CHAPTER 6

Functions

TOPICS

6.1 Focus on Software Engineering: Modular Programming

CONCEPT: A program may be broken up into manageable functions.

A function is a collection of statements that performs a specific task. So far you have experienced functions in two ways: (1) you have created a function named `main` in every program you've written, and (2) you have used library functions such as `pow` and `strcmp`. In this chapter you will learn how to create your own functions that can be used like library functions.

Functions are commonly used to break a problem down into small manageable pieces. Instead of writing one long function that contains all of the statements necessary to solve a problem, several small functions that each solve a specific part of the problem can be written. These small functions can then be executed in the desired order to solve the problem. This approach is sometimes called *divide and conquer* because a large problem is

divided into several smaller problems that are easily solved. Figure 6-1 illustrates this idea by comparing two programs: one that uses a long complex function containing all of the statements necessary to solve a problem, and another that divides a problem into smaller problems, each of which are handled by a separate function.

Figure 6-1

This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.

```
int main()
{
   statement;
   statement;
   statement;
}
```

main function

```
void function2()
{
   statement;
   statement;
   statement;
}
```

function 2

```
void function3()
{
   statement;
   statement;
   statement;
}
```

function 3

```
void function4()
{
   statement;
   statement;
   statement;
}
```

function 4

Another reason to write functions is that they simplify programs. If a specific task is performed in several places in a program, a function can be written once to perform that task, and then be executed anytime it is needed. This benefit of using functions is known as *code reuse* because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

6.2 Defining and Calling Functions

CONCEPT: **A function call is a statement that causes a function to execute. A function definition contains the statements that make up the function.**

When creating a function, you must write its *definition*. All function definitions have the following parts:

Return type: A function can send a value to the part of the program that executed it. The return type is the data type of the value that is sent from the function.

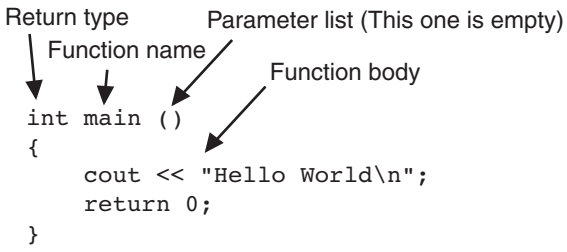
Name: You should give each function a descriptive name. In general, the same rules that apply to variable names also apply to function names.

Parameter list: The program can send data into a function. The parameter list is a list of variables that hold the values being passed to the function.

Body: The body of a function is the set of statements that perform the function's operation. They are enclosed in a set of braces.

Figure 6-2 shows the definition of a simple function with the various parts labeled.

Figure 6-2



The line in the definition that reads `int main()` is called the *function header*.

`void` Functions

You already know that a function can return a value. The `main` function in all of the programs you have seen in this book is declared to return an `int` value to the operating system. The `return 0;` statement causes the value 0 to be returned when the `main` function finishes executing.

It isn't necessary for all functions to return a value, however. Some functions simply perform one or more statements, which follows terminate. These are called *`void` functions*. The `displayMessage` function, which follows, is an example.

```
void displayMessage()
{
   cout << "Hello from the function displayMessage.\n";
}
```

The function's name is `displayMessage`. This name gives an indication of what the function does: It displays a message. You should always give functions names that reflect their purpose. Notice that the function's return type is `void`. This means the function does not return a value to the part of the program that executed it. Also notice the function has no `return` statement. It simply displays a message on the screen and exits.

Calling a Function

A function is executed when it is *called*. Function `main` is called automatically when a program starts, but all other functions must be executed by *function call* statements. When a function is called, the program branches to that function and executes the statements in its body. Let's look at Program 6-1, which contains two functions: `main` and `displayMessage`.

Program 6-1

```
1  // This program has two functions: main and displayMessage
2  #include <iostream>
3  using namespace std;
4
5  //***************************************
6  // Definition of function displayMessage  *
7  // This function displays a greeting.     *
8  //***************************************
9
10 void displayMessage()
11 {
12     cout << "Hello from the function displayMessage.\n";
13 }
14
15 //***************************************
16 // Function main                          *
17 //***************************************
18
19 int main()
20 {
21     cout << "Hello from main.\n";
22     displayMessage();
23     cout << "Back in function main again.\n";
24     return 0;
25 }
```

Program Output

```
Hello from main.
Hello from the function displayMessage.
Back in function main again.
```

The function `displayMessage` is called by the following statement in line 22:

```
displayMessage();
```

This statement is the function call. It is simply the name of the function followed by a set of parentheses and a semicolon. Let's compare this with the function header:

Function Header ⟶ `void displayMessage()`
Function Call ⟶ `displayMessage();`

The function header is part of the function definition. It declares the function's return type, name, and parameter list. It is not terminated with a semicolon because the definition of the function's body follows it.

The function call is a statement that executes the function, so it is terminated with a semicolon like all other C++ statements. The return type is not listed in the function call, and, if the program is not passing data into the function, the parentheses are left empty.

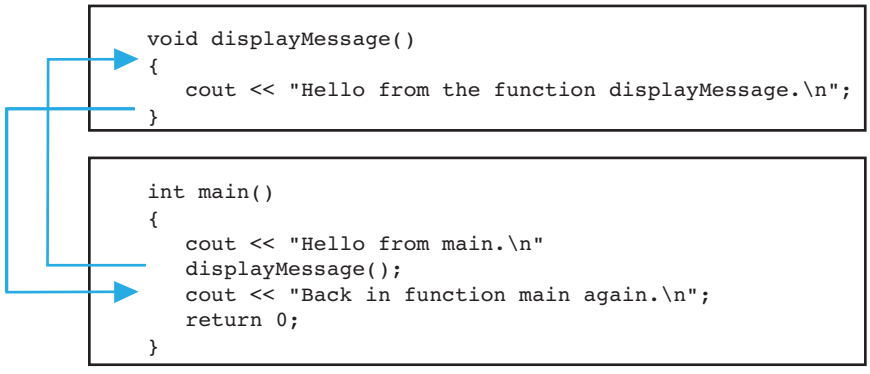
NOTE: Later in this chapter you will see how data can be passed into a function by being listed inside the parentheses.

Even though the program starts executing at `main`, the function `displayMessage` is defined first. This is because the compiler must know the function's return type, the number of parameters, and the type of each parameter before the function is called. One way to ensure the compiler will know this information is to place the function definition before all calls to that function. (Later you will see an alternative, preferred method of accomplishing this.)

NOTE: You should always document your functions by writing comments that describe what they do. These comments should appear just before the function definition.

Notice how Program 6-1 flows. It starts, of course, in function `main`. When the call to `displayMessage` is encountered, the program branches to that function and performs its statements. Once `displayMessage` has finished executing, the program branches back to function `main` and resumes with the line that follows the function call. This is illustrated in Figure 6-3.

Figure 6-3



Function call statements may be used in control structures like loops, `if` statements, and `switch` statements. Program 6-2 places the `displayMessage` function call inside a loop.

Program 6-2

```
1  // The function displayMessage is repeatedly called from a loop.
2  #include <iostream>
3  using namespace std;
4
5  //*****************************************
6  // Definition of function displayMessage  *
7  // This function displays a greeting.     *
8  //*****************************************
9
10 void displayMessage()
11 {
12     cout << "Hello from the function displayMessage.\n";
13 }
14
15 //*****************************************
16 // Function main                          *
17 //*****************************************
18
19 int main()
20 {
21     cout << "Hello from main.\n";
22     for (int count = 0; count < 5; count++)
23         displayMessage(); // Call displayMessage
24     cout << "Back in function main again.\n";
25     return 0;
26 }
```

Program Output

```
Hello from main.
Hello from the function displayMessage.
Hello from the function displayMessage.
Hello from the function displayMessage.
Hello from the function displayMessage.
Hello from the function displayMessage.
Back in function main again.
```

It is possible to have many functions and function calls in a program. Program 6-3 has three functions: `main`, `first`, and `second`.

Program 6-3

```
1  // This program has three functions: main, first, and second.
2  #include <iostream>
3  using namespace std;
4
```

```
5  //****************************************
6  // Definition of function first          *
7  // This function displays a message.     *
8  //****************************************
9
10 void first()
11 {
12     cout << "I am now inside the function first.\n";
13 }
14
15 //****************************************
16 // Definition of function second         *
17 // This function displays a message.     *
18 //****************************************
19
20 void second()
21 {
22     cout << "I am now inside the function second.\n";
23 }
24
25 //****************************************
26 // Function main                         *
27 //****************************************
28
29 int main()
30 {
31     cout << "I am starting in function main.\n";
32     first();    // Call function first
33     second();   // Call function second
34     cout << "Back in function main again. \n";
35     return 0;
36 }
```

Program Output

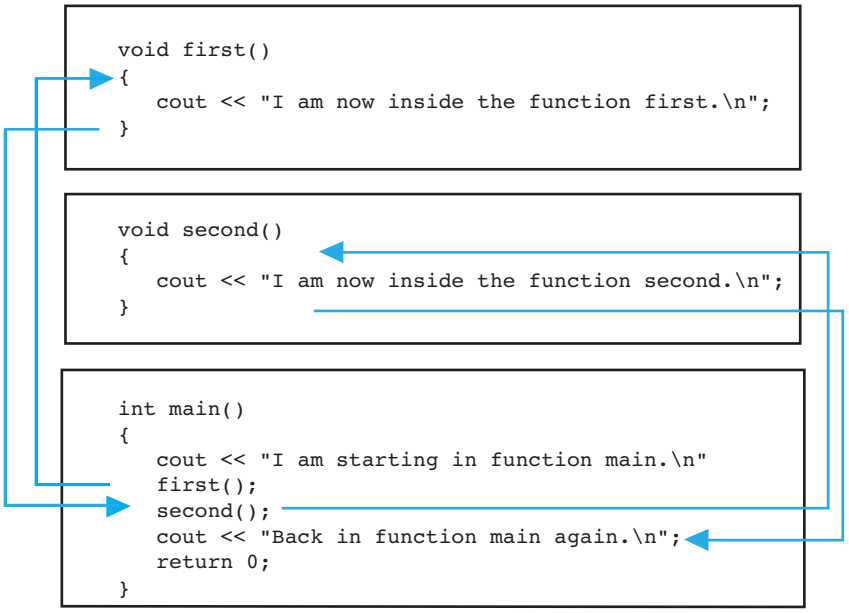
```
I am starting in function main.
I am now inside the function first.
I am now inside the function second.
Back in function main again.
```

In lines 32 and 33 of Program 6-3, function `main` contains a call to `first` and a call to `second`:

```
first();
second();
```

Each call statement causes the program to branch to a function and then back to `main` when the function is finished. Figure 6-4 illustrates the paths taken by the program.

Figure 6-4



Functions may also be called in a hierarchical, or layered, fashion. This is demonstrated by Program 6-4, which has three functions: `main`, `deep`, and `deeper`.

Program 6-4

```
1  // This program has three functions: main, deep, and deeper
2  #include <iostream>
3  using namespace std;
4
5  //****************************************
6  // Definition of function deeper         *
7  // This function displays a message.     *
8  //****************************************
9
10 void deeper()
11 {
12     cout << "I am now inside the function deeper.\n";
13 }
14
15 //****************************************
16 // Definition of function deep           *
17 // This function displays a message.     *
18 //****************************************
19
```

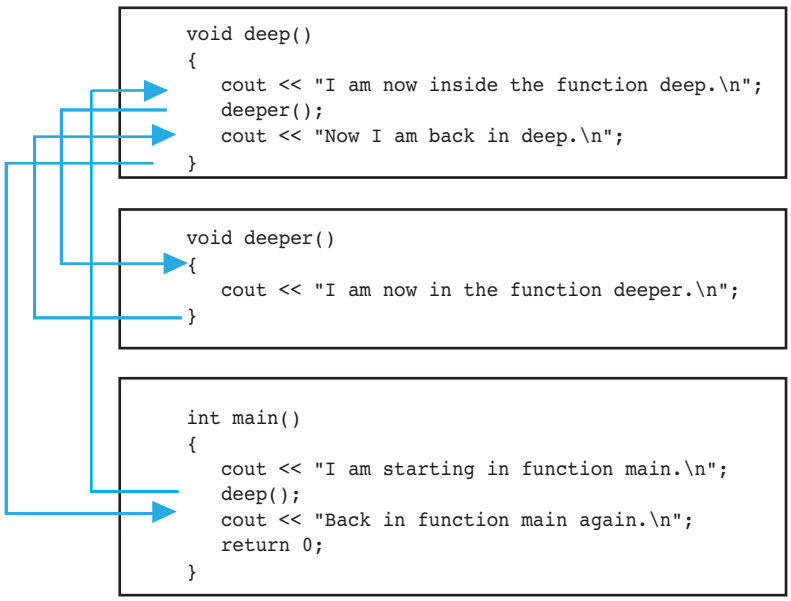
```
20  void deep()
21  {
22       cout << "I am now inside the function deep.\n";
23       deeper();     // Call function deeper
24       cout << "Now I am back in deep.\n";
25  }
26
27  //****************************************
28  // Function main                         *
29  //****************************************
30
31  int main()
32  {
33       cout << "I am starting in function main.\n";
34       deep();      // Call function deep
35       cout << "Back in function main again.\n";
36       return 0;
37  }
```

Program Output

```
I am starting in function main.
I am now inside the function deep.
I am now inside the function deeper.
Now I am back in deep.
Back in function main again.
```

In Program 6-4, function `main` only calls the function `deep`. In turn, `deep` calls `deeper`. The paths taken by the program are shown in Figure 6-5.

Figure 6-5



Checkpoint

6.1 Is the following a function header or a function call?

```
calcTotal();
```

6.2 Is the following a function header or a function call?

```
void showResults()
```

6.3 What will the output of the following program be if the user enters 10?

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

void func1()
{
   cout << "Able was I\n";
}

void func2()
{
   cout << "I saw Elba\n";
}

int main()
{
   int input;
   cout << "Enter a number: ";
   cin >> input;
   if (input < 10)
   {
      func1();
      func2();
   }
   else
   {
      func2();
      func1();
   }
   return 0;
}
```

6.4 The following program skeleton determines whether a person qualifies for a credit card. To qualify, the person must have worked on his or her current job for at least two years and make at least $17,000 per year. Finish the program by writing the definitions of the functions `qualify` and `noQualify`. The function `qualify` should explain that the applicant qualifies for the card and that the annual interest rate is 12%. The function `noQualify` should explain that the applicant does not qualify for the card and give a general explanation why.

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

// You must write definitions for the two functions qualify
// and noQualify.
```

```
int main()
{
   double salary;
   int years;

   cout << "This program will determine if you qualify\n";
   cout << "for our credit card.\n";
   cout << "What is your annual salary? ";
   cin >> salary;
   cout << "How many years have you worked at your ";
   cout << "current job? ";
   cin >> years;
   if (salary >= 17000.0 && years >= 2)
      qualify();
   else
      noQualify();
   return 0;
}
```

6.3 Function Prototypes

CONCEPT: **A function prototype eliminates the need to place a function definition before all calls to the function.**

Before the compiler encounters a call to a particular function, it must already know the function's return type, the number of parameters it uses, and the type of each parameter. (You will learn how to use parameters in the next section.)

