



## *chapter three*

# Top-Down Design with Functions and Classes

### Chapter Objectives

- To learn about functions and how to use them to modularize programs
- To understand the capabilities of functions in the C++ math library
- To introduce structure charts as a system documentation tool
- To understand how control flows between functions
- To learn how to pass information to functions using arguments
- To learn how to return a value from a function
- To understand how to use class `string` and `string` objects and functions

PROGRAMMERS WHO SOLVE PROBLEMS using the software development method can use information collected during the analysis and design phases to help plan and complete the finished program. Also, programmers can use segments of earlier program solutions as components in their new programs. Therefore, they don't have to construct each new program from square one.

In the first section of this chapter, we demonstrate how you can tap existing information and code in the form of predefined functions to write programs. In addition to using existing information, you can use top-down design techniques to simplify the development of algorithms and the structure of the resulting programs. To apply top-down design, you start with the broadest statement of the problem solution and work down to more detailed subproblems.

We also introduce the structure chart, which documents the relationships among subproblems. We illustrate the use of procedural abstraction to develop modular programs that use separate functions to implement each subproblem's solution. Finally, we continue our

discussion of data abstraction and provide further detail on class `string` and its member functions.

### 3.1 Building Programs from Existing Information

Programmers seldom start off with a blank slate (or empty screen) when they develop a program. Often some—or all—of the solution can be developed from information that already exists or from the solution to another problem, as we demonstrate next.

Carefully following the software development method generates important system documentation before you even begin coding. This documentation—a description of a problem's data requirements (developed during the analysis phase) and its solution algorithm (developed during the design phase)—summarizes your intentions and thought processes.

You can use this documentation as a starting point in coding your program. For example, you can begin by editing the problem data requirements to conform to the C++ syntax for constant and variable declarations, as shown in Listing 3.1 for the miles-to-kilometers conversion program. This approach is especially helpful if the documentation was created with a word processor and is in a file that you can edit.

**Listing 3.1** Edited data requirements and algorithm for a conversion program

---

```
// File: miles.cpp
// Converts distance in miles to kilometers.

#include <iostream>
using namespace std;

int main()                // start of main function
{
    const float KM_PER_MILE = 1.609; // 1.609 km in a mile
    float miles,                // input: distance in miles
          kms;                  // output: distance in kilometers

    // Get the distance in miles.

    // Convert the distance to kilometers.
    // Distance in kilometers is 1.609 * distance in miles.

    // Display the distance in kilometers.

    return 0;
}
```

---

To develop the executable statements in the main function, first use the initial algorithm and its refinements as program comments. The comments describe each algorithm step and provide program documentation that guides your C++ code. Listing 3.1 shows how the program will look at this point. After the comments are in place in the main function, you can begin to write the C++ statements. Place the C++ code for an unrefined step directly under that step. For a step that is refined, either edit the refinement to change it from English to C++ or replace it with C++ code. We illustrate this entire process in the next case study.

## case study

### Finding the Area and Circumference of a Circle

#### PROBLEM

Get the radius of a circle. Compute and display the circle's area and circumference.

#### ANALYSIS

Clearly, the problem input is the circle radius. Two output values are requested: the circle's area and circumference. These variables should be type `float` because the inputs and outputs may contain fractional parts. The geometric relationship of a circle's radius to its area and circumference are listed below, along with the data requirements.

#### DATA REQUIREMENTS

##### Problem Constant

`PI = 3.14159`

##### Problem Input

`float radius`      *// radius of a circle*

##### Problem Output

`float area`      *// area of a circle*

`float circum`      *// circumference of a circle*

#### FORMULAS

*area of a circle =  $\pi \times \text{radius}^2$*

*circumference of a circle =  $2 \times \pi \times \text{radius}$*

#### DESIGN

After identifying the problem inputs and outputs, list the steps necessary to solve the problem. Pay close attention to the order of the steps.

#### INITIAL ALGORITHM

1. Get the circle radius.
2. Compute the area of circle.

### 3. Compute the circumference of circle.

#### ALGORITHM REFINEMENTS

Next, refine any steps that don't have an obvious solution (Steps 2 and 3).

#### Step 2 Refinement

2.1. Assign  $\text{PI} * \text{radius} * \text{radius}$  to area.

#### Step 3 Refinement

3.1. Assign  $2 * \text{PI} * \text{radius}$  to circum.

#### IMPLEMENTATION

Listing 3.2 shows the C++ program so far. Function main consists of the initial algorithm with its refinements as comments. To write the final program, convert the refinements (Steps 2.1 and 3.1) to C++ and write C++ code for the unrefined steps (Steps 1 and 4). Listing 3.3 shows the final program.

#### TESTING

The sample output in Listing 3.3 provides a good test of the solution because it is relatively easy to compute by hand the area and circumference for a radius value of 5.0. The radius squared is 25.0, so the value of the area is correct. The circumference should be ten times  $\pi$ , which is also an easy number to compute by hand.

**Listing 3.2** Outline of area and circumference program

---

```
// Computes and displays the area and circumference of a circle

int main()
{
    const float PI = 3.14159;
    float radius;      // input: radius of circle
    float area;        // output: area of circle
    float circum;      // output: circumference of circle

    // Get the circle radius.

    // Compute area of circle.
        // Assign  $\text{PI} * \text{radius} * \text{radius}$  to area.

    // Compute circumference of circle.
        // Assign  $2 * \text{PI} * \text{radius}$  to circum.

    // Display area and circumference.

    return 0;
}
```

**Listing 3.3** Finding the area and circumference of a circle

```
// File: circle.cpp
// Computes and displays the area and circumference of a circle

#include <iostream>
using namespace std;

int main()
{
    const float PI = 3.14159;
    float radius;      // input: radius of circle
    float area;        // output: area of circle
    float circum;      // output: circumference of circle

    // Get the circle radius.
    cout << "Enter the circle radius: ";
    cin >> radius;

    // Compute area of circle.
    area = PI * radius * radius;

    // Compute circumference of circle.
    circum = 2 * PI * radius;

    // Display area and circumference.
    cout << "The area of the circle is " << area << endl;
    cout << "The circumference of the circle is " << circum << endl;

    return 0;
}
```

```
Enter the circle radius: 5.0
The area of the circle is 78.539749
The circumference of the circle is 31.415901
```

*case study***Computing the Weight of a Batch of Flat Washers**

Another way in which programmers use existing information is by *extending the solution for one problem to solve another*. For example, you can easily solve this new problem by building on the solution to the previous one.

**PROBLEM**

You work for a hardware company that manufactures flat washers. To estimate shipping costs, your company needs a program that computes the weight of a specified quantity of flat washers.

## ANALYSIS

A flat washer resembles a small donut. To compute the weight of a single flat washer, you need to know its rim area, its thickness, and the density of the material used in its construction. The last two quantities are problem inputs. However, the rim area (see Figure 3.1) must be computed from two measurements that are provided as inputs: the washer's outer diameter and its inner diameter (diameter of the hole).

In the following data requirements, we list the washer's inner and outer radius (half the diameter) as program variables. We also list the rim area and weight of one washer (`unitWeight`) as program variables.

## DATA REQUIREMENTS

## Problem Constant

PI = 3.14159

## Problem Inputs

float holeDiameter	// diameter of hole
float edgeDiameter	// diameter of outer edge
float thickness	// thickness of washer
float density	// density of material used
float quantity	// number of washers made

## Problem Outputs

float weight	// weight of batch of washers
--------------	-------------------------------

## Program Variables

float holeRadius	// radius of hole
float edgeRadius	// radius of outer edge
float rimArea	// area of rim
float unitWeight	// weight of 1 washer

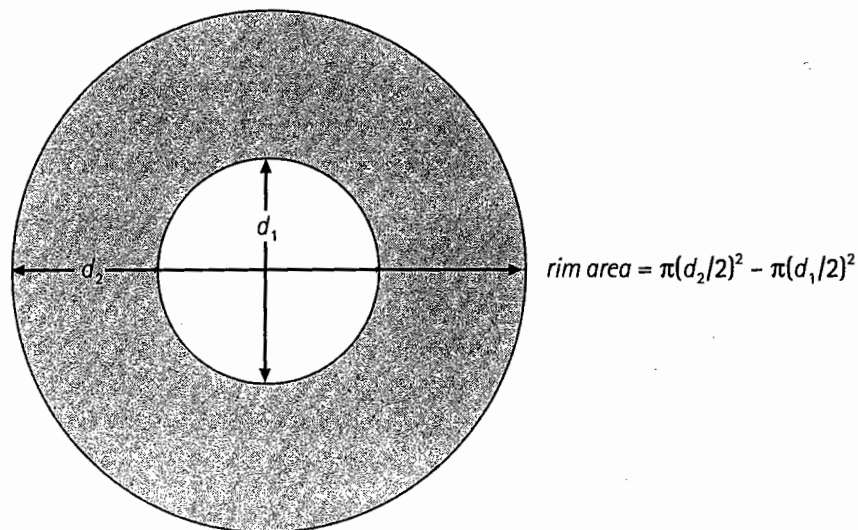


Figure 3.1 The rim area of a washer



**Relevant Formulas**

*area of circle* =  $\pi \times \text{radius}^2$

*radius of circle* = *diameter* / 2

*rim area* = *area of outer circle* – *area of hole*

*unit weight* = *rim area*  $\times$  *thickness*  $\times$  *density*

**DESIGN**

We list the algorithm next, followed by the refinement of Steps 3 and 4.

**INITIAL ALGORITHM**

1. Get the washer's inner diameter, outer diameter, and thickness.
2. Get the material density and quantity of washers manufactured.
3. Compute the rim area.
4. Compute the weight of one flat washer.
5. Compute the weight of the batch of washers.
6. Display the weight of the batch of washers.

**ALGORITHM REFINEMENTS****Step 3 Refinement**

3.1 Compute holeRadius and edgeRadius.

3.2 rimArea is  $\text{PI} * \text{edgeRadius} * \text{edgeRadius} -$   
 $\text{PI} * \text{holeRadius} * \text{holeRadius}$

**Step 4 Refinement**

4.1 unitWeight is  $\text{rimArea} * \text{thickness} * \text{density}$

**IMPLEMENTATION**

To write this program, edit the data requirements to write the variable declarations and use the initial algorithm with refinements as a starting point for the executable statements. Listing 3.4 shows the C++ program.

**TESTING**

To test this program, run it with inner and outer diameters such as 2 centimeters and 4 centimeters that lead to easy calculations for rim area ( $3 * \text{PI}$  square centimeters). You can verify that the program is computing the correct unit weight by entering 1 for quantity, and then verify that the batch weight is correct by running it for larger quantities.

## Listing 3.4 Washer program

---

```

// File: washers.cpp
// Computes the weight of a batch of flat washers.

#include <iostream>
using namespace std;

int main()
{
    const float PI = 3.14159;
    float holeDiameter; // input - diameter of hole
    float edgeDiameter; // input - diameter of outer edge
    float thickness;     // input - thickness of washer
    float density;       // input - density of material used
    float quantity;     // input - number of washers made
    float weight;        // output - weight of washer batch
    float holeRadius;    // radius of hole
    float edgeRadius;    // radius of outer edge
    float rimArea;       // area of rim
    float unitWeight;    // weight of 1 washer

    // Get the inner diameter, outer diameter, and thickness.
    cout << "Inner diameter in centimeters: ";
    cin >> holeDiameter;
    cout << "Outer diameter in centimeters: ";
    cin >> edgeDiameter;
    cout << "Thickness in centimeters: ";
    cin >> thickness;

    // Get the material density and quantity manufactured.
    cout << "Material density in grams per cubic centimeter: ";
    cin >> density;
    cout << "Quantity in batch: ";
    cin >> quantity;

    // Compute the rim area.
    holeRadius = holeDiameter / 2.0;
    edgeRadius = edgeDiameter / 2.0;
    rimArea = PI * edgeRadius * edgeRadius -
              PI * holeRadius * holeRadius;

    // Compute the weight of a flat washer.
    unitWeight = rimArea * thickness * density;

    // Compute the weight of the batch of washers.
    weight = unitWeight * quantity;

```

(continued)



**Listing 3.4** Washer program (continued)

---

```
// Display the weight of the batch of washers.
cout << "The expected weight of the batch is "
      << weight << " grams." << endl;
return 0;
}
Inner diameter in centimeters: 1.2
Outer diameter in centimeters: 2.4
Thickness in centimeters: 0.1
Material density in grams per cubic centimeter: 7.87
Quantity in batch: 1000
The expected weight of the batch is 2670.23 grams.
```

---

**EXERCISES FOR SECTION 3.1****Self-Check**

1. Describe the data requirements and algorithm for a program that computes the number of miles you can drive a car given the estimated number of miles per gallon and the number of gallons of gas you purchased as input data. Also compute and display the cost of the gasoline based on the cost per gallon (a data item).
2. Write a program outline from the algorithm you developed in Exercise 1. Show the declaration part of the program and the program comments corresponding to the algorithm and its refinements.
3. Change the solution to Exercise 1 to calculate the estimated cost of a trip given the distance of the trip, the estimated number of miles per gallon, and the average cost of a gallon of gasoline.
4. Describe the problem inputs and outputs and write the algorithm for a program that computes an employee's gross salary given the hours worked and the hourly rate.
5. Write a preliminary version of the program for Self-Check Exercise 4 showing the declaration part of the program and the program comments corresponding to the algorithm and its refinements.
6. In computing gross salary, what changes should you make in order to include overtime hours to be paid at 1.5 times an employee's normal hourly rate? Assume that overtime hours are entered separately.

**Programming**

1. Add refinements to the program outline below and write the final C++ program.
-

```

// Computes the sum and average of two numbers
#include <iostream>
using namespace std;

int main()
{
    // Declare any constants and variables you need.
    // Read two numbers.
    // Compute the sum of the two numbers.
    // Compute the average of the two numbers.
    // Display sum and average.
    return 0;
}

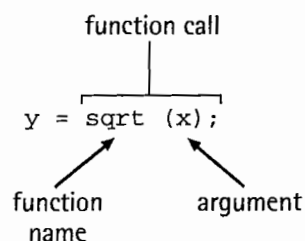
```

2. Write a complete C++ program for Self-Check Exercise 2.
3. Write a complete C++ program for Self-Check Exercise 3.
4. Assume that flat washers are manufactured by stamping them out from a rectangular piece of material of uniform thickness. Extend the washer program to compute (a) the number of square centimeters of material needed to manufacture a specified quantity of flat washers and (b) the weight of the leftover material.
5. Write a complete C++ program for Self-Check Exercise 5.
6. Write a complete C++ program for Self-Check Exercise 6.

## 3.2 Library Functions

A main goal of software engineering is to write error-free code. Code reuse—reusing program fragments that have already been written and tested whenever possible—is one efficient way to accomplish this goal. Stated more simply, why reinvent the wheel?

C++ promotes reuse by providing many predefined classes and functions in its standard library. The standard library `cmath` contains many functions that perform mathematical computations. For example, it defines a function named `sqrt` that performs the square root computation. The function call in the assignment statement



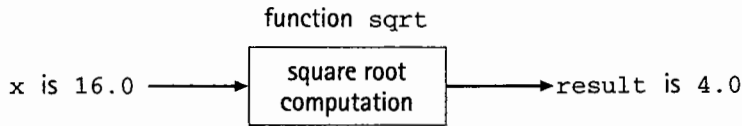


Figure 3.2 Function `sqrt` as a “black box”

activates the code for function `sqrt`, passing the *argument* `x` to the function. You activate a function by writing a *function call*. After the function executes, the function result is substituted for the function call. If `x` is `16.0`, the assignment statement above is evaluated as follows:

1. `x` is `16.0`, so function `sqrt` computes  $\sqrt{16.0}$ , or `4.0`.
2. The function result `4.0` is assigned to `y`.

A function can be thought of as a “black box” that is passed one or more input values and automatically returns a single output value. Figure 3.2 illustrates this for the call to function `sqrt`. The value of `x` (`16.0`) is the function input, and the function result, or output, is  $\sqrt{16.0}$  (result is `4.0`).

If `w` is `9.0`, C++ evaluates the assignment statement

```
z = 5.7 + sqrt(w);
```

as follows:

1. `w` is `9.0`, so function `sqrt` computes  $\sqrt{9.0}$ , or `3.0`.
2. The values `5.7` and `3.0` are added together.
3. The sum, `8.7`, is stored in `z`.

