the stream extraction operator.
43. Input from the standard input device is accomplished by using `cin` and the stream extraction operator `>>`.
44. When data is input in a program, the data items, such as numbers, are usually separated by blanks, lines, or tabs.
45. In C++, `<<` is called the stream insertion operator.
46. Output of the program to the standard output device is accomplished by using `cout` and the stream insertion operator `<<`.
47. The manipulator `endl` positions the insertion point at the beginning of the next line on an output device.

PROGRAMMING EXERCISES

1. Write a program that produces the following output:

```
**********************************
*     Programming Assignment 1    *
*      Computer Programming I     *
*          Author: ???            *
*    Due Date: Thursday, Jan. 24  *
**********************************
```

In your program, substitute `???` with your own name. If necessary, adjust the positions and the number of the stars to produce a rectangle.

2. Write a program that produces the following output:

```
CCCCCCCCC        ++                ++
CC               ++                ++
CC         ++++++++++++++    ++++++++++++++
CC         ++++++++++++++    ++++++++++++++
CC               ++                ++
CCCCCCCCC        ++                ++
```

3. Consider the following program segment:

```
//include statement(s)
//using namespace statement

int main()
{
    //variable declaration

    //executable statements

    //return statement
}
```

   a. Write C++ statements that include the header files `iostream`.
   b. Write a C++ statement that allows you to use `cin`, `cout`, and `endl` without the prefix `std::`.
   c. Write C++ statements that declare the following variables: `num1`, `num2`, `num3`, and `average` of type `int`.
   d. Write C++ statements that store `125` into `num1`, `28` into `num2`, and `-25` into `num3`.
   e. Write a C++ statement that stores the average of `num1`, `num2`, and `num3`, into `average`.
   f. Write C++ statements that output the values of `num1`, `num2`, `num3`, and `average`.
   g. Compile and run your program.

program that prompts the user to input the masses of the bodies and the distance between the bodies. The program then outputs the force between the bodies.

24. One metric ton is approximately 2205 pounds. Write a program that prompts the user to input the amount of rice, in pounds, in a bag. The program outputs the number of bags needed to store one metric ton of rice.

25. Cindy uses the services of a brokerage firm to buy and sell stocks. The firm charges 1.5% service charges on the total amount for each transaction, buy or sell. When Cindy sells stocks, she would like to know if she gained or lost on a particular investment. Write a program that allows Cindy to input the number of shares sold, the purchase price of each share, and the selling price of each share. The program outputs the amount invested, the total service charges, amount gained or lost, and the amount received after selling the stock.

As you can see in the preceding syntax, a single input statement can read more than one data item by using the operator >> several times. Every occurrence of >> extracts the next data item from the input stream. For example, you can read both payRate and hoursWorked via a single input statement by using the following code:

```
cin >> payRate >> hoursWorked;
```

There is no difference between the preceding input statement and the following two input statements. Which form you use is a matter of convenience and style.

```
cin >> payRate;
cin >> hoursWorked;
```

How does the extraction operator >> work? When scanning for the next input, >> skips all whitespace characters. Recall that whitespace characters consist of blanks and certain nonprintable characters, such as tabs and the newline character. Thus, whether you separate the input data by lines or blanks, the extraction operator >> simply finds the next input data in the input stream. For example, suppose that payRate and hoursWorked are double variables. Consider the following input statement:

```
cin >> payRate >> hoursWorked;
```

Whether the input is:

```
15.50 48.30
```

or:

```
15.50  48.30
```

or:

```
15.50
48.30
```

the preceding input statement would store 15.50 in payRate and 48.30 in hoursWorked. Note that the first input is separated by a blank, the second input is separated by a tab, and the third input is separated by a line.

Now suppose that the input is 2. How does the extraction operator >> distinguish between the character 2 and the number 2? The right-side operand of the extraction operator >> makes this distinction. If the right-side operand is a variable of the data type char, the input 2 is treated as the character 2 and, in this case, the ASCII value of 2 is stored. If the right-side operand is a variable of the data type int or double, the input 2 is treated as the number 2.

Next, consider the input 25 and the statement:

```
cin >> a;
```

where a is a variable of some simple data type. If a is of the data type char, only the single character 2 is stored in a. If a is of the data type int, 25 is stored in a. If a is of the data type

```
int main()
{
    double hours = 35.45;
    double rate = 15.00;
    double tolerance = 0.01000;

    cout << "hours = " << hours << ", rate = " << rate
         << ", pay = " << hours * rate
         << ", tolerance = " << tolerance << endl << endl;

    cout << scientific;
    cout << "Scientific notation: " << endl;
    cout << "hours = " << hours << ", rate = " << rate
         << ", pay = " << hours * rate
         << ", tolerance = " << tolerance << endl << endl;

    cout << fixed;
    cout << "Fixed decimal notation: " << endl;
    cout << "hours = " << hours << ", rate = " << rate
         << ", pay = " << hours * rate
         << ", tolerance = " << tolerance << endl << endl;

    return 0;
}
```


Sample Run:

```
hours = 35.45, rate = 15, pay = 531.75, tolerance = 0.01

Scientific notation:
hours = 3.545000e+001, rate = 1.500000e+001, pay = 5.317500e+002, tolerance = 1
.000000e-002

Fixed decimal notation:
hours = 35.450000, rate = 15.000000, pay = 531.750000, tolerance = 0.010000
```

The sample run shows that when the value of `rate` and `tolerance` are printed without setting the `scientific` or `fixed` manipulators, the trailing zeros are not shown and, in the case of `rate`, the decimal point is also not shown. After setting the manipulators, the values are printed to six decimal places. In the next section, we describe the manipulator `showpoint` to force the system to show the decimal point and trailing zeros. We will then give an example to show how to use the manipulators `setprecision`, `fixed`, and `showpoint` to get the desired output.

`showpoint` Manipulator

Suppose that the decimal part of a decimal number is zero. In this case, when you instruct the computer to output the decimal number in a fixed decimal format, the output may not show the decimal point and the decimal part. To force the output to show the decimal point and

```
    cout << "Line 23: volume = "
         << PI * radius * radius * height << endl;     //Line 23
    cout << "Line 24: PI = " << PI << endl << endl;    //Line 24

    cout << "Line 25: "
         << setprecision(3) << radius << ", "
         << setprecision(2) << height << ", "
         << setprecision(5) << PI << endl;             //Line 25

    return 0;                                          //Line 26
}                                                      //Line 27
```

Sample Run:

```
Line 10: setprecision(2)
Line 11: radius = 12.67
Line 12: height = 12.00
Line 13: volume = 6051.80
Line 14: PI = 3.14

Line 15: setprecision(3)
Line 16: radius = 12.670
Line 17: height = 12.000
Line 18: volume = 6051.797
Line 19: PI = 3.142

Line 20: setprecision(4)
Line 21: radius = 12.6700
Line 22: height = 12.0000
Line 23: volume = 6051.7969
Line 24: PI = 3.1416

Line 25: 12.670, 12.00, 3.14159
```


In this program, the statement in Line 2 includes the header file `iomanip`, and the statement in Line 4 declares the named constant `PI` and sets the value to eight decimal places. The statements in Lines 7 and 8 declare and initialize the variables `radius` and `height` to store the radius of the base and the height of a cylinder. The statement in Line 10 sets the output of floating-point numbers in a fixed decimal format with a decimal point and trailing zeros.

The statements in Lines 11, 12, 13, and 14 output the values of `radius`, `height`, the volume, and `PI` to two decimal places.

The statements in Lines 16, 17, 18, and 19 output the values of `radius`, `height`, the volume, and `PI` to three decimal places.

The statements in Lines 21, 22, 23, and 24 output the values of `radius`, `height`, the volume, and `PI` to four decimal places.

The statement in Line 25 outputs the value of `radius` to three decimal places, the value of `height` to two decimal places, and the value of `PI` to five decimal places.

where `ostreamVar` is an output stream variable. Disabling the manipulator `left` returns the output to the settings of the default output format. For example, the following statement disables the manipulator `left` on the standard output device:

```
cout.unsetf(ios::left);
```

The syntax to set the manipulator `right` is:

```
ostreamVar << right;
```

where `ostreamVar` is an output stream variable. For example, the following statement sets the output to be right-justified on the standard output device:

```
cout << right;
```


On some compliers, the statements `cin >> left;` and `cin >> right;` might not work. In this case, you can use `cin.setf(ios::left);` in place of `cin >> left;` and `cin.setf(ios::right);` in place of `cin >> right;`.

The program in Example 3-14 illustrates the effect of the manipulators `left` and `right`.

EXAMPLE 3-14

```
//Example: left justification

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int x = 15;                                          //Line 1
    int y = 7634;                                        //Line 2

    cout << left;                                        //Line 3

    cout << "12345678901234567890" << endl;              //Line 4
    cout << setw(5) << x << setw(7) << y
         << setw(8) << "Warm" << endl;                   //Line 5

    cout << setfill('*');                                //Line 6

    cout << setw(5) << x << setw(7) << y
         << setw(8) << "Warm" << endl;                   //Line 7

    cout << setw(5) << x << setw(7) << setfill('#')
         << y << setw(8) << "Warm" << endl;              //Line 8

    cout << setw(5) << setfill('@') << x
         << setw(7) << setfill('#') << y
```

```
    cout << "Enter temperature in Fahrenheit: ";          //Line 7
    cin >> fahrenheit;                                    //Line 8
    cout << endl;                                         //Line 9

    celsius = static_cast<int>
              (5.0 / 9 * (fahrenheit - 32) + 0.5);        //Line 10

    cout << fahrenheit << " degree F = "
         << celsius << " degree C. " << endl;             //Line 11

    return 0;                                             //Line 12
}                                                         //Line 13
```

Sample Run: In this sample run, the user input is shaded.

```
Enter temperature in Fahrenheit: 110

110 degree F = 43 degree C.
```

As we can see, using temporary cout statements, we were able to find the problem. After correcting the problem, the temporary cout statements are removed.

The temperature conversion program contained logic errors, not syntax errors. Using `cout` statements to print the values of expressions and/or variables to see the results of a calculation is an effective way to find and correct logic errors.

File Input/Output

The previous sections discussed in some detail how to get input from the keyboard (standard input device) and send output to the screen (standard output device). However, getting input from the keyboard and sending output to the screen have several limitations. Inputting data in a program from the keyboard is comfortable as long as the amount of input is very small. Sending output to the screen works well if the amount of data is small (no larger than the size of the screen) and you do not want to distribute the output in a printed format to others.

If the amount of input data is large, however, it is inefficient to type it at the keyboard each time you run a program. In addition to the inconvenience of typing large amounts of data, typing can generate errors, and unintentional typos cause erroneous results. You must have some way to get data into the program from other sources. By using alternative sources of data, you can prepare the data before running a program, and the program can access the data each time it runs.

Suppose you want to present the output of a program in a meeting. Distributing printed copies of the program output is a better approach than showing the output on a screen. For example, you might give a printed report to each member of a committee before an important meeting. Furthermore, output must sometimes be saved so that the output produced by one program can be used as an input to other programs.

This section discusses how to obtain data from other input devices, such as a disk (that is, secondary storage), and how to save the output to a disk. C++ allows a program to get

Here, `fileStreamVariable` is a file stream variable, and `sourceName` is the name of the input/output file.

Suppose you include the declaration from Step 2 in a program. Further suppose that the input data is stored in a file called `prog.dat`. The following statements associate `inData` with `prog.dat` and `outData` with `prog.out`. That is, the file `prog.dat` is opened for inputting data, and the file `prog.out` is opened for outputting data.

```
inData.open("prog.dat");  //open the input file;  Line 1
outData.open("prog.out"); //open the output file; Line 2
```

NOTE

IDEs such as Visual Studio .Net manage programs in the form of projects. That is, first you create a project, and then you add source files to the project. The statement in Line 1 assumes that the file `prog.dat` is in the same directory (subdirectory) as your project. However, if this is in a different directory (subdirectory), then you must specify the path where the file is located, along with the name of the file. For example, suppose that the file `prog.dat` is on a flash memory in drive H. Then the statement in Line 1 should be modified as follows:

```
inData.open("h:\\prog.dat");
```

Note that there are two \ after h:. Recall from Chapter 2 that in C++, \ is the escape character. Therefore, to produce a \ within a string, you need \\. (To be absolutely sure about specifying the source where the input file is stored, such as the drive `h:\\`, check your system's documentation.)

Similar conventions for the statement in Line 2.

NOTE

Suppose that a program reads data from a file. Because different computers have drives labeled differently, for simplicity, throughout the book, we assume that the file containing the data and the program reading data from the file are in the same directory (subdirectory).

NOTE

We typically use `.dat`, `.out`, or `.txt` as an extension for the input and output files and use Notepad, Wordpad, or TextPad to create and open these files. You can also use your IDE's editor, if any, to create `.txt` (text) files. (To be absolutely sure about it, check you IDE's documentation.)

Step 4 typically works as follows. You use the file stream variables with >>, <<, or other input/output functions. The syntax for using >> or << with file stream variables is exactly the same as the syntax for using `cin` and `cout`. Instead of using `cin` and `cout`, however, you use the file stream variable names that were declared. For example, the statement:

```
inData >> payRate;
```

reads the data from the file `prog.dat` and stores it in the variable `payRate`. The statement:

```
outData << "The paycheck is: $" << pay << endl;
```

PROGRAMMING EXAMPLE: Student Grade

Write a program that reads a student name followed by five test scores. The program should output the student name, the five test scores, and the average test score. Output the average test score with two decimal places.

The data to be read is stored in a file called `test.txt`. The output should be stored in a file called `testavg.out`.

Input: A file containing the student name and the five test scores. A sample input is:

```
Andrew Miller 87.50 89 65.75 37 98.50
```

Output: The student name, the five test scores, and the average of the five test scores, saved to a file.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

To find the average of the five test scores, you add the five test scores and divide the sum by 5. The input data is in the following form: the student name followed by the five test scores. Therefore, you must read the student name first and then read the five test scores. This problem analysis translates into the following algorithm:

1. Read the student name and the five test scores.
2. Output the student name and the five test scores.
3. Calculate the average.
4. Output the average.

You output the average test score in the fixed decimal format with two decimal places.

Variables

The program needs to read a student's first and last name and five test scores. Therefore, you need two variables to store the student name and five variables to store the five test scores.

To find the average, you must add the five test scores and then divide the sum by 5. Thus, you need a variable to store the average test score. Furthermore, because the input data is in a file, you need an `ifstream` variable to open the input file. Because the program output will be stored in a file, you need an `ofstream` variable to open the output file. The program, therefore, needs at least the following variables:

```
ifstream inFile;   //input file stream variable
ofstream outFile;  //output file stream variable

double test1, test2, test3, test4, test5; //variables to
                                  //read the five test scores
double average;    //variable to store the average test score
string firstName;  //variable to store the first name
string lastName;   //variable to store the last name
```

MAIN ALGORITHM

In the preceding sections, we analyzed the problem and determined the formulas to perform the calculations. We also determined the necessary variables and named

NOTE

The preceding program uses five variables—`test1`, `test2`, `test3`, `test4`, and `test5`—to read the five test scores and then find the average test score. The Web site accompanying this book contains a modified version of this program that uses only one variable, `testScore`, to read the test scores and another variable, `sum`, to find the sum of the test scores. The program is named `Ch3_AverageTestScoreVersion2.cpp`.

QUICK REVIEW

1. A stream in C++ is an infinite sequence of characters from a source to a destination.
2. An input stream is a stream from a source to a computer.
3. An output stream is a stream from a computer to a destination.
4. `cin`, which stands for common input, is an input stream object, typically initialized to the standard input device, which is the keyboard.
5. `cout`, which stands for common output, is an output stream object, typically initialized to the standard output device, which is the screen.
6. When the binary operator `>>` is used with an input stream object, such as `cin`, it is called the stream extraction operator. The left-side operand of `>>` must be an input stream variable, such as `cin`; the right-side operand must be a variable.
7. When the binary operator `<<` is used with an output stream object, such as `cout`, it is called the stream insertion operator. The left-side operand of `<<` must be an output stream variable, such as `cout`; the right-side operand of `<<` must be an expression or a manipulator.
8. When inputting data into a variable, the operator `>>` skips all leading whitespace characters.
9. To use `cin` and `cout`, the program must include the header file `iostream`.
10. The function `get` is used to read data on a character-by-character basis and does not skip any whitespace characters.
11. The function `ignore` is used to skip data in a line.
12. The function `putback` puts the last character retrieved by the function `get` back into the input stream.
13. The function `peek` returns the next character from the input stream but does not remove the character from the input stream.
14. Attempting to read invalid data into a variable causes the input stream to enter the fail state.
15. Once an input failure has occurred, you use the function `clear` to restore the input stream to a working state.

18. Suppose that `infile` is an `ifstream` variable and it is associated with the file that contains the following data: `27306 savings 7503.35`. Write the C++ statement(s) that reads and stores the first input in the `int` variable `acctNumber`, the second input in the `string` variable `accountType`, and the third input in the `double` variable `balance`.

19. Suppose that you have the following statements:

```
ofstream outfile;
double distance = 375;
double speed = 58;
double travelTime;
```

Write C++ statements to do the following:

a. Open the file `travel.dat` using the variable `outfile`.
b. Write the statement to format your output to two decimal places in fixed form.
c. Write the values of the variables `day`, `distance`, and `speed` in the file `travel.dat`.
d. Calculate and write the `travelTime` in the file `travel.dat`.
e. Which header files are needed to process the information in (a) to (d)?

PROGRAMMING EXERCISES

1. Consider the following incomplete C++ program:

```
#include <iostream>

int main()
{
  ...
}
```

a. Write a statement that includes the header files `fstream`, `string`, and `iomanip` in this program.
b. Write statements that declare `inFile` to be an `ifstream` variable and `outFile` to be an `ofstream` variable.
c. The program will read data from the file `inData.txt` and write output to the file `outData.txt`. Write statements to open both of these files, associate `inFile` with `inData.txt`, and associate `outFile` with `outData.txt`.
d. Suppose that the file `inData.txt` contains the following data:

```
10.20 5.35
15.6
Randy Gill 31
18500 3.5
A
```

`str1 > "Hen"`	**false** `str1 = "Hello"`. The first two characters of `str1` and `"Hen"` are the same, but the third character `'l'` of `str1` is less than the third character `'n'` of `"Hen"`. Therefore, `str1 > "Hen"` is **false**.
`str3 < "An"`	**true** `str3 = "Air"`. The first characters of `str3` and `"An"` are the same, but the second character `'i'` of `"Air"` is less than the second character `'n'` of `"An"`. Therefore, `str3 < "An"` is **true**.
`str1 == "hello"`	**false** `str1 = "Hello"`. The first character `'H'` of `str1` is less than the first character `'h'` of `"hello"` because the ASCII value of `'H'` is 72, and the ASCII value of `'h'` is 104. Therefore, `str1 == "hello"` is **false**.
`str3 <= str4`	**true** `str3 = "Air"` and `str4 = "Bill"`. The first character `'A'` of `str3` is less than the first character `'B'` of `str4`. Therefore, `str3 <= str4` is **true**.
`str2 > str4`	**true** `str2 = "Hi"` and `str4 = "Bill"`. The first character `'H'` of `str2` is greater than the first character `'B'` of `str4`. Therefore, `str2 > str4` is **true**.

If two strings of different lengths are compared and the character-by-character comparison is equal until it reaches the last character of the shorter string, the shorter string is evaluated as less than the larger string, as shown next.

Expression	Value/Explanation
`str4 >= "Billy"`	**false** `str4 = "Bill"`. It has four characters, and `"Billy"` has five characters. Therefore, `str4` is the shorter string. All four characters of `str4` are the same as the corresponding first four characters of `"Billy"`, and `"Billy"` is the larger string. Therefore, `str4 >= "Billy"` is **false**.
`str5 <= "Bigger"`	**true** `str5 = "Big"`. It has three characters, and `"Bigger"` has six characters. Therefore, `str5` is the shorter string. All three characters of `str5` are the same as the corresponding first three characters of `"Bigger"`, and `"Bigger"` is the larger string. Therefore, `str5 <= "Bigger"` is **true**.

Expression	Value / Explanation
`hours + overTime <= 75.00`	`true` Because `hours + overTime` is `45.30 + 15.00 = 60.30` and `60.30 <= 75.00` is `true`, it follows that `hours + overTime <= 75.00` evaluates to `true`.
`(count >= 0) && (count <= 100)`	`true` Now, `count` is `20`. Because `20 >= 0` is `true`, `count >= 0` is `true`. Also, `20 <= 100` is `true`, so `count <= 100` is `true`. Therefore, `(count >= 0) && (count <= 100)` is `true && true`, which evaluates to `true`.
`('A' <= ch && ch <= 'Z')`	`true` Here, `ch` is `'B'`. Because `'A' <= 'B'` is `true`, `'A' <= ch` evaluates to `true`. Also, because `'B' <= 'Z'` is `true`, `ch <= 'Z'` evaluates to `true`. Therefore, `('A' <= ch && ch <= 'Z')` is `true && true`, which evaluates to `true`.

The following program evaluates and outputs the values of these logical expressions. Note that if a logical expression evaluates to `true`, the corresponding output is `1`; if the logical expression evaluates to `false`, the corresponding output is `0`, as shown in the output at the end of the program. (Recall that if the value of a logical expression is `true`, it evaluates to `1`, and if the value of the logical expression is `false`, it evaluates to `0`.)

```
//Chapter 4 Logical operators

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    bool found = true;
    int age = 20;
    double hours = 45.30;
    double overTime = 15.00;
    int count = 20;
    char ch = 'B';

    cout << fixed << showpoint << setprecision(2);
    cout << "found = " << found << ", age = " << age
         << ", hours = " << hours << ", overTime = " << overTime
         << "," << endl << "count = " << count
         << ", ch = " << ch << endl << endl;

    cout << "!found evaluates to " << !found << endl;
    cout << "hours > 40.00 evaluates to " << (hours > 40.00) << endl;
    cout << "!age evaluates to " << !age << endl;
    cout << "!found && (hours >= 0) evaluates to "
         << (!found && (hours >= 0)) << endl;
```

EXAMPLE 4-12

The following statements show an example of a syntax error.

```
if (hours > 40.0);                                  //Line 1
    wages = 40.0 * rate +
            1.5 * rate * (hours - 40.0);            //Line 2
else                                                //Line 3
    wages = hours * rate;                           //Line 4
```

The semicolon at the end of the `if` statement (see Line 1) ends the `if` statement, so the statement in Line 2 separates the `else` clause from the `if` statement. That is, `else` is all by itself. Because there is no stand-alone `else` statement in C++, this code generates a syntax error. As shown in Example 4-10, in a one-way selection, the semicolon at the end of an `if` statement is a logical error, whereas as shown in this example, in a two-way selection, it is a syntax error.

EXAMPLE 4-13

The following program determines an employee's weekly wages. If the hours worked exceed `40`, wages include overtime payment.

```
//Program: Weekly wages

#include <iostream>
#include <iomanip>

using namespace std;

int main()

    double wages, rate, hours;

    cout << fixed << showpoint << setprecision(2);     //Line 1
    cout << "Line 2: Enter working hours and rate: "; //Line 2
    cin >> hours >> rate;                             //Line 3

    if (hours > 40.0)                                 //Line 4
        wages = 40.0 * rate +
                1.5 * rate * (hours - 40.0);          //Line 5
    else                                              //Line 6
        wages = hours * rate;                         //Line 7

    cout << endl;                                     //Line 8
    cout << "Line 9: The wages are $" << wages
         << endl;                                     //Line 9

    return 0;
}
```

Compound (Block of) Statements

The if and if...else structures control only one statement at a time. Suppose, however, that you want to execute more than one statement if the expression in an if or if...else statement evaluates to true. To permit more complex statements, C++ provides a structure called a **compound statement** or a **block of statements**. A compound statement takes the following form:

```
{
    statement_1
    statement_2
      .
      .
      .
    statement_n
}
```


That is, a compound statement consists of a sequence of statements enclosed in curly braces, {and }. In an if or if...else structure, a compound statement functions as if it was a single statement. Thus, instead of having a simple two-way selection similar to the following code:

```
if (age >= 18)
    cout << "Eligible to vote." << endl;
else
    cout << "Not eligible to vote." << endl;
```

you could include compound statements, similar to the following code:

```
if (age >= 18)
{
    cout << "Eligible to vote." << endl;
    cout << "No longer a minor." << endl;
}
else
{
    cout << "Not eligible to vote." << endl;
    cout << "Still a minor." << endl;
}
```

The compound statement is very useful and will be used in most of the structured statements in this chapter.

Multiple Selections: Nested if

In the previous sections, you learned how to implement one-way and two-way selections in a program. Some problems require the implementation of more than two alternatives. For example, suppose that if the checking account balance is more than $50,000, the interest rate is 7%; if the balance is between $25,000 and $49,999.99, the interest rate is 5%; if the balance is between $1,000 and $24,999.99, the interest rate is 3%; otherwise,

In this code, the `else` in Line 4 is paired with the `if` in Line 2. Note that for the `else` in Line 4, the most recent incomplete `if` is in Line 2. In this code, the `if` in Line 1 has no `else` and is a one-way selection. Once again, the indentation does not determine the pairing, but it communicates the pairing.

EXAMPLE 4-19

Assume that all variables are properly declared, and consider the following statements:

```
if (gender == 'M')                    //Line 1
    if (age < 21 )                    //Line 2
        policyRate = 0.05;            //Line 3
    else                              //Line 4
        policyRate = 0.035;           //Line 5
else if (gender == 'F')               //Line 6
    if (age < 21 )                    //Line 7
        policyRate = 0.04;            //Line 8
    else                              //Line 9
        policyRate = 0.03;            //Line 10
```


In this code, the `else` in Line 4 is paired with the `if` in Line 2. Note that for the `else` in Line 4, the most recent incomplete `if` is the `if` in Line 2. The `else` in Line 6 is paired with the `if` in Line 1. The `else` in Line 9 is paired with the `if` in Line 7. Once again, the indentation does not determine the pairing, but it communicates the pairing.

Comparing `if...else` Statements with a Series of `if` Statements

Consider the following C++ program segments, all of which accomplish the same task.

a.
```
if (month == 1)                       //Line 1
    cout << "January" << endl;        //Line 2
else if (month == 2)                  //Line 3
    cout << "February" << endl;       //Line 4
else if (month == 3)                  //Line 5
    cout << "March" << endl;          //Line 6
else if (month == 4)                  //Line 7
    cout << "April" << endl;          //Line 8
else if (month == 5)                  //Line 9
    cout << "May" << endl;            //Line 10
else if (month == 6)                  //Line 11
    cout << "June" << endl;           //Line 12
```

b.
```
if (month == 1)
    cout << "January" << endl;
if (month == 2)
    cout << "February" << endl;
if (month == 3)
    cout << "March" << endl;
```

For the expression in Line 1, suppose that the value of `age` is `25`. Because `(25 >= 21)` is `true` and the logical operator used in the expression is `||`, the expression evaluates to `true`. Due to short-circuit evaluation, the computer does not evaluate the expression `(x == 5)`. Similarly, for the expression in Line 2, suppose that the value of `grade` is `'B'`. Because `('B' == 'A')` is `false` and the logical operator used in the expression is `&&`, the expression evaluates to `false`. The computer does not evaluate `(x >= 7)`.

Comparing Floating-Point Numbers for Equality: A Precaution

Comparison of floating-point numbers for equality may not behave as you would expect. For example, consider the following program:


```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

int main()
{
    double x = 1.0;
    double y = 3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0;

    cout << fixed << showpoint << setprecision(17);

    cout << "3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 = "
         << 3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0 << endl;

    cout << "x = " << x << endl;
    cout << "y = " << y << endl;

    if (x == y)
        cout << "x and y are the same." << endl;
    else
        cout << "x and y are not the same." << endl;

    if (fabs(x - y) < 0.000001)
        cout << "x and y are the same within the tolerance "
             << "0.000001." << endl;
    else
        cout << " x and y are not the same within the "
             << "tolerance 0.000001." << endl;

    return 0;
}
```

```
    if (gpa >= 2.0)                                          //Line 9
    {                                                        //Line 10
        if (gpa >= 3.9)                                      //Line 11
            cout << "Dean\'s Honor List." << endl;          //Line 12
    }                                                        //Line 13
    else                                                     //Line 14
        cout << "The GPA is below the graduation "
             << "requirement. \nSee your  "
             << "academic advisor." << endl;                 //Line 15

    return 0;                                                //Line 16
}                                                            //Line 17
```

Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1:

```
Enter the GPA: 3.91

Dean's Honor List.
```

Sample Run 2:

```
Enter the GPA: 3.8
```

Sample Run 3:

```
Enter the GPA: 1.95

The GPA is below the graduation requirement.
See your academic advisor.
```


In cases such as this one, the general rule is that you cannot look inside of a block (that is, inside the braces) to pair an `else` with an `if`. The `else` in Line 14 cannot be paired with the `if` in Line 11 because the `if` statement in Line 11 is enclosed within braces, and the `else` in Line 14 cannot look inside those braces. Therefore, the `else` in Line 14 is paired with the `if` in Line 9.

In this book, the C++ programming concepts and techniques are presented in a logical order. When these concepts and techniques are learned one at a time in a logical order, they are simple enough to be understood completely. Understanding a concept or technique completely before using it will save you an enormous amount of debugging time.

Input Failure and the `if` Statement

In Chapter 3, you saw that an attempt to read invalid data causes the input stream to enter a fail state. Once an input stream enters a fail state, all subsequent input statements associated with that input stream are ignored, and the computer continues to execute the program, which produces erroneous results. You can use `if` statements to check the status of an input stream variable and, if the input stream enters the fail state, include instructions that stop program execution.

```
    {
        cout << "Cannot open the input file. "
             << "The program terminates." << endl;
        return  1;
    }

    outFile.open("testavg.out");  //open the output file

    outFile << fixed << showpoint;
    outFile << setprecision(2);

    cout << "Processing data" << endl;

    inFile >> firstName >> lastName;
    outFile << "Student name: " << firstName
            << " " << lastName << endl;

    inFile >> test1 >> test2 >> test3
           >> test4 >> test5;
    outFile << "Test scores: " << setw(4) << test1
            << setw(4) << test2 << setw(4) << test3
            << setw(4) << test4 << setw(4) << test5
            << endl;

    average = (test1 + test2 + test3 + test4 + test5) / 5.0;

    outFile << "Average test score: " << setw(6)
            << average << endl;

    inFile.close();
    outFile.close();

    return 0;
}
```


Confusion between the Equality Operator (==) and the Assignment Operator (=)

Recall that if the decision-making expression in the `if` statement evaluates to `true`, the `statement` part of the `if` statement executes. In addition, the `expression` is usually a logical expression. However, C++ allows you to use *any* expression that can be evaluated to either `true` or `false` as an `expression` in the `if` statement. Consider the following statement:

```
if (x = 5)
    cout << "The value is five." << endl;
```

The `expression`—that is, the decision maker—in the `if` statement is `x = 5`. The expression `x = 5` is called an assignment expression because the operator = appears in the expression and there is no semicolon at the end.

This expression is evaluated as follows. First, the right side of the operator `=` is evaluated, which evaluates to `5`. The value `5` is then assigned to `x`. Moreover, the value `5`—that is, the

If the statement in (a) is **true**, then **x** is larger. If the statement in (b) is **true**, then **y** is larger. However, for this code to work in concert to determine the larger of two integers, the computer needs to evaluate both expressions:

```
(x > y)     and     (y > x)
```

even if the first statement is **true**. Evaluating both expressions is a waste of computer time.

Let's rewrite this pseudo as follows:

```
if (x > y) then
    x is larger
else
    y is larger
```

Here, only one condition needs to be evaluated. This code looks okay, so let's put it into C++.


```
#include <iostream>

using namespace std;

int main()
{
     if (x > y)
```

Wait . . . once you begin translating the pseudo into a C++ program, you should immediately notice that there is no place to store the value of **x** or **y**. The variables were not declared, which is a very common oversight, especially for new programmers. If you examine the pseudo, you will see that the program needs three variables, and you might as well make them self-documenting. Let's start the program code again:

```
#include <iostream>

using namespace std;

int main()
{
    int  num1, num2, larger;      //Line 1

    if   (num1 > num2);           //Line 2; error
         larger = num1;           //Line 3
    else                          //Line 4
         larger = num2;           //Line 5

    return  0;
}
```

Compiling this program will result in the identification of a common syntax error (in Line 2). Recall that a semicolon cannot appear after the **expression** in the

```
int main()                                                    //Line 3
{                                                             //Line 4
    int testScore;                                            //Line 5

    cout << "Enter the test score: ";                         //Line 6
    cin >> testScore;                                         //Line 7
    cout << endl;                                             //Line 8

    switch (testScore / 10)                                   //Line 9
    {                                                         //Line 10
    case 0:                                                   //Line 11
    case 1:                                                   //Line 12
    case 2:                                                   //Line 13
    case 3:                                                   //Line 14
    case 4:                                                   //Line 15
    case 5:                                                   //Line 16
        cout << "The grade is F." << endl;                    //Line 17
    case 6:                                                   //Line 18
        cout << "The grade is D." << endl;                    //Line 19
    case 7:                                                   //Line 20
        cout << "The grade is C." << endl;                    //LIne 21
    case 8:                                                   //Line 22
        cout << "The grade is B." << endl;                    //Line 23
    case 9:                                                   //Line 24
    case 10:                                                  //Line 25
        cout << "The grade is A." << endl;                    //Line 26
    default:                                                  //Line 27
        cout << "Invalid test score." << endl;                //Line 28
    }                                                         //Line 29

    return 0;                                                 //Line 30
}                                                             //Line 31
```


Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1:

```
Enter the test score: 110

Invalid test score.
```

Sample Run 2:

```
Enter the test score: -70

Invalid test score.
```

Sample Run 3:

```
Enter the test score: 75

The grade is C.
The grade is B.
The grade is A.
Invalid test score.
```

Variables Because the program will ask the user to input the customer account number, customer code, number of premium channels, and number of basic service connections, you need variables to store all of this information. Also, because the program will calculate the billing amount, you need a variable to store the billing amount. Thus, the program needs at least the following variables to compute and print the bill:

```
int accountNumber;  //variable to store the customer's
                    //account number
char customerType;     //variable to store the customer code
int numOfPremChannels; //variable to store the number
                       //of premium channels to which the
                       //customer subscribes
int numOfBasicServConn; //variable to store the
                  //number of basic service connections
                  //to which the customer subscribes
double amountDue;  //variable to store the billing amount
```

Named Constants As you can see, the bill processing fee, the cost of a basic service connection, and the cost of a premium channel are fixed, and these values are needed to compute the bill. Although these values are constants in the program, the cable company can change them with little warning. To simplify the process of modifying the program later, instead of using these values directly in the program, you should declare them as named constants. Based on the problem analysis, you need to declare the following named constants:

```
      //Named constants - residential customers
const double RES_BILL_PROC_FEES = 4.50;
const double RES_BASIC_SERV_COST = 20.50;
const double RES_COST_PREM_CHANNEL = 7.50;

      //Named constants - business customers
const double BUS_BILL_PROC_FEES = 15.00;
const double BUS_BASIC_SERV_COST = 75.00;
const double BUS_BASIC_CONN_COST = 5.00;
const double BUS_COST_PREM_CHANNEL = 50.00;
```

Formulas The program uses a number of formulas to compute the billing amount. To compute the residential bill, you need to know only the number of premium channels to which the user subscribes. The following statement calculates the billing amount for a residential customer.

```
amountDue = RES_BILL_PROC_FEES + RES_BASIC_SERV_COST
          + numOfPremChannels * RES_COST_PREM_CHANNEL;
```

To compute the business bill, you need to know the number of basic service connections and the number of premium channels to which the user subscribes. If the number of basic service connections is less than or equal to 10, the cost of the

15. Write a program that calculates and prints the bill for a cellular telephone company. The company offers two types of service: regular and premium. Its rates vary, depending on the type of service. The rates are computed as follows:

Regular service:	$10.00 plus first 50 minutes are free. Charges for over 50 minutes are $0.20 per minute.
Premium service:	$25.00 plus:

    a. For calls made from 6:00 a.m. to 6:00 p.m., the first 75 minutes are free; charges for more than 75 minutes are $0.10 per minute.
    b. For calls made from 6:00 p.m. to 6:00 a.m., the first 100 minutes are free; charges for more than 100 minutes are $0.05 per minute.

    Your program should prompt the user to enter an account number, a service code (type `char`), and the number of minutes the service was used. A service code of `r` or `R` means regular service; a service code of `p` or `P` means premium service. Treat any other character as an error. Your program should output the account number, type of service, number of minutes the telephone service was used, and the amount due from the user. For the premium service, the customer may be using the service during the day and the night. Therefore, to calculate the bill, you must ask the user to input the number of minutes the service was used during the day and the number of minutes the service was used during the night.

16. Write a program to implement the algorithm that you designed in Exercise 22 of Chapter 1. (Assume that the account balance is stored in the file Ch4_Ex16_Data.txt.) Your program should output account balance before and after withdrawal and service charges. Also save the account balance after withdrawal in the file Ch4_Ex16_Output.txt.

17. You have several pictures of different sizes that you would like to frame. A local picture-framing store offers two types of frames—regular and fancy. The frames are available in white and can be ordered in any color the customer desires. Suppose that each frame is 1 inch wide. The cost of coloring the frame is $0.10 per inch. The cost of a regular frame is $0.15 per inch, and the cost of a fancy frame is $0.25 per inch. The cost of putting a cardboard paper behind the picture is $0.02 per square inch, and the cost of putting glass on top of the picture is $0.07 per square inch. The customer can also choose to put crowns on the corners, which costs $0.35 per crown. Write a program that prompts the user to input the following information and then output the cost of framing the picture:

    a. The length and width, in inches, of the picture
    b. The type of the frame
    c. Customer's choice of color to color the frame
    d. If the user wants to put the crowns, then the number of crowns

In Chapter 4, you saw how decisions are incorporated in programs. In this chapter, you learn how repetitions are incorporated in programs.

Why Is Repetition Needed?

Suppose you want to add five numbers to find their average. From what you have learned so far, you could proceed as follows (assume that all variables are properly declared):

```
cin >> num1 >> num2 >> num3 >> num4 >> num5;   //read five numbers
sum = num1 + num2 + num3 + num4 + num5;        //add the numbers
average = sum / 5;                             //find the average
```

But suppose you want to add and average 100, 1000, or more numbers. You would have to declare that many variables and list them again in `cin` statements and, perhaps, again in the output statements. This takes an exorbitant amount of space and time. Also, if you want to run this program again with different values or with a different number of values, you have to rewrite the program.

Suppose you want to add the following numbers:

```
5 3 7 9 4
```

Consider the following statements, in which `sum` and `num` are variables of type `int`:

1. `sum = 0;`
2. `cin >> num;`
3. `sum = sum + num;`

The first statement initializes `sum` to `0`. Let us execute statements 2 and 3. Statement 2 stores `5` in `num`; statement 3 updates the value of `sum` by adding `num` to it. After statement 3, the value of `sum` is `5`.

Let us repeat statements 2 and 3. After statement 2 (after the programming code reads the next number):

```
num = 3
```

After statement 3:

```
sum = sum + num = 5 + 3 = 8
```

At this point, `sum` contains the sum of the first two numbers. Let us again repeat statements 2 and 3 (a third time). After statement 2 (after the code reads the next number):

```
num = 7
```

After statement 3:

```
sum = sum + num = 8 + 7 = 15
```

Now, `sum` contains the sum of the first three numbers. If you repeat statements 2 and 3 two more times, `sum` will contain the sum of all five numbers.

```
    sum = 0;                                                    //Line 4
    counter = 0;                                                //Line 5

    cout << "Line 6: Enter " << limit
         << " integers." << endl;                               //Line 6

    while (counter < limit)                                     //Line 7
    {
        cin >> number;                                          //Line 8
        sum = sum + number;                                     //Line 9
        counter++;                                              //Line 10
    }

    cout << "Line 11: The sum of the " << limit
         << " numbers = " << sum << endl;                       //Line 11

    if (counter != 0)                                           //Line 12
        cout << "Line 13: The average = "
             << sum / counter << endl;                          //Line 13
    else                                                        //Line 14
        cout << "Line 15: No input." << endl;                   //Line 15

    return 0;                                                   //Line 16
}
```


Sample Run: In this sample run, the user input is shaded.

```
Line 1: Enter the number of integers in the list: 12

Line 6: Enter 12 integers.
8 9 2 3 90 38 56 8 23 89 7 2
Line 11: The sum of the 12 numbers = 335
Line 13: The average = 27
```

This program works as follows. The statement in Line 1 prompts the user to input the number of data items. The statement in Line 2 reads the next input line and stores it in the variable `limit`. The value of `limit` indicates the number of items in the list. The statements in Lines 4 and 5 initialize the variables `sum` and `counter` to `0`. (The variable `counter` is the loop control variable.) The statement in Line 6 prompts the user to input numbers. (In this sample run, the user is prompted to enter 12 integers.) The `while` statement in Line 7 checks the value of `counter` to determine how many items have been read. If `counter` is `less` than `limit`, the `while` loop proceeds for the next iteration. The statement in Line 8 reads the next number and stores it in the variable `number`. The statement in Line 9 updates the value of `sum` by adding the value of `number` to the previous value, and the statement in Line 10 increments the value of `counter` by `1`. The statement in Line 11 outputs the sum of the numbers; the statements in Lines 12 through 15 output the average.