One way of ensuring that the compiler has this information is to place the function definition before all calls to that function. This was the approach taken in Programs 6-1, 6-2, 6-3, and 6-4. Another method is to declare the function with a *function prototype*. Here is a prototype for the `displayMessage` function in Program 6-1:

```
void displayMessage();
```

The prototype looks similar to the function header, except there is a semicolon at the end. The statement above tells the compiler that the function `displayMessage` has a `void` return type (it doesn't return a value) and uses no parameters.

NOTE: Function prototypes are also known as *function declarations*.

WARNING! You must place either the function definition or either/the function prototype ahead of all calls to the function. Otherwise the program will not compile.

Function prototypes are usually placed near the top of a program so the compiler will encounter them before any function calls. Program 6-5 is a modification of Program 6-3. The definitions of the functions `first` and `second` have been placed after `main`, and a function prototype has been placed after the `using namespace std` statement.

Program 6-5

```
1  // This program has three functions: main, first, and second.
2  #include <iostream>
3  using namespace std;
4
5  // Function Prototypes
6  void first();
7  void second();
8
9  int main()
10 {
11     cout << "I am starting in function main.\n";
12     first();     // Call function first
13     second();    // Call function second
14     cout << "Back in function main again.\n";
15     return 0;
16 }
17
18 //*************************************
19 // Definition of function first.      *
20 // This function displays a message.  *
21 //*************************************
22
23 void first()
24 {
25     cout << "I am now inside the function first.\n";
26 }
27
28 //*************************************
29 // Definition of function second.     *
30 // This function displays a message.  *
31 //*************************************
32
33 void second()
34 {
35     cout << "I am now inside the function second.\n";
36 }
```

Program Output

(The program's output is the same as the output of Program 6-3.)

When the compiler is reading Program 6-5, it encounters the calls to the functions `first` and `second` in lines 12 and 13 before it has read the definition of those functions. Because of the function prototypes, however, the compiler already knows the return type and parameter information of `first` and `second`.

NOTE: Although some programmers make `main` the last function in the program, many prefer it to be first because it is the program's starting point.

6.4 Sending Data into a Function

CONCEPT: **When a function is called, the program may send values into the function.**

VideoNote
Functions and Arguments

Values that are sent into a function are called *arguments*. You're already familiar with how to use arguments in a function call. In the following statement the function `pow` is being called and two arguments, 2.0 and 4.0, are passed to it:

```
result = pow(2.0, 4.0);
```

By using *parameters*, you can design your own functions that accept data this way. A parameter is a special variable that holds a value being passed into a function. Here is the definition of a function that uses a parameter:

```
void displayValue(int num)
{
   cout << "The value is " << num << endl;
}
```

Notice the integer variable definition inside the parentheses `(int num)`. The variable `num` is a parameter. This enables the function `displayValue` to accept an integer value as an argument. Program 6-6 is a complete program using this function.

NOTE: In this text, the values that are passed into a function are called arguments, and the variables that receive those values are called parameters. There are several variations of these terms in use. Some call the arguments *actual parameters* and call the parameters *formal parameters*. Others use the terms *actual argument* and *formal argument*. Regardless of which set of terms you use, it is important to be consistent.

Program 6-6

```
1  // This program demonstrates a function with a parameter.
2  #include <iostream>
3  using namespace std;
4
5  // Function Prototype
6  void displayValue(int);
7
8  int main()
9  {
10     cout << "I am passing 5 to displayValue.\n";
11     displayValue(5);  // Call displayValue with argument 5
12     cout << "Now I am back in main.\n";
13     return 0;
14 }
15
```

(program continues)

Program 6-6 *(continued)*

```
16  //*********************************************************
17  // Definition of function displayValue.                   *
18  // It uses an integer parameter whose value is displayed. *
19  //*********************************************************
20
21  void displayValue(int num)
22  {
23      cout << "The value is " << num << endl;
24  }
```

Program Output

```
I am passing 5 to displayValue.
The value is 5
Now I am back in main.
```

First, notice the function prototype for `displayValue` in line 6:

```
void displayValue(int);
```

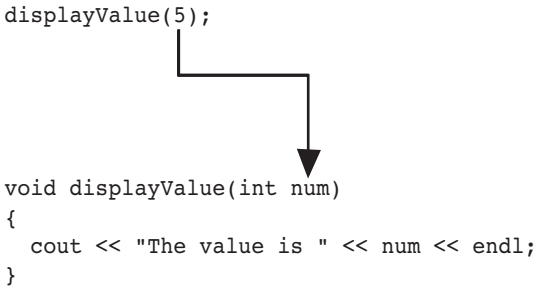
It is not necessary to list the name of the parameter variable inside the parentheses. Only its data type is required. The function prototype shown above could optionally have been written as:

```
void displayValue(int num);
```

However, the compiler ignores the name of the parameter variable in the function prototype.

In `main`, the `displayValue` function is called with the argument 5 inside the parentheses. The number 5 is passed into `num`, which is `displayValue`'s parameter. This is illustrated in Figure 6-6.

Figure 6-6



Any argument listed inside the parentheses of a function call is copied into the function's parameter variable. In essence, parameter variables are initialized to the value of their corresponding arguments. Program 6-7 shows the function `displayValue` being called several times with a different argument being passed each time.

Program 6-7

```
1  // This program demonstrates a function with a parameter.
2  #include <iostream>
3  using namespace std;
4
5  // Function Prototype
6  void displayValue(int);
7
8  int main()
9  {
10     cout << "I am passing several values to displayValue.\n";
11     displayValue(5);  // Call displayValue with argument 5
12     displayValue(10); // Call displayValue with argument 10
13     displayValue(2);  // Call displayValue with argument 2
14     displayValue(16); // Call displayValue with argument 16
15     cout << "Now I am back in main.\n";
16     return 0;
17 }
18
19 //********************************************************
20 // Definition of function displayValue.                  *
21 // It uses an integer parameter whose value is displayed. *
22 //********************************************************
23
24 void displayValue(int num)
25 {
26     cout << "The value is " << num << endl;
27 }
```

Program Output

```
I am passing several values to displayValue.
The value is 5
The value is 10
The value is 2
The value is 16
Now I am back in main.
```

WARNING! When passing a variable as an argument, simply write the variable name inside the parentheses of the function call. Do not write the data type of the argument variable in the function call. For example, the following function call will cause an error:

```
displayValue(int x); // Error!
```

The function call should appear as

```
displayValue(x);     // Correct
```

Each time the function is called in Program 6-7, `num` takes on a different value. Any expression whose value could normally be assigned to `num` may be used as an argument. For example, the following function call would pass the value 8 into `num`:

```
displayValue(3 + 5);
```

If you pass an argument whose type is not the same as the parameter's type, the argument will be promoted or demoted automatically. For instance, the argument in the following function call would be truncated, causing the value 4 to be passed to num:

```
displayValue(4.7);
```

Often, it's useful to pass several arguments into a function. Program 6-8 shows the definition of a function with three parameters.

Program 6-8

```
1  // This program demonstrates a function with three parameters.
2  #include <iostream>
3  using namespace std;
4
5  // Function Prototype
6  void showSum(int, int, int);
7
8  int main()
9  {
10     int value1, value2, value3;
11
12     // Get three integers.
13     cout << "Enter three integers and I will display ";
14     cout << "their sum: ";
15     cin >> value1 >> value2 >> value3;
16
17     // Call showSum passing three arguments.
18     showSum(value1, value2, value3);
19     return 0;
20 }
21
22 //*************************************************************
23 // Definition of function showSum.                            *
24 // It uses three integer parameters. Their sum is displayed. *
25 //*************************************************************
26
27 void showSum(int num1, int num2, int num3)
28 {
29     cout << (num1 + num2 + num3) << endl;
30 }
```

Program Output with Example Input Shown in Bold

```
Enter three integers and I will display their sum: 4 8 7 [Enter]
19
```

In the function header for showSum, the parameter list contains three variable definitions separated by commas:

```
void showSum(int num1, int num2, int num3)
```

WARNING! Each parameter variable in a parameter list must have a data type listed before its name. For example, a compiler error would occur if the parameter list for the `showSum` function were defined as shown in the following header:

```
void showSum(int num1, num2, num3) // Error!
```

A data type for all three of the parameter variables must be listed, as shown here:

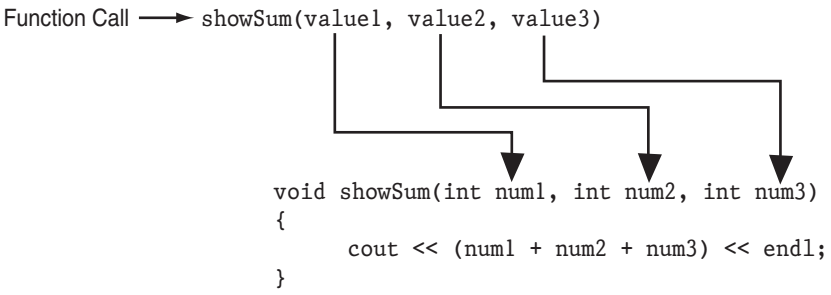
```
void showSum(int num1, int num2, int num3) // Correct
```

In the function call in line 18, the variables `value1`, `value2`, and `value3` are passed as arguments:

```
showSum(value1, value2, value3);
```

When a function with multiple parameters is called, the arguments are passed to the parameters in order. This is illustrated in Figure 6-7.

Figure 6-7



The following function call will cause 5 to be passed into the `num1` parameter, 10 to be passed into `num2`, and 15 to be passed into `num3`:

```
showSum(5, 10, 15);
```

However, the following function call will cause 15 to be passed into the `num1` parameter, 5 to be passed into `num2`, and 10 to be passed into `num3`:

```
showSum(15, 5, 10);
```

NOTE: The function prototype must list the data type of each parameter.

NOTE: Like all variables, parameters have a scope. The scope of a parameter is limited to the body of the function that uses it.

6.5 Passing Data by Value

CONCEPT: **When an argument is passed into a parameter, only a copy of the argument's value is passed. Changes to the parameter do not affect the original argument.**

As you've seen in this chapter, parameters are special-purpose variables that are defined inside the parentheses of a function definition. They are separate and distinct from the arguments that are listed inside the parentheses of a function call. The values that are stored in the parameter variables are copies of the arguments. Normally, when a parameter's value is changed inside a function, it has no effect on the original argument. Program 6-9 demonstrates this concept.

Program 6-9

```
1  // This program demonstrates that changes to a function parameter
2  // have no effect on the original argument.
3  #include <iostream>
4  using namespace std;
5
6  // Function Prototype
7  void changeMe(int);
8
9  int main()
10 {
11     int number = 12;
12
13     // Display the value in number.
14     cout << "number is " << number << endl;
15
16     // Call changeMe, passing the value in number
17     // as an argument.
18     changeMe(number);
19
20     // Display the value in number again.
21     cout << "Now back in main again, the value of ";
22     cout << "number is " << number << endl;
23     return 0;
24 }
25
26 //*************************************************************
27 // Definition of function changeMe.                           *
28 // This function changes the value of the parameter myValue.  *
29 //*************************************************************
```

```
30
31  void changeMe(int myValue)
32  {
33       // Change the value of myValue to 0.
34       myValue = 0;
35
36       // Display the value in myValue.
37       cout << "Now the value is " << myValue << endl;
38  }
```

Program Output

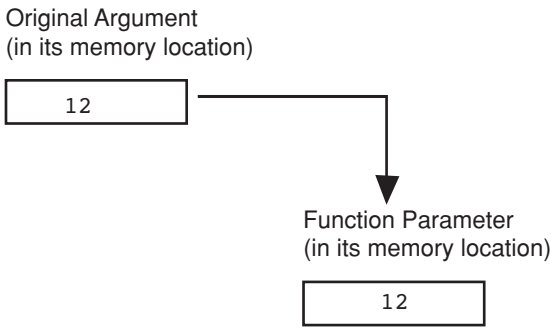
```
number is 12
Now the value is 0
Now back in main again, the value of number is 12
```

Even though the parameter variable `myValue` is changed in the `changeMe` function, the argument `number` is not modified. The `myValue` variable contains only a copy of the `number` variable.

The `changeMe` function does not have access to the original argument. When only a copy of an argument is passed to a function, it is said to be *passed by value*. This is because the function receives a copy of the argument's value and does not have access to the original argument.

Figure 6-8 illustrates that a parameter variable's storage location in memory is separate from that of the original argument.

Figure 6-8



NOTE: Later in this chapter you will learn ways to give a function access to its original arguments.

6.6 Focus on Software Engineering: Using Functions in a Menu-Driven Program

CONCEPT: **Functions are ideal for use in menu-driven programs. When the user selects an item from a menu, the program can call the appropriate function.**

In Chapters 4 and 5 you saw a menu-driven program that calculates the charges for a health club membership. Program 6-10 shows the program redesigned as a modular program. A *modular* program is broken up into functions that perform specific tasks.