### EXAMPLE 3.1

The program in Listing 3.5 displays the square root of two numbers provided as input data (`first` and `second`) and the square root of their sum. To do so, we must call the function `sqrt` three times:

```
answer = sqrt(first);
answer = sqrt(second);
answer = sqrt(first + second);
```

For the first two calls, the function arguments are variables (`first` and `second`). The third call shows that a function argument can also be an expression (`first + second`). For all three calls, the result returned by function `sqrt` is assigned to variable `answer`. Because the definition of function `sqrt` is found in the standard library `cmath`, the program begins with

```
#include <cmath>          // sqrt function
```

Listing 3.5 Illustration of the use of the C++ `sqrt` function

```

// File: squareRoot.cpp
// Performs three square root computations

#include <cmath>           // sqrt function
#include <iostream>        // i/o functions
using namespace std;

int main()
{
    float first;           // input: one of two data values
    float second;         // input: second of two data values
    float answer;          // output: a square root value

    // Get first number and display its square root.
    cout << "Enter the first number: ";
    cin >> first;
    answer = sqrt(first);
    cout << "The square root of the first number is "
         << answer << endl;

    // Get second number and display its square root.
    cout << "Enter the second number: ";
    cin >> second;
    answer = sqrt(second);
    cout << "The square root of the second number is "
         << answer << endl;

    // Display the square root of the sum of first and second.
    answer = sqrt(first + second);
    cout << "The square root of the sum of both numbers is "
         << answer << endl;
    return 0;
}

```

```

Enter the first number: 9
The square root of the first number is 3
Enter the second number: 25
The square root of the second number is 5
The square root of the sum of both numbers is 5.83095

```

## C++ Library Functions

Table 3.1 lists the names and descriptions of some of the most commonly used functions along with the name of the standard file to `#include` in order to have access to each function.

If one of the functions in Table 3.1 is called with a numeric argument that is not of the argument type listed, the argument value is converted to the required

Table 3.1 Some Mathematical Library Functions

Function	Standard Library	Purpose: Example	Argument(s)	Result
<code>abs(x)</code>	<code>&lt;cstdlib&gt;</code>	Returns the absolute value of its integer argument: if <code>x</code> is <code>-5</code> , <code>abs(x)</code> is <code>5</code>	<code>int</code>	<code>int</code>
<code>ceil(x)</code>	<code>&lt;cmath&gt;</code>	Returns the smallest integral value that is not less than <code>x</code> : if <code>x</code> is <code>45.23</code> , <code>ceil(x)</code> is <code>46.0</code>	<code>double</code>	<code>double</code>
<code>cos(x)</code>	<code>&lt;cmath&gt;</code>	Returns the cosine of angle <code>x</code> : if <code>x</code> is <code>0.0</code> , <code>cos(x)</code> is <code>1.0</code>	<code>double</code> (radians)	<code>double</code>
<code>exp(x)</code>	<code>&lt;cmath&gt;</code>	Returns $e^x$ where $e = 2.71828 \dots$ : if <code>x</code> is <code>1.0</code> , <code>exp(x)</code> is <code>2.71828</code>	<code>double</code>	<code>double</code>
<code>fabs(x)</code>	<code>&lt;cmath&gt;</code>	Returns the absolute value of its type <code>double</code> argument: if <code>x</code> is <code>-8.432</code> , <code>fabs(x)</code> is <code>8.432</code>	<code>double</code>	<code>double</code>
<code>floor(x)</code>	<code>&lt;cmath&gt;</code>	Returns the largest integral value that is not greater than <code>x</code> : if <code>x</code> is <code>45.23</code> , <code>floor(x)</code> is <code>45.0</code>	<code>double</code>	<code>double</code>
<code>log(x)</code>	<code>&lt;cmath&gt;</code>	Returns the natural logarithm of <code>x</code> for <code>x &gt; 0.0</code> : if <code>x</code> is <code>2.71828</code> , <code>log(x)</code> is <code>1.0</code>	<code>double</code>	<code>double</code>
<code>log10(x)</code>	<code>&lt;cmath&gt;</code>	Returns the base-10 logarithm of <code>x</code> for <code>x &gt; 0.0</code> : if <code>x</code> is <code>100.0</code> , <code>log10(x)</code> is <code>2.0</code>	<code>double</code>	<code>double</code>
<code>pow(x, y)</code>	<code>&lt;cmath&gt;</code>	Returns $x^y$ . If <code>x</code> is negative, <code>y</code> must be integral: if <code>x</code> is <code>0.16</code> and <code>y</code> is <code>0.5</code> , <code>pow(x, y)</code> is <code>0.4</code>	<code>double</code> , <code>double</code>	<code>double</code>
<code>sin(x)</code>	<code>&lt;cmath&gt;</code>	Returns the sine of angle <code>x</code> : if <code>x</code> is <code>1.5708</code> , <code>sin(x)</code> is <code>1.0</code>	<code>double</code> (radians)	<code>double</code>
<code>sqrt(x)</code>	<code>&lt;cmath&gt;</code>	Returns the non-negative square root of <code>x</code> $\sqrt{x}$ for <code>x ≥ 0.0</code> : if <code>x</code> is <code>2.25</code> , <code>sqrt(x)</code> is <code>1.5</code>	<code>double</code>	<code>double</code>
<code>tan(x)</code>	<code>&lt;cmath&gt;</code>	Returns the tangent of angle <code>x</code> : if <code>x</code> is <code>0.0</code> , <code>tan(x)</code> is <code>0.0</code>	<code>double</code> (radians)	<code>double</code>

type before it is used. Conversions of type `int` or type `float` to type `double` cause no problems, but a conversion of type `float` or type `double` to type `int` leads to the loss of any fractional part, just as in a mixed-type assignment. For example, if we call the `abs` function (in library `cstdlib`) with an argument value of `-3.47`, the argument is converted to `-3` and the result returned is the type `int` value `3`. For this reason, there is another absolute value function (`fabs` in library `cmath`) for floating-point arguments.

Most of the functions in Table 3.1 perform common mathematical computations. The arguments for `log` and `log10` must be positive; the argument for `sqrt` cannot be negative. The arguments for `sin`, `cos`, and `tan` must be expressed in radians, not in degrees.

### EXAMPLE 3.2

We can use the C++ functions `sqrt` and `pow` to compute the roots of a quadratic equation in  $x$  of the form

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

These two roots are defined as

$$\text{root}_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{root}_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

when the discriminant ( $b^2 - 4ac$ ) is greater than zero. If we assume that this is the case, we can use these assignment statements to assign values to `root1` and `root2`.

```
// Compute 2 roots, root1 & root2, for discriminant values > 0.
disc = pow(b, 2) - 4.0 * a * c;
root1 = (-b + sqrt(disc)) / (2.0 * a);
root2 = (-b - sqrt(disc)) / (2.0 * a);
```

### EXAMPLE 3.3

If we know the lengths of two sides ( $b$  and  $c$ ) of a triangle and the angle between them in degrees ( $\alpha$ ) (see Figure 3.3), we can compute the length of the third side,  $a$ , by using the formula

$$a^2 = b^2 + c^2 - 2bc(\cos(\alpha))$$

To use the `cmath` library cosine function (`cos`), we must express its argument angle in radians instead of degrees. To convert an angle from degrees to radians, we multiply the angle by  $\pi/180$ . If we assume `PI` represents the constant  $\pi$ , the assignment statement that follows computes the length of the unknown side.

```
a = sqrt(pow(b,2) + pow(c,2)
        - 2 * b * c * cos(alpha * PI / 180.0));
```

## A Look Ahead

C++ allows us to write our own functions such as `findArea` and `findCircum`:

- Function `findArea(r)` returns the area of a circle with radius  $r$ .
- Function `findCircum(r)` returns the circumference of a circle with radius  $r$ .



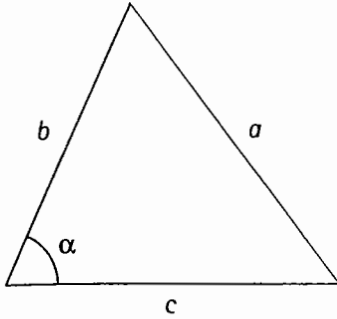


Figure 3.3 Triangle with unknown side  $a$

We can reuse these functions in two programs shown earlier in this chapter (see Listings 3.3 and 3.4). The program in Listing 3.3 computes the area and the circumference of a circle. The statements

```
area = findArea(radius);
circum = findCircum(radius);
```

can be used to find these values. The expression part for each of the assignment statements is a function call with argument `radius` (the circle radius). The result returned by each function execution is assigned to the variable listed to the left of the `=` (assignment) operator.

For the flat washer program (Listing 3.4), we can use the statement

```
rimArea = findArea(edgeRadius) - findArea(holeRadius);
```

to compute the rim area for a washer. This statement is clearer than the one shown in the original program. We show how to write these functions in Section 3.5.

## EXERCISES FOR SECTION 3.2

### Self-Check

1. Rewrite the following mathematical expressions using C++ functions:

- |                            |                     |
|----------------------------|---------------------|
| a. $\sqrt{u+v} \times w^2$ | c. $\sqrt{(x-y)^3}$ |
| b. $\log_e(x^y)$           | d. $ a/c - wz $     |

2. Evaluate the following:

- `floor(16.2)`
- `floor(16.7 + 0.5)`
- `ceil(-9.2) * pow(4.0, 3)`
- `sqrt(fabs(floor(-24.8)))`
- `log10(10000.0)`

3. What happens if you call `abs` with a type `float` argument?

4. What happens if you call `ceil` with a type `int` argument?

## Programming

1. Write statements that compute and display the absolute difference of two type double variables,  $x$  and  $y$  ( $|x - y|$ ).
2. Write a complete program that prompts the user for the Cartesian coordinates of two points  $(x_1, y_1)$  and  $(x_2, y_2)$  and displays the distance between them computed using the following formula:

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

3. Write a program that prompts the user for the lengths of two sides of a triangle and the angle between them (in degrees) and calculates the length of the third side.

### 3.3 Top-Down Design and Structure Charts

#### top-down design

A problem-solving method in which the programmer breaks a problem up into its major subproblems and then solves the subproblems to derive the solution to the original problem.

#### structure chart

A documentation tool that shows the relationships among the subproblems of a problem.

Often the algorithm needed to solve a problem is more complex than those we have seen so far and the programmer has to break up the problem into subproblems to develop the program solution. In attempting to solve a subproblem at one level, we introduce new subproblems at lower levels. This process, called **top-down design**, proceeds from the original problem at the top level to the subproblems at each lower level. The splitting of a problem into its related subproblems is analogous to the process of refining an algorithm. The following case study introduces a documentation tool—the **structure chart**—that will help you to keep track of the relationships among subproblems.

#### case study

#### Drawing Simple Figures

##### PROBLEM

You want to draw some simple diagrams on your screen. Two examples are the house and female stick figure shown in Figure 3.4.

##### ANALYSIS

The house is formed by displaying a triangle without its base on top of a rectangle. The stick figure consists of a circular shape, a triangle, and a triangle without its base. We can draw both figures with four basic components:

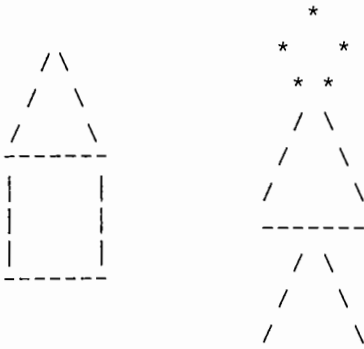


Figure 3.4 House and stick figure

- a circle
- a base line
- parallel lines
- intersecting lines

## DESIGN

To create the stick figure, you can divide the problem into three subproblems.

### INITIAL ALGORITHM

1. Draw a circle.
2. Draw a triangle.
3. Draw intersecting lines.

### ALGORITHM REFINEMENTS

Because a triangle is not a basic component, you must refine Step 2:

#### Step 2 Refinement

- 2.1. Draw intersecting lines.
- 2.2. Draw a base line.

You can use a structure chart to show the relationship between the original problem and its subproblems (Figure 3.5). The original problem (Level 0) is in the darker color and its three subordinate subproblems are shown at Level 1. The subproblem "Draw a triangle" is also in color because it has its own subproblems (shown at Level 2).

The subproblems appear in both the algorithm and the structure chart. The algorithm, not the structure chart, shows the order in which you carry out each step to solve the problem. The structure chart simply illustrates the subordination of subproblems to each other and to the original problem.

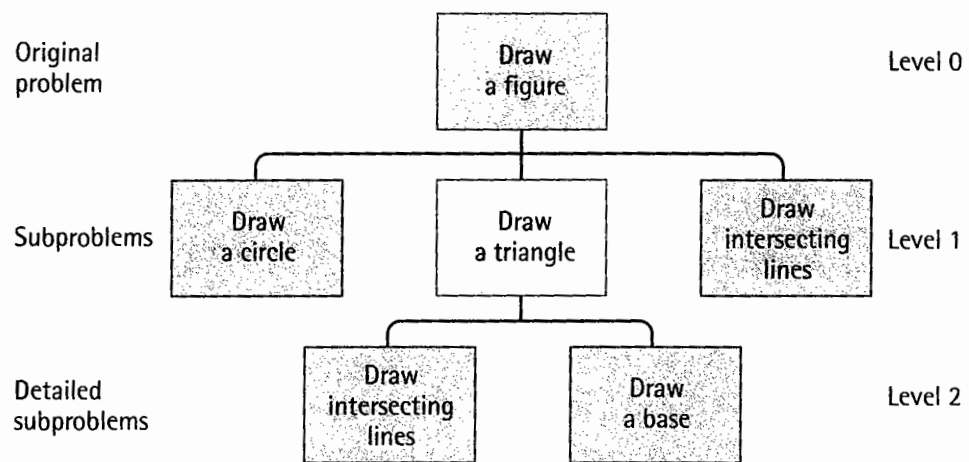


Figure 3.5 Structure chart for drawing a stick figure

### EXERCISES FOR SECTION 3.3

#### Self-Check

1. Draw the structure chart for the subproblem of drawing a house (see Figure 3.4).
2. Draw the structure chart for the problem of drawing a rocket ship that consists of a triangle on top of a rectangle that is on top of a second rectangle. The bottom rectangle rests on a triangular base.
3. In which phase of the software development method do you apply top-down design to break the problem into subproblems?

## 3.4 Functions without Arguments

One way that programmers use top-down design in their programs is by designing their own functions. Often a programmer will write one function subprogram for each subproblem in the structure chart. In this section we show how to use and define your own functions, focusing on simple functions that have no arguments and return no value.

As an example of top-down design with functions, you could use the main function in Listing 3.6 to draw the stick figure. In that figure, the three algorithm steps are coded as calls to three C++ function subprograms. For example, the statement

```
// Draw a circle.
drawCircle();
```

**Listing 3.6** Function prototypes and main function for drawing a stick figure

---

```
// Draws a stick figure (main function only)

#include <iostream>
using namespace std;

// Functions used...
void drawCircle();    // Draws a circle

void drawTriangle();  // Draws a triangle

void drawIntersect(); // Draws intersecting lines

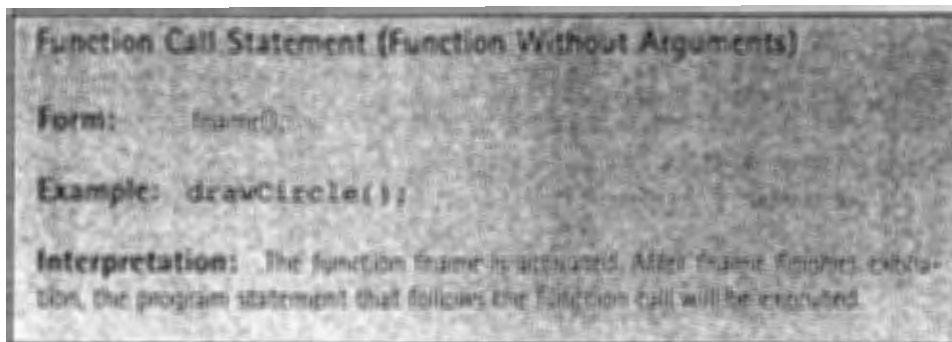
void drawBase();      // Draws a horizontal line

int main()
{
    // Draw a circle.
    drawCircle();
    // Draw a triangle.
    drawTriangle();
    // Draw intersecting lines.
    drawIntersect();

    return 0;
}
```

---

in function `main` calls function `drawCircle` to implement the algorithm step *Draw a Circle*. We call function `drawCircle` just like we call the functions in library `cmath`. The empty parentheses after the function name indicate that `drawCircle` requires no arguments.



## Function Prototypes

Just like other identifiers in C++, a function must be declared before it can be referenced. One way to declare a function is to insert a function prototype before the main function. A function prototype tells the C++ compiler the data type of the function, the function name, and information about the arguments that the function expects. The data type of a function is determined by the type of value returned by the function. In Listing 3.6, the prototype for `drawCircle` is written as:

```
void drawCircle();
```

### **void function**

A function that does not return a value.

The reserved word **void** indicates that `drawCircle` is a **void function**—that is, its type is **void**—because it doesn't return a value. The empty parentheses after the function name indicate that `drawCircle` has no arguments.

### Function Prototype (Function Without Arguments)

**Form:** `ftype fname();`

**Example:** `void skipThree();`

**Interpretation:** The identifier `fname` is declared to be the name of a function. The identifier `ftype` specifies the data type of the function result.