Note that `sum` is initialized to `0` in Line 4 in this program. In Line 9, after reading a number at Line 8, the program adds it to the sum of all the numbers scanned before the current number. The first number read will be added to zero (because `sum` is initialized to `0`), giving the correct sum of the first number. To find the average, divide `sum` by `counter`. If `counter`

```
srand(time(0));
num = rand() % 100;
```

The first statement sets the seed, and the second statement generates a random number greater than or equal to `0` and less than `100`. Note how the function `time` is used. It is used with an argument, that is, parameter, which is `0`.

The program uses the `bool` variable `isGuessed` to control the loop. The `bool` variable `isGuessed` is initialized to `false`. It is set to `true` when the user guesses the correct number.

```
//Flag-controlled while loop.
//Number guessing game.

#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
    //declare the variables
    int num;        //variable to store the random
                    //number
    int guess;      //variable to store the number
                    //guessed by the user
    bool isGuessed; //boolean variable to control
                    //the loop

    srand(time(0));                                  //Line 1
    num = rand() % 100;                              //Line 2

    isGuessed = false;                               //Line 3

    while (!isGuessed)                               //Line 4
    {                                                //Line 5
        cout << "Enter an integer greater"
             << " than or equal to 0 and "
             << "less than 100: ";                   //Line 6

        cin >> guess;                                //Line 7
        cout << endl;                                //Line 8

        if (guess == num)                            //Line 9
        {                                            //Line 10
            cout << "You guessed the correct "
                 << "number." << endl;               //Line 11
            isGuessed = true;                        //Line 12
        }                                            //Line 13
        else if (guess < num)                        //Line 14
            cout << "Your guess is lower than the "
                 << "number.\n Guess again!"
                 << endl;                            //Line 15
```

Case 4: EOF-Controlled while Loops

If the data file is frequently altered (for example, if data is frequently added or deleted), it's best not to read the data with a sentinel value. Someone might accidentally erase the sentinel value or add data past the sentinel, especially if the programmer and the data entry person are different people. Also, it can be difficult at times to select a good sentinel value. In such situations, you can use an **end-of-file (EOF)-controlled while loop**.

Until now, we have used an input stream variable, such as cin, and the extraction operator, >>, to read and store data into variables. However, the input stream variable can also return a value after reading data, as follows:

1. If the program has reached the end of the input data, the input stream variable returns the logical value false.
2. If the program reads any faulty data (such as a char value into an int variable), the input stream enters the fail state. Once a stream enters the fail state, any further I/O operations using that stream are considered to be null operations; that is, they have no effect. Unfortunately, the computer does not halt the program or give any error messages. It just continues executing the program, silently ignoring each additional attempt to use that stream. In this case, the input stream variable returns the value false.
3. In cases other than (1) and (2), the input stream variable returns the logical value true.

You can use the value returned by the input stream variable to determine whether the program has reached the end of the input data. Because the input stream variable returns the logical value true or false, in a while loop, it can be considered a logical expression.

The following is an example of an EOF-controlled while loop:

```
cin >> variable;      //initialize the loop control variable

while (cin)           //test the loop control variable
{
    .
    .
    .
    cin >> variable; //update the loop control variable
    .
    .
    .
}
```

Notice that here, the variable cin acts as the loop control variable.

eof Function

In addition to checking the value of an input stream variable, such as cin, to determine whether the end of the file has been reached, C++ provides a function that you can use with an input stream variable to determine the end-of-file status. This function is called

```
int current;     //variable to store the current
                 //Fibonacci number
int counter;     //loop control variable
int nthFibonacci; //variable to store the desired
                  //Fibonacci number
```

To calculate the third Fibonacci number, add the values of `previous1` and `previous2` and store the result in `current`. To calculate the fourth Fibonacci number, add the value of the second Fibonacci number (that is, `previous2`) and the value of the third Fibonacci number (that is, `current`). Thus, when the fourth Fibonacci number is calculated, you no longer need the first Fibonacci number. Instead of declaring additional variables, which could be too many, after calculating a Fibonacci number to determine the next Fibonacci number, `current` becomes `previous2` and `previous2` becomes `previous1`. Therefore, you can again use the variable `current` to store the next Fibonacci number. This process is repeated until the desired Fibonacci number is calculated. Initially, `previous1` and `previous2` are the first two elements of the sequence, supplied by the user. From the preceding discussion, it follows that you need five variables.

MAIN ALGORITHM

1. Prompt the user for the first two numbers—that is, `previous1` and `previous2`.
2. Read (input) the first two numbers into `previous1` and `previous2`.
3. Output the first two Fibonacci numbers. (Echo input.)
4. Prompt the user for the position of the desired Fibonacci number.
5. Read the position of the desired Fibonacci number into `nthFibonacci`.
6. a. `if` (`nthFibonacci == 1`)
   the desired Fibonacci number is the first Fibonacci number. Copy the value of `previous1` into `current`.
   b. `else if` (`nthFibonacci == 2`)
   the desired Fibonacci number is the second Fibonacci number. Copy the value of `previous2` into `current`.
   c. `else` `calculate` the desired Fibonacci number as follows:

   Because you already know the first two Fibonacci numbers of the sequence, start by determining the third Fibonacci number.

   c.1. Initialize `counter` to `3` to keep track of the calculated Fibonacci numbers.
   c.2. Calculate the next Fibonacci number, as follows:

   ```
   current = previous2 + previous1;
   ```

   c.3. Assign the value of `previous2` to `previous1`.
   c.4. Assign the value of `current` to `previous2`.
   c.5. Increment `counter`.

```
    else if (nthFibonacci == 2)                    //Step 6.b
        current = previous2;
    else                                           //Step 6.c
    {
        counter = 3;                               //Step 6.c.1

               //Steps 6.c.2 - 6.c.5
        while (counter <= nthFibonacci)
        {
            current = previous2 + previous1;       //Step 6.c.2
            previous1 = previous2;                 //Step 6.c.3
            previous2 = current;                   //Step 6.c.4
            counter++;                             //Step 6.c.5
        }//end while
    }//end else

    cout << "The Fibonacci number at position "
         << nthFibonacci << " is " << current
         << endl;                                  //Step 7

    return 0;
}//end main
```


Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1:

```
Enter the first two Fibonacci numbers: 12 16

The first two Fibonacci numbers are 12 and 16
Enter the position of the desired Fibonacci number: 10

The Fibonacci number at position 10 is 796
```

Sample Run 2:

```
Enter the first two Fibonacci numbers: 1 1

The first two Fibonacci numbers are 1 and 1
Enter the position of the desired Fibonacci number: 15

The Fibonacci number at position 15 is 610
```

for Looping (Repetition) Structure

The **while** loop discussed in the previous section is general enough to implement most forms of repetitions. The C++ **for** looping structure discussed here is a specialized form of the **while** loop. Its primary purpose is to simplify the writing of counter-controlled loops. For this reason, the **for** loop is typically called a counted or indexed **for** loop.

Next, the update statement increments the value of `i` by `1`, so the value of `i` becomes `11`. Now the loop condition evaluates to `false` and the `for` loop terminates. Note that the output statement in Line 2 executes only once.

4. Consider the following `for` loop:

```
for (i = 1; i <= 10; i++);  //Line 1
    cout << i << " ";       //Line 2
cout << endl;               //Line 3
```

This `for` loop has no effect on the output statement in Line 2. The semicolon at the end of the `for` statement terminates the `for` loop; the action of the `for` loop is thus empty. The output statement is all by itself and executes only once.

5. Consider the following `for` loop:

```
for (i = 1; ; i++)
    cout << i << " ";
cout << endl;
```

In this `for` loop, because the loop condition is omitted from the `for` statement, the loop condition is always `true`. This is an infinite loop.

EXAMPLE 5-15

In this example, a `for` loop reads five numbers and finds their sum and average. Consider the following program code, in which `i`, `newNum`, `sum`, and `average` are `int` variables.

```
sum = 0;

for (i = 1; i <= 5; i++)
{
   cin >> newNum;
   sum = sum + newNum;
}

average = sum / 5;
cout << "The sum is " << sum << endl;
cout << "The average is " << average << endl;
```

In the preceding `for` loop, after reading a `newNum`, this value is added to the previously calculated (partial) `sum` of all the numbers read before the current number. The variable `sum` is initialized to `0` before the `for` loop. Thus, after the program reads the first number and adds it to the value of `sum`, the variable `sum` holds the correct `sum` of the first number.

NOTE

The syntax of the **for** loop, which is:

```
for (initial expression; logical expression; update expression)
    statement
```

is functionally equivalent to the following **while** statement:

```
initial expression
while (expression)
{
     statement
     update expression
}
```

For example, the following **for** and **while** loops are equivalent:

```
for (int i = 0; i < 10; i++)
    cout << i <<  " ";
cout << endl;
```

```
int i = 0;
while (i < 10)
{
    cout << i <<  " ";
    i++;
}
cout << endl;
```


If the number of iterations of a loop is known or can be determined in advance, typically programmers use a **for** loop.

EXAMPLE 5-16 (FIBONACCI NUMBER PROGRAM: REVISITED)

The Programming Example: Fibonacci Number given in the previous section uses a **while** loop to determine the desired Fibonacci number. You can replace the **while** loop with an equivalent **for** loop as follows:

```
for (counter = 3; counter <= nthFibonacci; counter++)
{
    current = previous2 + previous1;
    previous1 = previous2;
    previous2 = current;
    counter++;
} //end for
```

The complete program listing of the program that uses a **for** loop to determine the desired Fibonacci number is given at the Web site accompanying this book. The program is named `Ch5_FibonacciNumberUsingAForLoop.cpp`.

In the following C++ program, we recommend that you walk through each step.

```
int counter;    //loop control variable
int number;     //variable to store the number read
int zeros;      //variable to store the zero count
int evens;      //variable to store the even count
int odds;       //variable to store the odd count
```

Clearly, you must initialize the variables `zeros`, `evens`, and `odds` to zero. You can initialize these variables when you declare them.

MAIN ALGORITHM

1. Initialize the variables.
2. Prompt the user to enter 20 numbers.
3. For each number in the list:
   a. Read the number.
   b. Output the number (echo input).
   c. If the number is even:
      {
      i. Increment the even count.
      ii. If the number is zero, increment the zero count.
      }
      otherwise
          Increment the odd count.
4. Print the results.

Before writing the C++ program, let us describe Steps 1–4 in greater detail. It will be much easier for you to then write the instructions in C++.

1. Initialize the variables. You can initialize the variables `zeros`, `evens`, and `odds` when you declare them.
2. Use an output statement to prompt the user to enter 20 numbers.
3. For Step 3, you can use a `for` loop to process and analyze the 20 numbers. In pseudocode, this step is written as follows:

```
for (counter = 1; counter <= 20; counter++)
{
    read the number;
    output number;

    switch (number % 2)     // check the remainder
    {
    case 0:
        increment even count;
        if (number == 0)
            increment zero count;
        break;
```

to sum, check whether num is negative. If num is negative, an error message appears on the screen and isNegative is set to true. In the next iteration, when the expression in the while statement is evaluated, it evaluates to false because !isNegative is false. (Note that because isNegative is true, !isNegative is false.)

The following while loop is written without using the variable isNegative:

```
sum = 0;
cin >> num;

while (cin)
{
    if (num < 0)    //if num is negative, terminate the loop
    {
        cout << "Negative number found in the data." << endl;
        break;
    }

    sum = sum + num;
    cin >> num;
}
```


In this form of the while loop, when a negative number is found, the expression in the if statement evaluates to true; after printing an appropriate message, the break statement terminates the loop. (After executing the break statement in a loop, the remaining statements in the loop are discarded.)

NOTE

The break statement is an effective way to avoid extra variables to control a loop and produce an elegant code. However, break statements must be used very sparingly within a loop. An excessive use of these statements in a loop will produce spaghetti-code (loops with many exit conditions) that can be very hard to understand and manage. You should be extra careful in using break statements and ensure that the use of the break statements makes the code more readable and not less readable. If you're not sure, don't use break statements.

The continue statement is used in while, for, and do...while structures. When the continue statement is executed in a loop, it skips the remaining statements in the loop and proceeds with the next iteration of the loop. In a while and do...while structure, the expression (that is, the loop-continue test) is evaluated immediately after the continue statement. In a for structure, the update statement is executed after the continue statement, and then the loop condition (that is, the loop-continue test) executes.

If the previous program segment encounters a negative number, the while loop terminates. If you want to discard the negative number and read the next number rather than terminate the loop, replace the break statement with the continue statement, as shown in the following example:

```
sum = 0;
cin >> num;
```

(Assume that ch is a variable of type char.) The general loop to process the data is:

```
infile >> ID;                                   //Line 1
while (infile)                                  //Line 2
{                                               //Line 3
    infile.get(ch);                             //Line 4
    getline(infile, name);                      //Line 5

    //process the numbers in each line          //Line 6
    //output the name and total votes
    infile >> ID;     //begin processing the next line
}
```

The code to read and sum up the voting data is:

```
sum = 0;                        //Line 6
infile >> num;                  //Line 7; read the first number
while (num != -999)             //Line 8
{                               //Line 9
    sum = sum + num;            //Line 10; update sum
    infile >> num;              //Line 11; read the next number
}                               //Line 12
```

We can now write the following nested loop to process data as follows:

```
infile >> ID;                           //Line 1
while (infile)                          //Line 2
{                                       //Line 3
    infile.get(ch);                     //Line 4
    getline(infile, name);              //Line 5
    sum = 0;                            //Line 6
    infile >> num;                      //Line 7; read the first number
    while (num != -999)                 //Line 8
    {                                   //Line 9
        sum = sum + num;                //Line 10; update sum
        infile >> num;                  //Line 11; read the next number
    }

    cout << "Name: " << name
         << ", Votes: " << sum
         << endl;                       //Line 12

    infile >> ID;       //Line 13; begin processing the next line
}
```


Avoiding Bugs by Avoiding Patches

Debugging sections in the previous chapters illustrated how to debug syntax and logical errors, and how to avoid partially understood concepts. In this section, we illustrate how to avoid a software patch to fix a code. A software patch is a piece of code written on top of an existing piece of code and intended to fix a bug in the original code.

program closely, we can see that the four lines are produced because the outer loop executes four times. The values assigned to loop control variable `i` are `1`, `2`, `3`, and `4`. This is an example of the classic "off-by-one" problem. (In an "off-by-one problem," either the loop executes one too many or one too few times.) We can eliminate this problem by correctly setting the values of the loop control variable. For example, we can rewrite the loops as follows:

```
for (i = 1; i <= 3; i++)
{
    sum = 0;

    for (j = 1; j <= 4; j++)
    {
        infile >> num;
        cout << num << " ";
        sum = sum + num;
    }

    cout << "sum = " << sum << endl;
}
```


This code fixes the original problem without using a software patch. It also represents good programming practice. The complete modified program is available at the Web site accompanying this book and is named `Ch5_LoopWithBugsCorrectedProgram.cpp`.

Debugging Loops

As we have seen in the earlier debugging sections, no matter how careful a program is designed and coded, errors are likely to occur. If there are syntax errors, the compiler will identify them. However, if there are logical errors, we must carefully look at the code or even maybe at the design and try to find the errors. To increase the reliability of the program, errors must be discovered and fixed before the program is released to the users.

Once an algorithm is written, the next step is to verify that it works properly. If the algorithm is a simple sequential flow or contains a branch, it can be hand traced or you can use the debugger, if any, provided by the IDE. Typically, loops are harder to debug. The correctness of a loop can be verified by using loop invariants. A loop invariant is a set of statements that remains true each time the loop body is executed. Let *p* be a loop invariant and *q* be the (logical) expression in a loop statement. Then *p* `&&` q remains true before each iteration of the loop and *p* `&&` not(*q*) is true after the loop terminates. The full discussion of loop invariants is beyond the scope of the book. However, you can learn about loop invariants in the book: *Discrete Mathematical Structures: Theory and Applications,* D.S. Malik and M.K. Sen, Course Technology, 2004. Here, we give a few tips that you can use to debug a loop.

As discussed in the previous section, the most common error associated with loops is off-by-one. If a loop turns out to be an infinite loop, the error is most likely in the logical expression that controls the execution of the loop. Check the logical expression carefully and see if you have reversed an inequality, an assignment statement symbol appears in place of the equality operator, or `&&` appears in place of `||`. If the loop changes the values of

16. Putting a semicolon at the end of the `for` loop (before the body of the `for` loop) is a semantic error. In this case, the action of the `for` loop is empty.
17. The syntax of the `do...while` statement is:

```
do
    statement
while (expression);
```

`statement` is called the body of the `do...while` loop.

18. Both `while` and `for` loops are called pretest loops. A `do...while` loop is called a posttest loop.
19. The `while` and `for` loops may not execute at all, but the `do...while` loop always executes at least once.
20. Executing a `break` statement in the body of a loop immediately terminates the loop.
21. Executing a `continue` statement in the body of a loop skips the loop's remaining statements and proceeds with the next iteration.
22. When a `continue` statement executes in a `while` or `do...while` loop, the expression update statement in the body of the loop may not execute.
23. After a `continue` statement executes in a `for` loop, the update statement is the next statement executed.

EXERCISES

1. Mark the following statements as true or false.
   a. In a counter-controlled `while` loop, it is not necessary to initialize the loop control variable.
   b. It is possible that the body of a `while` loop may not execute at all.
   c. In an infinite `while` loop, the `while` expression (the decision maker) is initially false, but after the first iteration it is always true.
   d. The `while` loop:

```
j = 0;
while (j <= 10)
    j++;
```

   terminates if `j > 10`.
   e. A sentinel-controlled `while` loop is an event-controlled `while` loop whose termination depends on a special value.
   f. A loop is a control structure that causes certain statements to execute over and over.
   g. To read data from a file of an unspecified length, an EOF-controlled loop is a good choice.

h. When a **while** loop terminates, the control first goes back to the statement just before the **while** statement, and then the control goes to the statement immediately following the **while** loop.

2. What is the output of the following C++ code?

```
int count = 1;
int y = 100;
while (count < 100)
{
    y = y - 1;
    count++;
}
cout << " y = " << y << " and count = " << count << endl;
```

3. What is the output of the following C++ code?

```
int num = 5;
while (num > 5)
    num = num + 2;
cout << num << endl;
```

4. What is the output of the following C++ code?

```
int num = 1;
while (num < 10)
{
    cout << num << " ";
    num = num + 2;
}
cout << endl;
```

5. When does the following **while** loop terminate?

```
ch = 'D';
while ('A' <= ch && ch <= 'Z')
    ch = static_cast<char>(static_cast<int>(ch) + 1);
```

6. Suppose that the input is `38 35 71 14 -1`. What is the output of the following code? Assume all variables are properly declared.

```
cin >> sum;
cin >> num;

for (j = 1; j <= 3; j++)
{
    cin >> num;
    sum = sum + num;
}
cout << "Sum = " << sum << endl;
```

7. Suppose that the input is `38 35 71 14 -1`. What is the output of the following code? Assume all variables are properly declared.

```
cin >> sum;
cin >> num;

while (num != -1)
```

```
{
    sum = sum + num;
    cin >> num;
}
cout << "Sum = " << sum << endl;
```

8. Suppose that the input is `38 35 71 14 -1`. What is the output of the following code? Assume all variables are properly declared.

```
cin >> num;
sum = num;

while (num != -1)
{
    cin >> num;
    sum = sum + num;
}
cout << "Sum = " << sum << endl;
```

9. Suppose that the input is `38 35 71 14 -1`. What is the output of the following code? Assume all variables are properly declared.

```
sum = 0;
cin >> num;
while (num != -1)
{
    sum = sum + num;
    cin >> num;
}
cout << "Sum = " << sum << endl;
```

10. Correct the following code so that it finds the sum of 20 numbers.

```
sum = 0;

while (count < 20)
    cin >> num;
    sum = sum + num;
    count++;
```

11. What is the output of the following program?

```
#include <iostream>

using namespace std;

int main()
{
    int  x, y, z;

    x = 4;   y = 5;
    z = y + 6;

    while(((z - x) % 4) != 0)
    {
        cout << z << " ";
        z = z + 7;
    }
```

```
    cout << endl;

    return 0;
}
```

12. Suppose that the input is:

```
58  23  46  75  98  150  12  176  145 -999
```

What is the output of the following program?

```
#include <iostream>

using namespace std;

int main()
{
    int num;

    cin >> num;

    while (num != -999)
    {
        cout << num % 25 << " ";
        cin >> num;
    }

    cout << endl;

    return 0;
}
```


13. The following program is designed to input two numbers and output their sum. It asks the user if he/she would like to run the program. If the answer is `Y` or `y`, it prompts the user to enter two numbers. After adding the numbers and displaying the results, it again asks the user if he/she would like to add more numbers. However, the program fails to do so. Correct the program so that it works properly.

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    char response;
    double num1;
    double num2;

    cout << "This program adds two numbers." << endl;
    cout << "Would you like to run the program: (Y/y) ";
    cin >> response;
    cout << endl;
```

40. Given the following program segment:

```
j = 2;
for (i = 1; i <= 5; i++);
{
   cout << setw(4) << j;
   j = j + 5;
}
cout << endl;
```

write a **while** loop and a **do**...**while** loop that have the same output.

41. What is the output of the following program?

```
#include <iostream>

using namespace std;

int main()
{
    int x, y, z;
    x = 4;  y = 5;
    z = y + 6;
    do
    {
        cout << z << " ";
        z = z + 7;
    }
    while (((z - x) % 4) != 0);

    cout << endl;

    return 0;
}
```

42. To learn how nested `for` loops work, do a walk-through of the following program segments and determine, in each case, the exact output.

a.
```
int i, j;

for (i = 1; i <= 5; i++)
{
    for (j = 1; j <= 5; j++)
        cout << setw(3) << i;
    cout << endl;
}
```

b.
```
int i, j;
for (i = 1; i <= 5; i++)
{
    for (j = (i + 1); j <= 5; j++)
        cout << setw(5) << j;
    cout << endl;
}
```

that prompts the user to enter the number of lockers in a school. After the game is over, the program outputs the number of lockers that are opened. Test run your program for the following inputs: `1000`, `5000`, `10000`. Do you see any pattern developing?

(*Hint*: Consider locker number 100. This locker is visited by student numbers 1, 2, 4, 5, 10, 20, 25, 50, and 100. These are the positive divisors of 100. Similarly, locker number 30 is visited by student numbers 1, 2, 3, 5, 6, 10, 15, and 30. Notice that if the number of positive divisors of a locker number is odd, then at the end of the game, the locker is opened. If the number of positive divisors of a locker number is even, then at the end of the game, the locker is closed.)

19. When you borrow money to buy a house, a car, or for some other purpose, you repay the loan by making periodic payments over a certain period of time. Of course, the lending company will charge interest on the loan. Every periodic payment consists of the interest on the loan and the payment toward the principal amount. To be specific, suppose that you borrow $1000 at the interest rate of 7.2% per year and the payments are monthly. Suppose that your monthly payment is $25. Now, the interest is 7.2% per year and the payments are monthly, so the interest rate per month is 7.2/12 = 0.6%. The first month's interest on $1000 is 1000 × 0.006 = 6. Because the payment is $25 and interest for the first month is $6, the payment toward the principal amount is 25 – 6 = 19. This means after making the first payment, the loan amount is 1000 – 19 = 981. For the second payment, the interest is calculated on $981. So the interest for the second month is 981 × 0.006 = 5.886, that is, approximately $5.89. This implies that the payment toward the principal is 25 – 5.89 = 19.11 and the remaining balance after the second payment is 981 – 19.11 = 961.89. This process is repeated until the loan is paid. Write a program that accepts as input the loan amount, the interest rate per year, and the monthly payment. (Enter the interest rate as a percentage. For example, if the interest rate is 7.2% per year, then enter 7.2.) The program then outputs the number of months it would take to repay the loan. (Note that if the monthly payment is less than the first month's interest, then after each payment, the loan amount will increase. In this case, the program must warn the borrower that the monthly payment is too low, and with this monthly payment, the loan amount could not be repaid.)

20. Enhance your program from Exercise 19 by first telling the user the minimum monthly payment and then prompting the user to enter the monthly payment. Your last payment might be more than the remaining loan amount and interest on it. In this case, output the loan amount before the last payment and the actual amount of the last payment. Also, output the total interest paid.

21. Write a complete program to test the code in Example 5-21.

22. Write a complete program to test the code in Example 5-22.

In C++, `return` is a reserved word.

When a `return` statement executes in a function, the function immediately terminates and the control goes back to the caller. Moreover, the function call statement is replaced by the value returned by the `return` statement. When a `return` statement executes in the function `main`, the program terminates.

To put the ideas in this discussion to work, let us write a function that determines the larger of two numbers. Because the function compares two numbers, it follows that this function has two parameters and that both parameters are numbers. Let us assume that the data type of these numbers is floating-point (decimal)—say, `double`. Because the larger number is of type `double`, the function's data type is also `double`. Let us name this function `larger`. The only thing you need to complete this function is the body of the function. Thus, following the syntax of a function, you can write this function as follows:


```
double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}
```

Note that the function `larger` requires that you use an additional variable `max` (called a **local declaration**, in which `max` is a variable local to the function `larger`). Figure 6-1 describes various parts of the function `larger`.

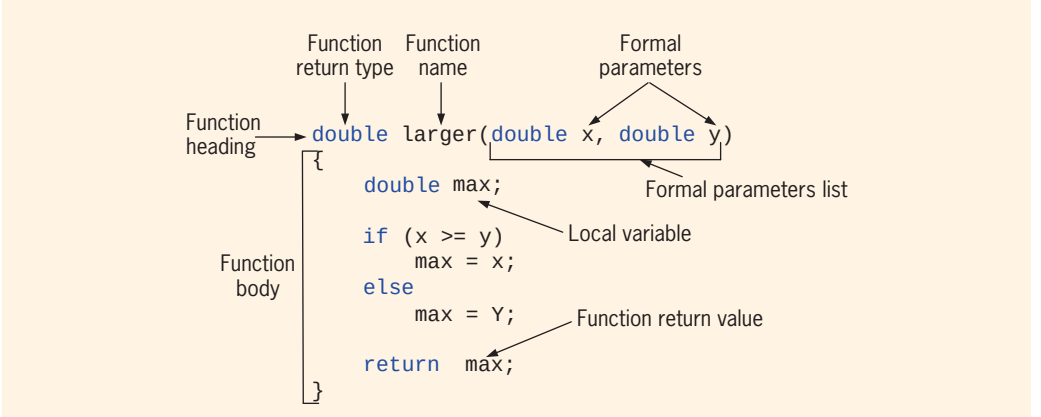


FIGURE 6-1 Various parts of the function `larger`

Syntax: Function Prototype

The general syntax of the function prototype of a value-returning function is:

```
functionType functionName(parameter list);
```

(Note that the function prototype ends with a semicolon.)

For the function `larger`, the prototype is:

```
double larger(double x, double y);
```

NOTE

When writing the function prototype, you do not have to specify the variable name in the parameter list. However, you must specify the data type of each parameter.

You can rewrite the function prototype of the function `larger` as follows:

```
double larger(double, double);
```


FINAL PROGRAM

You now know enough to write the entire program, compile it, and run it. The following program uses the functions `larger`, `compareThree`, and `main` to determine the larger/largest of two or three numbers.

```
//Program: Largest of three numbers

#include <iostream>

using namespace std;

double larger(double x, double y);
double compareThree(double x, double y, double z);

int main()
{
    double one, two;                                    //Line 1

    cout << "Line 2: The larger of 5 and 10 is "
         << larger(5, 10) << endl;                      //Line 2

    cout << "Line 3: Enter two numbers: ";              //Line 3
    cin >> one >> two;                                  //Line 4
    cout << endl;                                       //Line 5

    cout << "Line 6: The larger of " << one
         << " and " << two << " is "
         << larger(one, two) << endl;                   //Line 6
```

2. For each remaining number in the list:
   a. Read the next number. Store it in a variable called num.
   b. Compare num and max. If max < num, then num is the new largest number, so update the value of max by copying num into max. If max >= num, discard num; that is, do nothing.
3. Because max now contains the largest number, print it.

To find the larger of two numbers, the program uses the function larger.

COMPLETE PROGRAM LISTING

```
//*****************************************************
// Author: D.S. Malik
//
// This program finds the largest number of a set of 10
// numbers.
//*****************************************************

#include <iostream>
using namespace std;

double larger(double x, double y);

int main()
{
    double num; //variable to hold the current number
    double max; //variable to hold the larger number
    int count;  //loop control variable

    cout << "Enter 10 numbers." << endl;
    cin >> num;                                  //Step 1
    max = num;                                   //Step 1

    for (count = 1; count < 10; count++)         //Step 2
    {
        cin >> num;                              //Step 2a
        max = larger(max, num);                  //Step 2b
    }

    cout << "The largest number is " << max
         << endl;                                //Step 3

    return 0;
}//end main
```

c. Calculate the bill.
d. Return the amount due.

This function contains a statement to prompt the user to enter the number of premium channels (Step a) and a statement to read the number of premium channels (Step b). Other items needed to calculate the billing amount, such as the cost of basic service connection and bill-processing fees, are defined as named constants (before the definition of the function `main`). Therefore, to calculate the billing amount, this function does not need to get any value from the function `main`. This function, therefore, has no parameters.

Local Variables (Function residential)

From the previous discussion, it follows that the function `residential` requires variables to store both the number of premium channels and the billing amount. This function needs only two local variables to calculate the billing amount:

```
int noOfPChannels;  //number of premium channels
double bAmount;     //billing amount
```

The definition of the function `residential` can now be written as follows:

```
double residential()
{
    int noOfPChannels;  //number of premium channels
    double bAmount;     //billing amount
    cout << "Enter the number of premium "
         << "channels used: ";
    cin >> noOfPChannels;
    cout << endl;

    bAmount = RES_BILL_PROC_FEES +
              RES_BASIC_SERV_COST +
              noOfPChannels * RES_COST_PREM_CHANNEL;

    return bAmount;
}
```

Function business

To compute the business bill, you need to know the number of both the basic service connections and the premium channels to which the customer subscribes. Then, based on these numbers, you can calculate the billing amount. The billing amount is then returned using the `return` statement. The following six steps describe this function:

a. Prompt the user for the number of basic service connections.
b. Read the number of basic service connections.
c. Prompt the user for the number of premium channels.
d. Read the number of premium channels.
e. Calculate the bill.
f. Return the amount due.

```
Enter the number of basic service connections: 25

Enter the number of premium channels used: 9

Account number = 21341
Amount due = $615.00
```

QUICK REVIEW

1. Functions are like miniature programs and are called modules.
2. Functions enable you to divide a program into manageable tasks.
3. The C++ system provides the standard (predefined) functions.
4. To use a standard function, you must:
   i. Know the name of the header file that contains the function's specification,
   ii. Include that header file in the program, and
   iii. Know the name and type of the function and number and types of the parameters (arguments).
5. There are two types of user-defined functions: value-returning functions and void functions.
6. Variables defined in a function heading are called formal parameters.
7. Expressions, variables, or constant values used in a function call are called actual parameters.
8. In a function call, the number of actual parameters and their types must match with the formal parameters in the order given.
9. To call a function, use its name together with the actual parameter list.
10. A value-returning function returns a value. Therefore, a value-returning function is used (called) in either an expression or an output statement or as a parameter in a function call.
11. The general syntax of a user-defined function is:

```
functionType  functionName(formal parameter list)
{
    statements
}
```

12. The line `functionType functionName(formal parameter list)` is called the function heading (or function header). Statements enclosed between braces (`{` and `}`) are called the body of the function.
13. The function heading and the body of the function are called the definition of the function.

Your program must contain at least the following functions: a function that calculates and returns the mean and a function that calculates the standard deviation.

11. When you borrow money to buy a house, a car, or for some other purposes, then you typically repay it by making periodic payments. Suppose that the loan amount is L, r is the interest rate per year, m is the number of payments in a year, and the loan is for t years. Suppose that $i = (r / m)$ and r is in decimal. Then the periodic payment is:

$$R = \frac{Li}{1 - (1 + i)^{-mt}},$$

You can also calculate the unpaid loan balance after making certain payments. For example, the unpaid balance after making k payments is:

$$L' = R\left[\frac{1 - (1 + i)^{-(mt-k)}}{i}\right]$$

where R is the periodic payment. (Note that if the payments are monthly, then $m = 12$.)

Write a program that prompts the user to input the values of L, r, m, t, and k. The program then outputs the apropriate values. Your program must contain at least two functions, with appropriate parameters, to calculate the periodic payments and the unpaid balance after certain payments. Make the program menu driven and use a loop so that the user can repeat the program for different values.

12. During the tax season, every Friday, J&J accounting firm provides assistance to people who prepare their own tax returns. Their charges are as follows.

    a. If a person has low income (<= 25,000) and the consulting time is less than or equal to 30 minutes, there are no charges; otherwise, the service charges are 40% of the regular hourly rate for the time over 30 minutes.

    b. For others, if the consulting time is less than or equal to 20 minutes, there are no service charges; otherwise, service charges are 70% of the regular hourly rate for the time over 20 minutes.

(For example, suppose that a person has low income and spent 1 hour and 15 minutes, and the hourly rate is \$70.00. Then the billing amount is $70.00 \times 0.40 \times (45 / 60) = \$21.00$.)

Write a program that prompts the user to enter the hourly rate, the total consulting time, and whether the person has low income. The program should output the billing amount. Your program must contain a function that takes as input the hourly rate, the total consulting time, and a value indicating whether the person has low income. The function should return the billing amount. Your program may prompt the user to enter the consulting time in minutes.

Value Parameters

The previous section defined two types of parameters—value parameters and reference parameters. Example 7-3 shows a program that uses a function with parameters. Before considering more examples of void functions with parameters, let us make the following observation about value and reference parameters. When a function is called, the value of the actual parameter is copied into the corresponding formal parameter. If the formal parameter is a value parameter, then after copying the value of the actual parameter, there is no connection between the formal parameter and actual parameter; that is, the formal parameter has its own copy of the data. Therefore, during program execution, the formal parameter manipulates the data stored in its own memory space. The program in Example 7-4 further illustrates how a value parameter works.

EXAMPLE 7-4

The following program shows how a formal parameter of a primitive data type works.

```
//Example 7-4
//Program illustrating how a value parameter works.
#include <iostream>

using namespace std;

void funcValueParam(int num);

int main()
{
    int number = 6;                                    //Line 1

    cout << "Line 2: Before calling the function "
         << "funcValueParam, number = " << number
         << endl;                                      //Line 2

    funcValueParam(number);                            //Line 3

    cout << "Line 4: After calling the function "
         << "funcValueParam, number = " << number
         << endl;                                      //Line 4

      return 0;
}

void funcValueParam(int num)
{
    cout << "Line 5: In the function funcValueParam, "
         << "before changing, num = " << num
         << endl;                                      //Line 5

    num = 15;                                          //Line 6
```

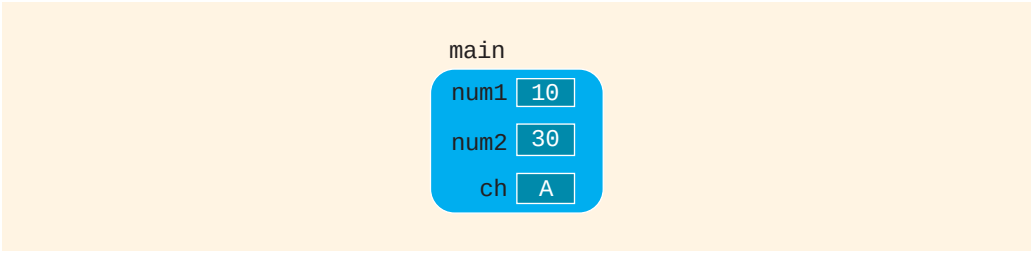


FIGURE 7-11 Values of the variables when control goes back to Line 6

Line 6 produces the following output:

```
Line 6: After funOne: num1 = 10, num2 = 30, and ch =
```

The statement in Line 7 is a function call to the function `funTwo`. Now `funTwo` has three parameters: `x`, `y`, and `w`. Also, `x` and `w` are reference parameters, and `y` is a value parameter. Thus, `x` receives the address of its corresponding actual parameter, which is `num2`, and `w` receives the address of its corresponding actual parameter, which is `ch`. The variable `y` copies the value `25` into its memory cell. Figure 7-12 shows the values before the statement in Line 14 executes.

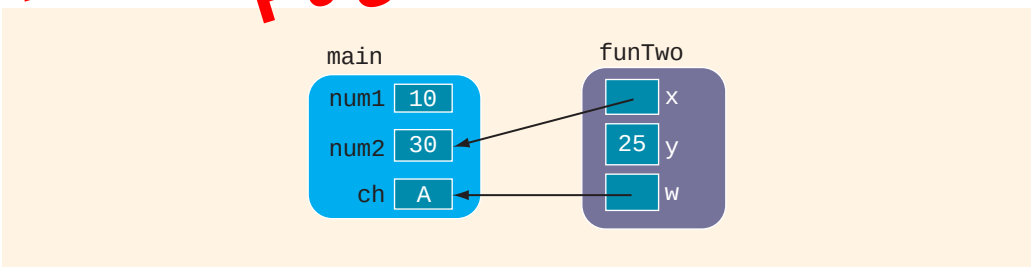


FIGURE 7-12 Values of the variables before the statement in Line 14 executes

After the statement in Line 14, `x++;`, executes, the variables are as shown in Figure 7-13. (Note that the variable `x` changed the value of `num2`.)

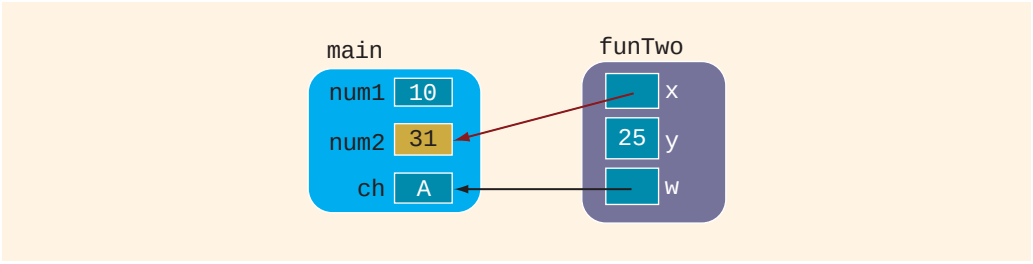


FIGURE 7-13 Values of the variables after the statement in Line 14 executes

Table 7-1 summarizes the scope (visibility) of the identifiers.

TABLE 7-1 Scope (Visibility) of the Identifiers

Identifier	Visibility in one	Visibility in two	Visibility in three	Visibility in Block four	Visibility in main
RATE (before main)	Y	Y	Y	Y	Y
z (before main)	Y	Y	N	N	N
t (before main)	Y	Y	Y	Y	Y
main	Y	Y	Y	Y	Y
local variables of main	N	N	N	N	Y
one (function name)	Y	Y	N	N	Y
x (one's formal parameter)	Y	N	N	N	N
y (one's formal parameter)	Y	N	N	N	N
w (before function two)	N	Y	Y	Y	N
two (function name)	Y	Y	Y	Y	Y
a (two's formal parameter)	N	Y	N	N	N
b (two's formal parameter)	N	Y	N	N	N
x (two's formal parameter)	N	Y	N	N	N
local variables of two	N	Y	N	N	N
three (function name)	Y	Y	Y	Y	Y
one (three's formal parameter)	N	N	Y	Y	N
y (three's formal parameter)	N	N	Y	Y	N
z (three's formal parameter)	N	N	Y	Y	N
ch (three's local variable)	N	N	Y	Y	N
a (three's local variable)	N	N	Y	N	N
x (Block four's local variable)	N	N	N	Y	N
a (Block four's local variable)	N	N	N	Y	N

global variables in one area of a program might be misunderstood as problems caused in another area.

For example, consider the following program:

```
//Global variable

#include <iostream>

using namespace std;

int t;

void funOne(int& a);

int main()
{
    t = 15;                                                    //Line 1

    cout << "Line 2: In main: t = " << t << endl;              //Line 2

    funOne(t);                                                 //Line 3

    cout << "Line 4: In main after funOne: "
         << " t = " << t << endl;                              //Line 4

    return 0;                                                  //Line 5
}

void funOne(int& a)
{
    cout << "Line 6: In funOne: a = " << a
         << " and t = " << t << endl;                          //Line 6

    a = a + 12;                                                //Line 7
    cout << "Line 8: In funOne: a = " << a
         << " and t = " << t << endl;                          //Line 8

    t = t + 13;                                                //Line 9

    cout << "Line 10: In funOne: a = " << a
         << " and t = " << t << endl;                          //Line 10
}
```


This program has a variable `t` that is declared before the definition of any function. Because none of the functions has an identifier `t`, the variable `t` is accessible anywhere in the program. Also, the program consists of a void function with a reference parameter.

In Line 3, the function `main` calls the function `funOne`, and the actual parameter passed to `funOne` is `t`. So, `a`, the formal parameter of `funOne`, receives the address of `t`. Any changes that `a` makes to its value immediately change `t`. Because `t` can be directly accessed anywhere in the program, in Line 9, the function `funOne` changes the value of `t`

```
    do
    {
        showChoices();
        cin >> choice;
        cout << endl;

        switch (choice)
        {
        case 1:
            cout << "Enter feet and inches: ";
            cin >> feet >> inches;
            cout << endl;
            feetAndInchesToMetersAndCent(feet, inches,
                                         meters, centimeters);
            cout << feet << " feet(foot), "
                 << inches << " inch(es) = "
                 << meters << " meter(s) "
                 << centimeters << " centimeter(s) " << endl;
            break;

        case 2:
            cout << "Enter meters and centimeters: ";
            cin >> meters >> centimeters;
            cout << endl;
            meterAndCentiToFeetAndInches(meters, centimeters,
                                         feet, inches);
            cout << meters << " meter(s), "
                 << centimeters << " centimeter(s) = "
                 << feet << " feet(foot), "
                 << inches << " inch(es)."
                 << endl;
            break;

        case 99:
            break;

        default:
            cout << "Invalid input." << endl;
        }
    }
    while (choice != 99);

    return 0;
}

void showChoices()
{
    cout << "Enter--" << endl;
    cout << "1: To convert from feet and inches to meters "
         << "and centimeters." << endl;
    cout << "2: To convert from meters and centimeters to feet "
         << "and inches." << endl;
    cout << "99: To quit the program." << endl;
}
```

```
void poolFillTime(double len, double wid, double dep,
                  double fRate, int& fTime)
{
    double poolWaterCapacity;

    poolWaterCapacity = poolCapacity(len, wid, dep);
    fTime = static_cast<int> (poolWaterCapacity / fRate + 0.5);
}

void print(int fTime)
{
    cout << "The time to fill the pool is approximately: "
         << ftime / 60 << " hour(s) and " << ftime % 60
         << " minute(s)." << endl;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter the length, width, and the depth of the pool (in feet): 30 15 10

Enter the rate of the water (in gallons per minute): 100

The time to fill the pool is approximately: 5 hour(s) and 37 minute(s).
```


As you can see, the program contains the function `poolCapacity` to find the amount of water needed to fill the pool, the function `poolFillTime` to find the time to fill the pool, and some other functions. Now, to calculate the time to fill the pool, you must know the amount of the water needed and the rate at which the water is released in the pool. Because the results of the function `poolCapacity` are needed in the function `poolFillTime`, the function `poolFillTime` cannot be tested alone. Does this mean that we must write the functions in a specific order? Not necessarily, especially when different people are working on different parts of the program. In situations such as these, we use function stubs. A function **stub** is a function that is not fully coded. For a void function, a function stub might consist of only a function header and a set of empty braces, `{}`, and for a value-returning function it might contain only a return statement with a plausible return value. For example, the function stub for the function `poolCapacity` can be:

```
double poolCapacity(double len, double wid, double dep)
{
  return 1000.00;
}
```

This allows the function `poolCapacity` to be called while the program is being coded. Ultimately, the stub for function `poolCapacity` is replaced with a function that properly calculates the amount of water needed to fill the pool based on the values of the parameters. In the meantime, the function stub allows work to continue on other parts of the program that call the function `poolCapacity`.

Before we look at some programming examples, another concept about functions is worth mentioning: function overloading.

NOTE

In the previous program, because the data is assumed to be input from the standard input device (the keyboard) and the function `getNumber` returns only one value, you can also write the function `getNumber` as a value-returning function. If written as a value-returning function, the definition of the function `getNumber` is:

```
int getNumber()
{
    int num;

    cin >> num;

    return num;
}
```

In this case, the statement (function call):

```
getNumber(number);
```

in the function `main` should be replaced by the statement:

```
number = getNumber();
```

Of course, you also need to change the function prototype.

PROGRAMMING EXAMPLE: Data Comparison

This programming example illustrates:

- How to read data from more than one file in the same program.
- How to send output to a file.
- How to generate bar graphs.
- With the help of functions and parameter passing, how to use the same program segment on different (but similar) sets of data.
- How to use structured design to solve a problem and how to perform parameter passing.

This program is broken into two parts. First, you learn how to read data from more than one file. Second, you learn how to generate bar graphs.

Two groups of students at a local university are enrolled in certain special courses during the summer semester. The courses are offered for the first time and are taught by different teachers. At the end of the semester, both groups are given the same tests for the same courses, and their scores are recorded in separate files. The data in each file is in the following form:

The definition of the function `printResult` follows:

```
void printResult(ofstream& outp, string courseID, int groupNo,
                 double avg)
{
    if (groupNo == 1)
       outp << "  " << courseID << "   ";
    else
       outp << "        ";

    outp << setw(8) << groupNo << setw(17) << avg << endl;
} //end printResult
```

Now that we have designed and defined the functions `calculateAverage` and `printResult`, we can describe the algorithm for the function `main`. Before outlining the algorithm, however, we note the following. It is quite possible that in both input files, the data is ordered according to the course IDs, but one file might have fewer courses than the other. We cannot discover this error until after we have processed both files and discovered that one file has unprocessed data. Make sure to check for this error before printing the final answer—that is, the averages for group 1 and group 2.

MAIN ALGORITHM: Function `main`

1. Declare the variables (local declaration).
2. Open the input files.
3. Print a message if you are unable to open a file and terminate the program.
4. Open the output file.
5. To output floating-point numbers in a fixed decimal format with the decimal point and trailing zeros, set the manipulators `fixed` and `showpoint`. Also, to output floating-point numbers to two decimal places, set the precision to two decimal places.
6. Initialize the course average for group 1 to `0.0`.
7. Initialize the course average for group 2 to `0.0`.
8. Initialize the number of courses to `0`.
9. Print the heading.
10. Get the course ID, `courseId1`, for group 1.
11. Get the course ID, `courseId2`, for group 2.
12. For each course in group 1 and group 2,

    a.
    ```
    if (courseId1 != courseId2)
    {
        cout << "Data error: Course IDs do not match.\n";
        return 1;
    }
    ```

```
    cin >> num;
    cout << endl;

    cout << "Take ";

    if (num == 1)
        func1();
    else if (num == 2)
        func2();
    else
        cout << "Invalid input. You must enter a 1 or 2" << endl;

    return 0;
}

void func1()
{
    cout << "Programming I." << endl;
}

void func2()
{
    cout << "Programming II." << endl;
}
```


a. What is the output if the input is 1?
b. What is the output if the input is 2?
c. What is the output if the input is 3?
d. What is the output if the input is -1?

5. Write the definition of a void function that takes as input a decimal number and as output 3 times the value of the decimal number. Format your output to two decimal places.
6. Write the definition of a void function that takes as input two decimal numbers. If the first number is nonzero, it outputs second number divided by the first number; otherwise, it outputs a message indicating that the second number cannot be divided by the first number because the first number is 0.
7. Write the definition of a `void` function with three reference parameters of type `int`, `double`, and `string`. The function sets the values of the `int` and `double` variables to `0` and the value of the `string` variable to the empty string.
8. Write the definition of a `void` function that takes as input two parameters of type `int`, say `sum` and `testScore`. The function updates the value of `sum` by adding the value of `testScore`. The new value of `sum` is reflected in the calling environment.
9. What is the output of the following program?

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

```
void find(int a, int& b, int& c,)

int main()
{
    int one, two, three;

    one = 5;
    two = 10;
    three = 15;

    find(one, two, three);
    cout << one << ", " << two << ", " << three << endl;

    find(two, one, three);
    cout << one << ", " << two << ", " << three << endl;

    find(three, two, one);
    cout << one << ", " << two << ", " << three << endl;

    find(two, three, one);
    cout << one << ", " << two << ", " << three << endl;

    return 0;
}

void find(int a, int& b, int& c)
{
    int temp;

    c = a + b;
    temp = a;
    a = b;
    b = 2 * temp;
}
```


10. What is the output of the following program?

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

int x;

void summer(int&, int);
void fall(int, int&);

int main()
{
    int intNum1 = 2;
    int intNum2 = 5;
    x = 6;

    summer(intNum1, intNum2);
    cout << intNum1 << " " << intNum2 << " " << x << endl;
```

```
    fall(intNum1, intNum2);
    cout << intNum1 << " " << intNum2 << " " << x << endl;
    return 0;
}

void summer(int& a, int b)
{
    int intNum1;
    intNum1 = b + 12;
    a = 2 * b + 5;
    b = intNum1 + 4;
}

void fall(int u, int& v)
{
    int intNum2;
    intNum2= x;
    v = intNum2 * 4;
    x = u - v;
}
```

11. In the following program, number the marked statements to show the order in which they will execute (the logical order of execution).

```
#include <iostream>

using namespace std;

void func(int val1, int val2);

int main()
{
      int num1, num2;
___   cout << "Please enter two integers." << endl;
___   cin >> num1 >> num2;
___   func (num1, num2);
___   cout << " The two integers are " << num1
           << ", " << num2 << endl;
___   return 0;
}
void func(int val1, int val2)
{
      int val3, val4;
___   val3 = val1 + val2;
___   val4 = val1 * val2;
___   cout << "The sum and product are " << val3
           << " and " << val4 << endl;
}
```

12. Consider the following program:

```
#include <iostream>
#include <cmath>
#include <iomanip>
```

Write a program that prompts the user to enter:

a. The width of the river
b. The distance of the factory downstream on the other side of the river
c. The cost of laying the power line under water
d. The cost of laying the power line over land

The program then outputs the length of the power line that should run under water and the length that should run over land so the cost of constructing the power line is at the minimum. The program should also output the total cost of constructing the power line.

16. (**Pipe problem, requires trigonometry**) A pipe is to be carried around the right-angled corner of two intersecting corridors. Suppose that the widths of the two intersecting corridors are 5 feet and 8 feet (see Figure 7-22). Your objective is to find the length of the longest pipe, rounded to the nearest foot, that can be carried level around the right-angled corner.

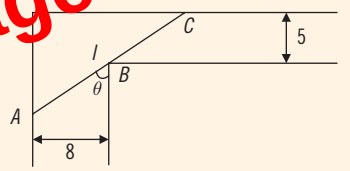


FIGURE 7-22 Pipe problem

Write a program that prompts the user to input the widths of both of the hallways. The program then outputs the length of the longest pipe, rounded to the nearest foot, that can be carried level around the right-angled corner. (Note that the length of the pipe is given by $l = AB + BC = 8 / \sin\theta + 5 / \cos\theta$, where $0 < \theta < \pi/2$.)

These are illegal enumeration types because none of the values is an identifier. The following, however, are legal enumeration types:

```
enum grades {A, B, C, D, F};
enum places {FIRST, SECOND, THIRD, FOURTH};
```

If a value has already been used in one enumeration type, it cannot be used by any other enumeration type in the same block. The same rules apply to enumeration types declared outside of any blocks. Example 8-4 illustrates this concept.

EXAMPLE 8-4

Consider the following statements:

```
enum mathStudent {JOHN, BILL, CINDY, LISA, RON};
enum compStudent {SUSAN, CATHY, JOHN, WILLIAM}; //illegal
```

Suppose that these statements are in the same program in the same block. The second enumeration type, `compStudent`, is not allowed because the value `JOHN` was used in the previous enumeration type, `mathStudent`.

Declaring Variables

Once a data type is defined, you can declare variables of that type. The syntax for declaring variables of an `enum` type is the same as before:

```
dataType identifier, identifier,...;
```

The statement:

```
enum sports {BASKETBALL, FOOTBALL, HOCKEY, BASEBALL, SOCCER,
             VOLLEYBALL};
```

defines an enumeration type called `sports`. The statement:

```
sports popularSport, mySport;
```

declares `popularSport` and `mySport` to be variables of type `sports`.

Assignment

Once a variable is declared, you can store values in it. Assuming the previous declaration, the statement:

```
popularSport = FOOTBALL;
```

```
typedef int Boolean;          //Line 1
const Boolean TRUE = 1;       //Line 2
const Boolean FALSE = 0;      //Line 3
Boolean flag;                 //Line 4
```

The statement in Line 1 creates an alias, `Boolean`, for the data type `int`. The statements in Lines 2 and 3 declare the named constants `TRUE` and `FALSE` and initialize them to `1` and `0`, respectively. The statement in Line 4 declares `flag` to be a variable of type `Boolean`. Because `flag` is a variable of type `Boolean`, the following statement is legal:

```
flag = TRUE;
```


PROGRAMMING EXAMPLE: The Game of Rock, Paper, and Scissors

Children often play the game of rock, paper, and scissors. This game has two players, each of whom chooses one of the three objects: rock, paper, or scissors. If player 1 chooses rock and player 2 chooses paper, player 2 wins the game because paper covers the rock. The game is played according to the following rules:

- If both players choose the same object, this play is a tie.
- If one player chooses rock and the other chooses scissors, the player choosing the rock wins this play because the rock breaks the scissors.
- If one player chooses rock and the other chooses paper, the player choosing the paper wins this play because the paper covers the rock.
- If one player chooses scissors and the other chooses paper, the player choosing the scissors wins this play because the scissors cut the paper.

Write an interactive program that allows two people to play this game.

Input This program has two types of input:

- The users' responses when asked to play the game.
- The players' choices.

Output The players' choices and the winner of each play. After the game is over, the total number of plays and the number of times that each player won should be output as well.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Two players play this game. Players enter their choices via the keyboard. Each player enters `R` or `r` for Rock, `P` or `p` for Paper, or `S` or `s` for Scissors. While the first player enters a choice, the second player looks elsewhere. Once both entries are in, if the entries are valid, the program outputs the players' choices and declares the winner of the play. The game continues until one of the players decides to quit

the game. After the game ends, the program outputs the total number of plays and the number of times that each player won. This discussion translates into the following algorithm:

1. Provide a brief explanation of the game and how it is played.
2. Ask the users if they want to play the game.
3. Get plays for both players.
4. If the plays are valid, output the plays and the winner.
5. Update the total game count and winner count.
6. Repeat Steps 2 through 5 while the users agree to play the game.
7. Output the number of plays and times that each player won.

We will use the enumeration type to describe the objects:

```
enum objectType {ROCK, PAPER, SCISSORS};
```

Variables (Function `main`)

It is clear that you need the following variables in the function `main`:

```
int gameCount; //variable to store the number of
               //games played
int winCount1; //variable to store the number of games
               //won by player 1
int winCount2; //variable to store the number of games
               //won by player 2
int gamewinner;
char response;  //variable to get the user's response to
                //play the game
char selection1;
char selection2;
objectType play1;  //player1's selection
objectType play2;  //player2's selection
```

This program is divided into the following functions, which the ensuing sections describe in detail.

- **displayRules**: This function displays some brief information about the game and its rules.
- **validSelection**: This function checks whether a player's selection is valid. The only valid selections are `R`, `r`, `P`, `p`, `S`, and `s`.
- **retrievePlay**: Because enumeration types cannot be read directly, this function converts the entered choice (`R`, `r`, `P`, `p`, `S`, or `s`) and returns the appropriate object type.
- **gameResult**: This function outputs the players' choices and the winner of the game.

```
b. else
   {
      1. Determine the winning object. (Call function winningObject)
      2. Output each player's choice.
      3. Determine the winning player.
      4. Return the winning player via a reference parameter to the
         function main so that the function main can update the
         winning player's win count.
   }
```

The definition of this function is:

```
void gameResult(objectType play1, objectType play2,
                int& winner)
{
    objectType winnerObject;

    if (play1 == play2)
    {
        winner = 0;
        cout << "Both players selected ";
        convertEnum(play1);
        cout << ". This game is a tie." << endl;
    }
    else
    {
        winnerObject = winningObject(play1, play2);

            //Output each player's choice
        cout << "Player 1 selected ";
        convertEnum(play1);
        cout << " and player 2 selected ";
        convertEnum(play2);
        cout << ". ";

            //Decide the winner
        if (play1 == winnerObject)
            winner = 1;
        else if (play2 == winnerObject)
            winner = 2;

            //Output the winner
        cout << "Player " << winner << " wins this game."
             << endl;
    }
}
```


Function convertEnum

Because enumeration types cannot be output directly, let's write the function `convertEnum` to output objects of the `enum` type `objectType`. This function has one parameter, of type `objectType`. It outputs the string that corresponds to the `objectType`. In pseudocode, this function is:

PROGRAM LISTING

```
//***************************************************************
// Author: D.S. Malik
//
// Program: Rock, Paper, and Scissors
// This program plays the game of rock, paper, and scissors.
//***************************************************************

#include <iostream>

using namespace std;

enum objectType {ROCK, PAPER, SCISSORS};

     //Function prototypes
void displayRules();
objectType retrievePlay(char selection);
bool validSelection(char selection);
void convertToString(objectType object);
objectType winningObject(objectType play1, objectType play2);
void gameResult(objectType play1, objectType play2, int& winner);
void displayResults(int gCount, int wCount1, int wCount2);

int main()
{
        //Step 1
    int gameCount; //variable to store the number of
                   //games played
    int winCount1; //variable to store the number of games
                   //won by player 1
    int winCount2; //variable to store the number of games
                   //won by player 2
    int gamewinner;
    char response;  //variable to get the user's response to
                    //play the game
    char selection1;
    char selection2;
    objectType play1;  //player1's selection
    objectType play2;  //player2's selection

        //Initialize variables; Step 2
    gameCount = 0;
    winCount1 = 0;
    winCount2 = 0;

    displayRules();                                  //Step 3

    cout << "Enter Y/y to play the game: ";          //Step 4
    cin >> response;                                 //Step 5
    cout << endl;
```

```
    y = std::pow(x, 2);
    .
    .
    .
}
```

This example accesses the function **pow** of the header file **cmath**.

EXAMPLE 8-11

Consider the following C++ code:

```
#include <iostream>
.
.
.
int main()
{
    using namespace std;
    .
    .
}
.
.
.
```


In this example, the function **main** can refer to the global identifiers of the header file **iostream** without using the prefix **std::** before the identifier name. The **using** statement appears inside the function **main**. Therefore, other functions (if any) should use the prefix **std::** before the name of the global identifier of the header file **iostream** unless the function has a similar **using** statement.

EXAMPLE 8-12

Consider the following C++ code:

```
#include <iostream>

using namespace std;          //Line 1

int t;                        //Line 2
double u;                     //Line 3

namespace expN
{
    int x;                    //Line 4
    char t;                   //Line 5
```

```
    double u;                   //Line 6
    void printResult();         //Line 7
}

using namespace expN;

int main()
{
    int one;                    //Line 8
    double t;                   //Line 9
    double three;               //Line 10

    .
    .
    .
}

void expN::printResult() //Definition of the function printResult
{
    .
    .
    .
}
```


In this C++ program:

1. To refer to the variable `t` in Line 2 in `main`, use the *scope resolution operator*, which is `::` (that is, refer to `t` as `::t`), because the function `main` has a variable named `t` (declared in Line 9). For example, to copy the value of `x` into `t`, you can use the statement `::t = x;`.
2. To refer to the member `t` (declared in Line 5) of the `namespace` `expN` in `main`, use the prefix `expN::` with `t` (that is, refer to `t` as `expN::t`) because there is a global variable named `t` (declared in Line 2) and a variable named `t` in `main`.
3. To refer to the member `u` (declared in Line 6) of the `namespace` `expN` in `main`, use the prefix `expN::` with `u` (that is, refer to `u` as `expN::u`) because there is a global variable named `u` (declared in Line 3).
4. You can reference the member `x` (declared in Line 4) of the `namespace` `expN` in `main` as either `x` or `expN::x` because there is no global identifier named `x` and the function `main` does not contain any identifier named `x`.
5. The definition of a function that is a member of a `namespace`, such as `printResult`, is usually written outside the `namespace` as in the preceding program. To write the definition of the function `printResult`, the name of the function in the function heading can be either `printResult` or `expN::printResult` (because no other global identifier is named `printResult`).

TABLE 8-1 Some `string` functions

Expression	Effect
`strVar.at(index)`	Returns the element at the position specified by `index`.
`strVar[index]`	Returns the element at the position specified by `index`.
`strVar.append(n, ch)`	Appends `n` copies of `ch` to `strVar`, in which `ch` is a `char` variable or a `char` constant.
`strVar.append(str)`	Appends `str` to `strVar`.
`strVar.clear()`	Deletes all the characters in `strVar`.
`strVar.compare(str)`	Compares `strVar` and `str`. (This operation is discussed in Chapter 4.)
`strVar.empty()`	Returns `true` if `strVar` is empty; otherwise, it returns `false`.
`strVar.erase()`	Deletes all the characters in `strVar`.
`strVar.erase(pos, n)`	Deletes `n` characters from `strVar` starting at position `pos`.
`strVar.find(str)`	Returns the index of the first occurrence of `str` in `strVar`. If `str` is not found, the special value `string::npos` is returned.
`strVar.find(str, pos)`	Returns the index of the first occurrence at or after `pos` where `str` is found in `strVar`.
`strVar.find_first_of (str, pos)`	Returns the index of the first occurrence of any character of `strVar` in `str`. The search starts at `pos`.
`strVar.find_first_not_of (str, pos)`	Returns the index of the first occurrence of any character of `str` not in `strVar`. The search starts at `pos`.
`strVar.insert(pos, n, ch);`	Inserts `n` occurrences of the character `ch` at index `pos` into `strVar`; `pos` and `n` are of type `string::size_type`; `ch` is a character.
`strVar.insert(pos, str);`	Inserts all the characters of `str` at index `pos` into `strVar`.
`strVar.length()`	Returns a value of type `string::size_type` giving the number of characters `strVar`.

```
    cout << endl;

    cout << "The pig Latin form of " << str << " is: "
         << pigLatinString(str) << endl;

    return 0;
}

//Place the definitions of the functions isVowel, rotate, and
//pigLatinString and as described previously here.
```

Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1:

```
Enter a string: eye

The pig Latin form of eye is: eye-way
```

Sample Run 2:

```
Enter a string: There

The pig Latin form of There is: ere-Thay
```

Sample Run 3:

```
Enter a string: why

The pig Latin form of why is: y-whay
```

Sample Run 4:

```
Enter a string: 123456

The pig Latin form of 123456 is: 123456-way
```


QUICK REVIEW

1. An enumeration type is a set of ordered values.
2. C++'s reserved word `enum` is used to create an enumeration type.
3. The syntax of `enum` is:

   ```
   enum typeName {value1, value2,...};
   ```

   in which `value1`, `value2`,... are identifiers, and `value1 < value2 < ...`.
4. No arithmetic operations are allowed on the enumeration type.

This page intentionally left blank

In previous chapters, you worked with simple data types. In Chapter 2, you learned that C++ data types fall into three categories. One of these categories is the structured data type. This chapter and the next few chapters focus on structured data types.

Recall that a data type is called **simple** if variables of that type can store only one value at a time. In contrast, in a **structured data type**, each data item is a collection of other data items. Simple data types are building blocks of structured data types. The first structured data type that we will discuss is an array. In Chapters 11 and 12, we will discuss other structured data types.

Before formally defining an array, let us consider the following problem. We want to write a C++ program that reads five numbers, finds their sum, and prints the numbers in reverse order.

In Chapter 5, you learned how to read numbers, print them, and find the sum. The difference here is that we want to print the numbers in reverse order. This means we cannot print the first four numbers until we have printed the fifth, and so on. To do this, we need to store all of the numbers before we start printing them in reverse order. From what we have learned so far, the following program accomplishes this task.


```
//Program to read five numbers, find their sum, and print the
//numbers in reverse order.

#include <iostream>

using namespace std;

int main()
{
    int item0, item1, item2, item3, item4;
    int sum;

    cout << "Enter five integers: ";
    cin >> item0 >> item1 >> item2 >> item3 >> item4;
    cout << endl;

    sum = item0 + item1 + item2 + item3 + item4;

    cout << "The sum of the numbers = " << sum << endl;
    cout << "The numbers in the reverse order are: ";
    cout << item4 << " " << item3 << " " << item2 << " "
         << item1 << " " << item0 << endl;

    return 0;
}
```

This program works fine. However, if you need to read 100 (or more) numbers and print them in reverse order, you would have to declare 100 variables and write many `cin` and `cout` statements. Thus, for large amounts of data, this type of program is not desirable.

Note the following in the previous program:

1. Five variables must be declared because the numbers are to be printed in reverse order.
2. All variables are of type `int`—that is, of the same data type.
3. The way in which these variables are declared indicates that the variables to store these numbers all have the same name—except the last character, which is a number.

Statement 1 tells you that you have to declare five variables. Statement 3 tells you that it would be convenient if you could somehow put the last character, which is a number, into a counter variable and use one `for` loop to count from `0` to `4` for reading and another `for` loop to count from `4` to `0` for printing. Finally, because all variables are of the same type, you should be able to specify how many variables must be declared—and their data type—with a simpler statement than the one we used earlier.

The data structure that lets you do all of these things in C++ is called an array.

Arrays

An **array** is a collection of a fixed number of components all of the same data type. A **one-dimensional array** is an array in which the components are arranged in a list form. This section discusses only one-dimensional arrays. Arrays of two dimensions or more are discussed later in this chapter.

The general form for declaring a one-dimensional array is:

```
dataType arrayName[intExp];
```

in which `intExp` is any constant expression that evaluates to a positive integer. Also, `intExp` specifies the number of components in the array.

EXAMPLE 9-1

The statement:

```
int num[5];
```

declares an array `num` of five components. Each component is of type `int`. The components are `num[0]`, `num[1]`, `num[2]`, `num[3]`, and `num[4]`. Figure 9-1 illustrates the array `num`.

The statement in Line 1 declares and initializes the array `myList`, and the statement in Line 2 declares the array `yourList`. Note that these arrays are of the same type and have the same number of components. Suppose that you want to copy the elements of `myList` into the corresponding elements of `yourList`. The following statement is illegal:

```
yourList = myList;  //illegal
```

In fact, this statement will generate a syntax error. C++ does not allow aggregate operations on an array. An **aggregate operation** on an array is any operation that manipulates the entire array as a single unit.

To copy one array into another array, you must copy it component-wise—that is, one component at a time. This can be done using a loop, such as the following:

```
for (int index = 0; index < 5; index ++)
     yourList[index] = myList[index];
```

Next, suppose that you want to read data into the array `yourList`. The following statement is illegal and, in fact, would generate a syntax error:

```
cin >> yourList; //illegal
```

To read data into `yourList`, you must read one component at a time, using a loop such as the following:

```
for (int index = 0; index < 5; index ++)
     cin >> yourList[index];
```

Similarly, determining whether two arrays have the same elements and printing the contents of an array must be done component-wise. Note that the following statements are illegal in the sense that they do not generate a syntax error; however, they do not give the desired results.

```
cout << yourList;

if (myList <= yourList)
.
.
.
```

We will comment on these statements in the section Base Address of an Array and Array in Computer Memory later in this chapter.

Arrays as Parameters to Functions

Now that you have seen how to work with arrays, a question naturally arises: How are arrays passed as parameters to functions?

By reference only: In C++, arrays are passed by reference only.

Because arrays are passed by reference only, you *do not* use the symbol `&` when declaring an array as a formal parameter.

```
        //Find and output the sum of the elements
        //of listA
    cout << "Line 14: The sum of the elements of "
         << "listA is: "
         << sumArray(listA, ARRAY_SIZE) << endl
         << endl;                                        //Line 14

        //Find and output the position of the largest
        //element in listA
    cout << "Line 15: The position of the largest "
         << "element in listA is: "
         << indexLargestElement(listA, ARRAY_SIZE)
         << endl;                                        //Line 15

        //Find and output the largest element
        //in listA
    cout << "Line 16: The largest element in "
         << "listA is: "
         << listA[indexLargestElement(listA, ARRAY_SIZE)]
         << endl << endl;                                //Line 16

        //Copy the elements of listA into listB using the
        //function copyArray
    copyArray(listA, listB, 0, ARRAY_SIZE);              //Line 17

    cout << "Line 18: After copying the elements "
         << "of listA into listB," << endl
         << "         listB elements are: ";             //Line 18

        //Output the elements of listB
    printArray(listB, ARRAY_SIZE);                       //Line 19
    cout << endl;                                        //Line 20

    return 0;
}

//Place the definitions of the functions initializeArray,
//fillArray, and so on here. Example 9-6 gives the definitions
//of these functions.
```

Sample Run: In this sample run, the user input is shaded.

```
Line 1: listA elements: 0 0 0 0 0 0 0 0 0 0
Line 5: ListB elements: 0 0 0 0 0 0 0 0 0 0

Line 8: Enter 10 integers: 33 77 25 63 56 48 98 39 5 12

Line 11: After filling listA, the elements are:
33 77 25 63 56 48 98 39 5 12

Line 14: The sum of the elements of listA is: 456
```

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

Print

By using a nested `for` loop, you can output the components of `matrix`. The following nested `for` loops print the components of `matrix`, one row per line:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cout << setw(5) << matrix[row][col] << " ";

    cout << endl;
}
```

Input

The following `for` loop inputs the data into row number 4; that is, the fifth row of `matrix`:

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    cin >> matrix[row][col];
```


As before, by putting the row number in a loop, you can input data into each component of `matrix`. The following `for` loop inputs data into each component of `matrix`:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cin >> matrix[row][col];
```

Sum by Row

The following `for` loop finds the sum of row number `4` of `matrix`; that is, it adds the components of row number `4`.

```
sum = 0;
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    sum = sum + matrix[row][col];
```

Once again, by putting the row number in a loop, we can find the sum of each row separately. Following is the C++ code to find the sum of each individual row:

```
  //Sum of each individual row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    sum = 0;
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        sum = sum + matrix[row][col];

    cout << "Sum of row " << row + 1 << " = " << sum << endl;
}
```

```
    if (length != length2)                                  //Step d
    {
        cout << "The original code and its copy "
             << "are not of the same length."
             << endl;
        return;
    }

    outfile << "Code Digit    Code Digit Copy"
            << endl;

    for (count = 0; count < length; count++)                //Step e
    {
        infile >> digit;                                    //Step e.1
        outfile << setw(5) << list[count]
                << setw(17) << digit;                       //Step e.2

        if (digit != list[count])                           //Step e.3
        {
            outfile << "  code digits are not the same"
                    << endl;
            codeOk = false;
        }
        else
            outfile << endl;
    }

    if (codeOk)                                             //Step f
        outfile << "Message transmitted OK."
                << endl;
    else
        outfile << "Error in transmission. "
                << "Retransmit!!" << endl;
}
```


Following is the algorithm for the function `main`.

Main Algorithm

1. Declare the variables.
2. Open the files.
3. Call the function `readCode` to read the secret code.
4. `if` (length of the secret code `<= 250`)

   Call the function `compareCode` to compare the codes.

   `else`

   Output an appropriate error message.

```
//*************************************************************
// Author: D.S. Malik
//
// Program: Check Code
// This program determines whether a code is transmitted
// correctly.
//*************************************************************

#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

const int MAX_CODE_SIZE = 250;

void readCode(ifstream& infile, int list[],
              int length, bool& lengthCodeOk);
void compareCode(ifstream& inFile, ofstream& outfile,
                 const int list[], int length);

int main()
{
        //Step 1
    int codeArray[MAX_CODE_SIZE]; //array to store the secret
                                  //code
    int codeLength;               //variable to store the
                                  //length of the secret code
    bool lengthCodeOk;  //variable to indicate if the length
                        //of the secret code is less than or
                        //equal to 250

    ifstream incode;    //input file stream variable
    ofstream outcode;   //output file stream variable

    char inputFile[51]; //variable to store the name of the
                        //input file
    char outputFile[51];    //variable to store the name of
                            //the output file

    cout << "Enter the input file name: ";
    cin >> inputFile;
    cout << endl;

        //Step 2
    incode.open(inputFile);
    if (!incode)
    {
        cout << "Cannot open the input file." << endl;
        return 1;
    }
```

PROGRAMMING EXAMPLE: Text Processing

(Line and letter count) Let us now write a program that reads a given text, outputs the text as is, and also prints the number of lines and the number of times each letter appears in the text. An uppercase letter and a lowercase letter are treated as being the same; that is, they are tallied together.

Because there are 26 letters, we use an array of `26` components to perform the letter count. We also need a variable to store the line count.

The text is stored in a file, which we will call `textin.txt`. The output will be stored in a file, which we will call `textout.out`.

Input A file containing the text to be processed.

Output A file containing the text, number of lines, and the number of times a letter appears in the text.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Based on the desired output, it is clear that we must output the text as is. That is, if the text contains any whitespace characters, they must be output as well. Furthermore, we must count the number of lines in the text. Therefore, we must know where the line ends, which means that we must trap the newline character. This requirement suggests that we cannot use the extraction operator to process the input file. Because we also need to perform the letter count, we use the `get` function to read the text.

Let us first describe the variables that are necessary to develop the program. This will simplify the discussion that follows.

Variables

We need to store the line count and the letter count. Therefore, we need a variable to store the line count and 26 variables to perform the letter count. We will use an array of `26` components to perform the letter count. We also need a variable to read and store each character in turn, because the input file is to be read character by character. Because data is to be read from an input file and output is to be saved in a file, we need an input stream variable to open the input file and an output stream variable to open the output file. These statements indicate that the function `main` needs (at least) the following variables:

```
int lineCount;          //variable to store the line count
int letterCount[26];    //array to store the letter count
char ch;                //variable to store a character
ifstream infile;        //input file stream variable
ofstream outfile;       //output file stream variable
```

In this declaration, `letterCount[0]` stores the `A` count, `letterCount[1]` stores the B count, and so on. Clearly, the variable `lineCount` and the array `letterCount` must be initialized to `0`.

4. Determine whether the following array declarations are valid. If a declaration is invaid, explain why.

   a. `int list75;`

   b.
   ```
   int size;
   double list[size];
   ```

   c. `int test[-10];`

   d. `double sales[40.5];`

5. What would be a valid range for the index of an array of size 50?

6. Write C++ statements to do the following:

   a. Declare an array `alpha` of 15 components of type `int`.

   b. Output the value of the tenth component of the array `alpha`.

   c. Set the value of the fifth component of the array `alpha` to 35.

   d. Set the value of the ninth component of the array `alpha` to the sum of the sixth and thirteenth components of the array `alpha`.

   e. Set the value of the fourth component of the array `alpha` to three times the value of the eighth component minus 57.

   f. Output `alpha` so that five components per line are printed.

7. What is the output of the following program segment?

```
int temp[5];

for (int i = 0; i < 5; i++)
    temp[i] = 2 * i - 3;

for (int i = 0; i < 5; i++)
    cout << temp[i] << " ";
cout << endl;

temp[0] = temp[4];
temp[4] = temp[1];
temp[2] = temp[3] + temp[0];

for (int i = 0; i < 5; i++)
    cout << temp[i] << " ";
cout << endl;
```

8. Suppose `list` is an array of five components of type `int`. What is stored in `list` after the following C++ code executes?