Program 6-10

```
1  // This is a menu-driven program that makes a function call
2  // for each selection the user makes.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // Function prototypes
8  void showMenu();
9  void showFees(double, int);
10
11 int main()
12 {
13     int choice;       // To hold a menu choice
14     int months;       // To hold a number of months
15
16     // Constants for the menu choices
17     const int ADULT_CHOICE = 1,
18               CHILD_CHOICE = 2,
19               SENIOR_CHOICE = 3,
20               QUIT_CHOICE = 4;
21
22     // Constants for membership rates
23     const double ADULT = 40.0,
24                  CHILD = 20.0;
25                  SENIOR = 30.0,
26
27     // Set up numeric output formatting.
28     cout << fixed << showpoint << setprecision(2);
29
30     do
31     {
32          // Display the menu and get the user's choice.
33          showMenu();
34          cin >> choice;
35
36          // Validate the menu selection.
37          while (choice < ADULT_CHOICE || choice > QUIT_CHOICE)
38          {
```

```
39              cout << "Please enter a valid menu choice: ";
40              cin >> choice;
41          }
42
43          // If the user does not want to quit, proceed.
44          if (choice != QUIT_CHOICE)
45          {
46              // Get the number of months.
47              cout << "For how many months? ";
48              cin >> months;
49
50              // Display the membership fees.
51              switch (choice)
52              {
53                  case ADULT_CHOICE:
54                       showFees(ADULT, months);
55                       break;
56                  case CHILD_CHOICE:
57                       showFees(CHILD, months);
58                       break;
59                  case SENIOR_CHOICE:
60                       showFees(SENIOR, months);
61              }
62          }
63      } while (choice != QUIT_CHOICE);
64      return 0;
65  }
66
67  //*****************************************************************
68  // Definition of function showMenu which displays the menu.      *
69  //*****************************************************************
70
71  void showMenu()
72  {
73      cout << "\n\t\tHealth Club Membership Menu\n\n"
74           << "1. Standard Adult Membership\n"
75           << "2. Child Membership\n"
76           << "3. Senior Citizen Membership\n"
77           << "4. Quit the Program\n\n"
78           << "Enter your choice: ";
79  }
80
81  //*****************************************************************
82  // Definition of function showFees. The memberRate parameter holds*
83  // the monthly membership rate and the months parameter holds the *
84  // number of months. The function displays the total charges.     *
85  //*****************************************************************
86
87  void showFees(double memberRate, int months)
88  {
89      cout << "The total charges are $"
90           << (memberRate * months) << endl;
91  }
```

(program output continues)

Program 6-10 *(continued)*

Program Output with Example Input Shown in Bold

```
              Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 1 [Enter]
For how many months? 12 [Enter]
The total charges are $480.00

              Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 4 [Enter]
```

Let's take a closer look at this program. First notice the `showMenu` function in lines 71 through 79. This function displays the menu and is called from the `main` function in line 33.

The `showFees` function appears in lines 87 through 91. Its purpose is to display the total fees for a membership lasting a specified number of months. The function accepts two arguments: the monthly membership fee (a `double`) and the number of months of membership (an `int`). The function uses these values to calculate and display the total charges. For example, if we wanted the function to display the fees for an adult membership lasting six months, we would pass the `ADULT` constant as the first argument and 6 as the second argument.

The `showFees` function is called from three different locations in the `switch` statement, which is in the `main` function. The first location is line 54. This statement is executed when the user has selected item 1, standard adult membership, from the menu. The `showFees` function is called with the `ADULT` constant and the `months` variable passed as arguments. The second location is line 57. This statement is executed when the user has selected item 2, child membership, from the menu. The `showFees` function is called in this line with the `CHILD` constant and the `months` variable passed as arguments. The third location is line 60. This statement is executed when the user has selected item 3, senior citizen membership, from the menu. The `showFees` function is called with the `SENIOR` constant and the `months` variable passed as arguments. Each time the `showFees` function is called, it displays the total membership fees for the specified type of membership, for the specified number of months.

Checkpoint

6.5 Indicate which of the following is the function prototype, the function header, and the function call:

```
void showNum(double num)

void showNum(double);

showNum(45.67);
```

6.6 Write a function named `timesTen`. The function should have an integer parameter named `number`. When `timesTen` is called, it should display the product of `number` times ten. (*Note*: just write the function. Do not write a complete program.)

6.7 Write a function prototype for the `timesTen` function you wrote in Question 6.6.

6.8 What is the output of the following program?

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

void showDouble(int); // Function prototype

int main()
{
   int num;

   for (num = 0; num < 10; num++)
      showDouble(num);
   return 0;
}

// Definition of function showDouble.
void showDouble(int value)
{
   cout << value << "\t" << (value * 2) << endl;
}
```

6.9 What is the output of the following program?

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

void func1(double, int); // Function prototype

int main()
{
   int x = 0;
   double y = 1.5;

   cout << x << " " << y << endl;
   func1(y, x);
   cout << x << " " << y << endl;
   return 0;
}
```

```
void func1(double a, int b)
{
   cout << a << " " << b << endl;
   a = 0.0;
   b = 10;
   cout << a << " " << b << endl;
}
```

6.10 The following program skeleton asks for the number of hours you've worked and your hourly pay rate. It then calculates and displays your wages. The function `showDollars`, which you are to write, formats the output of the wages.

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

void showDollars(double); // Function prototype

int main()
{
   double payRate, hoursWorked, wages;

   cout << "How many hours have you worked? "
   cin >> hoursWorked;
   cout << "What is your hourly pay rate? ";
   cin >> payRate;
   wages = hoursWorked * payRate;
   showDollars(wages);
   return 0;
}

// You must write the definition of the function showDollars
// here. It should take one parameter of the type double.
// The function should display the message "Your wages are $"
// followed by the value of the parameter. It should be displayed
// with 2 places of precision after the decimal point, in fixed
// notation, and the decimal point should always display.
```

6.7 The `return` Statement

CONCEPT: The **`return`** statement causes a function to end immediately.

When the last statement in a `void` function has finished executing, the function terminates and the program returns to the statement following the function call. It's possible, however, to force a function to return before the last statement has been executed. When the `return` statement is encountered, the function immediately terminates and control of the program returns to the statement that called the function. This is demonstrated in Program 6-11. The function `divide` shows the quotient of `arg1` divided by `arg2`. If `arg2` is set to zero, the function returns.

Program 6-11

```
1  // This program uses a function to perform division. If division
2  // by zero is detected, the function returns.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype.
7  void divide(double, double);
8
9  int main()
10 {
11     double num1, num2;
12
13     cout << "Enter two numbers and I will divide the first\n";
14     cout << "number by the second number: ";
15     cin >> num1 >> num2;
16     divide(num1, num2);
17     return 0;
18 }
19
20 //*************************************************************
21 // Definition of function divide.                             *
22 // Uses two parameters: arg1 and arg2. The function divides arg1 *
23 // by arg2 and shows the result. If arg2 is zero, however, the  *
24 // function returns.                                          *
25 //*************************************************************
26
27 void divide(double arg1, double arg2)
28 {
29     if (arg2 == 0.0)
30     {
31        cout << "Sorry, I cannot divide by zero.\n";
32        return;
33     }
34     cout << "The quotient is " << (arg1 / arg2) << endl;
35 }
```

Program Output with Example Input Shown in Bold

```
Enter two numbers and I will divide the first
number by the second number: 12 0 [Enter]
Sorry, I cannot divide by zero.
```

In the example running of the program, the user entered 12 and 0 as input. In line 16 the `divide` function was called, passing 12 into the `arg1` parameter and 0 into the `arg2` parameter. Inside the `divide` function, the `if` statement in line 29 executes. Because `arg2` is equal to 0.0, the code in lines 31 and 32 executes. When the `return` statement in line 32 executes, the `divide` function immediately ends. This means the `cout` statement in line 34 does not execute. The program resumes at line 17 in the `main` function.

6.8 Returning a Value from a Function

CONCEPT: **A function may send a value back to the part of the program that called the function.**

You've seen that data may be passed into a function by way of parameter variables. Data may also be returned from a function, back to the statement that called it. Functions that return a value are appropriately known as *value-returning functions*.


VideoNote
Value-Returning Functions

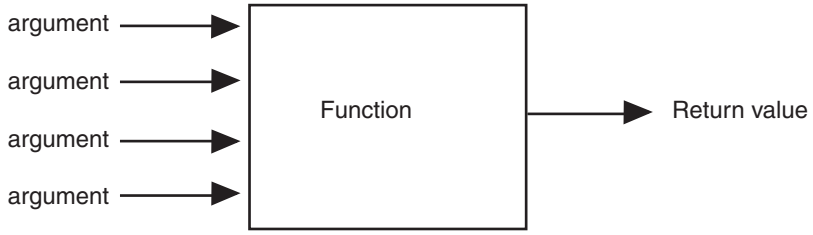
The pow function, which you have already seen, is an example of a value-returning function. Here is an example:

```
double x;
x = pow(4.0, 2.0);
```

The second line in this code calls the `pow` function, passing 4.0 and 2.0 as arguments. The function calculates the value of 4.0 raised to the power of 2.0 and returns that value. The value, which is 16.0, is assigned to the `x` variable by the `=` operator.

Although several arguments may be passed into a function, only one value may be returned from it. Think of a function as having multiple communication channels for receiving data (parameters), but only one channel for sending data (the return value). This is illustrated in Figure 6-9.

Figure 6-9



NOTE: It is possible to return multiple values from a function, but they must be "packaged" in such a way that they are treated as a single value. This is a topic of Chapter 11.

Defining a Value-Returning Function

When you are writing a value-returning function, you must decide what type of value the function will return. This is because you must specify the data type of the return value in the function header, and in the function prototype. Recall that a `void` function, which does not return a value, uses the key word `void` as its return type in the function header. A

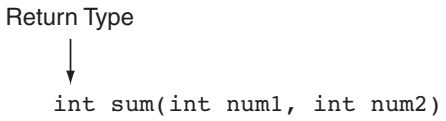
value-returning function will use `int`, `double`, `bool`, or any other valid data type in its header. Here is an example of a function that returns an `int` value:

```
int sum(int num1, int num2)
{
   int result;

   result = num1 + num2;
   return result;
}
```

The name of this function is `sum`. Notice in the function header that the return type is `int`, as illustrated in Figure 6-10.

Figure 6-10



This code defines a function named `sum` that accepts two `int` arguments. The arguments are passed into the parameter variables `num1` and `num2`. Inside the function, a variable, `result`, is defined. Variables that are defined inside a function are called *local variables*. After the variable definition, the parameter variables `num1` and `num2` are added, and their sum is assigned to the `result` variable. The last statement in the function is

```
return result;
```

This statement causes the function to end, and it sends the value of the `result` variable back to the statement that called the function. A value-returning function must have a `return` statement written in the following general format:

```
return expression;
```

In the general format, *`expression`* is the value to be returned. It can be any expression that has a value, such as a variable, literal, or mathematical expression. The value of the expression is converted to the data type that the function returns and is sent back to the statement that called the function. In this case, the `sum` function returns the value in the `result` variable.

However, we could have eliminated the `result` variable and returned the expression `num1 + num2`, as shown in the following code:

```
int sum(int num1, int num2)
{
   return num1 + num2;
}
```

When writing the prototype for a value-returning function, follow the same conventions that we have covered earlier. Here is the prototype for the `sum` function:

```
int sum(int, int);
```

Calling a Value-Returning Function

Program 6-12 shows an example of how to call the sum function.

Program 6-12

```
1  // This program uses a function that returns a value.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  int sum(int, int);
7
8  int main()
9  {
10     int value1 = 20,     // The first value
11         value2 = 40,     // The second value
12         total;           // To hold the total
13
14     // Call the sum function, passing the contents of
15     // value1 and value2 as arguments. Assign the return
16     // value to the total variable.
17     total = sum(value1, value2);
18
19     // Display the sum of the values.
20     cout << "The sum of " << value1 << " and "
21          << value2 << " is " << total << endl;
22     return 0;
23 }
24
25 //*************************************************
26 // Definition of function sum. This function returns *
27 // the sum of its two parameters.                    *
28 //*************************************************
29
30 int sum(int num1, int num2)
31 {
32     return num1 + num2;
33 }
```

Program Output

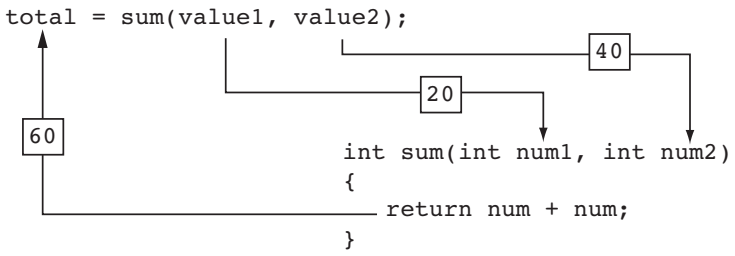
```
The sum of 20 and 40 is 60
```

Here is the statement in line 17 that calls the `sum` function, passing `value1` and `value2` as arguments.

```
total = sum(value1, value2);
```

This statement assigns the value returned by the sum function to the `total` variable. In this case, the function will return 60. Figure 6-11 shows how the arguments are passed into the function and how a value is passed back from the function.

Figure 6-11



When you call a value-returning function, you usually want to do something meaningful with the value it returns. Program 6-12 shows a function's return value being assigned to a variable. This is commonly how return values are used, but you can do many other things with them. For example, the following code shows a mathematical expression that uses a call to the `sum` function:

```
int x = 10, y = 15;
double average;
average = sum(x, y) / 2.0;
```

In the last statement, the `sum` function is called with `x` and `y` as its arguments. The function's return value, which is 25, is divided by 2.0. The result, 12.5, is assigned to `average`. Here is another example:

```
int x = 10, y = 15;
cout << "The sum is " << sum(x, y) << endl;
```

This code sends the `sum` function's return value to `cout` so it can be displayed on the screen. The message "The sum is 25" will be displayed.

Remember, a value-returning function returns a value of a specific data type. You can use the function's return value anywhere that you can use a regular value of the same data type. This means that anywhere an `int` value can be used, a call to an `int` value-returning function can be used. Likewise, anywhere a `double` value can be used, a call to a `double` value-returning function can be used. The same is true for all other data types.

Let's look at another example. Program 6-13, which calculates the area of a circle, has two functions in addition to `main`. One of the functions is named `square`, and it returns the square of any number passed to it as an argument. The `square` function is called in a mathematical statement. The program also has a function named `getRadius`, which prompts the user to enter the circle's radius. The value entered by the user is returned from the function.

Program 6-13

```
1  // This program demonstrates two value-returning functions.
2  // The square function is called in a mathematical statement.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
```

(program continues)

Program 6-13 *(continued)*

