Programming in C++

Online module to accompany *Invitation to Computer Science, 5th* Edition, ISBN-10: 0324788592; ISBN-13: 9780324788594 (Course Technology, a part of Cengage Learning, 2010).

1. Introduction to C++
- **1.1** A Simple C++ Program
- **1.2** Creating and Running a C++ Program

2. Virtual Data Storage

3. Statement Types
- **3.1** Input/Output Statements
- **3.2** The Assignment Statement
- **3.3** Control Statements

4. Another Example

5. Managing Complexity
- **5.1** Divide and Conquer
- **5.2** Using Functions
- **5.3** Writing Functions

6. Object-Oriented Programming
- **6.1** What Is It?
- **6.2** C++ and OOP
- **6.3** One More Example
- **6.4** What Have We Gained?

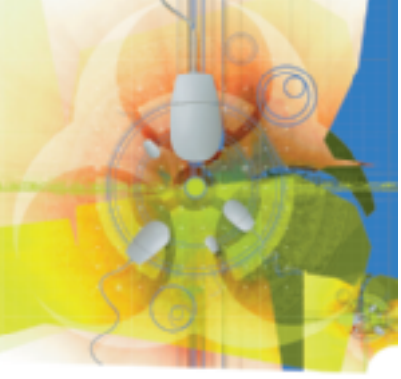
7. Graphical Programming
- **7.1** Graphics Primitives
- **7.2** An Example of Graphics Programming

8. Conclusion

EXERCISES

ANSWERS TO PRACTICE PROBLEMS

©2010 Course Technology, a part of Cengage Learning.



1 Introduction to C++

Hundreds of high-level programming languages have been developed; a fraction of these have become viable, commercially successful languages. There are a half-dozen or so languages that can illustrate some of the concepts of a high-level programming language, but this module uses C++ for this purpose. The popular C++ language was developed in the early 1980s by Bjarne Stroustrup at AT&T Labs and was commercially released by AT&T in 1985.

Our intent here is not to make you an expert programmer—any more than our purpose in Chapter 4 was to make you an expert circuit designer. Indeed, there is much about the language that we will not even discuss. You will, however, get a sense of what programming in a high-level language is like, and perhaps you will see why some people think it is one of the most fascinating of human endeavors.

1.1 *A Simple C++ Program*

Figure 1 shows a simple but complete C++ program. Even if you know nothing about the C++ language, it is not hard to get the general drift of what the program is doing.

Someone running this program (the "user") could have the following dialogue with the program, where boldface indicates what the user types:

```
Enter your speed in mph: 58
Enter your distance in miles: 657.5
At 58 mph, it will take
11.3362 hours to travel 657.5 miles.
```

The general form of a typical C++ program is shown in Figure 2. To compare our simple example program with this form, we have reproduced the example program in Figure 3 with a number in front of each line. The numbers are there for reference purposes only; they are *not* part of the program.

Lines 1–3 in the program of Figure 3 are C++ **comments**. Anything appearing on a line after the double slash symbol (//) is ignored by the compiler, just as anything following the double dash (--) is treated as a comment in the assembly language programs of Chapter 6. Although the computer ignores comments, they are important to include in a program because they give information to the human readers of the code. Every high-level language has some facility for including comments, because understanding code that someone else

©2010 Course Technology, a part of Cengage Learning.

FIGURE 1

A Simple C++ Program

```
//Computes and outputs travel time
//for a given speed and distance
//Written by J. Q. Programmer, 6/15/10

#include <iostream>
using namespace std;

void main()
{
     int speed;       //rate of travel
     double distance; //miles to travel
     double time;     //time needed for this travel

     cout << "Enter your speed in mph: ";
     cin >> speed;
     cout << "Enter your distance in miles: ";
     cin >> distance;

     time = distance/speed;

     cout << "At " << speed << " mph, "
          << "it will take " << endl;
     cout << time << " hours to travel "
          << distance << " miles." << endl;
}
```

has written (or understanding your own code after a period of time has passed) is very difficult without the notes and explanations that comments provide. Comments are one way to *document* a computer program to make it more understandable. The comments in the program of Figure 3 describe what the program does plus tell who wrote the program and when. These three comment lines together make up the program's **prologue comment** (the introductory comment that comes first). According to the general form of Figure 2, the prologue comment is optional, but providing it is always a good idea. It's almost like the headline in a newspaper, giving the big picture up front.

Blank lines in C++ programs are ignored and are used, like comments, to make the program more readable by human beings. In our example program, we've used blank lines (lines 4, 7, 13, 18, 20) to separate sections of the program, visually indicating groups of statements that are related.

FIGURE 2

The Overall Form of a Typical C++ Program

```
prologue comment     [optional]
include directives   [optional]
using directive      [optional]
functions            [optional]
main function
{
      declarations   [optional]
      main function body
}
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 3

The Program of Figure 1 (line numbers added for reference)

```
1. //Computes and outputs travel time
2. //for a given speed and distance
3. //Written by J. Q. Programmer, 6/15/10
4.
5. #include <iostream>
6. using namespace std;
7.
8. void main()
9. {
10.   int speed;       //rate of travel
11.   double distance; //miles to travel
12.   double time;     //time needed for this travel
13.
14.   cout << "Enter your speed in mph: ";
15.   cin >> speed;
16.   cout << "Enter your distance in miles: ";
17.   cin >> distance;
18.
19.   time = distance/speed;
20.
21.   cout << "At " << speed << " mph, "
22.        << "it will take " << endl;
23.   cout << time << " hours to travel "
24.        << distance << " miles." << endl;
25. }
```

Line 5 is an **include directive** to the compiler that refers to the *iostream* library. The eventual effect is that the linker includes object code from this library. The core C++ language does not provide a way to get data into a program or for a program to display results. The *iostream* library contains code for these purposes. Line 6 is a **using directive** that tells the compiler to look in the *std* namespace for the definition of any names not specifically defined within the program. In this program, *cin* and *cout* get their meaning (which is that input will come from the keyboard and output will go to the screen) from the *std* namespace. In addition to *iostream*, C++ has many other code libraries, such as mathematical and graphics libraries, and therefore many other include directives are possible. Include directives are also optional, but it would be a trivial program indeed that did not need input data or produce output results, so virtually every C++ program has at least the include directive and using directive shown in our example.

Our sample program has no functions other than the main function (note that such functions are optional). The purpose of additional functions is to do some calculation or perform some subtask for the main function, much as the Find Largest algorithm from Chapter 2 is used by the Selection Sort algorithm of Chapter 3.

Line 8 signals the beginning of the main function. The curly braces at lines 9 and 25 enclose the main **function body**, which is the heart of the sample program. Lines 10–12 are declarations that name and describe the items of data that are used within the main function. Descriptive names—*speed*, *distance*, and *time*—are used for these quantities to help document

©2010 Course Technology, a part of Cengage Learning.

A Remarkable History

Dr. Bjarne Stroustrup is an AT&T Fellow at AT&T Labs Research (formerly known as Bell Labs). He has been honored by AT&T for "fundamental contributions to the development of computer languages and object-oriented programming, culminating in the C++ programming language." In addition to his work in programming languages, his research interests include distributed systems, operating systems, and simulation. Dr. Stroustrup's accomplishment as the designer and original implementor of a new programming language is remarkable, but it is only one of many remarkable accomplishments achieved at AT&T Labs Research. Through its long history of technological research and innovation, AT&T Labs Research has become something of a national treasure.

The original Bell Labs was created as part of AT&T in 1925. That same year, Bell Labs was awarded its first patent, for a "clamping and supporting device." The transistor (see Chapter 4) was invented in 1947 by three Bell Lab scientists, John Bardeen, Walter Brattain, and William Shockley, who later shared the Nobel Prize for this work. In total, six Nobel prizes have been awarded to 11 researchers for work done while they were at Bell Labs.

In March 2003, AT&T Labs Research was awarded its 30,000th patent, this one for "mechanisms for guaranteeing Quality of Service in Internet Protocol (IP) networks, which should help make packet-based networks as reliable as today's telephone networks." To appreciate fully the magnitude of 30,000 patents, we should note that this averages to more than one patent per day, 365 days per year, over a period of 78 years! Scientists and engineers at AT&T Labs Research, working in the areas of IP network management and optical technology to automatic speech recognition and text-to-speech research, continue a remarkable pace of innovation and technological advances, averaging almost two patents per working day.

their purpose in the program, and comments provide further clarification. Line 10 describes an integer quantity (type "int") called *speed*. Lines 11 and 12 declare *distance* and *time* as real number quantities (type "double"). A real number quantity is one containing a decimal point, such as 28.3, 102.0, or −17.5. Declarations are also optional in the sense that if a program does not use any data, no declarations are needed, but again, it would be unusual to find such a trivial program.

Messages to the user begin with *cout*; the *cin* statements get the values the user entered for speed and distance and store them in *speed* and *distance*, respectively. Line 19 computes the time required to travel this distance at this speed. Finally, lines 21–24 print the output to the user's screen. The values of *speed*, *time*, and *distance* are inserted in appropriate places among the strings of text shown in double quotes.

You may have noticed that most of the statements in this program end with a semicolon. A semicolon must appear at the end of every executable C++ instruction, which means everywhere except at the end of a comment, an include directive, or the beginning of a function, such as

```
void main ()
```

The semicolon requirement is a bit of a pain in the neck, but the C++ compiler generates one or more error messages if you omit the semicolon, so after the first few hundred times this happens, you tend to remember to put it in.

C++, along with every other programming language, has very specific rules of **syntax**—the correct form for each component of the language. Having

©2010 Course Technology, a part of Cengage Learning.

a semicolon at the end of every executable statement is a C++ syntax rule. Any violation of the syntax rules generates an error message from the compiler, because the compiler does not recognize or know how to translate the offending code. In the case of a missing semicolon, the compiler cannot tell where the instruction ends. The syntax rules for a programming language are often defined by a formal grammar, much as correct English is defined by rules of grammar.

C++ is a **free-format language**, which means that it does not matter where things are placed on a line. For example, we could have written

```
          time =
distance       /
                 speed;
```

although this is clearly harder to read. The free-format characteristic explains why a semicolon is needed to mark the end of an instruction, which might be spread over several lines.

1.2 *Creating and Running a C++ Program*

Creating and running a C++ program is basically a three-step process. The first step is to type the program into a text editor. When you are finished, you save the file, giving it a name with the extension .cpp. So the file for Figure 1 could be named

TravelPlanner.cpp

As the second step, the program must be compiled using a C++ compiler for your computer, and the resulting object code linked with any C++ library object code. In our example, the program in the file *TravelPlanner.cpp* would be compiled, resulting in a file called

TravelPlanner.exe

The third step loads and executes the program file, in this case *TravelPlanner.exe*. Depending on your system, you may have to type operating system commands for the last two steps.

Another approach is to do all of your work in an **Integrated Development Environment**, or **IDE**. The IDE lets the programmer perform a number of tasks within the shell of a single application program, rather than having to use a separate program for each task. A modern programming IDE provides a text editor, a file manager, a compiler, a linker and loader, and tools for debugging, all within this one piece of software. The IDE usually has a GUI (graphical user interface) with menu choices for the different tasks. This can significantly speed up program development.

This C++ exercise is just a beginning. In the rest of this module, we'll examine the features of the language that will enable you to write your own C++ programs to carry out more sophisticated tasks.

©2010 Course Technology, a part of Cengage Learning.

C++ COMPILERS

There are many C++ compilers available. The C++ examples in this module (with the exception of the graphics programs in Section 7) were written and executed in Microsoft Visual C++ 2008, part of Microsoft Visual Studio 2008. This is an IDE (with a GUI interface) that supports many programming languages. Visual C++ 2008 Express Edition is a lightweight version that is freely downloadable from Microsoft at

www.microsoft.com/express/product/default.aspx

Visual C++ 2008 Express Edition runs on Windows XP or Windows Vista operating systems. Its use requires the Microsoft .NET framework. If this is not already on your Windows system, you will be alerted at installation, and you can go to

www.microsoft.com/net/Download.aspx

to download it.

You can also download the free open-source C++ command-line compiler (g++) that is part of the GNU Compiler Collection from

http://gcc.gnu.org

There are versions that run on Linux and Mac OS X systems as well as Windows systems.

A small C++ compiler is included with the laboratory software for the *Invitation to Computer Science, 5th* Edition text. If you use this compiler, the

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

code must be replaced by

```
#include<iostream.h>
```

The graphics library used in Section 7 of this module is also part of the *Invitation* C++ compiler.

2 Virtual Data Storage

One of the improvements we seek in a high-level language is freedom from having to manage data movement within memory. Assembly language does not require us to give the actual memory address of the storage location to be used for each item, as in machine language. However, we still have to move values, one by one, back and forth between memory and the arithmetic logic unit (ALU) as simple modifications are made, such as setting the value of A to the sum of the values of B and C. We want the computer to let us use data values by name in any appropriate computation without thinking about where they are stored or what is currently in some register in the ALU. In fact, we do not even want to know that there *is* such a thing as an ALU, where data are moved to be operated on; instead, we want the virtual machine to manage the details when we request that a computation be performed. A high-level language allows this, and it also allows the names for data items to be more meaningful than in assembly language.

©2010 Course Technology, a part of Cengage Learning.

Names in a programming language are called **identifiers**. Each language has its own specific rules for what a legal identifier can look like. In C++ an identifier can be any combination of letters, digits, and the underscore symbol (_), as long as it does not begin with a digit. However, identifiers beginning with underscore characters should be avoided; they are generally used for special purposes. An additional restriction is that an identifier cannot be one of the few **keywords**, such as "void," "int," "double," and so forth, that have a special meaning in C++ and that you would not be likely to use anyway. The three integers *B, C,* and *A* in our assembly language program can therefore have more descriptive names, such as *subTotal*, *tax*, and *finalTotal*. The use of descriptive identifiers is one of the greatest aids to human understanding of a program. Identifiers can be almost arbitrarily long, so be sure to use a meaningful identifier such as *finalTotal* instead of something like *A*; the improved readability is well worth the extra typing time. C++ is a **case-sensitive** language, which means that uppercase letters are distinguished from lowercase letters. Thus, *FinalTotal*, *Finaltotal*, and *finalTotal* are three different identifiers.

CAPITALIZATION OF IDENTIFIERS

There are two standard capitalization patterns for identifiers, particularly "multiple word" identifiers:

camel case: First word begins with a lowercase letter, additional words begin with uppercase letters (*finalTotal*)

Pascal case: All words begin with an uppercase letter (*FinalTotal*)

The code in this module uses the following convention for creating identifiers (examples included):

Simple variables – camel case: *speed*, *time*, *finalTotal*

Named constants – all uppercase: *PI, FREEZING_POINT*

Function names – camel case: *myFunction*, *getInput*

Class names – Pascal case: *MyClass*

Object names – camel case: *myObject*

The underscore character is not used except for named constants. Occasionally, however, we'll use single capital letters for identifiers in quick code fragments.

Data that a program uses can come in two varieties. Some quantities are fixed throughout the duration of the program, and their values are known ahead of time. These quantities are called **constants**. An example of a constant is the integer value 2. Another is an approximation to π, say 3.1416. The integer 2 is a constant that we don't have to name by an identifier, nor do we have to build the value 2 in memory manually by the equivalent of a .DATA pseudo-op. We can just use the symbol "2" in any program statement. When "2" is first encountered in a program statement, the binary representation of the integer 2 is automatically generated and stored in a memory location. Likewise, we can use "3.1416" for the real number value 3.1416, but if we are

©2010 Course Technology, a part of Cengage Learning.

really using this number as an approximation to π, it is more informative to use the identifier *PI*.

Some quantities used in a program have values that change as the program executes, or values that are not known ahead of time but must be obtained from the computer user (or from a data file previously prepared by the user) as the program runs. These quantities are called **variables**. For example, in a program doing computations with circles (where we might use the constant *PI*), we might need to obtain from the user or a data file the radius of the circle. This variable can be given the identifier *radius*.

Identifiers for variables serve the same purpose in program statements as pronouns do in ordinary English statements. The English statement "He will be home today" has specific meaning only when we plug in the value for which "He" stands. Similarly, a program statement such as

```
time = distance/speed;
```

becomes an actual computation only when numeric values have been stored in the memory locations referenced by the *distance* and *speed* identifiers.

We know that all data are represented internally in binary form. In Chapter 4 we noted that any one sequence of binary digits can be interpreted as a whole number, a negative number, a real number (one containing a decimal point, such as −17.5 or 28.342), or as a letter of the alphabet. C++ requires the following information about each variable in the program:

- What identifier we want to use for it (its name)
- What **data type** it represents (e.g., an integer or a letter of the alphabet)

The data type determines how many bytes will be needed to store the variable—that is, how many memory cells are to be considered as one **memory location** referenced by one identifier—and also how the string of bits in that memory location is to be interpreted. C++ provides several "primitive" data types that represent a single unit of information, as shown in Figure 4.

The way to give the necessary information within a C++ program is to declare each variable. A **variable declaration** consists of a data type followed by a list of one or more identifiers of that type. Our sample program used three declaration statements:

```
int speed;            //rate of travel
double distance;      //miles to travel
double time;          //time needed for this travel
```

but these could have been combined into two:

```
int speed;                //rate of travel
double distance, time;    //miles to travel and time
                          //needed for this travel
```

Where do the variable declarations go? Although the only requirement is that a variable must be declared before it can be used, all variable declarations are usually collected together at the top of the main function, as in our sample program. This gives the reader of the code quick information about the data that the program will be using.

©2010 Course Technology, a part of Cengage Learning.

FIGURE 4

Some of the C++ Primitive Data Types

int	an integer quantity
double	a real number
char	a character (a single keyboard character, such as 'a')

What about the constant *PI*? We want to assign the fixed value 3.1416 to the *PI* identifier. Constant declarations are just like variable declarations, with the addition of the keyword **const** and the assignment of the fixed value to the constant identifier.