**Note:** `ftype` is **void** if the function does not return a value. The argument list `()` indicates that the function has no arguments. The function prototype must appear before the first call to the function. We recommend you place all prototypes before function `main`.

## Function Definitions

Although the prototype specifies the number of arguments a function takes and the type of its result, it doesn't specify the function operation. To do this, you need to provide a definition for each function subprogram similar to the definition of the `main` function. Listing 3.7 shows the definition for function `drawCircle`.

**Listing 3.7** Function `drawCircle`

```
// Draws a circle
void drawCircle()
{
    cout << "    * " << endl;
    cout << " *    *" << endl;
    cout << " * * " << endl;
}
```



The function heading is similar to the function prototype in Listing 3.6 except that it doesn't end with a semicolon. The function body, enclosed in braces, consists of three output statements that cause the computer to display a circular shape. We omit the `return` statement because `drawCircle` doesn't return a result.

The function call statement

```
drawCircle();
```

causes the three output statements to execute. Control returns to the main function after the circle shape is displayed.

#### Function Definition (Function Without Arguments)

**Syntax:** `fctype fname()`  
 {  
     local declarations  
     executable statements  
 }

**Example:** *// Displays a block-letter H*  

```
void printH()
{
    cout << "***  **" << endl;
    cout << "***  **" << endl;
    cout << "*****" << endl;
    cout << "*****" << endl;
    cout << "***  **" << endl;
    cout << "***  **" << endl;
}
```

**Interpretation:** The function `fname` is defined. In the function heading, the identifier `fctype` specifies the data type of the function result. Notice that there are no semicolons after the function heading. The braces enclose the function body. Any identifiers that are declared in the optional local declarations are defined only during the execution of the function and can be referenced only within the function. The executable statements of the function body describe the data manipulation to be performed by the function.

**Note:** `fctype` is `void` if the function does not return a value. The argument list `()` indicates that the function has no arguments.

Each function body may contain declarations for its own variables. These variables are considered *local* to the function; in other words, they can be referenced only within the function. There will be more on this topic later.

The structure chart in Figure 3.5 shows that the subproblem *Draw a triangle* (Level 1) depends on the solutions to its subordinate subproblems *Draw intersecting lines* and *Draw a base* (both Level 2). Listing 3.8 shows how you can

use top-down design to code function `drawTriangle`. Instead of using output statements to display a triangular pattern, the body of function `drawTriangle` calls functions `drawIntersect` and `drawBase` to draw a triangle.

## Placement of Functions in a Program

Listing 3.9 shows the complete program with function subprograms. The subprogram prototypes precede the main function (after any `#include` directives) and the subprogram definitions follow the main function. The relative order of the function definitions doesn't affect their order of execution; that is determined by the order of execution of the function call statements.

### PROGRAM STYLE Use of Comments in Function Declarations and Definitions

Listing 3.9 includes several comments. A comment precedes each function and describes its purpose. The same comment follows the prototype declaration. For clarity, the right brace at the end of each function may be followed by a comment, such as

```
// end fname
```

identifying that function.

## Order of Execution of Functions

The prototypes for the function subprograms appear before the main function, so the compiler processes the function prototypes before it translates the main function. The information in each prototype lets the compiler correctly translate a call to that function. The compiler translates a function call statement as a transfer of control to the function.

After compiling the main function, the compiler translates each function subprogram. During translation, when the compiler reaches the end of a function body, it inserts a machine language statement that causes a *transfer of control* back from the function to the calling statement.

### Listing 3.8 Function `drawTriangle`

---

```
// Draws a triangle
void drawTriangle()
{
    drawIntersect();
    drawBase();
}
```

---

Listing 3.9 Program to draw a stick figure

---

```
// File: stickFigure.cpp
// Draws a stick figure

#include <iostream>
using namespace std;

// Functions used...
void drawCircle();    // Draws a circle

void drawTriangle(); // Draws a triangle

void drawIntersect(); // Draws intersecting lines

void drawBase();      // Draws a horizontal line

int main()
{
    // Draw a circle.
    drawCircle();

    // Draw a triangle.
    drawTriangle();

    // Draw intersecting lines.
    drawIntersect();

    return 0;
}

// Draws a circle
void drawCircle()
{
    cout << "  *  " << endl;
    cout << " *  *" << endl;
    cout << "  *  " << endl;
} // end drawCircle

// Draws a triangle
void drawTriangle()
{
    drawIntersect();
    drawBase();
} // end drawTriangle
```

(continued)

**Listing 3.9** Program to draw a stick figure (continued)

---

```

// Draws intersecting lines
void drawIntersect()
{
    cout << "    /\\" << endl;
    cout << "  /  \\" << endl;
    cout << " /    \\" << endl;
} // end drawIntersect

// Draws a horizontal line
void drawBase()
{
    cout << " _ _ _ _ _" << endl;
} // end drawBase

```

---

Figure 3.6 shows the main function and function `drawCircle` of the stick figure program in separate areas of memory. Although the C++ statements are shown in Figure 3.6, it is actually the object code corresponding to each statement that is stored in memory.

When we run the program, the first statement in the main function is the first statement executed (the call to `drawCircle` in Figure 3.6). When the computer executes a function call statement, it transfers control to the function that is referenced (indicated by the top arrow in Figure 3.6). The computer allocates any memory that may be needed for variables declared in the function and then performs the statements in the function body. After the last statement in function `drawCircle` is executed, control returns to the main function (indicated by the bottom arrow in Figure 3.6), and the computer releases any memory that was allocated to the function. After the return to the main function, the next statement is executed (the call to `drawTriangle`).

## Advantages of Using Function Subprograms

There are many advantages to using function subprograms. Their availability changes the way an individual programmer organizes the solution to a

in main function

```

drawCircle();
drawTriangle();
drawIntersect();

```

```

void drawCircle()
{
    cout << "...
    cout << "...
    cout << "...
    return to calling function
} //end drawCircle

```

**Figure 3.6** Flow of control between main function and subprogram

programming problem. For a team of programmers working together on a large program, subprograms make it easier to apportion programming tasks: Each programmer will be responsible for a particular set of functions. Finally, subprograms simplify programming tasks because existing functions can be reused as the building blocks for new programs.

## Procedural Abstraction

Function subprograms allow us to remove from the main function the code that provides the detailed solution to a subproblem. Because these details are provided in the function subprograms and not in the main function, we can write the main function as a sequence of function call statements as soon as we have specified the initial algorithm and before we refine any of the steps. We should delay writing the function for an algorithm step until we have finished refining that step. With this approach to program design, called **procedural abstraction**, we defer implementation details until we're ready to write an individual function subprogram. Focusing on one function at a time is much easier than trying to write the complete program all at once.

### procedural abstraction

A programming technique in which the main function consists of a sequence of function calls and each function is implemented separately.

## Reuse of Function Subprograms

Another advantage of using function subprograms is that functions can be executed more than once in a program. For example, function `drawIntersect` is called twice in Listing 3.9 (once by `drawTriangle` and once by the main function). Each time `drawIntersect` is called, the list of output statements in `drawIntersect` is executed and a pair of intersecting lines is drawn. Without functions, the output statements that draw the lines would

### EXAMPLE 3.4

Let's write a function (Listing 3.10) that displays instructions to a user of the program that computes the area and the circumference of a circle (see Listing 3.3). This simple function demonstrates one of the benefits of separating the statements that display user instructions from the main function body: Editing these instructions is simplified when they are separated from the code that performs the calculations.

If you place the prototype for function `instruct`

```
void instruct();
```

just before the main function, you can insert the function call statement

```
instruct();
```

as the first executable statement in the main function. The rest of the main function consists of the executable statements shown earlier. We show the output displayed by calling function `instruct` at the bottom of Listing 3.10.

Listing 3.10 Function instruct

---

```

// Displays instructions to user of area/circumference
// program

void instruct()
{
    cout << "This program computes the area and " << endl;
    cout << "circumference of a circle. " << endl << endl;
    cout << "To use this program, enter the radius of the "
        << endl;
    cout << "circle after the prompt" << endl << endl;
    cout << "Enter the circle radius: " << endl << endl;
    cout << "The circumference is computed in the same"
        << endl;
    cout << "units of measurement as the radius. The area "
        << endl;
    cout << "is computed in the same units squared."
        << endl << endl;
} // end instruct

```

---

This program computes the area and  
circumference of a circle.

To use this program, enter the radius of the  
circle after the prompt

Enter the circle radius:

The circumference is computed in the same  
units of measurement as the radius. The area  
is computed in the same units squared.

---

be listed twice in the main function, thereby increasing the main function's length and the chance of error.

Finally, once you have written and tested a function, you can use it in other programs or functions. The functions in the stick figure program, for example, could easily be reused in programs that draw other diagrams.

### Displaying User Instructions

The simple functions introduced in this section have limited capability. Without the ability to pass information into or out of a function, we can use functions only to display multiple lines of program output, such as instructions to a program user or a title page or a special message that precedes a program's results.

---



## EXERCISES FOR SECTION 3.4

### Self-Check

1. Assume that you have functions `printH`, `printI`, `printM`, and `printO`, each of which draws a large block letter (for example, `printO` draws a block letter `O`). What is the effect of executing the following main function?

```
int main()
{
    printO();
    cout << endl;
    printH();
    skipThree(); // see Programming Exercise 1

    printH();
    cout << endl;
    printI();
    cout << endl;
    printM();
}
```

2. Draw a structure chart for a program with three function subprograms that displays `POPS` in a vertical column of block letters.
3. If you write a program followed by a collection of functions, do the functions execute in the order in which they are listed before the main function (the function prototypes), the order in which they are listed after the main function (the function definitions), or neither? If your answer is neither, what determines the order in which the functions execute?
4. What error message do you get when a function prototype is missing?

### Programming

1. Write a function that skips three lines when it is called.
2. Write a function `drawParallel` that draws parallel lines and a function `drawRectangle` that uses `drawParallel` and `drawBase` to draw a rectangle.
3. Write a complete program for the problem described in Self-Check Exercise 2.
4. Rewrite the miles-to-kilometers conversion program shown in Figure 2.1, so that it includes a function that displays instructions to its user.
5. Show the revised program that calls function `instruct` for the circle area and circumference problem.

6. Write a main method that solves Self-Check Exercise 2 for Section 3.3. Assume you have functions `drawRectangle` (see Programming Exercise 2), `drawTriangle`, and `drawIntersect`.

### 3.5 Functions with Input Arguments

Programmers use functions like building blocks to construct large programs. Functions are more like Lego blocks (Figure 3.7) than the smooth-sided wooden blocks you might have used as a young child to demonstrate your potential as a budding architect. Your first blocks were big and did not link together, so buildings over a certain size would topple over. Legos, in contrast, have one surface with little protrusions and one surface with little cups. By placing the protrusions into the cups, you could build rather elaborate structures.

What does this have to do with programming? Simple functions such as `drawCircle` and `instruct` are like wooden blocks. They can display information on the screen, but they are not particularly useful. To be able to construct more interesting programs, we must provide functions with “protrusions” and “cups” so they can be easily interconnected.

The arguments of a function are used to carry information into the function subprogram from the main function (or from another function subprogram) or to return multiple results computed by a function subprogram. Arguments that carry information into the function subprogram are called **input arguments**; arguments that return results are called **output arguments**. We can also return a single result from a function by executing a `return` statement in the function body. We study functions with input arguments in this section and functions with output arguments in Chapter 6.

**input arguments**  
Arguments that pass  
information into a  
function.

**output arguments**  
Arguments that return  
results from a function.

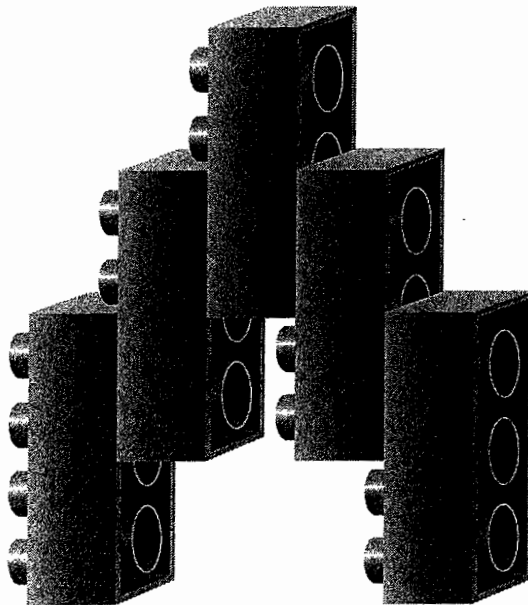


Figure 3.7 Lego blocks

The use of arguments is a very important concept in programming. Arguments make function subprograms more versatile because they enable a function to manipulate different data each time it is called. For example, in the statement

```
rimArea = findArea(edgeRadius) - findArea(holeRadius);
```

each call to function `findArea` calculates the area of a circle with a different radius.

### EXAMPLE 3.5

Function `drawCircleChar` (Listing 3.11) is an improved version of function `drawCircle` that enables the caller to specify the character drawn in the circle. When function `drawCircleChar` is called, the character that is its actual argument is passed into the function and is substituted for the formal parameter `symbol` (type `char`). The actual argument character is displayed wherever `symbol` appears in an output statement.

The function call

```
drawCircleChar('*');
```

draws the stick figure head shown earlier in Figure 3.4. Figure 3.8 shows the execution of this function call. You can use the function call

```
drawCircleChar('#');
```

to display the same circle shape, but with the character `#` displayed instead of `*`.

You must provide a prototype that declares `drawCircleChar` before function `main`. The prototype should indicate that `drawCircleChar` is type `void` (returns no result) and has an argument of type `char`. The prototype follows.

```
drawCircleChar(char);
```

You only need to specify the data type of the formal parameter, not its name, but you may specify the name if you like.

---

#### Listing 3.11 Function `drawCircleChar`

---

```
// Draws a circle using the character specified by symbol
void drawCircleChar(char symbol)
{
    cout << "    " << symbol << endl;
    cout << " " << symbol << " " << symbol << endl;
    cout << " " << symbol << " " << symbol << endl;
} // end drawCircle
```

---

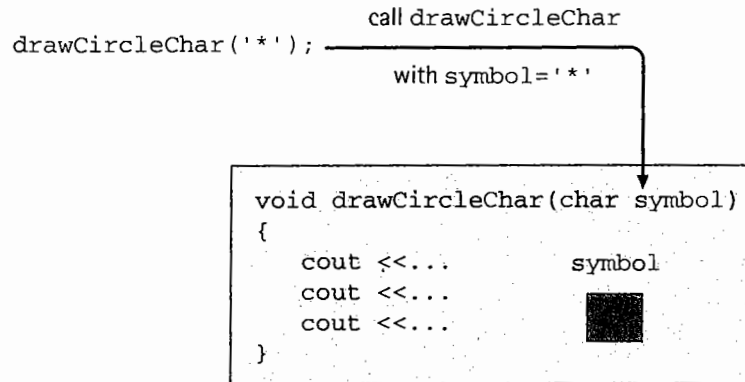


Figure 3.8 Execution of `drawCircleChar('*')`

## void Functions with Input Arguments

In the last section, we used void functions such as `instruct` and `drawCircle` to display several lines of program output. Recall that a void function does not return a result. The next example shows a new function, `drawCircleChar`, that has an input argument.

## Functions with Input Arguments and a Single Result

Next we show how to write functions with input arguments that return a single result, as diagrammed in Figure 3.9. We can reference these functions in expressions just like the library functions described in Section 3.2.

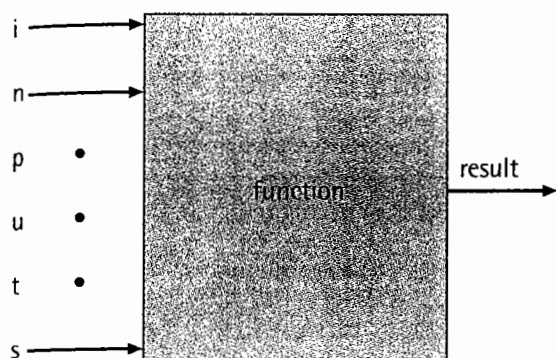
Let's reconsider the problem of finding the area and circumference of a circle using functions with just one argument. Section 3.2 described functions `findCircum` and `findArea`, each of which has a single input argument (a circle radius) and returns a single result (the circumference or area). Listing 3.12 shows these functions.

Each function heading begins with the word `float`, indicating that the function result is a real number. Both function bodies consist of a single return statement. When either function executes, the expression in its return statement is evaluated and returned as the function result. If `PI` is the constant 3.14159, calling function `findCircum` causes the expression `2.0 * 3.14159 * r` to be evaluated. To evaluate this expression, C++ substitutes the actual argument used in the function call for the formal parameter `r`.

