CHAPTER 3

Function Basics

3 Function Basics

Good things come in small packages.

Common saying

INTRODUCTION

If you have programmed in some other language, then the contents of this chapter will be familiar to you. You should still scan this chapter to see the C++ syntax and terminology for the basics of functions. Chapter 4 contains the material on functions that might be different in C++ than in other languages.

A program can be thought of as consisting of subparts such as obtaining the input data, calculating the output data, and displaying the output data. C++, like most programming languages, has facilities to name and code each of these subparts separately. In C++ these subparts are called *functions*. Most programming languages have functions or something similar to functions, although they are not always called by that name in other languages. The terms *procedure*, *subprogram*, and *method*, which you may have heard before, mean essentially the same thing as *function*. In C++ a function may return a value (produce a value) or may perform some action without returning a value, but whether the subpart returns a value or not, it is still called a function in C++. This chapter presents the basic details about C++ functions. Before telling you how to write your own functions, we will first tell you how to use some predefined C++ functions.

3.1 Predefined Functions

Do not reinvent the wheel.

Common saying

C++ comes with libraries of predefined functions that you can use in your programs. There are two kinds of functions in C++: functions that return (produce) a value and functions that do not return a value. Functions that do not return a value are called **`void` functions**. We first discuss functions that return a value and then discuss `void` functions.

`void` function

PREDEFINED FUNCTIONS THAT RETURN A VALUE

We will use the `sqrt` function to illustrate how you use a predefined function that returns a value. The `sqrt` function calculates the square root of a number. (The square

root of a number is that number which when multiplied by itself will produce the number you started out with. For example, the square root of 9 is 3 because 3^2 is equal to 9.) The function `sqrt` starts with a number, such as 9.0, and computes its square root, in this case 3.0. The value the function starts out with is called its **argument**. The value it computes is called the **value returned**. Some functions may have more than one argument, but no function has more than one value returned.

argument

value returned

The syntax for using functions in your program is simple. To set a variable named `theRoot` equal to the square root of `9.0`, you can use the following assignment statement:

```
theRoot = sqrt(9.0);
```

The expression `sqrt(9.0)` is known as a **function call** or **function invocation**. An argument in a function call can be a constant, such as `9.0`, a variable, or a more complicated expression. A function call is an expression that can be used like any other expression. For example, the value returned by `sqrt` is of type `double`; therefore, the following is legal (although perhaps stingy):

function call or function invocation

```
bonus = sqrt(sales)/10;
```

`sales` and `bonus` are variables that would normally be of type `double`. The function call `sqrt(sales)` is a single item, just as if it were enclosed in parentheses. Thus, the above assignment statement is equivalent to

```
bonus = (sqrt(sales))/10;
```

You can use a function call wherever it is legal to use an expression of the type specified for the value returned by the function.

Display 3.1 contains a complete program that uses the predefined function `sqrt`. The program computes the size of the largest square doghouse that can be built for the amount of money the user is willing to spend. The program asks the user for an amount of money and then determines how many square feet of floor space can be purchased for that amount. That calculation yields an area in square feet for the floor of the doghouse. The function `sqrt` yields the length of one side of the doghouse floor.

The `cmath` library contains the definition of the function `sqrt` and a number of other mathematical functions. If your program uses a predefined function from some library, then it must contain an **`include` directive** that names that library. For example, the program in Display 3.1 uses the `sqrt` function and so it contains

`#include` directive

```
#include <cmath>
```

This particular program has two `include` directives. It does not matter in what order you give these two `include` directives. `include` directives were discussed in Chapter 1.

Definitions for predefined functions normally place these functions in the `std` namespace and so also require the following `using` directive, as illustrated in Display 3.1:

```
using namespace std;
```

Display 3.1 A Predefined Function That Returns a Value


```
1  //Computes the size of a doghouse that can be purchased
2  //given the user's budget.
3  #include <iostream>
4  #include <cmath>
5  using namespace std;

6  int main( )
7  {
8      const double COST_PER_SQ_FT = 10.50;
9      double budget, area, lengthSide;

10     cout << "Enter the amount budgeted for your doghouse $";
11     cin >> budget;

12     area = budget/COST_PER_SQ_FT;
13     lengthSide = sqrt(area);

14     cout.setf(ios::fixed);
15     cout.setf(ios::showpoint);
16     cout.precision(2);
17        cout << "For a price of $" << budget << endl
18          << "I can build you a luxurious square doghouse\n"
19          << "that is " << lengthSide
20          << " feet on each side.\n";

21     return 0;
22 }
```

SAMPLE DIALOGUE

```
Enter the amount budgeted for your doghouse $25.00
For a price of $25.00
I can build you a luxurious square doghouse
that is 1.54 feet on each side.
```

#include may not be enough

Usually, all you need do to use a library is to place an `include` directive and a `using` directive for that library in the file with your program. If things work with just these directives, you need not worry about doing anything else. However, for some libraries on some systems you may need to give additional instructions to the compiler or explicitly run a linker program to link in the library. The details vary from one system to another; you will have to check your manual or a local expert to see exactly what is necessary.

abs and labs

A few predefined functions are described in Display 3.2. More predefined functions are described in Appendix 4. Notice that the absolute value functions **`abs`** and **`labs`** are

Functions That Return a Value

For a function that returns a value, a function call is an expression consisting of the function name followed by arguments enclosed in parentheses. If there is more than one argument, the arguments are separated by commas. If the function call returns a value, then the function call is an expression that can be used like any other expression of the type specified for the value returned by the function.

SYNTAX

Function_Name(*Argument_List*)

where the *Argument_List* is a comma-separated list of arguments:

Argument_1, *Argument_2*, . . ., *Argument_Last*

EXAMPLES

```
side = sqrt(area);
cout << "2.5 to the power 3.0 is "
     << pow(2.5, 3.0);
```

in the library with header file `cstdlib`, so any program that uses either of these functions must contain the following directive:

```
#include <cstdlib>
```

Also notice that there are three absolute value functions. If you want to produce the absolute value of a number of type `int`, use `abs`; if you want to produce the absolute value of a number of type `long`, use `labs`; and if you want to produce the absolute value of a number of type `double`, use `fabs`. To complicate things even more, `abs` and `labs` are in the library with header file `cstdlib`, whereas **`fabs`** is in the library with header file `cmath`. `fabs` is an abbreviation for *floating-point absolute value*. Recall that numbers with a fraction after the decimal point, such as numbers of type `double`, are often called *floating-point numbers*.

fabs

Another example of a predefined function is `pow`, which is in the library with header file `cmath`. The function `pow` can be used to do exponentiation in C++. For example, if you want to set a variable `result` equal to x^y, you can use the following:

pow

```
result = pow(x, y);
```

Hence, the following three lines of program code will output the number `9.0` to the screen, because $(3.0)^{2.0}$ is `9.0`:

```
double result, x = 3.0, y = 2.0;
result = pow(x, y);
cout << result;
```

Display 3.2 Some Predefined Functions

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0,3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

All these predefined functions require `using namespace std;` as well as an `include` directive.

arguments have a type

Notice that the previous call to `pow` returns `9.0`, not `9`. The function `pow` always returns a value of type `double`, not of type `int`. Also notice that the function `pow` requires two arguments. A function can have any number of arguments. Moreover, every argument position has a specified type, and the argument used in a function call should be of that type. In many cases, if you use an argument of the wrong type, some

automatic type conversion will be done for you by C++. However, the results may not be what you intended. When you call a function, you should use arguments of the type specified for that function. One exception to this caution is the automatic conversion of arguments from type `int` to type `double`. In many situations, including calls to

`void` Functions

A `void` function performs some action, but does not return a value. For a `void` function, a function call is a statement consisting of the function name followed by arguments enclosed in parentheses and then terminated with a semicolon. If there is more than one argument, the arguments are separated by commas. For a `void` function, a function invocation (function call) is a statement that can be used like any other C++ statement.

SYNTAX

Function_Name(*Argument_List*);

where the *Argument_List* is a comma-separated list of arguments:

Argument_1, *Argument_2*, . . . , *Argument_Last*

EXAMPLE

```
exit(1);
```

the function `pow`, you can safely use an argument of type `int` (or other integer type) when an argument of type `double` (or other floating-point type) is specified.