```
const double PI = 3.1416;
```

Many programmers use all uppercase letters to denote constant identifiers, but the compiler identifies a constant quantity only by the presence of **const** in the declaration. Once a quantity has been declared as a constant, any attempt later in the program to change its value generates an error message from the compiler.

In addition to variables of a primitive data type that hold only one unit of information, it is possible to declare a whole collection of related variables at one time. This allows storage to be set aside as needed to contain each of the values in this collection. For example, suppose we want to record the number of hits on a Web site for each month of the year. The value for each month is a single integer. We want a collection of 12 such integers, ordered in a particular way. An **array** groups together a collection of memory locations, all storing data of the same type. The following statement declares an array:

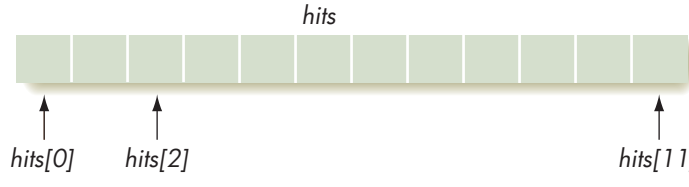
```
int hits[12];
```

The 12 indicates that there are to be 12 memory locations set aside, each to hold a variable of type *int*. The collection as a whole is referred to as *hits*, and the 12 individual array elements are numbered from *hits[0]* to *hits[11]*. (Notice that a C++ array counts from 0 up to 11, instead of from 1 up to 12.) Thus, we use *hits[0]* to refer to the first entry in *hits*, which represents the number of visits to the Web site during the first month of the year, January. Next, *hits[2]* refers to the number of visits during March, and *hits[11]* to the number of visits during December. In this way we use one declaration to set up 12 separate (but related) *int* storage locations. Figure 5 illustrates this array.

Here is an example of the power of a high-level language. In assembly language we can name only individual memory locations—that is, individual items of data—but in C++ we can also assign a name to an entire collection of related data items. An array thus enables us to talk about an entire table of values, or the individual elements making up that table. If we are writing C++

FIGURE 5

A 12-Element Array hits



©2010 Course Technology, a part of Cengage Learning.

programs to implement the data cleanup algorithms of Chapter 3, we can use an array of integers to store the 10 data items.

PRACTICE PROBLEMS

1. Which of the following are legitimate C++ identifiers?

   ```
   martinBradley    C3P_OH    Amy3    3Right    const
   ```

2. Write a declaration statement for a C++ program that uses one integer quantity called *number*.
3. Write a C++ statement that declares a type *double* constant called *TAX_RATE* that has the value 5.5.
4. Using the *hits* array of Figure 5, how do you reference the number of hits on the Web page for August?

3 Statement Types

Now that we can reserve memory for data items by simply naming what we want to store and describing its data type, we will examine additional kinds of programming instructions (statements) that C++ provides. These statements enable us to manipulate the data items and do something useful with them. The instructions in C++, or indeed in any high-level language, are designed as components for algorithmic problem solving, rather than as one-to-one translations of the underlying machine language instruction set of the computer. Thus they allow the programmer to work at a higher level of abstraction. In this section we examine three types of high-level programming language statements. They are consistent with the pseudocode operations we described in Chapter 2 (see Figure 2.9).

Input/output statements make up one type of statement. An **input statement** collects a value from the user for a variable within the program. In our TravelPlanner program, we need input statements to get the specific values of the speed and distance that are to be used in the computation. An **output statement** writes a message or the value of a program variable to the user's screen. Once the TravelPlanner program computes the time required to travel the given distance at the given speed, the output statement displays that value on the screen, along with other information about what that value means.

Another type of statement is the **assignment statement**, which assigns a value to a program variable. This is similar to what an input statement does, except that the value is not collected directly from the user, but is computed by the program. In pseudocode we called this a "computation operation."

Control statements, the third type of statement, affect the order in which instructions are executed. A program executes one instruction or program statement at a time. Without directions to the contrary, instructions are executed sequentially, from first to last in the program. (In Chapter 2 we called this a straight-line algorithm.) Imagine beside each program statement

©2010 Course Technology, a part of Cengage Learning.

a light bulb that lights up while that statement is being executed; you would see a ripple of lights from the top to the bottom of the program. Sometimes, however, we want to interrupt this sequential progression and jump around in the program (which is accomplished by the instructions JUMP, JUMPGT, and so on, in assembly language). The progression of lights, which may no longer be sequential, illustrates the **flow of control** in the program—that is, the path through the program that is traced by following the currently executing statement. Control statements direct this flow of control.

3.1 *Input/Output Statements*

Remember that the job of an input statement is to collect from the user specific values for variables in the program. In pseudocode, to get the value for *speed* in the TravelPlanner program, we would say something like

Get value for *speed*

C++ can do this task using an input statement of the form

```
cin >> speed;
```

Because all variables must be declared before they can be used, the declaration statement that says *speed* is to be a variable (of data type *int*) precedes this input statement. If the user enters a decimal number as the input value for *speed*, it will be truncated, and the digits behind the decimal point will be lost; thus, if the user enters 48.7, the value stored in *speed* will be the integer value 48.

Let's say that we have written the entire TravelPlanner program and it is now executing. When the preceding input statement is encountered, the program stops and waits for the user to enter a value for *speed* (by typing it at the keyboard, followed by pressing the ENTER key). For example, the user could type

```
58 <ENTER>
```

By this action, the user contributes a value to the **input stream**, the sequence of values entered at the keyboard. The input stream is named *cin* (pronounced "see-in"). The arrows (>>) in the input statement above stand for the **extraction operator** that removes (extracts) the next value from the input stream and stores it in the memory location referenced by the identifier *speed*. The code for the extraction operator and the definition of the *cin* stream are supplied by the *iostream* library and namespace *std*; that's why any C++ program that requires an input statement needs the directives

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

After the value of distance has been input using the statement

```
cin >> distance;
```


©2010 Course Technology, a part of Cengage Learning.

the value of the time can be computed and stored in the memory location referenced by *time*. A pseudocode operation for producing output would be something like

Print the value of *time*

Output in C++ is handled as the opposite of input. A value stored in memory—in this case the value of the variable *time*—is copied and inserted into the **output stream** by the **insertion operator <<**. The output stream that goes to the screen is called *cout* (pronounced "see-out"). The appropriate statement is

```
cout << time;
```

The code for the insertion operator and the definition of the *cout* stream are again supplied by the *iostream* library and namespace *std*.

It is easy to confuse the direction of the arrows for input and output. The extraction operator extracts a value from the input stream and puts it *into the variable to which it points*:

```
cin >> speed;
```

The insertion operator takes a value from a variable and inserts it *into the output stream to which it points*:

```
cout << time;
```

Depending on the size of the value, C++ may write out real number values in either **fixed-point format** or **scientific notation**. A sample value in fixed-point format is

```
11.3362
```

whereas in scientific notation (also called **floating-point format**), it is

```
1.13362e+001
```

which means 1.13362×10^1. (The "e" means "times 10 to the power of . . .".) It may be convenient to specify one output format or the other, rather than leaving this up to the system to decide. To force all subsequent output into fixed-point notation, we put the following somewhat mysterious formatting statement in the program:

```
cout.setf(ios::fixed);
```

To force all subsequent output into scientific notation, we use the statement

```
cout.setf(ios::scientific);
```

It is also possible to control the number of places behind the decimal point that are displayed in the output. Inserting the statement

```
cout.precision(2);
```


©2010 Course Technology, a part of Cengage Learning.

before the output statement results in a fixed-point output of

```
11.34
```

The corresponding result for scientific notation is

```
1.13e+001
```

Each value is rounded to two digits behind the decimal point (picking up the 2 from the cout.precision statement), although the fixed-point value shows a total of four significant digits, and the scientific notation format shows only three. The ability to specify the number of decimal digits in fixed-point output is particularly handy for dealing with dollar-and-cent values, where we always expect to see two digits behind the decimal point.

The programmer can also specify the total number of columns to be taken up by the next output value. Inserting into the output stream the "set width" expression

```
setw(n)
```

where *n* has some integer value, allots *n* columns for the next value that is output, including the decimal point. If *n* is too small, the entire value is written out anyway, overriding the width specification. If *n* is too big, the value is right-justified within the allotted space. The statement

```
cout << setw(8) << time;
```

requests eight columns for the value of *time*. Using *setw* helps to align columns of values but is generally less important when writing out single values. Unlike the fixed-point or floating-point format, which only needs to be set once, the *setw* expression must be used each time a value is to be written out. In addition, *setw* is available from a different set of library files, so another include statement is required in order to use it, namely

```
#include <iomanip>
```

If the user suddenly sees the number 11.34 on the screen, he or she may have no idea what it represents. Some additional text is needed to describe this value. Textual information can be inserted into the output stream by placing it within quotation marks. Text within quotation marks ("") is called a **literal string** and is printed out exactly as is. In the TravelPlanner program, we used the output statements

```
cout  << "At " << speed << " mph, "
      << "it will take " << endl;
cout  << time << " hours to travel "
      << distance << " miles." << endl;
```

There are two C++ output instructions (note the two terminating semicolons) that happen to take up four lines. They contribute five literal strings and the values of three variables to the output stream, each requiring an insertion

©2010 Course Technology, a part of Cengage Learning.

operator. Assuming that we are using fixed-point format with precision set to 2, the output is

```
At 58 mph, it will take
11.34 hours to travel 657.5 miles.
```

Note that in the program instruction we put spaces at the beginning and end of most of the literal strings, within the quotation marks so that they are part of the text. Without these spaces, the output would be

```
At58mph,it will take
11.34hours to travel657.5miles.
```

Output formatting largely determines how attractive and easy to read the output is. We might want to design the output to look like

```
At 58 mph, it will take
  11.34 hours
to travel 657.5 miles.
```

We can accomplish this with the five statements:

```
cout << "At " << speed << " mph, it will take " << endl;
cout << endl;
cout << setw(10) << time << " hours" << endl;
cout << endl;
cout << "to travel " << distance << " miles."
```

Each statement produces one line of output because *endl* (an abbreviation for End Line) sends the cursor to the next line on the screen. The result is that the next value in the output stream begins on a new line. Using *endl* is another way to format output. The second and fourth output statements contain neither a literal string nor an identifier; their effect is to write a blank line. The *setw* expression in the third output statement positions the numerical value of *time* right-justified within 10 columns, which produces the indenting effect.

Let's back up a bit and note that we also need to print some text information before the input statement, to alert the user that the program expects some input. A statement such as

```
cout << "Enter your speed in mph: ";
```

acts as a user **prompt**. Without a prompt, the user may be unaware that the program is waiting for some input; instead, it may simply seem to the user that the program is "hung up."

Assembling all of these bits and pieces, we can see that

```
cout << "Enter your speed in mph: ";
cin >> speed;
cout << "Enter your distance in miles: ";
cin >> distance;
```


©2010 Course Technology, a part of Cengage Learning.

is a series of prompt, input, prompt, input statements to get the data, and then

```
cout  << "At " << speed << " mph, "
      << "it will take " << endl;
cout  << time << " hours to travel "
      << distance << " miles." << endl;
```

writes out the computed value of the *time* along with the associated input values in an informative message. In the middle, we need a program statement to compute the value of *time*. We can do this with a single assignment statement; the assignment statement is explained in the next section.

PRACTICE PROBLEMS

1. Write two statements that prompt the user to enter an integer value and store that value in a (previously declared) variable called *quantity*.
2. A program has computed a value of 37 for the variable *height*. Write an output statement that prints this variable using six columns, and with successive output appearing on the next line.
3. What appears on the screen after execution of the following statement?

   ```
   cout << "This is" << "goodbye" << endl;
   ```

3.2 *The Assignment Statement*

As we said earlier, an assignment statement assigns a value to a program variable. This is accomplished by evaluating some expression and then writing the resulting value in the memory location referenced by the program variable. The general pseudocode operation

Set the value of "variable" to "arithmetic expression"

has as its C++ equivalent

variable = expression;

The expression on the right is evaluated, and the result is then written into the memory location named on the left. For example, suppose that *A*, *B*, and *C* have all been declared as integer variables in some program. The assignment statements

```
B = 2;
C = 5;
```


©2010 Course Technology, a part of Cengage Learning.

result in *B* taking on the value 2 and *C* taking on the value 5. After execution of

```
A = B + C;
```

A has the value that is the sum of the current values of *B* and *C*. Assignment is a destructive operation, so whatever *A*'s previous value was, it is gone. Note that this one assignment statement says to add the values of *B* and *C* and assign the result to *A*. This one high-level language statement is equivalent to three assembly language statements needed to do this same task (LOAD B, ADD C, STORE A). A high-level language program thus packs more power per line than an assembly language program. To state it another way, whereas a single assembly language instruction is equivalent to a single machine language instruction, a single C++ instruction is usually equivalent to many assembly language instructions or machine language instructions, and it allows us to think at a higher level of problem solving.

In the assignment statement, the expression on the right is evaluated first. Only then is the value of the variable on the left changed. This means that an assignment statement like

```
A = A + 1;
```

makes sense. If *A* has the value 7 before this statement is executed, then the expression evaluates to

```
7 + 1, or 8
```

and 8 then becomes the new value of *A*. (Here it becomes obvious that the assignment instruction symbol = is not the same as the mathematical equals sign =, because $A = A + 1$ does not make sense mathematically.)

All four basic arithmetic operations can be done in C++, where they are denoted by

```
+ Addition
- Subtraction
* Multiplication
/ Division
```

For the most part, this is standard mathematical notation, rather than the somewhat verbose assembly language op code mnemonics such as SUBTRACT. The reason a special symbol is used for multiplication is that $\times$ would be confused with *x*, an identifier, · (a multiplication dot) doesn't appear on the keyboard, and juxtaposition—writing *AB* for *A*B*—would look like a single identifier named *AB*.

We do have to pay some attention to data types. In particular, division has one peculiarity. If at least one of the two values being divided is a real number, then division behaves as we expect. Thus,

```
7.0/2  7/2.0  7.0/2.0
```

all result in the value 3.5. However, if the two values being divided are both integers, the result is an integer value; if the division doesn't "come out

©2010 Course Technology, a part of Cengage Learning.

even," the integer value is obtained by truncating the answer to an integer quotient. Thus,

```
7/2
```

results in the value 3. Think of grade-school long division of integers:

$$\begin{array}{r} 3 \\ 2\overline{)7} \\ \underline{6} \\ 1 \end{array}$$

Here the quotient is 3 and the remainder is 1. C++ also provides an operation, with the symbol %, to obtain the integer remainder. Using this operation,

```
7 % 2
```

results in the value 1. If the values are stored in type *int* variables, the same thing happens. For example,

```
int numerator;
int denominator;
numerator = 7;
denominator = 2;
cout << "The result of " << numerator << "/"
     << denominator << " is "
     << numerator/denominator << endl;
```

produces the output

```
The result of 7/2 is 3
```

As soon as an arithmetic operation involves one or more real (decimal) numbers, any integers are converted to their real number equivalent, and the calculations are done with real numbers.

Data types also play a role in assignment statements. Suppose the expression in an assignment statement evaluates to a real number and is then assigned to an identifier that has been declared as an integer. The real number is truncated, and the digits behind the decimal point are lost. We mentioned that this same problem occurs if you input a decimal value for an integer variable. Unlike the input situation, the C++ compiler can see what you are doing with the assignment statement and will usually give you a warning that says something about "possible loss of data." But assigning an integer value to a type *double* identifier merely changes the integer to its real number equivalent. C++ does this **type casting** (changing of data type) automatically. This type cast would merely change the integer 3, for example, to its real number equivalent 3.0.

This explains why we declared *distance* to be type *double* in the TravelPlanner program. The user can enter an integer value for distance, and C++ will type cast it to a real number. But if we had declared both *speed* and *distance* to be integers, then the division to compute *time* would only produce integer answers.

©2010 Course Technology, a part of Cengage Learning.

You should assign only an expression that has a character value to a variable that has been declared to be type *char*. Suppose that *letter* is a variable of type *char*. Then

```
letter = 'm';
```

is a legitimate assignment statement, giving *letter* the value of the character 'm'. Note that single quotation marks are used here, as opposed to the double quotation marks that enclose a literal string. The assignment

```
letter = '4';
```

is also acceptable; the single quotes around the 4 mean that it is being treated as just another character on the keyboard, not as the integer 4.

PRACTICE PROBLEMS

1. *newNumber* and *next* are integer variables in a C++ program. Write a statement to assign the value of *newNumber* to *next*.
2. What is the value of *average* after the following statements are executed? (*total* and *number* are type *int*, and *average* is type *double*.)

```
total = 277;
number = 5;
average = total/number;
```

3.3 *Control Statements*

We mentioned earlier that sequential flow of control is the default; that is, a program executes instructions sequentially from first to last. The flowchart in Figure 6 illustrates this, where S1, S2, . . ., Sk are program instructions (i.e., program statements).

As stated in Chapter 2, no matter how complicated the task to be done, only three types of control mechanisms are needed:

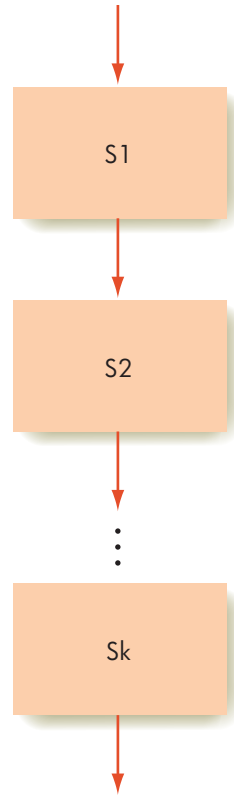
1. **Sequential:** Instructions are executed in order.
2. **Conditional:** Which instruction executes next depends on some condition.
3. **Looping:** A group of instructions may be executed many times.

Sequential flow of control, the default, is what occurs if the program does not contain any instances of the other two control structures. In the TravelPlanner program, for example, instructions are executed sequentially, beginning with the input statements, next the computation, and finally the output statement.

©2010 Course Technology, a part of Cengage Learning.

FIGURE 6

Sequential Flow of Control



In Chapter 2 we introduced pseudocode notation for conditional operations and looping. In Chapter 6 we learned how to write somewhat laborious assembly language code to implement conditional operations and looping. Now we'll see how C++ provides instructions that directly carry out these control structure mechanisms—more evidence of the power of high-level language instructions. We can think in a pseudocode algorithm design mode, as we did in Chapter 2, and then translate that pseudocode directly into C++ code.

Conditional flow of control begins with the evaluation of a **Boolean condition**, also called a **Boolean expression**, which can be either true or false. We discussed these "true/false conditions" in Chapter 2, and we also encountered Boolean expressions in Chapter 4, where they were used to design circuits. A Boolean condition often involves comparing the values of two expressions and determining whether they are equal, whether the first is greater than the second, and so on. Again assuming that *A*, *B*, and *C* are integer variables in a program, the following are legitimate Boolean conditions:

`A == 0`	(Does *A* currently have the value 0?)
`B < (A + C)`	(Is the current value of *B* less than the sum of the current values of *A* and *C*?)
`A != B`	(Does *A* currently have a different value than *B*?)

If the current values of *A*, *B*, and *C* are 2, 5, and 7, respectively, then the first condition is false (*A* does not have the value zero), the second condition is true (5 is less than 2 plus 7), and the third condition is true (*A* and *B* do not have equal values).

©2010 Course Technology, a part of Cengage Learning.

Comparisons need not be numeric. They can also be done between variables of type *char*, where the "ordering" is the usual alphabetic ordering. If *initial* is a value of type *char* with a current value of 'D', then

```
initial == 'F'
```

is false because *initial* does not have the value 'F', and

```
initial < 'P'
```

is true because 'D' precedes 'P' in the alphabet (or, more precisely, because the binary code for 'D' is numerically less than the binary code for 'P'). Note that the comparisons are case sensitive, so 'F' is not equal to 'f', but 'F' is less than 'f'.

Figure 7 shows the comparison operations available in C++. Note the use of the two equality signs to test whether two expressions have the same value. The single equality sign is used in an assignment statement, the double equality sign in a comparison.

Boolean conditions can be built up using the Boolean operators AND, OR, and NOT. Truth tables for these operators were given in Chapter 4 (Figures 4.12–4.14). The only new thing is the symbols that C++ uses for these operators, shown in Figure 8.

A conditional statement relies on the value of a Boolean condition (true or false) to decide which programming statement to execute next. If the condition is true, one statement is executed next, but if the condition is false, a different statement is executed next. Control is therefore no longer in a straight-line (sequential) flow, but hops to one place or to another. Figure 9 illustrates this situation. If the condition is true, the statement S1 is executed (and statement S2 is not); if the condition is false, the statement S2 is executed (and statement S1 is not). In either case, the flow of control then continues on to statement S3. We saw this same scenario when we discussed pseudocode conditional statements in Chapter 2 (Figure 2.4).

The C++ instruction that carries out conditional flow of control is called an **if-else** statement. It has the following form (note that the

FIGURE 7

C++ Comparison Operators

Comparison	Symbol	Example	Example Result
the same value as	==	2 == 5	false
less than	<	2 < 5	true
less than or equal to	<=	5 <= 5	true
greater than	>	2 > 5	false
greater than or equal to	>=	2 >= 5	false
not the same value as	!=	2 != 5	true

FIGURE 8

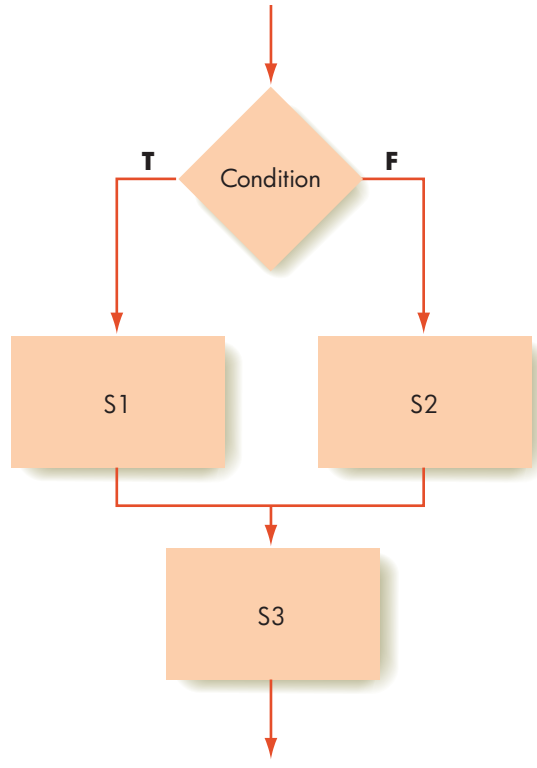
C++ Boolean Operators

Operator	Symbol	Example	Example Result
AND	&&	(2 < 5) && (2 > 7)	false
OR	||	(2 < 5) || (2 > 7)	true
NOT	!	!(2 == 5)	true

©2010 Course Technology, a part of Cengage Learning.

FIGURE 9

Conditional Flow of Control (if-else)



words *if* and *else* are lowercase and that the Boolean condition must be in parentheses).

