CHAPTER 9

Advanced Modularization Techniques

After studying this chapter, you will be able to:

- Write functions that require no parameters
- Write functions that require a single parameter
- Write functions that require multiple parameters
- Write functions that return values
- Pass entire arrays and single elements of an array to a function
- Pass arguments to functions by reference and by address
- Overload functions
- Use C++ built-in functions

In Chapter 2 of this book, you learned that local variables are variables that are declared within the function that uses them. You also learned that most programs consist of a main function that contains the mainline logic and then calls other functions to get specific work done in the program.

In this chapter, you learn more about functions in C++. You learn how to write functions that require no parameters, how to write functions that require a single parameter, how to write functions that require multiple parameters, and how to write functions that return a value. You also learn how to pass an array to a function, how to pass arguments by reference and by address, how to overload a function, and how to use some of the built-in functions found in C++. To help you learn about functions, you will study some C++ programs that implement the logic and design presented in *Programming Logic and Design, Eighth Edition*, by Joyce Farrell.

You should do the exercises and labs in this chapter after you have finished Chapter 9 of *Programming Logic and Design.*

Writing Functions with No Parameters

To begin learning about functions, review the C++ code for the Customer Bill program shown in Figure 9-1. Notice the line numbers in front of each line of code in this program. These line numbers are not actually part of the program but are included for reference only.

```
 1  /* Program Name: CustomerBill.cpp
 2     Function: This program uses a function to print a company name and address
 3     and then prints a customer name and balance.
 4     Input:  Interactive
 5     Output: Company name and address, customer name and balance
 6  */
 7  #include <iostream>
 8  #include <string>
 9  void nameAndAddress(); // function declaration
10  using namespace std;
11  int main()
12  {
13     // Declare variables local to main()
14     string firstName;
15     string lastName;
16     double balance;
17
18     // Get interactive input
19     cout << "Enter customer's first name: ";
20     cin >> firstName;
21     cout << "Enter customer's last name: ";
22     cin >> lastName;
23     cout << "Enter customer's balance: ";
24     cin >> balance;
```

Figure 9-1 C++ code for Customer Bill program *(continues)*

(continued)

```
25
26      // Call nameAndAddress() function
27      nameAndAddress();
28      // Output customer name and address
29      cout << "Customer Name:  " << firstName << " " << lastName << endl;
30      cout << "Customer Balance:  " << balance << endl;
31
32      return 0;
33   } // End of main() function
34
35   void nameAndAddress()
36   {
37      // Declare and initialize local, constant Strings
38      const string ADDRESS_LINE1 = "ABC Manufacturing";
39      const string ADDRESS_LINE2 = "47 Industrial Lane";
40      const string ADDRESS_LINE3 = "Wild Rose, WI 54984";
41
42      // Output
43      cout << ADDRESS_LINE1 << endl;
44      cout << ADDRESS_LINE2 << endl;
45      cout << ADDRESS_LINE3 << endl;
46   }  // End of nameAndAddress() function
```

Figure 9-1 C++ code for Customer Bill program

On lines 1–6, you see a multiline comment that describes the Customer Bill program followed by two `#include` preprocessor directives on lines 7 and 8 that give the program the ability to perform input and output and the ability to use `strings`. On line 9, you see the function declaration for the `nameAndAddress()` function. As with variables in C++, you must declare a function before you can call the function in your program. A **function declaration** (also known as a **function prototype**) should specify the data type of the value the function returns, the name of the function, and the data type of each of its arguments. On line 9 you see that the `nameAndAddress()` function returns nothing (`void`) and expects no arguments.

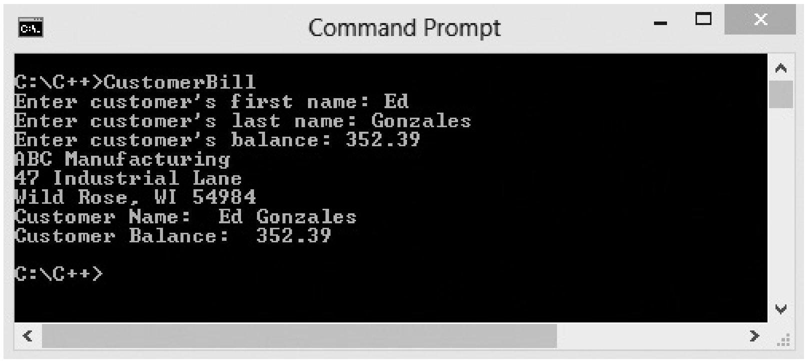
The program begins execution with the `main()` function, which is shown on line 11. This function contains the declaration of three variables (lines 14, 15, and 16), `firstName`, `lastName`, and `balance`, which are local to the `main()` function. Next, on lines 19 through 24, interactive input statements retrieve values for `firstName`, `lastName`, and `balance`.