```
 6
 7  //Function prototypes
 8  double getRadius();
 9  double square(double);
10
11  int main()
12  {
13      const double PI = 3.14159; // Constant for pi
14      double radius;             // To hold the circle's radius
15      double area;               // To hold the circle's area
16
17      // Set the numeric output formatting.
18      cout << fixed << showpoint << setprecision(2);
19
20      // Get the radius of the circle.
21      cout << "This program calculates the area of ";
22      cout << "a circle.\n";
23      radius = getRadius();
24
25      // Calculate the area of the circle.
26      area = PI * square(radius);
27
28      // Display the area.
29      cout << "The area is " << area << endl;
30      return 0;
31  }
32
33  //********************************************************
34  // Definition of function getRadius.                     *
35  // This function asks the user to enter the radius of    *
36  // the circle and then returns that number as a double.  *
37  //********************************************************
38
39  double getRadius()
40  {
41      double rad;
42
43      cout << "Enter the radius of the circle: ";
44      cin >> rad;
45      return rad;
46  }
47
48  //********************************************************
49  // Definition of function square.                        *
50  // This function accepts a double argument and returns   *
51  // the square of the argument as a double.               *
52  //********************************************************
53
54  double square(double number)
```

```
55  {
56       return number * number;
57  }
```

Program Output with Example Input Shown in Bold

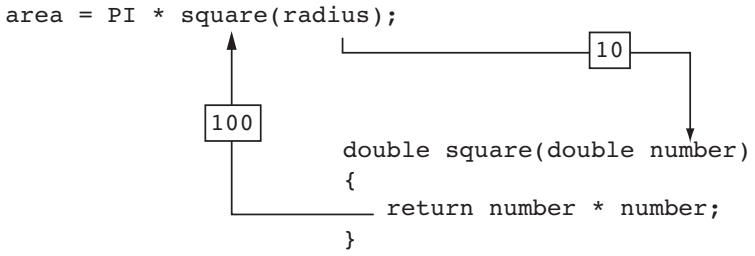
```
This program calculates the area of a circle.
Enter the radius of the circle: 10 [Enter]
The area is 314.16
```

First, look at the `getRadius` function defined in lines 39 through 46. The purpose of the function is to prompt the user to enter the radius of a circle. In line 41 the function defines a local variable, `rad`. Lines 43 and 44 prompt the user to enter the circle's radius, which is stored in the `rad` variable. In line 45 the value of the `rad` value is returned. The `getRadius` function is called in the `main` function, in line 23. The value that is returned from the function is assigned to the `radius` variable.

Next look at the `square` function, which is defined in lines 54 through 57. When the function is called, a `double` argument is passed to it. The function stores the argument in the `number` parameter. The `return` statement in line 56 returns the value of the expression `number * number`, which is the square of the `number` parameter. The `square` function is called in the `main` function, in line 26, with the value of `radius` passed as an argument. The function will return the square of the `radius` variable, and that value will be used in the mathematical expression.

Assuming the user has entered 10 as the radius, and this value is passed as an argument to the `square` function, the `square` function will return the value 100. Figure 6-12 illustrates how the value 100 is passed back to the mathematical expression in line 26. The value 100 will then be used in the mathematical expression.

Figure 6-12



Functions can return values of any type. Both the `getRadius` and `square` functions in Program 6-13 return a `double`. The `sum` function you saw in Program 6-12 returned an `int`. When a statement calls a value-returning function, it should properly handle the return value. For example, if you assign the return value of the `square` function to a variable, the variable should be a `double`. If the return value of the function has a fractional portion and you assign it to an `int` variable, the value will be truncated.

In the Spotlight:

Using Functions

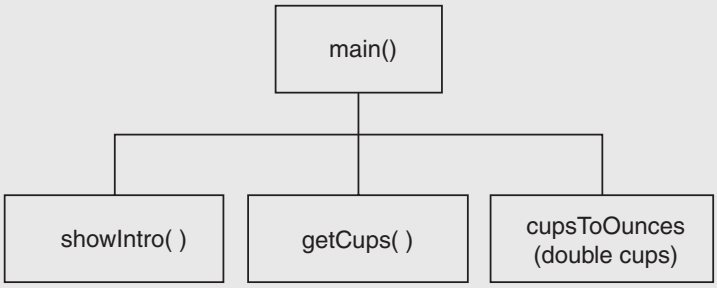
Your friend Michael runs a catering company. Some of the ingredients that his recipes require are measured in cups. When he goes to the grocery store to buy those ingredients, however, they are sold only by the fluid ounce. He has asked you to write a simple program that converts cups to fluid ounces.

You design the following algorithm:

1. *Display an introductory screen that explains what the program does.*
2. *Get the number of cups.*
3. *Convert the number of cups to fluid ounces and display the result.*

This algorithm lists the top level of tasks that the program needs to perform and becomes the basis of the program's `main` function. The hierarchy chart shown in Figure 6-13 shows how the program will broken down into functions.

Figure 6-13 Hierarchy chart for the program



As shown in the hierarchy chart, the `main` function will call three other functions. Here are summaries of those functions:

- `showIntro`—This function will display a message on the screen that explains what the program does.
- `getCups`—This function will prompt the user to enter the number of cups and then will return that value as a `double`.
- `cupsToOunces`—This function will accept the number of cups as an argument and then return an equivalent number of fluid ounces as a `double`.

Program 6-14 shows the code for the program.

Program 6-14

```
1  // This program converts cups to fluid ounces.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
```

```
5
6  // Function prototypes
7  void showIntro();
8  double getCups();
9  double cupsToOunces(double);
10
11 int main()
12 {
13     // Variables for the cups and ounces.
14     double cups, ounces;
15
16     // Set up numeric output formatting.
17     cout << fixed << showpoint << setprecision(1);
18
19     // Display an intro screen.
20     showIntro();
21
22     // Get the number of cups.
23     cups = getCups();
24
25     // Convert cups to fluid ounces.
26     ounces = cupsToOunces(cups);
27
28     // Display the number of ounces.
29     cout << cups << " cups equals "
30          << ounces << " ounces.\n";
31
32     return 0;
33 }
34
35 //****************************************
36 // The showIntro function displays an     *
37 // introductory screen.                   *
38 //****************************************
39
40 void showIntro()
41 {
42     cout << "This program converts measurements\n"
43          << "in cups to fluid ounces. For your\n"
44          << "reference the formula is:\n"
45          << " 1 cup = 8 fluid ounces\n\n";
46 }
47
48 //****************************************
49 // The getCups function prompts the user  *
50 // to enter the number of cups and then   *
51 // returns that value as a double.        *
52 //****************************************
53
54 double getCups()
```

(program continues)

Program 6-14 *(continued)*

```
55  {
56      double numCups;
57
58      cout << "Enter the number of cups: ";
59      cin >> numCups;
60      return numCups;
61  }
62
63  //****************************************
64  // The cupsToOunces function accepts a    *
65  // number of cups as an argument and      *
66  // returns the equivalent number of fluid *
67  // ounces as a double.                    *
68  //****************************************
69
70  double cupsToOunces(double numCups)
71  {
72      return numCups * 8.0;
73  }
```

Program Output with Example Input Shown in Bold

```
This program converts measurements
in cups to fluid ounces. For your
reference the formula is:
    1 cup = 8 fluid ounces

Enter the number of cups: 2 [Enter]
2.0 cups equals 16.0 ounces.
```

6.9 Returning a Boolean Value

CONCEPT: Functions may return `true` or `false` values.

Frequently there is a need for a function that tests an argument and returns a `true` or `false` value indicating whether or not a condition exists. Such a function would return a `bool` value. For example, the following function accepts an `int` argument and returns `true` if the argument is within the range of 1 through 100, or `false` otherwise.

```
bool isValid(int number)
{
   bool status;

   if (number >= 1 && number <= 100)
      status = true;
   else
      status = false;
   return status;
}
```

The following code shows an `if/else` statement that uses a call to the function:

```
int value = 20;
if (isValid(value))
   cout << "The value is within range.\n";
else
   cout << "The value is out of range.\n";
```

When this code executes, the message "The value is within range." will be displayed.

Program 6-15 shows another example. This program has a function named `isEven` which returns `true` if its argument is an even number. Otherwise, the function returns `false`.

Program 6-15

```
1  // This program uses a function that returns true or false.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  bool isEven(int);
7
8  int main()
9  {
10     int val;
11
12     // Get a number from the user.
13     cout << "Enter an integer and I will tell you ";
14     cout << "if it is even or odd: ";
15     cin >> val;
16
17     // Indicate whether it is even or odd.
18     if (isEven(val))
19        cout << val << " is even.\n";
20     else
21        cout << val << " is odd.\n";
22     return 0;
23 }
24
25 //*************************************************************
26 // Definition of function isEven. This function accepts an       *
27 // integer argument and tests it to be even or odd. The function *
28 // returns true if the argument is even or false if the argument *
29 // is odd. The return value is a bool.                           *
30 //*************************************************************
31
32 bool isEven(int number)
33 {
34     bool status;
35
36     if (number % 2 == 0)
37        status = true;  // The number is even if there is no remainder.
38     else
39        status = false; // Otherwise, the number is odd.
40     return status;
41 }
```

(program output continues)

Program 6-15 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter an integer and I will tell you if it is even or odd: 5 [Enter]
5 is odd.
```

The `isEven` function is called in line 18, in the following statement:

```
if (isEven(val))
```

When the `if` statement executes, `isEven` is called with `val` as its argument. If `val` is even, `isEven` returns `true` , otherwise it returns `false`.

Checkpoint

6.11 How many return values may a function have?

6.12 Write a header for a function named `distance`. The function should return a `double` and have two `double` parameters: `rate` and `time`.

6.13 Write a header for a function named `days`. The function should return an `int` and have three `int` parameters: `years`, `months`, and `weeks`.

6.14 Write a header for a function named `getKey`. The function should return a `char` and use no parameters.

6.15 Write a header for a function named `lightYears`. The function should return a `long` and have one `long` parameter: `miles`.

6.10 Local and Global Variables

CONCEPT: **A local variable is defined inside a function and is not accessible outside the function. A global variable is defined outside all functions and is accessible to all functions in its scope.**

Local Variables

Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them. Program 6-16 shows that because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

Program 6-16

```
1  // This program shows that variables defined in a function
2  // are hidden from other functions.
3  #include <iostream>
4  using namespace std;
5
6  void anotherFunction(); // Function prototype
```

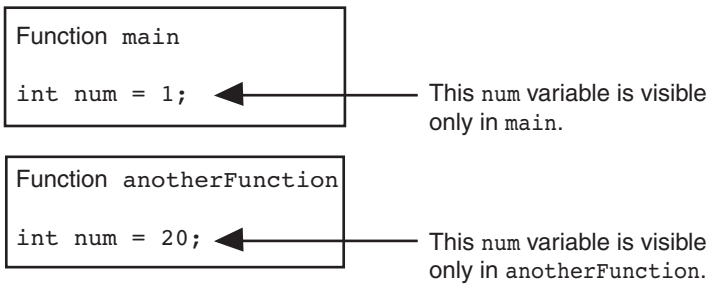
```
 7
 8  int main()
 9  {
10      int num = 1; // Local variable
11
12      cout << "In main, num is " << num << endl;
13      anotherFunction();
14      cout << "Back in main, num is " << num << endl;
15      return 0;
16  }
17
18  //*************************************************
19  // Definition of anotherFunction                  *
20  // It has a local variable, num, whose initial value *
21  // is displayed.                                  *
22  //*************************************************
23
24  void anotherFunction()
25  {
26      int num = 20; // Local variable
27
28      cout << "In anotherFunction, num is " << num << endl;
29  }
```

Program Output

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1
```

Even though there are two variables named num, the program can only "see" one of them at a time because they are in different functions. When the program is executing in `main`, the num variable defined in `main` is visible. When `anotherFunction` is called, however, only variables defined inside it are visible, so the num variable in `main` is hidden. Figure 6-14 illustrates the closed nature of the two functions. The boxes represent the scope of the variables.

Figure 6-14



Local Variable Lifetime

A function's local variables exist only while the function is executing. This is known as the *lifetime* of a local variable. When the function begins, its local variables and its parameter

variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed. This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Initializing Local Variables with Parameter Values

It is possible to use a parameter variable to initialize a local variable. Sometimes this simplifies the code in a function. For example, recall the first version of the `sum` function we discussed earlier:

```
int sum(int num1, int num2)
{
   int result;

   result = num1 + num2;
   return result;
}
```

In the body of the function, the `result` variable is defined and then a separate assignment statement assigns `num1 + num2` to `result`. We can combine these statements into one, as shown in the following modified version of the function.

```
int sum(int num1, int num2)
{
   int result = num1 + num2;
   return result;
}
```

Because the scope of a parameter variable is the entire function in which it is declared, we can use parameter variables to initialize local variables.

Global Variables

A global variable is any variable defined outside all the functions in a program. The scope of a global variable is the portion of the program from the variable definition to the end. This means that a global variable can be accessed by all functions that are defined after the global variable is defined. Program 6-17 shows two functions, `main` and `anotherFunction`, that access the same global variable, `num`.

Program 6-17

```
1  // This program shows that a global variable is visible
2  // to all the functions that appear in a program after
3  // the variable's declaration.
4  #include <iostream>
5  using namespace std;
6
7  void anotherFunction();  // Function prototype
8  int num = 2;             // Global variable
9
10 int main()
11 {
12     cout << "In main, num is " << num << endl;
```

```
13       anotherFunction();
14       cout << "Back in main, num is " << num << endl;
15       return 0;
16   }
17
18   //*****************************************************
19   // Definition of anotherFunction                      *
20   // This function changes the value of the             *
21   // global variable num.                               *
22   //*****************************************************
23
24   void anotherFunction()
25   {
26       cout << "In anotherFunction, num is " << num << endl;
27       num = 50;
28       cout << "But, it is now changed to " << num << endl;
29   }
```

Program Output

```
In main, num is 2
In anotherFunction, num is 2
But, it is now changed to 50
Back in main, num is 50
```

In Program 6-17, `num` is defined outside of all the functions. Because its definition appears before the definitions of `main` and `anotherFunction`, both functions have access to it.

Unless you explicitly initialize numeric global variables, they are automatically initialized to zero. Global character variables are initialized to NULL.* The variable `globalNum` in Program 6-18 is never set to any value by a statement, but because it is global, it is automatically set to zero.

Program 6-18

```
1   // This program has an uninitialized global variable.
2   #include <iostream>
3   using namespace std;
4
5   int globalNum; // Global variable, automatically set to zero
6
7   int main()
8   {
9       cout << "globalNum is " << globalNum << endl;
10      return 0;
11  }
```

Program Output