```
if (Boolean condition)
        S1;
else
        S2;
```

Below is a simple if-else statement, where we assume that *A*, *B*, and *C* are integer variables.

```
if (B < (A + C))
     A = 2*A;
else
     A = 3*A;
```

Suppose that when this statement is reached, the values of *A*, *B*, and *C* are 2, 5, and 7, respectively. As we noted before, the condition *B* < (*A* + *C*) is then true, so the statement

```
A = 2*A;
```

is executed, and the value of *A* is changed to 4. However, suppose that when this statement is reached, the values of *A*, *B*, and *C* are 2, 10, and 7, respectively. Then the condition *B* < (*A* + *C*) is false, the statement

```
A = 3*A;
```

is executed, and the value of *A* is changed to 6.

©2010 Course Technology, a part of Cengage Learning.

A variation on the if-else statement is to allow an "empty else" case. Here we want to do something if the condition is true, but if the condition is false, we want to do nothing. Figure 10 illustrates the empty else case. If the condition is true, statement S1 is executed, and after that the flow of control continues on to statement S3, but if the condition is false, nothing happens except to move the flow of control directly on to statement S3.

This *if* variation on the if-else statement can be accomplished by omitting the word *else*. This form of the instruction therefore looks like

```
if (Boolean condition)
        S1;
```

We could write

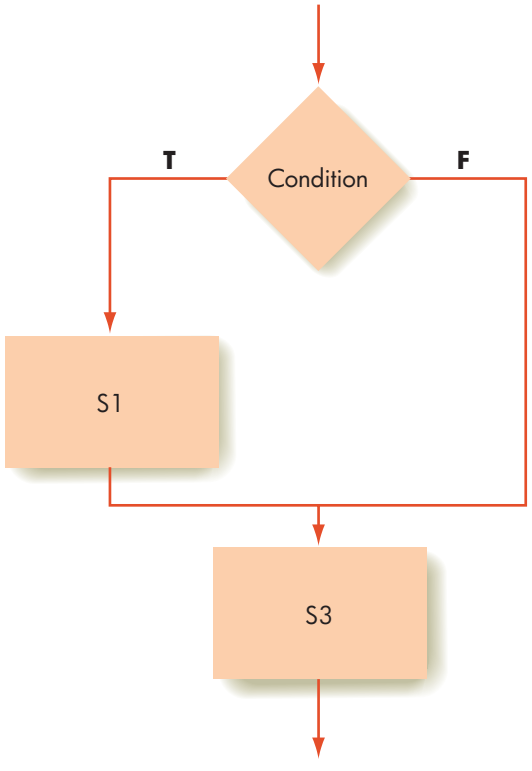
```
if (B < (A + C))
  A = 2*A;
```

This has the effect of doubling the value of A if the condition is true and of doing nothing if the condition is false.

It is possible to combine statements into a group by putting them within the curly braces { and }. The group is then treated as a single statement, called a **compound statement**. A compound statement can be used anywhere a single statement is allowed. For example,

```
{
  cout << "This is the first statement." << endl;
  cout << "This is the second statement." << endl;
  cout << "This is the third statement." << endl;
}
```

FIGURE 10
If-Else with Empty Else



©2010 Course Technology, a part of Cengage Learning.

is treated as a single statement. The implication is that in Figure 9, S1 or S2 might be compound statements. This makes the if-else statement potentially much more powerful and similar to the pseudocode conditional statement in Figure 2.9.

Let's expand on our TravelPlanner program and give the user of the program a choice of computing the time either as a decimal number (3.75 hours) or as hours and minutes (3 hours, 45 minutes). This situation is ideal for a conditional statement. Depending on what the user wants to do, the program does one of two tasks. For either task, the program still needs information about the speed and distance. The program must also collect information to indicate which task the user wishes to perform. We need an additional variable in the program to store this information. Let's use a variable called *choice* of type *char* to collect the user's choice of which task to perform. We also need two new integer variables to store the values of hours and minutes.

Figure 11 shows the new program, with the three additional declared variables. The condition evaluated at the beginning of the if-else statement tests whether *choice* has the value 'D'. If so, then the condition is true, and the first group of statements is executed—that is, the time is output in decimal format as we have been doing all along. If *choice* does not have the value 'D', then the condition is false. In this event, the second group of statements is executed. Note that because of the way the condition is written, if *choice* does not have the value 'D', it is assumed that the user wants to compute the time in hours and minutes, even though *choice* may have any other non-'D' value (including 'd') that the user may have typed in response to the prompt.

FIGURE 11

The TravelPlanner Program with a Conditional Statement

```
//Computes and outputs travel time
//for a given speed and distance
//Written by J. Q. Programmer, 6/28/10

#include <iostream>
using namespace std;

void main()
{
    int speed;          //rate of travel
    double distance;    //miles to travel
    double time;        //time needed for this travel
    int hours;          //time for travel in hours
    int minutes;        //leftover time in minutes
    char choice;        //choice of output as
                        //decimal hours
                        //or hours and minutes

    cout << "Enter your speed in mph: ";
    cin >> speed;
    cout << "Enter your distance in miles: ";
    cin >> distance;
    cout << "Enter your choice of format"
         << " for time, " << endl;
    cout << "decimal hours (D) "
         << "or hours and minutes (M): ";
    cin >> choice;
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 11

The TravelPlanner Program with a Conditional Statement (continued)

```
    if (choice == 'D')
    {
       time = distance/speed;
       cout << "At " << speed << " mph, "
            << "it will take " << endl;
       cout << time << " hours to travel "
            << distance << " miles." << endl;
    }
    else
    {
       time = distance/speed;
       hours = int(time);
       minutes = int((time - hours)*60);
       cout << "At " << speed << " mph, "
            << "it will take " << endl;
       cout << hours << " hours and "
            << minutes << " minutes to travel "
            << distance << " miles." << endl;
    }
}
```

To compute hours and minutes (the *else* clause of the if-else statement), time is computed in the usual way, which results in a decimal value. The whole number part of that decimal is the number of hours needed for the trip. We can get this number by type casting the decimal number to an integer. This is accomplished by

```
hours = int(time);
```

which drops all digits behind the decimal point and stores the resulting integer value in *hours*. To find the fractional part of the hour that we dropped, we subtract *hours* from *time*. We multiply this by 60 to turn it into some number of minutes, but this is still a decimal number. We do another type cast to truncate this to an integer value for *minutes*:

```
minutes = int((time - hours)*60);
```

For example, if the user enters data of 50 mph and 475 miles and requests output in hours and minutes, the following table shows the computed values.

Quantity	*Value*
speed	*50*
distance	*475*
time = distance/speed	*9.5*
hours = int(time)	*9*
time – hours	*0.5*
*(time – hours) *60*	*30.0*
*minutes = int((time – hours)*60)*	*30*

©2010 Course Technology, a part of Cengage Learning.

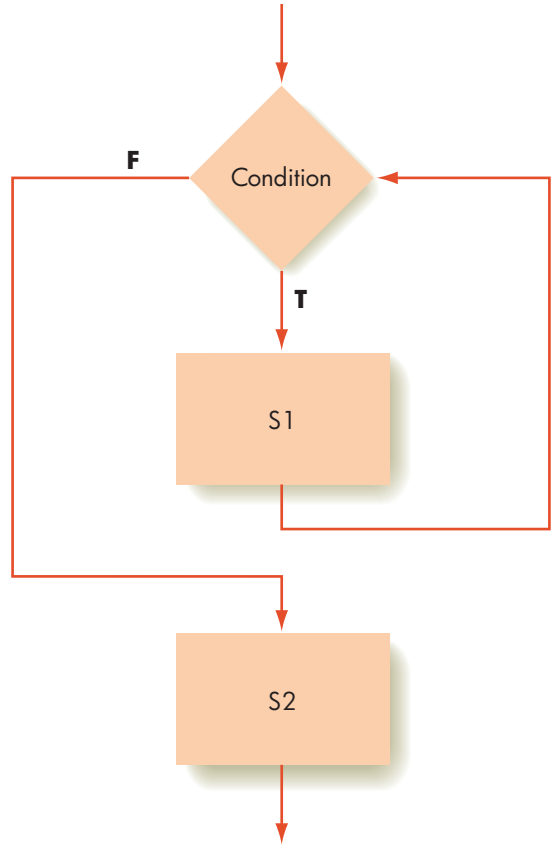
Here is the actual program output for this case:

```
Enter your speed in mph: 50
Enter your distance in miles: 475
Enter your choice of format for time,
decimal hours (D) or hours and minutes (M): M
At 50 mph, it will take
9 hours and 30 minutes to travel 475 miles.
```

The two statement groups in an if-else statement are identified by the enclosing curly braces, but in Figure 11 we also indented them to make them easier to pick out when looking at the program. Like comments, indentation is ignored by the computer but is valuable in helping people to more readily understand a program.

Now let's look at the third variation on flow of control, namely looping (iteration). We want to execute the same group of statements (called the **loop body**) repeatedly, depending on the result of a Boolean condition. As long as (while) the condition remains true, the loop body is executed. The condition is tested before each execution of the loop body. When the condition becomes false, the loop body is not executed again, which is usually expressed by saying that the algorithm *exits* the loop. To ensure that the algorithm ultimately exits the loop, the condition must be such that its truth value can be affected by what happens when the loop body is executed. Figure 12 illustrates

FIGURE 12
While Loop



©2010 Course Technology, a part of Cengage Learning.

the while loop. The loop body is statement S1 (which can be a compound statement), and S1 is executed *while* the condition is true. Once the condition is false, the flow of control moves on to statement S2. If the condition is false when it is first evaluated, then the body of the loop is never executed at all. We saw this same scenario when we discussed pseudocode looping statements in Chapter 2 (Figure 2.6).

C++ uses a while statement to implement this type of looping. The form of the statement is

```
while (Boolean condition)
    S1;
```

For example, suppose we want to write a program to add up a number of non-negative integers that the user supplies and write out the total. We need a variable to hold the total; we'll call this variable *sum*, and make its data type *int*. To handle the numbers to be added, we could declare a bunch of integer variables such as *n1*, *n2*, *n3*, . . . and do a series of input-and-add statements of the form

```
cin >> n1;
sum = sum + n1;
cin >> n2;
sum = sum + n2;
```

and so on. There are two problems with this approach. The first is that we may not know ahead of time how many numbers the user wants to add. If we declare variables *n1*, *n2*, . . ., *n25*, and the user wants to add 26 numbers, the program won't do the job. The second problem is that this approach requires too much effort. Suppose that we know the user wants to add 2000 numbers. We could declare 2000 variables (*n1*, . . ., *n2000*), and we could write the above input-and-add statements 2000 times, but it wouldn't be fun. Nor is it necessary—we are doing a very repetitive task here, and we should be able to use a loop mechanism to simplify the job. (We faced a similar situation in the first pass at a sequential search algorithm, Figure 2.11; our solution there was also to use iteration.)

Even if we use a loop mechanism, we are still adding a succession of values to *sum*. Unless we are sure that the value of *sum* is zero to begin with, we cannot be sure that the answer isn't nonsense. Remember that the identifier *sum* is simply an indirect way to designate a memory location in the computer. That memory location contains a pattern of bits, perhaps left over from whatever was stored there when some previous program was run. We cannot assume that just because this program hasn't used *sum*, its value is zero. (In contrast, the assembly language statement SUM: .DATA 0 reserves a memory location, assigns it the identifier SUM, and fills it with the value zero.) If we want the beginning value of *sum* to be zero, we must use an assignment statement. Using assignment statements to set the values of certain variables before they are used by the program is called **initialization of variables**.

Now on to the loop mechanism. First, let's note that once a number has been read in and added to *sum,* the program doesn't need to know the value of the number any longer. We can declare just one integer variable called *number*

©2010 Course Technology, a part of Cengage Learning.

and use it repeatedly to hold the first numerical value, then the second, and so on. The general idea is

```
sum = 0; //initialize sum
while (there are more numbers to add)
{
  cin >> number;
  sum = sum + number;
}
cout << "The total is " << sum << endl;
```

Now we have to figure out what the condition "there are more numbers to add" really means. Because we are adding nonnegative integers, we could ask the user to enter one extra integer that is not part of the legitimate data but is instead a signal that there *are* no more data. Such a value is called a **sentinel value**. For this problem, any negative number would be a good sentinel value. Because the numbers to be added are all nonnegative, the appearance of a negative number signals the end of the legitimate data. We don't want to process the sentinel value (because it is not a legitimate data item); we only want to use it to terminate the looping process. This might suggest the following code:

```
sum = 0;              //initialize sum
while (number >= 0)   //but there is a problem here,
                      //see following discussion
{
      cin >> number;
      sum = sum + number;
}
cout << "The total is " << sum << endl;
```

Here's the problem. How can we test whether *number* is greater than or equal to 0 if we haven't read the value of *number* yet? We need to do a preliminary input for the first value of *number* outside of the loop and then test that value in the loop condition. If it is nonnegative, we want to add it to *sum* and then read the next value and test it. Whenever the value of *number* is negative (including the first value), we want to do nothing with it—that is, we want to avoid executing the loop body. The following statements do this; we've also added instructions to the user.

```
sum = 0;           //initialize sum
cout << "Please enter numbers to add; ";
cout << "terminate with a negative number." << endl;
cin >> number;     //this will get the first value
while (number >= 0)
{
      sum = sum + number;
      cin >> number;
}
cout << "The total is " << sum << endl;
```

The value of *number* gets changed within the loop body by reading in a new value. The new value is tested, and if it is nonnegative, the loop body executes

©2010 Course Technology, a part of Cengage Learning.

again, adding the data value to *sum* and reading in a new value for *number*. The loop terminates when a negative value is read in. Remember the requirement that something within the loop body must be able to affect the truth value of the condition. In this case, it is reading in a new value for *number* that has the potential to change the value of the condition from true to false. Without this requirement, the condition, once true, would remain true forever, and the loop body would be endlessly executed. This results in what is called an **infinite loop**. A program that contains an infinite loop will execute forever (or until the programmer gets tired of waiting and interrupts the program, or until the program exceeds some preset time limit).

Here is a sample of the program output.

```
Please enter numbers to add; terminate with a
negative number.
5
6
10
-1
The total is 21
```

The problem we've solved here, adding nonnegative integers until a negative sentinel value occurs, is the same one solved using assembly language in Chapter 6. The preceding C++ code is almost identical to the pseudocode version of the algorithm shown in Figure 6.7. Thanks to the power of the language, the C++ code embodies the algorithm directly, at a high level of thinking, whereas in assembly language this same algorithm had to be translated into the lengthy and awkward code of Figure 6.8.

To process data for a number of different trips in the TravelPlanner program, we could use a while loop. During each pass through the loop, the program computes the time for a given speed and distance. The body of the loop is therefore exactly like our previous code. All we are adding here is the framework that provides looping. To terminate the loop, we could use a sentinel value, as we did for the program above. A negative value for *speed*, for example, is not a valid value and could serve as a sentinel value. Instead of that, let's allow the user to control loop termination by having the program ask the user whether he or she wishes to continue. We'll need a variable to hold the user's response to this question. Of course, the user could answer "N" at the first query, the loop body would never be executed at all, and the program would terminate. Figure 13 shows the complete program.

FIGURE 13

The TravelPlanner Program with Looping

```
//Computes and outputs travel time
//for a given speed and distance
//Written by J. Q. Programmer, 7/05/10

#include <iostream>
using namespace std;

void main()
{
    int speed;              //rate of travel
    double distance;        //miles to travel
    double time;            //time needed for this travel
    int hours;              //time for travel in hours
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 13

The TravelPlanner Program with Looping (continued)

```
    int minutes;              //leftover time in minutes
    char choice;              //choice of output as
                              //decimal hours
                              //or hours and minutes
    char more;                //user's choice to do
                              //another trip

    cout << "Do you want to plan a trip? "
         << "(Y or N): ";
    cin >> more;

    while (more == 'Y') //more trips to plan
    {
        cout << "Enter your speed in mph: ";
        cin >> speed;
        cout << "Enter your distance in miles: ";
        cin >> distance;
        cout << "Enter your choice of format"
             << " for time, " << endl;
        cout << "decimal hours (D) "
             << "or hours and minutes (M): ";
        cin >> choice;

        if (choice == 'D')
        {
            time = distance/speed;
            cout << "At " << speed << " mph, "
                 << "it will take " << endl;
            cout << time << " hours to travel "
                 << distance << " miles." << endl;
        }
        else
        {
            time = distance/speed;
            hours = int(time);
            minutes = int((time - hours)*60);
            cout << "At " << speed << " mph, "
                 << "it will take " << endl;
            cout << hours << " hours and "
                 << minutes << " minutes to travel "
                 << distance << " miles." << endl;
        }

        cout << endl;
        cout << "Do you want to plan another trip? "
             << "(Y or N): ";
        cin >> more;
    } //end of while loop
}
```


©2010 Course Technology, a part of Cengage Learning.

PRACTICE PROBLEMS

Assume all variables have previously been declared.

1. What is the output from the following section of code?