```
for (int i = 0; i < 5; i++)
{
    list[i] = 2 * i + 5;
    if (i % 2 == 0)
        list[i] = list[i] - 3;
}
```

29. Given the declaration:

```
char str1[21];
char str2[21];
```

a. Write a C++ statement that stores `"Sunny Day"` in `str1`.

b. Write a C++ statement that stores the length of `str1` into the `int` variable `length`.

c. Write a C++ statement that copies the value of `name` into `str2`.

d. Write C++ code that outputs `str1` if `str1` is less than or equal to `str2`, and otherwise outputs `str2`.

30. Assume the following declarations:

```
char name[21];
char yourName[21];
char studentName[31];
```

Mark the following statements as valid or invalid. If a statement is invalid, explain why.

a. `cin >> name;`

b. `cout << studentName;`

c. `yourName[0] = 'a';`

d. `yourName = studentName;`

e.
```
if (yourName == name)
    studentName = name;
```

f. `int x = strcmp(yourName, studentName);`

g. `strcpy(studentName, Name);`

h.
```
for (int j = 0; j < 21; j++)
    cout << name[j];
```

31. Define a two-dimensional array named `temp` of three rows and four columns of type `int` such that the first row is initialized to `6`, `8`, `12`, `9`; the second row is initialized to `17`, `5`, `10`, `6`; and the third row is initialized to `14`, `13`, `16`, `20`.

32. Suppose that array `temp` is as defined in Exercise 31. Write C++ statements to accomplish the following:

a. Output the contents of the first row and first column element of `temp`.

b. Output the contents of the first row and last column element of `temp`.

c. Output the contents of the last row and first column element of `temp`.

d. Output the contents of the last row and last column element of `temp`.

33. Consider the following declarations:

```
const int CAR_TYPES = 5;
const int COLOR_TYPES = 6;
```

at least, contain a function to read and store a number into an array and another function to output the sum of the numbers. (*Hint*: Read numbers as strings and store the digits of the number in the reverse order.)

12. Jason, Samantha, Ravi, Sheila, and Ankit are preparing for an upcoming marathon. Each day of the week, they run a certain number of miles and write them into a notebook. At the end of the week, they would like to know the number of miles run each day, the total miles for the week, and average miles run each day. Write a program to help them analyze their data. Your program must contain parallel arrays: an array to store the names of the runners and a two-dimensional array of five rows and seven columns to store the number of miles run by each runner each day. Furthermore, your program must contain at least the following functions: a function to read and store the runners' names and the numbers of miles run each day; a function to find the total miles run by each runner and the average number of miles run each day; and a function to output the results. (You may assume that the input data is stored in a file and each line of data is in the following form: `runnerName milesDay1 milesDay2 milesDay3 milesDay4 milesDay5 milesDay6 milesDay7`.)

13. Write a program to calculate students' average test scores and their grades. You may assume the following input data:

```
Johnson 85 83 77 91 76
Aniston 80 90 95 93 48
Cooper 78 81 11 90 73
Gupta 92 83 30 69 87
Blair 23 45 96 38 59
Clark 60 85 45 39 67
Kennedy 77 31 52 74 83
Bronson 93 94 89 77 97
Sunny 79 85 28 93 82
Smith 85 72 49 75 63
```

Use three arrays: a one-dimensional array to store the students' names, a (parallel) two-dimensional array to store the test scores, and a parallel one-dimensional array to store grades. Your program must contain at least the following functions: a function to read and store data into two arrays, a function to calculate the average test score and grade, and a function to output the results. Have your program also output the class average.

14. (**Airplane Seating Assignment**) Write a program that can be used to assign seats for a commercial airplane. The airplane has 13 rows, with six seats in each row. Rows 1 and 2 are first class, rows 3 through 7 are business class, and rows 8 through 13 are economy class. Your program must prompt the user to enter the following information:

    a. Ticket type (first class, business class, or economy class)
    b. Desired seat

Suppose that you have a list with 1000 elements. If the search item is the second item in the list, the sequential search makes two **key** (also called **item**) comparisons to determine whether the search item is in the list. Similarly, if the search item is the 900th item in the list, the sequential search makes 900 key comparisons to determine whether the search item is in the list. If the search item is not in the list, the sequential search makes 1000 key comparisons.

Therefore, if `searchItem` is always at the bottom of the list, it will take many comparisons to find it. Also, if `searchItem` is not in `list`, then we compare `searchItem` with every element in `list`. A sequential search is therefore not very efficient for large lists. In fact, it can be proved that, on average, the number of comparisons (key comparisons, not index comparisons) made by the sequential search is equal to half the size of the list. So, for a list size of 1000, on average, the sequential search makes about 500 key comparisons.

The sequential search algorithm does not assume that the list is sorted. If the list is sorted, then you can significantly improve the search algorithm as discussed in the section Binary Search of this chapter. However, first, we discuss how to sort a list.

Bubble Sort

There are many sorting algorithms. This section describes the sorting algorithm, called **bubble sort**, to sort a list.

Suppose `list[0]...list[n - 1]` is a list of `n` elements, indexed `0` to `n - 1`. We want to rearrange, that is, sort, the elements of `list` in increasing order. The bubble sort algorithm works as follows:

In a series of `n - 1` iterations, the successive elements `list[index]` and `list[index + 1]` of `list` are compared. If `list[index]` is greater than `list[index + 1]`, then the elements `list[index]` and `list[index + 1]` are swapped, that is, interchanged.

It follows that the smaller elements move toward the top (beginning), and the larger elements move toward the bottom (end) of the list.

In the first iteration, we consider `list[0]...list[n - 1]`; in the second iteration, we consider `list[0]...list[n - 2]`; in the third iteration, we consider `list[0]...list[n - 3]`, and so on. For example, consider `list[0]...list[4]`, as shown in Figure 10-1.

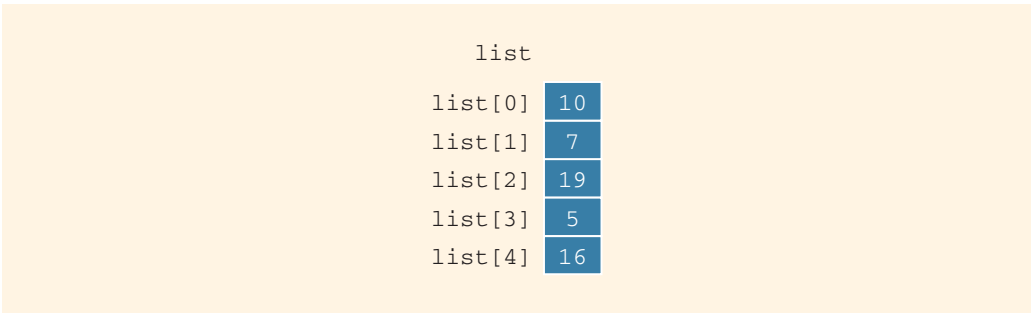


FIGURE 10-1 List of five elements

describes the sorting algorithm called insertion sort, which tries to improve—that is, reduce—the number of key comparisons.

The insertion sort algorithm sorts the list by moving each element to its proper place. Consider the list given in Figure 10-9.

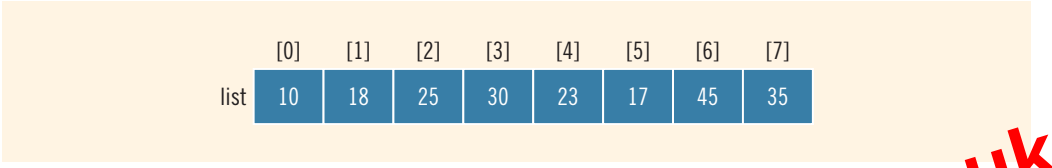


FIGURE 10-9 `list`

The length of the list is `8`. Moreover, the list elements `list[0]`, `list[1]`, `list[2]`, and `list[3]` are already in (ascending) order. That is, `list[0]..list[3]` is sorted (see Figure 10-10).

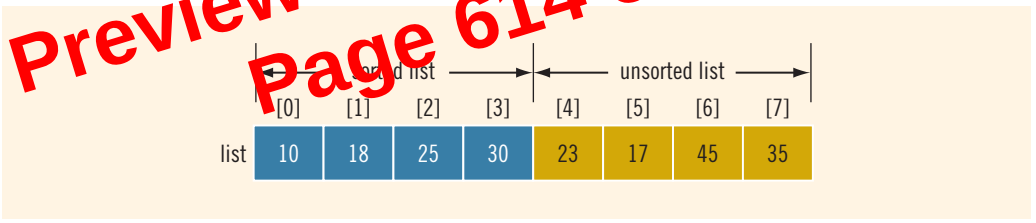


FIGURE 10-10 Sorted and unsorted portion of `list`

Next, we consider the element `list[4]`, the first element of the unsorted list. Because `list[4] < list[3]`, we need to move the element `list[4]` to its proper location. It thus follows that element `list[4]` should be moved to `list[2]` (see Figure 10-11).

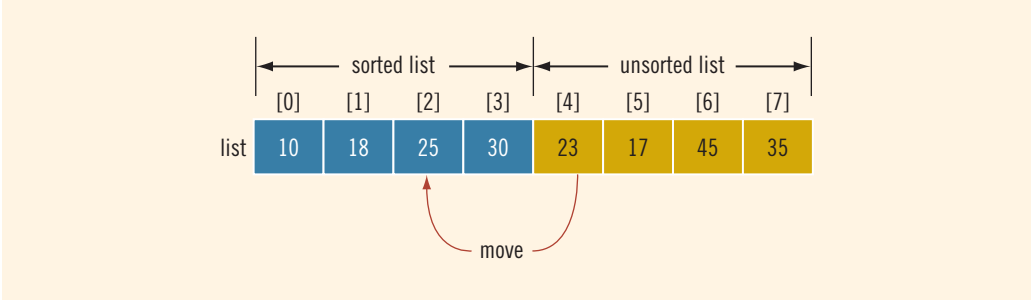


FIGURE 10-11 Move `list[4]` into `list[2]`

10

We now copy `temp` into `list[2]`. Figure 10-15 shows the resulting list.

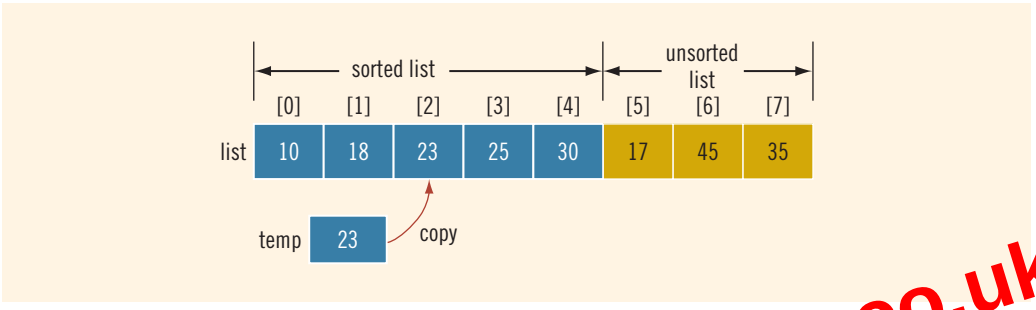


FIGURE 10-15 `list` after copying `temp` into `list[2]`

Now `list[0]...list[4]` is sorted, and `list[5]...list[7]` is unsorted. We repeat this process on the resulting list by moving the first element of the unsorted list into the proper place in the sorted list.

From this discussion, we see that during the sorting phase, the array containing the list is divided into two sublists, sorted and unsorted. Elements in the sorted sublist are sorted; elements in the unsorted sublist are to be moved to their proper places in the sorted sublist one at a time. We use an index—say, `firstOutOfOrder`—to point to the first element in the unsorted sublist. Initially, `firstOutOfOrder` is initialized to `1`.

This discussion translates into the following pseudocode:

```
for (firstOutOfOrder = 1; firstOutOfOrder < listLength;
                          firstOutOfOrder++)
  if (list[firstOutOfOrder] is less than list[firstOutOfOrder - 1])
  {
      copy list[firstOutOfOrder] into temp

      initialize location to firstOutOfOrder

      do
      {
         a. copy list[location - 1] into list[location]
         b. decrement location by 1 to consider the next element
            in the sorted portion of the array
      }
      while (location > 0 && the element in the upper list at
                             location - 1 is greater than temp)
  }
copy temp into list[location]
```

The following C++ function implements the previous algorithm:

```
void insertionSort(int list[], int listLength)
```

10

```
{
    int firstOutOfOrder, location;
    int temp;

    for (firstOutOfOrder = 1; firstOutOfOrder < listLength;
                              firstOutOfOrder++)
        if (list[firstOutOfOrder] < list[firstOutOfOrder - 1])
        {
            temp = list[firstOutOfOrder];
            location = firstOutOfOrder;

            do
            {
                list[location] = list[location - 1];
                location--;
            }
            while (location > 0 && list[location - 1] > temp);

            list[location] = temp;
        }
} //end insertionSort
```


We leave it as an exercise to write a program to test the insertion sort algorithm.

It is known that for a list of length n, on average, insertion sort makes about $\frac{n^2+3n-4}{4}$ key comparisons and about $\frac{n(n-1)}{4}$ item assignments. Therefore, if $n = 1000$, to sort the list, insertion sort makes about 250,000 key comparisons and about 250,000 item assignments.

This chapter presented three sorting algorithms. In fact, these are not the only sorting algorithms. You might be wondering why there are so many different sorting algorithms. The answer is that the performance of each sorting algorithm is different. Some algorithms make more comparisons, whereas others make fewer item assignments. Also, there are algorithms that make fewer comparisons, as well as fewer item assignments. The previous sections give the average number of comparisons and item assignments for the three sorting algorithms covered in this chapter. Analysis of the number of key comparisons and item assignments allows the user to decide which algorithm to use in a particular situation.

Binary Search

A sequential search is not very efficient for large lists. It typically searches about half of the list. However, if the list is sorted, you can use another search algorithm called **binary search**. A binary search is much faster than a sequential search. In order to apply a binary search, *the list must be sorted.*

Now that we know how to declare a **vector** object, let us discuss how to manipulate the data stored in a **vector** object. To do so, we must know the following basic operations:

- Item insertion
- Item deletion
- Stepping through the elements of a vector container

The type **vector** provides various operations to manipulate data stored in a vector object. Each of these operations is defined in the form of a function. Table 10-2 describes some of these functions and how to use them with a vector object. (Assume that **vecList** is a vector object. The name of the function is shown in **bold**.)

TABLE 10-2 Operations on a vector Object

Expression	Effect
vecList.**at**(index)	Returns the element at the position specified by **index**.
vecList[index]	Returns the element at the position specified by **index**.
vecList.**front**()	Returns the first element. (Does not check whether the object is empty.)
vecList.**back**()	Returns the last element. (Does not check whether the object is empty.)
vecList.**clear**()	Deletes all elements from the object.
vecList.**push_back**(elem)	A copy of **elem** is inserted into **vecList** at the end.
vecList.**pop_back**()	Delete the last element of **vecList**.
vecList.**empty**()	Returns **true** if the object **vecList** is empty and **false** otherwise.
vecList.**size**()	Returns the number of elements currently in the object **vecList**. The value returned is an **unsigned int** value.
vecList.**max_size**()	Returns the maximum number of elements that can be inserted into the object **vecList**.

EXERCISES

1. Mark the following statements as true or false.
   a. A sequential search of a list assumes that the list elements are sorted in ascending order.
   b. A binary search of a list assumes that the list is sorted.
   c. A binary search is faster on ordered lists and slower on unordered lists.
   d. A binary search is faster on large lists, but a sequential search is faster on small lists.
   e. When you declare a `vector` object and specify its size as `10`, then only 10 elements can be stored in the object.
2. Consider the following list:

   ```
   63 45 32 98 46 57 28 100
   ```

   Using a sequential search, how many comparisons are required to determine whether the following items are in the list or not? (Recall that comparisons mean item comparisons, not index comparisons.)

   a. 90 b. 57 c. 6 d. 120
3. a. Write a version of the sequential search algorithm that can be used to search a sorted list.
   b. Consider the following list:

   ```
   5 12 17 35 46 65 78 85 93 110 115
   ```

   Using a sequential search on ordered lists, which you designed in (a), how many comparisons are required to determine whether the following items are in the list or not? (Recall that comparisons mean item comparisons, not index comparisons.)

   i. 35 ii. 60 iii. 78 iv. 120
4. Consider the following list:

   ```
   2 10 17 45 49 55 68 85 92 98 110
   ```

   Using the binary search, how many comparisons are required to determine whether the following items are in the list or not? Show the values of `first`, `last`, and `middle` and the number of comparisons after each iteration of the loop.

   a. 15 b. 49 c. 98 d. 99
5. Sort the following list using the bubble sort algorithm as discussed in this chapter. Show the list after each iteration of the outer `for` loop.

   ```
   26, 45, 17, 65, 33, 55, 12, 18
   ```

21. Suppose that you have the following C++ code:

```
vector<int> myList(5);

unsigned int length;

myList[0] = 3;
for (int i = 1; i < 4; i++)
    myList[i] = 2 * myList[i - 1] - 5;

myList.push_back(46);
myList.push_back(57);
myList.push_back(35);
```

   a. Write a C++ statement that outputs the first and the last elements of myList. (Do not use the array subscripting operator or the index of the elements.)
   b. Write a C++ statement that stores the size of myList into length.
   c. Write a for loop that outputs the elements of myList.

22. What is the difference between the size and capacity of a vector?

PROGRAMMING EXERCISES

1. Write a program to test the function seqOrderedSearch. Use either the function bubbleSort or selectionSort to sort the list before the search.
2. Write a program to test the function binarySearch. Use either the function bubbleSort or selectionSort to sort the list before the search.
3. Write a function, remove, that takes three parameters: an array of integers, the number of elements in the array, and an integer (say, removeItem). The function should find and delete the first occurrence of removeItem in the array. If the value does not exist or the array is empty, output an appropriate message. (Note that after deleting the element, the number of elements in the array is reduced by 1.) Assume that the array is unsorted.
4. Write a function, removeAt, that takes three parameters: an array of integers, the number of elements in the array, and an integer (say, index). The function should delete the array element indicated by index. If index is out of range or the array is empty, output an appropriate message. (Note that after deleting the element, the number of elements in the array is reduced by 1.) Assume that the array is unsorted.
5. Write a function, removeAll, that takes three parameters: an array of integers, the number of elements in the array, and an integer (say, removeItem). The function should find and delete all of the occurrences of removeItem in the array. If the value does not exist or the array is empty, output an appropriate message. (Note that after deleting the element, the number of elements in the array is reduced.) Assume that the array is unsorted.

In C++, struct is a reserved word. The members of a struct, even though they are enclosed in braces (that is, they form a block), are not considered to form a compound statement. Thus, a semicolon (after the right brace) is essential to end the struct statement. A semicolon at the end of the struct definition is, therefore, a part of the syntax.

The statement:

```
struct employeeType
{
    string firstName;
    string lastName;
    string address1;
    string address2;
    double salary;
    string deptID;
};
```

defines a struct employeeType with six members. The members firstName, lastName, address1, address2, and deptID are of type string, and the member salary is of type double.

Like any type definition, a struct is a definition, not a declaration. That is, it defines only a data type; no memory is allocated.

Once a data type is defined, you can declare variables of that type. Let us first define a struct type, studentType, and then declare variables of that type.

```
struct studentType
{
    string firstName;
    string lastName;
    char courseGrade;
    int testScore;
    int programmingScore;
    double GPA;
};

    //variable declaration
studentType newStudent;
studentType student;
```

These statements declare two struct variables, newStudent and student, of type studentType. The memory allocated is large enough to store firstName, lastName, courseGrade, testScore, programmingScore, and GPA (see Figure 11-1).

The structVariableName.memberName is just like any other variable. For example, newStudent.courseGrade is a variable of type char, newStudent.firstName is a string variable, and so on. As a result, you can do just about anything with struct members that you normally do with variables. You can, for example, use them in assignment statements or input/output (where permitted) statements.

In C++, the dot (.) is an operator called the **member access operator**.

Suppose you want to initialize the member GPA of newStudent to 0.0. The following statement accomplishes this task:

```
newStudent.GPA = 0.0;
```

Similarly, the statements:

```
newStudent.firstName = "John";
newStudent.lastName = "Brown";
```

store "John" in the member firstName and "Brown" in the member lastName of newStudent.

After the preceding three assignment statements execute, newStudent is as shown in Figure 11-2.

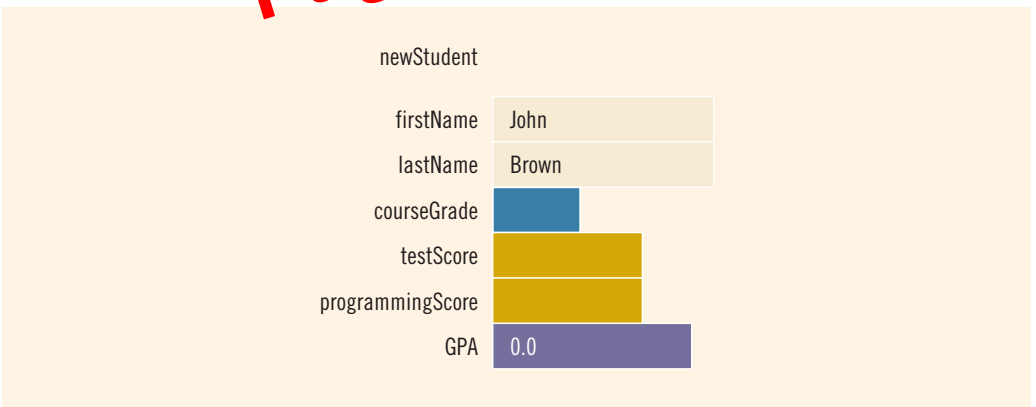


FIGURE 11-2 struct newStudent

The statement:

```
cin >> newStudent.firstName;
```

reads the next string from the standard input device and stores it in:

```
newStudent.firstName
```

The statement:

```
cin >> newStudent.testScore >> newStudent.programmingScore;
```

Suppose that a **struct** has several data members requiring a large amount of memory to store the data, and you need to pass a variable of that **struct** type by value. The corresponding formal parameter then receives a copy of the data of the variable. The compiler must then allocate memory for the formal parameter in order to copy the value of the actual parameter. This operation might require, in addition to a large amount of storage space, a considerable amount of computer time to copy the value of the actual parameter into the formal parameter.

On the other hand, if a variable is passed by reference, the formal parameter receives only the address of the actual parameter. Therefore, an efficient way to pass a variable as a parameter is by reference. If a variable is passed by reference, then when the formal parameter changes, the actual parameter also changes. Sometimes, however, you do not want the function to be able to change the values of the actual parameter. In C++, you can pass a variable by reference and still prevent the function from changing its value. This is done by using the keyword **const** in the formal parameter declaration, as shown in the definition of the function `seqSearch`.

Likewise, we can also rewrite the sorting, binary search, and other list-processing functions.

structs in Arrays

Suppose a company has 50 full-time employees. We need to print their monthly paychecks and keep track of how much money has been paid to each employee in the year-to-date. First, let's define an employee's record:

```
struct employeeType
{
    string firstName;
    string lastName;
    int    personID;
    string deptID;
    double yearlySalary;
    double monthlySalary;
    double yearToDatePaid;
    double monthlyBonus;
};
```

Each employee has the following members (components): first name, last name, personal ID, department ID, yearly salary, monthly salary, year-to-date paid, and monthly bonus.

Because we have 50 employees and the data type of each employee is the same, we can use an array of 50 components to process the employees' data.

```
employeeType employees[50];
```

This statement declares the array `employees` of `50` components of type `employeeType` (see Figure 11-7). Every element of `employees` is a **struct**. For example, Figure 11-7 also shows `employees[2]`.

middle name, as well as an address and a way to be contacted. You can, therefore, quickly put together a customer's record by using the `struct`s `nameType`, `addressType`, `contactType`, and the members specific to the customer.

Next, let us declare a variable of type `employeeType` and discuss how to access its members.

Consider the following statement:

```
employeeType newEmployee;
```

This statement declares `newEmployee` to be a `struct` variable of type `employeeType` (see Figure 11-8).

FIGURE 11-8 `struct` variable `newEmployee`

The next step is to process the sales data. Processing the sales data is quite straightforward. For each entry in the file containing the sales data:

1. Read the salesperson's ID, month, and sale amount for the month.
2. Search the array `salesPersonList` to locate the component corresponding to this salesperson.
3. Determine the quarter corresponding to the month.
4. Update the sales for the quarter by adding the sale amount for the month.

Once the sales data file is processed:

1. Calculate the total sales by salesperson.
2. Calculate the total sales by quarter.
3. Print the report.

This discussion translates into the following algorithm:

1. Initialize the array `salesPersonList`.
2. Process the sales data.
3. Calculate the total sales by quarter.
4. Calculate the total sales by salesperson.
5. Print the report.
6. Calculate and print the maximum sales by salesperson.
7. Calculate and print the maximum sales by quarter.

To reduce the complexity of the main program, let us write a separate function for each of these seven steps.

Function `initialize`

This function reads the salesperson's ID from the input file and stores the salesperson's ID in the array `salesPersonList`. It also initializes the quarterly sales amount and the total sales amount for each salesperson to `0`. The definition of this function is:

```
void initialize(ifstream& indata, salesPersonRec list[],
                int listSize)
{
    int index;
    int quarter;

    for (index = 0; index < listSize; index++)
    {
        indata >> list[index].ID; //get salesperson's ID

        for (quarter = 0; quarter < 4; quarter++)
            list[index].saleByQuarter[quarter] = 0.0;

        list[index].totalSale = 0.0;
    }
} //end initialize
```

Function getData

This function reads the sales data from the input file and stores the appropriate information in the array `salesPersonList`. The algorithm for this function is:

1. Read the salesperson's ID, month, and sales amount for the month.
2. Search the array `salesPersonList` to locate the component corresponding to the salesperson. (Because the salespeople's IDs are not sorted, we will use a sequential search to search the array.)
3. Determine the quarter corresponding to the month.
4. Update the sales for the quarter by adding the sales amount for the month.

Suppose that the entry read is:

```
57373 2 350
```

Here, the salesperson's ID is `57373`, the month is `2`, and the sales amount is `350`. Suppose that the array `salesPersonList` is as shown in Figure 11-11.

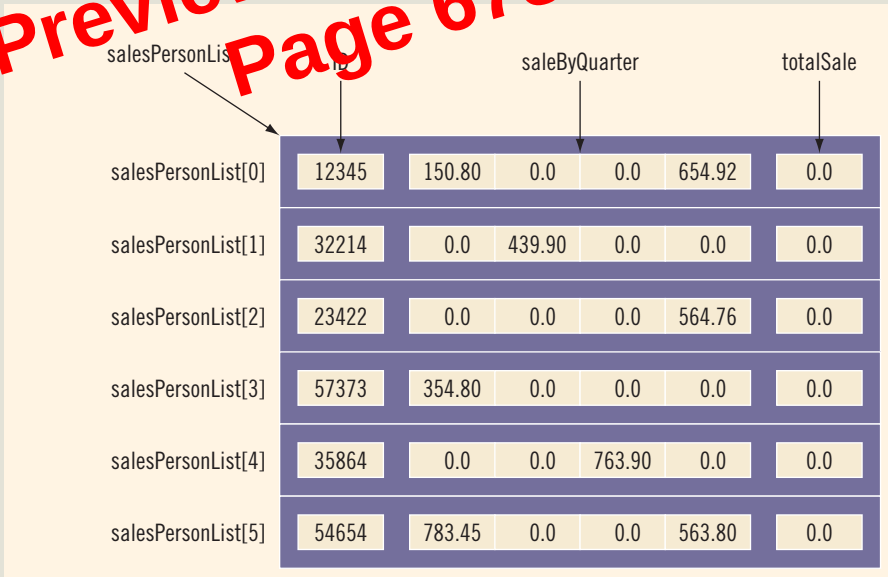


FIGURE 11-11 Array `salesPersonList`

Now, ID `57373` corresponds to the array component `salesPersonList[3]`, and month `2` corresponds to quarter `1`. Therefore, you add `350` to `354.80` to get the new amount, `704.80`. After processing this entry, the array `salesPersonList` is as shown in Figure 11-12.

7. Consider the following statements (`nameType` is as defined in Exercise 6):

```
struct employeeType
{
    nameType name;
    int performanceRating;
    int pID;
    string dept;
    double salary;
};
employeeType employees[100];
employeeType newEmployee;
```

Mark the following statements as valid or invalid. If a statement is invalid, explain why.

a. `newEmployee.name = "John Smith";`

b. `cout << newEmployee.name;`

c. `employees[35] = newEmployee;`

d. `if (employees[45].pID == 555334444)`
`    employees[45].performanceRating = 1;`

e. `employees.salary = 0;`

8. Assume the declarations of Exercises 6 and 7. Write C++ statements that do the following:

a. Store the following information in `newEmployee`:

```
name: Mickey Doe
pID:  111111111
performanceRating: 2
dept: ACCT
salary: 34567.78
```

b. In the array `employees`, initialize each `performanceRating` to `0`.

c. Copy the information of the 20th component of the array `employees` into `newEmployee`.

d. Update the salary of the 50th employee in the array `employees` by adding `5735.87` to its previous value.

9. Assume that you have the following definition of a `struct`.

```
struct partsType
{   string partName;
    int partNum;
    double price;
    int quantitiesInStock;
};
```

Declare an array, `inventory`, of 100 components of type `partsType`.

10. Assume the definition of Exercise 9.

    a. Write a C++ code to initialize each component of `inventory` as follows: `partName` to null string, `partNum` to `-1`, `price` to `0.0`, and `quantitiesInStock` to `0`.

    b. Write a C++ code that uses a loop to output the data stored in `inventory`. Assume that the variable `length` indicates the number of elements in `inventory`.

11. Assume the definition and declaration of Exercise 9. Write the definition of a void function that can be used to input data in a variable of type `partsType`. Also write a C++ code that uses your function to input data in `inventory`.

12. Suppose that you have the following definitions:

```
struct timeType
{
    int hr;
    double min;
    int sec;
};

struct tourType
{
    string cityName;
    int distance;
    timeType travelTime;
};
```

    a. Declare the variable `destination` of type `tourType`.

    b. Write C++ statements to store the following data in `destination`: `cityName`—Chicago, `distance`—550 miles, `travelTime`—9 hours and 30 minutes.

    c. Write the definition of a function to output the data stored in a variable of type `tourType`.

    d. Write the definition of a value-returning function that inputs data into a variable of type `tourType`.

    e. Write the definition of `void` function with a reference parameter of type `tourType` to input data in a variable of type `tourType`.

PROGRAMMING EXERCISES

1. Assume the definition of Exercise 4, which defines the `struct movieType`. Write a program that declares a variable of type `movieType`, prompts the user to input data about a movie, and outputs the movie data.

2. Write a program that reads students' names followed by their test scores. The program should output each student's name followed by the test scores and the relevant grade. It should also find and print the highest test score and the name of the students having the highest test score.

- Function `printCheck`: This function calculates and prints the check. (Note that the billing amount should include a 5% tax.)
  A sample output is:

```
Welcome to Johnny's Restaurant
Bacon and Egg     $2.45
Muffin            $0.99
Coffee            $0.50
Tax               $0.20
Amount Due        $4.14
```

Format your output with two decimal places. The name of each item in the output must be left justified. You may assume that the user selects only one item of a particular type.

5. Redo Exercise 4 so that the customer can select multiple items of a particular type. A sample output is this case is:

```
Welcome to Johnny's Restaurant
1 Bacon and Egg     $2.45
2 Muffin            $1.98
1 Coffee            $0.50
  Tax               $0.25
  Amount Due        $5.18
```


6. Write a program whose main function is merely a collection of variable declarations and function calls. This program reads a text and outputs the letters, together with their counts, as explained below in the function `printResult`. (There can be no global variables! All information must be passed in and out of the functions. Use a structure to store the information.) Your program must consist of at least the following functions:

   - Function `openFile`: Opens the input and output files. You must pass the file streams as parameters (by reference, of course). If the file does not exist, the program should print an appropriate message and exit. The program must ask the user for the names of the input and output files.
   - Function `count`: Counts every occurrence of capital letters `A-Z` and small letters `a-z` in the text file opened in the function `openFile`. This information must go into an array of structures. The array must be passed as a parameter, and the file identifier must also be passed as a parameter.
   - Function `printResult`: Prints the number of capital letters and small letters, as well as the percentage of capital letters for every letter `A-Z` and the percentage of small letters for every letter `a-z`. The percentages should look like this: "`25%`". This information must come from an array of structures, and this array must be passed as a parameter.

In Chapter 11, you learned how to group data items that are of different types by using a `struct`. The definition of a `struct` given in Chapter 11 is similar to the definition of a `C-struct`. However, the members of a C++ `struct` can be data items as well as functions. C++ provides another structured data type, called a **class**, which is specifically designed to group data and functions. This chapter first introduces classes and explains how to use them and then discusses the similarities and differences between a `struct` and a `class`.

NOTE

Chapter 11 is not a prerequisite for this chapter. In fact, a `struct` and a `class` have similar capabilities, as discussed in the section "A `struct` versus a `class`" in this chapter.

Classes

Chapter 1 introduced the problem-solving methodology called **object-oriented design (OOD)**. In OOD, the first step is to identify the components, called **objects**. An object combines data and the operations on that data in a single unit. In C++, the mechanism that allows you to combine data and the operations on that data in a single unit is called a class. Now that you know how to store and manipulate data in computer memory and how to construct your own functions, you are ready to learn how objects are constructed. This and subsequent chapters develop and implement programs using OOD. This chapter first explains how to define a class and use it in a program.

A **class** is a collection of a fixed number of components. The components of a class are called the **members** of the class.

The general syntax for defining a class is:

```
class classIdentifier
{
    classMembersList
};
```

in which `classMembersList` consists of variable declarations and/or functions. That is, a member of a class can be either a variable (to store data) or a function.

- If a member of a class is a variable, you declare it just like any other variable. Also, in the definition of the class, you cannot initialize a variable when you declare it.
- If a member of a class is a function, you typically use the function prototype to declare that member.
- If a member of a class is a function, it can (directly) access any member of the class—member variables and member functions. That is, when you write the definition of a member function, you can directly access any member variable of the class without passing it as a parameter. The only obvious condition is that you must declare an identifier before you can use it.

Functions and Classes

The following rules describe the relationship between functions and classes:

- Class objects can be passed as parameters to functions and returned as function values.
- As parameters to functions, class objects can be passed either by value or by reference.
- If a class object is passed by value, the contents of the member variables of the actual parameter are copied into the corresponding member variables of the formal parameter.

Reference Parameters and Class Objects (Variables)

Recall that when a variable is passed by value, the formal parameter copies the value of the actual parameter. That is, memory space to copy the value of the actual parameter is allocated for the formal parameter. As a parameter, a class object can be passed by value.

Suppose that a class has several member variables requiring a large amount of memory to store data, and you need to pass a variable by value. The corresponding formal parameter then receives a copy of the data of the variable. That is, the compiler must allocate memory for the formal parameter, so as to copy the value of the member variables of the actual parameter. This operation might require, in addition to a large amount of storage space, a considerable amount of computer time to copy the value of the actual parameter into the formal parameter.

On the other hand, if a variable is passed by reference, the formal parameter receives only the address of the actual parameter. Therefore, an efficient way to pass a variable as a parameter is by reference. If a variable is passed by reference, then when the formal parameter changes, the actual parameter also changes. Sometimes, however, you do not want the function to be able to change the values of the member variables. In C++, you can pass a variable by reference and still prevent the function from changing its value by using the keyword `const` in the formal parameter declaration. As an example, consider the following function definition:

```
void testTime(const clockType& otherClock)
{
    clockType dClock;
    .
    .
    .
}
```

The function `testTime` contains a reference parameter, `otherClock`. The parameter `otherClock` is declared using the keyword `const`. Thus, in a call to the function `testTime`, the formal parameter `otherClock` receives the address of the actual parameter, but `otherClock` cannot modify the contents of the actual parameter. For example, after the following statement executes, the value of `myClock` will not be altered:

```
testTime(myClock);
```

Next, let us give the definitions of the other member functions of the class clockType. The definitions of these functions are simple and easy to follow:

```
void clockType::getTime(int& hours, int& minutes,
                        int& seconds) const
{
    hours = hr;
    minutes = min;
    seconds = sec;
}

void clockType::printTime() const
{
    if (hr < 10)
        cout << "0";
    cout << hr << ":";

    if (min < 10)
        cout << "0";
    cout << min << ":";

    if (sec < 10)
        cout << "0";
    cout << sec;
}

void clockType::incrementHours()
{
    hr++;
    if (hr > 23)
        hr = 0;
}

void clockType::incrementMinutes()
{
    min++;
    if (min > 59)
    {
        min = 0;
        incrementHours(); //increment hours
    }
}

void clockType::incrementSeconds()
{
    sec++;

    if (sec > 59)
    {
        sec = 0;
        incrementMinutes(); //increment minutes
    }
}
```

Accessor and Mutator Functions

Let us look at the member functions of the `class` `clockType`. The function `setTime` sets the values of the member variables to the values specified by the user. In other words, it alters or modifies the values of the member variables. Similarly, the functions `incrementSeconds`, `incrementMinutes`, and `incrementHours` also modify the member variables. On the other hand, functions such as `getTime`, `printTime`, and `equalTime` only *access* the values of the member variables. They *do not* modify the member variables. We can, therefore, categorize the member functions of the `class` `clockType` into two categories: member functions that modify the member variables and member functions that only access, and do not modify, the member variables.

This is typically true for any class. That is, every class has member functions that only access and do not modify the member variables, called accessor functions, and member functions that modify the member variables, called mutator functions.

Accessor function: A member function of a class that only accesses (that is, does not modify) the value(s) of the member variable(s).

Mutator function: A member function of a class that modifies the value(s) of the member variable(s).

Because an accessor function only accesses the values of the member variables, as a safeguard, we typically include the reserved word `const` at the end of the headings of these functions. Moreover, a constant member function of a class cannot modify the member variables of that class. For example, see the headings of the member functions `getTime`, `printTime`, and `equalTime` of the `class` `clockType`.

A member function of a class is called a **constant function** if its heading contains the reserved word `const` at the end. For example, the member functions `getTime`, `printTime`, and `equalTime` of the `class` `clockType` are constant functions. A constant member function of a class cannot modify the member variables of that class, so these are accessor functions. One thing that should be remembered about constant member functions is that a constant member function of a class can *only* call other constant member functions of that class. Therefore, you should be careful when you make a member function constant.

Example 12-2 shows how to use the `class` `clockType` in a program. Note that we have combined the definition of the class, the definition of the member functions, and the main function to create a complete program. Later in this chapter, you will learn how to separate the definition of the `class` `clockType`, the definitions of the member functions, and the main program, using three files.

EXAMPLE 12-2

```
//The program listing of the program that defines
//and uses the class clockType
```

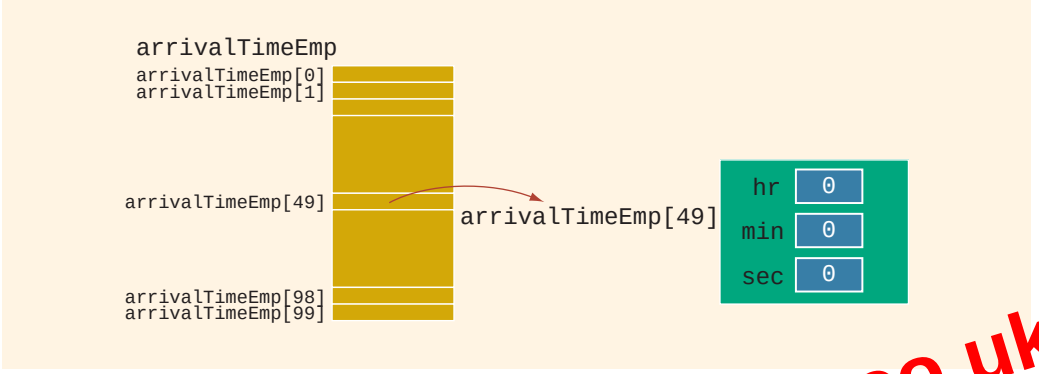


FIGURE 12-8 Array `arrivalTimeEmp`

You can now use the functions of the class `clockType` to manipulate the time for each employee. For example, the following statement sets the arrival time, that is, `hr`, `min`, and `sec`, of the 50th employee to 8, 5, and 10, respectively (see Figure 12-9).

```
arrivalTimeEmp[49].setTime(8, 5, 10);          //Line 2
```

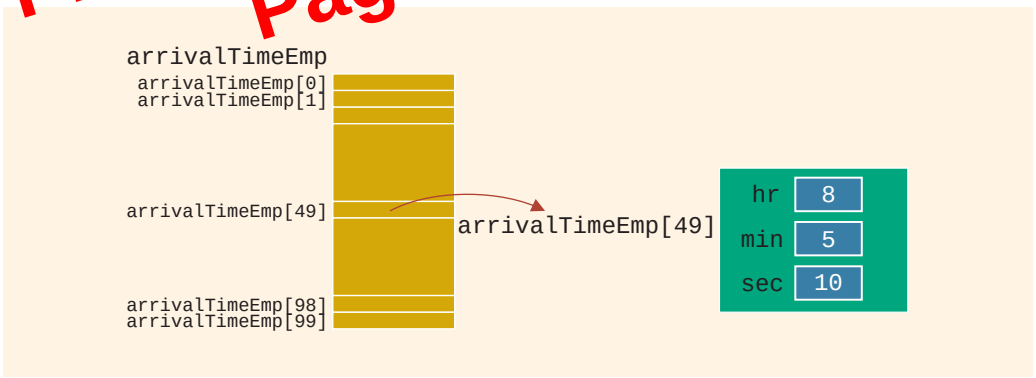



FIGURE 12-9 Array `arrivalTimeEmp` after setting the time of employee 49

To output the arrival time of each employee, you can use a loop, such as the following:

```
for (int j = 0; j < 100; j++)                  //Line 3
{
    cout << "Employee " << (j + 1)
         << " arrival time: ";
    arrivalTimeEmp[j].printTime();             //Line 4
    cout << endl;
}
```

The statement in Line 4 outputs the arrival time of an employee in the form `hr:min:sec`.

The following program shows how to use the `class` `die` in a program.