For the function call below

```
radius = 10.0;
circum = findCircum(radius);
```

the actual argument, `radius`, has a value of 10.0, so the function result is 62.8318 (`2.0 * 3.14159 * 10.0`). The function result is assigned to `circum`. Figure 3.10 shows the execution of this function call.



**Figure 3.9** Function with input arguments and one result

**Listing 3.12** Functions `findCircum` and `findArea`

---

```

// Computes the circumference of a circle with radius r.
// Pre: r is defined and is > 0.
//      PI is a constant.
//      Library cmath is included.
float findCircum(float r)
{
    return (2.0 * PI * r);
}

// Computes the area of a circle with radius r.
// Pre: r is defined and is > 0.
//      PI is a constant.
//      Library cmath is included.
float findArea(float r)
{
    return (PI * pow(r, 2));
}

```

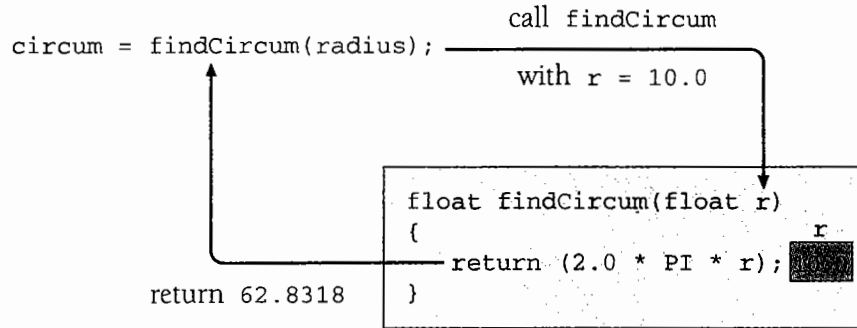
---

The function call to `findArea`

```
area = findArea(radius);
```

causes C++ to evaluate the expression `3.14159 * pow(r, 2)`, where `pow` is a library function (part of `cmath`) that raises its first argument to the power indicated by its second argument (`pow(r, 2)` computes  $r^2$ ). When `radius` is 10.0, `pow` returns 100.0 and `findArea` returns a result of 314.59, which is assigned to `area`. This example shows that a user-defined function can call a C++ library function.

You must provide prototypes that declare these functions before function `main`. The prototypes should indicate that each function is type `float`



**Figure 3.10** Execution of `circum = findCircum(radius);`

(returns a type float result) and has a formal parameter of type float. The prototypes follow.

```

float findArea(float);
float findCircum(float);
  
```

Like the function definition, the function prototype specifies the function type and its name. In the formal parameter list, you need to specify only the formal parameter types, not their names, but you can specify both. The function prototype, but not the heading, ends with a semicolon. Because function prototypes and function headings are so similar, we recommend that you write one of them first, and edit a copy of it (using your

#### Function Definition (Input Arguments and One Result)

**Syntax:** *// function interface comment*  
 ftype fname(formal-parameter-declaration-list)  
 {  
     local variable declarations  
     executable statements  
 }

**Example:** *// Finds the cube of its argument.*  
*// Pre: n is defined.*  
 int cube(int n)  
 {  
     return (n \* n \* n);  
 } *// end cube*

**Interpretation:** The function fname is defined. In the function heading, the identifier ftype specifies the data type of the function result. The formal-parameter-declaration-list, enclosed in parentheses, declares the type and name of each function parameter. Commas separate the parameters. Notice that there are no semicolons after the function heading. The braces enclose the function body. Any identifiers



that are declared in the optional local declarations are defined only during the execution of the function and can be referenced only within the function. The executable statements of the function body describe the data manipulation to be performed by the function in order to compute the result value. Execution of a **return** statement causes the function to return control to the statement that called it. The function returns the value of the expression following **return** as its result.

**Note:** `fctype` is `void` if the function does not return a value. The argument list `()` indicates that the function has no arguments.

### Function Prototype (With Parameters)

**Form:** `fctype fname(formal-parameter-type-list);`

**Example:** `int cube(int);`

**Interpretation:** The identifier `fname` is declared as the name of a function whose type is `fctype`. This declaration provides all the information that the C++ compiler needs to know to translate correctly all references to the function. The function definition will be provided after the `main` function. The `formal-parameter-type-list` enclosed in parentheses shows the data type of each formal parameter. Commas separate the data types. A semicolon terminates the prototype.

**Note:** C++ permits the specification of formal parameter names in function prototypes, as in

```
int cube(int n);
```

In this case, the heading of the function definition and the prototype should be identical except that the prototype ends with a semicolon.

word-processor) to create the other. Make sure you perform whatever editing operations are required—that is, remove (or add) a semicolon and remove (or add) the formal parameter names.

## PROGRAM STYLE      Function Interface Comments

The comments that begin each function in Listing 3.12 contain all the information required in order to use the function. The function interface comments begin with a statement of what the function does. Then the line

```
// Pre: r is defined.
```

describes the **precondition**—the condition that should be true before the function is called. You may also want to include a statement describing the **postcondition**—the condition that must be true after the function

### precondition

A condition assumed to be true before a function call.

### postcondition

A condition assumed to be true after a function executes.

completes execution, if some details of this postcondition are not included in the initial statement of the function's purpose.

We recommend that you begin all function definitions in this way. The function interface comments provide valuable documentation to other programmers who might want to reuse your functions in a new program.

## PROGRAM STYLE Problem Inputs versus Input Parameters

Make sure you understand the distinction between *problem inputs* and *input parameters*. *Problem inputs* are variables that receive their data from the program user through execution of an input statement. *Input parameters* receive their data through execution of a function call statement. Beginning programmers sometimes make the mistake of attempting to read a data value for a function's input parameter in the function body. The data passed to an input parameter must be defined *before* the function is called, not *after*.

### EXAMPLE 3.6

Function `scale` (Listing 3.13) multiplies its first argument (a real number) by 10 raised to the power indicated by its second argument (an integer). For example, the function call

```
scale(2.5, 2)
```

returns the value 250.0 ( $2.5 \times 10^2$ ). The function call

```
scale(2.5, -2)
```

returns the value 0.025 ( $2.5 \times 10^{-2}$ ).

In function `scale`, the statement

```
scaleFactor = pow(10, n);
```

calls function `pow` to raise 10 to the power specified by the second formal parameter `n`. Local variable `scaleFactor`, defined only during the execution of the function, stores this value. The `return` statement defines the function result as the product of the first formal parameter, `x`, and `scaleFactor`.

Listing 3.14 shows a very simple `main` function written to test function `scale`. The output statement calls function `scale` and displays the function result after it is returned. The arrows drawn in Listing 3.14 show the information flow between the two actual arguments and formal parameters. To clarify the information flow, we omitted the function interface comment. The argument list correspondence is shown below.

Actual Argument	corresponds to	Formal Parameter
num1		x
num2		n

## Functions with Multiple Arguments

Functions `findArea` and `findCircum` each have a single argument. We can also define functions with multiple arguments.

Listing 3.13 Function scale

---

```
// Multiplies its first argument by the power of 10
// specified by its second argument.
// Pre : x and n are defined and library cmath is
//      included.
float scale(float x, int n)
{
    float scaleFactor; // local variable
    scaleFactor = pow(10, n);
    return (x * scaleFactor);
}
```

---

Listing 3.14 Testing function scale

---

```
// File: testScale.cpp
// Tests function scale.

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

// Function prototype
float scale(float, int);

int main()
{
    float num1;
    int num2;
    // Get values for num1 and num2
    cout << "Enter a real number: ";
    cin >> num1;
    cout << "Enter an integer: ";
    cin >> num2;

    // Call scale and display result.
    cout << "Result of call to function scale is "
         << scale(num1, num2) // actual arguments
         << endl;
    return 0; // information flow
}

float scale(float x, int n) // formal parameters
```

---

(continued)

Listing 3.14 Testing function scale (continued)

---

```

{
    float scaleFactor; // local variable
    scaleFactor = pow(10, n);
    return (x * scaleFactor);
}

```

```
Enter a real number: 2.5
```

```
Enter an integer: -2
```

```
Result of call to function scale is 0.025
```

---

### Argument/Parameter List Correspondence

When using multiple-argument functions, you must be sure to include the correct number of arguments in the function call. Also, the order of the actual arguments used in the function call must correspond to the order of the formal parameters listed in the function prototype or heading.

Finally, if the function is to return meaningful results, assignment of each actual argument to the corresponding formal parameter must not cause any loss of information. Usually, you should use an actual argument of the same data type as the corresponding formal parameter, although this is not always essential. For example, the `<cmath>` library description indicates that both parameters of the function `pow` are of type `double`. Function `scale` calls `pow` with two actual arguments of type `int`. This call doesn't cause a problem because there is no loss of information when an `int` is assigned to a type `double` variable. If you pass an actual argument of type `float` or `double` to a formal parameter of type `int`, loss of the fractional part of the actual argument would likely lead to an unexpected function result. Next, we summarize these constraints on the **number**, **order**, and **type (not)** of input arguments.

### Argument/Parameter List Correspondence

- The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.
- The order of arguments in the lists determines correspondence. The first actual argument corresponds to the first formal parameter, the second actual argument corresponds to the second formal parameter, and so on.
- Each actual argument must be of a data type that can be assigned to the corresponding formal parameter with no unexpected loss of information.

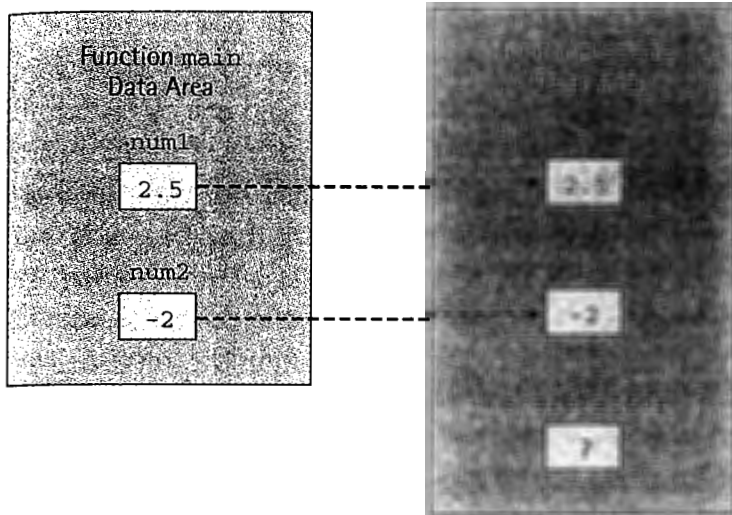


Figure 3.11 Data areas after call `scale(num1, num2);`

## The Function Data Area

Each time a function call is executed, an area of memory is allocated for storage of that function's data. Included in the function data area are storage cells for its formal parameters and any local variables that may be declared in the function. The function data area is always lost when the function terminates; it is re-created empty (all values undefined) when the function is called again.

Figure 3.11 shows the main function data area and the data area for function `scale` after the function call `scale(num1, num2)` executes. The values 2.5 and -2 are passed into the formal parameters `x` and `n`, respectively. The local variable, `scaleFactor`, is initially undefined; the execution of the function body changes the value of this variable to 0.01.

The local variable `scaleFactor` can be accessed only in function `scale`. Similarly, the variables `num1` and `num2` declared in function `main` can be accessed only in function `main`. If you want a function subprogram to use the value stored in `num1`, you must provide `num1` as an actual argument when you call the function.

## Testing Functions Using Drivers

A function is an independent program module, and as such it can be tested separately from the program that uses it. To run such a test, you should write a short **driver** function that defines the function arguments, calls the function, and displays the value returned. For example, the function `main` in Listing 3.14 acts as a driver to test function `scale`.

### driver

A short function written to test another function by defining its arguments, calling it, and displaying its result.

## EXERCISES FOR SECTION 3.5

## Self-Check

- Evaluate each of the following:
  - `scale(2.55 + 5.23, 4 / 2)`
  - `findCircum(3.0)`
  - `findArea(2.0)`
  - `scale(findArea(5.0), -1)`
- Explain the effect of reversing the function arguments in the call to `scale` shown in Example 3.6—that is, `scale(num2, num1)`.
- How does the use of function arguments make it possible to write larger, more useful programs?
- Write the function prototypes for functions `drawCircleChar`, `findCircum`, and `findArea`.
- What error message do you get if a function prototype has a different number of parameters than the function definition?

## Programming

- Revise the flat-washer program (Listing 3.4) to use function sub-programs `findArea`, `findRimArea`, `findUnitWeight`, and `instruct`. Show the complete program.
- Write a function that calculates the elapsed time in minutes between a start time and an end time expressed as integers on a 24-hour clock (8:30 P.M. = 2030). You need to deal only with end times occurring later on the same day as the start time. Also write a driver program to test your function.
- Assume your friend has solved Programming Exercise 2. Write another function that uses your friend's function to calculate the speed (km/h) one must average to reach a certain destination by a designated time. Function inputs include same-day departure and arrival times as integers on a 24-hour clock and the distance to the destination in kilometers. Also write a driver program to test your function.

## 3.6 Scope of Names

**scope of a name**  
The region in a program where a particular meaning of a name is visible or can be referenced.

The **scope of a name** refers to the region in a program where a particular meaning of a name is visible or can be referenced. For example, we stated earlier that variable `num1` (declared in `main` in Listing 3.14) can be referenced only in function `main`, which means that that is its scope. Let's consider the names in the program outline shown in Listing 3.15. The identifiers `MAX` and `LIMIT` are defined as constants and their scope begins at their definition and continues to the end of the source file. This means that all three functions can access the constants `MAX` and `LIMIT`.

The scope of the function subprogram name `funTwo` begins with its prototype and continues to the end of the source file. This means that function `funTwo` can be called by functions `one`, `main`, and itself. The situation is different for function `one` because `one` is also used as a formal parameter name in function `funTwo`. Therefore, function `one` can be called by the `main` function and itself but not by function `funTwo`. This shows how you can use the same name for different purposes in different functions—generally not a good idea because it is confusing to human readers of the program, but C++ has no problem handling it.

All of the formal parameters and local variables in Listing 3.15 are visible only from their declaration to the closing brace of the function in which they are declared. For example, from the line that is marked with the comment `// header 1` to the line marked `// end one`, the identifier `anArg` means an integer parameter in the data area of function `one`. From the line with the comment `// header 2` through the line marked `// end funTwo`, this name refers to a character parameter in the data area of `funTwo`. In the rest of the file, the name `anArg` is not visible.

**Listing 3.15** Outline of program for studying scope of names

---

```

void one(int anArg, double second); // prototype 1

int funTwo(int one, char anArg);    // prototype 2

const int MAX = 950;
const int LIMIT = 200;

int main()
{
    int localVar;
    . . .
} // end main

void one(int anArg, double second) // header 1
{
    int oneLocal;                  // local 1
    . . .
} // end one

int funTwo(int one, char anArg)    // header 2
{
    int localVar;                  // local 2
    . . .
} // end funTwo

```

---

Table 3.2 shows which identifiers are visible within each of the three functions

**Table 3.2** Scope of Names in Listing 3.15

Name	Visible in One	Visible in funTwo	Visible in main
MAX	yes	yes	yes
LIMIT	yes	yes	yes
main	yes	yes	yes
localVar (in main)	no	no	yes
one (the function)	yes	no	yes
anArg (int parameter)	yes	no	no
second	yes	no	no
oneLocal	yes	no	no
funTwo	yes	yes	yes
one (formal parameter)	no	yes	no
anArg (char parameter)	no	yes	no
localVar (in funTwo)	no	yes	no

## EXERCISES FOR SECTION 3.6

### Self-Check

1. Develop a table similar to Table 3.2 for the identifiers declared in Listing 3.14.
2. Develop a table similar to Table 3.2 for the program below.