The function `nameAndAddress()` is then called on line 27, with no arguments listed within its parentheses. Remember that **arguments**, which are sometimes called **actual parameters**, are data items sent to functions. There are no arguments for the `nameAndAddress()` function because this function requires no data. You will learn about passing arguments to functions later in this chapter. The two statements on lines 29 and 30 in the `main()` function are print statements that output the customer's `firstName`, `lastName`, and `balance`.

Next, on line 35, you see the header for the `nameAndAddress()` function. The **header** begins with the `void` keyword, followed by the function name, which is `nameAndAddress`. As you learned in Chapter 1 of this book, the `void` keyword indicates that the `nameAndAddress()` function does not return a value. You learn more about functions that return values later in this chapter.

Also, notice there are no formal parameters within the parentheses. Remember that **formal parameters** are the variables in the function header that accept the values from the actual parameters. (You will learn about writing functions that accept parameters in the next section of this chapter.) In the next part of the Customer Bill program, you see three constants that are local to the `nameAndAddress()` function: `ADDRESS_LINE1`, `ADDRESS_LINE2`, and `ADDRESS_LINE3`. These constants are declared and initialized on lines 38, 39, and 40, and then printed on lines 43, 44, and 45.

When the input to this program is Ed Gonzales (name) and 352.39 (balance), the output is shown in Figure 9-2.



```
C:\C++>CustomerBill
Enter customer's first name: Ed
Enter customer's last name: Gonzales
Enter customer's balance: 352.39
ABC Manufacturing
47 Industrial Lane
Wild Rose, WI 54984
Customer Name:  Ed Gonzales
Customer Balance:  352.39

C:\C++>
```

Figure 9-2 Output from the Customer Bill program

Exercise 9-1: Writing Functions with No Parameters

In this exercise, you use what you have learned about writing functions with no parameters to answer Questions 1–2.