```
globalNum is 0
```

*The NULL character is stored as ASCII code 0.

Now that you've had a basic introduction to global variables, I must warn you to restrict your use of them. When beginning students first learn to write programs with multiple functions, they are sometimes tempted to make all their variables global. This is usually because global variables can be accessed by any function in the program without being passed as arguments. Although this approach might make a program easier to create, it usually causes problems later. The reasons are as follows:

- Global variables make debugging difficult. Any statement in a program can change the value of a global variable. If you find that the wrong value is being stored in a global variable, you have to track down every statement that accesses it to determine where the bad value is coming from. In a program with thousands of lines of code, this can be difficult.
- Functions that use global variables are usually dependent on those variables. If you want to use such a function in a different program, most likely you will have to redesign it so it does not rely on the global variable.
- Global variables make a program hard to understand. A global variable can be modified by any statement in the program. If you are to understand any part of the program that uses a global variable, you have to be aware of all the other parts of the program that access the global variable.

Because of this, you should not use global variables for the conventional purposes of storing, manipulating, and retrieving data. In most cases, you should declare variables locally and pass them as arguments to the functions that need to access them.

Global Constants

Although you should try to avoid the use of global variables, it is generally permissible to use global constants in a program. A *global constant* is a named constant that is available to every function in a program. Because a global constant's value cannot be changed during the program's execution, you do not have to worry about the potential hazards that are associated with the use of global variables.

Global constants are typically used to represent unchanging values that are needed throughout a program. For example, suppose a banking program uses a named constant to represent an interest rate. If the interest rate is used in several functions, it is easier to create a global constant, rather than a local named constant in each function. This also simplifies maintenance. If the interest rate changes, only the declaration of the global constant has to be changed, instead of several local declarations.

Program 6-19 shows an example of how global constants might be used. The program calculates an employee's gross pay, including overtime. In addition to `main`, this program has two functions: `getBasePay` and `getOvertimePay`. The `getBasePay` function accepts the number of hours worked and returns the amount of pay for the non-overtime hours. The `getOvertimePay` function accepts the number of hours worked and returns the amount of pay for the overtime hours, if any.

Program 6-19

```
1  // This program calculates gross pay.
2  #include <iostream>
3  #include <iomanip>
```

```
4  using namespace std;
5
6  // Global constants
7  const double PAY_RATE = 22.55;     // Hourly pay rate
8  const double BASE_HOURS = 40.0;    // Max non-overtime hours
9  const double OT_MULTIPLIER = 1.5; // Overtime multiplier
10
11 // Function prototypes
12 double getBasePay(double);
13 double getOvertimePay(double);
14
15 int main()
16 {
17     double hours,            // Hours worked
18            basePay,          // Base pay
19            overtime = 0.0,   // Overtime pay
20            totalPay;         // Total pay
21
22     // Get the number of hours worked.
23     cout << "How many hours did you work? ";
24     cin >> hours;
25
26     // Get the amount of base pay.
27     basePay = getBasePay(hours);
28
29     // Get overtime pay, if any.
30     if (hours > BASE_HOURS)
31        overtime = getOvertimePay(hours);
32
33     // Calculate the total pay.
34     totalPay = basePay + overtime;
35
36     // Set up numeric output formatting.
37     cout << setprecision(2) << fixed << showpoint;
38
39     // Display the pay.
40     cout << "Base pay: $" << basePay << endl
41          << "Overtime pay $" << overtime << endl
42          << "Total pay $" << totalPay << endl;
43     return 0;
44 }
45
46 //************************************************
47 // The getBasePay function accepts the number of *
48 // hours worked as an argument and returns the   *
49 // employee's pay for non-overtime hours.        *
50 //************************************************
51
52 double getBasePay(double hoursWorked)
53 {
54     double basePay; // To hold base pay
55
```

(program continues)

Program 6-19 *(continued)*

```
56      // Determine base pay.
57      if (hoursWorked > BASE_HOURS)
58         basePay = BASE_HOURS * PAY_RATE;
59      else
60         basePay = hoursWorked * PAY_RATE;
61
62      return basePay;
63  }
64
65  //************************************************
66  // The getOvertimePay function accepts the number *
67  // of hours worked as an argument and returns the *
68  // employee's overtime pay.                       *
69  //************************************************
70
71  double getOvertimePay(double hoursWorked)
72  {
73      double overtimePay; // To hold overtime pay
74
75      // Determine overtime pay.
76      if (hoursWorked > BASE_HOURS)
77      {
78         overtimePay = (hoursWorked - BASE_HOURS) *
79                  PAY_RATE * OT_MULTIPLIER;
80      }
81      else
82         overtimePay = 0.0;
83
84      return overtimePay;
85  }
```

Program Output with Example Input Shown in Bold

```
How many hours did you work? 48 [Enter]
Base pay: $902.00
Overtime pay: $270.60
Total pay: $1172.60
```

Let's take a closer look at the program. Three global constants are defined in lines 7, 8, and 9. The `PAY_RATE` constant is set to the employee's hourly pay rate, which is 22.55. The `BASE_HOURS` constant is set to 40, which is the number of hours an employee can work in a week without getting paid overtime. The `OT_MULTIPLIER` constant is set to 1.5, which is the pay rate multiplier for overtime hours. This means that the employee's hourly pay rate is multiplied by 1.5 for all overtime hours.

Because these constants are global and are defined before all of the functions in the program, all the functions may access them. For example, the `getBasePay` function accesses the `BASE_HOURS` constant in lines 57 and 58 and accesses the `PAY_RATE` constant in lines 58 and 60. The `getOvertimePay` function accesses the `BASE_HOURS` constant in lines 76 and 78, the `PAY_RATE` constant in line 79, and the `OT_MULTIPLIER` constant in line 79.

Local and Global Variables with the Same Name

You cannot have two local variables with the same name in the same function. This applies to parameter variables as well. A parameter variable is, in essence, a local variable. So, you cannot give a parameter variable and a local variable in the same function the same name.

However, you can have a local variable or a parameter variable with the same name as a global variable, or a global constant. When you do, the name of the local or parameter variable *shadows* the name of the global variable or global constant. This means that the global variable or constant's name is hidden by the name of the local or parameter variable. For example, look at Program 6-20. This program has a global constant named `BIRDS`, set to 500. The `california` function has a local constant named `BIRDS`, set to 10000.

Program 6-20

```
1  // This program demonstrates how a local variable
2  // can shadow the name of a global constant.
3  #include <iostream>
4  using namespace std;
5
6  // Global constant.
7  const int BIRDS = 500;
8
9  // Function prototype
10 void california();
11
12 int main()
13 {
14     cout << "In main there are " << BIRDS
15          << " birds.\n";
16     california();
17     return 0;
18 }
19
20 //*****************************************
21 // california function                    *
22 //*****************************************
23
24 void california()
25 {
26     const int BIRDS = 10000;
27     cout << "In california there are " << BIRDS
28          << " birds.\n";
29 }
```

Program Output

```
In main there are 500 birds.
In california there are 10000 birds.
```

When the program is executing in the `main` function, the global constant `BIRDS`, which is set to 500, is visible. The `cout` statement in lines 14 and 15 displays "In main there are 500 birds." (My apologies to folks living in Maine for the difference in spelling.) When the program is executing in the `california` function, however, the local constant `BIRDS` shadows the global constant `BIRDS`. When the `california` function accesses `BIRDS`, it accesses the local constant. That is why the `cout` statement in lines 27 and 28 displays "In california there are 10000 birds."

6.11 Static Local Variables

If a function is called more than once in a program, the values stored in the function's local variables do not persist between function calls. This is because the local variables are destroyed when the function terminates and are then re-created when the function starts again. This is shown in Program 6-21.

Program 6-21

```
1  // This program shows that local variables do not retain
2  // their values between function calls.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  void showLocal();
8
9  int main()
10 {
11     showLocal();
12     showLocal();
13     return 0;
14 }
15
16 //*********************************************************
17 // Definition of function showLocal.                      *
18 // The initial value of localNum, which is 5, is displayed. *
19 // The value of localNum is then changed to 99 before the  *
20 // function returns.                                      *
21 //*********************************************************
22
23 void showLocal()
24 {
25     int localNum = 5; // Local variable
26
27     cout << "localNum is " << localNum << endl;
28     localNum = 99;
29 }
```

Program Output

```
localNum is 5
localNum is 5
```

Even though in line 28 the last statement in the `showLocal` function stores 99 in `localNum`, the variable is destroyed when the function returns. The next time the function is called, `localNum` is re-created and initialized to 5 again.

Sometimes it's desirable for a program to "remember" what value is stored in a local variable between function calls. This can be accomplished by making the variable `static`.

Static local variables are not destroyed when a function returns. They exist for the lifetime of the program, even though their scope is only the function in which they are defined. Program 6-22 demonstrates some characteristics of `static` local variables:

Program 6-22

```
1  // This program uses a static local variable.
2  #include <iostream>
3  using namespace std;
4
5  void showStatic(); // Function prototype
6
7  int main()
8  {
9       // Call the showStatic function five times.
10      for (int count = 0; count < 5; count++)
11         showStatic();
12      return 0;
13 }
14
15 //*************************************************************
16 // Definition of function showStatic.                         *
17 // statNum is a static local variable. Its value is displayed *
18 // and then incremented just before the function returns.     *
19 //*************************************************************
20
21 void showStatic()
22 {
23      static int statNum;
24
25      cout << "statNum is " << statNum << endl;
26      statNum++;
27 }
```

Program Output

```
statNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
```

In line 26 of Program 6-22, `statNum` is incremented in the `showStatic` function, and it retains its value between each function call. Notice that even though `statNum` is not explicitly initialized, it starts at zero. Like global variables, all `static` local variables are initialized to zero by default. (Of course, you can provide your own initialization value, if necessary.)

If you do provide an initialization value for a `static` local variable, the initialization only occurs once. This is because initialization normally happens when the variable is created, and `static` local variables are only created once during the running of a program. Program 6-23, which is a slight modification of Program 6-22, illustrates this point.

Program 6-23

```
1  // This program shows that a static local variable is only
2  // initialized once.
3  #include <iostream>
4  using namespace std;
5
6  void showStatic(); // Function prototype
7
8  int main()
9  {
10     // Call the showStatic function five times.
11     for (int count = 0; count < 5; count++)
12        showStatic();
13     return 0;
14 }
15
16 //*************************************************************
17 // Definition of function showStatic.                         *
18 // statNum is a static local variable. Its value is displayed *
19 // and then incremented just before the function returns.     *
20 //*************************************************************
21
22 void showStatic()
23 {
24     static int statNum = 5;
25
26     cout << "statNum is " << statNum << endl;
27     statNum++;
28 }
```

Program Output

```
statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9
```

Even though the statement that defines `statNum` in line 24 initializes it to 5, the initialization does not happen each time the function is called. If it did, the variable would not be able to retain its value between function calls.

Checkpoint

6.16 What is the difference between a static local variable and a global variable?

6.17 What is the output of the following program?

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

void myFunc();  // Function prototype

int main()
{
```

```
    int var = 100;

    cout << var << endl;
    myFunc();
    cout << var << endl;
    return 0;
}

// Definition of function myFunc
void myFunc()
{
    int var = 50;

    cout << var << endl;
}
```

6.18 What is the output of the following program?

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

void showVar(); // Function prototype

int main()
{
    for (int count = 0; count < 10; count++)
       showVar();
    return 0;
}

// Definition of function showVar
void showVar()
{
    static int var = 10;

    cout << var << endl;
    var++;
}
```

6.12 Default Arguments

CONCEPT: **Default arguments are passed to parameters automatically if no argument is provided in the function call.**

It's possible to assign *default arguments* to function parameters. A default argument is passed to the parameter when the actual argument is left out of the function call. The default arguments are usually listed in the function prototype. Here is an example:

```
void showArea(double = 20.0, double = 10.0);
```

Default arguments are literal values or constants with an = operator in front of them, appearing after the data types listed in a function prototype. Since parameter names are optional in function prototypes, the example prototype could also be declared as

```
void showArea(double length = 20.0, double width = 10.0);
```

In both example prototypes, the function `showArea` has two `double` parameters. The first is assigned the default argument 20.0 and the second is assigned the default argument 10.0. Here is the definition of the function:

```
void showArea(double length, double width)
{
   double area = length * width;
   cout << "The area is " << area << endl;
}
```

The default argument for `length` is 20.0 and the default argument for `width` is 10.0. Because both parameters have default arguments, they may optionally be omitted in the function call, as shown here:

```
showArea();
```

In this function call, both default arguments will be passed to the parameters. The parameter `length` will take the value 20.0 and `width` will take the value 10.0. The output of the function will be

```
The area is 200
```

The default arguments are only used when the actual arguments are omitted from the function call. In the call below, the first argument is specified, but the second is omitted:

```
showArea(12.0);
```

The value 12.0 will be passed to `length`, while the default value 10.0 will be passed to `width`. The output of the function will be

```
The area is 120
```

Of course, all the default arguments may be overridden. In the function call below, arguments are supplied for both parameters:

```
showArea(12.0, 5.5);
```

The output of the function call above will be

```
The area is 66
```

NOTE: If a function does not have a prototype, default arguments may be specified in the function header. The `showArea` function could be defined as follows:

```
void showArea(double length = 20.0, double width = 10.0)
{
   double area = length * width;
   cout << "The area is " << area << endl;
}
```

WARNING! A function's default arguments should be assigned in the earliest occurrence of the function name. This will usually be the function prototype.

Program 6-24 uses a function that displays asterisks on the screen. Arguments are passed to the function specifying how many columns and rows of asterisks to display. Default arguments are provided to display one row of 10 asterisks.

Program 6-24

```
1  // This program demonstrates default function arguments.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype with default arguments
6  void displayStars(int = 10, int = 1);
7
8  int main()
9  {
10     displayStars();      // Use default values for cols and rows.
11     cout << endl;
12     displayStars(5);     // Use default value for rows.
13     cout << endl;
14     displayStars(7, 3);  // Use 7 for cols and 3 for rows.
15     return 0;
16 }
17
18 //*********************************************************
19 // Definition of function displayStars.                   *
20 // The default argument for cols is 10 and for rows is 1.*
21 // This function displays a square made of asterisks.     *
22 //*********************************************************
23
24 void displayStars(int cols, int rows)
25 {
26     // Nested loop. The outer loop controls the rows
27     // and the inner loop controls the columns.
28     for (int down = 0; down < rows; down++)
29     {
30          for (int across = 0; across < cols; across++)
31              cout << "*";
32          cout << endl;
33     }
34 }
```

Program Output

```
**********

*****

*******
*******
*******
```

Although C++'s default arguments are very convenient, they are not totally flexible in their use. When an argument is left out of a function call, all arguments that come after it must be left out as well. In the `displayStars` function in Program 6-24, it is not possible to omit the argument for `cols` without also omitting the argument for `rows`. For example, the following function call would be illegal:

```
displayStars(, 3); // Illegal function call.
```

It's possible for a function to have some parameters with default arguments and some without. For example, in the following function (which displays an employee's gross pay), only the last parameter has a default argument:

```
// Function prototype
void calcPay(int empNum, double payRate, double hours = 40.0);

// Definition of function calcPay
void calcPay(int empNum, double payRate, double hours)
{
   double wages;

   wages = payRate * hours;
   cout << fixed << showpoint << setprecision(2);
   cout << "Gross pay for employee number ";
   cout << empNum << " is " << wages << endl;
}
```

When calling this function, arguments must always be specified for the first two parameters (`empNum` and `payRate`) since they have no default arguments. Here are examples of valid calls:

```
calcPay(769, 15.75);       // Use default arg for 40 hours
calcPay(142, 12.00, 20);   // Specify number of hours
```

When a function uses a mixture of parameters with and without default arguments, the parameters with default arguments must be defined last. In the `calcPay` function, `hours` could not have been defined before either of the other parameters. The following prototypes are illegal:

```
// Illegal prototype
void calcPay(int empNum, double hours = 40.0, double payRate);

// Illegal prototype
void calcPay(double hours = 40.0, int empNum, double payRate);
```

Here is a summary of the important points about default arguments:

- The value of a default argument must be a literal value or a named constant.
- When an argument is left out of a function call (because it has a default value), all the arguments that come after it must be left out too.
- When a function has a mixture of parameters both with and without default arguments, the parameters with default arguments must be declared last.

6.13 Using Reference Variables as Parameters

CONCEPT: **When used as parameters, reference variables allow a function to access the parameter's original argument. Changes to the parameter are also made to the argument.**

Earlier you saw that arguments are normally passed to a function by value, and that the function cannot change the source of the argument. C++ provides a special type of variable

called a *reference variable* that, when used as a function parameter, allows access to the original argument.

A reference variable is an alias for another variable. Any changes made to the reference variable are actually performed on the variable for which it is an alias. By using a reference variable as a parameter, a function may change a variable that is defined in another function.

Reference variables are defined like regular variables, except you place an ampersand (&) in front of the name. For example, the following function definition makes the parameter `refVar` a reference variable:

```
void doubleNum(int &refVar)
{
   refVar *= 2;
}
```

NOTE: The variable `refVar` is called "a reference to an `int`."

This function doubles `refVar` by multiplying it by 2. Since `refVar` is a reference variable, this action is actually performed on the variable that was passed to the function as an argument. When prototyping a function with a reference variable, be sure to include the ampersand after the data type. Here is the prototype for the `doubleNum` function:

```
void doubleNum(int &);
```

NOTE: Some programmers prefer not to put a space between the data type and the ampersand. The following prototype is equivalent to the one above:

```
void doubleNum(int &);
```

NOTE: The ampersand must appear in both the prototype and the header of any function that uses a reference variable as a parameter. It does not appear in the function call.

Program 6-25 demonstrates how the `doubleNum` function works.

Program 6-25

```
1  // This program uses a reference variable as a function
2  // parameter.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype. The parameter is a reference variable.
7  void doubleNum(int &);
8
```

(program continues)

Program 6-25 *(continued)*

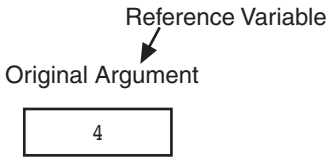
```
 9  int main()
10  {
11      int value = 4;
12
13      cout << "In main, value is " << value << endl;
14      cout << "Now calling doubleNum..." << endl;
15      doubleNum(value);
16      cout << "Now back in main. value is " << value << endl;
17      return 0;
18  }
19
20  //*********************************************************
21  // Definition of doubleNum.                               *
22  // The parameter refVar is a reference variable. The value*
23  // in refVar is doubled.                                  *
24  //*********************************************************
25
26  void doubleNum (int &refVar)
27  {
28      refVar *= 2;
29  }
```

Program Output

```
In main, value is 4
Now calling doubleNum...
Now back in main. value is 8
```

The parameter `refVar` in Program 6-25 "points" to the `value` variable in function `main`. When a program works with a reference variable, it is actually working with the variable it references, or points to. This is illustrated in Figure 6-15.

Figure 6-15



Recall that function arguments are normally passed by value, which means a copy of the argument's value is passed into the parameter variable. When a reference parameter is used, it is said that the argument is *passed by reference*.

Program 6-26 is a modification of Program 6-25. The function `getNum` has been added. The function asks the user to enter a number, which is stored in `userNum`. `userNum` is a reference to `main`'s variable `value`.

Program 6-26

```
1  // This program uses reference variables as function parameters.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototypes. Both functions use reference variables
6  // as parameters.
7  void doubleNum(int &);
8  void getNum(int &);
9
10 int main()
11 {
12     int value;
13
14     // Get a number and store it in value.
15     getNum(value);
16
17     // Double the number stored in value.
18     doubleNum(value);
19
20     // Display the resulting number.
21     cout << "That value doubled is " << value << endl;
22     return 0;
23 }
24
25 //*************************************************************
26 // Definition of getNum.                                      *
27 // The parameter userNum is a reference variable. The user is *
28 // asked to enter a number, which is stored in userNum.       *
29 //*************************************************************
30
31 void getNum(int &userNum)
32 {
33     cout << "Enter a number: ";
34     cin >> userNum;
35 }
36
37 //*************************************************************
38 // Definition of doubleNum.                                   *
39 // The parameter refVar is a reference variable. The value    *
40 // in refVar is doubled.                                      *
41 //*************************************************************
42
43 void doubleNum (int &refVar)
44 {
45     refVar *= 2;
46 }
```

Program Output with Example Input Shown in Bold

```
Enter a number: 12 [Enter]
That value doubled is 24
```

NOTE: Only variables may be passed by reference. If you attempt to pass a nonvariable argument, such as a literal, a constant, or an expression, into a reference parameter, an error will result. Using the `doubleNum` function as an example, the following statements will generate an error.

```
doubleNum(5);              // Error
doubleNum(userNum + 10);   // Error
```

If a function uses more than one reference variable as a parameter, be sure to place the ampersand before each reference variable name. Here is the prototype and definition for a function that uses four reference variable parameters:

```
// Function prototype with four reference variables
// as parameters.
void addThree(int &, int &, int &, int &);

// Definition of addThree.
// All four parameters are reference variables.
void addThree(int &sum, int &num1, int &num2, int &num3)
{
   cout << "Enter three integer values: ";
   cin >> num1 >> num2 >> num3;
   sum = num1 + num2 + num3;
}
```

WARNING! Don't get carried away with using reference variables as function parameters. Any time you allow a function to alter a variable that's outside the function, you are creating potential debugging problems. Reference variables should only be used as parameters when the situation requires them.

Checkpoint

6.19 What kinds of values may be specified as default arguments?

6.20 Write the prototype and header for a function called `compute`. The function should have three parameters: an `int`, a `double`, and a `long` (not necessarily in that order). The `int` parameter should have a default argument of 5, and the `long` parameter should have a default argument of 65536. The `double` parameter should not have a default argument.

6.21 Write the prototype and header for a function called `calculate`. The function should have three parameters: an `int`, a reference to a `double`, and a `long` (not necessarily in that order.) Only the `int` parameter should have a default argument, which is 47.

6.22 What is the output of the following program?

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

void test(int = 2, int = 4, int = 6);
```

```
int main()
{
   test();
   test(6);
   test(3, 9);
   test(1, 5, 7);
   return 0;
}

void test (int first, int second, int third)
{
   first += 3;
   second += 6;
   third += 9;
   cout << first << " " << second << " " << third << endl;
}
```

6.23 The following program asks the user to enter two numbers. What is the output of the program if the user enters 12 and 14?

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

void func1(int &, int &);
void func2(int &, int &, int &);
void func3(int, int, int);

int main()
{
   int x = 0, y = 0, z = 0;

   cout << x << " " << y << " " << z << endl;
   func1(x, y);
   cout << x << " " << y << " " << z << endl;
   func2(x, y, z);
   cout << x << " " << y << " " << z << endl;
   func3(x, y, z);
   cout << x << " " << y << " " << z << endl;
   return 0;
}

void func1(int &a, int &b)
{
   cout << "Enter two numbers: ";
   cin >> a >> b;
}
void func2(int &a, int &b, int &c)
{
   b++;
   c--;
   a = b + c;
}

void func3(int a, int b, int c)
{
   a = b - c;
}
```

6.14 Overloading Functions

CONCEPT: **Two or more functions may have the same name, as long as their parameter lists are different.**

Sometimes you will create two or more functions that perform the same operation, but use a different set of parameters or parameters of different data types. For instance, in Program 6-13 there is a `square` function that uses a `double` parameter. But, suppose you also wanted a `square` function that works exclusively with integers, accepting an `int` as its argument. Both functions would do the same thing: return the square of their argument. The only difference is the data type involved in the operation. If you were to use both these functions in the same program, you could assign a unique name to each function. For example, the function that squares an `int` might be named `squareInt`, and the one that squares a `double` might be named `squareDouble`. C++, however, allows you to *overload* function names. That means you may assign the same name to multiple functions, as long as their parameter lists are different. Program 6-27 uses two overloaded `square` functions.

Program 6-27

```
1  // This program uses overloaded functions.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Function prototypes
7  int square(int);
8  double square(double);
9
10 int main()
11 {
12     int userInt;
13     double userFloat;
14
15     // Get an int and a double.
16     cout << fixed << showpoint << setprecision(2);
17     cout << "Enter an integer and a floating-point value: ";
18     cin >> userInt >> userFloat;
19
20     // Display their squares.
21     cout << "Here are their squares: ";
22     cout << square(userInt) << " and " << square(userFloat);
23     return 0;
24 }
25
26 //*************************************************************
27 // Definition of overloaded function square.                  *
28 // This function uses an int parameter, number. It returns the *
29 // square of number as an int.                                *
30 //*************************************************************
```

```
31
32  int square(int number)
33  {
34      return number * number;
35  }
36
37  //*************************************************************
38  // Definition of overloaded function square.                  *
39  // This function uses a double parameter, number. It returns  *
40  // the square of number as a double.                          *
41  //*************************************************************
42
43  double square(double number)
44  {
45      return number * number;
46  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and a floating-point value: 12 4.2 [Enter]
Here are their squares: 144 and 17.64
```

Here are the headers for the `square` functions used in Program 6-27:

```
int square(int number)

double square(double number)
```

In C++, each function has a signature. The *function signature* is the name of the function and the data types of the function's parameters in the proper order. The `square` functions in Program 6-27 would have the following signatures:

```
square(int)

square(double)
```

When an overloaded function is called, C++ uses the function signature to distinguish it from other functions with the same name. In Program 6-27, when an `int` argument is passed to `square`, the version of the function that has an `int` parameter is called. Likewise, when a `double` argument is passed to `square`, the version with a `double` parameter is called.

Note that the function's return value is not part of the signature. The following functions could not be used in the same program because their parameter lists aren't different.

```
int square(int number)
{
   return number * number
}

double square(int number) // Wrong! Parameter lists must differ
{
   return number * number
}
```

Overloading is also convenient when there are similar functions that use a different number of parameters. For example, consider a program with functions that return the sum of integers. One returns the sum of two integers, another returns the sum of three integers, and yet another returns the sum of four integers. Here are their function headers:

```
int sum(int num1, int num2)

int sum(int num1, int num2, int num3)

int sum(int num1, int num2, int num3, int num4)
```

Because the number of parameters is different in each, they all may be used in the same program. Program 6-28 is an example that uses two functions, each named `calcWeeklyPay`, to determine an employee's gross weekly pay. One version of the function uses an `int` and a `double` parameter, while the other version only uses a `double` parameter.

Program 6-28

```
1  // This program demonstrates overloaded functions to calculate
2  // the gross weekly pay of hourly paid or salaried employees.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // Function prototypes
8  void getChoice(char &);
9  double calcWeeklyPay(int, double);
10 double calcWeeklyPay(double);
11
12 int main()
13 {
14     char selection;  // Menu selection
15     int worked;      // Hours worked
16     double rate;     // Hourly pay rate
17     double yearly;   // Yearly salary
18
19     // Set numeric output formatting.
20     cout << fixed << showpoint << setprecision(2);
21
22     // Display the menu and get a selection.
23     cout << "Do you want to calculate the weekly pay of\n";
24     cout << "(H) an hourly paid employee, or \n";
25     cout << "(S) a salaried employee?\n";
26     getChoice(selection);
27
28     // Process the menu selection.
29     switch (selection)
30     {
31         // Hourly paid employee
32         case 'H' :
33         case 'h' : cout << "How many hours were worked? ";
```

```
34                      cin >> worked;
35                      cout << "What is the hourly pay rate? ";
36                      cin >> rate;
37                      cout << "The gross weekly pay is $";
38                      cout << calcWeeklyPay(worked, rate) << endl;
39                      break;
40
41           // Salaried employee
42           case 'S' :
43           case 's' : cout << "What is the annual salary? ";
44                      cin >> yearly;
45                      cout << "The gross weekly pay is $";
46                      cout << calcWeeklyPay(yearly) << endl;
47                      break;
48      }
49      return 0;
50  }
51
52  //*************************************************************
53  // Definition of function getChoice.                          *
54  // The parameter letter is a reference to a char.             *
55  // This function asks the user for an H or an S and returns   *
56  // the validated input.                                       *
57  //*************************************************************
58
59  void getChoice(char & letter)
60  {
61      // Get the user's selection.
62      cout << "Enter your choice (H or S): ";
63      cin >> letter;
64
65      // Validate the selection.
66      while (letter != 'H' && letter != 'h' &&
67             letter != 'S' && letter != 's')
68      {
69           cout << "Please enter H or S: ";
70           cin >> letter;
71      }
72  }
73
74  //*************************************************************
75  // Definition of overloaded function calcWeeklyPay.           *
76  // This function calculates the gross weekly pay of           *
77  // an hourly paid employee. The parameter hours holds the     *
78  // number of hours worked. The parameter payRate holds the    *
79  // hourly pay rate. The function returns the weekly salary.   *
80  //*************************************************************
81
82  double calcWeeklyPay(int hours, double payRate)
83  {
```

(program continues)

Program 6-28 *(continued)*