```
number1 = 15;
number2 = 7;
if (number1 >= number2)
     cout << 2*number1 << endl;
else
     cout << 2*number2 << endl;
```

2. What is the output from the following section of code?

```
scores = 1;
while (scores < 20)
{
     scores = scores + 2;
     cout << scores << endl;
}
```

3. What is the output from the following section of code?

```
quotaThisMonth = 7;
quotaLastMonth = quotaThisMonth + 1;
if ((quotaThisMonth > quotaLastMonth)||
    (quotaLastMonth >= 8))
{
     cout << "Yes";
     quotaLastMonth = quotaLastMonth + 1;
}
else
{
     cout << "No";
     quotaThisMonth = quotaThisMonth + 1;
}
```

4. How many times is the *cout* statement executed in the following section of code?

```
left = 10;
right = 20;
while (left <= right)
{
     cout << left << endl;
     left = left + 2;
}
```

5. Write a C++ statement that outputs "Equal" if the integer values of *night* and *day* are the same, but otherwise does nothing.

©2010 Course Technology, a part of Cengage Learning.

4 Another Example

Let's briefly review the types of C++ programming statements we've learned. We can do input and output—reading values from the user into memory, writing values out of memory for the user to see, being sure to use meaningful variable identifiers to reference memory locations. We can assign values to variables within the program. And we can direct the flow of control by using conditional statements or looping. Although many other statement types are available in C++, you can do almost everything using only the modest collection of statements we have described. The power of C++ lies in how these statements are combined and nested within groups to produce ever more complex courses of action.

For example, suppose we write a program to assist SportsWorld, a company that installs circular swimming pools. In order to estimate their costs for swimming pool covers or for fencing to surround the pool, SportsWorld needs to know the area or circumference of a pool, given its radius. A pseudocode version of the program is shown in Figure 14.

We should be able to translate this pseudocode fairly directly into the body of the main function. Other things we need to add to complete the program are:

- A prologue comment to explain what the program does (optional but always recommended for program documentation)
- An include directive for *iostream* and a using directive for namespace *std* (necessary because our program uses *cin* and *cout*)
- A declaration for the constant value PI (3.1416)
- Variable declarations
- Some output formatting to control the number of digits behind the decimal point

Figure 15 gives the complete program. Figure 16 shows what actually appears on the screen when this program is executed with some sample data.

FIGURE 14

A Pseudocode Version of the SportsWorld Program

```
Get value for user's choice about continuing
While user wants to continue, do the following steps
    Get value for pool radius
    Get value for choice of task
    If task choice is circumference
        Compute pool circumference
        Print output
    Else (task choice is area)
        Compute pool area
        Print output
    Get value for user's choice about continuing
Stop
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 15

The SportsWorld Program

```
//This program helps SportsWorld estimate costs
//for pool covers and pool fencing by computing
//the area or circumference of a circle
//with a given radius.
//Any number of circles can be processed.
//Written by M. Phelps, 10/05/10

#include <iostream>
using namespace std;

void main()
{
    const double PI = 3.1416;//value of pi
    double radius;           //radius of a pool - given
    double circumference;    //circumference of a pool -
                             //computed
    double area;             //area of a pool -
                             //computed
    char taskToDo;           //holds user choice to
                             //compute circumference
                             //or area
    char more;               //controls loop for
                             //processing more pools

    cout.setf(ios::fixed);
    cout.precision(2);

    cout << "Do you want to process a pool? (Y or N): ";
    cin >> more;
    cout << endl;

    while (more == 'Y') //more circles to process
    {
        cout << "Enter the value of the radius of a "
             << "pool: ";
        cin >> radius;

        //See what user wants to compute
        cout << "Enter your choice of task." << endl;
        cout << "C to compute circumference, "
             << "A to compute area: ";
        cin >> taskToDo;

        if (taskToDo == 'C') //compute circumference
        {
            circumference = 2*PI*radius;
            cout << "The circumference for a pool "
                 << "of radius " << radius << " is "
                 << circumference << endl;
        }
        else              //compute area
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 15

The SportsWorld Program (continued)

```
        {
            area = PI*radius*radius;
            cout << "The area for a pool "
                 << "of radius " << radius << " is "
                 << area << endl;
        }
        cout << endl;
        cout << "Do you want to process more pools? "
             << "(Y or N): ";
        cin >> more;
        cout << endl;
    } //end of while loop

    //finish up
    cout << "Program will now terminate." << endl;
}
```

FIGURE 16

A Sample Session Using the Program of Figure 15

```
Do you want to process a pool? (Y or N): Y
Enter the value of the radius of a pool: 2.7
Enter your choice of task.
C to compute circumference, A to compute area: C
The circumference for a pool of radius 2.70 is 16.96
Do you want to process more pools? (Y or N): Y
Enter the value of the radius of a pool: 2.7
Enter your choice of task.
C to compute circumference, A to compute area: A
The area for a pool of radius 2.70 is 22.90
Do you want to process more pools? (Y or N): Y
Enter the value of the radius of a pool: 14.53
Enter your choice of task.
C to compute circumference, A to compute area: C
The circumference for a pool of radius 14.53 is 91.29
Do you want to process more pools? (Y or N): N
Program will now terminate.
```

PRACTICE PROBLEMS

1. Write a complete C++ program to read in the user's first and last initials and write them out. (*Hint*: By using multiple extraction operators, you can use a single input statement to collect more than one value from the input stream.)
2. Write a complete C++ program that asks for the price of an item and the quantity purchased, and writes out the total cost.

©2010 Course Technology, a part of Cengage Learning.

PRACTICE PROBLEMS

3. Write a complete C++ program that asks for a number. If the number is less than 5, it is written out, but if it is greater than or equal to 5, twice that number is written out.
4. Write a complete C++ program that asks the user for a positive integer number and then writes out all the numbers from 1 up to and including that number.

5 Managing Complexity

The programs we have written have been relatively simple. More complex problems require more complex programs to solve them. Although it is fairly easy to understand what is happening in the 40 or so lines of the SportsWorld program, imagine trying to understand a program that is 50,000 lines long. Imagine trying to write such a program! It is not possible to understand—all at once—everything that goes on in a 50,000-line program.

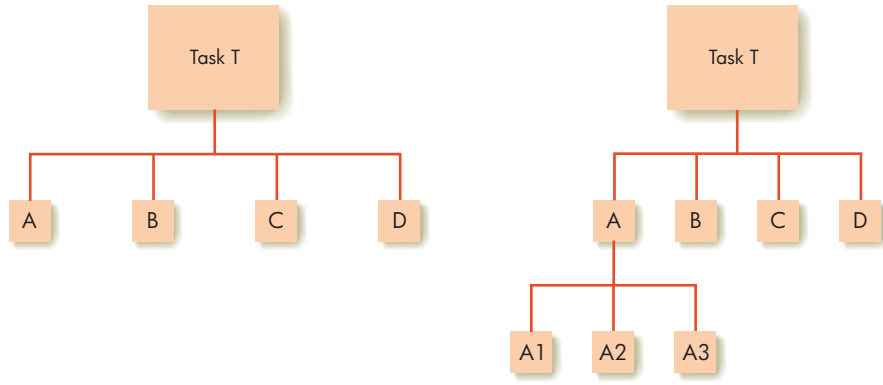
5.1 *Divide and Conquer*

Writing large programs is an exercise in managing complexity. The solution is a problem-solving approach called **divide and conquer**. Suppose a program is to be written to do a certain task; let's call it task T. Suppose further that we can divide this task into smaller tasks, say A, B, C, and D, such that, if we can do those four tasks in the right order, we can do task T. Then our high-level understanding of the problem need only be concerned with *what* A, B, C, and D do and how they must work together to accomplish T. We do not, at this stage, need to understand *how* A, B, C, and D can be done. Figure 17(a), an example of a **structure chart** or **structure diagram**, illustrates this situation. Task T is composed in some way of subtasks A, B, C, and D. Later we can turn our attention to, say, subtask A and see if it too can be decomposed into smaller subtasks, as in Figure 17(b). In this way, we continue to break the task down into smaller and smaller pieces, finally arriving at subtasks that are simple enough that it is easy to write the code to carry them out. By *dividing* the problem into small pieces, we can *conquer* the complexity that is overwhelming if we look at the problem as a whole.

Divide and conquer is a problem-solving approach and not just a computer programming technique. Outlining a term paper into major and minor topics is a divide-and-conquer approach to writing the paper. Doing a Form 1040 Individual Tax Return for the Internal Revenue Service can involve the subtasks of completing Schedules A, B, C, D, and so on and then reassembling the results. Designing a house can be broken down into subtasks of designing floor plans, wiring, plumbing, and the like. Large companies

©2010 Course Technology, a part of Cengage Learning.

FIGURE 17
Structure Charts



(a) Basic decomposition

(b) More detailed decomposition

organize their management responsibilities using a divide-and-conquer approach; what we have called structure charts become, in the business world, organization charts.

How is the divide-and-conquer problem-solving approach reflected in the resulting computer program? If we think about the problem in terms of subtasks, then the program should show that same structure; that is, part of the code should do subtask A, part should do subtask B, and so on. We divide the code into *modules* or *subprograms*, each of which does some part of the overall task. Then we empower these modules to work together to solve the original problem.

5.2 *Using Functions*

In C++, modules of code are called **functions**. Each function in a program should do one and only one subtask. These "subtask functions" are the optional functions listed before the mandatory main function in the C++ program outline of Figure 2. When subtask functions are used, the main function consists primarily of invoking these subtask functions in the correct order.

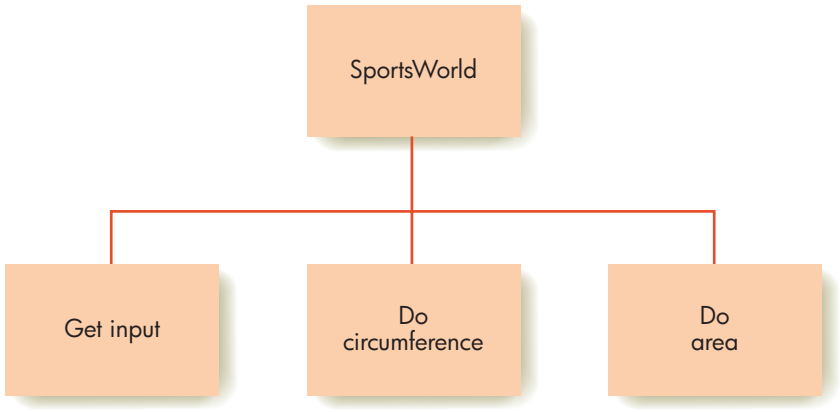
Let's review the main function body of the SportsWorld program (Figure 15) with an eye to further subdividing the task. There is a loop that does some operations as long as the user wants. What gets done? Input is obtained from the user about the radius of the circle and the choice of task to be done (compute circumference or compute area). Then the circumference or the area gets computed and written out. We've identified three subtasks, as shown in the structure chart of Figure 18.

We can visualize the main function body of the program at a pseudocode level, as shown in Figure 19. This divide-and-conquer approach to solving the problem can (and should) be planned first in pseudocode, without regard to the details of the programming language to be used. If the three subtasks (input, circumference, area) can all be done, then arranging them within the structure of Figure 19 solves the problem. We can write a function for each of the subtasks. Although we now know what form the main function body will take, we have pushed the details of how to do each of the subtasks off into the other functions. Execution of the program begins with the main function. Every time the flow of control reaches the equivalent of a "do subtask"

©2010 Course Technology, a part of Cengage Learning.

FIGURE 18

Structure Chart for the SportsWorld Task



instruction, it transfers execution to the appropriate function code. When execution of the function code is complete, flow of control returns to the main function and picks up where it left off.

Before we look at the details of how to write a function, we need to examine the mechanism that allows the functions to work with each other and with the main function. This mechanism consists of passing information about various quantities in the program back and forth between the other functions and the main function. Because each function is doing only one subtask of the entire task, it does not need to know the values of all variables in the program. It only needs to know the values of the variables with which its particular subtask is concerned. Allowing a function access only to pertinent variables prevents that function from inadvertently changing a value it has no business changing.

When the main function wants a subtask function to be executed, it gives the name of the function (which is an ordinary C++ identifier) and also a list of the identifiers for variables pertinent to that function. This is called an **argument list**. In our SportsWorld program, let's name the three functions *getInput*, *doCircumference*, and *doArea* (names that are descriptive of the subtasks these functions carry out). The *getInput* function collects the values for the variables *radius* and *taskToDo*. The main program invokes the *getInput* function with the statement

```
getInput(radius, taskToDo);
```

FIGURE 19

A High-Level Modular View of the SportsWorld Program

```
Get value for user's choice about continuing
While the user wants to continue
  Do the input subtask
  If (Task = 'C') then
    do the circumference subtask
  else
    do the area subtask
  Get value for user's choice about continuing
```


©2010 Course Technology, a part of Cengage Learning.

which takes the place of the "Do the input subtask" line in Figure 19. When this statement is reached, control passes to the *getInput* function. After execution of this function, control returns to the main function, and the variables *radius* and *taskToDo* have the values obtained for them within *getInput*.

The *doCircumference* function computes and writes out the value of the circumference, and in order to do that it needs to know the radius. Therefore, the variable *radius* is a legitimate argument for this function. The main function contains the statement

```
doCircumference(radius);
```

in place of the "do the circumference subtask" in Figure 19. When this statement is reached, the variable *radius* conveys the value of the radius to the *doCircumference* function, and the function computes and writes out the circumference. The variable *circumference*, then, is also of interest to the *doCircumference* function, but it is of interest to this function alone, in the sense that *doCircumference* does the computation and writes out the result. No other use is made of the circumference in the entire program, so no other function, including the main function, has anything to do with *circumference*. Instead of being declared in the body of the main function, *circumference* will be declared (and can be used) only within the *doCircumference* function; it will be **local** to that function. Any function can have its own **local constants** and **local variables**, declared within and known only to that function.

The *doCircumference* function also needs to know the value of the constant *PI*. We could declare *PI* as a constant local to *doCircumference,* but *doArea* needs the same constant, so we will declare *PI* right after the program *#include* directives, not within any function. This will make *PI* a **global constant** whose value is known everywhere. The value of a constant cannot be changed, so there is no reason to prevent any function from having access to its value.

The *doArea* function computes and writes out the area and needs to know the radius, so the line "do the area subtask" in Figure 19 is replaced by

```
doArea(radius);
```

Within *doArea*, *area* is a local variable.

Now we can write the main function of the modularized version of the SportsWorld program, shown in Figure 20. The main function body is a direct translation of Figure 19. If, in starting from scratch to write this program, we had taken a divide-and-conquer approach, broken the original problem down into three subtasks, and come up with the outline of Figure 19, it would have been easy to get from there to Figure 20. The only additional task would have been determining the variables needed.

At a glance, the main function in Figure 20 does not look a great deal different from our former main function. However, it is conceptually quite different. The subtasks of getting the input values, computing and writing out the circumference, and computing and writing out the area have been relegated to functions. The details (such as the formulas for computing circumference and area) are now hidden and have been replaced by function invocations. If these subtasks had required many lines of code, our new main function would indeed be shorter—and easier to understand—than before.

©2010 Course Technology, a part of Cengage Learning.

FIGURE 20

The Main Function in a Modularized Version of the SportsWorld Program

```
void main()
{
    double radius;      //radius of a pool - given
    char taskToDo;      //holds user choice to
                        //compute circumference or area
    char more;          //controls loop for
                        //processing more pools

    cout.setf(ios::fixed);
    cout.precision(2);

    cout << "Do you want to process a pool? (Y or N): ";
    cin >> more;
    cout << endl;
    while (more == 'Y') //more circles to process
    {
        getInput(radius, taskToDo);

        if (taskToDo == 'C') //compute circumference
            doCircumference(radius);
        else //compute area
            doArea(radius);

        cout << endl;
        cout << "Do you want to process more pools? (Y or N): ";
        cin >> more;
        cout << endl;
    }

    //finish up
    cout << "Program will now terminate." << endl;
}
```

5.3 *Writing Functions*

Now we know how the main function can invoke another function. (In fact, using the same process, any function can invoke another function. A function can even invoke itself.) It is time to see how to write the code for these other, nonmain functions. The general outline for a C++ function is shown in Figure 21.

FIGURE 21

The Outline for a C++ Function

```
function header
{
  local declarations   [optional]
  function body
}
```


©2010 Course Technology, a part of Cengage Learning.

Where Art Thou, C++?

The C++ programming language has been used in a variety of applications. The following list is a sampling:

- 3D modeling and animation tools used in movies, video game development, architecture, and medical simulation
- Telecommunications network failure recovery software
- Web search engines
- Chip design and manufacturing software
- Mars Rover autonomous driving system, including scene analysis and route planning
- Electronic-controlled fuel injection system for very large diesel engines used in container ships and tankers
- Operating systems for cellular telephones
- Software for tracking orders on European stock markets
- Data analysis for large high-energy physics experiments
- The Apple iPod user interface
- Environments and tools for software development
- Support for data center services such as travel reservation systems, vital statistics (birth, death, marriage) registration, and patient medical records
- The "classic" Seti@home project, enlisting the use of idle home computers to search for signs of extraterrestrial life. From 1999–2005, nearly 5.5 million users generated 2,092,538,656 results by means of about 2,433,980 years of CPU time. The Seti@home project continues on a different software platform.

The function header consists of three parts:

- A return indicator
- The function identifier
- A parameter list

The **return indicator** classifies a function as a "void" or a "nonvoid" function. We'll explain this distinction later, but the three functions for the circle program are all void functions, so the return indicator is the keyword **void**. (All of our main functions have been void functions as well.) The **function identifier** can be any legitimate C++ identifier. The parameters in the **parameter list** correspond to the arguments in the statement that invoke this function; that is, the first parameter in the list matches the first argument given in the statement that invokes the function, the second parameter matches the second argument, and so on. It is through this correspondence between parameters and arguments that information (data) flows from the main function to other functions, and vice versa. The data type of each parameter must be given as part of the parameter list, and it must match the data type of the corresponding argument. For example, because the *getInput* function is invoked with the two arguments *radius* and *taskToDo*, the parameter list for the *getInput* function header has two parameters, the first of type **double** and the second of type **char**. Parameters may have, but do not have to have, the same identifiers as the corresponding arguments; arguments and parameters correspond by virtue of their respective positions in the argument list and the parameter list, regardless of the identifiers used. For the *getInput* function, we choose the parameter identifiers *radius* and *taskToDo*, matching the argument identifiers. No semicolon is used at the end of a function header.

One additional aspect of the parameter list in the function header concerns the use the function will make of each parameter. Consider the statement that invokes the function; an argument in the invoking statement carries a data value to the corresponding parameter in the function header.

©2010 Course Technology, a part of Cengage Learning.

If the value is one that the function must know to do its job but should not change, then the argument is **passed by value**. The function receives a copy of the data value but never knows the memory location where the original value is stored. If the function changes the value of its copy, this change has no effect when control returns to the main function. If, however, the value passed to the function is one that the function should change, and the main function should know the new value, then the argument is **passed by reference**. The function receives access to the memory location where the value is stored, and any changes it makes to the value are seen by the main function after control returns there. Included in this category are arguments whose values are unknown when the function is invoked (which really means that they are meaningless values of whatever happens to be in the memory location associated with that identifier), but the function changes those unknown values into meaningful values.

By default, arguments in C++ are passed by value, which protects them from change by the function. Explicit action must be taken by the programmer to pass an argument by reference; specifically, the ampersand symbol (&) must appear in front of the corresponding parameter in the function parameter list.

How do we decide whether to pass an argument by value or by reference? If the main function needs to obtain a new value back from a function when execution of that function terminates, then the argument must be passed by reference (by inserting the & into the parameter list). Otherwise, the argument should be passed by value, the default arrangement.

In the *getInput* function, both *radius* and *taskToDo* are values that *getInput* obtains from the user and that the main function needs to know when *getInput* terminates, so both of these are passed by reference. The header for the *getInput* function is shown below, along with the invoking statement from the main function. Note that the parameters *radius* and *taskToDo* are in the right order, have been given the correct data types, and are both marked for passing by reference. Also remember that, although the arguments are named *radius* and *taskToDo* because those are the variable identifiers declared in the main function, the parameters could have different identifiers, and it is the parameter identifiers that are used within the body of the function.