1. Given the following function calls, write the function's header and function declaration:

   a. `printMailingLabel();`

   ____

   b. `displayOrderNumbers();`

   ____

   c. `displayTVListing();`

   ____

2. Given the following function headers, write a function call:

   a. `void printCellPhoneNumbers()`

   ____

   b. `void displayTeamMembers()`

   ____

   c. `void showOrderInfo()`

   ____

Lab 9-1: Writing Functions with No Parameters

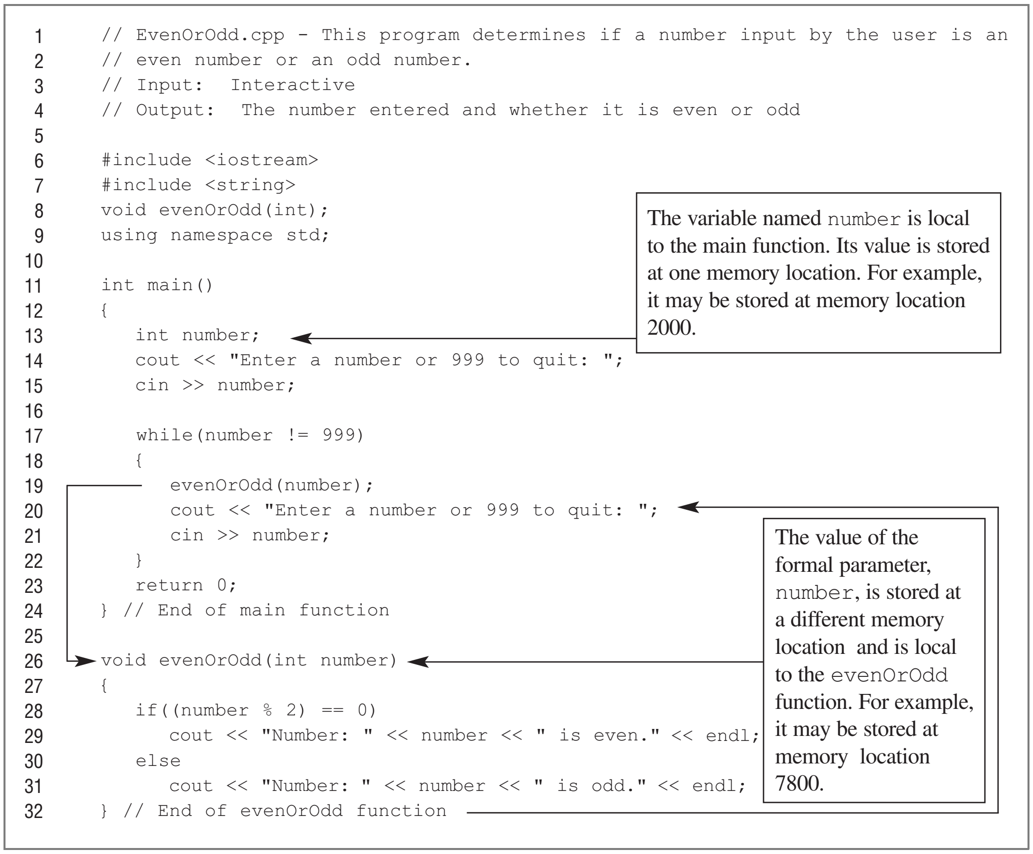
In this lab, you complete a partially prewritten C++ program that includes functions with no parameters. The program asks the user if he or she has preregistered for tickets to an art show. If the user has preregistered, the program should call a function named `discount()` that displays the message, `"You are preregistered and qualify for a 5 percent discount."` If the user has not preregistered, the program should call a function named `noDiscount()` that displays the message, `"Sorry, you did not preregister and do not qualify for a 5 percent discount."` The source code file provided for this lab includes the necessary variable declarations and the input statement. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `ArtShowDiscount.cpp` using Notepad or the text editor of your choice.
2. Write the C++ statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file, `ArtShowDiscount.cpp`.
5. Execute the program.

Writing Functions that Require a Single Parameter

As you learned in *Programming Logic and Design*, some functions require data to accomplish their task. You also learned that designing a program that sends data (which can be different each time the program runs) to a function (which does not change) keeps you from having to write multiple functions to handle similar situations. For example, suppose you are writing a program that has to determine if a number is even or odd. It is certainly better to write a single function that receives a number entered by the user in the main program than to write individual functions for every number.

In Figure 9-3, you see the C++ code for a program that includes a function that can determine if a number is odd or even. The line numbers are not actually part of the program but are included for reference only. The program allows the user to enter a number, and then it passes that number to a function as an argument. After it receives the argument, the function can determine if the number is an even number or an odd number.



```
1   // EvenOrOdd.cpp - This program determines if a number input by the user is an
2   // even number or an odd number.
3   // Input:  Interactive
4   // Output:  The number entered and whether it is even or odd
5
6   #include <iostream>
7   #include <string>
8   void evenOrOdd(int);
9   using namespace std;
10
11  int main()
12  {
13     int number;
14     cout << "Enter a number or 999 to quit: ";
15     cin >> number;
16
17     while(number != 999)
18     {
19        evenOrOdd(number);
20        cout << "Enter a number or 999 to quit: ";
21        cin >> number;
22     }
23     return 0;
24  } // End of main function
25
26  void evenOrOdd(int number)
27  {
28     if((number % 2) == 0)
29        cout << "Number: " << number << " is even." << endl;
30     else
31        cout << "Number: " << number << " is odd." << endl;
32  } // End of evenOrOdd function
```

The variable named `number` is local to the main function. Its value is stored at one memory location. For example, it may be stored at memory location 2000.

The value of the formal parameter, `number`, is stored at a different memory location and is local to the `evenOrOdd` function. For example, it may be stored at memory location 7800.