```
//The user program that uses the class die

#include <iostream>
#include "die.h"

using namespace std;

int main()
{                                                              //Line 1
    die die1;                                                  //Line 2
    die die2;                                                  //Line 3

    cout << "Line 4: die1: " << die1.getNum() << endl;         //Line 4

    cout << "Line 5: die2: " << die2.getNum() << endl;         //Line 5

    cout << "Line 6: After rolling die1: "
         << die1.roll() << endl;                               //Line 6

    cout << "Line 7: After rolling die2: "
         << die2.roll() << endl;                               //Line 7
    cout << "Line 8: The sum of the numbers rolled"
         << " by the dice is: "
         << die1.getNum() + die2.getNum() << endl;             //Line 8

    cout << "Line 9: After again rolling, the sum of "
         << "the numbers rolled is: "
         << die1.roll() + die2.roll() << endl;                 //Line 9

    return 0;                                                  //Line 10
}//end main                                                    //Line 11
```

Sample Run:

```
Line 4: die1: 1
Line 5: die2: 1
Line 6: After rolling die1: 3
Line 7: After rolling die2: 4
Line 8: The sum of the numbers rolled by the dice is: 7
Line 9: After again rolling, the sum of the numbers rolled is: 5
```

The preceding program works as follows. The statements in Lines 2 and 3 create the objects `die1` and `die2`, and, using the default constructor, set both the dice to 1. The statements in Lines 4 and 5 output the number of both the dice. The statement in Line 6 rolls `die1` and outputs the number rolled. Similarly, the statement in Line 7 rolls `die2` and outputs the number rolled. The statement in Line 8 outputs the sum of the numbers rolled by `die1` and `die2`. The statement in Line 9 again rolls both the dice and outputs the sum of the numbers rolled.

The output of the statement:

```
illusObject1.print();
```

is:

```
x = 3, y = 1, count = 1
```

Similarly, the output of the statement:

```
illusObject2.print();
```

is:

```
x = 5, y = 1, count = 1
```

Now consider the statement:

```
illustrate::count++;
```

After this statement executes, the objects and static members are as shown in Figure 12-14.

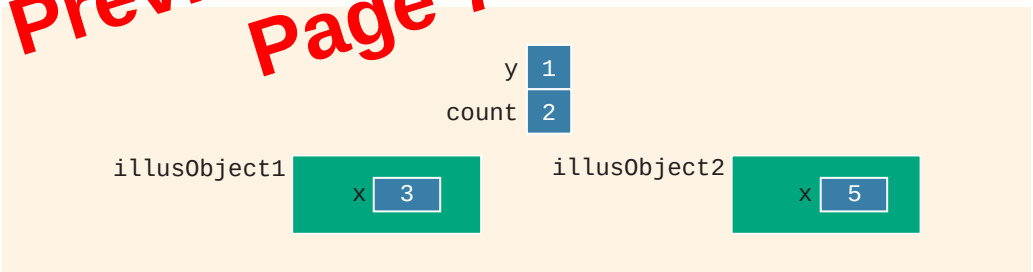


FIGURE 12-14 `illusObject1` and `illusObject2` after the statement `illustrate::count++;` executes

The output of the statements:

```
illusObject1.print();
illusObject2.print();
```

is:

```
x = 3, y = 1, count = 2
x = 5, y = 1, count = 2
```

The program in Example 12-11 further illustrates how **`static`** members of a class work.

definition of the class, in the heading of the definition of the constructor, we do not specify the default value. The definition of the constructor is as follows:

```
cashRegister::cashRegister(int cashIn)
{
    if (cashIn >= 0)
        cashOnHand = cashIn;
    else
        cashOnHand = 500;
}
```

Note that the definition of the constructor checks for valid values of the parameter `cashIn`. If the value of `cashIn` is less than `0`, the value assigned to the member variable `cashOnHand` is `500`.

Dispenser

The dispenser releases the selected item if it is not empty. It should show the number of items in the dispenser and the cost of the item. The following class defines the properties of a dispenser. Let us call this class `dispenserType`:

```
class dispenserType
{
public:
    int getNoOfItems() const;
      //Function to show the number of items in the machine.
      //Postcondition: The value of numberOfItems is returned.

    int getCost() const;
      //Function to show the cost of the item.
      //Postcondition: The value of cost is returned.

    void makeSale();
      //Function to reduce the number of items by 1.
      //Postcondition: numberOfItems--;

    dispenserType(int setNoOfItems = 50, int setCost = 50);
      //Constructor
      //Sets the cost and number of items in the dispenser
      //to the values specified by the user.
      //Postcondition: numberOfItems = setNoOfItems;
      //               cost = setCost;
      //               If no value is specified for a
      //               parameter, then its default value is
      //               assigned to the corresponding member
      //               variable.

private:
    int numberOfItems;   //variable to store the number of
                         //items in the dispenser
    int cost;  //variable to store the cost of an item
};
```

deposited by the customer, the cash register is updated by adding the money entered by the user.)

From this discussion, it is clear that the function `sellProduct` must have access to the dispenser holding the product (to decrement the number of items in the dispenser by `1` and to show the cost of the item) as well as the cash register (to update the cash). Therefore, this function has two parameters: one corresponding to the dispenser and the other corresponding to the cash register. Furthermore, both parameters must be referenced.

In pseudocode, the algorithm for this function is:

1. If the dispenser is not empty,
   a. Show and prompt the customer to enter the cost of the item.
   b. Get the amount entered by the customer.
   c. If the amount entered by the customer is less than the cost of the product,
      i. Show and prompt the customer to enter the additional amount.
      ii. Calculate the total amount entered by the customer.
   d. If the amount entered by the customer is at least the cost of the product,
      i. Update the amount in the cash register.
      ii. Sell the product—that is, decrement the number of items in the dispenser by `1`.
      iii. Display an appropriate message.
   e. If the amount entered by the user is less than the cost of the item, return the amount.
2. If the dispenser is empty, tell the user that this product is sold out.

This definition of the function `sellProduct` is:

```
void sellProduct(dispenserType& product,
                 cashRegister& pCounter)
{
    int amount;  //variable to hold the amount entered
    int amount2; //variable to hold the extra amount needed

    if (product.getNoOfItems() > 0) //if the dispenser is not
                                    //empty
    {
        cout << "Please deposit " << product.getCost()
             << " cents" << endl;
        cin >> amount;
```

```
myClass::incrementCount();
myObject1.printCount();
cout << endl;
myObject2.printCount();
cout << endl;
myObject2.printX();
cout << endl;
myObject1.setX(14);
myObject1.incrementCount();
myObject1.printX();
cout << endl;
myObject1.printCount();
cout << endl;
myObject2.printCount();
cout << endl;
```

14. In Example 12-8, we designed the `class die`. Using this class, declare an array named `rolls`, of 100 components of type `die`. Write C++ statements to roll each die of the array `rolls`, find and output the highest number rolled and the number of times this number was rolled, and find and output the number that is rolled the maximum number of times together with its count. Also write a program to test your statements.

PROGRAMMING EXERCISES

1. Write a program that converts a number entered in Roman numerals to decimal. Your program should consist of a `class`, say, `romanType`. An object of type `romanType` should do the following:

   a. Store the number as a Roman numeral.

   b. Convert and store the number into decimal form.

   c. Print the number as a Roman numeral or decimal number as requested by the user.

   The decimal values of the Roman numerals are:

M	1000
D	500
C	100
L	50
X	10
V	5
I	1

   d. Test your program using the following Roman numerals: `MCXIV`, `CCCLIX`, `MDCLXVI`.

   ii. Include the member functions to perform the various operations on objects of type `bookType`. For example, the usual operations that can be performed on the title are to show the title, set the title, and check whether a title is the same as the actual title of the book. Similarly, the typical operations that can be performed on the number of copies in stock are to show the number of copies in stock, set the number of copies in stock, update the number of copies in stock, and return the number of copies in stock. Add similar operations for the publisher, ISBN, book price, and authors. Add the appropriate constructors and a destructor (if one is needed).

   b. Write the definitions of the member functions of the `class bookType`.

   c. Write a program that uses the `class bookType` and tests various operations on the objects of the `class bookType`. Declare an array of 100 components of type `bookType`. Some of the operations that you should perform are to search for a book by its title, search by ISBN, and update the number of copies of a book.

7. In this exercise, you will design a `class memberType`.

   a. Each object of `memberType` can hold the name of a person, member ID, number of books bought, and amount spent.

   b. Include the member functions to perform the various operations on the objects of `memberType`—for example, modify, set, and show a person's name. Similarly, update, modify, and show the number of books bought and the amount spent.

   c. Add the appropriate constructors.

   d. Write the definitions of the member functions of `memberType`.

   e. Write a program to test various operations of your `class memberType`.

8. Using the classes designed in Programming Exercises 6 and 7, write a program to simulate a bookstore. The bookstore has two types of customers: those who are members of the bookstore and those who buy books from the bookstore only occasionally. Each member has to pay a $10 yearly membership fee and receives a 5% discount on each book purchased.

   For each member, the bookstore keeps track of the number of books purchased and the total amount spent. For every eleventh book that a member buys, the bookstore takes the average of the total amount of the last 10 books purchased, applies this amount as a discount, and then resets the total amount spent to `0`.

   Write a program that can process up to 1000 book titles and 500 members. Your program should contain a menu that gives the user different choices to effectively run the program; in other words, your program should be user driven.

9. The method `sellProduct` of the Candy Machine programming example gives the user only two chances to enter enough money to buy the product.

base classes. The derived classes inherit the properties of the base classes. So rather than create completely new classes from scratch, we can take advantage of inheritance and reduce software complexity.

Each derived class, in turn, becomes a base class for a future derived class. Inheritance can be either single inheritance or multiple inheritance. In **single inheritance**, the derived class is derived from a single base class; in **multiple inheritance**, the derived class is derived from more than one base class. This chapter concentrates on single inheritance.

Inheritance can be viewed as a tree-like, or hierarchical, structure wherein a base class is shown with its derived classes. Consider the tree diagram shown in Figure 13-1.

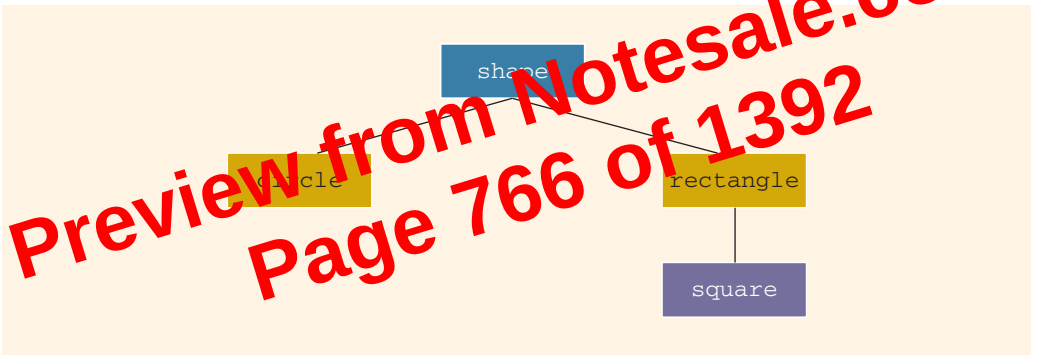


FIGURE 13-1 Inheritance hierarchy

In this diagram, `shape` is the base class. The `classes` `circle` and `rectangle` are derived from `shape`, and the `class` `square` is derived from `rectangle`. Every `circle` and every `rectangle` is a `shape`. Every `square` is a `rectangle`.

The general syntax of a derived class is:

```
class className: memberAccessSpecifier baseClassName
{
    member list
};
```

in which `memberAccessSpecifier` is `public`, `protected`, or `private`. When no `memberAccessSpecifier` is specified, it is assumed to be a `private` inheritance. (We discuss `protected` inheritance later in this chapter.)

points, B is equivalent to three points, C is equivalent to two points, D is equivalent to one point, and F is equivalent to zero points.

Input A file containing the data in the form given previously. For easy reference, let us assume that the name of the input file is stData.txt.

Output A file containing the output in the form given previously.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

We must first identify the main components of the program. The university has students, and every student takes courses. Thus, the two main components are the student and the course.

Let us first describe the course component.

Course

The main characteristics of a course are the course name, course number, and number of credit hours.

Some of the basic operations that need to be performed on an object of the course type are:

1. Set the course information.
2. Print the course information.
3. Show the credit hours.
4. Show the course number.

The following class defines the course as an ADT:

```
class courseType
{
public:
    void setCourseInfo(string cName, string cNo, int credits);
      //Function to set the course information.
      //The course information is set according to the
      //parameters.
      //Postcondition: courseName = cName; courseNo = cNo;
      //               courseCredits = credits;

    void print(ostream& outF);
      //Function to print the course information.
      //This function sends the course information to the
      //output device specified by the parameter outF. If the
      //actual parameter to this function is the object cout,
      //then the output is shown on the standard output device.
      //If the actual parameter is an ofstream variable, say,
      //outFile, then the output goes to the file specified by
      //outFile.

    int getCredits();
      //Function to return the credit hours.
      //Postcondition: The value of courseCredits is returned.
```

The definition of the function `print` is as follows:

```
void studentType::print(ostream& outF, double tuitionRate)
{
    int i;

    outF << "Student Name: " << getFirstName()
         << " " << getLastName() << endl;           //Step 1

    outF << "Student ID: " << sId << endl;          //Step 2

    outF << "Number of courses enrolled: "
         << numberOfCourses << endl;                //Step 3
    outF << endl;

    outF << left;
    outF << "Course No" << setw(10) << "Course Name"
         << setw(8) << "Credit"
         << setw(6) << "Grade" << endl;             //Step 4

    outF << right;
    for (i = 0; i < numberOfCourses; i++)           //Step 5
    {
        coursesEnrolled[i].print(outF);             //Step 5a

        if (isTuitionPaid)                          //Step 5b
            outF <<setw(4) << coursesGrade[i] << endl;
        else
            outF << setw(4) << "***" << endl;
    }
    outF << endl;

    outF << "Total number of credit hours: "
         << getHoursEnrolled() << endl;             //Step 6

    outF << fixed << showpoint << setprecision(2);  //Step 7

    if (isTuitionPaid)                              //Step 8
        outF << "Mid-Semester GPA: " << getGpa()
             << endl;
    else
    {
        outF << "*** Grades are being held for not paying "
             << "the tuition. ***" << endl;
        outF << "Amount Due: $" << billingAmount(tuitionRate)
             << endl;
    }

    outF << "-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*"
         << "-*-*-*-*-" << endl << endl;
} //end print
```

10. If in the heading of the definition of a derived class's constructor, no call to a constructor (with parameters) of a base class is specified, then during the derived class's object declaration and initialization, the default constructor (if any) of the base class executes.
11. When initializing the object of a derived class, the constructor of the base class is executed first.
12. Review the inheritance rules given in this chapter.
13. In composition (aggregation), a member of a class is an object of another class.
14. In composition (aggregation), a call to the constructor of the member objects is specified in the heading of the definition of the class's constructor.
15. The three basic principles of OOD are encapsulation, inheritance, and polymorphism.
16. An easy way to identify classes, objects, and operations is to describe the problem in English and then identify all the nouns and verbs. Choose your classes (objects) from the list of nouns and operations from the list of verbs.

EXERCISES

1. Mark the following statements as true or false.
   a. The constructor of a derived class can specify a call to the constructor of the base class in the heading of the function definition.
   b. The constructor of a derived class can specify a call to the constructor of the base class using the name of the class.
   c. Suppose that `x` and `y` are classes, one of the member variables of `x` is an object of type `y`, and both classes have constructors. The constructor of `x` specifies a call to the constructor of `y` by using the object name of type `y`.
2. Draw a class hierarchy in which several classes are derived from a single base class.
3. Suppose that a `class employeeType` is derived from the `class personType` (see Example 12-9 in Chapter 12). Give examples of members—data and functions—that can be added to the `class employeeType`.
4. Consider the following statements:

```
class dog: public animal
{
    ...
};
```

In this declaration, which class is the base class, and which class is the derived class?

check whether the date is valid before storing the date in the member variables. Rewrite the definitions of the function `setDate` and the constructor so that the values for the month, day, and year are checked before storing the date into the member variables. Add a member function, `isLeapYear`, to check whether a year is a leap year. Moreover, write a test program to test your class.

3. A point in the x-y plane is represented by its x-coordinate and y-coordinate. Design a `class`, `pointType`, that can store and process a point in the x-y plane. You should then perform operations on the point, such as setting the coordinates of the point, printing the coordinates of the point, returning the x-coordinate, and returning the y-coordinate. Also, write a program to test various operations on the point.

4. Every circle has a center and a radius. Given the radius, we can determine the circle's area and circumference. Given the center, we can determine its position in the x-y plane. The center of a circle is a point in the x-y plane. Design a `class`, `circleType`, that can store the radius and center of the circle. Because the center is a point in the x-y plane and you designed the class to capture the properties of a point in Programming Exercise 3, you must derive the `class` `circleType` from the `class` `pointType`. You should be able to perform the usual operations on the circle, such as setting the radius, printing the radius, calculating and printing the area and circumference, and carrying out the usual operations on the center. Also, write a program to test various operations on a circle.

5. Every cylinder has a base and height, wherein the base is a circle. Design a `class`, `cylinderType`, that can capture the properties of a cylinder and perform the usual operations on the cylinder. Derive this class from the `class` `circleType` designed in Programming Exercise 4. Some of the operations that can be performed on a cylinder are as follows: calculate and print the volume, calculate and print the surface area, set the height, set the radius of the base, and set the center of the base. Also, write a program to test various operations on a cylinder.

6. Using classes, design an online address book to keep track of the names, addresses, phone numbers, and dates of birth of family members, close friends, and certain business associates. Your program should be able to handle a maximum of 500 entries.

   a. Define a `class`, `addressType`, that can store a street address, city, state, and ZIP code. Use the appropriate functions to print and store the address. Also, use constructors to automatically initialize the member variables.

   b. Define a `class` `extPersonType` using the `class` `personType` (as defined in Example 12-9, Chapter 12), the `class` `dateType` (as designed in this chapter's Programming Exercise 2), and the `class` `addressType`. Add a member variable to this class to classify the person as a family

In Chapter 2, you learned that C++'s data types are classified into three categories: simple, structured, and pointers. Until now, you have studied only the first two data types. This chapter discusses the third data type called the pointer data type. You will first learn how to declare pointer variables (or pointers, for short) and manipulate the data to which they point. Later, you will use these concepts when you study dynamic arrays and linked lists. Linked lists are discussed in Chapter 18.

Pointer Data Type and Pointer Variables

Chapter 2 defined a data type as a set of values together with a set of operations. Recall that the set of values is called the domain of the data type. In addition to these two properties, until now, all of the data types you have encountered have one more thing associated with them: the name of the data type. For example, there is a data type called `int`. The set of values belonging to this data type includes integers that range between –2147483648 and 2147483647, and the operations allowed on these values are the arithmetic operators described in Chapter 2. To manipulate numeric integer data in the range –2147483648 to 2147483647, you can declare variables using the word `int`. The name of the data type allows you to declare a variable. Next, we describe the pointer data type.

The values belonging to pointer data types are the memory addresses of your computer. As in many other languages, there is no name associated with the pointer data type in C++. Because the domain—that is, the set of values of a pointer data type—is the addresses (memory locations), a pointer variable is a variable whose content is an address, that is, a memory location.

Pointer variable: A variable whose content is an address (that is, a memory address).

Declaring Pointer Variables

As remarked previously, there is no name associated with pointer data types. Moreover, pointer variables store memory addresses. So the obvious question is: If no name is associated with a pointer data type, how do you declare pointer variables?

The value of a pointer variable is an address. That is, the value refers to another memory space. The data is typically stored in this memory space. Therefore, when you declare a pointer variable, you also specify the data type of the value to be stored in the memory location pointed to by the pointer variable.

In C++, you declare a pointer variable by using the asterisk symbol (*) between the data type and the variable name. The general syntax to declare a pointer variable is:

```
dataType *identifier;
```

As an example, consider the following statements:

```
int *p;
char *ch;
```

the statement:

```
p = &x;
```

assigns the address of `x` to p. That is, `x` and the value of `p` refer to the same memory location.

Dereferencing Operator (*)

Every chapter until now has used the asterisk character, `*`, as the binary multiplication operator. C++ also uses `*` as a unary operator. When used as a unary operator, `*`, commonly referred to as the **dereferencing operator** or **indirection operator**, refers to the object to which its operand (that is, the pointer) points. For example, given the statements:

```
int x = 25;
int *p;
p = &x;    //store the address of x in p
```

the statement:

```
cout << *p << endl;
```

prints the value stored in the memory space pointed to by `p`, which is the value of `x`. Also, the statement:

```
*p = 55;
```

stores `55` in the memory location pointed to by `p`—that is, in `x`.

EXAMPLE 14-1

Let us consider the following statements:

```
int *p;
int num;
```

In these statements, `p` is a pointer variable of type `int`, and `num` is a variable of type `int`. Let us assume that memory location `1200` is allocated for `p`, and memory location `1800` is allocated for `num`. (See Figure 14-1.)

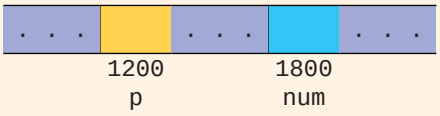


FIGURE 14-1 Variables `p` and `num`

Let us note the following:

1. `p` is a pointer variable.
2. The content of `p` points only to a memory location of type `int`.
3. Memory location `x` exists and is of type `int`. Therefore, the assignment statement:
   `p = &x;`
   is legal. After this assignment statement executes, `*p` is valid and meaningful.

The program in Example 14-3 further illustrates how a pointer variable works.

EXAMPLE 14-3

The following program illustrates how pointer variables work:


```
//Chapter 14: Example 14-3

#include <iostream>
using namespace std;

int main()
{
    int *p;
    int x = 37;

    cout << "Line 1: x = " << x << endl;                //Line 1

    p = &x;                                             //Line 2

    cout << "Line 3: *p = " << *p
         << ", x = " << x << endl;                      //Line 3

    *p = 58;                                            //Line 4

    cout << "Line 5: *p = " << *p
         << ", x = " << x << endl;                      //Line 5

    cout << "Line 6: Address of p = " << &p << endl;    //Line 6

    cout << "Line 7: Value of p = " << p << endl;       //Line 7

    cout << "Line 8: Value of the memory location "
         << "pointed to by *p = " << *p << endl;        //Line 8
    cout << "Line 9: Address of x = " << &x << endl;    //Line 9
    cout << "Line 10: Value of x = " << x << endl;      //Line 10

    return 0;
}
```

Before describing how to overcome this deficiency, let us describe one more situation that could also lead to a shallow copying of the data. The solution to both these problems is the same.

Recall that as parameters to a function, class objects can be passed either by reference or by value. Remember that the `class` `ptrMemberVarType` has the destructor, which deallocates the memory space pointed to by `p`. Suppose that `objectOne` is as shown in Figure 14-19.

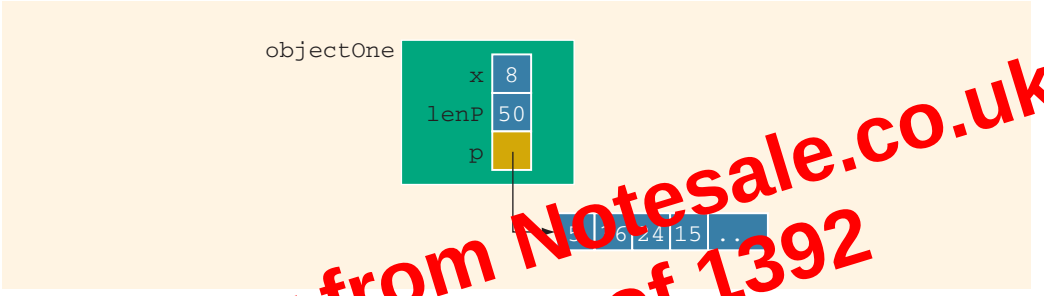


FIGURE 14-19 Object `objectOne`

Let us consider the following function prototype:

```
void destroyList(ptrMemberVarType paramObject);
```

The function `destroyList` has a formal value parameter, `paramObject`. Now consider the following statement:

```
destroyList(objectOne);
```

In this statement, `objectOne` is passed as a parameter to the function `destroyList`. Because `paramObject` is a value parameter, the copy constructor copies the member variables of `objectOne` into the corresponding member variables of `paramObject`. Just as in the previous case, `paramObject.p` and `objectOne.p` would point to the same memory space, as shown in Figure 14-20.

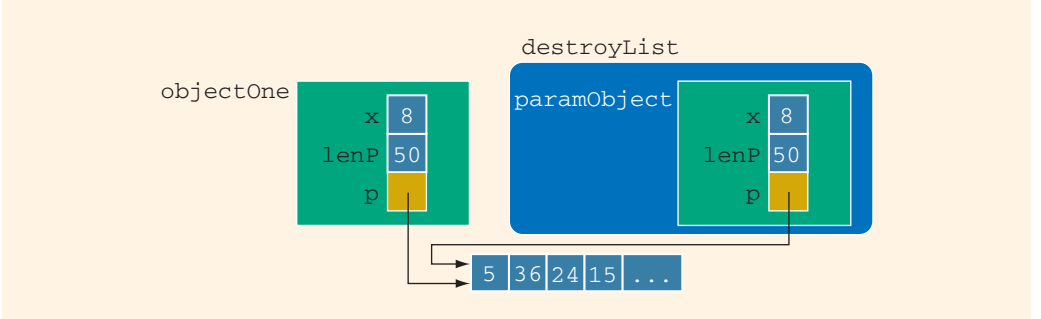


FIGURE 14-20 Pointer member variables of objects `objectOne` and `paramObject` pointing to the same array

When the function `destroyList` exits, the formal parameter `paramObject` goes out of scope, and the destructor for the object `paramObject` deallocates the memory space pointed to by `paramObject.p`. However, this deallocation has no effect on `objectOne`.

The general syntax to include the copy constructor in the definition of a class is:

```
className(const className& otherObject);
```

Notice that the formal parameter of the copy constructor is a constant reference parameter.

Example 14-7 illustrates how to include the copy constructor in a class and how it works.

EXAMPLE 14-7

Consider the following class:

```
class ptrMemberVarType
{
public:
    void print() const;
      //Function to output the data stored in the array p.

    void insertAt(int index, int num);
      //Function to insert num into the array p at the
      //position specified by index.
      //If index is out of bounds, the program is terminated.
      //If index is within bounds, but greater than the index
      //of the last item in the list, num is added at the end
      //of the list.

    ptrMemberVarType(int size = 10);
      //Constructor
      //Creates an array of the size specified by the
      //parameter size; the default array size is 10.

    ~ptrMemberVarType();
      //Destructor
      //deallocates the memory space occupied by the array p.

    ptrMemberVarType(const ptrMemberVarType& otherObject);
      //Copy constructor

private:
    int maxSize; //variable to store the maximum size of p
    int length;  //variable to store the number elements in p
    int *p;      //pointer to an int array
};
```

Suppose that the definitions of the members of the `class ptrMemberVarType` are as follows:

```
void ptrMemberVarType::print() const
{
    for (int i = 0; i < length; i++)
        cout << p[i] << " ";
}

void ptrMemberVarType::insertAt(int index, int num)
{
      //if index is out of bounds, terminate the program
    assert(index >= 0 && index < maxSize);

    if (index < length)
        p[index] = num;
    else
    {
        p[length] = num;
        length++;
    }
}

ptrMemberVarType::ptrMemberVarType(int size)
{
    if (size <= 0)
    {
        cout << "The array size must be positive." << endl;
        cout << "Creating an array of the size 10." << endl;

        maxSize = 10;
    }
    else
        maxSize = size;

    length = 0;

    p = new int[maxSize];

}

ptrMemberVarType::~ptrMemberVarType()
{
    delete [] p;
}

         //copy constructor
ptrMemberVarType::ptrMemberVarType
                  (const ptrMemberVarType& otherObject)
{
    maxSize = otherObject.maxSize;
    length = otherObject.length;
```

the default array size. The `for` loop in Line 5 reads and stores five integers in `listOne.p`. The statement in Line 9 outputs the numbers stored in `listOne`, that is, the five numbers stored in `p`. (See the output of the line marked Line 8 in the sample run.)

The statement in Line 11 declares `listTwo` to be an object of type `ptrMemberVarType` and also initializes `listTwo` using the values of `listOne`. The statement in Line 13 outputs the numbers stored in `listTwo`. (See the output of the line marked Line 12 in the sample run.)

The statements in Lines 15 and 16 modify `listTwo`, and the statement in Line 18 outputs the modified data of `listTwo`. (See the output of the line marked Line 17 in the sample run.) The statement in Line 21 outputs the data stored in `listOne`. Notice that the data stored in `listOne` is unchanged, even though `listTwo` modified its data. It follows that the copy constructor used to initialize `listTwo` using `listOne` (at Line 10) provides `listTwo` its own copy of the data.

The statements in Lines 23 through 28 show that when `listOne` is passed as a parameter by value to the function `testCopyConst` (see Line 24), the corresponding formal parameter `temp` has its own copy of data. Notice that the function `testCopyConst` modifies the object `temp`; however, the object `listOne` remains unchanged. See the outputs of the lines marked Line 24 (before the function `testCopyConst` is called) and Line 25 (after the function `testCopyConst` terminates) in the sample run. Also notice that when the function `testCopyConst` terminates, the destructor of the `class` `ptrMemberVarType` deallocates the memory space occupied by `temp.p`, which has no effect on `listOne.p`.

For classes with pointer member variables, three things are normally done:

1. Include the destructor in the class.
2. Overload the assignment operator for the class.
3. Include the copy constructor.

Chapter 15 discusses overloading the assignment operator. Until then, whenever we discuss classes with pointer member variables, out of the three items in the previous list, we will implement only the destructor and the copy constructor.

Inheritance, Pointers, and Virtual Functions

Recall that as a parameter, a class object can be passed either by value or by reference. Earlier chapters also said that the types of the actual and formal parameters must match. However, in the case of classes, C++ *allows the user to pass an object of a derived class to a formal parameter of the base class type.*

First, let us discuss the case in which the formal parameter is either a reference parameter or a pointer. To be specific, let us consider the following classes:

```
class petType
{
public:
```

```
class petType
{
public:
    virtual void print();       //virtual function
    petType(string n = "");

private:
    string name;
};

class dogType: public petType
{
public:
    void print();
    dogType(string n = "", string b = "");

private:
    string breed;
};
```

Note that we need to declare a virtual function only in the base class.

The definition of the member function print is the same as before. If we execute the previous program with these modifications, the output is as follows.

Sample Run:

```
Name: Lucky
Name: Tommy, Breed: German Shepherd
*** Calling the function callPrint ***
Name: Lucky
Name: Tommy, Breed: German Shepherd
```

This output shows that for the statement in Line 9, the `print` function of `dogType` is executed (see the last two lines of the output).

The previous discussion also applies when a formal parameter is a pointer to a class, and a pointer of the derived class is passed as an actual parameter. To illustrate this feature, suppose we have the preceding classes. (We assume that the definition of the `class` `petType` is in the header file `petType.h`, and the definition of the `class` `dogType` is in the header file `dogType.h`.) Consider the following program:

```
#include <iostream>
#include "petType.h"
#include "dogType.h"

using namespace std;

void callPrint(petType *p);

int main()
{
    petType *q;                                          //Line 1
    dogType *r;                                          //Line 2

    q = new petType("Lucky");                            //Line 3
    r = new dogType("Tommy", "German Shepherd");         //Line 4
```

```
private:
    double empSalary;
    double empBonus;
};
```

The definitions of the constructor and functions of the `class` `fullTimeEmployee` are:

```
void fullTimeEmployee::set(string first, string last,
                           long id,
                           double salary, double bonus)
{
    setName(first, last);
    setId(id);
    empSalary = salary;
    empBonus = bonus;
}

void fullTimeEmployee::setSalary(double salary)
{
    empSalary = salary;
}

double fullTimeEmployee::getSalary()
{
    return empSalary;
}

void fullTimeEmployee::setBonus(double bonus)
{
    empBonus = bonus;
}

double fullTimeEmployee::getBonus()
{
    return empBonus;
}

void fullTimeEmployee::print() const
{
    cout << "Id: " << getId() << endl;
    cout << "Name: ";
    personType::print();
    cout << endl;
    cout << "Wages: $" << calculatePay() << endl;
}

double fullTimeEmployee::calculatePay() const
{
    return empSalary + empBonus;
}
```

```
fullTimeEmployee::fullTimeEmployee(string first, string last,
                                   long id, double salary,
                                   double bonus)
                : employeeType(first, last, id)
{
    empSalary = salary;
    empBonus = bonus;
}
```

The definition of the `class` partTimeEmployee is:

```
#include "employeeType.h"

class partTimeEmployee: public employeeType
{
public:
    void set(string first, string last, long id, double rate,
             double hours);
      //Function to set the first name, last name, id,
      //payRate, and hoursWorked according to the
      //parameters.
      //Postcondition: firstName = first; lastName = last;
      //               personId = id;
      //               payRate = rate; hoursWorked = hours

    double calculatePay() const;
      //Function to calculate and return the wages.
      //Postcondition: Pay is calculated and returned.

    void setPayRate(double rate);
      //Function to set the salary.
      //Postcondition: payRate = rate;

    double getPayRate();
      //Function to retrieve the salary.
      //Postcondition: returns payRate;

    void setHoursWorked(double hours);
      //Function to set the bonus.
      //Postcondition: hoursWorked = hours

    double getHoursWorked();
      //Function to retrieve the bonus.
      //Postcondition: returns empBonus;

    void print() const;
      //Function to output the id, first name, last name,
      //and the wages.
      //Postcondition: Outputs
      //        Id:
      //        Name: firstName lastName
      //        Wages: $$$$.$$
```

16. What is the output of the following code?

```
int *secret;
int j;

secret = new int[10];
secret[0] = 10;
for (j = 1; j < 10; j++)
    secret[j] = secret[j - 1] + 5;
for (j = 0; j < 10; j++)
    cout << secret[j] << " ";
cout << endl;
```

17. Consider the following statement:

```
int *num;
```

a. Write the C++ statement that dynamically creates an array of 10 components of type `int` and `num` contains the base address of the array.

b. Write a C++ code that inputs data into the array `num` from the standard input device.

c. Write a C++ statement that deallocates the memory space of array to which `num` points.

18. Consider the following C++ code:

```
int *p;
p = new int[10];
for (int j = 0; j < 10; j++)
    p[i] = 2 * j - 2;
```

Write the C++ statement that deallocates the memory space occupied by the array to which `p` points.

19. Explain the difference between a shallow copy and a deep copy of data.

20. What is wrong with the following code?

```
int *p;                              //Line 1
int *q;                              //Line 2

p = new int[5];                      //Line 3
*p = 2;                              //Line 4

for (int i = 1; i < 5; i++)          //Line 5
     p[i] = p[i - 1] + i;            //Line 6

q = p;                               //Line 7

delete [] p;                         //Line 8

for (int j = 0; j < 5; j++)          //Line 9
     cout << q[j] << " ";            //Line 10

cout << endl;                        //Line 11
```

Rewrite the definition of the `class` `studentType` so that the functions `print` and `calculateGPA` are pure `virtual` functions.

31. Suppose that the definitions of the `class`es `employeeType`, `fullTimeEmployee`, and `partTimeEmployee` are as given in Example 14-8 of this chapter. Which of the following statements is legal?

    a. `employeeType tempEmp;`

    b. `fullTimeEmployee newEmp();`

    c. `partTimeEmployee pEmp("Molly", "Burton", 101, 0.0, 0);`

PROGRAMMING EXERCISES

1. Redo Programming Exercise 5 of Chapter 9 using dynamic arrays.
2. Redo Programming Exercise 6 of Chapter 9 using dynamic arrays.
3. Redo Programming Exercise 7 of Chapter 9 using dynamic arrays. You must ask the user for the number of candidates and then create the appropriate arrays to store the data.
4. Programming Exercise 11 in Chapter 9 explains how to add large integers using arrays. However, in that exercise, the program could add only integers of, at most, 20 digits. This chapter explains how to work with dynamic integers. Design a `class` named `largeIntegers` such that an object of this class can store an integer of any number of digits. Add operations to add, subtract, multiply, and compare integers stored in two objects. Also add constructors to properly initialize objects and functions to set, retrieve, and print the values of objects.
5. Banks offer various types of accounts, such as savings, checking, certificate of deposits, and money market, to attract customers as well as meet with their specific needs. Two of the most commonly used accounts are savings and checking. Each of these accounts has various options. For example, you may have a savings account that requires no minimum balance but has a lower interest rate. Similarly, you may have a checking account that limits the number of checks you may write. Another type of account that is used to save money for the long term is certificate of deposit (CD).

   In this programming exercise, you use abstract classes and pure virtual functions to design classes to manipulate various types of accounts. For simplicity, assume that the bank offers three types of accounts: savings, checking, and certificate of deposit, as described next.

   Savings accounts: Suppose that the bank offers two types of savings accounts: one that has no minimum balance and a lower interest rate and another that requires a minimum balance and has a higher interest rate.

   Checking accounts: Suppose that the bank offers three types of checking accounts: one with a monthly service charge, limited check writing, no

minimum balance, and no interest; another with no monthly service charge, a minimum balance requirement, unlimited check writing and lower interest; and a third with no monthly service charge, a higher minimum requirement, a higher interest rate, and unlimited check writing.

Certificate of deposit (CD): In an account of this type, money is left for some time, and these accounts draw higher interest rates than savings or checking accounts. Suppose that you purchase a CD for six months. Then we say that the CD will mature in six months. Penalty for early withdrawal is stiff.

Figure 14-22 shows the inheritance hierarchy of these bank accounts.

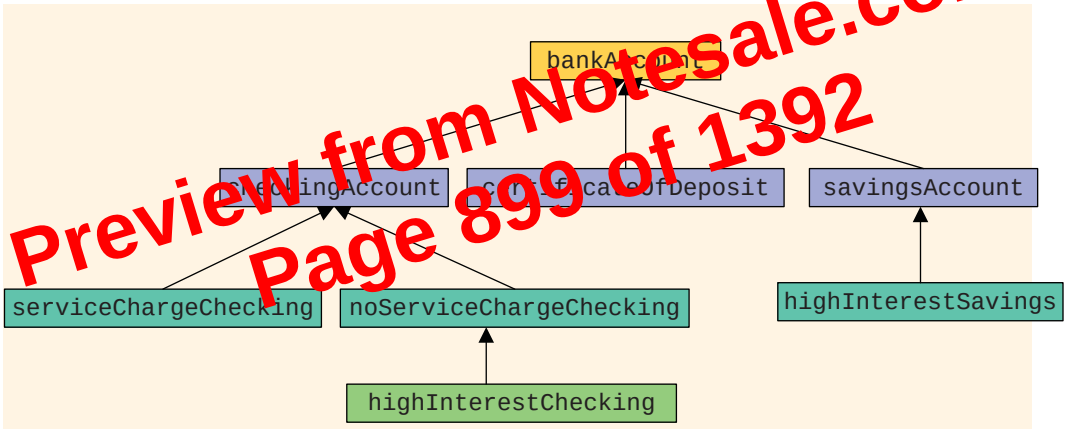


FIGURE 14-22 Inheritance hierarchy of banking accounts

Note that the classes `bankAccount` and `checkingAccount` are abstract. That is, we cannot instantiate objects of these classes. The other classes in Figure 14-22 are not abstract.

`bankAccount:` Every bank account has an account number, the name of the owner, and a balance. Therefore, instance variables such as `name`, `accountNumber`, and `balance` should be declared in the abstract `class` `bankAccount`. Some operations common to all types of accounts are retrieve account owner's name, account number, and account balance; make deposits; withdraw money; and create monthly statement. So include functions to implement these operations. Some of these functions will be pure virtual.

`checkingAccount:` A checking account is a bank account. Therefore, it inherits all the properties of a bank account. Because one of the objectives of a checking account is to be able to write checks, include the pure virtual function `writeCheck` to write a check.

Syntax for Operator Functions

The result of an operation is a value. Therefore, the operator function is a value-returning function.

The syntax of the heading for an operator function is:

```
returnType operator operatorSymbol(formal parameter list)
```

In C++, `operator` is a reserved word.

Recall that the only built-in operations on classes are assignment (=) and member selection. To use other operators on class objects, they must be explicitly overloaded. Operator overloading provides the same concise expressions for user-defined data types as it does for built-in data types.

To overload an operator for a class:

1. Include the statement to declare the function to overload the operator (that is, the operator function) prototype in the definition of the class.
2. Write the definition of the operator function.

Certain rules must be followed when you include an operator function in a class definition. These rules are described in the section, "Operator Functions as Member Functions and Nonmember Functions" later in this chapter.

Overloading an Operator: Some Restrictions

When overloading an operator, keep the following in mind:

1. You cannot change the precedence of an operator.
2. The associativity cannot be changed. (For example, the associativity of the arithmetic operator addition is from left to right, and it cannot be changed.)
3. Default parameters cannot be used with an overloaded operator.
4. You cannot change the number of parameters an operator takes.
5. You cannot create new operators. Only existing operators can be overloaded.
6. The operators that cannot be overloaded are:

   ```
   .   .*   ::   ?:   sizeof
   ```

7. The meaning of how an operator works with built-in types, such as `int`, remains the same.
8. Operators can be overloaded either for objects of the user-defined types, or for a combination of objects of the user-defined type and objects of the built-in type.

type rectangleType, the operator function that overloads the insertion operator for rectangleType must be a *nonmember* function of the class rectangleType.

Similarly, the operator function that overloads the stream extraction operator for rectangleType must be a nonmember function of the class rectangleType.

OVERLOADING THE STREAM INSERTION OPERATOR (<<)

The general syntax to overload the stream insertion operator, <<, for a class is described next.

Function Prototype (to be included in the definition of the class):

```
friend ostream& operator<<(ostream&, const className&);
```

Function Definition:

```
ostream& operator<<(ostream& osObject, const className& cObject)
{
    //local declaration if any
    //Output the members of cObject
    //osObject << . . .