```
                                        //function header
void getInput(double &radius, char &taskToDo)

getInput(radius, taskToDo);     //function invocation
```

The body of the *getInput* function comes from the corresponding part of Figure 15. If we hadn't already written this code, we could have done a pseudocode plan first. The complete function appears in Figure 22, where a comment has been added to document the purpose of the function.

The *doCircumference* function needs to know the value of *radius* but does not change that value. Therefore, *radius* is passed by value. Why is the distinction between arguments passed by value and those passed by reference important? If functions are to affect any changes at all, then clearly reference parameters are necessary, but why not just make everything a reference parameter? Suppose that in this example *radius* is made a reference parameter. If an instruction within *doCircumference* were to inadvertently change the value of *radius,* then that new value would be returned to the main function, and any subsequent calculations using this value (there are

©2010 Course Technology, a part of Cengage Learning.

FIGURE 22

The getInput Function

```
void getInput(double &radius, char &taskToDo)
//gets radius and choice of task from the user
{
    cout << "Enter the value of the radius "
         << "of a pool: ";
    cin >> radius;

    //See what user wants to compute
    cout << "Enter your choice of task." << endl;
    cout << "C to compute circumference, A to compute area: ";
    cin >> taskToDo;
}
```

none in this example) would be in error. Making *radius* a value parameter prevents this. How could one possibly write a program statement that changes the value of a variable inadvertently? In something as short and simple as our example, this probably would not happen, but in a more complicated program, it might. Distinguishing between passing by value and passing by reference is just a further step in controlling a function's access to data values, to limit the damage the function might do. The code for the *doCircumference* function appears in Figure 23.

The *doArea* function is very similar. Let's reassemble everything and give the complete modularized version of the program. In Figure 24, only the main function needs to know the value of *more*. No other function needs access to this value, so this variable is never passed as an argument. The main function header

```
void main()
```

also follows the form for any function header. In other words, the main function truly is a C++ function. It has an empty parameter list because it is the starting point for the program, and there's no other place that could pass argument values to it.

Because it seems to have been a lot of effort to arrive at this complete, modularized version of our SportsWorld program (which, after all, does the same thing as the program in Figure 15), let's review why this effort is worthwhile.

FIGURE 23

The doCircumference Function

```
void doCircumference(double radius)
//computes and writes out the circumference of
//a circle with given radius
{
    double circumference;
    circumference = 2*PI*radius;
    cout << "The circumference for a pool "
         << "of radius " << radius << " is "
         << circumference << endl;
}
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 24

The Complete Modularized SportsWorld Program

```
//This program helps SportsWorld estimate costs
//for pool covers and pool fencing by computing
//the area or circumference of a circle
//with a given radius.
//Any number of circles can be processed.
//Illustrates functions
//Written by M. Phelps, 10/23/10

#include <iostream>
using namespace std;

const double PI = 3.1416; //value of pi

void getInput(double &radius, char &taskToDo)
//gets radius and choice of task from the user
{
    cout << "Enter the value of the radius "
         << "of a pool: ";
    cin >> radius;

    //See what user wants to compute
    cout << "Enter your choice of task." << endl;
    cout << "C to compute circumference, A to compute area: ";
    cin >> taskToDo;
}

void doCircumference(double radius)
//computes and writes out the circumference of
//a circle with given radius
{
    double circumference;
    circumference = 2*PI*radius;
    cout << "The circumference for a pool "
         << "of radius " << radius << " is "
         << circumference << endl;
}

void doArea (double radius)
//computes and writes out the area of
//a circle with given radius
{
    double area;
    area = PI*radius*radius;
    cout << "The area for a pool "
         << "of radius " << radius << " is "
         << area << endl;
}

void main()
{
    double radius;     //radius of a pool - given
    char taskToDo;     //holds user choice to
                       //compute circumference or area
    char more;         //controls loop for
                       //processing more pools
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 24

The Complete Modularized SportsWorld Program (continued)

```
    cout.setf(ios::fixed);
    cout.precision(2);

    cout << "Do you want to process a pool? (Y or N): ";
    cin >> more;
    cout << endl;
    while (more == 'Y') //more circles to process
    {
      getInput(radius, taskToDo);
      if (taskToDo == 'C') //compute circumference
         doCircumference(radius);
      else  //compute area
         doArea(radius);

      cout << endl;
      cout << "Do you want to process more pools? (Y or N): ";
      cin >> more;
      cout << endl;
    }

    //finish up
    cout << "Program will now terminate." << endl;
}
```

The modularized version of the program is compartmentalized in two ways. First, it is compartmentalized with respect to task. The major task is accomplished by a series of subtasks, and the work for each subtask takes place within a separate function. This leaves the main function free of details and consisting primarily of invoking the appropriate function at the appropriate point. As an analogy, think of the president of a company calling on various assistants to carry out tasks as needed. The president does not need to know *how* a task is done, only the name of the person responsible for carrying it out. Second, the program is compartmentalized with respect to data, in the sense that the data values known to the various functions are controlled by parameter lists and by the use of value instead of reference parameters where appropriate. In our analogy, the president gives each assistant the information he or she needs to do the assigned task, and expects relevant information to be returned—but not all assistants know all information.

This compartmentalization is useful in many ways. It is useful when we *plan the solution* to a problem, because it allows us to use a divide-and-conquer approach. We can think about the problem in terms of subtasks. This makes it easier for us to understand how to achieve a solution to a large and complex problem. It is also useful when we *code the solution* to a problem, because it allows us to concentrate on writing one section of the code at a time. We can write a function and then fit it into the program, so that the program gradually expands rather than having to be written all at once. Developing a large software project is a team effort, and different parts of the team can be writing different functions at the same time. It is useful when we *test the program*, because we can test one new function at a time as the program grows, and any errors are localized to the function being added. (The main function can be tested early by writing appropriate headers with empty bodies for the remaining functions.) Compartmentalization is useful when we *modify the program*, because changes

©2010 Course Technology, a part of Cengage Learning.

tend to be within certain subtasks and hence within certain functions in the code. And finally it is useful for anyone (including the programmer) who wants to *read* the resulting program. The overall idea of how the program works, without the details, can be gleaned from reading the main function; if and when the details become important, the appropriate code for the other functions can be consulted. In other words, modularizing a program is useful for its

- Planning
- Coding
- Testing
- Modifying
- Reading

A special type of C++ function can be written to compute a single value rather than to carry out a subtask. For example, *doCircumference* does everything connected with the circumference, both calculating the value and writing it out. We can write a different *doCircumference* function that only computes the value of the circumference and then returns that value to the main function, which writes it out. A function that returns a single value to the section of the program that invoked it is a **nonvoid** function. Instead of using the word *void* as the return indicator in the function header, a nonvoid function uses the data type of the single returned value. In addition, a nonvoid function must contain a return statement, which consists of the keyword **return** followed by an expression for the value to be returned. (This explains why we have always written the main function as a void function; it is never invoked anywhere else in the program and does not return a value.)

The code for this new *doCircumference* function would be simply

```
double doCircumference(double radius)
//computes the circumference of a circle
//with given radius
{
  return 2*PI*radius;
}
```

A nonvoid function is invoked wherever the returned value is to be used, rather than in a separate statement. For example, the statement

```
cout << doCircumference(radius);
```

invokes the *doCircumference* function by giving its name and argument, and this invocation actually becomes the value returned by the *doCircumference* function, which is then written out.

Figure 25 shows a third version of the SportsWorld program using nonvoid *doCircumference* and *doArea* functions. There are no variables anywhere for the circumference and the area of the circle. The *doCircumference* and *doArea* functions use the usual formulas for their computations, but instead of using local variables for circumference and area, we've compressed the code into a single return statement. These functions are now invoked within an output statement, so the values get printed out without being stored anywhere.

Figure 26 summarizes several sets of terms introduced in this section.

©2010 Course Technology, a part of Cengage Learning.

FIGURE 25

The SportsWorld Program Using Nonvoid Functions

```
//This program helps SportsWorld estimate costs
//for pool covers and pool fencing by computing
//the area or circumference of a circle
//with a given radius.
//Any number of circles can be processed.
//Illustrates nonvoid functions
//Written by M. Phelps, 10/23/10

#include <iostream>
using namespace std;

const double PI = 3.1416; //value of pi

void getInput(double &radius, char &taskToDo)
//gets radius and choice of task from the user
{
    cout << "Enter the value of the radius "
         << "of a pool: ";
    cin >> radius;

    //See what user wants to compute
    cout << "Enter your choice of task." << endl;
    cout << "C to compute circumference, A to compute area: ";
    cin >> taskToDo;
}

double doCircumference(double radius)
//computes the circumference of a circle with given radius
{
   return 2*PI*radius;
}

double doArea(double radius)
//computes the area of a circle with given radius
{
   return PI*radius*radius;
}

void main()
{
    double radius; //radius of a circle - given
    char taskToDo; //holds user choice to
                   //compute circumference or area
    char more;     //controls loop for processing more circles

    cout.setf(ios::fixed);
    cout.precision(2);

    cout << "Do you want to process a pool? (Y or N): ";
    cin >> more;
    cout << endl;
    while (more == 'Y') //more circles to process
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 25

The SportsWorld Program Using Nonvoid Functions (continued)

```
    {
        getInput(radius, taskToDo);
        if (taskToDo == 'C') //compute circumference
        {
           cout << "The circumference for a pool "
                << "of radius " << radius << " is "
                << doCircumference(radius) << endl;
        }
        else               //compute area
        {
           cout << "The area for a pool "
                << "of radius " << radius << " is "
                << doArea(radius) << endl;
        }
        cout << endl;
        cout << "Do you want to process more pools? (Y or N): ";
        cin >> more;
        cout << endl;
    }

    //finish up
    cout << "Program will now terminate." << endl;
}
```

FIGURE 26

Some C++ Terminology

Term	Meaning	Term	Meaning
Local variable	Declared and known only within a function	Global constant	Declared outside any function and known everywhere
Argument passed by value	Function receives a copy of the value and can make no permanent changes in the value	Argument passed by reference	Function gets access to memory location where the value is stored; changes it makes to the value persist after control returns to main function
Void function	Performs a task, function invocation is a complete C++ statement	Nonvoid function	Computes a value; must include a return statement; function invocation is used within another C++ statement

PRACTICE PROBLEMS

1. What is the output of the following C++ program?

```
#include <iostream>
using namespace std;
void doIt(int &number)
{
  number = number + 4;
}
```


©2010 Course Technology, a part of Cengage Learning.

```
void main()
{
  int number;

  number = 7;
  doIt(number);
  cout << number << endl;
}
```

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

```
#include <iostream>
using namespace std;
void doIt(int number)
{
  number = number + 4;
}
void main()
{
  int number;

  number = 7;
  doIt(number);
  cout << number << endl;
}
```

3. Write a C++ function that performs an input task for the main program, collecting two integer values *one* and *two* from the user.

4. Suppose a nonvoid function called *tax* gets a value *subtotal* from the main function, multiplies it by a global constant tax rate called *RATE*, and returns the resulting tax value. All quantities are type *double*.
   a. Write the function header.
   b. Write the return statement in the function body.
   c. Write the statement in the main program that writes out the tax.

6 Object-Oriented Programming

6.1 *What Is It?*

The divide-and-conquer approach to programming is a "traditional" approach. The focus is on the overall task to be done: How to break it down into subtasks, and how to write algorithms for the various subtasks that are carried out by communicating modules (in the case of C++, by functions). The program can be thought of as a giant statement executor designed to carry out the major task, even though the main function may simply call on, in turn, the various other modules that do the subtask work.

Object-oriented programming (OOP) takes a somewhat different approach. A program is considered a simulation of some part of the world that is

©2010 Course Technology, a part of Cengage Learning.

the domain of interest. "Objects" populate this domain. Objects in a banking system, for example, might be savings accounts, checking accounts, and loans. Objects in a company personnel system might be employees. Objects in a medical office might be patients and doctors. Each object is an example drawn from a class of similar objects. The savings account "class" in a bank has certain properties associated with it, such as name, Social Security number, account type, and account balance. Each individual savings account at the bank is an example of (an object of) the savings account class, and each has specific values for these common properties; that is, each savings account has a specific value for the name of the account holder, a specific value for the account balance, and so forth. Each object of a class therefore has its own data values.

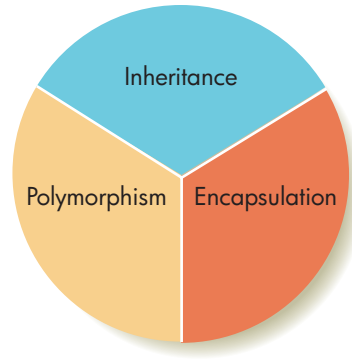
So far, this is similar to the idea of a data type in C++; in the SportsWorld program, *radius*, *circumference*, and *area* are all examples (objects) from the data type (class) "double"; the class has one property (a numeric quantity), and each object has its own specific value for that property. However, in object-oriented programming, a class also has one or more subtasks associated with it, and all objects from that class can perform those subtasks. In carrying out its subtask, each object can be thought of as providing some service. A savings account, for example, can compute compound interest due on the balance. When an object-oriented program is executed, the program generates requests for services that go to the various objects. The objects respond by performing the requested service—that is, carrying out the subtask. Thus, the main function in a C++ program, acting as a user of the savings account class, might request a particular savings account object to perform the service of computing interest due on its account balance. An object always knows its own data values and may use them in performing the requested service.

There are three terms often associated with object-oriented programming, as illustrated in Figure 27. The first term is **encapsulation**. Each class has its own program modules to perform each of its subtasks. Any user of the class (such as the main program) can ask an object of that class to invoke the appropriate module and thereby perform the subtask service. What the class user sees is the **interface** of the class, which describes the services provided and explains how to request an object to perform that service. The details of the module code are known only to the class. (In the savings account example, the details of the algorithm used to compute interest due belong only to the class.) The advantage to this separation of powers is that a given class's modules may be modified in any way, as long as the interface remains unchanged. (If the bank wants to change how it computes interest, only the code for the savings account class needs to be modified; any programs that use the services of the savings account class can remain unchanged.) A class therefore consists of two components, its properties and its subtask modules, and both components are "encapsulated"—bundled—with the class.

A second term associated with object-oriented programming is **inheritance**. Once a class A of objects is defined, a class B of objects can be defined as a "subclass" of A. Every object of class B is also an object of class A; this is sometimes called an "is a" relationship. Objects in the B class "inherit" all of the properties of objects in class A (including the ability to do what those objects can do), but they may also be given some special property or ability. The benefit is that class B does not have to be built from the ground up, but rather can take advantage of the fact that class A already exists. In the banking example, a senior citizen's savings account would be a subclass of the savings account class. Any senior citizens' savings account object is also a savings account object, but it may have special properties or be able to provide special services.

©2010 Course Technology, a part of Cengage Learning.

FIGURE 27
Three Key Elements of OOP



The third term is **polymorphism**. *Poly* means "many." Objects of different classes may provide services that should logically have the same name because they do roughly the same thing, but the details differ. In the banking example, both savings account objects and checking account objects should provide a "compute interest" service, but the details of how interest is computed differ in these two cases. Thus, one name, the name of the service to be performed, has several meanings, depending on the class of the object providing the service. It may even be the case that more than one service with the same name exists for the same class, although there must be some way to tell which service is meant when it is invoked by an object of that class.

Let's change analogies from the banking world to something more fanciful, and consider a football team. Every member of the team's backfield is an "object" of the "backfield" class. The quarterback is the only "object" of the "quarterback" class. Each backfield object can perform the service of carrying the ball if he (or she) receives the ball from the quarterback; ball carrying is a subtask of the backfield class. The quarterback who hands the ball off to a backfield object is requesting that the backfield object perform that subtask because it is "public knowledge" that the backfield class carries the ball and that this service is invoked by handing off the ball to a backfield object. The "program" to carry out this subtask is *encapsulated* within the backfield class, in the sense that it may have evolved over the week's practice and may depend on specific knowledge of the opposing team, but at any rate, its details need not be known to other players. *Inheritance* can be illustrated by the halfback subclass within the backfield class. A halfback object can do everything a backfield object can but may also be a pass receiver. And *polymorphism* can be illustrated by the fact that the backfield may invoke a different "program" depending on where on the field the ball is handed off. Of course our analogy is imperfect, because not all human "objects" from the same class behave in precisely the same way—fullbacks sometimes receive passes and so on.

6.2 *C++ and OOP*

How do these ideas get translated into real programs? The details, of course, vary with the programming language used, and not every language supports object-oriented programming. C++, however, does support object-oriented programming. When we write a class, we specify the properties (called **member variables**) common to any object of that class. We also specify the services (called **member functions**) that any object of that class can perform.

©2010 Course Technology, a part of Cengage Learning.

Let's rewrite the SportsWorld program one more time, this time as an object-oriented program. What are the objects of interest within the scope of this problem? SportsWorld deals with circular swimming pools, but they are basically just circles. So let's create a *Circle* class, and have the SportsWorld program create objects of (**instances of**) that class. The objects are individual circles. A Circle object has a radius. A Circle object, which knows the value of its own radius, should be able to perform the services of computing its own circumference and its own area. At this point, we have answered the two major questions about our *Circle* class:

- What are the properties common to any object of this class? (In this case, there is a single property—the radius.)
- What are the services that any object of the class should be able to perform? (In this case, it must compute its circumference and compute its area, although as we will see shortly, we will need other services as well.)