Many implementations of `pow` have a restriction on what arguments can be used. In these implementations, if the first argument to `pow` is negative, then the second argument must be a whole number. It might be easiest and safest to use `pow` only when the first argument is nonnegative.

restrictions on `pow`

PREDEFINED `void` FUNCTIONS

A `void` function performs some action, but does not return a value. Since it performs an action, a `void` function invocation is a statement. The function call for a `void` function is written similar to a function call for a function that returns a value, except that it is terminated with a semicolon and is used as a statement rather than as an expression. Predefined `void` functions are handled in the same way as predefined functions that return a value. Thus, to use a predefined `void` function, your program must have an `include` directive that gives the name of the library that defines the function.

For example, the function **`exit`** is defined in the library `cstdlib`, and so a program that uses that function must contain the following at (or near) the start of the file:

`exit`

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

The exit Function

The `exit` function is a predefined `void` function that takes one argument of type `int`. Thus, an invocation of the `exit` function is a statement written as follows:

```
exit(Integer_Value);
```

When the `exit` function is invoked (that is, when the above statement is executed), the program ends immediately. Any *Integer_Value* may be used, but by convention, `1` is used for a call to `exit` that is caused by an error, and `0` is used in other cases.
The `exit` function definition is in the library `cstdlib` and it places the `exit` function in the `std` namespace. Therefore, any program that uses the `exit` function must contain the following two directives:

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

The following is a sample invocation (sample call) of the function `exit`:

```
exit(1);
```

An invocation of the `exit` function ends the program immediately. Display 3.3 contains a toy program that demonstrates the `exit` function.

Display 3.3 A Function Call for a Predefined `void` Function


```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;

4  int main( )
5  {
6      cout << "Hello Out There!\n";
7      exit(1);

8      cout << "This statement is pointless,\n"
9           << "because it will never be executed.\n"
10          << "This is just a toy program to illustrate exit.\n";

11     return 0;
12 }
```

This is just a toy example. It would produce the same output if you omitted these lines.

SAMPLE DIALOGUE

```
Hello Out There!
```

Note that the function `exit` has one argument, which is of type `int`. The argument is given to the operating system. As far as your C++ program is concerned, you can use any `int` value as the argument, but by convention, `1` is used for a call to `exit` that is caused by an error, and `0` is used in other cases.

A `void` function can have any number of arguments. The details on arguments for `void` functions are the same as they were for functions that return a value. In particular, if you use an argument of the wrong type, then, in many cases, some automatic type conversion will be done for you by C++. However, the results may not be what you intended.

SELF-TEST EXERCISES

1. Determine the value of each of the following arithmetic expressions.

```
sqrt(16.0)          sqrt(16)                pow(2.0, 3.0)
pow(2, 3)           pow(2.0, 3)             pow(1.1, 2)
abs(3)              abs(-3)                 abs(0)
fabs(-3.0)          fabs(-3.5)              fabs(3.5)
ceil(5.1)           ceil(5.8)               floor(5.1)
floor(5.8)          pow(3.0, 2)/2.0         pow(3.0, 2)/2
7/abs(-2)           (7 + sqrt(4.0))/3.0     sqrt(pow(3, 2))
```

2. Convert each of the following mathematical expressions to a C++ arithmetic expression.

   a. $\sqrt{x+y}$ b. x^{y+7} c. $\sqrt{area + fudge}$

   d. $\frac{\sqrt{time + tide}}{nobody}$ e. $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ f. $|x - y|$

3. Write a complete C++ program to compute and output the square roots of the whole numbers from 1 to 10.

4. What is the function of the `int` argument to the `void` function `exit`?

A RANDOM NUMBER GENERATOR

A random number generator is a function that returns a "randomly chosen" number. It is unlike the functions we have seen so far in that the value returned is not determined by the arguments (of which there are usually none) but rather by some global

conditions. Since you can think of the value returned as being a random number, you can use a random number generator to simulate random events, such as the result of throwing dice or flipping a coin. In addition to simulating games of chance, random number generators can be used to simulate things that strictly speaking may not be random but that appear to us to be random, such as the amount of time between the arrival of cars at a toll booth.

rand

The C++ library with header file <cstdlib> contains a random number function named rand. This function has no arguments. When your program invokes rand, the function returns an integer in the range 0 to RAND_MAX, inclusive. (The number generated might be equal to 0 or RAND_MAX.) RAND_MAX is a defined integer constant whose definition is also in the library with header file <cstdlib>. The exact value of RAND_MAX is system-dependent but will always be at least 32767 (the maximum two-byte positive integer). For example, the following outputs a list of ten "random" numbers in the range 0 to RAND_MAX:

RAND_MAX

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

You are more likely to want a random number in some smaller range, such as the range 0 to 10. To ensure that the value is in the range 0 to 10 (including the end points), you can use

```
rand( ) % 11
```

scaling

This is called **scaling**. The following outputs ten "random" integers in the range 0 to 10 (inclusive):

```
int i;
for (i = 0; i < 10; i++)
    cout << (rand( ) % 11) << endl;
```

Random number generators, such as the function rand, do not generate truly random numbers. (That's the reason for all the quotes around "random.") A sequence of calls to the function rand (or almost any random number generator) will produce a sequence of numbers (the values returned by rand) that appear to be random. However, if you could return the computer to the state it was in when the sequence of calls to rand began, you would get the same sequence of "random numbers." Numbers that appear to be random but really are not, such as a sequence of numbers generated by calls to rand, are called **pseudorandom numbers**.

pseudorandom number

seed

A sequence of pseudorandom numbers is usually determined by one number known as the **seed**. If you start the random number generator with the same seed, over and over, then each time it will produce the same (random-looking) sequence of numbers. You can use the function srand to set the seed for the function rand. The void

function srand takes one (positive) integer argument, which is the seed. For example, the following will output two identical sequences of ten pseudorandom numbers: srand

```
int i;
srand(99);
for (i = 0; i < 10; i++)
    cout << (rand( ) % 11) << endl;
srand(99);
for (i = 0; i < 10; i++)
    cout << (rand( ) % 11) << endl;
```

There is nothing special about the number 99, other than the fact that we used the same number for both calls to srand.

Note that the sequence of pseudorandom numbers produced for a given seed might be system-dependent. If rerun on a different system with the same seed, the sequence of pseudorandom numbers might be different on that system. However, as long as you are on the same system using the same implementation of C++, the same seed will produce the same sequence of pseudorandom numbers.

Pseudorandom Numbers

The function rand takes no arguments and returns a pseudorandom integer in the range 0 to RAND_MAX (inclusive). The void function srand takes one argument, which is the seed for the random number generator rand. The argument to srand is of type unsigned int, so the argument must be nonnegative. The functions rand and srand, as well as the defined constant RAND_MAX, are defined in the library cstdlib, so programs that use them must contain the following directives:

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

These pseudorandom numbers are close enough to true random numbers for most applications. In fact, they are often preferable to true random numbers. A pseudorandom number generator has one big advantage over a true random number generator: The sequence of numbers it produces is repeatable. If run twice with the same seed value, it will produce the same sequence of numbers. This can be very handy for a number of purposes. When an error is discovered and fixed, the program can be rerun with the same sequence of pseudorandom numbers as those that exposed the error. Similarly, a particularly interesting run of the program can be repeated, provided a pseudorandom number generator is used. With a true random number generator every run of the program is likely to be different.

Display 3.4 shows a program that uses the random number generator rand to "predict" the weather. In this case the prediction is random, but some people think that is

about as good as weather prediction gets. (Weather prediction can actually be very accurate, but this program is just a game to illustrate pseudorandom numbers.)