Figure 9-3 C++ code for Even or Odd program

On line 14 in this program, the user is asked to enter a number or the sentinel value, 999, when he or she is finished entering numbers and wants to quit the program. (You learned about sentinel values in Chapter 5 of this book.) On line 15, the input value is retrieved and then stored in the variable named `number`. Next, if the user did not enter the sentinel value 999, the `while` loop is entered and the function named `evenOrOdd()` is called (line 19) using the following syntax:

```
evenOrOdd(number);
```

Notice the function declaration on line 8 specifies that the `evenOrOdd()` function does not return a value (`void`) and expects a single argument of data type `int`.

The int variable number is placed within the parentheses, which means the value of number is passed to the evenOrOdd() function. This is referred to as passing an argument by value. **Passing an argument by value** means that a copy of the value of the argument is passed to the function. Within the function, the value is stored in the formal parameter at a different memory location, and it is considered local to that function. In this example, as shown on line 26, the value is stored in the formal parameter named number.

It is important that the data types of the formal parameter and the actual parameter are the same.

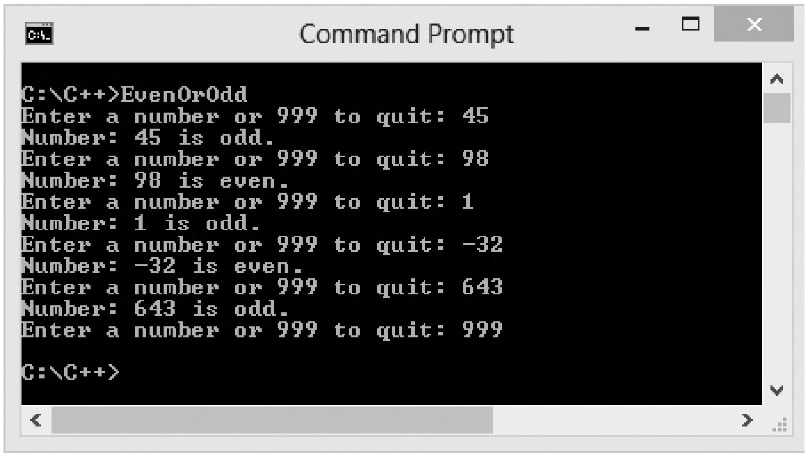
Program control is now transferred to the evenOrOdd() function. The header for the evenOrOdd() function on line 26 includes the void keyword, so you know the function will not return a value. The name of the function follows, and within the parentheses that follow the function name, the parameter number is given a local name and declared as the int data type.

Remember that even though the parameter number has the same name as the local variable number in the main() function, the values are stored at different memory locations. Figure 9-3 illustrates that the variable number that is local to main() is stored at one memory location and the parameter number in the evenOrOdd() function is stored at a different memory location.

Within the function on line 28, the modulus operator (%) is used in the test portion of the if statement to determine if the value of the local number is even or odd. The user is then informed if number is even (line 29) or odd (line 31), and program control is transferred back to the statement that follows the call to evenOrOdd() in the main() function (line 20).

Back in the main() function, the user is asked to enter another number on line 20, and the while loop continues to execute, calling the evenOrOdd() function with a new input value. The loop is exited when the user enters the sentinel value 999 and the program ends. When the input to this program is 45, 98, 1, -32, 643, and 999, the output is shown in Figure 9-4.

In the next section, you will learn how to pass more than one value to a function.



```
C:\C++>EvenOrOdd
Enter a number or 999 to quit: 45
Number: 45 is odd.
Enter a number or 999 to quit: 98
Number: 98 is even.
Enter a number or 999 to quit: 1
Number: 1 is odd.
Enter a number or 999 to quit: -32
Number: -32 is even.
Enter a number or 999 to quit: 643
Number: 643 is odd.
Enter a number or 999 to quit: 999

C:\C++>
```

Figure 9-4 Output from the Even or Odd program

Exercise 9-2: Writing Functions that Require a Single Parameter

In this exercise, you use what you have learned about writing functions that require a single parameter to answer Questions 1–2.