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

const float PI = 3.14159;

// function prototypes
float findCircum(float);
float findArea(float);

int main()
{
    float radius;
    float circumf, area;
```



```

cout << "Enter radius: ";
cin >> radius;

circumf = findCircum(radius);
area = findArea(radius);

cout << "For a circle with radius " << radius
    << ", area is " << area
    << ", circumference is " << circumf << endl;

return 0;
}

float findCircum(float r)
{
    return (2.0 * PI * r);
}

float findArea(float r)
{
    return (PI * pow(r, 2));
}

```

## 3.7 Extending C++ through Classes: Using Class `string`

You have seen the advantages of procedural abstraction and the use of functions as program building blocks. C++ also facilitates **data abstraction** through its class feature, which enables users to define new data types and their associated operations. In this section we study the `string` class, which is part of the C++ standard library. In Chapter 10, we show how to define your own classes.

**data abstraction**  
Modeling real-world data using the data types available in a programming language or additional class types.

### The `string` Class

In Section 2.3 we introduced the C++ `string` class, which defines a new data type `string` and many operators for `string` objects. Two attributes of a `string` object are the character sequence it stores and the length of that sequence. Some of the common operators you have seen so far (`<<`, `>>`, `=`, `+`) can be used with `string` objects. Listing 3.16 illustrates how we use these familiar operators and some new ones with `string` objects as operands.

Listing 3.16 Illustrating string operations

---

```

// File: stringOperations.cpp
// Illustrates string operations

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

int main()
{
    string firstName, lastName; // inputs - first and
                                //          last names
    string wholeName;           // output - whole name
    string greeting = "Hello "; // output - a greeting
                                //          string

    // Read first and last names.
    cout << "Enter your first name: ";
    cin >> firstName;
    cout << "Enter your last name: ";
    cin >> lastName;

    // Join names in whole name
    wholeName = firstName + " " + lastName;

    // Display results
    cout << greeting << wholeName << "!" << endl;
    cout << "You have " << (wholeName.length() - 1)
         << " letters in your name." << endl;

    // Display initials
    cout << "Your initials are " << firstName.at(0)
         << lastName.at(0) << endl;

    return 0;
}

```

```

Enter your first name: Caryn
Enter your last name: Jackson
Hello Caryn Jackson!
You have 12 letters in your name.
Your initials are CJ

```

## Declaring `string` Objects

The declarations

```
string firstName, lastName; // inputs - first and last
                        //          names
string wholeName;          // output - whole name
```

allocate storage for three `string` objects that initially contain empty strings (a string with zero characters). You can store new data in these strings through a string assignment or by reading a string value typed at the keyboard. You can also store data in a `string` object when you declare it. The statement

```
string greeting = "Hello "; // output - a greeting string
```

stores the string "Hello" in the `string` object `greeting`.

## Reading and Displaying `string` Objects

In Section 2.4, we saw that the extraction operator can be used with `string` operands. The statement

```
cin >> firstName;
```

reads keyboard data into `firstName`. It stores in `firstName` all data characters up to (but not including) the first blank or return. You must not enclose these characters in quotes. The statement

```
cout << greeting << wholeName << '!' << endl;
```

displays `string` objects `greeting` and `wholeName` followed by the character value '!'.

What if you want to read a data string that contains a blank? For example, you may want to read the characters Van Winkle into `lastName`. You can do this using the `getline` function. For example, the statement

```
getline(cin, lastName, '\n');
```

reads into `string` object `lastName` all characters typed at the keyboard (stream `cin`) up to (but not including) the character `'\n'`. The escape sequence `'\n'` is a text control character (see Table 2.5) and it represents the *newline* character. You enter this character at the keyboard by pressing RETURN or ENTER. (On some systems, you may need to press ENTER twice.)

You can specify another terminator character by changing the third argument. For example, use '\*' as the third argument to make the symbol \* the terminator character. In this case, you can type your data string over multiple lines, and it will not terminate until you enter the symbol \*. All data characters entered, including any newline characters (but not the \*), will be stored in the `string` object used as the second argument.

## String Assignment and Concatenation

The assignment statement

```
wholeName = firstName + " " + lastName;
```

stores in `wholeName` the new string formed by joining the three strings that are operands of the operator `+`, which means to join, or *concatenate*, when its operands are strings. Notice that we insert a space between string objects `firstName` and `lastName` by joining the string value `" "`, not the character value `' '`. Remember to use quotes, not apostrophes, to enclose string values in C++ statements.

## Operator Overloading

Until now, the operator `+` has always meant addition, but now we see that it means concatenation when its operands are `string` objects. We use the term **operator overloading** to indicate that an operator's function may vary depending on its operands. Operator overloading is a very powerful concept because it allows C++ to give multiple meanings to a single operator; C++ can determine the correct interpretation of the operator from the way we use it.

**operator overloading**  
The ability of an operator to perform different operations depending on the data type of its operands.

## Dot Notation: Calling Functions **length** and **at**

Listing 3.16 also uses member functions `length` and `at` from the `string` class. To call these functions (and most member functions), we use *dot notation*. Rather than passing an object as an argument to a member function, we write the name of the object, a dot (or period), and then the function to be applied to this object. The function reference

```
wholeName.length()
```

applies member function `length` (no arguments) to string object `wholeName`. This call to function `length` returns a value of 13 (`wholeName` contains the 13-character string "Caryn Jackson"); however, we subtract one

before printing the result because of the blank character between the first and last names:

```
cout << "You have " << (wholeName.length() - 1)
    << " letters in your name." << endl;
```

The expression

```
firstName.at(0)
```

retrieves the character in `firstName` that is at position 0, where we number the characters from the left, starting with 0 for the first character, 1 for the second, and so on. By this numbering system, the last character in string object `wholeName` (the letter `n`) is at position `wholeName.length() - 1`, or position 12.

#### Dot Notation

**Syntax:** `object.function-call`

**Example:** `firstName.at(0)`

**Interpretation:** The member function specified in the function-call is applied to object.

**Note:** The member function may change the attributes of object.

## Member Functions for Word-Processing Operations

We would like to be able to perform on a `string` object all the operations that are available on most word processors. C++ provides member functions for searching for a string (`find`), inserting a string at a particular position (`insert`), deleting a portion of a string (`erase`), and replacing part of a string with another string (`replace`). We illustrate some simple examples of these functions next.

For the strings `firstName`, `lastName`, and `wholeName` in Listing 3.16, the expression

```
wholeName.find(firstName)
```

returns 0 (the position of the first character of "Caryn" in "Caryn Jackson") and the expression

```
wholeName.find(lastName)
```

returns 6 (the position of the first character of "Jackson" in "Caryn Jackson").

The first statement below inserts a string at the beginning of `string` object `wholeName` (at position 0) and the second statement inserts a string in the middle of `wholeName` (at new position 10).

```
wholeName.insert(0, "Ms. ");           // Change to
                                         // "Ms. Caryn Jackson"
wholeName.insert(10, "Heather ");      // Change to
                                         // "Ms. Caryn Heather
                                         // Jackson"
```

Notice that member function `insert` changes the string stored in object `wholeName`.

You can use the following statement to change the middle name from Heather to Amy:

```
wholeName.replace(10, 7, "Amy");       // Change to
                                         // "Ms. Caryn Amy Jackson"
```

This statement means: start at position 10 of `wholeName` (at the letter H) and replace the next seven characters (Heather) with the string "Amy". Finally, you can delete the new middle name altogether by using the statement

```
wholeName.erase(10, 4);               // Change back to
                                         // "Ms. Caryn Jackson"
```

which means: start at position 10 and erase four characters (Amy and a space) from `string` object `wholeName`.

## Assigning a Substring to a **string** Object

You can also store a *substring* (portion of a string) in another `string` object using member function `assign`. For example, if `title` is a `string` object, the statement

```
title.assign(wholeName, 0, 3);         // Store "Ms." in title
```

stores in `title` the first three characters of `wholeName`. The content of `wholeName` is not changed.

Table 3.3 summarizes the member functions described in this section.

**Table 3.3** Some Member Functions for string Operations

Function	Purpose
<code>getline(cin, aString, '\n')</code>	Extracts data characters up to (but not including) the first newline character from stream <code>cin</code> and stores them in <code>aString</code> . The first newline character is extracted from <code>cin</code> but not stored in <code>aString</code> .
<code>aString.length()</code>	Returns the count of characters (an integer) in <code>aString</code> .
<code>aString.at(i)</code>	Returns the character in position <code>i</code> of <code>aString</code> where the leftmost character is at position 0 and the rightmost character is at position <code>aString.length() - 1</code> .
<code>aString.find(target)</code>	Returns the starting position (an integer) of string <code>target</code> in <code>aString</code> . If <code>target</code> is not in <code>aString</code> , returns a value that is outside the range of valid positions; that is, it returns a value $< 0$ or a value $\geq$ the length of <code>aString</code> .
<code>aString.insert(start, newString)</code>	Inserts <code>newString</code> at position <code>start</code> of <code>aString</code> .
<code>aString.replace(start, count, newString)</code>	Starting at position <code>start</code> of <code>aString</code> , replaces the next <code>count</code> characters with <code>newString</code> .
<code>aString.erase(start, count)</code>	Starting at position <code>start</code> of <code>aString</code> , removes the next <code>count</code> characters.
<code>aString.assign(oldString, start, count)</code>	Starting at position <code>start</code> of <code>oldString</code> , assigns to <code>aString</code> the next <code>count</code> characters.

**EXERCISES FOR SECTION 3.7****Self-Check**

1. Explain the effect of each of the following; assume a data line containing the characters `Jones***John Boy` is typed in.

```
String name;
cout << "name: ";
getline(cin, name, '\n');
int start = name.find("***");
name.erase(start, 3);
name.insert(start, ", ");
cout << name << endl;
```

2. Trace each of the following statements:

```
string author;
author = "John";
author = author + " Steinbeck";
cout << author.length() << endl;
cout << author.at(0) << author.at(4) << endl;
```

```

cout << author.at(author.length() - 1)
      << endl;
cout << author.find("n");
cout << author.find("nb");
author.insert(4, "ny");
author.replace(0, 6, "Jonathan");
author.erase(3, 5);

```

3. Write a statement that stores in `wholeName` the strings stored in `firstName` and `lastName` in the form *lastName, firstName*.
4. What happens if the first argument for `insert` is larger than the number of characters in the string? What happens if it is negative?

#### Programming

1. Write a program that reads three strings and displays the strings in all possible sequences, one sequence per output line. Display the symbol \* between the strings on each line.

## 3.8 Introduction to Computer Graphics (Optional)

### text mode

A display mode in which a C++ program displays only characters.

### graphics mode

A display mode in which a C++ program draws graphics patterns and shapes in an output window.

In normal computer display mode (called **text mode**), we use `cout` to display lines of characters to the standard output device, or console. In Section 3.3, we showed how to write C++ functions for drawing simple stick figures using text mode. In several optional sections beginning with this one, we discuss another display mode (called **graphics mode**) that enables a C++ program to draw pictures or graphical patterns in an output window. To write graphics programs, you must learn how to use special graphics functions that enable you to draw lines and various geometric shapes (for example, rectangles, circles, ellipses) anywhere on your screen and color and shade them in different ways.

Several programming languages include graphics libraries. Although there is no standard graphics library for C++, several libraries have been developed for use with C++. We will study a simple graphics library called `WinBGIm` which is based on Borland Turbo Pascal Graphics, with the addition of mouse control. This library was developed by Professor Michael Main of the Computer Science Department, University of Colorado, Boulder. The principles we study will apply to other graphics libraries you may use in the future.

### Composition of a Window

In text mode, you don't pay much attention to the position of each line of characters displayed on the screen. In graphics programming, you control



the location of each line or shape that you draw in a window. Consequently, you must know your window size and how to reference the individual picture elements (called **pixels**) in a window.

You can visualize a window as an  $x$ - $y$  grid of pixels. Assume that your window has the dimensions  $400 \times 300$ . Figure 3.12 shows the coordinates for the four pixels at the corners. The pixel at the top-left corner has  $x$ - $y$  coordinates  $(0, 0)$ , and the pixel at the top-right corner has  $x$ - $y$  coordinates  $(400, 0)$ .

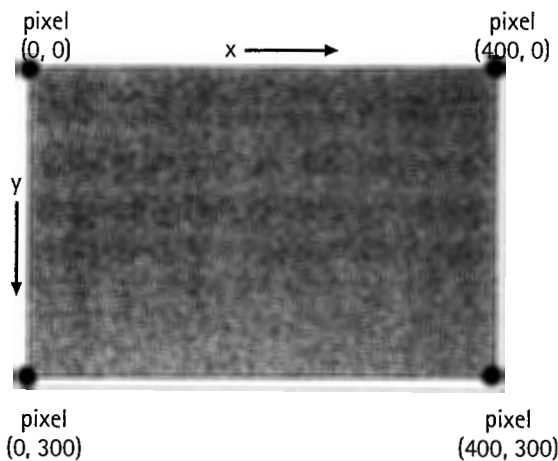
Notice the pixels in the  $y$ -direction are numbered differently from how we are accustomed. The pixel  $(0, 0)$  is at the top-left corner of the screen, and the  $y$ -coordinate values increase as we move down the screen. In a normal  $x$ - $y$  coordinate system, the point  $(0, 0)$  is at the bottom-left corner.

pixel  
A picture element on a  
computer screen

## Some Common Graphics Functions

A graphics program is a sequence of statements that call graphics functions to do the work. Listing 3.17 is a program that uses several key functions. The functions `getmaxwidth` and `getmaxheight` return the position of the last pixel in the  $X$  and  $Y$ -directions on your computer screen. Some typical screen dimensions are  $640 \times 480$  for 14-inch screens and  $1024 \times 768$  for 15- or 17-inch screens, but the largest window will be slightly smaller. Therefore, the statements

```
bigx = getmaxwidth(); // get largest x-coordinate.
bigy = getmaxheight(); // get largest y-coordinate.
initwindow(bigx, bigy,
    "Full screen window - press a character to close window");
```



**Figure 3.12** Referencing pixels in a window

**Listing 3.17** Drawing intersecting lines

---

```

// Draws intersecting lines
// File: lines.cpp

#include <graphics.h>
#include <iostream>

using namespace std;

int main()
{
    int bigX;                // largest x-coordinate
    int bigY;                // largest y-coordinate

    bigX = getmaxwidth();    // get largest x-coordinate
    bigY = getmaxheight();   // get largest y-coordinate
    initwindow(bigX, bigY,
               "Full screen window - press a key to close");

    // Draw intersecting lines
    line(0, 0, bigX, bigY);  // Draw white line from (0, 0) to (bigX, bigY)
    setcolor(LIGHTGRAY);    // Change color to gray
    line(bigX, 0, 0, bigY);  // Draw gray line from (bigX, 0) to (0, bigY)

    // Close screen when ready
    getch();                 // pause until user presses a key
    closegraph();            // close the window

    // Display window size in console
    cout << "Window size is " << bigX << " X " << bigY << endl;

    return 0;
}

```

Window size is 1018 X 736

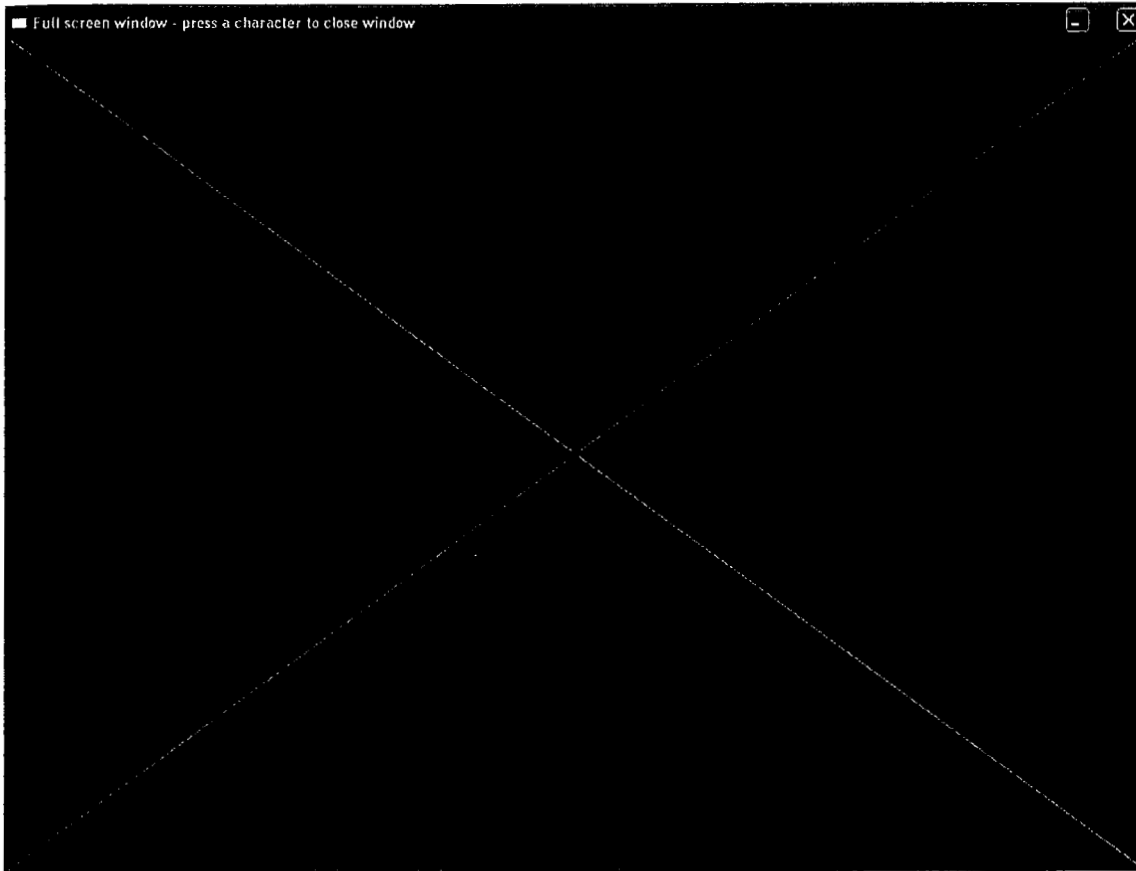
pop up a window of maximum width and height with the third argument as the window label (see Figure 3.13). The window position is set by the optional fourth and fifth arguments. If they are missing, the top-left corner of the window is at (0, 0).

In the statements