Note that in Display 3.4, the seed value used for the argument of `srand` is the month times the day. That way if the program is rerun and the same date is entered,

Display 3.4 A Function Using a Random Number Generator (part 1 of 2)


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

int main( )
{
    int month, day;
    cout << "Welcome to your friendly weather program.\n"
         << "Enter today's date as two integers for the month and the day:\n";
    cin >> month;
    cin >> day;
    srand(month*day);
    int prediction;
    char ans;
    cout << "Weather for today:\n";
    do
    {
        prediction = rand( ) % 3;
        switch (prediction)
        {
            case 0:
                cout << "The day will be sunny!!\n";
                break;
            case 1:
                cout << "The day will be cloudy.\n";
                break;
            case 2:
                cout << "The day will be stormy!\n";
                break;
            default:
                cout << "Weather program is not functioning properly.\n";
        }
        cout << "Want the weather for the next day?(y/n): ";
        cin >> ans;
    } while (ans == 'y' || ans == 'Y');
    cout << "That's it from your 24-hour weather program.\n";
    return 0;
}
```

(continued)

Display 3.4 A Function Using a Random Number Generator (part 2 of 2)

SAMPLE DIALOGUE

```
Welcome to your friendly weather program.
Enter today's date as two integers for the month and the day:
2  14
Weather for today:
The day will be cloudy.
Want the weather for the next day?(y/n): y
The day will be cloudy.
Want the weather for the next day?(y/n): y
The day will be stormy!
Want the weather for the next day?(y/n): y
The day will be stormy!
Want the weather for the next day?(y/n): y
The day will be sunny!!
Want the weather for the next day?(y/n): n
That's it from your 24-hour weather program.
```

the same prediction will be made. (Of course, this program is still pretty simple. The prediction for the day after the 14th may or may not be the same as the 15th, but this program will do as a simple example.)

floating-point random numbers

Probabilities are usually expressed as a floating-point number in the range 0.0 to 1.0. Suppose you want a random probability instead of a random integer. This can be produced by another form of scaling. The following generates a pseudorandom floating-point value between `0.0` and `1.0`:

```
(RAND_MAX - rand( ))/static_cast<double>(RAND_MAX)
```

The type cast is made so that we get floating-point division rather than integer division.

SELF-TEST EXERCISES

5. Give an expression to produce a pseudorandom integer number in the range 5 to 10 (inclusive).
6. Write a complete program that asks the user for a seed and then outputs a list of ten random numbers based on that seed. The numbers should be floating-point numbers in the range `0.0` to `1.0` (inclusive).

3.2 Programmer-Defined Functions

A custom-tailored suit always fits better than one off the rack.

My uncle, the tailor

The previous section told you how to use predefined functions. This section tells you how to define your own functions.

DEFINING FUNCTIONS THAT RETURN A VALUE

You can define your own functions, either in the same file as the `main` part of your program or in a separate file so that the functions can be used by several different programs. The definition is the same in either case, but for now we will assume that the function definition will be in the same file as the `main` part of your program. This subsection discusses only functions that return a value. A later subsection tells you how to define `void` functions.

Display 3.5 contains a sample function definition in a complete program that demonstrates a call to the function. The function is called `totalCost` and takes two arguments—the price for one item and the number of items for a purchase. The function returns the total cost, including sales tax, for that many items at the specified price. The function is called in the same way a predefined function is called. The definition of the function, which the programmer must write, is a bit more complicated.

function declaration or function prototype

The description of the function is given in two parts. The first part is called the **function declaration** or **function prototype**. The following is the function declaration (function prototype) for the function defined in Display 3.5:

```
double totalCost(int numberParameter, double priceParameter);
```

type for value returned

The first word in a function declaration specifies the type of the value returned by the function. Thus, for the function `totalCost`, the type of the value returned is `double`. Next, the function declaration tells you the name of the function; in this case, `totalCost`. The function declaration tells you (and the compiler) everything you need to know in order to write and use a call to the function. It tells you how many arguments the function needs and what type the arguments should be; in this case, the function `totalCost` takes two arguments, the first one of type `int` and the second one of type `double`. The identifiers `numberParameter` and `priceParameter` are called **formal parameters**, or **parameters** for short. A formal parameter is used as a kind of blank, or placeholder, to stand in for the argument. When you write a function declaration, you do not know what the arguments will be, so you use the formal parameters in place of the arguments. Names of formal parameters can be any valid identifiers. Notice that a function declaration ends with a semicolon.

formal parameter

Display 3.5 A Function Using a Random Number Generator


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

3  double totalCost(int numberParameter, double priceParameter);
4  //Computes the total cost, including 5% sales tax,
5  //on numberParameter items at a cost of priceParameter each.

6  int main( )
7  {
8      double price, bill;
9      int number;

10     cout << "Enter the number of items purchased: ";
11     cin >> number;
12     cout << "Enter the price per item $";
13     cin >> price;

14     bill = totalCost(number, price);

15     cout.setf(ios::fixed);
16     cout.setf(ios::showpoint);
17     cout.precision(2);
18     cout << number << " items at "
19          << "$" << price << " each.\n"
20          << "Final bill, including tax, is $" << bill
21          << endl;

22     return 0;
23 }

24 double totalCost(int numberParameter, double priceParameter)
25 {
26     const double TAXRATE = 0.05; //5% sales tax
27     double subtotal;

28     subtotal = priceParameter * numberParameter;
29     return (subtotal + subtotal*TAXRATE);
30 }
```

Function declaration; also called the function prototype

Function call

Function head

Function body

Function definition

SAMPLE DIALOGUE

Enter the number of items purchased: **2**
Enter the price per item: $**10.10**
2 items at $10.10 each.
Final bill, including tax, is $21.21

function definition

Although the function declaration tells you all you need to know to write a function call, it does not tell you what value will be returned. The value returned is determined by the function definition. In Display 3.5 the function definition is in lines 24 to 30 of the program. A **function definition** describes how the function computes the value it returns. A function definition consists of a *function header* followed by a *function body*. The **function header** is written similar to the function declaration, except that the header does *not* have a semicolon at the end. The value returned is determined by the statements in the *function body*.

function header

function body

The **function body** follows the function header and completes the function definition. The function body consists of declarations and executable statements enclosed within a pair of braces. Thus, the function body is just like the body of the `main` part of a program. When the function is called, the argument values are plugged in for the formal parameters, and then the statements in the body are executed. The value returned by the function is determined when the function executes a `return` *statement*. (The details of this "plugging in" will be discussed in Chapter 4.)

`return` statement

A **`return` statement** consists of the keyword `return` followed by an expression. The function definition in Display 3.5 contains the following `return` statement:

```
return (subtotal + subtotal*TAXRATE);
```

When this `return` statement is executed, the value of the following expression is returned as the value of the function call:

```
(subtotal + subtotal*TAXRATE)
```

The parentheses are not needed. The program will run the same if the parentheses are omitted. However, with longer expressions, the parentheses make the `return` statement easier to read. For consistency, some programmers advocate using these parentheses even with simple expressions. In the function definition in Display 3.5 there are no statements after the `return` statement, but if there were, they would not be executed. When a `return` statement is executed, the function call ends.

Note that the function body can contain any C++ statements and that the statements will be executed when the function is called. Thus, a function that returns a value may do any other action as well as return a value. In most cases, however, the main purpose of a function that returns a value is to return that value.

Either the complete function definition or the function declaration (function prototype) must appear in the code before the function is called. The most typical arrangement is for the function declaration and the `main` part of the program to appear in one or more files, with the function declaration before the `main` part of the program, and for the function definition to appear in another file. We have not yet discussed dividing a program across more than one file, and so we will place the function definitions after the `main` part of the program. If the full function definition is placed before the `main` part of the program, the function declaration can be omitted.

ALTERNATE FORM FOR FUNCTION DECLARATIONS

You are not required to list formal parameter names in a function declaration (function prototype). The following two function declarations are equivalent:

```
double totalCost(int numberParameter, double priceParameter);
```

and

```
double totalCost(int, double);
```

We will usually use the first form so that we can refer to the formal parameters in the comment that accompanies the function declaration. However, you will often see the second form in manuals.

This alternate form applies only to function declarations. *A function definition must always list the formal parameter names.*

PITFALL

Arguments in the Wrong Order

When a function is called, the computer substitutes the first argument for the first formal parameter, the second argument for the second formal parameter, and so forth. Although the computer checks the type of each argument, it does not check for reasonableness. If you confuse the order of the arguments, the program will not do what you want it to do. If there is a type violation due to an argument of the wrong type, then you will get an error message. If there is no type violation, your program will probably run normally but produce an incorrect value for the value returned by the function.

PITFALL

Use of the Terms *Parameter* and *Argument*

The use of the terms *formal parameter* and *argument* that we follow in this book is consistent with common usage, but people also often use the terms *parameter* and *argument* interchangeably. When you see the terms *parameter* and *argument,* you must determine their exact meaning from context. Many people use the term *parameter* for both what we call *formal parameters* and what we call *arguments.* Other people use the term *argument* both for what we call *formal parameters* and what we call *arguments.* Do not expect consistency in how people use these two terms. (In this book we sometimes use the term *parameter* to mean *formal parameter,* but this is more of an abbreviation than a true inconsistency.)