1. Given the following variable declarations and function calls, write the function's header and function declaration:

   a.
   ```
   string name;
   printNameBadge(name);
   ```
   ____

   b.
   ```
   double side_length;
   calculateSquareArea(side_length);
   ```
   ____

   c.
   ```
   int hours;
   displaySecondsInMinutes(minutes);
   ```
   ____

2. Given the following function headers and variable declarations, write a function call:

   a.
   ```
   string petName = "Mindre";
   void displayPetName(string petName)
   ```
   ____

   b.
   ```
   int currentMonth;
   void printBirthdays(int month)
   ```
   ____

   c.
   ```
   string password;
   void checkValidPassword(string id)
   ```
   ____

Lab 9-2: Writing Functions that Require a Single Parameter

In this lab, you complete a partially written C++ program that includes two functions that require a single parameter. The program continuously prompts the user for an integer until the user enters 0. The program then passes the value to a function that computes the sum of all the whole numbers from 1 up to and including the entered number. Next, the program passes the value to another function that computes the product of all the whole numbers up to and including the entered number. The source code file provided for this lab includes the necessary variable declarations and the input statement. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `SumAndProduct.cpp` using Notepad or the text editor of your choice.
2. Write the C++ statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `SumAndProduct.cpp`.
5. Execute the program.

Writing Functions that Require Multiple Parameters

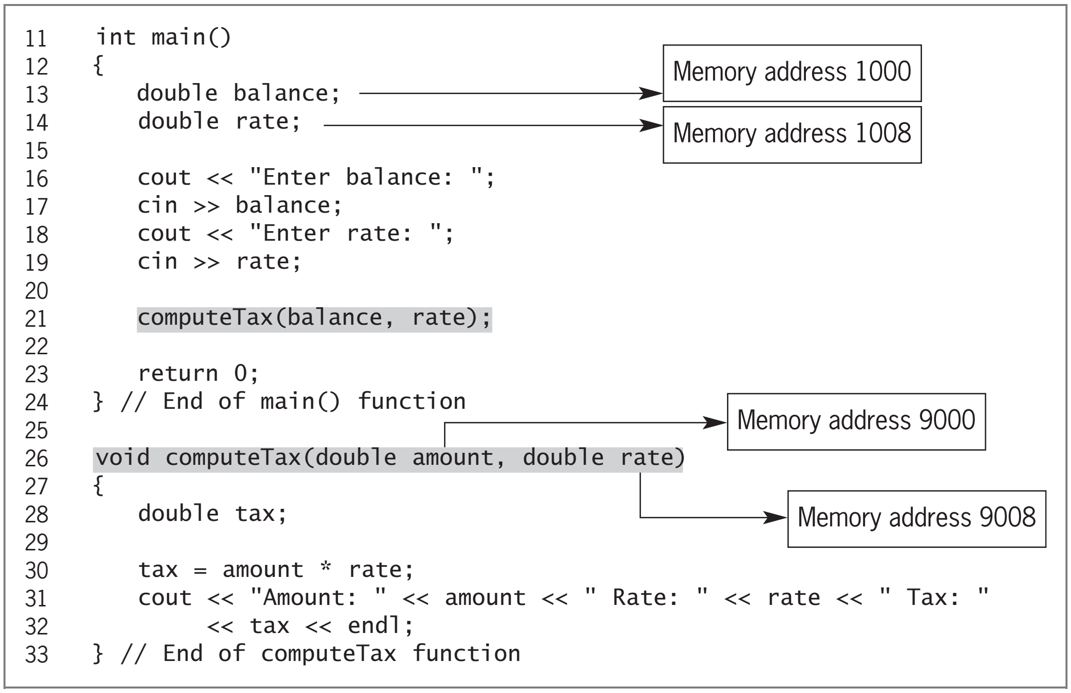
In Chapter 9 of *Programming Logic and Design,* you learned that a function often requires more than one parameter in order to accomplish its task. To specify that a function requires multiple parameters, you include a list of data types and local identifiers separated by commas as part of the function's header. To call a function that expects multiple parameters, you list the actual parameters (separated by commas) in the call to the function.

In Figure 9-5, you see the C++ code for a program that includes a function named `computeTax()` that you designed in *Programming Logic and Design*. The line numbers are not actually part of the program but are included for reference only.

```
 1   // ComputeTax.cpp - This program computes tax given a balance
 2   // and a rate
 3   // Input:  Interactive
 4   // Output:  The balance, tax rate, and computed tax
 5
 6   #include <iostream>
 7   #include <string>
 8   void computeTax(double, double);
 9   using namespace std;
10
```