    //Return the stream object.
    return osObject;
}
```


In this function definition:

- Both parameters are reference parameters.
- The first parameter—that is, osObject— is a reference to an ostream object.
- The second parameter is usually a const reference to a particular class, because (recall from Chapter 12) the most effective way to pass an object as a parameter to a class is by reference. In this case, the formal parameter does not need to copy the member variables of the actual parameter. The word const appears before the class name because we want to print only the member variables of the object. That is, the function should not modify the member variables of the object.
- The function return type is a reference to an ostream object.

The return type of the function to overload the operator << must be a reference to an ostream object for the following reasons.

Suppose that the operator << is overloaded for the class rectangleType. The statement:

```
cout << myRectangle;
```

is equivalent to the statement:

```
operator<<(cout, myRectangle);
```

the formal parameter `rightObject` also refers to the object `myRectangle`. Therefore, in the expression:

```
this != &rightObject
```

`this` and `&rightObject` both mean the address of `myRectangle`. Thus, the expression will evaluate to `false` and, therefore, the body of the `if` statement will be skipped.

NOTE

This note illustrates another reason why the body of the operator function must prevent self-assignments. Let us consider the following class:

```
class arrayClass
{
public:
    const arrayClass& operator= (const& arrayClass);
    .
    .
    .
private:
    int *list;
    int length;
    int maxSize;
};
```


The `class` `arrayClass` has a pointer member variable, `list`, which is used to create an array to store integers. Suppose that the definition of the function to overload the assignment operator for the `class` `arrayClass` is written without the `if` statement, as follows:

```
const arrayClass & arrayClass::operator=
                    (const arrayClass& otherList)
{
    delete [] list;                          //Line 1
    maxSize = otherList.maxSize;             //Line 2
    length = otherList.length;               //Line 3

    list = new int[maxSize];                 //Line 4

    for (int i = 0; i < length; i++)         //Line 5
        list[i] = otherList.list[i];         //Line 6

    return *this;                            //Line 7
}
```

Suppose that we have the following declaration in a user program:

```
arrayClass myList;
```

Consider the following statement:

```
myList = myList;
```

```
    int i;                                                  //Line 12
    int number;                                             //Line 13

    cout << "Line 14: Enter 5 integers: ";                  //Line 14

    for (i = 0; i < 5; i++)                                 //Line 15
    {
        cin >> number;                                      //Line 16
        intList1.insertEnd(number);                         //Line 17
    }

    cout << endl;                                           //Line 18
    cout << "Line 19: intList1: ";                          //Line 19
    intList1.print();                                       //Line 20

    intList3 = intList2 = intList1;                         //Line 21

    cout << "Line 22: intList2: ";                          //Line 22
    intList2.print();                                       //Line 23

    intList2.destroyList();                                 //Line 24

    cout << endl;                                           //Line 25
    cout << "Line 26: intList2: ";                          //Line 26
    intList2.print();                                       //Line 27

    cout << "Line 28: After destroying intList2, "
         << "intList1: ";                                   //Line 28
    intList1.print();                                       //Line 29

    cout << "Line 30: After destroying intList2, "
         << "intList3: ";                                   //Line 30
    intList3.print();                                       //Line 31
    cout << endl;                                           //Line 32

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Line 14: Enter 5 integers: 8 5 3 7 2

Line 19: intList1: 8 5 3 7 2
Line 22: intList2: 8 5 3 7 2

Line 26: intList2: The list is empty.
Line 28: After destroying intList2, intList1: 8 5 3 7 2
Line 30: After destroying intList2, intList3: 8 5 3 7 2
```

The statement in Line 9 creates `intList1` of size `10`; the statements in Lines 10 and 11 create `intList2` and `intList3` of (default) size `50`. The statements in Lines 15 through 17 input the data into `intList1`, and the statement in Line 20 outputs `intList1`. The

```
    //Overload the equality operator.
bool clockType::operator==(const clockType& otherClock) const
{
    return (hr == otherClock.hr && min == otherClock.min
            && sec == otherClock.sec);
}
```

The definition of the function operator<= is given next. The first time is less than or equal to the second time if:

1. The hours of the first time are less than the hours of the second time, or
2. The hours of the first time and the second time are the same, but the minutes of the first time are less than the minutes of the second time, or
3. The hours and minutes of the first time and the second time are the same, but the seconds of the first time are less than or equal to the seconds of the second time.

The definition of the function operator<= is:

```
    //Overload the less than or equal to operator.
bool clockType::operator<=(const clockType& otherClock) const
{
    return ((hr < otherClock.hr) ||
            (hr == otherClock.hr && min < otherClock.min) ||
            (hr == otherClock.hr && min == otherClock.min &&
             sec <= otherClock.sec));
}
```

In a similar manner, we can write the definitions of the other relational operator functions as follows:

```
    //Overload the not equal operator.
bool clockType::operator!=(const clockType& otherClock) const
{
    return (hr != otherClock.hr || min != otherClock.min
            || sec != otherClock.sec);
}

    //Overload the less than operator.
bool clockType::operator<(const clockType& otherClock) const
{
    return ((hr < otherClock.hr) ||
            (hr == otherClock.hr && min < otherClock.min) ||
            (hr == otherClock.hr && min == otherClock.min &&
             sec < otherClock.sec));
}

    //Overload the greater than or equal to operator.
bool clockType::operator>=(const clockType& otherClock) const
```

```
    cout << "Line 4: yourClock = " << yourClock
         << endl;                                          //Line 4

    cout << "Line 5: Enter the time in the form "
         << "hr:min:sec ";                                 //Line 5
    cin >> myClock;                                        //Line 6
    cout << endl;                                          //Line 7

    cout << "Line 8: The new time of myClock = "
         << myClock << endl;                               //Line 8

    ++myClock;                                             //Line 9

    cout << "Line 10: After incrementing the time, "
         << "myClock = " << myClock << endl;               //Line 10

    yourClock.setTime(13, 35, 38);                         //Line 11

    cout << "Line 12: After setting the time, "
         << "yourClock = " << yourClock << endl;           //Line 12

    if (myClock == yourClock)                              //Line 13
        cout << "Line 14: The times of myClock and "
             << "yourClock are equal." << endl;            //Line 14
    else                                                   //Line 15
        cout << "Line 16: The times of myClock and "
             << "yourClock are not equal." << endl;        //Line 16

    if (myClock <= yourClock)                              //Line 17
        cout << "Line 18: The time of myClock is "
             << "less than or equal to " << endl
             << "the time of yourClock." << endl;          //Line 18
    else                                                   //Line 19
        cout << "Line 20: The time of myClock is "
             << "greater than the time of "
             << "yourClock." << endl;                      //Line 20

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Line 3: myClock = 05:06:23
Line 4: yourClock = 00:00:00
Line 5: Enter the time in the form hr:min:sec 4:50:59

Line 8: The new time of myClock = 04:50:59
Line 10: After incrementing the time, myClock = 04:51:00
Line 12: After setting the time, yourClock = 13:35:38
Line 16: The times of myClock and yourClock are not equal.
Line 18: The time of myClock is less than or equal to
the time of yourClock.
```

To output a complex number in the form:

```
(a, b)
```

in which a is the real part and b is the imaginary part, clearly the algorithm is:

a. Output the left parenthesis, (.
b. Output the real part.
c. Output the comma and a space.
d. Output the imaginary part.
e. Output the right parenthesis,).

Therefore, the definition of the function operator<< is:

```
ostream& operator<<(ostream& osObject,
                    const complexType& complex)

{
    osObject << "(";                          //Step a
    osObject << complex.realPart;              //Step b
    osObject << ", ";                          //Step c
    osObject << complex.imaginaryPart;         //Step d
    osObject << ")";                           //Step e

    return osObject;    //return the ostream object
}
```


Next, we discuss the definition of the function to overload the stream extraction operator, >>.

The input is of the form:

```
(3, 5)
```

In this input, the real part of the complex number is 3, and the imaginary part is 5. Clearly, the algorithm to read this complex number is:

a. Read and discard the left parenthesis.
b. Read and store the real part.
c. Read and discard the comma.
d. Read and store the imaginary part.
e. Read and discard the right parenthesis.

Following these steps, the definition of the function operator>> is:

```
istream& operator>>(istream& isObject, complexType& complex)
{
    char ch;

    isObject >> ch;                        //Step a
    isObject >> complex.realPart;          //Step b
```

```
    Type& operator[](int index);
      //Overload the operator for nonconstant arrays
    const Type& operator[](int index) const;
      //Overload the operator for constant arrays
      .
      .
      .
private:
    Type *list; //pointer to the array
    int arraySize;
};
```

in which `Type` is the data type of the array elements.

The definitions of the functions to overload the operator `[]` for `classTest` are:

```
    //Overload the operator [] for nonconstant arrays
Type& classTest::operator[](int index)
{
    assert(0 <= index && index < arraySize);
    return list[index];  //return a pointer of the
                         //array component
}
  //Overload the operator [] for constant arrays
const Type& classTest::operator[](int index) const
{
    assert(0 <= index && index < arraySize);
    return list[index];  //return a pointer of the
                         //array component
}
```


NOTE

The preceding function definitions use the `assert` statement. (For an explanation of the `assert` statement, see Chapter 4 or the Appendix.)

Consider the following statements:

```
classTest list1;
classTest list2;
const classTest list3;
```

In the case of the statement:

```
list1[2] = list2[3];
```

the body of the operator function `operator[]` for nonconstant arrays is executed. In the case of the statement:

```
list1[2] = list3[5];
```

first, the body of the operator function `operator[]` for constant arrays is executed because `list3` is a constant array. Next, the body of the operator function `operator[]` for nonconstant arrays is executed to complete the execution of the assignment statement.

```
    //Default constructor to store the null string
newString::newString()
{
    strLength = 0;
    strPtr = new char[1];
    strcpy(strPtr, "");
}

newString::newString(const newString& rightStr)  //copy constructor
{
    strLength = rightStr.strLength;
    strPtr = new char[strLength + 1];
    strcpy(strPtr, rightStr.strPtr);
}

newString::~newString()  //destructor
{
   delete [] strPtr;
}

  //overloaded the assignment operator
const newString& newString::operator=(const newString& rightStr)
{
    if (this != &rightStr) //avoid self-copy
    {
       delete [] strPtr;
       strLength = rightStr.strLength;
       strPtr = new char[strLength + 1];
       strcpy(strPtr, rightStr.strPtr);
    }

    return *this;
}

char& newString::operator[] (int index)
{
    assert(0 <= index && index < strLength);
    return strPtr[index];
}

const char& newString::operator[](int index) const
{
    assert(0 <= index && index < strLength);
    return strPtr[index];
}

  //Overload the relational operators.
bool newString::operator==(const newString& rightStr) const
{
    return (strcmp(strPtr, rightStr.strPtr) == 0);
}
```

Most of these functions are quite straightforward. Let us explain the functions that overload the conversion constructor, the assignment operator, and the copy constructor.

The **conversion constructor** is a single-parameter function that converts its argument to an object of the constructor's class. In our case, the conversion constructor converts a string to an object of the `newString` type.

Note that the assignment operator is explicitly overloaded only for objects of the `newString` type. However, the overloaded assignment operator also works if we want to store a C-string into a `newString` object. Consider the declaration:

```
newString str;
```

and the statement:

```
str = "Hello there";
```

The compiler translates this statement into:

```
str.operator=("Hello there");
```

1. First, the compiler automatically invokes the conversion constructor to create an object of the `newString` type to temporarily store the string `"Hello there"`.
2. Second, the compiler invokes the overloaded assignment operator to assign the temporary `newString` object to the object `str`.

Hence, it is not necessary to explicitly overload the assignment operator to store a C-string into an object of type `newString`.

Next, we write a C++ program that tests some of the operations of the `class newString`.

```
//***********************************************************
// Author: D.S. Malik
//
// This program shows how to use the class newString.
//***********************************************************

#include <iostream>
#include "myString.h"

using namespace std;

int main()
{
    newString str1 = "Sunny";       //initialize str1 using
                                    //the assignment operator
    const newString str2("Warm");   //initialize str2 using the
                                    //conversion constructor
```

```
Line 6: Enter a string with a length of at least 7: 123456789
Line 9: The new value of str1 = 123456789
Line 11: str3 = Birth Day, str4 = Birth Day
Line 13: The new value of str3 = 123456789
Line 16: After replacing the second character of str3 = 1t3456789
Line 18: After replacing the third character of str3 = 1tW456789
Line 20: After replacing the sixth character of str3 = 1tW45g789
```

The preceding program works as follows. The statement in Line 1 outputs the values of `str1`, `str2`, and `str3`. Notice that the value of `str3` is to be printed between `***` and `###`. Because `str3` is empty, nothing is printed between `***` and `###`; see Line 1 in the sample run. The statements in Lines 2 through 5 compare `str1` and `str2` and output the result. The statement in Line 7 inputs a string with length of at least 7 into `str1`, and the statement in Line 9 outputs the new value of `str1`. Note that in the statement (see Line 10):

```
str4 = str3 = "Birth Day";
```

Because the associativity of the assignment operator is from right to left, first the statement `str3 = "Birth Day";` executes and then the statement `str4 = str3;` executes. The statement in Line 11 outputs the values of `str3` and `str4`. The statements in Lines 13 and 19 use the array subscripting operator `[]` to individually manipulate the characters of `str3`. The meanings of the remaining statements are straightforward.

Function Overloading

The previous section discussed operator overloading. Operator overloading provides the programmer with the same concise notation for user-defined data types as the operator has for built-in types. The types of parameters used with an operator determine the action to take. Similar to operator overloading, C++ allows the programmer to overload a function name. Chapter 7 introduced function overloading. For easy reference in the following discussion, let us review this concept.

Recall that a class can have more than one constructor, but all constructors of a class have the same name, which is the name of the class. This is an example of overloading a function. Further recall that overloading a function refers to having several functions with the same name but different parameter lists. The parameter list determines which function will execute.

For function overloading to work, we must give the definition of each function. The next section teaches you how to overload functions with a single code segment and leave the job of generating code for separate functions for the compiler.

```
    void insert(const elemType& newElement);
      //Function to insert newElement in the list.
      //Precondition: Prior to insertion, the list must
      //              not be full.
      //Postcondition: The list is the old list plus
      //               newElement.

    void remove(const elemType& removeElement);
      //Function to remove removeElement from the list.
      //Postcondition: If removeElement is found in the list,
      //               it is deleted from the list, and the
      //               list is the old list minus removeElement.
      //               If the list is empty, output the message
      //               "Cannot delete from the empty list."

    void destroyList();
      //Function to destroy the list.
      //Postcondition: length = 0;

     void printList();
      //Function to output the elements of the list.

    listType();
      //Default constructor
      //Sets the length of the list to 0.
      //Postcondition: length = 0;

protected:
    elemType list[100];   //array to hold the list elements
    int length;           //variable to store the number of
                          //elements in the list
};
```


This definition of the class template `listType` is a generic definition and includes only the basic operations on a list. To derive a specific list from this list and to add or rewrite the operations, we declare the array containing the list elements and the length of the list as `protected`.

Next, we describe a specific list. Suppose that you want to create a list to process integer data. The statement:

```
listType<int> intList;                  //Line 1
```

declares `intList` to be an object of `listType`. The `protected` member `list` is an array of `100` components, with each component being of type `int`. Similarly, the statement:

```
listType<newString> stringList;         //Line 2
```

declares `stringList` to be an object of `listType`. The `protected` member `list` is an array of `100` components, with each component being of type `newString`.

e. The precedence of an operator cannot be changed, but its associativity can be changed.

f. Every instance of an overloaded function has the same number of parameters.

g. It is not necessary to overload relational operators for classes that have only `int` member variables.

h. The member function of a `class` template is a function template.

i. When writing the definition of a `friend` function, the keyword `friend` must appear in the function heading.

j. Templates provide the capability for software reuse.

k. The function heading of the operator function to overload the pre-increment operator `(++)` and the post-increment operator `(++)` is the same because both operators have the same symbol.

2. What is a `friend` function?

3. What is the difference between a `friend` function of a class and a member function of a class?

4. Consider the definition of the `class` `dateType` given in Chapter 13.

   a. Write the statement that includes a `friend` function named `before` in the `class` `dateType` that takes as parameters two objects of type `dateType` and returns `true` if the date represented by the first object comes before the date represented by the second object; otherwise the function returns `false`.

   b. Write the definition of the function you defined in part a.

5. Suppose that the operator `<<` is to be overloaded for a user-defined `class` `mystery`. Why must `<<` be overloaded as a `friend` function?

6. Suppose that the binary operator `+` is overloaded as a member function for a `class` `strange`. How many parameters does the function `operator+` have?

7. When should a class overload the assignment operator and define the copy constructor?

8. Consider the following declaration:

```
class strange
{
    .
    .
    .
};
```

   a. Write a statement that shows the declaration in the `class` `strange` to overload the operator `>>`.

   b. Write a statement that shows the declaration in the `class` `strange` to overload the operator `=`.

   c. Write a statement that shows the declaration in the `class` `strange` to overload the binary operator `+` as a member function.

4. a. The increment and relational operators in the class clockType are overloaded as member functions. Rewrite the definition of the class clockType so that these operators are overloaded as nonmember functions. Also, overload the post-increment operator for the class clockType as a nonmember.
   b. Write the definitions of the member functions of the class clockType as designed in part a.
   c. Write a test program that tests various operations on the class as designed in parts a and b.
5. a. Extend the definition of the class complexType so that it performs the subtraction and division operations. Overload the operators subtraction and division for this class as member functions.
   If (a, b) and (c, d) are complex numbers:
   $(a, b) - (c, d) = (a - c, b - d)$.
   If (c, d) is nonzero:
   $(a, b) / (c, d) = ((ac + bd) / (c^2 + d^2), (bc - ad) / (c^2 + d^2))$.
   b. Write the definitions of the functions to overload the operators - and / as defined in part a.
   c. Write a test program that tests various operations on the class complexType. Format your answer with two decimal places.
6. a. Rewrite the definition of the class complexType so that the arithmetic and relational operators are overloaded as nonmember functions.
   b. Write the definitions of the member functions of the class complexType as designed in part a.
   c. Write a test program that tests various operations on the class complexType as designed in parts a and b. Format your answer with two decimal places.
7. a. Extend the definition of the class newString as follows:
      i. Overload the operators + and += to perform the string concatenation operations.
      ii. Add the function length to return the length of the string.
   b. Write the definition of the function to implement the operations defined in part a.
   c. Write a test program to test various operations on the newString objects.
8. a. Rewrite the definition of the class newString as defined and extended in Programming Exercise 7 so that the relational operators are overloaded as nonmember functions.
   b. Write the definition of the class newString as designed in part a.
   c. Write a test program that tests various operations on the class newString.

vert a Roman number into its equivalent decimal number.

Modify the definition of the `class` `romanType` so that the member variables are declared as `protected`. Use the `class` `newString`, as designed in Programming Exercise 7, to manipulate strings. Furthermore, overload the stream insertion and stream extraction operators for easy input and output. The stream insertion operator outputs the Roman number in the Roman format.

Also, include a member function, `decimalToRoman`, that converts the decimal number (the decimal number must be a positive integer) to an equivalent Roman number format. Write the definition of the member function `decimalToRoman`.

For simplicity, we assume that only the letter `I` can appear in front of another letter and that it appears only in front of the letters `V` and `X`. For example, `4` is represented as `IV`, `9` is represented as `IX`, `39` is represented as `XXXIX`, and `49` is represented as `XXXXIX`. Also, `40` will be represented as `XXXX`, `190` will be represented as `CIXXX`, and so on.

b. Derive a `class` `extRomanType` from the `class` `romanType` to do the following: In the `class` `extRomanType`, overload the arithmetic operators `+`, `-`, `*`, and `/` so that arithmetic operations can be performed on Roman numbers. Also, overload the pre- and post-increment and decrement operators as member functions of the `class` `extRomanType`.

   To add (subtract, multiply, or divide) Roman numbers, add (subtract, multiply, or divide, respectively) their decimal representations and then convert the result to the Roman number format. For subtraction, if the first number is smaller than the second number, output a message saying that, "`Because the first number is smaller than the second, the numbers cannot be subtracted`". Similarly, for division, the numerator must be larger than the denominator. Use similar conventions for the increment and decrement operators.

c. Write the definitions of the functions to overload the operators described in part b.

d. Test your `class` `extRomanType` on the following program. (Include the appropriate header files.)

```
int main()
{
    extRomanType num1("XXXIV");
    extRomanType num2("XV");
    extRomanType num3;

    cout << "Num1 = " << num1 << endl;
    cout << "Num2 = " << num2 << endl;
    cout << "Num1 + Num2 = " << num1 + num2 << endl;
```

```
*********  First Investor's Heaven  **********
*********     Financial Report       **********
Stock              Today                        Previous  Percent
Symbol  Open     Close    High     Low          Close     Gain        Volume
------  -----    -----    -----    -----        --------  -------     ------
   ABC  123.45   130.95   132.00   125.00       120.50     8.67%        10000
  AOLK   80.00    75.00    82.00    74.00        83.00    -9.64%         5000
  CSCO  100.00   102.00   105.00    98.00       101.00     0.99%        25000
   IBD   68.00    71.00    72.00    67.00        75.00    -5.33%        15000
  MSET  120.00   140.00   145.00   140.00       115.00    21.74%        30920
Closing Assets: $9628300.00
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

Develop this programming exercise in two steps. In the first step (part a), design and implement a stock object. In the second step (part b), design and implement an object to maintain a list of stocks.

a. (Stock Object) Design and implement the stock object. Call the class that captures the various characteristics of a stock object `stockType`.

   The main components of a stock are the stock symbol, stock price, and number of shares. Moreover, we need to output the opening price, closing price, high price, low price, previous price, and the percent gain/loss for the day. These are also all the characteristics of a stock. Therefore, the stock object should store all this information.

   Perform the following operations on each stock object:

   i. Set the stock information.
   ii. Print the stock information.
   iii. Show the different prices.
   iv. Calculate and print the percent gain/loss.
   v. Show the number of shares.

      a.1. The natural ordering of the stock list is by stock symbol. Overload the relational operators to compare two stock objects by their symbols.

      a.2. Overload the insertion operator, `<<`, for easy output.

      a.3. Because the data is stored in a file, overload the stream extraction operator, `>>`, for easy input.

   For example, suppose `infile` is an `ifstream` object and the input file was opened using the object `infile`. Further suppose that `myStock` is a stock object. Then, the statement:

   ```
   infile >> myStock;
   ```

   reads the data from the input file and stores it in the object `myStock`. (Note that this statement reads and stores the data in the relevant components of `myStock`.)

b. Now that you have designed and implemented the **class** **stockType** to implement a stock object in a program, it is time to create a list of stock objects.

Let us call the class to implement a list of stock objects **stockListType**.

The **class** **stockListType** must be derived from the **class** **listType**, which you designed and implemented in the previous exercise. However, the **class** **stockListType** is a very specific class, designed to create a list of stock objects. Therefore, the **class** **stockListType** is no longer a template.

Add and/or overwrite the operations of the **class** **listType** to implement the necessary operations on a stock list.

The following statement derives the **class** **stockListType** from the **class** **listType**.

```
class stockListType: public listType<stockType>
{
    //member list
};
```

The member variables to hold the list elements, the length of the list, and the **max** **listSize** were declared as **protected** in the **class** **listType**. Therefore, these members can be directly accessed in the **class** **stockListType**.

Because the company also requires you to produce the list ordered by the percent gain/loss, you need to sort the stock list by this component. However, you are not to physically sort the list by the component percent gain/loss. Instead, you will provide a logical ordering with respect to this component.

To do so, add a member variable, an array, to hold the indices of the stock list ordered by the component percent gain/loss. Call this array **sortIndicesGainLoss**. When printing the list ordered by the component percent gain/loss, use the array **sortIndicesGainLoss** to print the list. The elements of the array **sortIndicesGainLoss** will tell which component of the stock list to print next.

c. Write a program that uses these two classes to automate the company's analysis of stock data.

Suppose there is a statement that can generate an exception, for example, division by 0. Usually, before executing such a statement, we check whether certain conditions are met. For example, before performing the division, we check whether the divisor is nonzero. If the conditions are not met, we typically generate an exception, which in C++ terminology is called throwing an exception. This is typically done using the **throw** statement, which we will explain shortly. We will show what is typically thrown to generate an exception.

Let us now note the following about **try**/**catch** blocks.

- If no exception is thrown in a **try** block, all **catch** blocks associated with that **try** block are ignored and program execution resumes after the last **catch** block.
- If an exception is thrown in a **try** block, the remaining statements in that **try** block are ignored. The program searches the **catch** blocks in the order they appear after the **try** block and looks for an appropriate exception handler. If the type of thrown exception matches the parameter type in one of the **catch** blocks, the code of that **catch** block executes, and the remaining **catch** blocks after this **catch** block are ignored.
- The last **catch** block that has an ellipsis (three dots) is designed to catch any type of exception.

Consider the following **catch** block:

```
catch (int x)
{
    //exception-handling code
}
```

In this **catch** block:

- The identifier **x** acts as a parameter. In fact, it is called a **catch** block parameter.
- The data type **int** specifies that this **catch** block can catch an exception of type **int**.
- A **catch** block can have *at most* one **catch** block parameter.

Essentially, the **catch** block parameter becomes a placeholder for the value thrown. In this case, **x** becomes a placeholder for any thrown value that is of type **int**. In other words, if the thrown value is caught by this **catch** block, then the thrown value is stored in the **catch** block parameter. This way, if the exception-handling code wants to do something with that value, it can be accessed via the **catch** block parameter.

Suppose in a **catch** block heading only the data type is specified, that is, there is no **catch** block parameter. The thrown value then *may not* be accessible in the **catch** block exception-handling code.

THROWING AN EXCEPTION

In order for an exception to occur in a **try** block and be caught by a **catch** block, the exception must be thrown in the **try** block. The general syntax to **throw** an exception is:

```
throw expression;
```

```
void doDivision()
{
    int dividend, divisor, quotient;                    //Line 3

    try
    {
        cout << "Line 4: Enter the dividend: ";         //Line 4
        cin >> dividend;                                //Line 5
        cout << endl;                                   //Line 6

        cout << "Line 7: Enter the divisor: ";          //Line 7
        cin >> divisor;                                 //Line 8
        cout << endl;                                   //Line 9

        if (divisor == 0)                               //Line 10
            throw divisionByZero();                     //Line 11

        quotient = dividend / divisor;                  //Line 12
        cout << "Line 13: Quotient = " << quotient
             << endl;                                   //Line 13
    }
    catch (divisionByZero divByZeroObj)                 //Line 14
    {
        cout << "Line 15: In the function "
             << "doDivision: "
             << divByZeroObj.what() << endl;            //Line 15
    }
}
```


Sample Run 1: In this sample run, the user input is shaded.

```
Line 4: Enter the dividend: 34

Line 7: Enter the divisor: 5

Line 13: Quotient = 6
```

Sample Run 2: In this sample run, the user input is shaded.

```
Line 4: Enter the dividend: 56

Line 7: Enter the divisor: 0

Line 15: In the function doDivision: Division by zero
```

Rethrowing and Throwing an Exception

When an exception occurs in a `try` block, control immediately passes to one of the `catch` blocks. Typically, a `catch` block either handles the exception or partially processes the exception and then rethrows the same exception, or it rethrows another exception in order for the calling environment to handle the exception. The `catch` block in Examples 16-4 through 16-13 handles the exception. The mechanism of rethrowing or throwing an exception is quite useful in cases in which a `catch` block catches the exception but cannot handle the exception, or if the `catch` block decides that the exception should be

```
int main()
{
    try                                                          //Line 1
    {
        doDivision();                                            //Line 2
    }
    catch (divisionByZero divByZeroObj)                          //Line 3
    {
        cout << "Line 4: In main: "
             << divByZeroObj.what() << endl;                     //Line 4
    }

    return 0;                                                    //Line 5
}

void doDivision() throw (divisionByZero)
{
    int dividend, divisor, quotient;                             //Line 6

    try                                                          //Line 7
    {
        cout << "Line 8: Enter the dividend: ";                  //Line 8
        cin >> dividend;                                         //Line 9
        cout << endl;                                            //Line 10

        cout << "Line 11: Enter the divisor: ";                  //Line 11
        cin >> divisor;                                          //Line 12
        cout << endl;                                            //Line 13

        if (divisor == 0)                                        //Line 14
            throw divisionByZero("Found division by 0!");        //Line 15

        quotient = dividend / divisor;                           //Line 16
        cout << "Line 17: Quotient = " << quotient
             << endl;                                            //Line 17
    }
    catch (divisionByZero)                                       //Line 18
    {
        throw;                                                   //Line 19
    }
}
```


Sample Run 1: In this sample run, the user input is shaded.

```
Line 8: Enter the dividend: 34

Line 11: Enter the divisor: 5

Line 17: Quotient = 6
```

Sample Run 2: In this sample run, the user input is shaded.

```
Line 8: Enter the dividend: 56

Line 11: Enter the divisor: 0

Line 4: In main: Found division by 0!
```

```
            cout << endl;                                   //Line 10

            if (!cin)                                       //Line 11
                throw str;                                  //Line 12

            done = true;                                    //Line 13
            cout << "Line 14: Number = " << number
                 << endl;                                   //Line 14
        }                                                   //Line 15
        catch (string messageStr)                           //Line 16
        {                                                   //Line 17
            cout << "Line 18: " << messageStr
                 << endl;                                   //Line 18
            cout << "Line 19: Restoring the "
                 << "input stream." << endl;                //Line 19
            cin.clear();                                    //Line 20
            cin.ignore(100, '\n');                          //Line 21
        }                                                   //Line 22
    }
    while (!done);                                          //Line 23

    return 0;                                               //Line 24
}
```


Sample Run: In this sample run, the user input is shaded.

```
Line 8: Enter an integer: r5

Line 18: The input stream is in the fail state.
Line 19: Restoring the input stream.
Line 8: Enter an integer: d45

Line 18: The input stream is in the fail state.
Line 19: Restoring the input stream.
Line 8: Enter an integer: hw3

Line 18: The input stream is in the fail state.
Line 19: Restoring the input stream.
Line 8: Enter an integer: 48

Line 14: Number = 48
```

This program prompts the user to enter an integer. If the input is invalid, the standard input stream enters the fail state. In the `try` block, the statement in Line 12 throws an exception, which is a string object. Control passes to the `catch` block, and the exception is caught and processed. The statement in Line 20 restores the input stream to its good state, and the statement in Line 21 clears the rest of the input from the line. The `do...while` loop continues to prompt the user until the user inputs a valid number.

QUICK REVIEW

1. An exception is an occurrence of an undesirable situation that can be detected during program execution.
2. Some typical ways of dealing with exceptions are to use an `if` statement or the `assert` function.
3. The function `assert` can check whether an expression meets the required condition(s). If the conditions are not met, it terminates the program.
4. The `try`/`catch` block is used to handle exceptions within a program.
5. Statements that may generate an exception are placed in a `try` block. The `try` block also contains statements that should not be executed if an exception occurs.
6. The `try` block is followed by one or more `catch` blocks.
7. A `catch` block specifies the type of exception it can catch and contains an exception handler.
8. If the heading of a `catch` block contains ... (ellipses) in place of parameters, then this `catch` block can catch exceptions of all types.
9. If no exceptions are thrown in a `try` block, all `catch` blocks associated with that `try` block are ignored and program execution resumes after the last `catch` block.
10. If an exception is thrown in a `try` block, the remaining statements in the `try` block are ignored. The program searches the `catch` blocks, in the order they appear after the `try` block, and looks for an appropriate exception handler. If the type of the thrown exception matches the parameter type in one of the `catch` blocks, then the code in that `catch` block executes and the remaining `catch` blocks after this `catch` block are ignored.
11. The data type of the `catch` block parameter specifies the type of exception that the `catch` block can catch.
12. A `catch` block can have, at most, one `catch` block parameter.
13. If only the data type is specified in a `catch` block heading, that is, if there is no `catch` block parameter, then the thrown value may not be accessible in the `catch` block exception-handling code.
14. In order for an exception to occur in a `try` block and be caught by a `catch` block, the exception must be thrown in the `try` block.
15. The general syntax to `throw` an exception is:

    ```
    throw expression;
    ```

    in which `expression` is a constant value, variable, or object. The object being thrown can be either a specific object or an anonymous object.
16. C++ provides support to handle exceptions via a hierarchy of classes.

17. The `class exception` is the base class of the exception classes provided by C++.
18. The function `what` returns the string containing the exception object thrown by C++'s built-in exception classes.
19. The `class exception` is contained in the header file `exception`.
20. The two classes that are immediately derived from the `class exception` are `logic_error` and `runtime_error`. Both of these classes are defined in the header file `stdexcept`.
21. The `class invalid_argument` is designed to deal with illegal arguments used in a function call.
22. The `class out_of_range` deals with the string subscript `out_of_range` error.
23. If a length greater than the maximum allowed for a string object is used, the `class length_error` deals with the error that occurs when a length greater than the maximum size allowed for the object being manipulated is used.
24. If the operator `new` cannot allocate memory space, this operator throws a `bad_alloc` exception.
25. The `class runtime_error` is designed to deal with errors that can be detected only during program execution. For example, to deal with arithmetic overflow and underflow exceptions, the classes `overflow_error` and `underflow_error` are derived from the `class runtime_error`.
26. A `catch` block typically handles the exception or partially processes the exception and then either rethrows the same exception or rethrows another exception in order for the calling environment to handle the exception.
27. C++ enables programmers to create their own exception classes to handle both the exceptions not covered by C++'s exception classes and their own exceptions.
28. C++ uses the same mechanism to process the exceptions you define as it uses for built-in exceptions. However, you must throw your own exceptions using the `throw` statement.
29. In C++, any class can be considered an exception class. It need not be inherited from the `class exception`. What makes a class an exception is how it is used.
30. The general syntax to rethrow an exception caught by a `catch` block is:

    ```
    throw;
    ```

    (in this case, the same exception is rethrown) or:

    ```
    throw expression;
    ```

    in which `expression` is a constant value, variable, or object. The object being thrown can be either a specific object or an anonymous object.

In previous chapters, to devise solutions to problems, we used the most common technique called iteration. For certain problems, however, using the iterative technique to obtain the solution is quite complicated. This chapter introduces another problem-solving technique called recursion and provides several examples demonstrating how recursion works.

Recursive Definitions

The process of solving a problem by reducing it to smaller versions of itself is called **recursion**. Recursion is a very powerful way to solve certain problems for which the solution would otherwise be very complicated. Let us consider a problem that is familiar to most everyone.

In mathematics, the factorial of a nonnegative integer is defined as follows:

$$0! = 1 \qquad (17\text{-}1)$$

$$n! = n \times (n-1)! \quad \text{if} \quad n > 0 \qquad (17\text{-}2)$$

In this definition, 0! is defined to be 1, and if n is an integer greater than 0, first we find $(n-1)!$ and then multiply it by n. To find $(n-1)!$, we apply the definition again. If $(n-1) > 0$, then we use Equation 17-2; otherwise, we use Equation 17-1. Thus, for an integer n greater than 0, $n!$ is obtained by first finding $(n-1)!$ (that is, $n!$ is reduced to a smaller version of itself) and then multiplying $(n-1)!$ by n.

Let us apply this definition to find 3!. Here, $n = 3$. Because $n > 0$, we use Equation 17-2 to obtain:

$$3! = 3 \times 2!$$

Next, we find 2! Here, $n = 2$. Because $n > 0$, we use Equation 17-2 to obtain:

$$2! = 2 \times 1!$$

Now, to find 1!, we again use Equation 17-2 because $n = 1 > 0$. Thus:

$$1! = 1 \times 0!$$

Finally, we use Equation 17-1 to find 0!, which is 1. Substituting 0! into 1! gives $1! = 1$. This gives $2! = 2 \times 1! = 2 \times 1 = 2$, which, in turn, gives $3! = 3 \times 2! = 3 \times 2 = 6$.

The solution in Equation 17-1 is direct—that is, the right side of the equation contains no factorial notation. The solution in Equation 17-2 is given in terms of a smaller version of itself. The definition of the factorial given in Equations 17-1 and 17-2 is called a **recursive definition**. Equation 17-1 is called the **base case** (that is, the case for which the solution is obtained directly); Equation 17-2 is called the **general case**.

Recursive definition: A definition in which something is defined in terms of a smaller version of itself.

The following recursive function implements this algorithm.

```
int rFibNum(int a, int b, int n)
{
    if (n == 1)
        return a;
    else if (n == 2)
        return b;
    else
        return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

Let us trace the execution of the following statement:

```
cout << rFibNum(2, 3, 5) << endl;
```

In this statement, the first number is 2, the second number is 3, and we want to determine the fifth Fibonacci number of the sequence. Figure 17-5 traces the execution of the expression `rFibNum(2,3,5)`. The value returned is 13, which is the fifth Fibonacci number of the sequence whose first number is 2 and second number is 3.

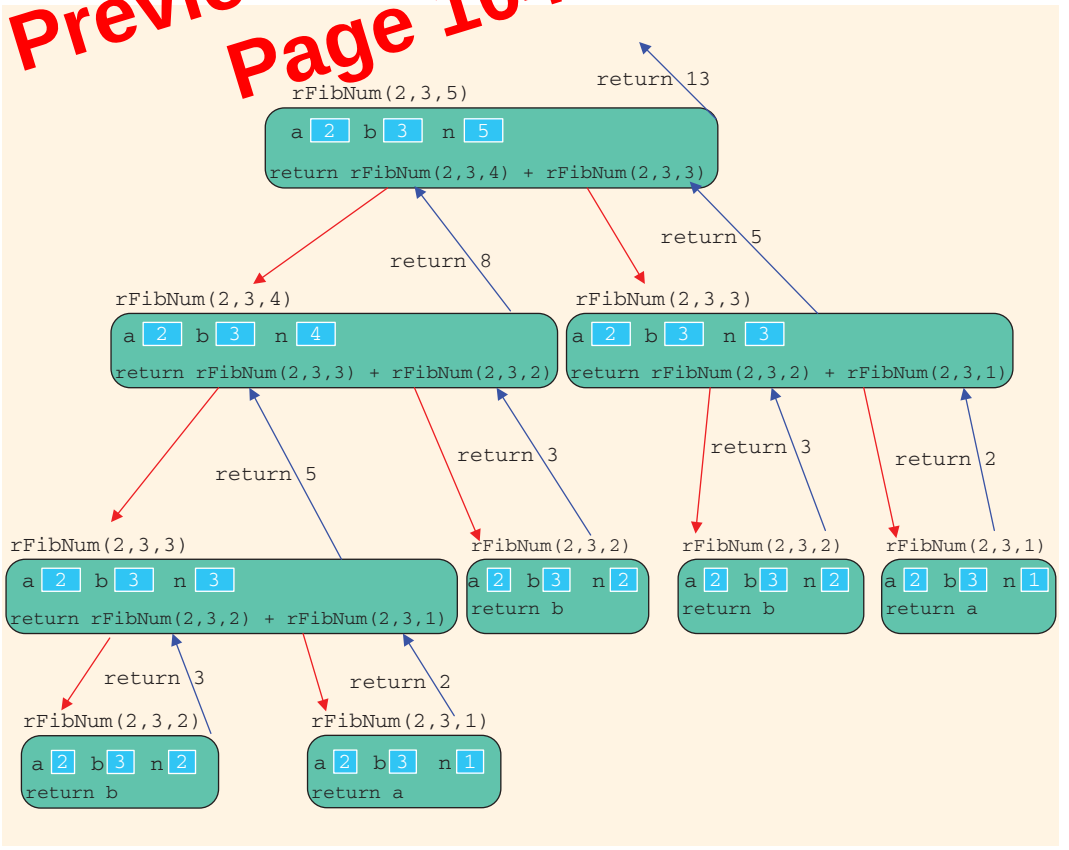


FIGURE 17-5 Execution of `rFibNum(2, 3, 5)`

The following C++ program uses the function `rFibNum`:

```
//Chapter 17: Fibonacci Number

#include <iostream>

using namespace std;

int rFibNum(int a, int b, int n);

int main()
{
    int firstFibNum;
    int secondFibNum;
    int nth;

    cout << "Enter the first Fibonacci number: ";
    cin >> firstFibNum;
    cout << endl;

    cout << "Enter the second Fibonacci number: ";
    cin >> secondFibNum;
    cout << endl;

    cout << "Enter the position of the desired Fibonacci number: ";
    cin >> nth;
    cout << endl;

    cout << "The Fibonacci number at position " << nth
         << " is: " << rFibNum(firstFibNum, secondFibNum, nth)
         << endl;

    return 0;
}

int rFibNum(int a, int b, int n)
{
    if (n == 1)
        return a;
    else if (n == 2)
        return b;
    else
        return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```


Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1

```
Enter the first Fibonacci number: 2

Enter the second Fibonacci number: 5
```

Because the `if` statement in call 5 fails, this call does not print anything. The first output is produced by call 4, which prints `1`; the second output is produced by call 3, which prints `1`; the third output is produced by call 2, which prints `0`; and the fourth output is produced by call 1, which prints `1`. Thus, the output of the statement:

```
decToBin(13, 2);
```

is:

```
1101
```

The following C++ program tests the function `decToBin`.

```
//*************************************************
// Author: D. S. Malik
//
// Program: Decimal to binary
// This program uses recursion to find the binary
// representation of a nonnegative integer.
//*************************************************

#include <iostream>
using namespace std;

void decToBin(int num, int base);

int main()
{
    int decimalNum;
    int base;

    base = 2;

    cout << "Enter number in decimal: ";
    cin >> decimalNum;
    cout << endl;

    cout << "Decimal " << decimalNum << " = ";
    decToBin(decimalNum, base);
    cout << " binary" << endl;

    return 0;
}

void decToBin(int num, int base)
{
    if (num > 0)
    {
        decToBin(num / base, base);
        cout << num % base;
    }
}
```

17. To design a recursive function, you must do the following:
    a. Understand the problem requirements.
    b. Determine the limiting conditions. For example, for a list, the limiting condition is the number of elements in the list.
    c. Identify the base cases and provide a direct solution to each base case.
    d. Identify the general cases and provide a solution to each general case in terms of smaller versions of itself.

EXERCISES

1. Mark the following statements as true or false.
    a. Every recursive definition must have one or more base cases.
    b. Every recursive function must have one or more base cases.
    c. The general case stops the recursion.
    d. In the general case, the solution to the problem is obtained directly.
    e. A recursive function always returns a value.
2. What is a base case?
3. What is a recursive case?
4. What is direct recursion?
5. What is indirect recursion?
6. What is tail recursion?
7. Consider the following recursive function:

```
int mystery(int number)                          //Line 1
{
    if (number == 0)                             //Line 2
        return number;                           //Line 3
    else                                         //Line 4
       return(number + mystery(number - 1));     //Line 5
}
```

    a. Identify the base case.
    b. Identify the general case.
    c. What valid values can be passed as parameters to the function `mystery`?
    d. If `mystery(0)` is a valid call, what is its value? If not, explain why.
    e. If `mystery(5)` is a valid call, what is its value? If not, explain why.
    f. If `mystery(-3)` is a valid call, what is its value? If not, explain why.
8. Consider the following recursive function:

```
void funcRec(int u, char v)                  //Line 1
{
   if (u == 0)                               //Line 2
      cout << v;                             //Line 3
```

This linked list has four nodes. The address of the first node is stored in the pointer `head`. Each node has two components: `info`, to store the info, and `link`, to store the address of the next node. For simplicity, we assume that `info` is of type `int`.

Suppose that the first node is at location `2000`, the second node is at location `2800`, the third node is at location `1500`, and the fourth node is at location `3600`. Therefore, the value of `head` is `2000`, the value of the component `link` of the first node is `2800`, the value of the component `link` of the second node is `1500`, and so on. Also, the value `0` in the component link of the last node means that this value is `NULL`, which we indicate by drawing a down arrow. The number at the top of each node is the address of that node. The following table shows the values of `head` and some other nodes in the list shown in Figure 18-4.