FUNCTIONS CALLING FUNCTIONS

A function body may contain a call to another function. The situation for these sorts of function calls is the same as if the function call had occurred in the `main` part of the program; the only restriction is that the function declaration (or function definition) must appear before the function is used. If you set up your programs as we have been doing, this will happen automatically, since all function declarations come before the `main` part of the program and all function definitions come after the `main` part of the program. Although you may include a function *call* within the definition of another function, you cannot place the *definition* of one function within the body of another function definition.

EXAMPLE

A Rounding Function

The table of predefined functions (Display 3.2) does not include any function for rounding a number. The functions `ceil` and `floor` are almost, but not quite, rounding functions. The function `ceil` always returns the next-highest whole number (or its argument if it happens to be a whole number). So, `ceil(2.1)` returns `3.0`, not `2.0`. The function `floor` always returns the nearest whole number less than (or equal to) the argument. So, `floor(2.9)` returns `2.0`, not `3.0`. Fortunately, it is easy to define a function that does true rounding. The function is defined in Display 3.6. The function `round` rounds its argument to the nearest integer. For example, `round(2.3)` returns 2, and `round(2.6)` returns 3.

To see that `round` works correctly, let's look at some examples. Consider `round(2.4)`. The value returned is the following (converted to an `int` value):

```
floor(2.4 + 0.5)
```

which is `floor(2.9)`, or `2.0`. In fact, for any number that is greater than or equal to `2.0` and strictly less than `2.5`, that number plus `0.5` will be less than `3.0`, and so `floor` applied to that number plus `0.5` will return `2.0`. Thus, `round` applied to any number that is greater than or equal to `2.0` and strictly less than `2.5` will return 2. (Since the function declaration for `round` specifies that the type for the value returned is `int`, we have type cast the computed value to the type `int`.)

Now consider numbers greater than or equal to `2.5`; for example, `2.6`. The value returned by the call `round(2.6)` is the following (converted to an `int` value):

```
floor(2.6 + 0.5)
```

which is `floor(3.1)`, or `3.0`. In fact, for any number that is greater than `2.5` and less than or equal to `3.0`, that number plus `0.5` will be greater than `3.0`. Thus, `round` called with any number that is greater than `2.5` and less than or equal to `3.0` will return 3.

Thus, `round` works correctly for all arguments between `2.0` and `3.0`. Clearly, there is nothing special about arguments between `2.0` and `3.0`. A similar argument applies to all nonnegative numbers. So, `round` works correctly for all nonnegative arguments.

Display 3.6 The Function round

Testing program for the function round

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;

4  int round(double number);
5  //Assumes number >= 0.
6  //Returns number rounded to the nearest integer.

7  int main( )
8  {
9      double doubleValue;
10     char ans;

11     do
12     {
13         cout << "Enter a double value: ";
14         cin >> doubleValue;
15         cout << "Rounded that number is " << round(doubleValue) << endl;
16         cout << "Again? (y/n): ";
17         cin >> ans;
18     }while (ans == 'y' || ans == 'Y');
19     cout << "End of testing.\n";

20     return 0;
21 }

22 //Uses cmath:
23 int round(double number)
24 {
25     return static_cast<int>(floor(number + 0.5));
26 }
```

SAMPLE DIALOGUE

Enter a double value: **9.6**
Rounded, that number is 10
Again? (y/n): **y**
Enter a double value: **2.49**
Rounded, that number is 2
Again? (y/n): **n**
End of testing.

SELF-TEST EXERCISES

7. What is the output produced by the following program?

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

char mystery(int firstParameter, int secondParameter);

int main( )
{
    cout << mystery(10, 9) << "ow\n";
    return 0;
}
char mystery(int firstParameter, int secondParameter)
{
    if (firstParameter >= secondParameter)
        return 'W';
    else
        return 'H';
}
```

8. Write a function declaration (function prototype) and a function definition for a function that takes three arguments, all of type `int`, and that returns the sum of its three arguments.
9. Write a function declaration and a function definition for a function that takes one argument of type `double`. The function returns the character value `'P'` if its argument is positive and returns `'N'` if its argument is zero or negative.
10. Can a function definition appear inside the body of another function definition?
11. List the similarities and differences between how you invoke (call) a predefined (that is, library) function and a user-defined function.

FUNCTIONS THAT RETURN A BOOLEAN VALUE

The returned type for a function can be the type `bool`. A call to such a function returns one of the values `true` or `false` and can be used anywhere that a Boolean expression is allowed. For example, it can be used in a Boolean expression to control an `if-else` statement or to control a loop statement. This can often make a program easier to read. By means of a function declaration, you can associate a complex Boolean expression with a meaningful name. For example, the statement

```
if (((rate >= 10) && (rate < 20)) || (rate == 0))
{
    ...
}
```

can be made to read

```
if (appropriate(rate))
{
     ...
}
```

provided that the following function has been defined:

```
bool appropriate(int rate)
{
    return (((rate >= 10) && (rate < 20)) || (rate == 0));
}
```

SELF-TEST EXERCISES

12. Write a function definition for a function called `inOrder` that takes three arguments of type `int`. The function returns `true` if the three arguments are in ascending order; otherwise, it returns `false`. For example, `inOrder(1, 2, 3)` and `inOrder(1, 2, 2)` both return `true`, whereas `inOrder(1, 3, 2)` returns `false`.
13. Write a function definition for a function called `even` that takes one argument of type `int` and returns a `bool` value. The function returns `true` if its one argument is an even number; otherwise, it returns `false`.
14. Write a function definition for a function `isDigit` that takes one argument of type `char` and returns a `bool` value. The function returns `true` if the argument is a decimal digit; otherwise, it returns `false`.

DEFINING `void` FUNCTIONS

In C++ a `void` function is defined in a way similar to that of functions that return a value. For example, the following is a `void` function that outputs the result of a calculation that converts a temperature expressed in degrees Fahrenheit to a temperature expressed in degrees Celsius. The actual calculation would be done elsewhere in the program. This `void` function implements only the subtask for outputting the results of the calculation.

```
void showResults(double fDegrees, double cDegrees)
{
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << fDegrees
         << " degrees Fahrenheit is equivalent to\n"
         << cDegrees << " degrees Celsius.\n";
}
```

void function definition

As the above function definition illustrates, there are only two differences between a function definition for a `void` function and for a function that returns a value. One difference is that we use the keyword `void` where we would normally specify the type of the value to be returned. This tells the compiler that this function will not return any value. The name `void` is used as a way of saying "no value is returned by this function." The second difference is that a `void` function definition does not require a `return` statement. The function execution ends when the last statement in the function body is executed.

void function call

A `void` function call is an executable statement. For example, the above function `showResults` might be called as follows:

```
showResults(32.5, 0.3);
```

If the above statement were executed in a program, it would cause the following to appear on the screen:

```
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

Notice that the function call ends with a semicolon, which tells the compiler that the function call is an executable statement.

When a `void` function is called, the arguments are substituted for the formal parameters, and the statements in the function body are executed. For example, a call to the `void` function `showResults`, which we gave earlier in this section, will cause some output to be written to the screen. One way to think of a call to a `void` function is to imagine that the body of the function definition is copied into the program in place of the function call. When the function is called, the arguments are substituted for the formal parameters, and then it is just as if the body of the function were lines in the program. (Chapter 4 describes the process of substituting arguments for formal parameters in detail. Until then, we will use only simple examples that should be clear enough without a formal description of the substitution process.)

functions with no arguments

It is perfectly legal, and sometimes useful, to have a function with no arguments. In that case there simply are no formal parameters listed in the function declaration and no arguments are used when the function is called. For example, the `void` function `initializeScreen`, defined below, simply sends a newline command to the screen:

```
void initializeScreen( )
{
    cout << endl;
}
```

If your program includes the following call to this function as its first executable statement, then the output from the previously run program will be separated from the output for your program:

```
initializeScreen( );
```

Be sure to notice that even when there are no parameters to a function, you still must include the parentheses in the function declaration and in a call to the function.

Placement of the function declaration (function prototype) and the function definition is the same for `void` functions as what we described for functions that return a value.

Function Declaration (Function Prototype)

A function declaration (function prototype) tells you all you need to know to write a call to the function. A function declaration (or the full function definition) must appear in your code prior to a call to the function. Function declarations are normally placed before the `main` part of your program.

Do not forget this semicolon.

SYNTAX

Type_Returned_Or_void FunctionName(*Parameter_List*);

where the *Parameter_List* is a comma-separated list of parameters:

Type_1 Formal_Parameter_1, *Type_2 Formal_Parameter_2*, ...
..., *Type_Last Formal_Parameter_Last*

EXAMPLES

```
double totalWeight(int number, double weightOfOne);
//Returns the total weight of number items that
//each weigh weightOfOne.

void showResults(double fDegrees, double cDegrees);
//Displays a message saying fDegrees Fahrenheit
//is equivalent to cDegrees Celsius.
```

`return` STATEMENTS IN `void` FUNCTIONS

`void` functions and `return` statements

Both `void` functions and functions that return a value can have `return` statements. In the case of a function that returns a value, the `return` statement specifies the value returned. In the case of a `void` function, the `return` statement does not include any expression for a value returned. A `return` statement in a `void` function simply ends the function call. Every function that returns a value must end by executing a `return` statement. However, a `void` function need not contain a `return` statement. If it does not contain a `return` statement, it will end after executing the code in the function body. It is as if there were an implicit `return` statement just before the final closing brace, `}`, at the end of the function body.

The fact that there is an implicit `return` statement before the final closing brace in a function body does not mean that you never need a `return` statement in a `void` function. For example, the function definition in Display 3.7 might be used as part of a restaurant-management program. That function outputs instructions for dividing a given amount of ice cream among the people at a table. If there are no people at the

Display 3.7 Use of return in a void Function


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

3  void iceCreamDivision(int number, double totalWeight);
4  //Outputs instructions for dividing totalWeight ounces of ice cream among
5  //number customers. If number is 0, only an error message is output.

6  int main( )
7  {
8      int number;
9      double totalWeight;

10     cout << "Enter the number of customers: ";
11     cin >> number;
12     cout << "Enter weight of ice cream to divide (in ounces): ";
13     cin >> totalWeight;

14     iceCreamDivision(number, totalWeight);

15     return 0;
16 }

17 void iceCreamDivision(int number, double totalWeight)
18 {
19     double portion;

20     if (number == 0)
21     {
22         cout << "Cannot divide among zero customers.\n";
23         return;
24     }
25     portion = totalWeight/number;
26     cout << "Each one receives "
27          << portion << " ounces of ice cream." << endl;
28 }
```

If `number` *is* `0`*, then the function execution ends here.*

SAMPLE DIALOGUE

Enter the number of customers: **0**
Enter weight of ice cream to divide (in ounces): **12**
Cannot divide among zero customers.

table (that is, if number equals 0), then the return statement within the if statement terminates the function call and avoids a division by zero. If number is not 0, then the function call ends when the last cout statement is executed at the end of the function body.

PRECONDITIONS AND POSTCONDITIONS

One good way to write a function declaration comment is to break it down into two kinds of information called the *precondition* and the *postcondition*. The **precondition** states what is assumed to be true when the function is called. The function should not be used and cannot be expected to perform correctly unless the precondition holds. The **postcondition** describes the effect of the function call; that is, the postcondition tells what will be true after the function is executed in a situation in which the precondition holds. For a function that returns a value, the postcondition will describe the value returned by the function. For a function that changes the value of some argument variables, the postcondition will describe all the changes made to the values of the arguments.

precondition

postcondition

For example, the following is a function declaration with precondition and postcondition:

```
void showInterest(double balance, double rate);
//Precondition: balance is a nonnegative savings account balance.
//rate is the interest rate expressed as a percentage, such as 5 for 5%.
//Postcondition: The amount of interest on the given balance
//at the given rate is shown on the screen.
```

You do not need to know the definition of the function showInterest in order to use this function. All that you need to know in order to use this function is given by the precondition and postcondition.

When the only postcondition is a description of the value returned, programmers usually omit the word Postcondition, as in the following example:

```
double celsius(double fahrenheit);
//Precondition: fahrenheit is a temperature in degrees Fahrenheit.
//Returns the equivalent temperature expressed in degrees Celsius.
```

Some programmers choose not to use the words *precondition* and *postcondition* in their function comments. However, whether you use the words or not, you should always think in terms of precondition and postcondition when designing a function and when deciding what to include in the function comment.

main IS A FUNCTION

As we already noted, the main part of a program is actually the definition of a function called main. When the program is run, the function main is automatically called; it, in turn, may call other functions. Although it may seem that the return statement in the

main part of a program should be optional, practically speaking it is not. The C++ standard says that you can omit the return 0 statement in the main part of the program, but many compilers still require it and almost all compilers allow you to include it. For the sake of portability, you should include return 0 statement in the main function. You should consider the main part of a program to be a function that returns a value of type int and thus requires a return statement. Treating the main part of your program as a function that returns an integer may sound strange, but that's the tradition which many compilers enforce.

Although some compilers may allow you to get away with it, you should not include a call to main in your code. Only the system should call main, which it does when you run your program.

RECURSIVE FUNCTIONS

C++ does allow you to define recursive functions. Recursive functions are covered in Chapter 13. If you do not know what recursive functions are, there is no need to be concerned until you reach that chapter. If you want to read about recursive functions early, you can read Sections 13.1 and 13.2 of Chapter 13 after you complete Chapter 4. Note that the main function should not be called recursively.

SELF-TEST EXERCISES

15. What is the output of the following program?

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

void friendly( );

void shy(int audienceCount);
int main( )
{
    friendly( );
    shy(6);
    cout << "One more time:\n";
    shy(2);
    friendly( );
    cout << "End of program.\n";
    return 0;
}
```

SELF-TEST EXERCISES (continued)

```
void friendly( )
{
    cout << "Hello\n";
}

void shy(int audienceCount)
{
    if (audienceCount < 5)
        return;
    cout << "Goodbye\n";
}
```

16. Suppose you omitted the `return` statement in the function definition for `iceCreamDivision` in Display 3.7. What effect would it have on the program? Would the program compile? Would it run? Would the program behave any differently?
17. Write a definition for a `void` function that has three arguments of type `int` and that outputs to the screen the product of these three arguments. Put the definition in a complete program that reads in three numbers and then calls this function.
18. Does your compiler allow `void main( )` and `int main( )`? What warnings are issued if you have `int main( )` and do not supply a `return 0;` statement? To find out, write several small test programs and perhaps ask your instructor or a local guru.
19. Give a precondition and a postcondition for the predefined function `sqrt`, which returns the square root of its argument.

3.3 Scope Rules

Let the end be legitimate, let it be within the scope of the constitution, . . .

John Marshall, Chief Justice U.S. Supreme Court,
McCulloch v. Maryland (1819)

Functions should be self-contained units that do not interfere with other functions—or any other code for that matter. To achieve this you often need to give the function variables of its own that are distinct from any other variables that are declared outside the function definition and that may have the same names as the variables that belong to the function. These variables that are declared in a function definition are called *local variables* and are the topic of this section.

LOCAL VARIABLES

Look back at the program in Display 3.1. It includes a call to the predefined function `sqrt`. We did not need to know anything about the details of the function definition for `sqrt` in order to use this function. In particular, we did not need to know what variables were declared in the definition of `sqrt`. A function that you define is no different. Variable declarations within a function definition are the same as if they were variable declarations in a predefined function or in another program. If you declare a variable in a function definition and then declare another variable of the same name in the `main` function of the program (or in the body of some other function definition), then these two variables are two different variables, even though they have the same name. Let's look at an example.

The program in Display 3.8 has two variables named `averagePea`; one is declared and used in the function definition for the function `estimateOfTotal`, and the other is declared and used in the `main` function of the program. The variable `averagePea` in the function definition for `estimateOfTotal` and the variable `averagePea` in the `main` function are two different variables. It is the same as if the function `estimateOfTotal` were a predefined function. The two variables named `averagePea` will not interfere with each other any more than two variables in two completely different programs would. When the variable `averagePea` is given a value in the function call to `estimateOfTotal`, this does not change the value of the variable in the `main` function that is also named `averagePea`.

local variable
scope

Variables that are declared within the body of a function definition are said to be **local** to that function or to have that function as their **scope**. If a variable is local to some function, we sometimes simply call it a *local variable*, without specifying the function.

Another example of local variables can be seen in Display 3.5. The definition of the function `totalCost` in that program begins as follows:

```
double totalCost(int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05; //5% sales tax
    double subtotal;
```

The variable `subtotal` is local to the function `totalCost`. The named constant `TAXRATE` is also local to the function `totalCost`. (A named constant is in fact nothing but a variable that is initialized to a value and that cannot have that value changed.)

PROCEDURAL ABSTRACTION

A person who uses a program should not need to know the details of how the program is coded. Imagine how miserable your life would be if you had to know and remember the code for the compiler you use. A program has a job to do, such as compiling your program or checking the spelling of words in your paper. You need to know *what* the program's job is so that you can use the program, but you do not (or at least should

Display 3.8 **Local Variables** (part 1 of 2)

```
1  //Computes the average yield on an experimental pea growing patch.
2  #include <iostream>
3  using namespace std;

4  double estimateOfTotal(int minPeas, int maxPeas, int podCount);
5  //Returns an estimate of the total number of peas harvested.
6  //The formal parameter podCount is the number of pods.
7  //The formal parameters minPeas and maxPeas are the minimum
8  //and maximum number of peas in a pod.

9  int main( )
10 {
11     int maxCount, minCount, podCount;
12     double averagePea, yield;

13     cout << "Enter minimum and maximum number of peas in a pod: ";
14     cin >> minCount >> maxCount;
15     cout << "Enter the number of pods: ";
16     cin >> podCount;
17     cout << "Enter the weight of an average pea (in ounces): ";
18     cin >> averagePea;

19     yield =
20           estimateOfTotal(minCount, maxCount, podCount) * averagePea;

21     cout.setf(ios::fixed);
22     cout.setf(ios::showpoint);
23     cout.precision(3);
24     cout << "Min number of peas per pod = " << minCount << endl
25          << "Max number of peas per pod = " << maxCount << endl
26          << "Pod count = " << podCount << endl
27          << "Average pea weight = "
28          << averagePea << " ounces" << endl
29          << "Estimated average yield = " << yield << " ounces"
30          << endl;

31     return 0;
32 }
33
34 double estimateOfTotal(int minPeas, int maxPeas, int podCount)
35 {
36     double averagePea;

37     averagePea = (maxPeas + minPeas)/2.0;
38     return (podCount * averagePea);
39 }
```

This variable named `averagePea` *is local to the* `main` *function.*

This variable named `averagePea` *is local to the function* `estimateOfTotal`.

(continued)

Display 3.8 Local Variables (part 2 of 2)

SAMPLE DIALOGUE

```
Enter minimum and maximum number of peas in a pod: 4 6
Enter the number of pods: 10
Enter the weight of an average pea (in ounces): 0.5
Min number of peas per pod = 4
Max number of peas per pod = 6
Pod count = 10
Average pea weight = 0.500 ounces
Estimated average yield = 25.000 ounces
```

Local Variables

Variables that are declared within the body of a function definition are said to be *local to that function* or to have that function as their *scope*. If a variable is local to a function, then you can have another variable (or other kind of item) with the same name that is declared in another function definition; these will be two different variables, even though they have the same name. (In particular, this is true even if one of the functions is the `main` function.)

not) need to know *how* the program does its job. A function is like a small program and should be used in a similar way. A programmer who uses a function in a program needs to know *what* the function does (such as calculate a square root or convert a temperature from degrees Fahrenheit to degrees Celsius), but should not need to know *how* the function accomplishes its task. This is often referred to as treating the function like a *black box*.

black box

Calling something a **black box** is a figure of speech intended to convey the image of a physical device that you know how to use but whose method of operation is a mystery because it is enclosed in a black box that you cannot see inside of (and cannot pry open). If a function is well designed, the programmer can use the function as if it were a black box. All the programmer needs to know is that if he or she puts appropriate arguments into the black box, then it will take some appropriate action. Designing a function so that it can be used as a black box is sometimes called **information hiding** to emphasize the fact that the programmer acts as if the body of the function were hidden from view.

information hiding

procedural abstraction

Writing and using functions as if they were black boxes is also called **procedural abstraction**. When programming in C++ it might make more sense to call it *functional abstraction*. However, *procedure* is a more general term than *function* and computer scientists use it for all "function-like" sets of instructions, and so they prefer the term *procedural abstraction*. The term *abstraction* is intended to convey the idea that

Procedural Abstraction

When applied to a function definition, the principle of *procedural abstraction* means that your function should be written so that it can be used like a black box. This means that the programmer who uses the function should not need to look at the body of the function definition to see how the function works. The function declaration and the accompanying comment should be all the programmer needs to know in order to use the function. To ensure that your function definitions have this important property, you should strictly adhere to the following rules:

HOW TO WRITE A BLACK-BOX FUNCTION DEFINITION

- The function declaration comment should tell the programmer any and all conditions that are required of the arguments to the function and should describe the result of a function invocation.
- All variables used in the function body should be declared in the function body. (The formal parameters do not need to be declared, because they are listed in the function heading.)

when you use a function as a black box, you are abstracting away the details of the code contained in the function body. You can call this technique the *black box principle* or the *principle of procedural abstraction* or *information hiding*. The three terms mean the same thing. Whatever you call this principle, the important point is that you should use it when designing and writing your function definitions.

GLOBAL CONSTANTS AND GLOBAL VARIABLES

As we noted in Chapter 1, you can and should name constant values using the `const` modifier. For example, in Display 3.5 we used the `const` modifier to give a name to the rate of sales tax with the following declaration:

```
const double TAXRATE = 0.05; //5% sales tax
```

If this declaration is inside the definition of a function, as in Display 3.5, then the name `TAXRATE` is local to the function definition, which means that outside the definition of the function that contains the declaration, you can use the name `TAXRATE` for another named constant, or variable, or anything else.

On the other hand, if this declaration were to appear at the beginning of your program, outside the body of all the functions (and outside the body of the `main` part of your program), then the named constant is said to be a **global named constant** and the named constant can be used in any function definition that follows the constant declaration.

global named constant

Display 3.9 **A Global Named Constant** (part 1 of 2)


```
1  //Computes the area of a circle and the volume of a sphere.
2  //Uses the same radius for both calculations.
3  #include <iostream>
4  #include <cmath>
5  using namespace std;

6  const double PI = 3.14159;

7  double area(double radius);
8  //Returns the area of a circle with the specified radius.

9  double volume(double radius);
10 //Returns the volume of a sphere with the specified radius.

11 int main( )
12 {
13     double radiusOfBoth, areaOfCircle, volumeOfSphere;

14     cout << "Enter a radius to use for both a circle\n"
15          << "and a sphere (in inches): ";
16     cin >> radiusOfBoth;

17     areaOfCircle = area(radiusOfBoth);
18     volumeOfSphere = volume(radiusOfBoth);

19     cout << "Radius = " << radiusOfBoth << " inches\n"
20          << "Area of circle = " << areaOfCircle
21          << " square inches\n"
22          << "Volume of sphere = " << volumeOfSphere
23          << " cubic inches\n";

24     return 0;
25 }
26
27 double area(double radius)
28 {
29     return (PI * pow(radius, 2));
30 }

31 double volume(double radius)
32 {
33     return ((4.0/3.0) * PI * pow(radius, 3));
34 }
```

Display 3.9 **A Global Named Constant** (part 2 of 2)

SAMPLE DIALOGUE

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```

Display 3.9 shows a program with an example of a global named constant. The program asks for a radius and then computes both the area of a circle and the volume of a sphere with that radius, using the following formulas:

$$area = \pi \times (radius)^2$$
$$volume = (4/3) \times \pi \times (radius)^3$$

Both formulas include the constant π, which is approximately equal to 3.14159. The symbol π is the Greek letter called "pi." The program thus uses the following global named constant:

```
const double PI = 3.14159;
```

which appears outside the definition of any function (including outside the definition of `main`).

The compiler allows you wide latitude in where you place the declarations for your global named constants. To aid readability, however, you should place all your `include` directives together, all your global named constant declarations together in another group, and all your function declarations (function prototypes) together. We will follow standard practice and place all our global named constant declarations after our `include` and `using` directives and before our function declarations.

Placing all named constant declarations at the start of your program can aid readability even if the named constant is used by only one function. If the named constant might need to be changed in a future version of your program, it will be easier to find if it is at the beginning of your program. For example, placing the constant declaration for the sales tax rate at the beginning of an accounting program will make it easy to revise the program should the tax rate change.

global variable

It is possible to declare ordinary variables, without the `const` modifier, as **global variables**, which are accessible to all function definitions in the file. This is done similar to the way it is done for global named constants, except that the modifier `const` is not used in the variable declaration. However, there is seldom any need to use such global variables. Moreover, global variables can make a program harder to understand and maintain, so we urge you to avoid using them.

SELF-TEST EXERCISES

20. If you use a variable in a function definition, where should you declare the variable? In the function definition? In the `main` function? Any place that is convenient?
21. Suppose a function named `function1` has a variable named `sam` declared within the definition of `function1`, and a function named `function2` also has a variable named `sam` declared within the definition of `function2`. Will the program compile (assuming everything else is correct)? If the program will compile, will it run (assuming that everything else is correct)? If it runs, will it generate an error message when run (assuming everything else is correct)? If it runs and does not produce an error message when run, will it give the correct output (assuming everything else is correct)?
22. What is the purpose of the comment that accompanies a function declaration?
23. What is the principle of procedural abstraction as applied to function definitions?
24. What does it mean when we say the programmer who uses a function should be able to treat the function like a black box? (This question is very closely related to the previous question.)

BLOCKS

A variable declared inside a compound statement (that is, inside a pair of braces) is local to the compound statement. The name of the variable can be used for something else, such as the name of a different variable, outside the compound statement.

block

A compound statement with declarations is usually called a **block**. Actually, *block* and *compound statement* are two terms for the same thing. However, when we focus on variables declared within a compound statement, we normally use the term *block* rather than *compound statement* and we say that the variables declared within the block are *local to the block*.

If a variable is declared in a block, then the definition applies from the location of the declaration to the end of the block. This is usually expressed by saying that the *scope* of the declaration is from the location of the declaration to the end of the block. So if a variable is declared at the start of a block, its scope is the entire block. If the variable is declared part way through the block, the declaration does not take effect until the program reaches the location of the declaration (see Self-Test Exercise 25).

Notice that the body of a function definition is a block. Thus, a variable that is local to a function is the same thing as a variable that is local to the body of the function definition (which is a block).

Blocks

A *block* is some C++ code enclosed in braces. The variables declared in a block are local to the block, and so the variable names can be used outside the block for something else (such as being reused as the names for different variables).

NESTED SCOPES

Suppose you have one block nested inside another block, and suppose that one identifier is declared as a variable in each of these two blocks. These are two different variables with the same name. One variable exists only within the inner block and cannot be accessed outside that inner block. The other variable exists only in the outer block and cannot be accessed in the inner block. The two variables are distinct, so changes made to one of these variables will have no effect on the other of these two variables.

Scope Rule for Nested Blocks

If an identifier is declared as a variable in each of two blocks, one within the other, then these are two different variables with the same name. One variable exists only within the inner block and cannot be accessed outside of the inner block. The other variable exists only in the outer block and cannot be accessed in the inner block. The two variables are distinct, so changes made to one of these variables will have no effect on the other of these two variables.

TIP

Use Function Calls in Branching and Loop Statements

The `switch` statement and the `if-else` statement allow you to place several different statements in each branch. However, doing so can make the `switch` statement or `if-else` statement difficult to read. Rather than placing a compound statement in a branching statement, it is usually preferable to convert the compound statement to a function definition and place a function call in the branch. Similarly, if a loop body is large, it is preferable to convert the compound statement to a function definition and make the loop body a function call.

VARIABLES DECLARED IN A for LOOP

A variable may be declared in the heading of a `for` statement so that the variable is both declared and initialized at the start of the `for` statement. For example,

```
for (int n = 1; n <= 10; n++)
        sum = sum + n;
```

The ANSI/ISO C++ standard requires that a C++ compiler that claims compliance with the standard treat any declaration in a `for` loop initializer as if it were local to the body of the loop. Earlier C++ compilers did not do this. You should determine how your compiler treats variables declared in a `for` loop initializer. If portability is critical to your application, you should not write code that depends on this behavior. Eventually, all widely used C++ compilers will likely comply with this rule, but compilers presently available may or may not comply.

SELF-TEST EXERCISES

25. Though we urge you not to program using this style, we are providing an exercise that uses nested blocks to help you understand the scope rules. State the output that this code fragment would produce if embedded in an otherwise complete, correct program.

```
{
   int x = 1;
   cout << x << endl;
   {
      cout << x << endl;
      int x = 2;
      cout << x << endl;
      {
         cout << x << endl;
         int x = 3;
         cout << x << endl;
      }
      cout << x << endl;
   }
   cout << x << endl;
}
```

CHAPTER SUMMARY

- There are two kinds of functions in C++: functions that return a value and `void` functions.
- A function should be defined so that it can be used as a black box. The programmer who uses the function should not need to know any details about how the function is coded. All the programmer should need to know is the function declaration and the accompanying comment that describes the value returned. This rule is sometimes called the principle of procedural abstraction.
- A good way to write a function declaration comment is to use a precondition and a postcondition. The precondition states what is assumed to be true when the function is called. The postcondition describes the effect of the function call; that is, the postcondition tells what will be true after the function is executed in a situation in which the precondition holds.
- A variable that is declared in a function definition is said to be local to the function.
- A formal parameter is a kind of placeholder that is filled in with a function argument when the function is called. The details on this "filling in" process are covered in Chapter 4.

ANSWERS TO SELF-TEST EXERCISES

1.
```
4.0  4.0  8.0
8.0  8.0  1.21
3    3    0
3.0  3.5  3.5
6.0  6.0  5.0
5.0  4.5  4.5
3    3.0  3.0
```

2. a. `sqrt(x + y)`

   b. `pow(x, y + 7)`

   c. `sqrt(area + fudge)`

   d. `sqrt(time+tide)/nobody`

   e. `(–b ± sqrt(b*b – 4*a*c))/(2*a)`

   f. `abs(x – y)` or `labs(x – y)` or `fabs(x – y)`

3.
```
#include <iostream>
#include <cmath>
using namespace std;
int main( )
{
    int i;
    for (i = 1; i <= 10; i++)
    cout << "The square root of  " << i
         << " is " << sqrt(i) << endl;
    return 0;
}
```

4. The argument is given to the operating system. As far as your C++ program is concerned, you can use any `int` value as the argument. By convention, however, `1` is used for a call to `exit` that is caused by an error, and `0` is used in other cases.

5. `(5 + (rand( ) % 6))`

6.
```
#include <iostream>
#include <cstdlib>
using namespace std;

int main( )
{
  cout << "Enter a nonnegative integer to use as the\n"
       << "seed for the random number generator: ";
  unsigned int seed;
  cin >> seed;
  srand(seed);

  cout << "Here are ten random probabilities:\n";
  int i;
  for (i = 0; i < 10; i++)
  cout << ((RAND_MAX - rand( ))/static_cast<double>(RAND_MAX))
       << endl;

  return 0;
}
```

7. `Wow`

8. The function declaration is

```
int sum(int n1, int n2, int n3);
//Returns the sum of n1, n2, and n3.
```

The function definition is

```
int sum(int n1, int n2, int n3)
{
    return (n1 + n2 + n3);
}
```

9. The function declaration is

```
char positiveTest(double number);
//Returns 'P' if number is positive.
//Returns 'N' if number is negative or zero.
```

The function definition is