```

line(0, 0, bigx, bigy); // Draw white line from(0, 0) to
                        // (bigx, bigy)
setcolor(LIGHTGRAY);   // Change color to gray
line(bigx, 0, 0, bigy); // Draw gray line from (bigx, 0)
                        // to (0, bigy)


```



**Figure 3.13** Window drawn by `lines.cpp`

the two calls to function `line` draw a pair of intersecting lines in this window. Method `setcolor` changes the drawing color from white (the default color) to gray for the second line. Then, the statements

```
getch();           // pause until user presses a character
closegraph();      // close the window
```

call the `getch` function (also part of `graphics.h`) to pause the program until the user presses a key. Once a key is pressed, or the user clicks on the close icon on the top right , the `closegraph` function closes the window. Finally, the statement beginning with `cout` displays the window size in the console window for normal text output.

## Background Color and Foreground Color

In graphics mode, the computer displays all pixels continuously in one of 16 colors. The default color used to display a pixel is called the background color. Consequently, your initial display window appears empty because all its pixels

**background color**  
The default color for all of the pixels in a display window.

**foreground color**  
The new color of pixels that are part of a graphics object in a display window.

are displayed in the **background color**. When you draw a line or a shape, the pixels it contains stand out because they are changed to the **foreground color**.

Black and white are the default values for the background and foreground colors, respectively. The statements

```
setbkcolor(GREEN); // GREEN is the background color.
setcolor(RED)      // RED is the foreground color.
```

reset the background color to GREEN and the foreground color to RED where GREEN and RED are color constants defined in `graphics.h`. You select a color constant from the list shown in Table 3.4, using either the constant name or its numeric value as a parameter (for example, RED or 4). Once you change the foreground or background color, it retains its new value until you change it again.

## Drawing Rectangles

We use function `rectangle` to draw a rectangle in a window. The statement

```
rectangle(x1, y1, x2, y2);
```

draws a rectangle that has one diagonal with end points  $(x1, y1)$  and  $(x2, y2)$ .

### EXAMPLE 3.7

The program in Listing 3.18 draws a house (Figure 3.14). The program begins by defining the corner points of the house, where the roof is a pair of lines intersecting at point  $(x2, y2)$ . The first call to `rectangle` draws the rest of the house, and the second call to `rectangle` draws the door. We drew the house in a window of size  $640 \times 500$ , with the top-left corner of the window at point  $(100, 50)$  of the screen.

**Table 3.4** Color Constants

Constant	Value	Constant	Value
BLACK	0	DARKGRAY	8
BLUE	1	LIGHTBLUE	9
GREEN	2	LIGHTGREEN	10
CYAN	3	LIGHTCYAN	11
RED	4	LIGHTRED	12
MAGENTA	5	LIGHTMAGENTA	13
BROWN	6	YELLOW	14
LIGHTGRAY	7	WHITE	15

**Listing 3.18** Drawing a house

---

```

// File: house.cpp
// Draws a house

#include <graphics.h>

int main()
{
    // Define corners of house
    int x1 = 100; int y1 = 200;           // top-left corner
    int x2 = 300; int y2 = 100;           // roof peak
    int x3 = 500; int y3 = 200;           // top-right corner
    int x4 = 500; int y4 = 400;           // bottom-right corner
    int x5 = 325; int y5 = 400;           // bottom-right corner of door
    int x6 = 275; int y6 = 325;           // top-left corner of door

    initwindow(640, 500, "House - press a key to close", 100, 50);

    // Draw roof
    line(x1, y1, x2, y2);                  // Draw line from (x1, y1) to (x2,y2)
    line(x2, y2, x3, y3);                  // Draw line from (x2, y2) to (x3,y3)

    // Draw rest of house
    rectangle(x1, y1, x4, y4);

    // Draw door
    rectangle(x5, y5, x6, y6);

    getch();                               // pause until user presses a key
    closegraph();                           // close the window
    return 0;
}

```

---

## Drawing Circles, Ellipses, and Arcs

We use function `circle` to draw a circle. The function call statement

```
circle(x, y, radius);
```

draws a circle whose center is at  $(x, y)$ . The third parameter is the circle radius.

---

Listing 3.19 Program to draw a happy face

---

```

// File: happyFace.cpp
// Draws a happy face

#include <graphics.h>

int main()
{
    int midX, midY;                // coordinates of center point
    int leftEyeX, rightEyeX, eyeY; // eye center points
    int noseX, noseY;             // nose center point
    int headRadius;               // head radius
    int eyeNoseRadius;            // eye/nose radius
    int smileRadius;              // smile radius
    int stepX, stepY;             // x and y increments

    initwindow(500, 400, "Happy Face - press key to close");

    // Draw head
    midX = getmaxx() / 2;          // center head in x-direction
    midY = getmaxy() / 2;          // center head in y-direction
    headRadius = getmaxy() / 4;    // head will fill half the window
    circle(midX, midY, headRadius); // draw head.

    // Draw eyes
    stepX = headRadius / 4;        // x-offset for eyes
    stepY = stepX;                 // y-offset for eyes and nose
    leftEyeX = midX - stepX;        // x-coordinate for left eye
    rightEyeX = midX + stepX;       // x-coordinate for right eye
    eyeY = midY - stepY;           // y-coordinate for both eyes
    eyeNoseRadius = headRadius / 10;
    circle(leftEyeX, eyeY, eyeNoseRadius); // draw left eye
    circle(rightEyeX, eyeY, eyeNoseRadius); // draw right eye

    // Draw nose
    noseX = midX;                  // nose is centered in x direction
    noseY = midY + stepY;
    circle(noseX, noseY, eyeNoseRadius);

    // Draw smile
    smileRadius = int(0.75 * headRadius + 0.5); // 3/4 of head radius
    arc(midX, midY, 210, 330, smileRadius);

    getch();
    closegraph();
    return 0;
}

```

---

## PROGRAM STYLE Writing General Graphics Programs

The program in Listing 3.19 is general and bases the happy face position and dimensions on the window dimensions as determined by `getmaxx` and `getmaxy`. If you change the window dimensions, the happy face will expand or shrink to fit. Conversely, the size of the house drawn by the program in Listing 3.18 is fixed and is independent of the window size. If the window is too small, part of the house may be missing. It is generally easier to draw figures with fixed dimensions; however, with a little practice you should be able to draw general graphics figures. To generalize the program in Listing 3.18, you could base the coordinates of the house corners on the window dimensions as follows (see Self-Check Exercise 4).

```
x1 = getmaxx() / 4; y1 = getmaxy() / 2; // top-left corner
x2 = getmaxx() / 2; y2 = getmaxy() / 4; // roof peak
```

### Drawing Filled Figures

So far, all our graphics figures have been line drawings. To fill in sections of the screen using different colors and patterns, you would use function `setfillstyle` to specify the pattern and color. The function call statement

```
setfillstyle(SLASH_FILL, RED);
```

sets the fill pattern to red slashes until you change it through another call to `setfillstyle`. Table 3.5 shows the options for the first parameter in a call to `setfillstyle`. Use either the constant name or its value (for example, `SLASH_FILL` or 4) and any color constant in Table 3.4 as the second parameter.

Use function `floodfill` to actually fill in a portion of a diagram. The function call statement

```
floodfill(x, y, WHITE);
```

**Table 3.5** Fill Pattern Constants

Constant	Value	Fill Pattern	Constant	Value	Fill Pattern
<code>EMPTY_FILL</code>	0	Background color	<code>LTBKSLASH_FILL</code>	6	\\ (light)
<code>SOLID_FILL</code>	1	Solid color	<code>HATCH_FILL</code>	7	Hatch (light)
<code>LINE_FILL</code>	2	---	<code>XHATCH_FILL</code>	8	Crosshatch
<code>LTSLASH_FILL</code>	3	/// (light)	<code>INTERLEAVE_FILL</code>	9	Interleaving line
<code>SLASH_FILL</code>	4	/// (heavy)	<code>WIDE_DOT_FILL</code>	10	Dots (light)
<code>BKSLASH_FILL</code>	5	\\\ (heavy)	<code>CLOSE_DOT_FILL</code>	11	Dots (heavy)

fills with the current fill pattern an area on the screen that contains the point  $(x, y)$  and is bounded by white lines. If the point  $(x, y)$  is outside an area bounded by white lines, the exterior of the bounded figure is filled.

The `floodfill` function sometimes gives unexpected results and is a bit inefficient. As an alternative to drawing a rectangle and then filling it, function `bar` draws a filled rectangle. The function call statement

```
setfillstyle(SOLID_FILL, BLUE);
bar(x1, y1, x2, y2);
```

draws a filled rectangle that has a diagonal with end points  $(x1, y1)$  and  $(x2, y2)$ .

### EXAMPLE 3.8

We can insert the following program fragment at the end of the program in Listing 3.18 to paint the house (Figure 3.16).

```
// Paint the house
setfillstyle(HATCH_FILL, LIGHTGRAY);
floodfill(x2, y2 + 10, WHITE);    // Paint the roof
setfillstyle(LINE_FILL, WHITE);
floodfill(x2, y1 + 10, WHITE);    // Paint the house front
```

In the two calls to `floodfill`, we use `x2`, the midpoint in the X-direction, as the first parameter. The second parameter (the Y-coordinate) determines which section (roof or house front) to fill in. All boundary lines for the house are white (the default foreground color), so the third parameter is `WHITE`. The roof is painted in a gray hatch pattern, the house itself is painted in a white lined pattern.

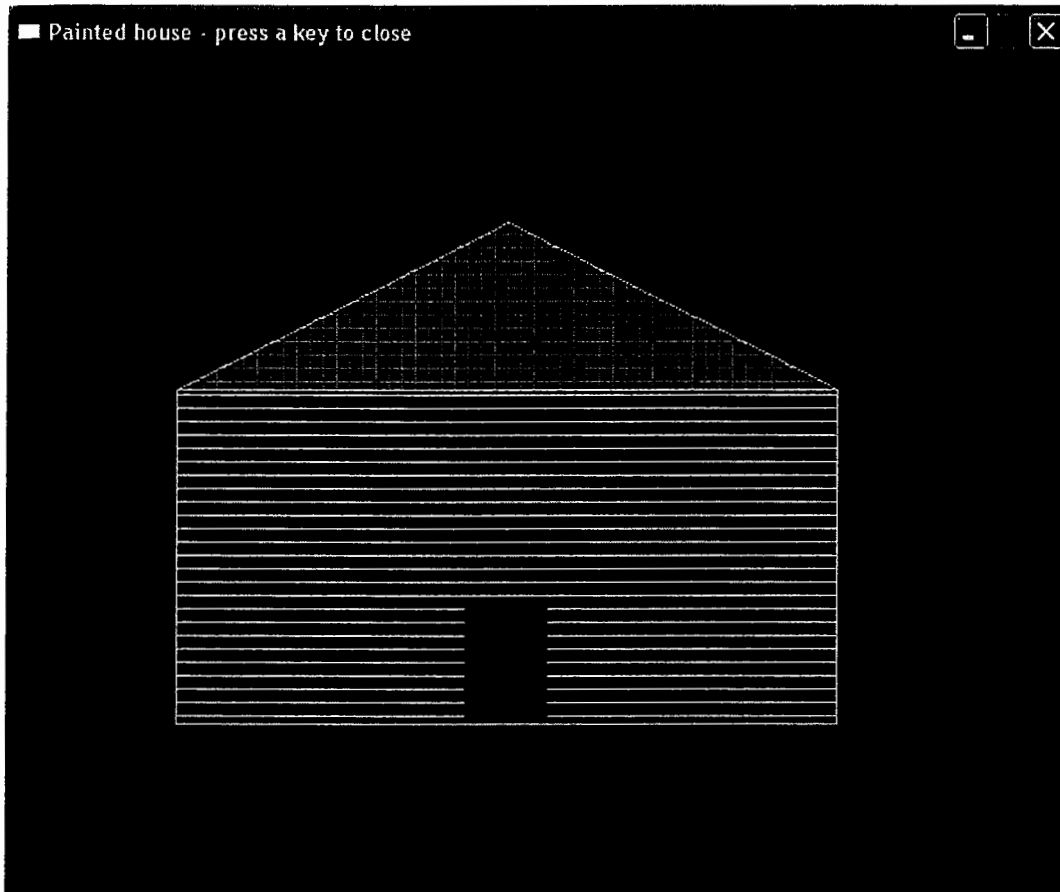
The call to function `bar` below paints the door blue. It replaces the earlier call to function `rectangle` with the same arguments. Listing 3.20 shows the complete program.

```
setfillstyle(SOLID_FILL, BLUE);
bar(x5, y5, x6, y6);    // Draw blue door
```

### PROGRAM PITFALL Incorrect Order of Function Calls Draws Over Earlier Results

The order of statement execution is critical in all programs, but an incorrect order can cause strange results in graphics program. If you call function `bar` to paint the door blue before calling `floodfill` to paint the house front, `floodfill` will change the color of the pixels in the door to white, and it will not appear in the window.





**Figure 3.16** Painted house drawn by `paintedHouse.cpp`

**Listing 3.20** Program to paint a house

```
// File: paintedHouse.cpp
// Paints a house

#include <graphics.h>

int main()
{
    // Define corners of house
    int x1 = 100; int y1 = 200;           // top-left corner
    int x2 = 300; int y2 = 100;           // roof peak
    int x3 = 500; int y3 = 200;           // top-right corner
    int x4 = 500; int y4 = 400;           // bottom-right corner
    int x5 = 325; int y5 = 400;           // bottom-right corner of door
    int x6 = 275; int y6 = 325;           // top-left corner of door

    initwindow(640, 500, "House - press a key to close", 100, 50);
```

(continued)

**Listing 3.20** Program to paint a house (continued)

---

```

    // Draw roof
    line(x1, y1, x2, y2);           // Draw line from (x1, y1) to (x2, y2)
    line(x2, y2, x3, y3);           // Draw line from (x2, y2) to (x3, y3)

    // Draw rest of house
    rectangle(x1, y1, x4, y4);

    // Paint the house
    setfillstyle(HATCH_FILL, LIGHTGRAY);
    floodfill(x2, y2 + 10, WHITE);    // Paint the roof
    setfillstyle(LINE_FILL, WHITE);
    floodfill(x2, y1 + 10, WHITE);    // Paint the house

    setfillstyle(SOLID_FILL, BLUE);
    bar(x5, y5, x6, y6);              // Draw blue door

    getch();
    closegraph();
    return 0;
}

```

---

## Pie Slices and Filled Ellipses

Function `pieslice` draws a filled pie slice (section of a circle) and `filledellipse` draws a filled ellipse or circle. Insert the first statement below at the end of Listing 3.19 to change the happy face smile to a frown (replaces the original call to function `arc`). The last statement calls `pieslice` to draw an eyepatch over the pirate's right eye (Figure 3.17).