	Value	Explanation
`head`	2000	
`head->info`	17	Because `head` is 2000 and the `info` of the node at location 2000 is 17
`head->link`	2800	
`head->link->info`	92	Because `head->link` is 2800 and the `info` of the node at location 2800 is 92

Suppose that `current` is a pointer of the same type as the pointer `head`. Then, the statement:

```
current = head;
```

copies the value of `head` into `current` (see Figure 18-5).

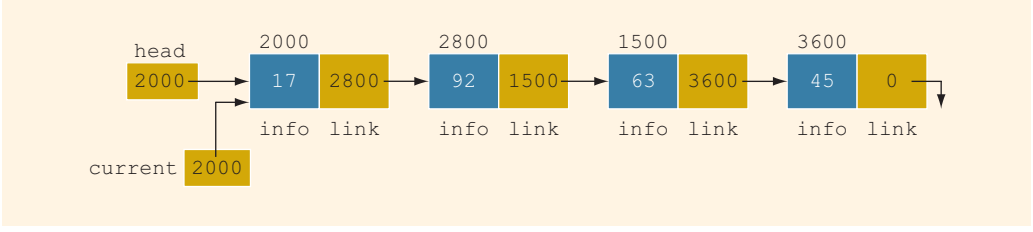
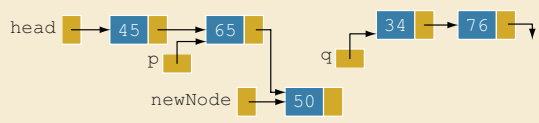
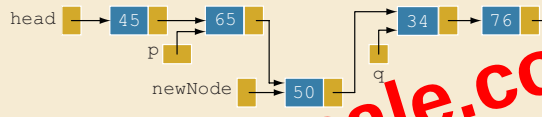


FIGURE 18-5 Linked list after the statement `current = head;` executes

Clearly, in Figure 18-5:

	Value
`current`	2000
`current->info`	17
`current->link`	2800
`current->link->info`	92

TABLE 18-2 Inserting a Node in a Linked List Using Two Pointers

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

Deletion

Consider the linked list shown in Figure 18-10.

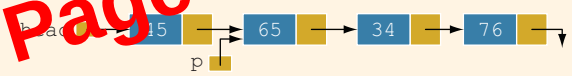


FIGURE 18-10 Node to be deleted is with `info 34`

Suppose that the node with `info 34` is to be deleted from the list. The following statement removes the node from the list.

```
p->link = p->link->link;
```

Figure 18-11 shows the resulting list after the preceding statement executes.

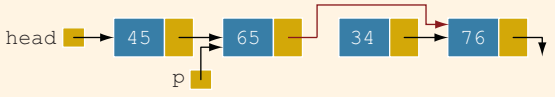


FIGURE 18-11 List after the statement `newNode->link = q;` executes

From Figure 18-11, it is clear that the node with `info 34` is removed from the list. However, the memory is still occupied by this node, and this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The

```
template <class Type>
bool linkedListIterator<Type>::operator!=
                  (const linkedListIterator<Type>& right) const
{
    return (current != right.current);
}
```

Now that we have defined the classes to implement the node of a linked list and an iterator to a linked list, next we describe the `class linkedListType` to implement the basic properties of a linked list.

The following abstract class defines the basic properties of a linked list as an ADT.


```
template <class Type>
class linkedListType
{
public:
    const linkedListType<Type>& operator=
                          (const linkedListType<Type>&);
      //Overload the assignment operator.

    void initializeList();
      //Initialize the list to an empty state.
      //Postcondition: first = NULL, last = NULL, count = 0;

    bool isEmptyList() const;
      //Function to determine whether the list is empty.
      //Postcondition: Returns true if the list is empty,
      //               otherwise it returns false.

    void print() const;
      //Function to output the data contained in each node.
      //Postcondition: none

    int length() const;
      //Function to return the number of nodes in the list.
      //Postcondition: The value of count is returned.

    void destroyList();
      //Function to delete all the nodes from the list.
      //Postcondition: first = NULL, last = NULL, count = 0;

    Type front() const;
      //Function to return the first element of the list.
      //Precondition: The list must exist and must not be
      //              empty.
      //Postcondition: If the list is empty, the program
      //               terminates; otherwise, the first
      //               element of the list is returned.

    Type back() const;
      //Function to return the last element of the list.
      //Precondition: The list must exist and must not be
      //              empty.
      //Postcondition: If the list is empty, the program
      //               terminates; otherwise, the last
      //               element of the list is returned.
```

```
    nodeType<Type> *first; //pointer to the first node of the list
    nodeType<Type> *last;  //pointer to the last node of the list

private:
    void copyList(const linkedListType<Type>& otherList);
      //Function to make a copy of otherList.
      //Postcondition: A copy of otherList is created and
      //               assigned to this list.
};
```

Figure 18-20 shows the UML class diagram of the **class** `linkedListType`.

FIGURE 18-20 UML class diagram of the **class** `linkedListType`

Note that typically, in the UML diagram, the name of an abstract class and abstract function is shown in italics.

The instance variables **`first`** and **`last`**, as defined earlier, of the **class** **`linkedListType`** are **`protected`**, not **`private`**, because as noted previously, we will derive the **class**es **`unorderedLinkedList`** and **`orderedLinkedList`** from the **class** **`linkedListType`**. Because each of the **class**es **`unorderedLinkedList`**

```
    newNode->link = first;      //insert newNode before first
    first = newNode;            //make first point to the
                                //actual first node
    count++;                    //increment count

    if (last == NULL)    //if the list was empty, newNode is also
                         //the last node in the list
        last = newNode;
}//end insertFirst
```

Insert the Last Node

The definition of the member function `insertLast` is similar to the definition of the member function `insertFirst`. Here, we insert the new node after `last`. Essentially, the function `insertLast` is:


```
template <class Type>
void unorderedLinkedList<Type>::insertLast(const Type& newItem)
{
    nodeType<Type> *newNode; //pointer to create the new node

    newNode = new nodeType<Type>; //create the new node
    newNode->info = newItem;    //store the new item in the node
    newNode->link = NULL;       //set the link field of newNode
                                //to NULL

    if (first == NULL)  //if the list is empty, newNode is
                        //both the first and last node
    {
        first = newNode;
        last = newNode;
        count++;        //increment count
    }
    else    //the list is not empty, insert newNode after last
    {
        last->link = newNode; //insert newNode after last
        last = newNode; //make last point to the actual
                        //last node in the list
        count++;        //increment count
    }
}//end insertLast
```

DELETE A NODE

Next, we discuss the implementation of the member function `deleteNode`, which deletes a node from the list with a given `info`. We need to consider several cases:

Case 1: The list is empty.

Case 2: The first node is the node with the given `info`. In this case, we need to adjust the pointer `first`.

values of `first` and `last`. The link field of the previous node—that is, `17`—changes. After deletion, the node with `info` `17` contains the address of the node with `24`.)

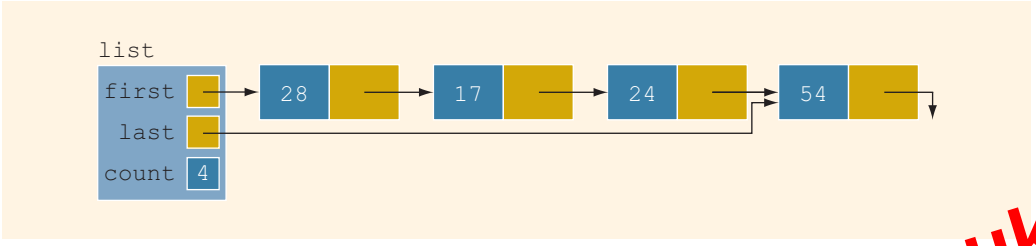


FIGURE 18-26 `list` after deleting 37

Case 3b: The node to be deleted is the last node.

Consider the list shown in Figure 18-27. Suppose that the node to be deleted is `54`.

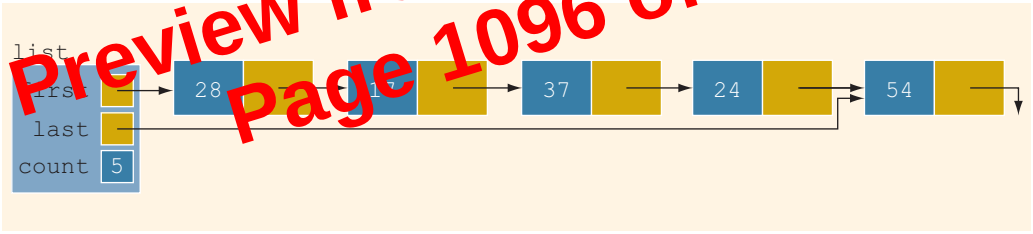


FIGURE 18-27 `list` before deleting 54

After deleting `54`, the node with `info` `24` becomes the last node. Therefore, the deletion of `54` requires us to change the value of the pointer `last`. After deleting `54`, `last` contains the address of the node with `info` `24`. Also, `count` is decremented by `1`. Figure 18-28 shows the resulting list.

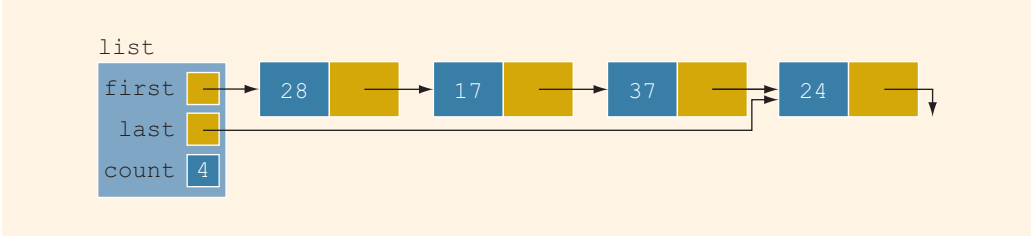


FIGURE 18-28 `list` after deleting 54

Case 4: The node to be deleted is not in the list. In this case, the list requires no adjustment. We simply output an error message, indicating that the item to be deleted is not in the list.

Suppose that `10` is to be inserted. After inserting `10` in the list, the node with `info` `10` becomes the first node of `list`. This requires us to change the value of `first`. Also, `count` is incremented by `1`. Figure 18-32 shows the resulting list.

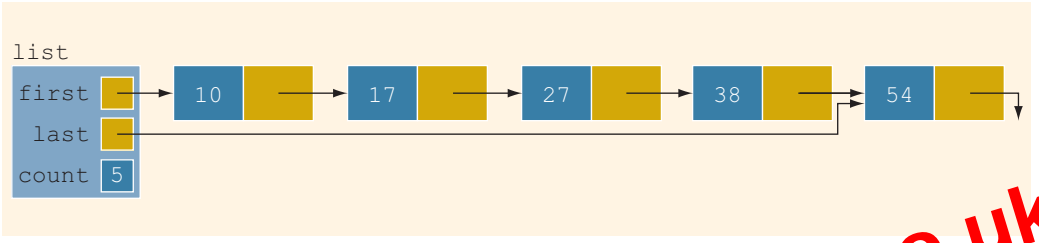


FIGURE 18-32 `list` after inserting `10`

Case 3: The list is not empty, and the item to be inserted is larger than the first item in the list. As indicated previously, this case has two scenarios.

Case 3a: The item to be inserted is larger than the largest item in the list; that is, it goes at the end of the list. Consider the list shown in Figure 18-33.

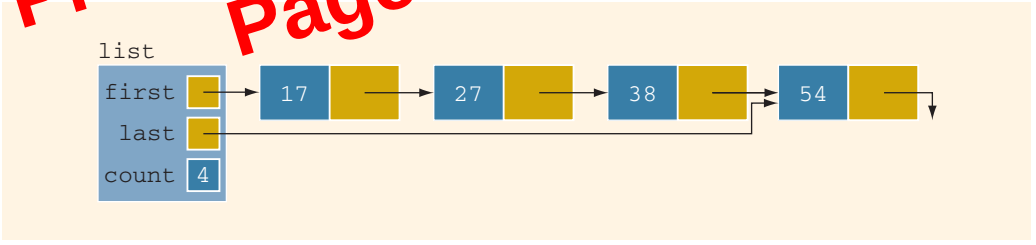


FIGURE 18-33 `list` before inserting `65`

Suppose that we want to insert `65` in the list. After inserting `65`, the resulting list is as shown in Figure 18-34.

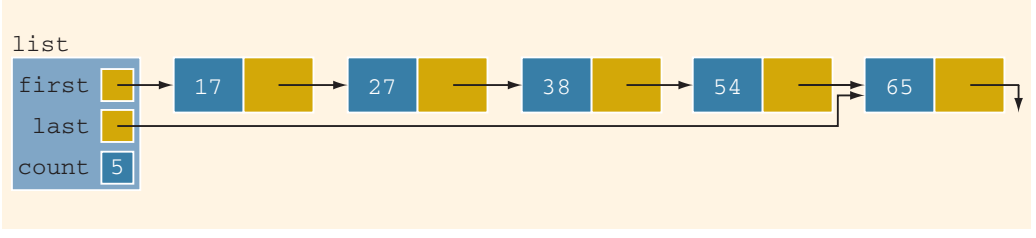


FIGURE 18-34 `list` after inserting `65`

Doubly Linked Lists

A doubly linked list is a linked list in which every node has a next pointer and a back pointer. In other words, every node contains the address of the next node (except the last node), and every node contains the address of the previous node (except the first node) (see Figure 18-39).

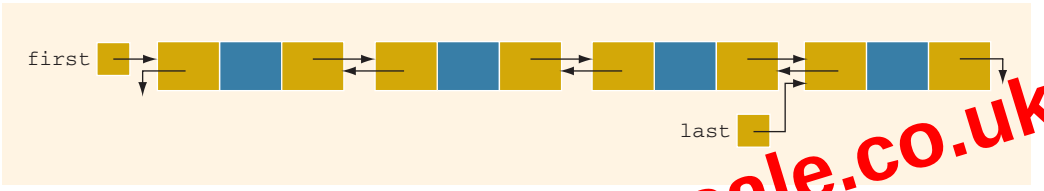


FIGURE 18-39 Doubly linked list

A doubly linked list can be traversed in either direction. That is, we can traverse the list starting at the first node or, if a pointer to the last node is given, we can traverse the list starting at the last node.

As before, the typical operations on a doubly linked list are:

1. Initialize the list.
2. Destroy the list.
3. Determine whether the list is empty.
4. Search the list for a given item.
5. Retrieve the first element of the list.
6. Retrieve the last element of the list.
7. Insert an item in the list.
8. Delete an item from the list.
9. Find the length of the list.
10. Print the list.
11. Make a copy of the doubly linked list.

Next, we describe these operations for an ordered doubly linked list. The following class defines a doubly linked list as an ADT.

```
  //Definition of the node
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *next;
    nodeType<Type> *back;
};
```

```
template <class Type>
class doublyLinkedList
{
public:
    const doublyLinkedList<Type>& operator=
                          (const doublyLinkedList<Type> &);
      //Overload the assignment operator.

    void initializeList();
      //Function to initialize the list to an empty state.
      //Postcondition: first = NULL; last = NULL; count = 0;

    bool isEmptyList() const;
      //Function to determine whether the list is empty.
      //Postcondition: Returns true if the list is empty,
      //               otherwise returns false.

    void destroy();
      //Function to delete all the nodes from the list
      //Postcondition: first = NULL; last = NULL; count = 0;

    void print() const;
      //Function to output the info contained in each node.

    void reversePrint() const;
      //Function to output the info contained in each node
      //in reverse order.

    int length() const;
      //Function to return the number of nodes in the list.
      //Postcondition: The value of count is returned.

    Type front() const;
      //Function to return the first element of the list.
      //Precondition: The list must exist and must not be empty.
      //Postcondition: If the list is empty, the program
      //               terminates; otherwise, the first
      //               element of the list is returned.

    Type back() const;
      //Function to return the last element of the list.
      //Precondition: The list must exist and must not be empty.
      //Postcondition: If the list is empty, the program
      //               terminates; otherwise, the last
      //               element of the list is returned.

    bool search(const Type& searchItem) const;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is found in
      //               the list, otherwise returns false.
```

of the functions `copyList`, the copy constructor, overloading the assignment operator, and the destructor are left as exercises for you. (See Programming Exercise 11 at the end of this chapter.) Moreover, the function `copyList` is used only to implement the copy constructor and overload the assignment operator.

Default Constructor

The default constructor initializes the doubly linked list to an empty state. It sets `first` and `last` to `NULL` and `count` to `0`.

```
template <class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
    first= NULL;
    last = NULL;
    count = 0;
}
```


isEmptyList

This operation returns `true` if the list is empty, otherwise, it returns `false`. The list is empty if the pointer `first` is `NULL`.

```
template <class Type>
bool doublyLinkedList<Type>::isEmptyList() const
{
    return (first == NULL);
}
```

Destroy the List

This operation deletes all of the nodes in the list, leaving the list in an empty state. We traverse the list starting at the first node and then delete each node. Furthermore, `count` is set to `0`.

```
template <class Type>
void doublyLinkedList<Type>::destroy()
{
    nodeType<Type> *temp; //pointer to delete the node

    while (first != NULL)
    {
        temp = first;
        first = first->next;
        delete temp;
    }

    last = NULL;
    count = 0;
}
```

```
    //Postcondition: videoTitle = title; movieStar1 = star1;
    //     movieStar2 = star2; movieProducer = producer;
    //     movieDirector = director;
    //     movieProductionCo = productionCo;
    //     copiesInStock = setInStock;

  int getNoOfCopiesInStock() const;
    //Function to check the number of copies in stock.
    //Postcondition: The value of copiesInStock is returned.

  void checkOut();
    //Function to rent a video.
    //Postcondition: The number of copies in stock is
    //               decremented by one.

  void checkIn();
    //Function to check in a video
    //Postcondition: The number of copies in stock is
    //               incremented by one.

  void printTitle() const;
    //Function to print the title of a movie.

  void printInfo() const;
    //Function to print the details of a video.
    //Postcondition: The title of the movie, stars,
    //               director, and so on are displayed
    //               on the screen.

  bool checkTitle(string title);
    //Function to check whether the title is the same as the
    //title of the video.
    //Postcondition: Returns the value true if the title
    //               is the same as the title of the video;
    //               false otherwise.

  void updateInStock(int num);
    //Function to increment the number of copies in stock by
    //adding the value of the parameter num.
    //Postcondition: copiesInStock = copiesInStock + num;

  void setCopiesInStock(int num);
    //Function to set the number of copies in stock.
    //Postcondition: copiesInStock = num;

  string getTitle() const;
    //Function to return the title of the video.
    //Postcondition: The title of the video is returned.

  videoType(string title = "", string star1 = "",
            string star2 = "", string producer = "",
```

Now:

```
current->info
```

refers to the `info` part of the node. Suppose that we want to know whether the title of the video stored in this node is the same as the title specified by the variable `title`. The expression:

```
current->info.checkTitle(title)
```

is `true` if the title of the video stored in this node is the same as the title specified by the parameter `title`, and `false` otherwise. (Note that the member function `checkTitle` is a value-returning function. See its declaration in the `class videoType`.)

As another example, suppose that we want to set `copiesInStock` of this node to 10. Because `copiesInStock` is a `private` member, it cannot be accessed directly. Therefore, the statement:

```
current->info.copiesInStock = 10;  //illegal
```

is incorrect and will generate a compile-time error. We have to use the member function `setCopiesInStock` as follows:

```
current->info.setCopiesInStock(10);
```


Now that we know how to access a member variable of a video stored in a node, let us describe the algorithm to search the video list.

```
while (not found)
    if the title of the current video is the same as the desired
        title, stop the search
    else
        check the next node
```

The following function definition performs the desired search.

```
void videoListType::searchVideoList(string title, bool& found,
                          nodeType<videoType>* &current) const
{
    found = false;   //set found to false

    current = first; //set current to point to the first node
                     //in the list

    while (current != NULL && !found)     //search the list
        if (current->info.checkTitle(title)) //the item is found
            found = true;
        else
            current = current->link; //advance current to
                                     //the next node
}//end searchVideoList
```

list of videos owned by the video store. The data in the input file is in the following form:

```
video title (that is, the name of the movie)
movie star1
movie star2
movie producer
movie director
movie production co.
number of copies
.
.
.
```

We will write a function, `createVideoList`, to read the data from the input file and create the list of videos. We will also write a function `displayMenu`, to show the different choices—such as check in a movie or check out a movie—that the user can make. The algorithm of the function `main` is:

1. Open the input file.
   If the input file does not exist, exit the program.
2. Create the list of videos (`createVideoList`).
3. Show the menu (`displayMenu`).
4. While not done
   Perform various operations.

Opening the input file is straightforward. Let us describe Steps 2 and 3, which are accomplished by writing two separate functions: `createVideoList` and `displayMenu`.

createVideoList

This function reads the data from the input file and creates a linked list of videos. Because the data will be read from a file and the input file was opened in the function `main`, we pass the input file pointer to this function. We also pass the video list pointer, declared in the function `main`, to this function. Both parameters are reference parameters. Next, we read the data for each video and then insert the video in the list. The general algorithm is:

a. Read the data and store it in a video object.
b. Insert the video in the list.
c. Repeat steps a and b for each video's data in the file.

displayMenu

This function informs the user what to do. It contains the following output statements:

Select one of the following:

1. To check whether the store carries a particular video
2. To check out a video

```
    //process the requests
while (choice != 9)
{
    switch (choice)
    {
    case 1:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (videoList.videoSearch(title))
            cout << "The store carries " << title
                 << endl;
        else
            cout << "The store does not carry "
                 << title << endl;
        break;

    case 2:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (videoList.videoSearch(title))
        {
            if (videoList.isVideoAvailable(title))
            {
                videoList.videoCheckOut(title);
                cout << "Enjoy your movie: "
                     << title << endl;
            }
            else
                cout << "Currently " << title
                     << " is out of stock." << endl;
        }
        else
            cout << "The store does not carry "
                 << title << endl;
        break;

    case 3:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (videoList.videoSearch(title))
        {
            videoList.videoCheckIn(title);
            cout << "Thanks for returning "
                 << title << endl;
        }
```

```
            else
                cout << "The store does not carry "
                     << title << endl;
            break;

        case 4:
            cout << "Enter the title: ";
            getline(cin, title);
            cout << endl;

            if (videoList.videoSearch(title))
            {
                if (videoList.isVideoAvailable(title))
                    cout << title << " is currently in "
                         << "stock." << endl;
                else
                    cout << title << " is currently out "
                         << "of stock." << endl;
            }
            else
                cout << "The store does not carry "
                     << title << endl;
            break;

        case 5:
            videoList.videoPrintTitle();
            break;

        case 6:
            videoList.print();
            break;

        default:
            cout << "Invalid selection." << endl;
        }//end switch

        displayMenu();      //display menu

        cout << "Enter your choice: ";
        cin >> choice;      //get the next request
        cin.get(ch);
        cout << endl;
    }//end while

    return 0;
}
```

This chapter discusses two very useful data structures, stacks and queues. Both stacks and queues have numerous applications in computer science.

Stacks

Suppose that you have a program with several functions. To be specific, suppose that you have functions A, B, C, and D in your program. Now suppose that function A calls function B, function B calls function C, and function C calls function D. When function D terminates, control goes back to function C; when function C terminates, control goes back to function B; and when function B terminates, control goes back to function A. During program execution, how do you think the computer keeps track of the function calls? What about recursive functions? How does the computer keep track of the recursive calls? In Chapter 18, we designed a recursive function to print a linked list backward. What if you want to write a nonrecursive algorithm to print a linked list backward?

This section discusses the data structure called the **stack**, which the computer uses to implement function calls. You can also use stacks to convert recursive algorithms into nonrecursive algorithms, especially recursive algorithms that are not tail recursive. Stacks have numerous applications in computer science. After developing the tools necessary to implement a stack, we will examine some applications of stacks.

A stack is a list of homogeneous elements in which the addition and deletion of elements occur only at one end, called the **top** of the stack. For example, in a cafeteria, the second tray in a stack of trays can be removed only if the first tray has been removed. For another example, to get to your favorite computer science book, which is underneath your math and history books, you must first remove the math and history books. After removing these books, the computer science book becomes the top book—that is, the top element of the stack. Figure 19-1 shows some examples of stacks.

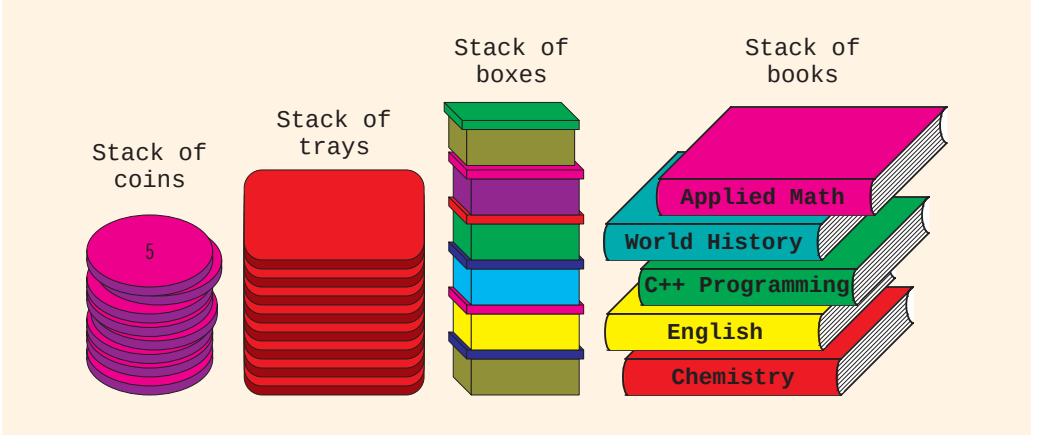


FIGURE 19-1 Various types of stacks

Implementation of Stacks as Arrays

Because all of the elements of a stack are of the same type, you can use an array to implement a stack. The first element of the stack can be put in the first array slot, the second element of the stack in the second array slot, and so on. The top of the stack is the index of the last element added to the stack.

In this implementation of a stack, stack elements are stored in an array, and an array is a random access data structure; that is, you can directly access any element of the array. However, by definition, a stack is a data structure in which the elements are accessed (popped or pushed) at only one end—that is, a Last In First Out data structure. Thus, a stack element is accessed only through the top, not through the bottom or middle. This feature of a stack is extremely important and must be recognized in the beginning.

To keep track of the top position of the array, we can implicitly declare another variable called `stackTop`.

The following `class`, `stackType`, implements the functions of the abstract `class` `stackADT`. By using a pointer, we can dynamically allocate arrays, so we will leave it for the user to specify the size of the array (that is, the stack size). We assume that the default stack size is 100. Because the `class` `stackType` has a pointer member variable (the pointer to the array to store the stack elements), we must overload the assignment operator and include the copy constructor and destructor. Moreover, we give a generic definition of the stack. Depending on the specific application, we can pass the stack element type when we declare a stack object.


```
template <class Type>
class stackType: public stackADT<Type>
{
public:
    const stackType<Type>& operator=(const stackType<Type>&);
      //Overload the assignment operator.

    void initializeStack();
      //Function to initialize the stack to an empty state.
      //Postcondition: stackTop = 0;

    bool isEmptyStack() const;
      //Function to determine whether the stack is empty.
      //Postcondition: Returns true if the stack is empty,
      //               otherwise returns false.

    bool isFullStack() const;
      //Function to determine whether the stack is full.
      //Postcondition: Returns true if the stack is full,
      //               otherwise returns false.

    void push(const Type& newItem);
      //Function to add newItem to the stack.
      //Precondition: The stack exists and is not full.
      //Postcondition: The stack is changed and newItem
      //               is added to the top of the stack.
```

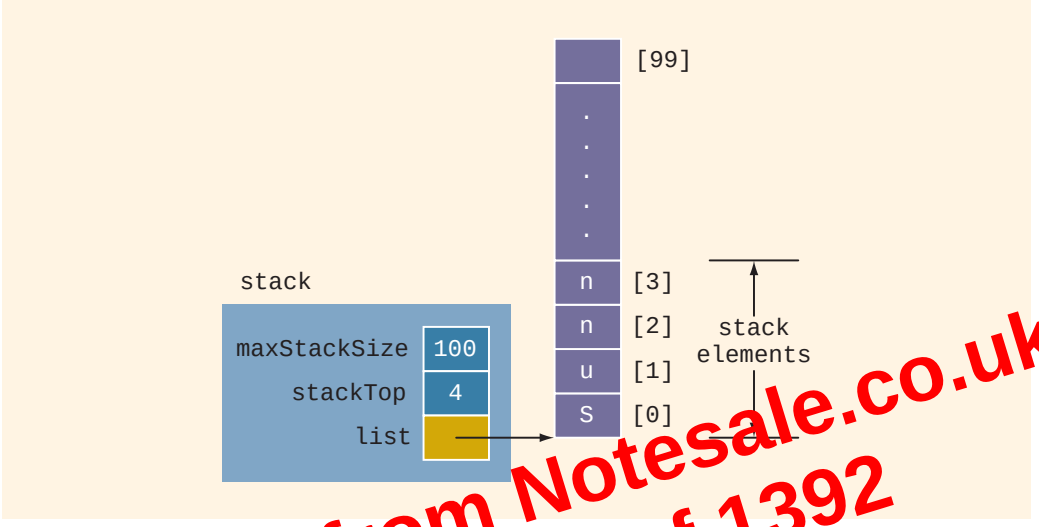


FIGURE 19-8 Stack before pushing `y`

Assume `newItem` is `'y'`. After the `push` operation, the stack is as shown in Figure 19-9.

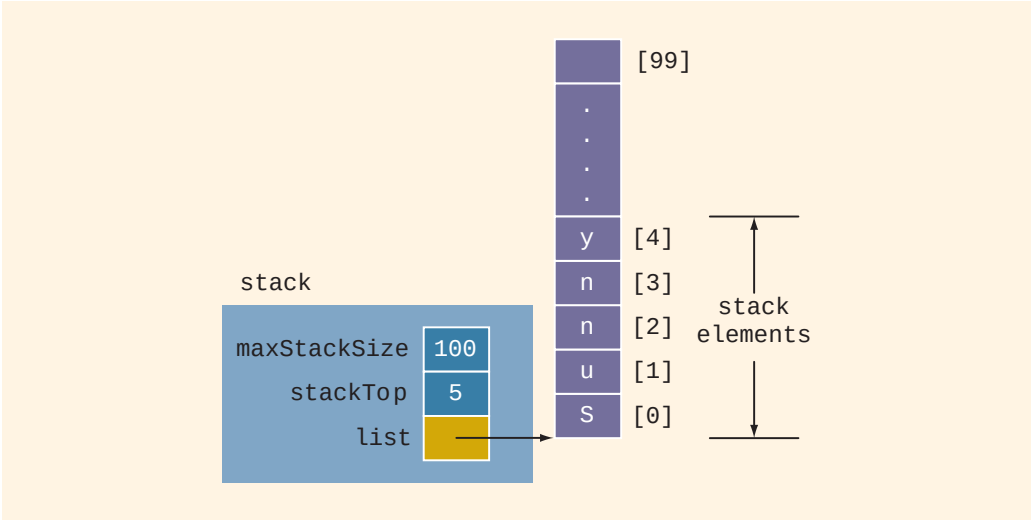


FIGURE 19-9 Stack after pushing `y`

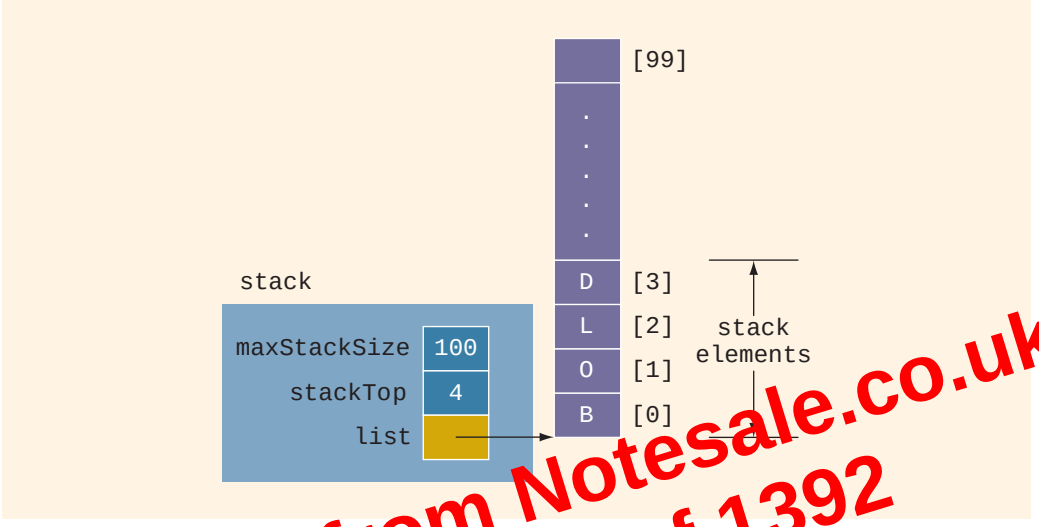


FIGURE 19-10 Stack before popping D

After the pop operation, the stack is as shown in Figure 19-11.

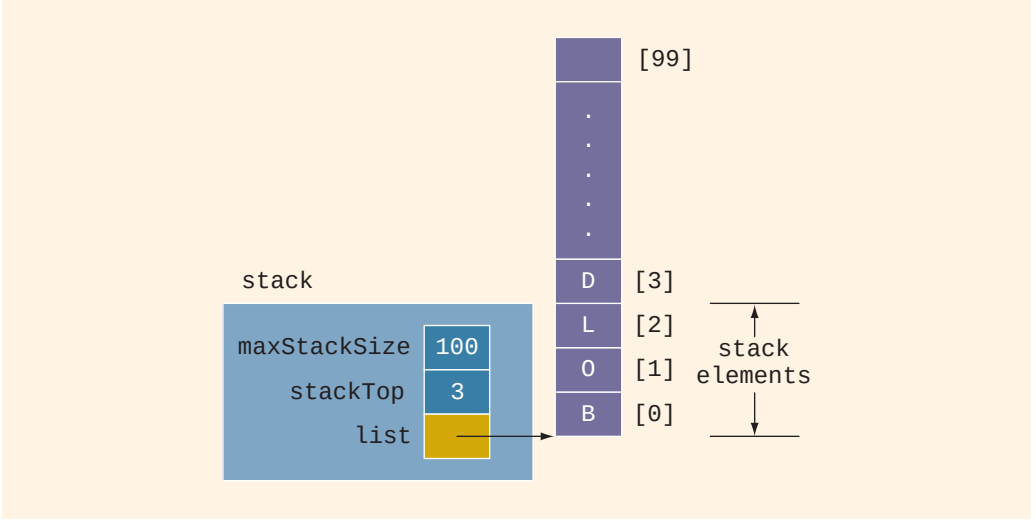


FIGURE 19-11 Stack after popping D

Output The highest GPA and all of the names associated with the highest GPA. For example, for the above data, the highest GPA is `3.9`, and the students with that GPA are `Kathy` and `David`.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

We read the first GPA and the name of the student. Because this data is the first item read, it is the highest GPA so far. Next, we read the second GPA and the name of the student. We then compare this (second) GPA with the highest GPA so far. Three cases arise:

1. The new GPA is greater than the highest GPA so far. In this case, we:
   a. Update the value of the highest GPA so far.
   b. Initialize the stack—that is, remove the names of the students from the stack.
   c. Save the name of the student having the highest GPA so far in the stack.
2. The new GPA is equal to the highest GPA so far. In this case, we add the name of the new student to the stack.
3. The new GPA is smaller than the highest GPA so far. In this case, we discard the name of the student having this grade.

We then read the next GPA and the name of the student and repeat Steps 1 through 3. We continue this process until we reach the end of the input file.

From this discussion, it is clear that we need the following variables:

```
double GPA;          //variable to hold the current GPA
double highestGPA;   //variable to hold the highest GPA
string name;         //variable to hold the name of the student
stackType<string> stack(100); //object to implement the stack
```

The preceding discussion translates into the following algorithm:

1. Declare the variables and initialize stack.
2. Open the input file.
3. If the input file does not exist, exit the program.
4. Set the output of the floating-point numbers to a fixed decimal format with a decimal point and trailing zeroes. Also, set the precision to two decimal places.
5. Read the GPA and the student name.
6. `highestGPA = GPA;`

```
infile.open("HighestGPAData.txt");          //Step 2

if (!infile)                                //Step 3
{
    cout << "The input file does not "
         << "exist. Program terminates!"
         << endl;
    return 1;
}

cout << fixed << showpoint;                 //Step 4
cout << setprecision(2);                    //Step 4

infile >> GPA >> name;                      //Step 5

highestGPA = GPA;                           //Step 6

while (infile)                              //Step 7
{
    if (GPA > highestGPA)                   //Step 7.1
    {
        stack.initializeStack();            //Step 7.1.1

        if (!stack.isFullStack())           //Step 7.1.2
            stack.push(name);

        highestGPA = GPA;                   //Step 7.1.3
    }
    else if (GPA == highestGPA)             //Step 7.2
        if (!stack.isFullStack())
            stack.push(name);
        else
        {
            cout << "Stack overflows. "
                 << "Program terminates!"
                 << endl;
            return 1;  //exit program
        }
    infile >> GPA >> name;                  //Step 7.3
}

cout << "Highest GPA = " << highestGPA
     << endl;                               //Step 8
cout << "The students holding the "
     << "highest GPA are:" << endl;

while (!stack.isEmptyStack())               //Step 9
{
    cout << stack.top() << endl;
    stack.pop();
}
```

```
{
    return (stackTop == NULL);
} //end isEmptyStack

template <class Type>
bool linkedStackType<Type>:: isFullStack() const
{
    return false;
} //end isFullStack
```

Recall that in the linked implementation of stacks, the function `isFullStack` does not apply because, logically, the stack is never full. However, you must provide its definition because it is included as an abstract function in the parent `class` `stackADT`.

Initialize Stack

The operation `initializeStack` reinitializes the stack to an empty state. Because the stack may contain some elements and we are using a linked implementation of a stack, we must deallocate the memory occupied by the stack elements and set `stackTop` to `NULL`. The definition of this function is:

```
template <class Type>
void linkedStackType<Type>:: initializeStack()
{
    nodeType<Type> *temp; //pointer to delete the node

    while (stackTop != NULL)  //while there are elements in
                              //the stack
    {
        temp = stackTop;     //set temp to point to the
                             //current node
        stackTop = stackTop->link;  //advance stackTop to the
                                    //next node
        delete temp;     //deallocate memory occupied by temp
    }
} //end initializeStack
```

Next, we consider the `push`, `top`, and `pop` operations. From Figure 19-12(b), it is clear that the `newElement` will be added (in the case of `push`) at the beginning of the linked list pointed to by `stackTop`. In the case of `pop`, the node pointed to by `stackTop` will be removed. In both cases, the value of the pointer `stackTop` is updated. The operation `top` returns the `info` of the node that `stackTop` is pointing to.

Push

Consider the stack shown in Figure 19-13.

```
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

    return *this;
}//end operator=
```

The definition of a stack and the functions to implement the stack operations discussed previously are generic. Also, as in the case of an array representation of a stack, in the linked representation of a stack, we must put the definition of the stack and the functions to implement the stack operations together in a (header) file. A client's program can include this header file via the `include` statement.

Example 19-3 illustrates how a `linkedStack` object is used in a program.

EXAMPLE 19-3

We assume that the definition of the class `linkedStackType` and the functions to implement the stack operations are included in the header file `"linkedStack.h"`.

```
//This program tests various operations of a linked stack
#include <iostream>
#include "linkedStack.h"

using namespace std;

void testCopy(linkedStackType<int> OStack);

int main()
{
    linkedStackType<int> stack;
    linkedStackType<int> otherStack;
    linkedStackType<int> newStack;

        //Add elements into stack
    stack.push(34);
    stack.push(43);
    stack.push(27);

        //Use the assignment operator to copy the elements
        //of stack into newStack
    newStack = stack;

    cout << "After the assignment operator, newStack: "
         << endl;

        //Output the elements of newStack
    while (!newStack.isEmptyStack())
    {
        cout << newStack.top() << endl;
        newStack.pop();
    }
```

```
template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    unorderedLinkedList<Type>::insertFirst(newElement);
}

template <class Type>
Type linkedStackType<Type>::top() const
{
    return unorderedLinkedList<Type>::front();
}

template <class Type>
void linkedStackType<Type>::pop()
{
    nodeType<Type> *temp;

    temp = first;
    first = first->link;
    delete temp;
}
```


Application of Stacks: Postfix Expressions Calculator

The usual notation for writing arithmetic expressions (the notation we learned in elementary school) is called **infix** notation, in which the operator is written between the operands. For example, in the expression $a + b$, the operator + is between the operands a and b. In infix notation, the operators have precedence. That is, we must evaluate expressions from left to right, and multiplication and division have higher precedence than do addition and subtraction. If we want to evaluate the expression in a different order, we must include parentheses. For example, in the expression $a + b * c$, we first evaluate * using the operands b and c, and then we evaluate + using the operand a and the result of $b * c$.