```
char positiveTest(double number)
{
    if (number > 0)
        return 'P';
    else
        return 'N';
}
```

10. No, a function definition cannot appear inside the body of another function definition.

11. Predefined functions and user-defined functions are invoked (called) in the same way.

12.
```
bool inOrder(int n1, int n2, int n3)
{
     return ((n1 <= n2) && (n2 <= n3));
}
```

13.
```
bool even(int n)
{
     return ((n % 2) == 0);
}
```

14.
```
bool isDigit(char ch)
{
   return ('0' <= ch) && (ch <= '9');
}
```

15.
```
Hello
Goodbye
One more time:
Hello
End of program.
```

16. If you omitted the `return` statement in the function definition for `iceCreamDivision` in Display 3.7, the program would compile and run. However, if you input zero for the number of customers, then the program would produce a run-time error because of a division by zero.

17.
```
#include <iostream>
using namespace std;
void productOut(int n1, int n2, int n3);

int main( )
{
    int num1, num2, num3;
    cout << "Enter three integers: ";
    cin >> num1 >> num2 >> num3;
    productOut(num1, num2, num3);
    return 0;
}

void productOut(int n1, int n2, int n3)
{
    cout << "The product of the three numbers "
         << n1 << ", " << n2 << ", and "
         << n3 << " is " << (n1*n2*n3) << endl;
}
```

18. These answers are system-dependent.

19.
```
double sqrt(double n);
//Precondition: n >= 0.
//Returns the square root of n.
```

You can rewrite the second comment line as the following if you prefer, but the version above is the usual form used for a function that returns a value:

```
//Postcondition: Returns the square root of n.
```

20. If you use a variable in a function definition, you should declare the variable in the body of the function definition.

21. Everything will be fine. The program will compile (assuming everything else is correct). The program will run (assuming that everything else is correct). The program will not generate an error message when run (assuming everything else is correct). The program will give the correct output (assuming everything else is correct).

22. The comment explains what action the function takes, including any value returned, and gives any other information that you need to know in order to use the function.

23. The principle of procedural abstraction says that a function should be written so that it can be used like a black box. This means that the programmer who uses the function need not look at the body of the function definition to see how the function works. The function declaration and accompanying comment should be all the programmer needs in order to use the function.

24. When we say that the programmer who uses a function should be able to treat the function like a black box, we mean the programmer should not need to look at the body of the function definition to see how the function works. The function declaration and accompanying comment should be all the programmer needs in order to use the function.

25. It helps to slightly change the code fragment to understand to which declaration each usage resolves. The code has three different variables named `x`. In the following we have renamed these three variables `x1`, `x2`, and `x3`. The output is given in the comments.

```
{
   int x1 = 1;// output in this column
   cout << x1 << endl;// 1<new line>
   {
      cout << x1 << endl;// 1<new line>
      int x2 = 2;
      cout << x2 << endl;// 2<new line>
      {
         cout << x2 << endl;// 2<new line>
         int x3 = 3;
         cout << x3 << endl;// 3<new line>
      }
      cout << x2 << endl;// 2<new line>
   }
   cout << x1 << endl;// 1<new line>
}
```

PROGRAMMING PROJECTS

Many of these Programming Projects can be solved using AW's CodeMate. To access these please go to: www.aw-bc.com/codemate.

1. A liter is 0.264179 gallons. Write a program that will read in the number of liters of gasoline consumed by the user's car and the number of miles traveled by the car and will then output the number of miles per gallon the car delivered. Your program should allow the user to repeat this calculation as often as the user wishes. Define a function to compute the number of miles per gallon. Your program should use a globally defined constant for the number of liters per gallon.

2. Write a program to gauge the rate of inflation for the past year. The program asks for the price of an item (such as a hot dog or a one-carat diamond) both one year ago and

today. It estimates the inflation rate as the difference in price divided by the year-ago price. Your program should allow the user to repeat this calculation as often as the user wishes. Define a function to compute the rate of inflation. The inflation rate should be a value of type `double` giving the rate as a percentage, for example 5.3 for 5.3%.

3. Enhance your program from the previous exercise by having it also print out the estimated price of the item in one and in two years from the time of the calculation. The increase in cost over one year is estimated as the inflation rate times the price at the start of the year. Define a second function to determine the estimated cost of an item in a specified number of years, given the current price of the item and the inflation rate as arguments.

4. The gravitational attractive force between two bodies with masses m_1 and m_2 separated by a distance d is given by the following formula:

$$F = \frac{Gm_1 m_2}{d^2}$$

where G is the universal gravitational constant:

$$G = 6.673 \times 10^{-8}\ \text{cm}^3/(\text{g} \bullet \text{sec}^2)$$

Write a function definition that takes arguments for the masses of two bodies and the distance between them and returns the gravitational force between them. Since you will use the above formula, the gravitational force will be in dynes. One dyne equals a

$$\text{g} \bullet \text{cm/sec}^2$$

You should use a globally defined constant for the universal gravitational constant. Embed your function definition in a complete program that computes the gravitational force between two objects given suitable inputs. Your program should allow the user to repeat this calculation as often as the user wishes.

5. Write a program that asks for the user's height, weight, and age, and then computes clothing sizes according to the following formulas.
   - Hat size = weight in pounds divided by height in inches and all that multiplied by 2.9.
   - Jacket size (chest in inches) = height times weight divided by 288 and then adjusted by adding one-eighth of an inch for each 10 years over age 30. (Note that the adjustment only takes place after a full 10 years. So, there is no adjustment for ages 30 through 39, but one-eighth of an inch is added for age 40.)
   - Waist in inches = weight divided by 5.7 and then adjusted by adding one-tenth of an inch for each 2 years over age 28. (Note that the adjustment only takes place after a full 2 years. So, there is no adjustment for age 29, but one-tenth of an inch is added for age 30.)

   Use functions for each calculation. Your program should allow the user to repeat this calculation as often as the user wishes.

6. Write a function that computes the average and standard deviation of four scores. The standard deviation is defined to be the square root of the average of the four values:

$(s_i - a)^2$, where a is the average of the four scores s_1, s_2, s_3, and s_4. The function will have six parameters and will call two other functions. Embed the function in a program that allows you to test the function again and again until you tell the program you are finished.

7. In cold weather, meteorologists report an index called the *wind chill factor*, which takes into account the wind speed and the temperature. The index provides a measure of the chilling effect of wind at a given air temperature. Wind chill may be approximated by the following formula:

$$W = 33 - \frac{(10\sqrt{v} - v + 10.5)(33 - t)}{23.1}$$

where

v = wind speed in m/sec

t = temperature in degrees Celsius: $t <= 10$

W = wind chill index (in degrees Celsius)

Write a function that returns the wind chill index. Your code should ensure that the restriction on the temperature is not violated. Look up some weather reports in back issues of a newspaper in your library and compare the wind chill index you calculate with the result reported in the newspaper.

8. Write a program that outputs all 99 stanzas of the "Ninety-Nine Bottles of Beer on the Wall" song. Your program should print the number of bottles in English, not as a number:

Ninety-nine bottles of beer on the wall,
Ninety-nine bottles of beer,
Take one down, pass it around,
Ninety-eight bottles of beer on the wall.
…
One bottle of beer on the wall,
One bottle of beer,
Take one down, pass it around,
Zero bottles of beer on the wall.

Your program should not use ninety-nine different output statements!

9. In the game of craps, a "Pass Line" bet proceeds as follows. The first roll of the two, six-sided dice in a craps round is called the "come out roll." The bet immediately wins when the come out roll is 7 or 11, and loses when the come out roll is 2, 3, or 12. If 4, 5, 6, 8, 9, or 10 is rolled on the come out roll, that number becomes "the point." The player keeps rolling the dice until either 7 or the point is rolled. If the point is rolled first, then the player wins the bet. If the player rolls a 7 first, then the player loses.

Write a program that plays craps using those rules so that it simulates a game without human input. Instead of asking for a wager, the program should calculate whether the player would win or lose. Create a function that simulates rolling the two dice and returns the sum. Add a loop so that the program plays 10,000 games. Add counters that

count how many times the player wins, and how many times the player loses. At the end of the 10,000 games, compute the probability of winning, as Wins / (Wins + Losses), and output this value. Over the long run, who is going to win more games of craps, you or the house?

10. One way to estimate the height of a child is to use the following formula, which uses the height of the parents:

$H_{male\_child} = ((H_{mother} \times 13 / 12) + H_{father})/2$

$H_{female\_child} = ((H_{father} \times 12 / 13) + H_{mother})/2$

All heights are in inches. Write a function that takes as input parameters the gender of the child, height of the mother in inches, and height of the father in inches, and outputs the estimated height of the child in inches. Embed your function in a program that allows you to test the function over and over again until telling the program to exit. The user should be able to input the heights in feet and inches, and the program should output the estimated height of the child in feet and inches. Use the integer data type to store the heights.