```
84       return hours * payRate;
85   }
86
87   //*********************************************************
88   // Definition of overloaded function calcWeeklyPay.       *
89   // This function calculates the gross weekly pay of       *
90   // a salaried employee. The parameter holds the employee's *
91   // annual salary. The function returns the weekly salary.  *
92   //*********************************************************
93
94   double calcWeeklyPay(double annSalary)
95   {
96       return annSalary / 52;
97   }
```

Program Output with Example Input Shown in Bold

```
Do you want to calculate the weekly pay of
(H) an hourly paid employee, or
(S) a salaried employee?
Enter your choice (H or S): H [Enter]
How many hours were worked? 40 [Enter]
What is the hourly pay rate? 18.50 [Enter]
The gross weekly pay is $740.00
```

Program Output with Example Input Shown in Bold

```
Do you want to calculate the weekly pay of
(H) an hourly paid employee, or
(S) a salaried employee?
Enter your choice (H or S): S [Enter]
What is the annual salary? 68000.00 [Enter]
The gross weekly pay is $1307.69
```

6.15 The `exit()` Function

CONCEPT: The `exit()` function causes a program to terminate, regardless of which function or control mechanism is executing.

A C++ program stops executing when the `return` statement in function `main` is encountered. When other functions end, however, the program does not stop. Control of the program goes back to the place immediately following the function call. Sometimes, rare circumstances make it necessary to terminate a program in a function other than `main`. To accomplish this, the `exit` function is used.

When the `exit` function is called, it causes the program to stop, regardless of which function contains the call. Program 6-29 demonstrates its use.

Program 6-29

```
1  // This program shows how the exit function causes a program
2  // to stop executing.
3  #include <iostream>
4  #include <cstdlib>  // Needed for the exit function
5  using namespace std;
6
7  void function();    // Function prototype
8
9  int main()
10 {
11    function();
12    return 0;
13 }
14
15 //*************************************************************
16 // This function simply demonstrates that exit can be used *
17 // to terminate a program from a function other than main. *
18 //*************************************************************
19
20 void function()
21 {
22     cout << "This program terminates with the exit function.\n";
23     cout << "Bye!\n";
24     exit(0);
25     cout << "This message will never be displayed\n";
26     cout << "because the program has already terminated.\n";
27 }
```

Program Output

```
This program terminates with the exit function.
Bye!
```

To use the `exit` function, you must include the `cstdlib` header file. Notice the function takes an integer argument. This argument is the exit code you wish the program to pass back to the computer's operating system. This code is sometimes used outside of the program to indicate whether the program ended successfully or as the result of a failure. In Program 6-29, the exit code zero is passed, which commonly indicates a successful exit. If you are unsure which code to use with the `exit` function, there are two named constants, `EXIT_FAILURE` and `EXIT_SUCCESS`, defined in `cstdlib` for you to use. The constant `EXIT_FAILURE` is defined as the termination code that commonly represents an unsuccessful exit under the current operating system. Here is an example of its use:

```
exit(EXIT_FAILURE);
```

The constant `EXIT_SUCCESS` is defined as the termination code that commonly represents a successful exit under the current operating system. Here is an example:

```
exit(EXIT_SUCCESS);
```

NOTE: Generally, the exit code is important only if you know it will be tested outside the program. If it is not used, just pass zero, or `EXIT_SUCCESS`.

WARNING! The `exit()` function unconditionally shuts down your program. Because it bypasses a program's normal logical flow, you should use it with caution.

Checkpoint

6.24 What is the output of the following program?

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

void showVals(double, double);

int main()
{
   double x = 1.2, y = 4.5;

   showVals(x, y);
   return 0;
}

void showVals(double p1, double p2)
{
   cout << p1 << endl;
   exit(0);
   cout << p2 << endl;
}
```

6.25 What is the output of the following program?

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

int manip(int);
int manip(int, int);
int manip(int, double);

int main()
{
   int x = 2, y= 4, z;
   double a = 3.1;

   z = manip(x) + manip(x, y) + manip(y, a);
   cout << z << endl;
   return 0;
}
int manip(int val)
{
   return val + val * 2;
}

int manip(int val1, int val2)
{
   return (val1 + val2) * 2;
}

int manip(int val1, double val2)
{
   return val1 * static_cast<int>(val2);
}
```

6.16 Stubs and Drivers

Stubs and *drivers* are very helpful tools for testing and debugging programs that use functions. They allow you to test the individual functions in a program, in isolation from the parts of the program that call the functions.

A *stub* is a dummy function that is called instead of the actual function it represents. It usually displays a test message acknowledging that it was called, and nothing more. For example, if a stub were used for the `showFees` function in Program 6-10 (the modular health club membership program), it might look like this:

```
void showFees(double memberRate, int months)
{
   cout << "The showFees function was called with "
        << "the following arguments:\n"
        << "memberRate: " << memberRate << endl
        << "months: " << months << endl;
}
```

The following is an example output of the program if it were run with the stub instead of the actual `showFees` function. (A version of the health club program using this stub function is available from the book's companion Web site at www.pearsonhighered.com/gaddis. The program is named `HealthClubWithStub.cpp`.)

```
                    Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 1 [Enter]
For how many months? 4 [Enter]
The showFees function was called with the following arguments:
memberRate: 40.00
months: 4

                    Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 4 [Enter]
```

As you can see, by replacing an actual function with a stub, you can concentrate your testing efforts on the parts of the program that call the function. Primarily, the stub allows you to determine whether your program is calling a function when you expect it to, and to confirm that valid values are being passed to the function. If the stub represents a function that returns a value, then the stub should return a test value. This helps you confirm that the return value is being handled properly. When the parts of the program that call a function are debugged to your satisfaction, you can move on to testing and debugging the actual functions themselves. This is where *drivers* become useful.

A driver is a program that tests a function by simply calling it. If the function accepts arguments, the driver passes test data. If the function returns a value, the driver displays the return value on the screen. This allows you to see how the function performs in isolation from the rest of the program it will eventually be part of. Program 6-30 shows a driver for testing the `showFees` function in the health club membership program.

Program 6-30

```
// This program is a driver for testing the showFees function.
#include <iostream>
using namespace std;

// Prototype
void showFees(double, int);

int main()
{
    // Constants for membership rates
    const double ADULT = 40.0;
    const double SENIOR = 30.0;
    const double CHILD = 20.0;

    // Perform a test for adult membership.
    cout << "Testing an adult membership...\n"
         << "Calling the showFees function with arguments "
         << ADULT << " and 10.\n";
    showFees(ADULT, 10);

    // Perform a test for senior citizen membership.
    cout << "\nTesting a senior citizen membership...\n"
         << "Calling the showFees function with arguments "
         << SENIOR << " and 10.\n";
    showFees(SENIOR, 10);

    // Perform a test for child membership.
    cout << "\nTesting a child membership...\n"
         << "\nCalling the showFees function with arguments "
         << CHILD << " and 10.\n";
    showFees(CHILD, 10);
    return 0;
}

//*************************************************************
// Definition of function showFees. The memberRate parameter holds*
// the monthly membership rate and the months parameter holds the *
// number of months. The function displays the total charges.     *
//*************************************************************

void showFees(double memberRate, int months)
{
    cout << "The total charges are $"
         << (memberRate * months) << endl;
}
```

Program Output

```
Testing an adult membership...
Calling the showFees function with arguments 40 and 10.
The total charges are $400

Testing a senior citizen membership...
Calling the showFees function with arguments 30 and 10.
The total charges are $300

Testing a child membership...

Calling the showFees function with arguments 20 and 10.
The total charges are $200
```

As shown in Program 6-30, a driver can be used to thoroughly test a function. It can repeatedly call the function with different test values as arguments. When the function performs as desired, it can be placed into the actual program it will be part of.

Case Study: See High Adventure Travel Agency Part 1 Case Study on the book's companion Web site at www.pearsonhighered.com/gaddis.

Review Questions and Exercises

Short Answer

1. Why do local variables lose their values between calls to the function in which they are defined?
2. What is the difference between an argument and a parameter variable?
3. Where do you define parameter variables?
4. If you are writing a function that accepts an argument and you want to make sure the function cannot change the value of the argument, what do you do?
5. When a function accepts multiple arguments, does it matter in what order the arguments are passed?
6. How do you return a value from a function?
7. What is the advantage of breaking your application's code into several small procedures?
8. How would a `static` local variable be useful?
9. Give an example where passing an argument by reference would be useful.

Fill-in-the-Blank

10. The _____ is the part of a function definition that shows the function name, return type, and parameter list.
11. If a function doesn't return a value, the word _____ will appear as its return type.
12. Either a function's _____ or its _____ must precede all calls to the function.
13. Values that are sent into a function are called _____.

14. Special variables that hold copies of function arguments are called _____.
15. When only a copy of an argument is passed to a function, it is said to be passed by _____.
16. A(n) _____ eliminates the need to place a function definition before all calls to the function.
17. A(n) _____ variable is defined inside a function and is not accessible outside the function.
18. _____ variables are defined outside all functions and are accessible to any function within their scope.
19. _____ variables provide an easy way to share large amounts of data among all the functions in a program.
20. Unless you explicitly initialize global variables, they are automatically initialized to _____.
21. If a function has a local variable with the same name as a global variable, only the _____ variable can be seen by the function.
22. _____ local variables retain their value between function calls.
23. The _____ statement causes a function to end immediately.
24. _____ arguments are passed to parameters automatically if no argument is provided in the function call.
25. When a function uses a mixture of parameters with and without default arguments, the parameters with default arguments must be defined _____.
26. The value of a default argument must be a(n) _____.
27. When used as parameters, _____ variables allow a function to access the parameter's original argument.
28. Reference variables are defined like regular variables, except there is a(n) _____ in front of the name.
29. Reference variables allow arguments to be passed by _____.
30. The _____ function causes a program to terminate.
31. Two or more functions may have the same name, as long as their _____ are different.

Algorithm Workbench

32. Examine the following function header, then write an example call to the function.

```
void showValue(int quantity)
```

33. The following statement calls a function named `half`. The `half` function returns a value that is half that of the argument. Write the function.

```
result = half(number);
```

34. A program contains the following function.

```
int cube(int num)
{
   return num * num * num;
}
```

Write a statement that passes the value 4 to this function and assigns its return value to the variable `result`.

35. Write a function named `timesTen` that accepts an argument. When the function is called, it should display the product of its argument multiplied times 10.
36. A program contains the following function.

```
void display(int arg1, double arg2, char arg3)
{
   cout << "Here are the values: "
        << arg1 << " " << arg2 << " "
        << arg3 << endl;
}
```

Write a statement that calls the procedure and passes the following variables to it:

```
int age;
double income;
char initial;
```

37. Write a function named `getNumber` that uses a reference parameter variable to accept an integer argument. The function should prompt the user to enter a number in the range of 1 through 100. The input should be validated and stored in the parameter variable.

True or False

38. T F Functions should be given names that reflect their purpose.
39. T F Function headers are terminated with a semicolon.
40. T F Function prototypes are terminated with a semicolon.
41. T F If other functions are defined before `main`, the program still starts executing at function `main`.
42. T F When a function terminates, it always branches back to `main`, regardless of where it was called from.
43. T F Arguments are passed to the function parameters in the order they appear in the function call.
44. T F The scope of a parameter is limited to the function which uses it.
45. T F Changes to a function parameter always affect the original argument as well.
46. T F In a function prototype, the names of the parameter variables may be left out.
47. T F Many functions may have local variables with the same name.
48. T F Overuse of global variables can lead to problems.
49. T F Static local variables are not destroyed when a function returns.
50. T F All static local variables are initialized to –1 by default.
51. T F Initialization of static local variables only happens once, regardless of how many times the function in which they are defined is called.
52. T F When a function with default arguments is called and an argument is left out, all arguments that come after it must be left out as well.
53. T F It is not possible for a function to have some parameters with default arguments and some without.

54. T F The `exit` function can only be called from `main`.
55. T F A stub is a dummy function that is called instead of the actual function it represents.

Find the Errors

Each of the following functions has errors. Locate as many errors as you can.

56.
```
void total(int value1, value2, value3)
{
   return value1 + value2 + value3;
}
```

57.
```
double average(int value1, int value2, int value3)
{
   double average;