In the early 1920s, the Polish mathematician Jan Lukasiewicz discovered that if operators were written before the operands (**prefix** or **Polish** notation; for example, $+ a b$), the parentheses could be omitted. In the late 1950s, the Australian philosopher and early computer scientist Charles L. Hamblin proposed a scheme in which the operators *follow* the operands (postfix operators), resulting in the **Reverse Polish** notation. This has the advantage that the operators appear in the order required for computation.

For example, the expression:

$a + b * c$

in a postfix expression is:

$a\ b\ c * +$

The following example shows various infix expressions and their equivalent postfix expressions.

EXAMPLE 19-4

Infix Expression	Equivalent Postfix Expression
$a + b$	$a\ b\ +$
$a + b * c$	$a\ b\ c\ * +$
$a * b + c$	$a\ b\ *\ c +$
$(a + b) * c$	$a\ b +\ c\ *$
$(a - b) * (c + d)$	$a\ b -\ c\ d +\ *$
$(a + b) * (c - d / e) + f$	$a\ b +\ c\ d\ e / -\ *\ f +$

Shortly after Lukasiewicz's discovery, it was realized that postfix notation had important applications in computer science. In fact, many compilers now first translate arithmetic expressions into some form of postfix notation and then translate this postfix expression into machine code. Postfix expressions can be evaluated using the following algorithm:

Scan the expression from left to right. When an operator is found, back up to get the required number of operands, perform the operation, and continue.

Consider the following postfix expression:

```
6 3 + 2 * =
```

Let us evaluate this expression using a stack and the previous algorithm. Figure 19-17 shows how this expression gets evaluated.

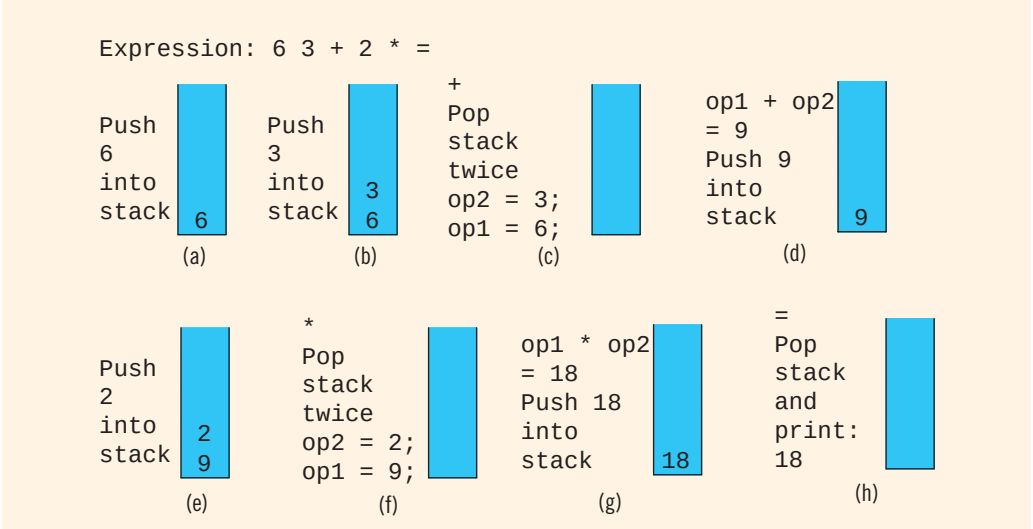


FIGURE 19-17 Evaluating the postfix expression: 6 3 + 2 * =

```
    if no error was found, then
    {
        read next ch;
        output ch;
    }
    else
        Discard the expression
} //end while
```

From this algorithm, it follows that this method has five parameters—one to access the input file, one to access the output file, one to access the stack, one to pass a character of the expression, and one to indicate whether there is an error in the expression. The definition of this function is:

```
void evaluateExpression(ifstream& inpF, ofstream& outF,
                        stackType<double>& stack,
                        char& ch, bool& isExpOk)
{
    double num;

    while (ch != '=')
    {
        switch (ch)
        {
        case '#':
            inpF >> num;
            outF << num << " ";
            if (!stack.isFullStack())
                stack.push(num);
            else
            {
                cout << "Stack overflow. "
                     << "Program terminates!" << endl;
                exit(0);  //terminate the program
            }

            break;
        default:
            evaluateOpr(outF, stack, ch, isExpOk);
        }//end switch

        if (isExpOk) //if no error
        {
            inpF >> ch;
            outF << ch;

            if (ch != '#')
                outF << " ";
        }
        else
            discardExp(inpF, outF, ch);
    } //end while (!= '=')
}
```

Because the array containing the queue is circular, we can use the following statement to advance `queueRear` (`queueFront`) to the next array position.

```
queueRear = (queueRear + 1) % maxQueueSize;
```

If `queueRear < maxQueueSize - 1`, then `queueRear + 1 <= maxQueueSize - 1`, so `(queueRear + 1) % maxQueueSize = queueRear + 1`. If `queueRear == maxQueueSize - 1` (that is, `queueRear` points to the last array position), `queueRear + 1 == maxQueueSize`, so `(queueRear + 1) % maxQueueSize = 0`. In this case, `queueRear` will be set to `0`, which is the first array position.

This queue design seems to work well. Before we write the algorithms to implement the queue operations, consider the following two cases.

Case 1: Suppose that after certain operations, the array containing the queue is as shown in Figure 19-32(a).

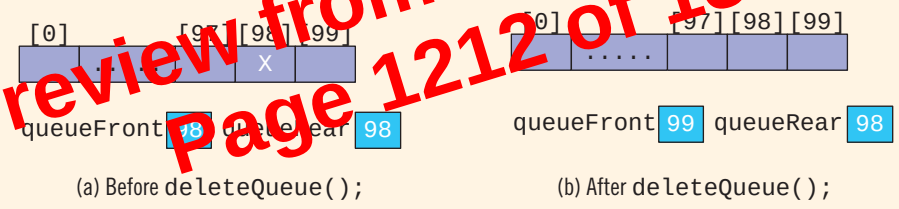


FIGURE 19-32 Queue before and after the delete operation

After the operation `deleteQueue();`, the resulting array is as shown in Figure 19-32(b).

Case 2: Let us now consider the queue shown in Figure 19-33(a).

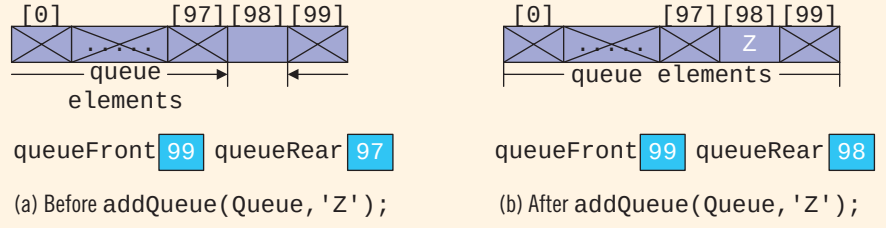


FIGURE 19-33 Queue before and after the add operation

After the operation `addQueue(Queue,'Z');`, the resulting array is as shown in Figure 19-33(b).

```
    void initializeQueue();
      //Function to initialize the queue to an empty state.
      //Postcondition: queueFront = NULL; queueRear = NULL

    Type front() const;
      //Function to return the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //               terminates; otherwise, the first
      //               element of the queue is returned.

    Type back() const;
      //Function to return the last element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //               terminates; otherwise, the last
      //               element of the queue is returned.

    void addQueue(const Type& queueElement);
      //Function to add queueElement to the queue.
      //Precondition: The queue exists and is not full.
      //Postcondition: The queue is changed and queueElement
      //               is added to the queue.

    void deleteQueue();
      //Function to remove the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: The queue is changed and the first
      //               element is removed from the queue.

    linkedQueueType();
      //Default constructor

    linkedQueueType(const linkedQueueType<Type>& otherQueue);
      //Copy constructor

    ~linkedQueueType();
      //Destructor

private:
    nodeType<Type> *queueFront; //pointer to the front of
                                //the queue
    nodeType<Type> *queueRear;  //pointer to the rear of
                                //the queue
};
```

The UML class diagram of the class linkedQueueType is left as an exercise for you. (See Exercise 29 at the end of this chapter.)

Next, we write the definitions of the functions of the class linkedQueueType.

EMPTY AND FULL QUEUE

The queue is empty if queueFront is NULL. Memory to store the queue elements is allocated dynamically. Therefore, the queue is never full, so the function to implement

the isFullQueue operation returns the value false. (The queue is full only if we run out of memory.)

```
template <class Type>
bool linkedQueueType<Type>::isEmptyQueue() const
{
    return (queueFront == NULL);
} //end

template <class Type>
bool linkedQueueType<Type>::isFullQueue() const
{
    return false;
} //end isFullQueue
```

Note that in reality, in the linked implementation of queues, the function isFullQueue does not apply because, logically, the queue is never full. However, you must provide its definition because it is included as an abstract function in the parent class queueADT.

INITIALIZE QUEUE

The operation initializeQueue initializes the queue to an empty state. The queue is empty if there are no elements in the queue. Note that the constructor initializes the queue when the queue variable is declared. So this operation must remove all of the elements, if any, from the queue. Therefore, this operation traverses the list containing the queue starting at the first node, and it deallocates the memory occupied by the queue elements. The definition of this function is:

```
template <class Type>
void linkedQueueType<Type>::initializeQueue()
{
    nodeType<Type> *temp;

    while (queueFront!= NULL)  //while there are elements left
                               //in the queue
    {
        temp = queueFront;  //set temp to point to the
                            //current node
        queueFront = queueFront->link;  //advance first to
                                        //the next node
        delete temp;    //deallocate memory occupied by temp
    }

    queueRear = NULL;  //set rear to NULL
} //end initializeQueue
```

addQueue, front, back, AND deleteQueue OPERATIONS

The addQueue operation adds a new element at the end of the queue. To implement this operation, we access the pointer queueRear.

```
customerType::customerType(int customerN, int arrvTime,
                           int wTime, int tTime)
{
    setCustomerInfo(customerN, arrvTime, wTime, tTime);
}
```

The function `getWaitingTime` returns the current waiting time. The definition of the function `getWaitingTime` is:

```
int customerType::getWaitingTime() const
{
    return waitingTime;
}
```

The function `incrementWaitingTime` increments the value of `waitingTime`. Its definition is:

```
void customerType::incrementWaitingTime()
{
    waitingTime++;
}
```

The definitions of the functions `setWaitingTime`, `getArrivalTime`, `getTransactionTime`, and `setCustomerNumber` are left as an exercise for you.

Server

At any given time unit, the server is either busy serving a customer or is free. We use a `string` variable to set the status of the server. Every server has a timer and, because the program might need to know which customer is served by which server, the server also stores the information of the customer being served. Thus, three member variables are associated with a server: the `status`, the `transactionTime`, and the `currentCustomer`. Some of the basic operations that must be performed on a server are as follows: check whether the server is free; set the server as free; set the server as busy; set the transaction time (that is, how long it takes to serve the customer); return the remaining transaction time (to determine whether the server should be set to free); if the server is busy after each time unit, decrement the transaction time by one time unit; and so on. The following `class`, `serverType`, implements the server as an ADT.

```
class serverType
{
public:
    serverType();
      //Default constructor
      //Sets the values of the instance variables to their default
      //values.
      //Postcondition: currentCustomer is initialized by its
      //               default constructor; status = "free"; and
      //               the transaction time is initialized to 0.
```

```
bool isFree() const;
  //Function to determine if the server is free.
  //Postcondition: Returns true if the server is free,
  //               otherwise returns false.

void setBusy();
  //Function to set the status of the server to busy.
  //Postcondition: status = "busy";

void setFree();
  //Function to set the status of the server to "free".
  //Postcondition: status = "free";

void setTransactionTime(int t);
  //Function to set the transaction time according to the
  //parameter t.
  //Postcondition: transactionTime = t;

void setTransactionTime();
  //Function to set the transaction time according to
  //the transaction time of the current customer.
  //Postcondition:
  //  transactionTime = currentCustomer.transactionTime;

int getRemainingTransactionTime() const;
  //Function to return the remaining transaction time.
  //Postcondition: The value of transactionTime is returned.

void decreaseTransactionTime();
  //Function to decrease the transactionTime by one unit.
  //Postcondition: transactionTime--;

void setCurrentCustomer(customerType cCustomer);
  //Function to set the info of the current customer
  //according to the parameter cCustomer.
  //Postcondition: currentCustomer = cCustomer;

int getCurrentCustomerNumber() const;
  //Function to return the customer number of the current
  //customer.
  //Postcondition: The value of customerNumber of the
  //               current customer is returned.

int getCurrentCustomerArrivalTime() const;
  //Function to return the arrival time of the current
  //customer.
  //Postcondition: The value of arrivalTime of the current
  //               customer is returned.

int getCurrentCustomerWaitingTime() const;
  //Function to return the current waiting time of the
  //current customer.
```

The `addQueue` operation inserts the element at the end of the queue. If we perform the `deleteQueue` operation followed by the `addQueue` operation for each element of the queue, then eventually the front element again becomes the front element. Given that each `deleteQueue` operation is followed by an `addQueue` operation, how do we determine that all of the elements of the queue have been processed? We cannot use the `isEmptyQueue` or `isFullQueue` operations on the queue, because the queue will never be empty or full.

One solution to this problem is to create a temporary queue. Every element of the original queue is removed, processed, and inserted into the temporary queue. When the original queue becomes empty, all of the elements in the queue are processed. We can then copy the elements from the temporary queue back into the original queue. However, this solution requires us to use extra memory space, which could be significant. Also, if the queue is large, extra computer time is needed to copy the elements from the temporary queue back into the original queue. Let us look into another solution.

In the second solution, before starting to update the elements of the queue, we can insert a dummy customer with a wait time of, say, -1. During the update process, when we arrive at the customer with the wait time of -1, we can stop the update process without processing the customer with the wait time of -1. If we do not process the customer with the wait time of -1, this customer is removed from the queue and, after processing all of the elements of the queue, the queue will contain no extra elements. This solution does not require us to create a temporary queue, so we do not need extra computer time to copy the elements back into the original queue. We will use this solution to update the queue. Therefore, the definition of the function `updateWaitingQueue` is:

```
void waitingCustomerQueueType::updateWaitingQueue()
{
    customerType cust;

    cust.setWaitingTime(-1);
    int wTime = 0;

    addQueue(cust);

    while (wTime != -1)
    {
        cust = front();
        deleteQueue();

        wTime = cust.getWaitingTime();
        if (wTime == -1)
            break;
        cust.incrementWaitingTime();
        addQueue(cust);
    }
}
```

3. Set the free server to begin the transaction.

```
serverList.setServerBusy(serverID, customer, transTime);
```

To run the simulation, we need to know the number of customers arriving at a given time unit and how long it takes to serve the customer. We use the Poisson distribution from statistics, which says that the probability of y events occurring at a given time is given by the formula:

$$P(y) = \frac{\lambda^y e^{-\lambda}}{y!}, y = 0, 1, 2, \ldots,$$

in which λ is the expected value that y events occur at that time. Suppose that, on average, a customer arrives every four minutes. During this four-minute period, the customer can arrive at any one of the four minutes. Assuming an equal likelihood of each of the four minutes, the expected value that a customer arrives in each of the four minutes is, therefore, 1 / 4 = .25. Next, we need to determine whether or not the customer actually arrives at a given minute.

Now, $P(0) = e^{-\lambda}$ is the probability that no event occurs at a given time. One of the basic assumptions of the Poisson distribution is that the probability of more than one outcome occurring in a short time interval is negligible. For simplicity, we assume that only one customer arrives at a given time unit. Thus, we use $e^{-\lambda}$ as the cutoff point to determine whether a customer arrives at a given time unit. Suppose that, on average, a customer arrives every four minutes. Then, $\lambda = 0.25$. We can use an algorithm to generate a number between 0 and 1. If the value of the number generated is $> e^{-0.25}$, we can assume that the customer arrived at a particular time unit. For example, suppose that *rNum* is a random number such that $0 \leq rNum \leq 1$. If $rNum > e^{-0.25}$, the customer arrived at the given time unit.

We now describe the function `runSimulation` to implement the simulation. Suppose that we run the simulation for 100 time units and customers arrive at time units 93, 96, and 100. The average transaction time is five minutes—that is, five time units. For simplicity, assume that we have only one server and that the server becomes free at time unit 97, and that all customers arriving before time unit 93 have been served. When the server becomes free at time unit 97, the customer arriving at time unit 93 starts the transaction. Because the transaction of the customer arriving at time unit 93 starts at time unit 97 and it takes five minutes to complete a transaction, when the simulation loop ends, the customer arriving at time unit 93 is still at the server. Moreover, customers arriving at time units 96 and 100 are in the queue. For simplicity, we assume that when the simulation loop ends, the customers at the servers are considered served. The general algorithm for this function is:

1. Declare and initialize the variables, such as the simulation parameters, customer number, clock, total and average waiting times, number of customers arrived, number of customers served, number of customers left in the waiting queue, number of customers left with the servers, `waitingCustomersQueue`, and a list of servers.

```
Customer number 3 arrived at time unit 9
Customer number 4 arrived at time unit 12
From server number 2 customer number 2
     departed at time unit 13
From server number 1 customer number 3
     departed at time unit 14
From server number 2 customer number 4
     departed at time unit 18
Customer number 5 arrived at time unit 21
From server number 1 customer number 5
     departed at time unit 26
Customer number 6 arrived at time unit 37
Customer number 7 arrived at time unit 38
Customer number 8 arrived at time unit 41
From server number 1 customer number 6
     departed at time unit 42
From server number 2 customer number 7
     departed at time unit 43
Customer number 9 arrived at time unit 43
Customer number 10 arrived at time unit 44
From server number 1 customer number 8
     departed at time unit 47
From server number 2 customer number 9
     departed at time unit 48
Customer number 11 arrived at time unit 49
Customer number 12 arrived at time unit 51
From server number 1 customer number 10
     departed at time unit 52
Customer number 13 arrived at time unit 52
Customer number 14 arrived at time unit 53
From server number 2 customer number 11
     departed at time unit 54
Customer number 15 arrived at time unit 54
From server number 1 customer number 12
     departed at time unit 57
From server number 2 customer number 13
     departed at time unit 59
Customer number 16 arrived at time unit 59
From server number 1 customer number 14
     departed at time unit 62
From server number 2 customer number 15
     departed at time unit 64
Customer number 17 arrived at time unit 66
From server number 1 customer number 16
     departed at time unit 67
From server number 2 customer number 17
     departed at time unit 71
Customer number 18 arrived at time unit 71
From server number 1 customer number 18
     departed at time unit 76
Customer number 19 arrived at time unit 78
From server number 1 customer number 19
     departed at time unit 83
Customer number 20 arrived at time unit 90
Customer number 21 arrived at time unit 92
From server number 1 customer number 20
     departed at time unit 95
```

```
    for (int i = 0; i < 7; i++)
        s1.push(list[i]);

    mystery(s1, s2);

    while (!s2.isEmptyStack())
    {
        cout << s2.top() << " ";
        s2.pop();
    }
    cout << endl;
}

template <class type>
void mystery(stackType<type>& s, stackType<type>& t)
{
    while (!s.isEmptyStack())
    {
        t.push(s.top());
        s.pop();
    }
}
```


9. What is the output of the following program?

```
#include <iostream>
#include <string>
#include "myStack.h"

using namespace std;

void mystery(stackType<int>& s, stackType<int>& t);

int main()
{
    int list[] = {5, 10, 15, 20, 25};

    stackType<int> s1;
    stackType<int> s2;

    for (int i = 0; i < 5; i++)
        s1.push(list[i]);

    mystery(s1, s2);

    while (!s2.isEmptyStack())
    {
        cout << s2.top() << " ";
        s2.pop();
    }
    cout << endl;
}
```

Operator	Associativity
`|`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Right to left
`=  +=  -=  *=  /=  %=`	Right to left
`<<=  >>=  &=  |=  ^=`	Right to left
`throw`	Right to left
`,` (the sequencing operator)	Left to right

This page intentionally left blank

We use the weight of each bit to find the equivalent decimal number. For each bit, we multiply the bit by 2 to the power of its weight and then we add all of the numbers. For the above binary number, the equivalent decimal number is:

$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 64 + 0 + 0 + 8 + 4 + 0 + 1$

$= 77.$

Converting a Binary Number (Base 2) to Octal (Base 8) and Hexadecimal (Base 16)

The previous sections described how to convert a binary number to a decimal number (base 2). Even though the language of a computer is binary, if the binary number is too long, then it will be hard to manipulate it manually. To effectively deal with binary numbers, two more number systems, octal (base 8) and hexadecimal (base 16), are of interest to computer scientists.

The digits in the octal number system are 0, 1, 2, 3, 4, 5, 6, and 7. The digits in the hexadecimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. So A in hexadecimal is 10 in decimal, B in hexadecimal is 11 in decimal, and so on.

The algorithm to convert a binary number into an equivalent number in octal (or hexadecimal) is quite simple. Before we describe the method to do so, let us review some notations. Suppose a_b represents the number a to the base b. For example, $2A0_{16}$ means 2A0 to the base 16, and 63_8 means 63 to the base 8.

First we describe how to convert a binary number into an equivalent octal number and vice versa. Table E-1 describes the first eight octal numbers.

TABLE E-1 Binary representation of first eight octal numbers

Binary	Octal	Binary	Octal
000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

Consider the binary number 1101100010101. To find the equivalent octal number, starting from right to left we consider three digits at a time and write their octal representation. Note that the binary number 1101100010101 has only 13 digits. So when

Function Name and Parameters	Parameter(s) Type	Function Return Value
`strcpy(destStr, srcStr)`	`destStr` and `srcStr` are null-terminated `char` arrays	The base address of `destStr` is returned; `srcStr` is copied into `destStr`
`strlen(str)`	`str` is a null-terminated `char` array	An integer value ≥ 0 specifying the length of the `str` (excluding the `'\0'`) is returned

HEADER FILE `string`

This header file—not to be confused with the header file `cstring`—supplies a programmer-defined data type named `string`. Associated with the `string` type are a data type `string::size_type` and a named constant `string::npos`. These are defined as follows:

`string::size_type`	An unsigned integer type
`string::npos`	The maximum value of type `string::size_type`

The type `string` contains several functions for string manipulation. In addition to the string functions listed in Table 8-1, the following table describes additional string functions. In this table, we assume that `strVar` is a `string` variable and `str` is a string variable, a string constant, or a character array.

Expression	Effect
`getline(istreamVar, strVar);`	`istreamVar` is an input stream variable (of type `istream` or `ifstream`). Characters until the newline character are input from `istreamVar` and stored in `strVar`. (The newline character is read but not stored into `strVar`.) The value returned by this function is usually ignored.
`strVar.append(str, n)`	The first `n` characters of the character array `str` are appended to `strVar`.
`strVar.c_str()`	The base address of a null-terminated C-string corresponding to the characters in `strVar`.

Random Number Generator

To generate a random number, you can use the C++ function `rand`. To use the function `rand`, the program must include the header file `cstdlib`. The header file `cstdlib` also contains the constant `RAND_MAX`. Typically, the value of `RAND_MAX` is `32767`. To find the exact value of `RAND_MAX`, check your system's documentation. The function `rand` generates an integer between `0` and `RAND_MAX`. The following program illustrates how to use the function `rand`. It also prints the value of `RAND_MAX`:

```
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main()
{
    cout << fixed << showpoint << setprecision(5);
      cout << "The value of RAND_MAX: " << RAND_MAX << endl;
    cout << "A random number: " << rand() << endl;
    cout << "A random number between 0 and 9: "
         << rand() % 10 << endl;
    cout << "A random number between 0 and 1: "
         << static_cast<double> (rand())
              / static_cast<double>(RAND_MAX)
         << endl;

    return 0;
}
```


Sample Run:

```
The value of RAND_MAX: 32767
A random number: 41
A random number between 0 and 9: 7
A random number between 0 and 1: 0.19330
```

```
vector<int> intList;
```

declares `intList` to be a vector and the component type is `int`. Similarly, the statement:

```
vector<string> stringList;
```

declares `stringList` to be a `vector` container and the component type is `string`.

DECLARING VECTOR OBJECTS

The `class` `vector` contains several constructors, including the default constructor. Therefore, a vector container can be declared and initialized several ways. Table H-1 describes how a vector container of a specific type can be declared and initialized.

TABLE H-1 Various Ways to Declare and Initialize a Vector Container

Statement	Effect
`vector<elemType> vecList;`	Creates the empty vector container `vecList`. (The default constructor is invoked.)
`vector<elemType> vecList(otherVecList);`	Creates the vector container `vecList`, and initializes `vecList` to the elements of the vector `otherVecList`. `vecList` and `otherVecList` are of the same type.
`vector<elemType> vecList(size);`	Creates the vector container `vecList` of size `size`. `vecList` is initialized using the default constructor.
`vector<elemType> vecList(n, elm);`	Creates the vector container `vecList` of size `n`. `vecList` is initialized using `n` copies of the element `elm`.
`vector<elemType> vecList(beg, end);`	Creates the vector container `vecList`. `vecList` is initialized to the elements in the range `[beg, end)`, that is, all the elements in the range `beg...end-1`. Both `beg` and `end` are pointers, called iterators in STL terminology. (Later in this appendix, we explain how iterators are used.)

TABLE H-5 Operations Common to All Containers (continued)

Member Function	Description
`ct.size()`	Returns the number of elements currently in container `ct`.
`ct.max_size()`	Returns the maximum number of elements that can be inserted in container `ct`.
`ct1.swap(ct2)`	Swaps the elements of containers `ct1` and `ct2`.
`ct.begin()`	Returns an iterator to the first element in container `ct`.
`ct.end()`	Returns an iterator to the position after the last element into container `ct`.
`ct.rbegin()`	Reverse begin. Returns a pointer to the last element into container `ct`. This function is used to process the elements of `ct` in reverse.
`ct.rend()`	Reverse end. Returns a pointer to the position before the first element into container `ct`.
`ct.insert(position, elem)`	Inserts `elem` into container `ct` at the position specified by `position`. Note that here `position` is an iterator.
`ct.erase(beg, end)`	Deletes all the elements between `beg...end-1` from container `ct`. Both `beg` and `end` are iterators.
`ct.clear()`	Deletes all the elements from the container. After a call to this function, container `ct` is empty.
Operator Functions	
`ct1 = ct2;`	Copies the elements of `ct2` into `ct1`. After this operation, the elements in both containers are the same.
`ct1 == ct2`	Returns `true` if containers `ct1` and `ct2` are equal, `false` otherwise.
`ct1 != ct2`	Returns `true` if containers `ct1` and `ct2` are not equal, `false` otherwise.

```
    copy(vecList.begin(), vecList.end(), screen);     //Line 17
    cout << endl;                                     //Line 18

    return 0;
}
```

Sample Run:

```
Line 4: intArray: 5 6 8 3 40 36 98 29 75
Line 8: vecList: 5 6 8 3 40 36 98 29 75
Line 12: After shifting the elements one position to the left,
         intArray: 6 8 3 40 36 98 29 75 75
Line 16: After shifting the elements down by two positions,
         vecList: 5 6 5 6 8 3 40 36 98
```


Sequence Container: deque

This section describes the sequence container `deque`. The term **deque** stands for double-ended queue. `Deque` containers are implemented as dynamic arrays in such a way that the elements can be inserted at both ends. Thus, a `deque` can expand in either direction. Elements can also be inserted in the middle. Inserting elements at the beginning or the end is fast; inserting elements in the middle, however, is time consuming because the elements in the queue need to be shifted.

The name of the class defining the deque containers is `deque`. Also, the definition of the `class deque`, and the functions to implement the various operations on a `deque` object, are contained in the header file `deque`. Therefore, to use a deque container in a program, the program must include the following statement:

```
#include <deque>
```

The `class deque` contains several constructors. Thus, a `deque` object can be initialized in various ways when it is declared. Table H-7 describes various ways a `deque` object can be declared.

TABLE H-7 Various Ways to Declare a `deque` Object

Statement	Description
`deque<elementType> deq;`	Creates an empty `deque` container `deq`. (The default constructor is invoked.)
`deque<elementType> deq(otherDeq);`	Creates the `deque` container `deq` and initializes `it` to the `elements` of `otherDeq`; `deq` and `otherDeq` are of the same type.

```
    copy(intDeq.begin(), intDeq.end(), screen);    //Line 23
    cout << endl;                                  //Line 24

    return 0;
}
```

Sample Run:

```
Line 7: intDeq: 13 75 28 35
Line 12: After adding two more elements, one at the front
         and one at the back, intDeq: 0 13 75 28 35 100
Line 17: After removing the first two elements,
         intDeq: 75 28 35 100
Line 22: After removing the last two elements,
         intDeq: 75 28
```


The statement in Line 1 declares a `deque` container `intDeq` of type `int`, that is, all the elements of `intDeq` are of type `int`. The statement in Line 2 declares `screen` to be an `ostream` iterator initialized to the standard output device. The statements in Lines 3 through 6 use the `push_back` operation to insert four numbers—`13`, `75`, `28`, and `35`—into `intDeq`. The statement in Line 8 outputs the elements of `intDeq`. In the output, see the line marked Line 7, which contains the output of the statements in Lines 7 through 9.

The statement in Line 10 inserts `0` at the beginning of `intDeq`; the statement in Line 11 inserts `100` at the end of `intDeq`. The statement in Line 13 outputs the modified `intDeq`.

The statements in Lines 15 and 16 use the operation `pop_front` to remove the first two elements of `intDeq`, and the statement in Line 18 outputs the modified `intDeq`. The statements in Lines 20 and 21 use the operation `pop_back` to remove the last two elements of `intDeq`, and the statement in Line 23 outputs the modified `intDeq`.

Sequence Container: `list`

This section describes the sequence container `list`. List containers are implemented as doubly linked lists. Thus, every element in a list points to its immediate predecessor and immediate successor (except the first and the last elements). Recall that a linked list is not a random access data structure, such as an array. Therefore, to access, say, the fifth element in a list, we must first traverse the first four elements.

The name of the class containing the definition of the `class list` is `list`. Also, the definition of the `class list`, and the definitions of the functions to implement the various operations on a list, are contained in the header file `list`. Therefore, to use `list` in a program, the program must include the following statement:

```
#include <list>
```

Like other container classes, the `class` `list` also contains several constructors. Thus, a `list` object can be initialized several ways when it is declared. Table H-9 shows various ways to declare and initialize a `list` object.

TABLE H-9 Various Ways to Declare a `list` Object

Statement	Description
`list<elementType> listCont;`	Creates the empty `list` container `listCont`. (The default constructor is involved.)
`list<elementType> listCont(otherList);`	Creates the `list` container `listCont` and initializes it to the elements of `otherList`. `listCont` and `otherList` are of the same type.
`list<elementType> listCont(size);`	Creates the `list` container `listCont` of size `size`. `listCont` is initialized using the default constructor.
`list<elementType> listCont(n, elm);`	Creates the `list` container `listCont` of size `n`. `listCont` is initialized using `n` copies of the element `elm`.
`list<elementType> listCont(beg, end);`	Creates the `list` container `listCont`. `listCont` is initialized to the elements in the range `[beg, end)`, that is, all the elements in the range `beg...end-1`. Both `beg` and `end` are iterators.

Table H-5 described the operations that are common to all containers, and Table H-6 described the operations that are common to all sequence containers. In addition to these common operations, Table H-10 describes operations that are specific to a `list` container. The name of the function implementing the operation is shown in bold. (Suppose that `listCont`, `listCont1`, and `listCont2` are containers of type `list`.)

APPENDIX I
ANSWERS TO ODD-NUMBERED EXERCISES

Chapter 1

1. a. false; b. false; c. true; d. false; e. false; f. false; g. false; h. true; i. true; j. false; k. true; l. false

3. Screen and printer.

5. An operating system monitors the overall activity of the computer and provides services. Some of these services include memory management, input/output activities, and storage management.

7. In machine language, the programs are written using the binary codes, whereas in high-level language, the programs are closer to the natural language. For execution, a high-level language program is translated into machine language, whereas a machine language need not be translated into any other language.

9. Because the computer cannot directly execute instructions written in a high-level language, a compiler is needed to translate a program written in high-level language into machine code.

11. Every computer directly understands its own machine language. Therefore, for the computer to execute a program written in a high-level language, the high-level language program must be translated into the computer's machine language.

13. In linking, an object program is combined with other programs in the library used in the program to create the executable code.

15. To find the weighted average of the four test scores, first you need to know each test score and its weight. Next, you multiply each test score with its weight and then add these numbers to get the average. Therefore:

    1. Get `testScore1, weightTestScore1`
    2. Get `testScore2, weightTestScore2`
    3. Get `testScore3, weightTestScore3`
    4. Get `testScore4, weightTestScore4`
    5.
```
weightedAverage = testScore1 * weightTestScore1 +
                  testScore2 * weightTestScore2 +
                  testScore3 * weightTestScore3 +
                  testScore4 * weightTestScore4;
```

Chapter 2

1. a. false; b. false; c. false; d. true; e. true; f. false; g. true; h. true; i. false; j. true; k. false

3. b, d, e

5. The identifiers `firstName` and `FirstName` are not the same. C++ is case sensitive. The first letter of `firstName` is lowercase `f`, whereas the first character of `FirstName` is uppercase `F`. So these identifiers are different.

7. a. `3`
   b. Not possible. Both of the operands of the operator `%` must be integers. Because the second operand, `w`, is a floating-point value, the expression is invalid.
   c. Not possible. Both of the operands of the operator `%` must be integers. Because the first operand, which is `y + w`, is a floating-point value, the expression is invalid.
   d. `38.5`
   e. `1`
   f. `2`
   g. `2`
   h. `420.0`

9. `7`

11. `a` and `c` are valid.

13. a. `32 * a + b`
    b. `'8'`
    c. `"Julie Nelson"`
    d. `(b * b - 4 * a * c) / (2 * a)`
    e. `(a + b) / c * (e * f) - g * h`
    f. `(-b + (b * b - 4 * a * c)) / (2 * a)`

15.
```
x = 20
y = 15
z = 6
w = 11.5
t = 4.5
```

17. a. `0.50;` b. `24.50;` c. `37.6;` d. `8.3;` e. `10;` f. `38.75`

19. a and c are correct.

21. a.
```
int num1;
int num2;
```
    b.
```
cout << "Enter two numbers separated by spaces." << endl;
```
    c.
```
cin >> num1 >> num2;
```
    d.
```
cout << "num1 = " << num1 << "num2 = " << num2
     << "2 * num1 - num2 = " << 2 * num1 - num2 << endl;
```

```
int main()
{
    string firstName, lastName;
    int num;
    double salary;

    cout << "Enter first name: ";
    cin >> firstName;
    cout << endl;

    cout << "Enter last name: ";
    cin >> lastName;
    cout << endl;

    cout << "Enter a positive integer less than 70: ";
    cin >> num;
    cout << endl;

    salary = num * X;

    cout << "Name: " << firstName << BLANK << lastName << endl;
    cout << "Wages: $" << salary << endl;
    cout << "X = " << X << endl;
    cout << X << " + " << X << " = " << X + Y << endl;

    return 0;
}
```


Chapter 3

1. a. true; b. true; c. false; d. false; e. true; f. true
3. a. `x = 37, y = 86, z = 0.56`
   b. `x = 37, y = 32, z = 86.56`
   c. Input failure: `z = 37.0`, `x = 86`, trying to read the . (period) into `y`.
5. Input failure: Trying to read `A` into `y`, which is an `int` variable. `x = 46`, `y = 18`, and `z = 'A'`. The values of `y` and `z` are unchanged.
7. `iomanip`
9. `getline(cin, name);`
11. a. `name = " Lance Grant", age = 23`
    b. `name = " ", age = 23`
13.
```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    int num1, num2;
    ifstream infile;
    ostream outfile;
```

29. a. both
    b. `do ... while`
    c. `while`
    d. `while`

31. In a pretest loop, the loop condition is evaluated before executing the body of the loop. In a posttest loop, the loop condition is evaluated after executing the body of the loop. A posttest loop executes at least once, whereas a pretest loop may not execute at all.

33.
```
int num;
do
{
    cout << "Enter a number less than 20 or greater than 75: ";
    cin >> num;
}
while (20 <= num && num <= 75);
```

35.
```
int i = 0, value = 0;
do
{
    if (i % 2 == 0 && i <= 10)
        value = value + i; i;
    else if (i % 2 == 0 && i > 10)
        value = value + i;
    else
        value = value - i;
    i = i + 1;
}
while (i <= 20);

cout << "value = " << value << endl;
```

```
The Output is: Value = 200
```

37.
```
cin >> number;
while (number != -1)
{
    total = total + number;
    cin >> number;
}
cout << endl;
cout << total << endl;
```

39. a.
```
number = 1;
while (number <= 10)
{
    cout << setw(3) << number;
    number++;
}
```

9. `10, 12, 18, 21, 25, 28, 30, 71, 32, 58, 15`
11. Bubble sort: 49,995,000; selection sort: 49,995,000; insertion sort: 25,007,499
13. 26
15. To use a `vector` object in a program, the program must include the header file `vector`.
17. `1 3 5 7 9`
19. a.
```
vector<int> secretList;
```
b.
```
secretList.push_back(56);
secretList.push_back(28);
secretList.push_back(32);
secretList.push_back(96);
secretList.push_back(75);
```
c.
```
for (unsigned int i = 0; i < secretList.size(); i++)
        cout << secretList[i] << " ";

cout << endl;
```
21. a.
```
cout << myList.front() << " " << myList.back() << endl;
```
b.
```
length = myList.size();
```
c.
```
for (int i = 0; i < myList.size(); i++)
        cout << myList[i] << " ";

cout << endl;
```


Chapter 11

1. a. false; b. false; c. true; d. true; e. true; f. true; g. false
3.
```
checkingAccount newAcct;

newAcct.name = "Jason Miller";
newAcct.accountNum = 17328910;
newAcct.balance = 24476.38;
newAcct.interestRate = 0.025;
```
5.
```
movieType newRelease;

newRelease.name = "Summer Vacation";
newRelease.director = "Tom Blair";
newRelease.producer =  "Rajiv Merchant";
newRelease.yearReleased = 2005;
newRelease.copiesInStock = 34;
```

f.
```
xClass::xClass()
{
    u = 0;
    w = 0;
}
```

g. `x.print();`

h. `xClass t(20, 35.0);`

5. a.
```
int testClass::sum()
{
    return x + y;
}

void testClass::print() const
{
    cout << "x = " << x << ", y = " << y << endl;
}

testClass::testClass()
{
    x = 0;
    y = 0;
}

testClass::testClass(int a, int b)
{
    x = a;
    y = b;
}
```

b. One possible solution. (We assume that the name of the header file containing the definition of the `class` `testClass` is `Exercise5Ch12.h`.)

```
#include <iostream>
#include "Exercise5Ch12.h"

int main()
{
    testClass one;
    testClass two(4, 5);

    one.print();
    two.print();

    return 0;
}
```

7. a. `personType student("Buddy", "Arora");`

b. `student.print();`

c. `student.setName("Susan", "Gilbert");`

13. The members `setX`, `print`, `y`, and `setY` are `protected` members in `class third`. The `private` member `x` of `class first` is hidden in `class third`, and it can be accessed in `class` third only through the `protected` and `public` members of `class first`.

15. Because the `memberAccessSpecifier` is not specified, it is a private inheritance. Therefore, all of the members of the `class first` become `private` members in `class fifth`.

17. a.
```
void two::setData(int a, int b, int c)
{
    one::setData(a, b);
    z = c;
}
```
b.
```
void two::print() const
{
     one::print();
     cout << z << endl;
}
```

19.
```
In base: x = 7
In derived: x = 3, y = 8; x + y = 11
****7
####11
```


Chapter 14

1. a. false; b. false; c. false; d. true; e. true; f. true; g. false; h. false

3. The operator `*` is used to declare a pointer variable and to access the memory space to which a pointer variable points.

5.
```
98 98
98 98
```

7. b and c

9.
```
78 78
```

11.
```
27 35
73 27
36 36
```

13.
```
4 4 5 7 10 14 19 25 32 40
```

15. The operator `delete` deallocates the memory space to which a pointer points.

17. a.
```
num = new int[10];
```
b.
```
for (int j = 0; j < 10; j++)
   cin >> num[j];
```
c.
```
delete [] num;
```

19. In a shallow copy of data, two or more pointers point to the same memory space. In a deep copy of data, each pointer has its own copy of the data.

23.
```
template <class Type>
void reverseStack(stackType<Type> &s)
{
    linkedQueueType<Type> q;
    Type elem;

    while (!s.isEmptyStack())
    {
        elem = s.top();
        s.pop();
        q.addQueue(elem);
    }

    while (!q.isEmptyQueue())
    {
        elem = q.front();
        q.deleteQueue();
        s.push(elem);
    }
}
```

25.
```
template <class Type>
int queueType<Type>::queueCount()
{
    return count;
}
```


27. Answer to this question is available at the Web site accompanying this book.

29. Answer to this question is available at the Web site accompanying this book.

S

W