Figure 28 shows the complete object-oriented version of SportsWorld. After the usual opening stuff, the program is divided into three major sections (identified by comments): the class interface, the main function, and the class implementation. (The *getInput* function does not fall within any of these sections.)

FIGURE 28

An Object-Oriented SportsWorld Program

```
//This program helps SportsWorld estimate costs
//for pool covers and pool fencing by computing
//the area or circumference of a circle
//with a given radius.
//Any number of circles can be processed.
//Uses class Circle
//Written by M. Phelps, 10/23/10

#include <iostream>
using namespace std;

const double PI = 3.1416; //value of pi

//class interface
class Circle
{
public:
    void setRadius(double value);
    //sets radius equal to value

    double getRadius();
    //returns current radius

    double doCircumference();
    //computes and returns circumference of a circle

    double doArea();
    //computes and returns area of a circle

private:
    double radius;
};  //end of class interface
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 28

An Object-Oriented SportsWorld Program (continued)

```
void getInput(double &newRadius, char &taskToDo)
//gets radius and choice of task from the user
{

    cout << "Enter the value of the radius "
         << "of a pool: ";
    cin >> newRadius;

    //See what user wants to compute
    cout << "Enter your choice of task." << endl;
    cout << "C to compute circumference, A to compute area: ";
    cin >> taskToDo;
}

//main function
void main()
{
    double newRadius;     //radius of a pool - given
    char taskToDo;        //holds user choice to
                          //compute circumference or area
    char more;            //controls loop for
                          //processing more pools
    Circle swimmingPool;  //create a Circle object
    cout.setf(ios::fixed);
    cout.precision(2);

    cout << "Do you want to process a pool? (Y or N): ";
    cin >> more;
    cout << endl;
    while (more == 'Y') //more circles to process
    {
      getInput(newRadius, taskToDo);
      swimmingPool.setRadius(newRadius);
      if (taskToDo == 'C') //compute circumference
         cout << "The circumference for a pool "
              << "of radius " << swimmingPool.getRadius()
              << " is " << swimmingPool.doCircumference() << endl;
      else       //compute area
         cout << "The area for a pool "
              << "of radius is " << swimmingPool.getRadius()
              << " is " << swimmingPool.doArea() << endl;

      cout << endl;
      cout << "Do you want to process more pools? (Y or N): ";
      cin >> more;
      cout << endl;
    }

    //finish up
    cout << "Program will now terminate." << endl;
}

//class implementation
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 28

An Object-Oriented SportsWorld Program (continued)

```
void Circle::setRadius(double value)
//sets radius equal to value
{
    radius = value;
}

double Circle::getRadius()
//returns current radius
{
    return radius;
}

double Circle::doCircumference()
//computes and returns circumference of a circle
{
    return 2*PI*radius;
}

double Circle::doArea()
//computes and returns area of a circle
{
    return PI*radius*radius;
}
```

In the class interface, four member functions are declared, although their code is not given. The function declaration gives the compiler enough information to check for correct usage of the function—for example, whether the statement that invokes the function passes to it the correct number of arguments. The first member function is a void function, and the remaining three return values. All of these must be invoked by an object of the *Circle* class. One member variable is given. The member functions of the *Circle* class are all declared using the keyword **public**. Public functions can be used anywhere, including within the main function and indeed in any C++ program that wants to make use of this class. Think of the *Circle* class as handing out a business card that advertises these services: Hey, you want a Circle object that can find its own area? Find its own circumference? Set the value of its own radius? I'm your class! (Class member functions can also be **private**, but a private member function is a sort of helping task that can be used only within the class in which it occurs.)

The single member variable of the class (*radius*) is declared using the keyword private. Only functions in the *Circle* class itself can use this variable. Note that *doCircumference* and *doArea* have no parameter for the value of the radius; as functions of this class, they know at all times the current value of *radius* for the object that invoked them, and it does not have to be passed to them as an argument. Because *radius* has been declared private, however, the main function cannot use the value of *radius*. It cannot write out that value or directly change that value by some assignment statement. It can, however, request a Circle object to invoke the *getRadius* member function to return the current value of the radius in order to write it out. It can also request a Circle object to invoke the *setRadius* member function to change the value of its radius; *setRadius* does have a parameter to receive a new value for *radius*. Member variables are generally declared private instead of public, to protect the data in an object from reckless changes some application program might

©2010 Course Technology, a part of Cengage Learning.

try to make. Changes in the values of member variables should be performed only under the control of class objects through functions such as *setRadius*.

The main function, as before, handles all of the user interaction and makes use of the *Circle* class. It creates a Circle object, an instance of the *Circle* class, by the following statement:

```
Circle swimmingPool;
```

This looks just like an ordinary variable declaration such as

```
int number;
```

After

```
Circle swimmingPool;
```

the object *swimmingPool* exists, and the main function can ask *swimmingPool* to perform the various services of which instances of the *Circle* class are capable.

The syntax to request an object to invoke a member function is to give the name of the object, followed by a dot, followed by the name of the member function, followed by any arguments the function may need.

object-identifier.function-identifier(argument list)

The object that invokes a function is the **calling object**. Therefore the expression

```
swimmingPool.doCircumference()
```

in the main function uses *swimmingPool* as the calling object to invoke the *doCircumference* function of the *Circle* class. No arguments are needed because this function has no parameters, but the empty parentheses must be present.

The class implementation section contains the actual code for the various functions "advertised" in the class interface. Each function begins with a modified form of the usual C++ function header. The modification consists of putting the class name and two colons in front of the function name so that the function code is associated with the proper class. Looking at the code for these member functions in Figure 28, we see that the *setRadius* member function uses an assignment statement to change the value of *radius* to whatever quantity is passed to the parameter *value*. The *getRadius* function body is a single return statement. The *doCircumference* and *doArea* functions again consist of single statements that compute and return the proper value.

There is no declaration in the main function for a variable called *radius*. There is a declaration for *newRadius*, and *newRadius* receives the value entered by the user for the radius of the circle. Therefore, isn't *newRadius* serving the same purpose as *radius* did in the old program? No—this is rather subtle, so pay attention: While *newRadius* holds the number the user wants for the circle radius, it is not itself the radius of *swimmingPool*. The radius of *swimmingPool* is the member variable *radius*, and only functions of the class can change the member variables of an object of that class. The *Circle* class provides the *setRadius* member function for this purpose. The main function must ask the object *swimmingPool* to invoke *setRadius* to set the value of its

©2010 Course Technology, a part of Cengage Learning.

radius equal to the value contained in *newRadius*. The *newRadius* argument corresponds to the *value* parameter in the *setRadius* function, which then gets assigned to the member variable *radius*.

```
swimmingPool.setRadius(newRadius);


void setRadius(double value)
//sets radius equal to value
{
  radius = value;
}
```

The *setRadius* function is a void function because it returns no information to the invoking statement; it contains no return statement. The invocation of this function is a complete C++ statement.

Finally, the output statements in the main function that print the values of the circumference and area also have *swimmingPool* invoke the *getRadius* member function to return its current *radius* value so it can be printed as part of the output. We could have used the variable *newRadius* here instead. However, *newRadius* is what we THINK has been used in the computation, whereas *radius* is what has REALLY been used.

6.3 One More Example

The object-oriented version of our SportsWorld program illustrates encapsulation. All data and calculations concerning circles are encapsulated in the *Circle* class. Let's look at one final example that illustrates the other two watchwords of OOP—polymorphism and inheritance.

In Figure 29 the domain of interest is that of geometric shapes. In the class interfaces section, four different classes are described: *Circle*, *Rectangle*, *Square*, and *Square2*. Each class description consists of a public part and a private or protected part. The public part describes, in the form of C++ function headers, the services or subtasks that an object from the class can perform. The private or protected part describes the properties that any object of the class possesses. A Circle object has a radius property, whereas a Rectangle object has a width property and a height property. Any Circle object can set the value of its radius and can compute its area. A Square object has a side property, as one might expect, but a Square2 object doesn't seem to have any properties or, for that matter, any way to compute its area. We will explain the difference between the *Square* class and the *Square2* class shortly.

The main function uses these classes. It creates objects from the various classes. After each object is created, the main function requests the object to set its dimensions, using the values given, and to compute its area as part of an output statement giving information about the object. For example, the statement

```
joe.setRadius(23.5);
```

instructs the circle named *joe* to invoke the *setRadius* function of *joe*'s class, thereby setting *joe*'s radius to 23.5. Figure 30 shows the output after the program in Figure 29 is run.

©2010 Course Technology, a part of Cengage Learning.

FIGURE 29

A C++ Program with Polymorphism and Inheritance

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

const double PI = 3.1416;

//class interfaces
class Circle
{
public:
   void setRadius(double value);
   //sets radius of the circle equal to value

   double getRadius();
   //returns the radius value

   double doArea();
   //computes and returns area of circle

private:
   double radius;
};

class Rectangle
{
public:
   void setWidth(double value);
   //sets width of rectangle equal to value

   void setHeight(double value);
   //sets height of rectangle equal to value

   double getWidth();
   //returns width

   double getHeight();
   //returns height

   double doArea();
   //computes and returns area of rectangle

protected:
   double width;
   double height;
};

class Square
{
public:
   void setSide(double value);
   //sets the side of the square equal to value

   double getSide();
   //returns side

   double doArea();
   //computes and returns area of the square
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 29

A C++ Program with Polymorphism and Inheritance (continued)

```
private: double side;
};

class Square2: public Rectangle
//Square is derived class of Rectangle,
//uses the inherited height and width
//properties and the inherited doArea function
{
public:
   void setSide(double value);
   //sets the side of the square equal to value

   double getSide();
   //returns the value of side (width)
};

//main program
void main()
{
   Circle joe;
   joe.setRadius(23.5);
   cout << "The area of a circle with "
        << "radius " << joe.getRadius()
        << " is " << joe.doArea() << endl;

   Rectangle luis;
   luis.setWidth(12.4);
   luis.setHeight(18.1);
   cout << "The area of a rectangle with "
        << "dimensions " << luis.getWidth()
        << " and " << luis.getHeight()
        << " is " << luis.doArea() << endl;

   Square anastasia;
   anastasia.setSide(3);
   cout << "The area of a square with "
        << "side " << anastasia.getSide()
        << " is " << anastasia.doArea() << endl;

   Square2 tyler;
   tyler.setSide(4.2);
   cout << "The area of a square with "
        << "side " << tyler.getSide()
        << " is " << tyler.doArea() << endl;
}

//class implementations
void Circle::setRadius(double value)
{
   radius = value;
}
double Circle::getRadius()
{
   return radius;
}
double Circle::doArea()
{
   return PI*radius*radius;
}
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 29

A C++ Program with Polymorphism and Inheritance (continued)

```
void Rectangle::setWidth(double value)
{
   width = value;
}

void Rectangle::setHeight(double value)
{
   height = value;
}

double Rectangle::getWidth()
{
   return width;
}

double Rectangle::getHeight()
{
   return height;
}
double Rectangle::doArea()
{
   return width*height;
}

void Square::setSide(double value)
{
   side = value;
}

double Square::getSide()
{
   return side;
}

double Square::doArea()
{
   return side*side;
}

void Square2::setSide(double value)
{
   height = value;
   width = value;
}

double Square2::getSide()
{
   return width;
}
```


©2010 Course Technology, a part of Cengage Learning.

FIGURE 30

Output from the Program of Figure 29