   average = value1 + value2 + value3 / 3;
}
```

58.
```
void area(int length = 30, int width)
{
   return length * width;
}
```

59.
```
void getValue(int value&)
{
   cout << "Enter a value: ";
   cin >> value&;
}
```

60. (*Overloaded functions*)
```
int getValue()
{
   int inputValue;
   cout << "Enter an integer: ";
   cin >> inputValue;
   return inputValue;
}
double getValue()
{
   double inputValue;
   cout << "Enter a floating-point number: ";
   cin >> inputValue;
   return inputValue;
}
```

Programming Challenges

1. **Markup**

VideoNote
Solving the Markup Problem

Write a program that asks the user to enter an item's wholesale cost and its markup percentage. It should then display the item's retail price. For example:

- If an item's wholesale cost is 5.00 and its markup percentage is 100%, then the item's retail price is 10.00.

- If an item's wholesale cost is 5.00 and its markup percentage is 50%, then the item's retail price is 7.50.

The program should have a function named `calculateRetail` that receives the wholesale cost and the markup percentage as arguments and returns the retail price of the item.

Input Validation: Do not accept negative values for either the wholesale cost of the item or the markup percentage.

2. Rectangle Area—Complete the Program

If you have downloaded this book's source code from the companion Web site, you will find a partially written program named `AreaRectangle.cpp` in the Chapter 06 folder. (The companion Web site is at www.pearsonhighered.com/gaddis.) Your job is to complete the program. When it is complete, the program will ask the user to enter the width and length of a rectangle and then display the rectangle's area. The program calls the following functions, which have not been written:

- `getLength` – This function should ask the user to enter the rectangle's length and then return that value as a `double`.
- `getWidth` – This function should ask the user to enter the rectangle's width and then return that value as a `double`.
- `getArea` – This function should accept the rectangle's length and width as arguments and return the rectangle's area. The area is calculated by multiplying the length by the width.
- `displayData` – This function should accept the rectangle's length, width, and area as arguments and display them in an appropriate message on the screen.

3. Winning Division

Write a program that determines which of a company's four divisions (Northeast, Southeast, Northwest, and Southwest) had the greatest sales for a quarter. It should include the following two functions, which are called by `main`.

- `double getSales()` is passed the name of a division. It asks the user for a division's quarterly sales figure, validates the input, then returns it. It should be called once for each division.
- `void findHighest()` is passed the four sales totals. It determines which is the largest and prints the name of the high grossing division, along with its sales figure.

Input Validation: Do not accept dollar amounts less than $0.00.

4. Safest Driving Area

Write a program that determines which of five geographic regions within a major city (north, south, east, west, and central) had the fewest reported automobile accidents last year. It should have the following two functions, which are called by `main`.

- `int getNumAccidents()` is passed the name of a region. It asks the user for the number of automobile accidents reported in that region during the last year, validates the input, then returns it. It should be called once for each city region.
- `void findLowest()` is passed the five accident totals. It determines which is the smallest and prints the name of the region, along with its accident figure.

Input Validation: Do not accept an accident number that is less than 0.

5. **Falling Distance**

   When an object is falling because of gravity, the following formula can be used to determine the distance the object falls in a specific time period:

   $d = {}^1/_2 gt^2$

   The variables in the formula are as follows: d is the distance in meters, g is 9.8, and t is the amount of time, in seconds, that the object has been falling.

   Write a function named `fallingDistance` that accepts an object's falling time (in seconds) as an argument. The function should return the distance, in meters, that the object has fallen during that time interval. Write a program that demonstrates the function by calling it in a loop that passes the values 1 through 10 as arguments and displays the return value.

6. **Kinetic Energy**

   In physics, an object that is in motion is said to have kinetic energy. The following formula can be used to determine a moving object's kinetic energy:

   $KE = {}^1/_2\, mv^2$

   The variables in the formula are as follows: KE is the kinetic energy, m is the object's mass in kilograms, and v is the object's velocity, in meters per second.

   Write a function named `kineticEnergy` that accepts an object's mass (in kilograms) and velocity (in meters per second) as arguments. The function should return the amount of kinetic energy that the object has. Demonstrate the function by calling it in a program that asks the user to enter values for mass and velocity.

7. **Celsius Temperature Table**

   The formula for converting a temperature from Fahrenheit to Celsius is

   $$C = \frac{5}{9}(F - 32)$$

   where F is the Fahrenheit temperature and C is the Celsius temperature. Write a function named `celsius` that accepts a Fahrenheit temperature as an argument. The function should return the temperature, converted to Celsius. Demonstrate the function by calling it in a loop that displays a table of the Fahrenheit temperatures 0 through 20 and their Celsius equivalents.

8. **Coin Toss**

   Write a function named `coinToss` that simulates the tossing of a coin. When you call the function, it should generate a random number in the range of 1 through 2. If the random number is 1, the function should display "heads." If the random number is 2, the function should display "tails." Demonstrate the function in a program that asks the user how many times the coin should be tossed and then simulates the tossing of the coin that number of times.

9. **Present Value**

   Suppose you want to deposit a certain amount of money into a savings account and then leave it alone to draw interest for the next 10 years. At the end of 10 years you would like to have $10,000 in the account. How much do you need to deposit today to

make that happen? You can use the following formula, which is known as the present value formula, to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the **present value**, or the amount that you need to deposit today.
- F is the **future value** that you want in the account. (In this case, F is $10,000.)
- r is the **annual interest rate**.
- n is the **number of years** that you plan to let the money sit in the account.

Write a program that has a function named `presentValue` that performs this calculation. The function should accept the future value, annual interest rate, and number of years as arguments. It should return the present value, which is the amount that you need to deposit today. Demonstrate the function in a program that lets the user experiment with different values for the formula's terms.

10. Future Value

Suppose you have a certain amount of money in a savings account that earns compound monthly interest, and you want to calculate the amount that you will have after a specific number of months. The formula, which is known as the future value formula, is:

$$F = P \times (1 + i)^t$$

The terms in the formula are as follows:

- F is the **future value** of the account after the specified time period.
- P is the **present value** of the account.
- i is the **monthly interest rate**.
- t is the **number of months**.

Write a program that prompts the user to enter the account's present value, monthly interest rate, and the number of months that the money will be left in the account. The program should pass these values to a function named `futureValue` that returns the future value of the account, after the specified number of months. The program should display the account's future value.

11. Lowest Score Drop

Write a program that calculates the average of a group of test scores, where the lowest score in the group is dropped. It should use the following functions:

- `void getScore()` should ask the user for a test score, store it in a reference parameter variable, and validate it. This function should be called by `main` once for each of the five scores to be entered.
- `void calcAverage()` should calculate and display the average of the four highest scores. This function should be called just once by `main` and should be passed the five scores.
- `int findLowest()` should find and return the lowest of the five scores passed to it. It should be called by `calcAverage`, which uses the function to determine which of the five scores to drop.

Input Validation: Do not accept test scores lower than 0 or higher than 100.

12. Star Search

A particular talent competition has five judges, each of whom awards a score between 0 and 10 to each performer. Fractional scores, such as 8.3, are allowed. A performer's final score is determined by dropping the highest and lowest score received, then averaging the three remaining scores. Write a program that uses this method to calculate a contestant's score. It should include the following functions:

- `void getJudgeData()` should ask the user for a judge's score, store it in a reference parameter variable, and validate it. This function should be called by `main` once for each of the five judges.
- `void calcScore()` should calculate and display the average of the three scores that remain after dropping the highest and lowest scores the performer received. This function should be called just once by `main` and should be passed the five scores.

The last two functions, described below, should be called by `calcScore`, which uses the returned information to determine which of the scores to drop.

- `int findLowest()` should find and return the lowest of the five scores passed to it.
- `int findHighest()` should find and return the highest of the five scores passed to it.

Input Validation: Do not accept judge scores lower than 0 or higher than 10.

13. Days Out

Write a program that calculates the average number of days a company's employees are absent. The program should have the following functions:

- A function called by `main` that asks the user for the number of employees in the company. This value should be returned as an `int`. (The function accepts no arguments.)
- A function called by `main` that accepts one argument: the number of employees in the company. The function should ask the user to enter the number of days each employee missed during the past year. The total of these days should be returned as an `int`.
- A function called by `main` that takes two arguments: the number of employees in the company and the total number of days absent for all employees during the year. The function should return, as a `double`, the average number of days absent. (This function does not perform screen output and does not ask the user for input.)

Input Validation: Do not accept a number less than 1 for the number of employees. Do not accept a negative number for the days any employee missed.

14. Order Status

The Middletown Wholesale Copper Wire Company sells spools of copper wiring for $100 each. Write a program that displays the status of an order. The program should have a function that asks for the following data:

- The number of spools ordered.
- The number of spools in stock.
- Whether there are special shipping and handling charges.

(Shipping and handling is normally $10 per spool.) If there are special charges, the program should ask for the special charges per spool.

The gathered data should be passed as arguments to another function that displays

- The number of spools ready to ship from current stock.
- The number of spools on backorder (if the number ordered is greater than what is in stock).
- Subtotal of the portion ready to ship (the number of spools ready to ship times $100).
- Total shipping and handling charges on the portion ready to ship.
- Total of the order ready to ship.

The shipping and handling parameter in the second function should have the default argument 10.00.

Input Validation: Do not accept numbers less than 1 for spools ordered. Do not accept a number less than 0 for spools in stock or shipping and handling charges.

15. Overloaded Hospital

Write a program that computes and displays the charges for a patient's hospital stay. First, the program should ask if the patient was admitted as an in-patient or an out-patient. If the patient was an in-patient, the following data should be entered:

- The number of days spent in the hospital
- The daily rate
- Hospital medication charges
- Charges for hospital services (lab tests, etc.)

The program should ask for the following data if the patient was an out-patient:

- Charges for hospital services (lab tests, etc.)
- Hospital medication charges

The program should use two overloaded functions to calculate the total charges. One of the functions should accept arguments for the in-patient data, while the other function accepts arguments for out-patient information. Both functions should return the total charges.

Input Validation: Do not accept negative numbers for any data.

16. Population

In a population, the birth rate is the percentage increase of the population due to births, and the death rate is the percentage decrease of the population due to deaths. Write a program that displays the size of a population for any number of years. The program should ask for the following data:

- The starting size of a population
- The annual birth rate
- The annual death rate
- The number of years to display

Write a function that calculates the size of the population for a year. The formula is

```
N = P + BP - DP
```

where `N` is the new population size, `P` is the previous population size, `B` is the birth rate, and `D` is the death rate.

Input Validation: Do not accept numbers less than 2 for the starting size. Do not accept negative numbers for birth rate or death rate. Do not accept numbers less than 1 for the number of years.

17. Transient Population

Modify Programming Challenge 16 to also consider the effect on population caused by people moving into or out of a geographic area. Given as input a starting population size, the annual birth rate, the annual death rate, the number of individuals who typically move into the area each year, and the number of individuals who typically leave the area each year, the program should project what the population will be `numYears` from now. You can either prompt the user to input a value for `numYears`, or you can set it within the program.

Input Validation: Do not accept numbers less than 2 for the starting size. Do not accept negative numbers for birth rate, death rate, arrivals, or departures.

18. Paint Job Estimator

A painting company has determined that for every 110 square feet of wall space, one gallon of paint and eight hours of labor will be required. The company charges $25.00 per hour for labor. Write a modular program that allows the user to enter the number of rooms that are to be painted and the price of the paint per gallon. It should also ask for the square feet of wall space in each room. It should then display the following data:

- The number of gallons of paint required
- The hours of labor required
- The cost of the paint
- The labor charges
- The total cost of the paint job

Input validation: Do not accept a value less than 1 for the number of rooms. Do not accept a value less than $10.00 for the price of paint. Do not accept a negative value for square footage of wall space.

19. Using Files—Hospital Report

Modify Programming Challenge 15, Overloaded Hospital, to write the report it creates to a file.

20. Stock Profit

The profit from the sale of a stock can be calculated as follows:

$$\text{Profit} = ((NS \times SP) - SC) - ((NS \times PP) + PC)$$

where NS is the number of shares, SP is the sale price per share, SC is the sale commission paid, PP is the purchase price per share, and PC is the purchase commission paid. If the calculation yields a positive value, then the sale of the stock resulted in a profit. If the calculation yields a negative number, then the sale resulted in a loss.

Write a function that accepts as arguments the number of shares, the purchase price per share, the purchase commission paid, the sale price per share, and the sale commission paid. The function should return the profit (or loss) from the sale of stock.

Demonstrate the function in a program that asks the user to enter the necessary data and displays the amount of the profit or loss.

21. Multiple Stock Sales

Use the function that you wrote for Programming Challenge 20 (Stock Profit) in a program that calculates the total profit or loss from the sale of multiple stocks. The program should ask the user for the number of stock sales and the necessary data for each stock sale. It should accumulate the profit or loss for each stock sale and then display the total

22. `isPrime` Function

A prime number is a number that is only evenly divisible by itself and 1. For example, the number 5 is prime because it can only be evenly divided by 1 and 5. The number 6, however, is not prime because it can be divided evenly by 1, 2, 3, and 6.

Write a function name `isPrime`, which takes an integer as an argument and returns true if the argument is a prime number, or false otherwise. Demonstrate the function in a complete program.

TIP: Recall that the % operator divides one number by another, and returns the remainder of the division. In an expression such as `num1 % num2`, the % operator will return 0 if `num1` is evenly divisible by `num2`.

23. Prime Number List

Use the `isPrime` function that you wrote in Programming Challenge 22 in a program that stores a list of all the prime numbers from 1 through 100 in a file.

24. Rock, Paper, Scissors Game

Write a program that lets the user play the game of Rock, Paper, Scissors against the computer. The program should work as follows.

1. When the program begins, a random number in the range of 1 through 3 is generated. If the number is 1, then the computer has chosen rock. If the number is 2, then the computer has chosen paper. If the number is 3, then the computer has chosen scissors. (Don't display the computer's choice yet.)
2. The user enters his or her choice of "rock", "paper", or "scissors" at the keyboard. (You can use a menu if you prefer.)
3. The computer's choice is displayed.
4. A winner is selected according to the following rules:
   - If one player chooses rock and the other player chooses scissors, then rock wins. (The rock smashes the scissors.)
   - If one player chooses scissors and the other player chooses paper, then scissors wins. (Scissors cuts paper.)
   - If one player chooses paper and the other player chooses rock, then paper wins. (Paper wraps rock.)
   - If both players make the same choice, the game must be played again to determine the winner.

Be sure to divide the program into functions that perform each major task.

Group Project

25. Travel Expenses

This program should be designed and written by a team of students. Here are some suggestions:

- One student should design function `main`, which will call the other functions in the program. The remainder of the functions will be designed by other members of the team.
- The requirements of the program should be analyzed so each student is given about the same workload.
- The parameters and return types of each function should be decided in advance.
- Stubs and drivers should be used to test and debug the program.
- The program can be implemented as a multifile program, or all the functions can be cut and pasted into the main file.

Here is the assignment: Write a program that calculates and displays the total travel expenses of a businessperson on a trip. The program should have functions that ask for and return the following:

- The total number of days spent on the trip
- The time of departure on the first day of the trip, and the time of arrival back home on the last day of the trip
- The amount of any round-trip airfare
- The amount of any car rentals
- Miles driven, if a private vehicle was used. Calculate the vehicle expense as $0.27 per mile driven
- Parking fees (The company allows up to $6 per day. Anything in excess of this must be paid by the employee.)
- Taxi fees, if a taxi was used anytime during the trip (The company allows up to $10 per day, for each day a taxi was used. Anything in excess of this must be paid by the employee.)
- Conference or seminar registration fees
- Hotel expenses (The company allows up to $90 per night for lodging. Anything in excess of this must be paid by the employee.)
- The amount of *each* meal eaten. On the first day of the trip, breakfast is allowed as an expense if the time of departure is before 7 a.m. Lunch is allowed if the time of departure is before 12 noon. Dinner is allowed on the first day if the time of departure is before 6 p.m. On the last day of the trip, breakfast is allowed if the time of arrival is after 8 a.m. Lunch is allowed if the time of arrival is after 1 p.m. Dinner is allowed on the last day if the time of arrival is after 7 p.m. The program should only ask for the amounts of allowable meals. (The company allows up to $9 for breakfast, $12 for lunch, and $16 for dinner. Anything in excess of this must be paid by the employee.)

The program should calculate and display the total expenses incurred by the businessperson, the total allowable expenses for the trip, the excess that must be reimbursed by the businessperson, if any, and the amount saved by the businessperson if the expenses were under the total allowed.

Input Validation: Do not accept negative numbers for any dollar amount or for miles driven in a private vehicle. Do not accept numbers less than 1 for the number of days. Only accept valid times for the time of departure and the time of arrival.