```

    arc(midX, midY + headRadius, 65, 115, smileRadius / 2);
                                     // Draw frown
    setfillstyle(CLOSE_DOT_FILL, WHITE);
    pieslice(midX, midY, 10, 60, smileRadius);
                                     // Draw eye patch

```

The eye patch is a pie slice with radius `smileRadius` centered at point `(midX, midY)`. The pie slice goes from 10 degrees to 60 degrees. The final program is shown in Listing 3.21.

You can use the `pieslice` and `bar` functions to draw pie charts and bar graphs (see Programming Project 13 at the end of the chapter). We describe how to display the message shown under the happy face next.

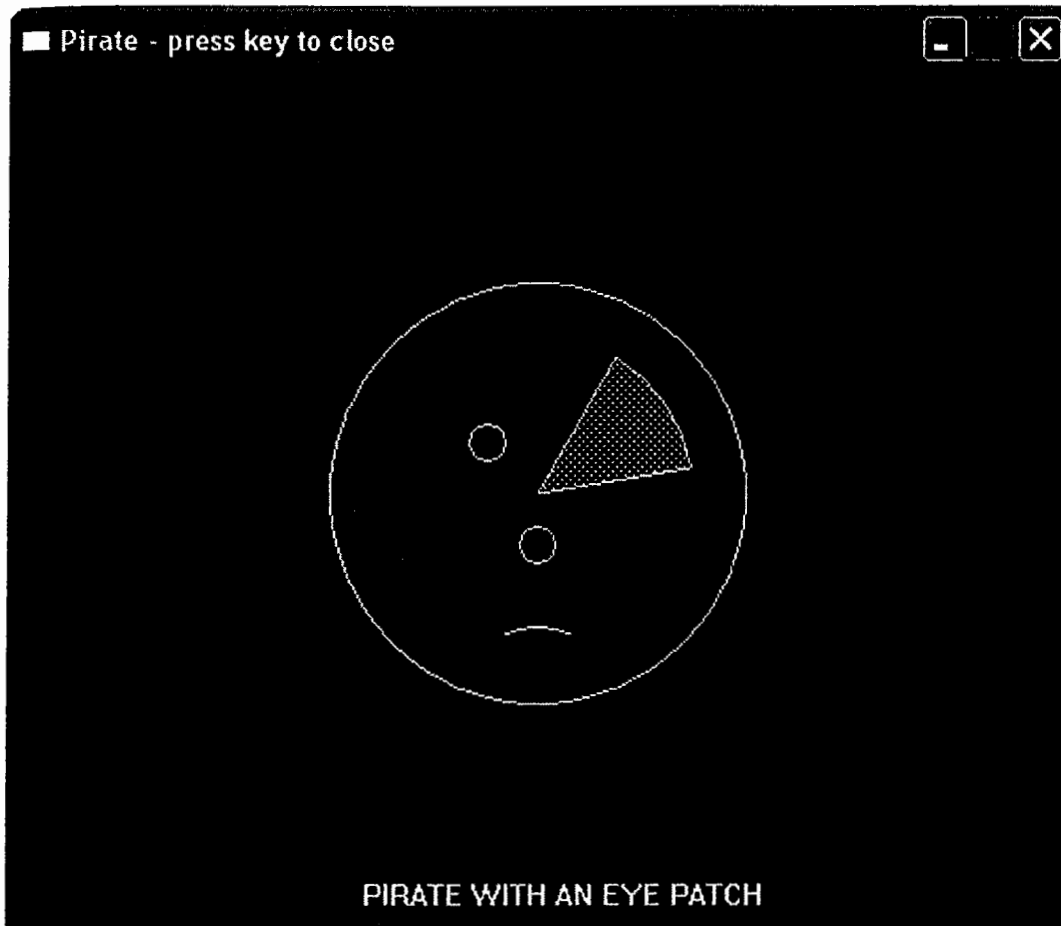


Figure 3.17 Pirate drawn by pirate.cpp

Listing 3.21 Program to draw a pirate

```
// File: pirate.cpp
// Draws a pirate

#include <graphics.h>

int main()
{
    int midX, midY;                // coordinates of center point
    int leftEyeX, rightEyeX, eyeY; // eye center points
    int noseX, noseY;              // nose center point
    int headRadius;                // head radius
    int eyeNoseRadius;             // eye/nose radius
    int smileRadius;               // smile radius
    int stepX, stepY;              // x and y increments
```

(continued)

Listing 3.21 Program to draw a pirate (continued)

---

```

initwindow(500, 400, "Pirate - press key to close", 200, 150);

// Draw head
midX = getmaxx() / 2;           // center head in x-direction
midY = getmaxy() / 2;           // center head in y-direction
headRadius = getmaxy() / 4;     // head will fill half the window
circle(midX, midY, headRadius); // draw head.

// Draw eyes
stepX = headRadius / 4;         // x-offset for eyes
stepY = stepX;                  // y-offset for eyes and nose
leftEyeX = midX - stepX;        // x-coordinate for left eye
rightEyeX = midX + stepX;       // x-coordinate for right eye
eyeY = midY - stepY;            // y-coordinate for both eyes
eyeNoseRadius = headRadius / 10;
circle(leftEyeX, eyeY, eyeNoseRadius); // draw left eye
circle(rightEyeX, eyeY, eyeNoseRadius); // draw right eye

// Draw nose
noseX = midX;                   // nose is centered in x direction
noseY = midY + stepY;
circle(noseX, noseY, eyeNoseRadius);

// Draw frown
smileRadius = int(0.75 * headRadius + 0.5); // 3/4 of head radius
arc(midX, midY + headRadius, 65, 115, smileRadius / 2); // Draw frown

setfillstyle(CLOSE_DOT_FILL, WHITE);
pieslice(midX, midY, 10, 60, smileRadius); // Draw eye patch

outtextxy(getmaxx() / 3, getmaxy() - 20, "PIRATE WITH AN EYE PATCH");

getch();
closegraph();
return 0;
}

```

---

### Adding Text to Drawings

In graphics mode, you cannot use `cout` to display characters, but you must instead draw characters just as you draw other shapes. Fortunately, the graphics library provides a function that does this. The function call statement

```

outtextxy(getmaxx() / 3, getmaxy() - 20,
          "PIRATE WITH AN EYE PATCH");

```

draws each character in a string (its third parameter) starting at the pixel whose X-Y coordinates are specified by its first two parameters (see Figure 3.17). You may see a warning message associated with `outtextxy` during program compilation. You can ignore it for now; we explain what it means in Section 9.12.

Table 3.6 shows the functions in the graphics library. With the exception of `label` (in `init`) and `textString` (in `outtextxy`), all arguments are type

**Table 3.6** Functions in Graphics Library

Function	Effect
<code>arc(x, y, stAng, endAng, r)</code>	draws an arc from angle <code>stAng</code> to <code>endAng</code> with center at <code>(x, y)</code> and radius <code>r</code>
<code>bar(x1, y1, x2, y2)</code>	draws a filled rectangle with a diagonal through points <code>(x1, y1)</code> and <code>(x2, y2)</code>
<code>circle(x, y, r)</code>	draws a circle with center at <code>(x, y)</code> and radius <code>r</code>
<code>closegraph()</code>	closes graphics mode
<code>ellipse(x, y, stAng, endAng, xRad, yRad)</code>	Draws an ellipse with center at <code>(x, y)</code> from <code>stAng</code> to <code>endAng</code> with <code>xRad</code> as horizontal radius and <code>yRad</code> as vertical radius
<code>fillellipse(x, y, xRad, yRad)</code>	Draws a filled ellipse with center at <code>(x, y)</code> with <code>xRad</code> as horizontal radius and <code>yRad</code> as vertical radius
<code>floodfill(x, y, border)</code>	fills with the current fill pattern the figure containing the point <code>(x, y)</code> and bounded by lines with color <code>border</code>
<code>getch()</code>	pauses the program until the user enters a character
<code>getmaxheight()</code>	returns the position of the last pixel in the y-direction in the screen
<code>getmaxwidth()</code>	returns the position of the last pixel in the x-direction in the screen
<code>getmaxx()</code>	returns the window width in pixels
<code>getmaxy()</code>	returns the window height in pixels
<code>initgraph(x, y, label)</code>	displays a window <code>x</code> pixels wide and <code>y</code> pixels high with the given <code>label</code> and top-left corner at <code>(0, 0)</code>
<code>initgraph(x, y, label, x0, y0)</code>	displays a window <code>x</code> pixels wide and <code>y</code> pixels high with the given <code>label</code> and top-left corner at <code>(x0, y0)</code>
<code>line(x1, y1, x2, y2)</code>	draws a line with end points <code>(x1, y1)</code> and <code>(x2, y2)</code>
<code>outtextxy(x, y, textString)</code>	draws the characters for <code>textString</code> starting at point <code>(x, y)</code>
<code>pieslice(x, y, stAng, endAng, r)</code>	draws a filled pie slice with center at <code>(x, y)</code> from angle <code>stAng</code> to <code>endAng</code> with radius <code>r</code>
<code>rectangle(x1, y1, x2, y2)</code>	draws a rectangle with a diagonal through points <code>(x1, y1)</code> and <code>(x2, y2)</code>
<code>setbkcolor(backColor)</code>	sets the background color to <code>backColor</code>
<code>setcolor(foreColor)</code>	sets the foreground color to <code>foreColor</code>
<code>setfillstyle(filPat, filCol)</code>	sets the fill pattern to <code>filPat</code> and the fill color to <code>filCol</code>

int. We will pass string literals as arguments corresponding to label and textString.

## EXERCISES FOR SECTION 3.8

### Self-Check

1. In Listing 3.21, what is the reason for basing the head radius on getmaxy and not getmaxx?
2. Describe or show the drawing produced by the following fragment. Assume a  $640 \times 480$  palette.

```
circle(200, 50, 25);
line(200, 75, 100, 100);
line(200, 75, 300, 100);
pieslice(200, 75, 245, 295, 100);
line(200, 150, 100, 250);
line(200, 150, 300, 250);
bar(50, 250, 350, 300)
```

3. Write statements to add two windows to the second floor of the house in Figure 3.20.
4. Modify the program in Figure 3.20 so that it draws the house in the center of the screen and with the same relative size regardless of the actual palette dimensions.

### Programming

1. Write the statements to draw a tennis racket in the appropriate place in the figure for Self-Check Exercise 2. At the end of a thin bar, draw a circle and fill it with a green crosshatch pattern.
2. Write a graphics program that draws a rocket ship consisting of a triangle (the cone) on top of a rectangle (the body). Draw a pair of intersecting lines under the rectangle. Fill the cone with a blue hatch pattern and the body with a red solid pattern.
3. Write a program that draws a pair of nested rectangles at the center of the screen, filling the inner rectangle in red and the outer rectangle in white. The outer rectangle should have a width  $1/4$  of the X-dimension and a height  $1/4$  of the Y-dimension of the screen. The height and width of the inner rectangle should be half that of the outer rectangle.
4. Write a program that draws a male and a female stick figure side-by-side.

## 3.9 Common Programming Errors

Remember to use a `#include` compiler directive for every standard library that your program accesses. If you're including a user-defined library, enclose the header file name in quotes, not angle brackets.

If you're using function subprograms, place their prototypes in the source file before the `main` function. Place the actual function definitions after the `main` function.

Syntax or run-time errors may occur when you use functions. The acronym **not** summarizes the requirements for argument list correspondence. Provide the required number of arguments and make sure the order of arguments is correct. Make sure that each function argument is the correct type or that conversion to the correct type will lose no information. For user-defined functions, verify that each argument list is correct by comparing it to the formal parameter list in the function heading or prototype. Also be careful when using functions that are undefined on some range of values. For example, if the argument for function `sqrt`, `log`, or `log10` is negative, a run-time error will occur. The following are additional errors to look for.

- *Semicolons in a function heading and prototype:* A missing semicolon at the end of a function prototype may cause a "Statement missing ;" or "Declaration terminated incorrectly" diagnostic. (A prototype is a declaration and must be terminated with a semicolon.) However, make sure you don't insert a semicolon after the function heading.
- *Wrong number of arguments:* If the number of arguments in a function call is not the same as the number in the prototype, your compiler may generate an error message such as the following:

```
Incorrect number of arguments in call to intPower(int, int).
```

As shown above, in most cases the error message lists the function prototype—`intPower(int, int)` in this case—to help you determine the exact nature of the error.

- *Argument mismatches:* Verify that each actual argument in a function call is in the right position relative to its corresponding formal parameter. You do this by comparing each actual argument to the type and description of its corresponding formal parameter in the function prototype. Remember, the actual argument name is not what's important; it's the positional correspondence that's critical.

C++ permits most type mismatches that you're likely to create and usually won't generate even a warning message. Instead, the compiler will

perform a *standard conversion* on the actual argument, converting it to the type specified by the formal parameter. These conversions may produce incorrect results, which in turn can cause other errors during the execution of your program. If you pass a real number to a type `char` formal parameter, your compiler may generate a warning diagnostic such as

Float or double assigned to integer or character data type.

- *Function prototype and definition mismatches:* The compiler will not detect type mismatches between a function prototype and the function definition, but they might be detected by the linker program for your C++ system.

undefined symbol `intPower(int, int)` in module square.

This message may be caused by a missing function definition or by the use of a function prototype that doesn't match the function definition. If the parameters in a function prototype don't match those in the definition, the linker assumes that the prototype refers to a different function, one for which there is no definition.

- *Return statement errors:* All functions that you write, except for type `void` functions, should terminate execution with a `return` statement that includes an expression indicating the value to be returned. The data type of the value returned should match the type specified in the function heading. Make sure that you don't omit the expression (or even worse, the entire `return` statement). If you forget to specify a return value when one is expected, you'll see a message such as

Return value expected.

or

Function should return a value . . .

If you specify a return value for a `void` function, you'll see a message such as

Function cannot return a value . . .

- *Missing object name in call to a member function:* If you omit the object name and dot when attempting to apply a member function to an object, you'll get an error message such as

Call to undefined function

Because the object name is missing, the compiler can't determine the class library in which the function is defined and assumes the function definition is missing.



- *Missing #include line or incorrect library name in #include:* If you forget to include a library header file or write the wrong header (for example, `<cmath>` instead of `<string>`), you'll see multiple error messages because the compiler will not be able to access the symbols and functions defined in that library. The error messages will tell you that symbols and functions from that library are undefined. A similar error occurs if you omit the `using namespace std;` statement.
- *Argument mismatch in a call to a member function:* If you have argument type mismatches or an incorrect number of arguments in a call to a member function, you'll get an error message such as

```
Could not find a match for getline(string, char)
```

This message is displayed if you call function `getline` with a `string` and `char` argument instead of a `stream`, `string`, and `char` argument.

- *Logic errors in your program—testing a program and checking your results:* Many errors, such as the incorrect specification of computations (in writing mathematical formulas) may go undetected by the compiler, yet produce incorrect results. For example, if you're given the formula

$$y = 3k^2 - 9k + 7$$

to program, and you write the C++ code

```
y = 9 * pow(k, 2) 2 3 - k + 7
```

(accidentally reversing the coefficients 9 and 3), no error will be detected by the compiler. As far as the compiler is concerned, no mistake has been made—the expression is perfectly legal C++ code.

## Separately Testing Function Subprograms

There is usually just one way to find logic errors, and that's by testing your program using carefully chosen *test data samples* and verifying, for each sample, that your answer is correct. Such testing is a critical part of the programming process and cannot be omitted.

As we proceed through the text, we will have more to say about testing strategies. We discussed one testing strategy that involves breaking down a problem into subproblems, writing the solutions to the subproblems using separate functions, and then separately testing these functions with short driver functions. This strategy can help simplify the testing process and make it easier for you to perform a more thorough test of your entire program.

## Chapter Review

1. Develop your program solutions from existing information. Use the system documentation derived from applying the software development method as the initial framework for the program.
  - Edit the data requirements to obtain the main function declarations.
  - Use the refined algorithm as the starting point for the executable statements in the main function.
2. If a new problem is an extension of a previous one, modify the previous program rather than starting from scratch.
3. Use C++ library functions to simplify mathematical computations through the reuse of code that has already been written and tested. Write a function call (consisting of the function name and arguments) to activate a library function. After the function executes, the function result is substituted for the function call.
4. Use a structure chart to show subordinate relationships between subproblems.
5. Utilize modular programming by writing separate function subprograms to implement the different subproblems in a structure chart. Ideally, your main function will consist of a sequence of function call statements that activate the function subprograms.
6. You can write functions without arguments and results to display a list of instructions to a program user or to draw a diagram on the screen. Use a function call consisting of the function name followed by an empty pair of parentheses ( ) to activate such a function.
7. Write functions that have input arguments and that return a single result to perform computations similar to those performed by library functions. When you call such a function, each actual argument value is assigned to its corresponding formal parameter.
8. Place prototypes (similar to function headings) for each function subprogram before the main function, and place the function definitions after the main function in a source file. Use ( ) to indicate that a function has no parameters. List only the formal parameter types, not their names, in the prototype. Use a semicolon after the prototype but not after the function heading.
9. You can use the standard `string` class and its member functions to process textual data. Insert the compiler directive `#include <string>` and the statement `using namespace std;` in your program. You should use dot notation to apply a member function to an object (*object.function-call*).

## Summary of New C++ Constructs

Construct	Effect
Function Prototype <code>void display();</code>	Prototype for a function that has no arguments and returns no result.
<code>float average(float, float);</code>	Prototype for a function with two type <code>float</code> input arguments and a type <code>float</code> result.
Function Call <code>display();</code>	Calls void function <code>display</code> , causing it to begin execution. When execution is complete, control returns to the statement in the calling function that immediately follows the call.
<code>x = average(2, 6.5) + 3.0;</code>	Calls function <code>average</code> with actual arguments 2 and 6.5. When execution is complete, adds 3.0 to the function result and stores the sum in variable <code>x</code> .
Member Function Call Using Dot Notation <code>cout &lt;&lt; wholeName.at(0);</code>	Calls <code>string</code> member function <code>at</code> , applying it to object <code>wholeName</code> . Displays the first character in <code>wholeName</code> .
Function Definition <code>// Displays a diamond of stars</code> <code>void display()</code> <code>{</code> <code>cout &lt;&lt; " * " &lt;&lt; endl;</code> <code>cout &lt;&lt; " * * " &lt;&lt; endl;</code> <code>cout &lt;&lt; " * * " &lt;&lt; endl;</code> <code>cout &lt;&lt; "* * " &lt;&lt; endl;</code> <code>cout &lt;&lt; " * * " &lt;&lt; endl;</code> <code>cout &lt;&lt; " * * " &lt;&lt; endl;</code> <code>cout &lt;&lt; " * " &lt;&lt; endl;</code> <code>}</code> <code>// end display</code> <code>// Computes average of two integers</code> <code>float average(float, float)</code> <code>{</code> <code>return (m1 + m2) / 2.0;</code> <code>}</code> <code>// end average</code>	Definition of a function that has no arguments and returns no result.
	Definition of a function with two type <code>float</code> input arguments and a type <code>float</code> result.
	Returns a type <code>float</code> result.

## Quick-Check Exercises

1. Each function in a program executes exactly once. True or false?
2. State the order of declarations used in a program source file.
3. What is the difference between an actual argument and a formal parameter? How is correspondence between arguments and parameters determined?

4. Can you use an expression for a formal parameter or for an actual argument? If so, for which?
5. What is a structure chart? Explain how a structure chart differs from an algorithm.
6. What does the function below do?