```
The area of a circle with radius 23.5 is 1734.95
The area of a rectangle with dimensions 12.4 and 18.1 is 224.44
The area of a square with side 3 is 9
The area of a square with side 4.2 is 17.64
```

Here we see polymorphism at work, because there are lots of *doArea* functions; when the program executes, the correct function is used on the basis of the class to which the object invoking the function belongs. After all, computing the area of a circle is quite different from computing the area of a rectangle. The algorithms themselves are straightforward; they employ assignment statements to set the dimensions and the usual formulas to compute the area of a circle, rectangle, and square. The functions can use the properties of the objects that invoke them without having the values of those properties passed as arguments.

Square is a stand-alone class with a *side* property and a *doArea* function. The *Square2* class, however, recognizes the fact that squares are special kinds of rectangles. The *Square2* class is a subclass of the *Rectangle* class, as is indicated by the reference to *Rectangle* in the class interface of *Square2*. It inherits the *width* and *height* properties from the "parent" *Rectangle* class; the "protected" status of these properties in the *Rectangle* class indicates that they can be extended to any subclass. *Square2* also inherits the *setWidth*, *setHeight*, and *doArea* functions. In addition, *Square2* has its own function, *setSide*, because setting the value of the "side" makes sense for a square but not for an arbitrary rectangle. What the user of the *Square2* class doesn't know is that there really isn't a "side" property; the *setSide* function merely sets the inherited *width* and *height* properties to the same value. To compute the area, then, the *doArea* function inherited from the *Rectangle* class can be used, and there's no need to redefine it or even to copy the existing code. Here we see inheritance at work.

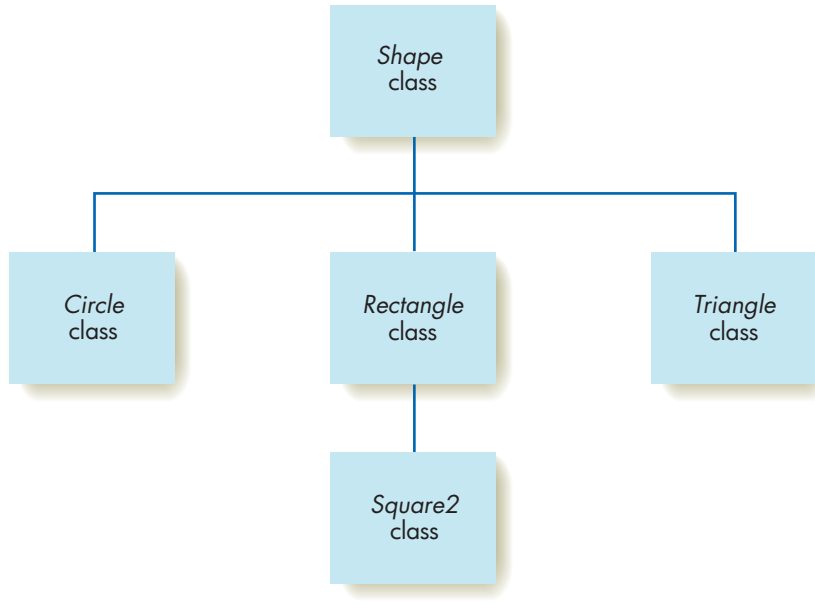
Inheritance can be carried through multiple "generations." We might redesign the program so that there is one "superclass" that is a general *Shape* class, of which *Circle* and *Rectangle* are subclasses, *Square2* being a subclass of *Rectangle* (see Figure 31 for a possible class hierarchy).

Although the program of Figure 29 can be kept in one file, it can also be split into separate files, roughly in the three sections described. The class interfaces can be kept in a **header file**; a programmer wishing to use these classes in an application program can look at this file, discover what properties and services are available, and learn how to use them. The main function—the application that the programmer is creating—is a separate program in a separate file. And the implementation of the classes is kept in a third file to be used as needed by the main function. In fact, the implementation of the classes may be compiled into object code, stored in a library, and linked with the main function when the program executes. The programmer doesn't see the implementations; the object code gets included by the linker because the application program contains the proper include directives. Here we see encapsulation: the wrapping of implementation with the class and not with the class user. The class can change the implementation code, and as long as the class interface remains the same, the application code need not change. If the objects of the class perform the advertised services, the user of the class need not see the details.

©2010 Course Technology, a part of Cengage Learning.

FIGURE 31

A Hierarchy of Geometric Classes



6.4 What Have We Gained?

Now that we have some idea of the flavor of object-oriented programming, we should ask what we gain by this approach. There are two major reasons why OOP is a popular way to program:

- Software reuse
- A more natural "worldview"

SOFTWARE REUSE. Manufacturing productivity took a great leap forward when Henry Ford invented the assembly line. Automobiles could be assembled using identical parts so that each car did not have to be treated as a unique creation. Computer scientists are striving to make software development more of an assembly-line operation and less of a handcrafted, start-over-each-time process. Object-oriented programming is a step toward this goal: A useful class that has been implemented and tested becomes a component available for use in future software development. Anyone who wants to write an application program involving circles, for example, can use the already written, tried, and tested *Circle* class. As the "parts list" (the class library) grows, it becomes easier and easier to find a "part" that fits, and less and less time has to be devoted to writing original code. If the class doesn't quite fit, perhaps it can be modified to fit by creating a subclass; this is still less work than starting from scratch. Software reuse implies more than just faster code generation. It also means improvements in *reliability*; these classes have already been tested, and if properly used, they will work correctly. And it means improvements in *maintainability*. Thanks to the encapsulation property of object-oriented programming, changes can be made in class implementations without affecting other code, although such change requires retesting the classes.

©2010 Course Technology, a part of Cengage Learning.

A MORE NATURAL "WORLDVIEW." The traditional view of programming is procedure-oriented, with a focus on tasks, subtasks, and algorithms. But wait—didn't we talk about subtasks in OOP? Haven't we said that computer science is all about algorithms? Does OOP abandon these ideas? Not at all. It is more a question of *when* these ideas come into play. Object-oriented programming recognizes that in the "real world," tasks are done by entities (objects). Object-oriented program design begins by identifying those objects that are important in the domain of the program because their actions contribute to the mix of activities present in the banking enterprise, the medical office, or wherever. Then it is determined what data should be associated with each object and what subtasks the object contributes to this mix. Finally, an algorithm to carry out each subtask must be designed. We saw in the modularized version of the SportsWorld program in Figure 24 how the overall algorithm could be broken down into pieces that are isolated within functions. Object-oriented programming repackages those functions by encapsulating them within the appropriate class of objects.

Object-oriented programming is an approach that allows the programmer to come closer to modeling or simulating the world as we see it, rather than to mimic the sequential actions of the Von Neumann machine. It provides another buffer between the real world and the machine, another level of abstraction in which the programmer can create a virtual problem solution that is ultimately translated into electronic signals on hardware circuitry.

Finally, we should mention that a graphical user interface, with its windows, icons, buttons, and menus, is an example of object-oriented programming at work. A general button class, for example, can have properties of height, width, location on the screen, text that may appear on the button, and so forth. Each individual button object has specific values for those properties. The button class can perform certain services by responding to messages, which are generated by events (for example, the user clicking the mouse on a button triggers a "mouse-click" event). Each particular button object individualizes the code to respond to these messages in unique ways. We will not go into details of how to develop graphical user interfaces in C++, but in the next section you will see a bit of the programming mechanics that can be used to draw the graphics items that make up a visual interface.

PRACTICE PROBLEMS

1. What is the output from the following section of code if it is added to the main function of the C++ program in Figure 29?

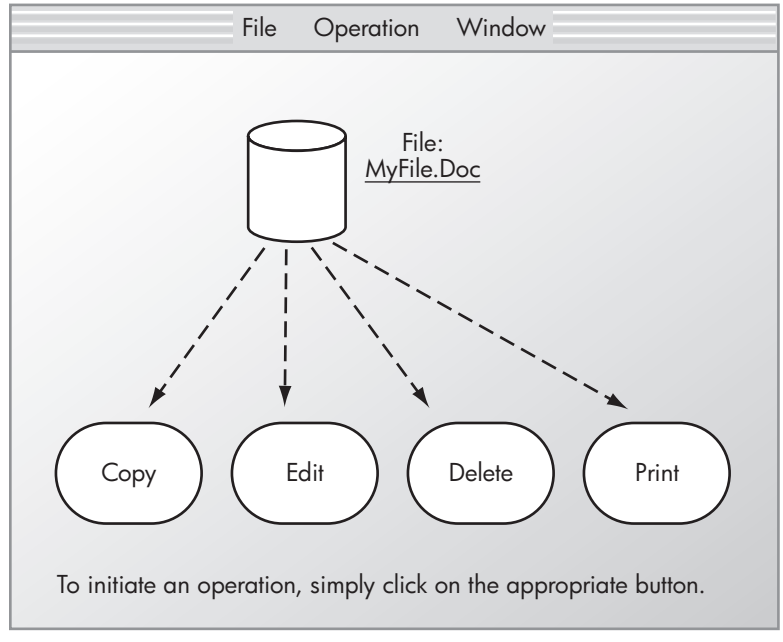
```
Square one;
one.setSide(10);
cout << "The area of a square with "
     << "side " << one.getSide()
     << " is " << one.doArea() << endl;
```

2. In the hierarchy of Figure 31, suppose that the *Triangle* class is able to perform a *doArea* function. What two properties should any triangle object have?

©2010 Course Technology, a part of Cengage Learning.

The programs that we have looked at so far all produce *text output*—output composed of the characters {A . . . Z, a . . . z, 0 . . . 9} along with a few punctuation marks. For the first 30 to 35 years of software development, text was virtually the only method of displaying results in human-readable form, and in those early days it was quite common for programs to produce huge stacks of alphanumeric output. These days an alternative form of output—*graphics*—has become much more widely used. With graphics, we are no longer limited to 100 or so printable characters; instead, programmers are free to construct whatever shapes and images they desire.

The intelligent and well-planned use of graphical output can produce some phenomenal improvements in software. We discussed this issue in Chapter 6, where we described the move away from the text-oriented operating systems of the 1970s and 1980s, such as MS-DOS and VMS, to operating systems with more powerful and user-friendly graphical user interfaces (GUIs), such as Windows Vista and Mac OS X. Instead of requiring users to learn dozens of complex text-oriented commands for such things as copying, editing, deleting, moving, and printing files, GUIs can present users with simple and easy-to-understand visual metaphors for these operations, as shown below.



Not only does graphics make it easier to manage the tasks of the operating system, it can help us visualize and make sense of massive amounts of output produced by programs that model complex physical, social, and mathematical systems. (We discuss modeling and visualization in Chapter 13.) Finally, there are many applications of computers that would simply be impossible without the ability to display output visually. Applications such as virtual reality, computer-aided design/computer-aided manufacturing (CAD/CAM), games and entertainment, medical imaging, and computer mapping would not be anywhere near as important as they are without the enormous improvements that have occurred in the areas of graphics and visualization.

©2010 Course Technology, a part of Cengage Learning.

So, we know that graphical programming is important. The question is: What features must be added to a programming language like C++ to produce graphical output?

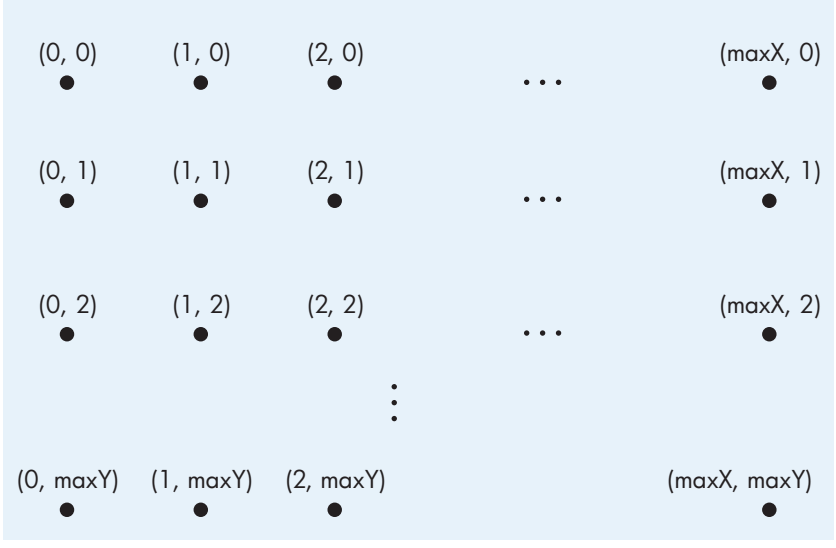
7.1 *Graphics Primitives*

Modern computer terminals use what is called a **bitmapped display**, in which the screen is made up of thousands of individual picture elements, or **pixels**, laid out in a two-dimensional grid. These are the same pixels used in visual images, as discussed in Chapter 4. In fact, the display is simply one large visual image. The number of pixels on the screen varies from system to system; typical values range from 800 × 600 up to 1560 × 1280. Terminals with a high density of pixels are called **high-resolution** terminals. The higher the resolution—that is, the more pixels available in a given amount of space—the sharper the visual image because each individual pixel is smaller. However, if the screen size itself is small, then a high-resolution image can be too tiny to read. A 30" wide-screen monitor might support a resolution of 2560 × 1600, but that would not be suitable for a laptop screen. In Chapter 4 you learned that a color display requires 24 bits per pixel, with 8 bits used to represent the value of each of the three colors red, green, and blue. The memory that stores the actual screen image is called a **frame buffer**. A high-resolution color display might need a frame buffer with (1560 × 1280) pixels × 24 bits/pixel = 47,923,000 bits, or about 6 MB, of memory for a single image. (One of the problems with graphics is that it requires many times the amount of memory needed for storing text.)

The individual pixels in the display are addressed using a two-dimensional coordinate grid system, the pixel in the upper-left corner being (0, 0). The overall pixel-numbering system is summarized in Figure 32. The specific values for *maxX* and *maxY* in Figure 32 are, as mentioned earlier, system-dependent. (Note that this coordinate system is not the usual mathematical one. Here, the origin is in the upper-left corner, and *y* values are measured downward.)

The terminal hardware displays on the screen the frame buffer value of every individual pixel. For example, if the frame buffer value on a color monitor for position (24, 47) is RGB (0, 0, 0), the hardware sets the color of the

FIGURE 32
Pixel-Numbering System in a Bitmapped Display



©2010 Course Technology, a part of Cengage Learning.

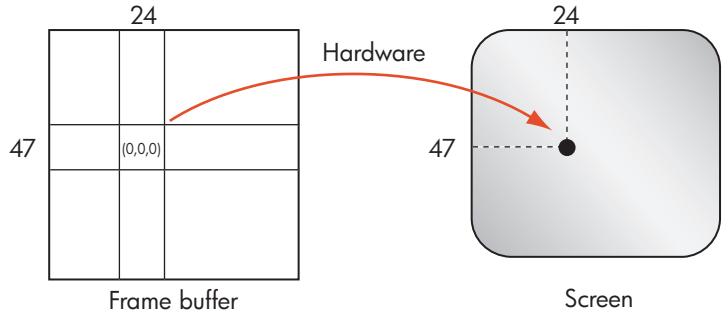


FIGURE 33
Display of Information on the Terminal

pixel located at column 24, row 47 to black, as shown in Figure 33. The operation diagrammed in Figure 33 must be repeated for all of the 500,000 to 2 million pixels on the screen. However, the setting of a pixel is not permanent; on the contrary, its color and intensity fade quickly. Therefore, each pixel must be "repainted" often enough so that our eyes do not detect any "flicker," or change in intensity. This requires the screen to be completely updated, or refreshed, 30–50 times per second. By setting various sequences of pixels to different colors, the user can have the screen display any desired shape or image. This is the fundamental way in which graphical output is achieved.

To control the setting and clearing of pixels, programmers use a collection of functions that are part of a special software package called a **graphics library**. Virtually all modern programming languages, including most implementations of C++, come with an extensive and powerful graphics library for creating different shapes and images. Typically, an "industrial strength" graphics library includes hundreds of functions for everything from drawing simple geometric shapes like lines and circles, to creating and selecting colors, to more complex operations such as displaying scrolling windows, pull-down menus, and buttons. We restrict our discussion to the more modest set of functions available in the laboratory software package created for *Invitation to Computer Science*. Although the set is unrealistically small, the 10 graphics functions in our graphics library give you a good idea of what visual programming is like, and enable you to produce some interesting, nontrivial images on the screen. These functions are described in the following paragraphs.

1. *clearscreen(I)*. If the integer parameter *I* <= 0, then the output window (the window in which results are displayed) is cleared to a white background. If *I* >= 1, then the output window is cleared to a black background. It is recommended that you initially clear the window before doing any other graphical operations.
2. *moveto(x, y)*. The execution of this instruction causes the cursor to move from where it is currently located to output window position (*x, y*) *without* drawing anything on the screen. The value of *x* must be between 0 and *maxX*, and the value of *y* must be between 0 and *maxY*. The pixels are numbered beginning with (0, 0) in the upper-left corner, as shown in Figure 32. Thus, for example, if *maxX* = 600 and *maxY* = 800, then the operation *moveto(300, 400)* will move the cursor to the middle of the output window.
3. *getmaxx( )*
4. *getmaxy( )*

©2010 Course Technology, a part of Cengage Learning.

These two functions return the integer value of *maxX* and *maxY*, respectively, for the current output window. To move the cursor to the middle of the window, regardless of its size, we can write:

```
X = getmaxx();      //this is the number of
                    //pixels horizontally
Y = getmaxy();      //this is the number of
                    //pixels vertically
moveto(X/2, Y/2);  //now move the cursor to the
                   //midpoint of the screen
```

If the output window is resized during program execution, a second call to these two functions returns the new values of *maxX* and *maxY* for the output window.

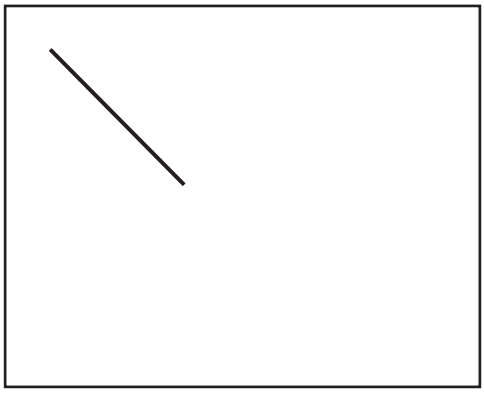
5. *setcolor(I)*. This command sets the color of the pen that draws lines and shapes on the screen. If you have a color display, then a function like this typically allows you to set the pen to any one of hundreds or even thousands of different colors. However, our simple graphics package only uses black and white. If $I = 0$, then the pen color is set to white; if $I = 1$, then the pen color is set to black.

A white pen does not show up on a white background, nor does a black pen on a black background. This feature can be used to do erasure. Simply reset the pen color to the color of the background and redraw your image to make it disappear.

6. *lineto(x, y)*. This operation draws a straight line from the current position of the cursor to output window position (*x, y*) using the current pen color. For example, the sequence of commands

```
clearscreen(0);   //clear the screen to white
setcolor(1);      //set the pen color to black
moveto(20, 20);   //move the cursor to screen
                  //position (20, 20)
lineto(100, 100); //now draw a line from (20, 20)
                  //to (100, 100)
```

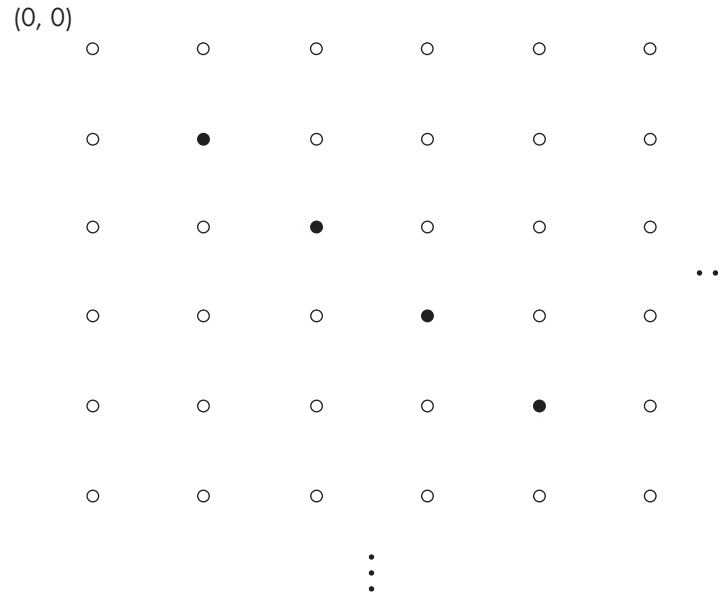
will draw a black line from cursor position (20, 20) to position (100, 100), producing something like this:



(On your system, the exact location and length of the line may be slightly different because of differences in screen resolution.)

©2010 Course Technology, a part of Cengage Learning.

What actually happens internally when you execute a *lineto* command? The answer is that the terminal hardware determines (using some simple geometry and trigonometry) exactly which pixels on the screen must be "turned on" (set to the current value of the pen color) to draw a straight line between the specified coordinates. For example, if the pen color is black (i.e., 1), then the two commands *moveto(1, 1)* and *lineto(4, 4)* set the following four pixels in the frame buffer to the RGB value (0, 0, 0).

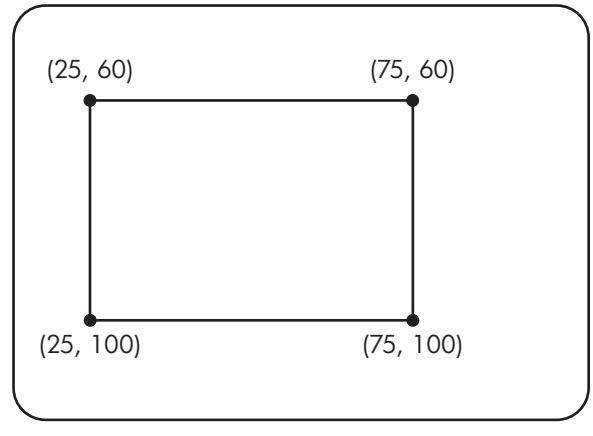


Now, when the hardware draws the frame buffer on the screen, these four pixels are colored black. Because the pixels are approximately $1/100^{th}$ of an inch apart, our eyes perceive not four individual black dots but an unbroken line segment.

7. *rectangle(x1, y1, x2, y2)*. This function draws a rectangle whose upper-left and lower-right corners are located at coordinates (*x*1, *y*1) and (*x*2, *y*2). The instruction

```
rectangle(25, 60, 75, 100);
```

produces the following image:



©2010 Course Technology, a part of Cengage Learning.

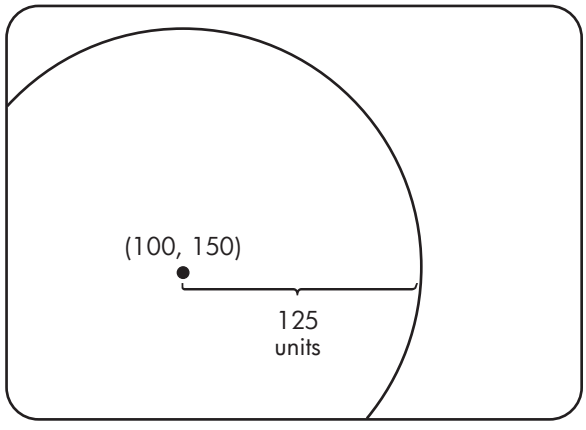
The same diagram is produced by the instruction

```
rectangle(75, 100, 25, 60);
```

8. *circle(x, y, r)*. This function draws a circle whose center is located at position (*x, y*) and whose radius is *r*, the units being pixels. The instruction

   ```
   circle(100, 150, 125);
   ```

   draws the figure



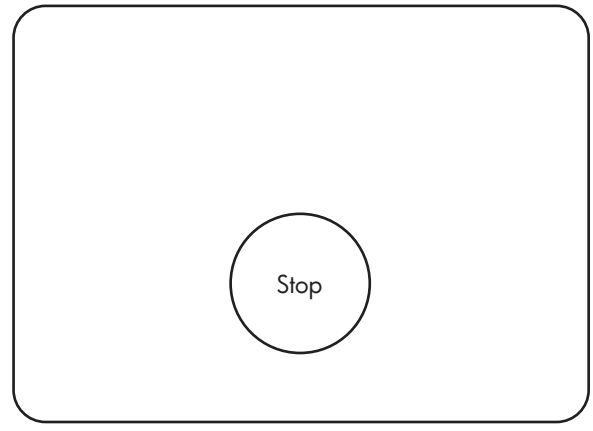
   Note that portions of the circle beyond the edge of the window are discarded, a graphics operation called **clipping**. All functions in our graphics library clip those parts of the image that lie outside the window boundaries.

9. *writedraw(value, x, y)*. This operation causes the indicated value to be displayed at the specific (*x, y*) window coordinates. The value can be either an integer (such as 234), an individual character ('A'), or a string ("Press here"). The (*x, y*) coordinates represent the position of the bottom-left pixel of the leftmost part of the value. For example, if we want to display the string "Hello" in the output window, we should enter as the (*x, y*) coordinate the exact position of the lower-left pixel in the letter 'H'.

   The inclusion of the "writedraw" operation allows us to mix both graphic and text output on a single screen. For example, to produce a circular button containing the message "stop", we can do the following:

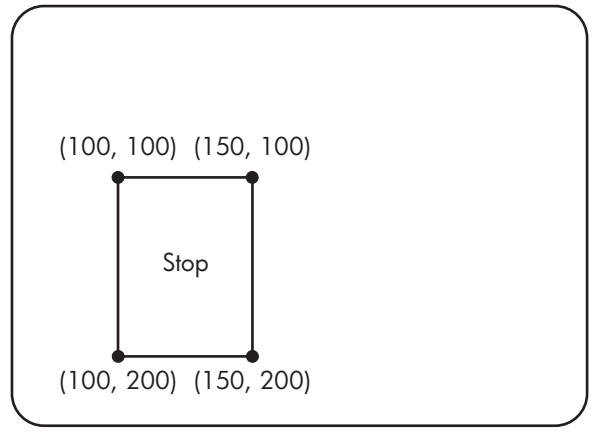
```
circle(200, 200, 40);          //first draw a
                               //circle
writedraw("Stop", 186, 200);   //then put the
                               //label "Stop"
                               //inside the
                               //circle
```


©2010 Course Technology, a part of Cengage Learning.



This looks somewhat like the labeled buttons used in modern graphical interfaces. This example demonstrates how you can combine primitive operations such as "circle" and "writedraw" to generate more complex images such as a labeled button.

10. *getmouse(x, y)*. This function is used not for output but for input. The *getmouse* routine stores the (*x, y*) coordinates of the cursor at the instant the mouse button is clicked. This allows the user to input information via the mouse. For example, suppose we have the following rectangular stop button on the screen:



and we ask users to click inside the button if they wish to quit the program. We can use the *getmouse* function to obtain the cursor coordinates and determine whether they lie inside the button boundaries. Using the numbers shown above, we need to determine whether the cursor coordinates lie within a rectangle located at (100, 100), (100, 200), (150, 200), (150, 100) as follows:

```
getmouse(X, Y);
//see if the user clicked inside the
//rectangular bounds of the button
if ((X >= 100) && (X <= 150)
  && (Y >= 100) && (Y <= 200))
{. . . //the user clicked inside the
       //button and wants to stop}
```

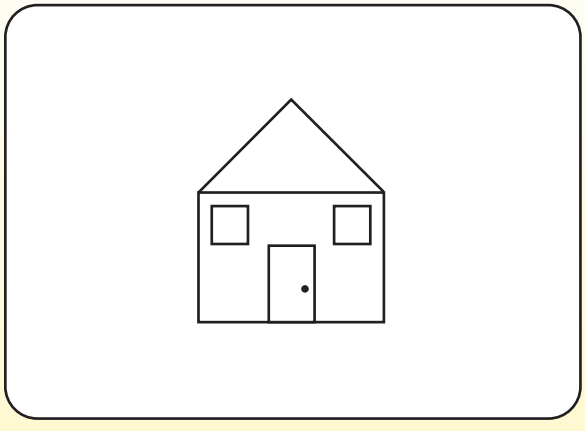

©2010 Course Technology, a part of Cengage Learning.