```
void nonsense(char c, char d)
{
    cout << "*****" << endl;
    cout << c << "!!!" << d << endl;
    cout << "*****" << endl;
} // end nonsense
```

7. Given the function nonsense from Exercise 6, describe the output that's produced when the following lines are executed.

```
nonsense('a', 'a');
nonsense('A', 'Z');
nonsense('!', '!');
```

8. Explain what dot notation is and how you use it.
9. Trace the statements below:

```
string flower = "rose";
flower = flower + " of Sharon";
cout << flower.at(0) << flower.at(8) << endl;
cout << flower.find("s") << " " << flower.find("S")
<< endl;
flower.replace(5, 2, "from");
flower.erase(0, 4);
flower.insert(0, "thorn");
```

10. Explain the role of a function prototype and a function definition. Which comes first? Which must contain the formal parameter names? Which ends with a semicolon?
11. Write the following equation as a C++ statement using functions exp, log, and pow:

$$y = (e^{n \ln b})^2$$

### Review Questions

1. Discuss the strategy of divide and conquer.
2. Provide guidelines for the use of function interface comments.

3. Briefly describe the steps you would take to derive an algorithm for a given problem.
4. The diagram that shows the algorithm steps and their interdependencies is called a \_\_\_\_\_.
5. What are three advantages of using functions?
6. A C++ program is a collection of one or more \_\_\_\_\_, one of which must be named \_\_\_\_\_.
7. When is a function executed? What is the difference between a function prototype and its definition? Which comes first?
8. Is the use of functions a more efficient use of the programmer's or the computer's time? Explain your answer.
9. Write a program that reads into a string object a name with three blanks between the first and last names. Then extract the first and last names and store them in separate string objects. Write a function subprogram that displays its string argument two times. Call this function to display the first name four times and then to display the last name six times.
10. Write a program that reads into a string object your name with the symbol \* between first and last names. Then extract your first and last names and store them in separate string objects. Write a function subprogram that displays its string argument three times. Call this function to display your first name three times and then to display your last name three times.

## Programming Projects

1. Add one or more of your own unique functions to the stick figure program presented in Section 3.2. Create several more pictures combining the `drawCircle`, `drawIntersect`, `drawBase`, and `drawParallel` functions with your own. Make any modifications to these functions that you need in order to make the picture components fit nicely.
2. Write functions that display each of your initials in block letter form. Use these functions to display your initials.
3. Write three functions, one that displays a circle, one that displays a rectangle, and one that displays a triangle. Use these functions to write a complete C++ program from the following outline:

```
int main()
{
    // Draw circle.
    // Draw triangle.
```

```

        // Draw rectangle.
        // Display 2 blank lines.
        // Draw triangle.
        // Draw circle.
        // Draw rectangle.

    return 0;
}

```

4. Write a computer program that computes the duration of a projectile's flight and its height above the ground when it reaches the target. As part of your solution, write and call a function that displays instructions to the program user.

#### Problem Constant

$G = 32.17$  // gravitational constant

#### Problem Inputs

```

float theta    // input - angle (radians) of elevation
float distance // input - distance (ft) to target
float velocity // input - projectile velocity (ft/sec)

```

#### Problem Outputs

```

float time    // output - time (sec) of flight
float height  // output - height at impact

```

#### Relevant Formulas

$$time = \frac{distance}{velocity \times \cos(\theta)}$$

$$height = velocity \times \sin(\theta) \times time - \frac{g \times time^2}{2}$$

Try your program on these data sets.

Inputs	Data Set 1	Data Set 2
Angle of elevation	0.3 radian	0.71 radian
Velocity	800 ft/sec	1,600 ft/sec
Distance to target	11,000 ft	78,670 ft

5. Write a program that takes a positive number with a fractional part and rounds it to two decimal places. For example, 32.4851 would round to 32.49, and 32.4431 would round to 32.44.
6. Four track stars entered the mile race at the Penn Relays. Write a program that will read the last name and the race time in minutes and seconds for one runner and compute and print the speed in feet per

second and in meters per second after the runner's name. (Hints: There are 5280 feet in one mile, and one kilometer equals 3281 feet; one meter is equal to 3.281 feet.) Test your program on each of the times below.

Name	Minutes	Seconds
Deavers	3	52.83
Jackson	3	59.83
Smith	4	00.03
Rivera	4	16.22

Write and call a function that displays instructions to the program user. Write two other functions, one to compute the speed in meters per second and the other to compute the speed in feet per second.

7. A cyclist coasting on a level road slows from a speed of 10 miles/hr to 2.5 miles/hr in one minute. Write a computer program that calculates the cyclist's constant rate of deceleration and determines how long it will take the cyclist to come to rest, given an initial speed of 10 miles/hr. (Hint: Use the equation

$$a = (v_f - v_i) / t,$$

where  $a$  is acceleration,  $t$  is time interval,  $v_i$  is initial velocity, and  $v_f$  is the final velocity.) Write and call a function that displays instructions to the program user and another function that computes and returns the deceleration given  $v_f$ ,  $v_i$ , and  $t$ .

8. In shopping for a new house, you must consider several factors. In this problem the initial cost of the house, estimated annual fuel costs, and annual tax rate are available. Write a program that will determine the total cost after a five-year period for each set of house data below. You should be able to inspect your program output to determine the "best buy."

Initial House Cost	Annual Fuel Cost	Tax Rate
\$175,000	\$2500	0.025
\$200,000	\$2800	0.025
\$210,000	\$2050	0.020

To calculate the house cost, add the fuel cost for five years to the initial cost, then add the taxes for five years. Taxes for one year are computed by multiplying the tax rate by the initial cost. Write and call a function that displays instructions to the program user and another function that computes and returns the house cost given the initial cost, the annual fuel cost, and the tax rate.

9. Write a program that reads a string containing exactly four words (separated by \* symbols) into a single string object. Next, extract each word from the original string and store each word in a string object. Then concatenate the words in reverse order to form another string. Display both the original and final strings. (Hint: To extract the words, you should use the `find` member function to find each symbol \*, assign the characters up to the \* to one of the four string objects, and then remove those characters from the original string.)
10. Write a program to take a depth (in kilometers) inside the earth as input data; compute and display the temperature at this depth in degrees Celsius and Fahrenheit. The relevant formulas are

$$\text{Celsius} = 10 \times (\text{depth}) + 20 \quad (\text{Celsius temperature at depth in km})$$

$$\text{Fahrenheit} = 1.8 \times (\text{Celsius}) + 32$$

Include two functions in your program. Function `celsiusAtDepth` should compute and return the Celsius temperature at a depth measured in kilometers. Function `toFahrenheit` should convert a Celsius temperature to Fahrenheit.

11. The ratio between successive speeds of a six-speed gearbox (assuming that the gears are evenly spaced to allow for whole teeth) is

$$\sqrt[5]{M/m}$$

where  $M$  is the maximum speed in revolutions per minute and  $m$  is the minimum speed. Write a function `speedsRatio` that calculates this ratio for any maximum and minimum speeds. Write a main function that prompts for maximum and minimum speeds (rpm), calls `speedsRatio` to calculate the ratio, and displays the results.

12. Write a program that calculates and displays the volume of a box and the surface area of a box. The box dimensions are provided as input data. The volume is equal to the area of the base times the height of the box. Define and call a function `rectangleArea` to calculate the area of each rectangle in the box.
13. For any integer  $n > 0$ ,  $n!$  is defined as the product  $n \times n - 1 \times n - 2 \dots \times 2 \times 1$ .  $0!$  is defined to be 1. It is sometimes useful to have a closed-form definition instead; for this purpose, an approximation can be used. R.W. Gosper proposed the following such approximation formula:

$$n! \approx n^n e^{-n} \sqrt{\left(2n + \frac{1}{3}\right)\pi}$$



Create a program that prompts the user to enter an integer  $n$ , uses Gosper's formula to approximate  $n!$ , and then displays the result. The message displaying the result should look something like this:

```
5! equals approximately 119.97003
```

Your program will be easier to debug if you use some intermediate values instead of trying to compute the result in a single expression. If you are not getting the correct results, then you can compare the results of your intermediate values to what you get when you do the calculations by hand. Use at least two intermediate variables—one for  $2n + \frac{1}{3}$  and one for  $\sqrt{(2n + \frac{1}{3})\pi}$ . Display each of these intermediate values to simplify debugging. Be sure to use a named constant for  $\pi$ , and use the approximation 3.14159265. Test the program on non-negative integers less than 8.

14. Write a program that calculates the speed of sound ( $a$ ) in air of a given temperature  $T$  (°F). Formula to compute the speed in ft/sec:

$$a = 1086 \sqrt{\frac{5T + 297}{247}}$$

Be sure your program does not lose the fractional part of the quotient in the formula shown. As part of your solution, write and call a function that displays instructions to the program user.

15. After studying the population growth of Gotham City in the last decade of the twentieth century, we have modeled Gotham's population function as

$$P(t) = 52.966 + 2.184t$$

where  $t$  is years after 1990, and  $P$  is population in thousands. Thus,  $P(0)$  represents the population in 1990, which was 52.966 thousand people. Write a program that defines a function named `population` that predicts Gotham's population in the year provided as an input argument. Write a program that calls the function and interacts with the user as follows:

```
Enter a year after 1990> 2015
Predicted Gotham City population for 2010 (in thousands):
107.566
```

## Graphics Projects

16. Use graphics functions in programs that draw a rocket ship (triangle over rectangle over intersecting lines), a male stick figure (circle over rectangle over intersecting lines), and a female stick figure standing on the head of a male stick figure.

17. Redo Programming Project 2 using graphics mode.
18. Read in five values that represent the monthly amount spent on budget categories: food, clothing, transportation, entertainment, and rent. Write a program that displays a bar graph showing these values. In a bar graph, the length of each bar is proportional to the value it represents. Use a different color for each bar. (Hint: Multiply each value by the scale factor  $\text{maxy}() / \text{maxExpense}$ , where  $\text{MaxExpense}$  is the largest possible expense.
19. Redo Programming Project 18 using a pie chart. For the pie chart, the arc length of each section should be proportional to the amount it represents. Use a different fill pattern and color for each section.
20. Redo Programming Project 18 drawing a line graph. The first line should begin at the height representing the first budget category value and end at the height representing the second budget category value; the second line should begin at the height representing the second budget category value and end at the height representing the third budget category value; and so on.

#### Answers to Quick-Check Exercises

1. False; a function is only executed when it is called. It can execute more than once or not at all.
2. We declare function prototypes before function `main`. Next comes function `main` followed by any other function definitions. Within each function, we declare any local constants and variables that are defined only during the execution of that function.
3. An actual argument is listed in a function call; a formal parameter is used inside the function body to describe the function operation. When a function call executes, the value of each actual argument is passed into the corresponding formal parameter. Correspondence is determined by relative position in the lists of arguments and parameters (that is, the first actual argument corresponds to the first formal parameter, and so on).
4. An actual argument can be an expression but not a formal parameter.
5. A structure chart is a diagram used to show an algorithm's subproblems and their interdependence, so it shows the hierarchical relationship between subproblems. An algorithm lists the sequence in which subproblems are performed.
6. It would display a line of five stars, followed by a line showing the first argument, three exclamation points, and the second argument, followed by a line of five stars.

7. It displays the following:

```
*****
a!!!a
*****
*****
A!!!Z
*****
*****
!!!!!!
*****
```

8. In dot notation, you write an object name, a dot, and a member function name. You use dot notation to apply the member function to the object that precedes the dot.
9. Allocate a string object `flower` and initialize it to "rose".  
 Change `flower` to "rose of Sharon".  
 Display `rS`.  
 Display 2 8.  
 Change `flower` to "rose from Sharon".  
 Change `flower` to " from Sharon".  
 Change `flower` to "thorn from Sharon".
10. A function prototype gives the C++ compiler enough information to translate each call to the function. The function definition provides a complete description of the function operation and is translated into machine language by the compiler. The prototype comes first; the definition must contain formal parameter names; the prototype ends with a semicolon.
11. `y = pow(exp(n * log(b)), 2);`

# Mark Hall

*Mark Hall is an expert on compilers. He works at Microsoft as the software architect for the Visual C++ compiler. Before that he was a computer scientist at Pyramid Technology. He received his B.S. in computer science from the College of William and Mary and his Ph.D. from the University of Colorado.*



**How did you decide to study computer science?**

I remember filling out my college application in the kitchen of my parents' house. The application asked, "in what field do you expect to pursue a career?" I was a hopeless tinkerer as a kid, and when I saw the "computer design" option, I checked the box without hesitation. Of course, there was no checkbox for "software design" at that time, and I wouldn't have checked it if there was one, because I had no idea what value software could possibly have. I was thinking purely hardware. In fact, my first week in an introductory programming course was quite a struggle for me. I was a math major in high school, and I just couldn't fathom what  $x = x + 1$  could possibly mean. Then one night I was in the card punch room in the basement of the math building, and a little voice said, "You dummy, that's assignment, not equality," and the true nature of computers was revealed to me. I was shocked! Until then I thought computers were sophisticated, but you had to instruct them to do even the simplest task. The real

sophistication was in the software.

When I realized you could get from  $x = x + 1$  to the video game Galaxians on the same piece of hardware, I knew my career would be in software. I was hooked. I could tinker in software to my heart's delight and build anything I wanted.

In my fourth year of college I took the first compiler course offered at my school. The fact that a compiler was a software program that read other software programs, *including itself*, was totally mind-blowing. Then, one fateful day we studied DeRemer's thesis on constructing LR(k) parsers, and when the professor built the parse states for a grammar on the board, my jaw just hung open. I kept saying, "No way!" I've been working in the compiler field ever since.

**Describe your work at Microsoft.**

Every day is basically a balancing act between standard's conformance, bugs, new features, and platform support. Customers rightfully want it all, and they think Microsoft is big enough to deliver it all. But it's actually pretty

difficult to hire and retain gifted compiler developers. It really has to be in your blood, because it isn't the most glamorous or hip software job at Microsoft (just think about Xbox, for example). Plus, our product offers so much functionality. Just compiling the language is challenging, but when you add in all the visualization tools like IntelliSense, wizards, designers, browsers, and debuggers, you're talking about many millions of lines of code. At the end of the day, we try to maximize the overall customer benefit of our work.

Lately I've been very busy thinking about the productivity enhancements we want to deliver to C++ programmers over the next decade. The C++ developer is a precious commodity. The vast majority of all revenue in the software business derives from programs written in C++. Fortunately, we have many great ideas for enhancing the development process. The great thing about Microsoft is how the company wants you to make the big, long-term bets. That's why I like working here.

**What is the most challenging part of your job?**

I'd say the most difficult part of the job for me is making the hard tradeoffs needed to ship the product. We have limited resources, competitive pressures, and changing markets, so we never get to implement all of the features or fix all the bugs we want to. This is frustrating, but at some point the customer needs

the product and the access it provides to the new platform features.

**Do you have any advice about programming in C++?**

Yes. My advice is, "Don't be afraid to program in C++." C++ has gotten a bad rap for being too complex. But some of that complexity is useful. I have attended many design meetings for technology frameworks that ironically are more complex than they need to be because the implementation language lacked features available in C++. Let's face it: software is an intellectual pursuit. If you're solving hard problems in software, you need a sophisticated tool. Hammers and screwdrivers are simple, and you can probably build a car with them, but why would you want to? And remember: you don't always have to use all the power of C++ to solve every problem, nor should you. It's quite easy to write simple, easy-to-read code in C++.

**How do you see Visual C++ evolving over the next few years?**

I see a lot of productivity enhancements coming. Visualization and code analysis tools will become much more powerful and precise. This will happen because the compiler itself will fundamentally change from a monolithic program to a set of services that are available to everyone. That will spawn an unprecedented ecology of C++ development tools. I think the language itself will evolve in response to the demand for new and better tools.