```
else
{. . . //the user clicked outside the button}
```

These are the 10 graphics functions included in the C++ compiler component of our laboratory software. As we have said, the number of functions found in a production software development package is much larger. However, the functions described here enable you to do some interesting graphics and, even more important, give you an appreciation for how visually-oriented software is developed.

PRACTICE PROBLEM

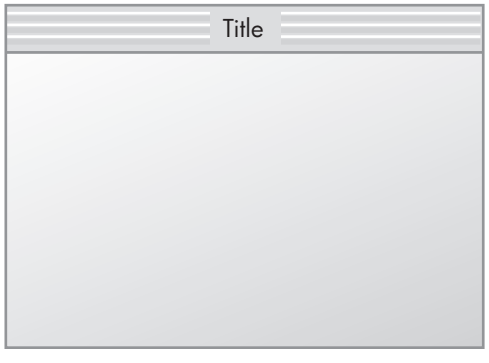
Write the sequence of commands to draw the following "house" on the screen.



Create the house using four rectangles (for the base of the house, the door, and the two windows), two line segments (for the roof), and one circle (for the doorknob). Locate the house anywhere you want on the screen.

7.2 *An Example of Graphics Programming*

We finish this section with a somewhat larger and more "computer-oriented" example of the use of graphics. We use the routines in our graphics library to create a *titled window*—a rectangular window with a second rectangle on top that contains a text label. Here is an example of a titled window:



©2010 Course Technology, a part of Cengage Learning.

Titled windows are quite common; they are part of just about every graphical interface. We show how one can be drawn using our basic graphics functions.

Although you may see the diagram above as one large rectangle, in reality there are two separate rectangles positioned with one directly on top of the other.

Therefore, we first draw two separate rectangles, making sure that their coordinates are set so that they are properly positioned with one on top of the other. For this example, we make the size of the large bottom rectangle 300 × 250 pixels and the size of the smaller top rectangle 300 × 30 pixels. Before doing this drawing, we clear the screen.

```
clearscreen(0);              //clear the screen to
                             //a white background
setcolor(1);                 //set the color to
                             //black
rectangle(50, 50, 350, 80);  //this produces the
                             //top rectangle
rectangle(50, 80, 350, 330); //and this produces
                             //the larger bottom one
```

After these four commands are executed, the screen contains

©2010 Course Technology, a part of Cengage Learning.

We now must put the four horizontal lines inside the top rectangle. We draw these lines so that they divide the upper rectangle into thirds both horizontally and vertically.

Now that we have decided on the layout, here are the commands to draw these four lines in the desired position:

```
moveto(50, 60); lineto(150, 60);
moveto(50, 70); lineto(150, 70);
moveto(250, 60); lineto(350, 60);
moveto(250, 70); lineto(350, 70);
```

Finally, we need to position the title in the middle of the white space created by the four lines that we just drew. This can be done with our *writedraw* function:

```
writedraw("Title", 180, 70);  //write "Title" in the
                              //middle of the top
                              //rectangle
```

The complete sequence of commands needed to create our rectangular titled window is summarized in Figure 34. (Because of differences in screen size and resolution, you may have to slightly modify the parameters for these commands to get your display positioned just right.) In the exercises at the end of this module, we ask you to add some other features to this window. These modifications can all be done using the basic drawing functions described in this section.

FIGURE 34

Commands to Produce a Titled Window

```
clearscreen(0);
setcolor(1);
rectangle(50, 50, 350, 80);
rectangle(50, 80, 350, 300);
moveto(50, 60);
lineto(150, 60);
moveto(50, 70);
lineto(150, 70);
moveto(250, 60);
lineto(350, 60);
moveto(250, 70);
lineto(350, 70);
writedraw("Title", 180, 70);
```


©2010 Course Technology, a part of Cengage Learning.

8 Conclusion

In this module we looked at one representative high-level programming language, C++. Of course, there is much about this language that has been left unsaid, but we have seen how the use of a high-level language overcomes many of the disadvantages of assembly language programming, creating a more comfortable and useful environment for the programmer. In a high-level language, the programmer need not manage the storage or movement of data values in memory. The programmer can think about the problem at a higher level, can use program instructions that are both more powerful and more natural language–like, and can write a program that is much more portable among various hardware platforms. We also saw how modularization, through the use of functions and parameters, allows the program to be more cleanly structured and how object-oriented programming allows a more intuitive view of the problem solution and provides the possibility for reuse of helpful classes. We even had a glimpse of graphical programming.

C++ is not the only high-level language. You might be interested in looking at the other online language modules for languages similar to C++ (Java, Python, C#, Ada). Still other languages take quite a different approach to problem solving. In Chapter 10 of *Invitation to Computer Science*, we look at some other languages and language approaches and also address the question of why there are so many different programming languages.

©2010 Course Technology, a part of Cengage Learning.

EXERCISES

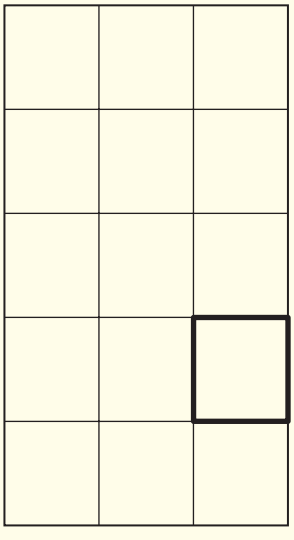
1. Write a C++ declaration for one real number quantity to be called *rate*.
2. Write a C++ declaration for two integer quantities called *orderOne* and *orderTwo*.
3. Write a C++ declaration for a constant quantity called *EVAPORATION_RATE*, which is to have the value 6.15.
4. A C++ main function needs one constant *STOCK_TIME* with a value of 4, one integer variable *inventory*, and one real number variable *sales*. Write the necessary declarations.
5. You want to write a C++ program to compute the average of three quiz grades for a single student. Decide what variables your program needs, and write the necessary declarations.
6. Given the declaration

   ```
   int list[10];
   ```

   how do you refer to the eighth number in the array?
7. An array declaration such as

   ```
   int table[5][3];
   ```

   represents a two-dimensional table of values with 5 rows and 3 columns. Rows and columns are numbered in C++ starting at 0, not at 1. Given this declaration, how do you refer to the marked cell below?



8. Write C++ statements to prompt for and collect values for the time in hours and minutes (two integer quantities).
9. An output statement may contain more than one variable identifier. Say a program computes two integer quantities *inventoryNumber* and *numberOrdered*. Write a single output statement that prints these two quantities along with appropriate text information.
10. The integer quantities *A*, *B*, *C*, and *D* currently have the values 13, 4, 621, and 18, respectively. Write the exact output generated by the following statement, using b to denote a blank space.

    ```
    cout << setw(5) << A << setw(3)
    << B << setw(3) << C << setw(4)
    << D << endl;
    ```

11. Write C++ formatting and output statements to generate the following output, assuming that *density* is a type *double* variable with the value 63.78.

    ```
    The current density is 63.8,
    to within one decimal place.
    ```

12. What is the output after the following sequence of statements is executed? (Assume that the integer variables *A* and *B* have been declared.)

    ```
    A = 12;
    B = 20;
    B = B + 1;
    A = A + B;
    cout << 2*A << endl;
    ```

13. Write the body of a C++ main function that gets the length and width of a rectangle from the user and computes and writes out the area. Assume that the variables have all been declared.
14. a. In the SportsWorld program of Figure 15, the user must respond with "C" to choose the circumference task. In such a situation, it is preferable to accept either uppercase or lowercase letters. Rewrite the condition in the program to allow this.
    b. In the SportsWorld program, rewrite the condition for continuation of the program to allow either an uppercase or a lowercase response.
15. Write a C++ main function that gets a single character from the user and writes out a congratulatory message if the character is a vowel (a, e, i, o, or u), but otherwise writes out a "You lose, better luck next time" message.
16. Insert the missing line of code so that the following adds the integers from 1 to 10, inclusive.

    ```
    value = 0;
    top = 10;
    score = 1;
    while (score <= top)
    {
        value = value + score;
          - - - -  //the missing line
    }
    ```


©2010 Course Technology, a part of Cengage Learning.

17. What is the output after the following main function is executed?

```
void main()
{
  int low, high;
  low = 1;
  high = 20;
  while (low < high)
  {
    cout << low << " " << high
    << endl;
    low = low + 1;
    high = high - 1;
  }
}
```

18. Write a C++ main function that outputs the even integers from 2 through 30, one per line. Use a while loop.

19. In a while loop, the Boolean condition that tests for loop continuation is done at the top of the loop, before each iteration of the loop body. As a consequence, the loop body might not be executed at all. Our pseudocode language of Chapter 2 contains a do-while loop construction in which a test for loop termination occurs at the bottom of the loop rather than at the top, so that the loop body always executes at least once. C++ contains a do-while statement that tests for loop continuation at the bottom of the loop. The form of the statement is

```
do
  S1;
while (Boolean condition);
```

where, as usual, S1 can be a compound statement. Write a C++ main function to add up a number of nonnegative integers that the user supplies and to write out the total. Use a negative value as a sentinel, and assume that the first value is nonnegative. Use a do-while statement.

20. Write a C++ program that asks for a duration of time in hours and minutes and writes out the duration only in minutes.

21. Write a C++ program that asks for the user's age in years. If the user is under 35, then quote an insurance rate of $2.23 per $100 for life insurance; otherwise, quote a rate of $4.32.

22. Write a C++ program that reads integer values until a 0 value is encountered and then writes out the sum of the positive values read and the sum of the negative values read.

23. Write a C++ program that reads in a series of positive integers and writes out the product of all the integers less than 25 and the sum of all the integers greater than or equal to 25. Use 0 as a sentinel value.

24. a. Write a C++ program that reads in 10 integer quiz grades and computes the average grade. (*Hint*: Remember the peculiarity of integer division.)

b. Write a C++ program that asks the user for the number of quiz grades, reads them in, and computes the average grade.

25. Write a (void) C++ function that receives two integer arguments and writes out their sum and their product.

26. Write a (void) C++ function that receives an integer argument representing the number of DVDs rented so far this month, and a real number argument representing the sales amount for DVDs sold so far this month. The function asks the user for the number of DVDs rented today and the sales amount for DVDs sold today, and then returns the updated figures to the main function.

27. Write a (nonvoid) C++ function that receives three integer arguments and returns the maximum of the three values.

28. Write a (nonvoid) C++ function that receives miles driven as a type *double* argument and gallons of gas used as a type *int* argument, and returns miles per gallon.

29. Write a C++ program that uses an input function to get the miles driven (type *double*) and the gallons of gas used (type *int*), then writes out the miles per gallon, using the function from Exercise 28.

30. Write a C++ program to balance a checkbook. The program needs to get the initial balance, the amounts of deposits, and the amounts of checks. Allow the user to process as many transactions as desired; use separate functions to handle deposits and checks.

31. Write a C++ program to compute the cost of carpeting three rooms. Make the carpet cost a constant of $8.95 per square yard. Use four separate functions to collect the dimensions of a room in feet, convert feet into yards, compute the area, and compute the cost per room. The main function should use a loop to process each of the three rooms, then add the three costs, and write out the total cost. (*Hint*: The function to convert feet into yards must be used twice for each room, with two different arguments. Hence, it does not make sense to try to give the parameter the same name as the argument.)

32. a. Write a C++ *doPerimeter* function for the *Rectangle* class of Figure 29.

b. Write C++ code that creates a new Rectangle object called *yuri*, then writes out information about this object and its perimeter using the *doPerimeter* function from part (a).

33. Draw a class hierarchy diagram similar to Figure 31 for the following classes: *Student*, *UndergraduateStudent*, *GraduateStudent*, *Sophomore*, *Senior*, *PhDStudent*.

34. Imagine that you are writing a program using an object-oriented programming language. Your program will be used to maintain records for a real estate office. Decide on one class in your program and a service that objects of that class might provide.

35. Determine the resolution of the screen on your computer (ask your instructor or the local computer center how to

©2010 Course Technology, a part of Cengage Learning.

do this). Using this information, determine how many bytes of memory are required for the frame buffer to store:

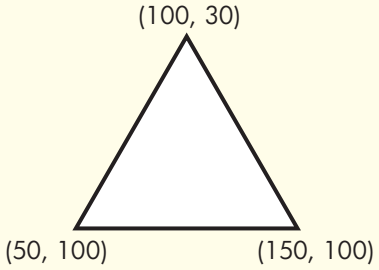
a. A black-and-white image (1 bit per pixel)

b. A grayscale image (8 bits per pixel)

c. A color image (24 bits per pixel)

36. Using the routines called *getmaxx* and *getmaxy,* determine the resolution of the laboratory software output window for your computer. After printing these values, resize the output window and print the new values. What is the largest size you can get for your output window?

37. Using the *moveto* and *lineto* commands described in Section 7.1, draw an isosceles triangle with the following configuration:



38. Discuss what problem the display hardware might encounter while attempting to execute the following operations, and explain how this problem could be solved.

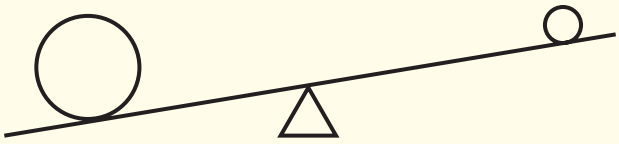
```
moveto(1,1);
lineto(4,5);
```

39. Draw a square with sides 100 pixels in length. Then inscribe a circle of radius 50 inside the square. Position the square and the inscribed circle in the middle of the screen.

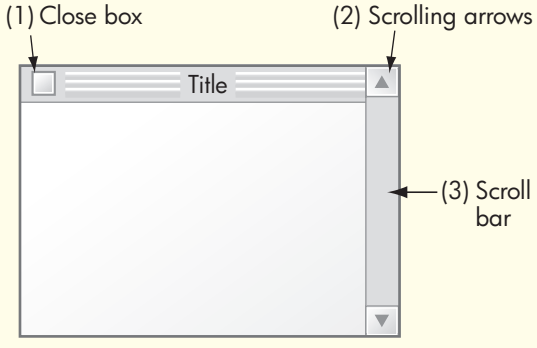
40. Create the following three labeled rectangular buttons in the output window.

Have the space between the Start and Stop buttons be the same as the space between the Stop and Pause buttons.

41. Create the following image of a "teeter-totter":



42. Add the three features shown in the following diagram to the titled window that was described in Section 7.2 and implemented by the code in Figure 34.



43. Write a program that inputs the coordinates of three mouse clicks from the user and then draws a triangle in the output window using those three points.

©2010 Course Technology, a part of Cengage Learning.

©2010 Course Technology, a part of Cengage Learning.

ANSWERS TO PRACTICE PROBLEMS

Section 2

1. The first three.
   `martinBradley` (camel case)
   `C3P_OH` (although best not to use underscore character)
   `Amy3` (Pascal case)
   `3Right` (not acceptable, begins with digit)
   `const` (not acceptable, C++ reserved word)
2. `int number;`
3. `const double TAX_RATE = 5.5;`
4. `hits[7]`

Section 3.1

1. `cout << "Enter a value for quantity" << endl;`
   `cin >> quantity;`
2. `cout << setw(6) << height << endl;`
3. `This isgoodbye`

Section 3.2

1. `next = newNumber;`
2. 55.0

Section 3.3

1. 30
2. 3
   5
   7
   9
   11
   13
   15
   17
   19
   21
3. Yes
4. 6
5. ```
   if (night == day)
       cout << "Equal" << endl;
   ```

Section 4

```
//program to read in and write out
//user's initials
#include <iostream>
using namespace std;
void main()
```


©2010 Course Technology, a part of Cengage Learning.
```

```
{
  char firstInitial, lastInitial;
  cout << "Give your first and last initials."
       << endl;
  cin >> firstInitial >> lastInitial;
  cout << "Your initials are " << firstInitial
       << lastInitial << endl;
}
```

2.

```
//program to compute cost based on price per item
//and quantity purchased
#include <iostream>
using namespace std;
void main()
{
  double price, cost;
  int quantity;
  cout.setf(ios::fixed);
  cout.precision(2);
  cout << "What is the price of the item? "
       << endl;
  cin >> price;
  cout << "How many of this item are being "
       << " purchased?" << endl;
  cin >> quantity;
  cost = price*quantity;
  cout << "The total cost for this item is $"
       << cost << endl;
}
```

3.

```
//program to test a number relative to 5
//and write out the number or its double
#include <iostream>
using namespace std;
void main()
{
  int number;
  cout << "Enter a number: ";
  cin >> number;
  if (number < 5)
       cout << number;
  else
       cout << 2*number;
  cout << endl;
}
```

4.

```
//program to collect a number, then write all
//the values from 1 to that number
#include <iostream>
using namespace std;
void main()
{
  int number;
  int counter;
```


©2010 Course Technology, a part of Cengage Learning.

```
  cout << "Enter a positive number: ";
  cin >> number;
  counter = 1;
  while (counter <= number)
  {
    cout << counter << endl;
    counter = counter + 1;
  }
}
```

Section 5.3

1. 11
2. 7
3.

```
void getInput(int &one, int &two)
{
  cout << "Please enter two integers" << endl;
  cin >> one >> two;
}
```

4. a. `double tax(double subtotal)`
   b. `return subtotal*RATE;`
   c. `cout << "The tax is " << tax(subtotal) << endl;`

Section 6.4

1. `The area of a square with side 10 is 100`
2. height and base

Section 7.1

```
rectangle(150, 150, 400, 350);
rectangle(170, 170, 210, 210);
rectangle(340, 170, 380, 210);
rectangle(250, 270, 300, 350);
moveto(150, 150);
lineto(275, 75);
lineto(400, 150);
circle(290, 310, 5);
```


©2010 Course Technology, a part of Cengage Learning.

©2010 Course Technology, a part of Cengage Learning.