Figure 9-5 C++ code for the Compute Tax program *(continues)*

(continued)



```
11   int main()
12   {
13      double balance;
14      double rate;
15
16      cout << "Enter balance: ";
17      cin >> balance;
18      cout << "Enter rate: ";
19      cin >> rate;
20
21      computeTax(balance, rate);
22
23      return 0;
24   } // End of main() function
25
26   void computeTax(double amount, double rate)
27   {
28      double tax;
29
30      tax = amount * rate;
31      cout << "Amount: " << amount << " Rate: " << rate << " Tax: "
32           << tax << endl;
33   } // End of computeTax function
```

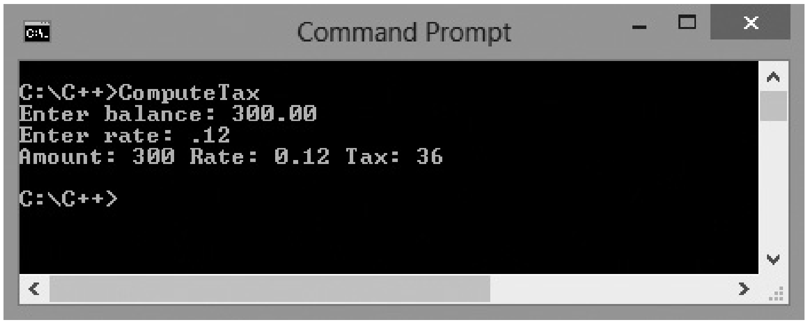
Figure 9-5 C++ code for the Compute Tax program

In C++, when you write a function that expects more than one parameter, you must list the parameters separately, even if they have the same data type.

There is no limit to the number of arguments you can pass to a function, but when multiple arguments are passed to a function, the call to the function and the function's header must match. This means that the number of arguments/parameters must be the same, their data types must be the same, and the order in which they are listed must be the same.

Notice the function declaration on line 8 specifies two parameters (arguments) that are both data type `double`. This agrees with the two `double`s passed to `computeTax()` on line 21 and received as two `double`s in the function header on line 26.

In the C++ code shown in Figure 9-5, you see that the highlighted call to `computeTax()` on line 21 includes the names of the local variables `balance` and `rate` within the parentheses and that they are separated by commas. These are the arguments (actual parameters) that are passed to the `computeTax()` function. You can also see that the `computeTax()` function header on line 26 is highlighted and includes two formal parameters, `double amount` and `double rate`, listed within parentheses and separated by commas. The value of the variable named `balance` is passed by value to the `computeTax()` function as an actual parameter and is stored in the formal parameter named `amount`. The value of the variable named `rate` is passed by value to the `computeTax()` function as an actual parameter and is stored in the formal parameter named `rate`. As illustrated in Figure 9-5, it does not matter that one of the parameters being passed, `rate`, has the same name as the parameter received, `rate`, because they occupy different memory locations. When the input to this program is 300.00 (`balance`) and .12 (`rate`), the output is shown in Figure 9-6.



```
C:\C++>ComputeTax
Enter balance: 300.00
Enter rate: .12
Amount: 300 Rate: 0.12 Tax: 36

C:\C++>
```

Figure 9-6 Output from the Compute Tax program

Exercise 9-3: Writing Functions that Require Multiple Parameters

In this exercise, you use what you have learned about writing functions that require multiple parameters to answer Questions 1–2.

1. Given the following function calls and variable declarations, write the function's header and function declaration:

   a.
   ```
   string name, address;
   printLabel(name, address);
   ```

   ____

   b.
   ```
   double side1, side2;
   calculateRectangleArea(side1, side2);
   ```

   ____

   c.
   ```
   int day, month, year;
   birthdayInvitation(day, month, year);
   ```

   ____

2. Given the following function headers and variable declarations, write a function call:

   a.
   ```
   string customerName = "Smith";
   double balance = 54000;
   void printBill(string name, double balance)
   ```

   ____

b. `int val1 = 10, val2 = 20;`
   `void findProduct(int num1, int num2)`

____

c. `double balance = 37500, interest = .10;`
   `void newBalance(double bal, double pcnt)`

____

Lab 9-3: Writing Functions that Require Multiple Parameters

In this lab, you complete a partially written C++ program that includes a function requiring multiple parameters (arguments). The program prompts the user for two numeric values. Both values should be passed to functions named `sum()`, `difference()`, and `product()`. The functions compute the sum of the two values, the difference between the two values, and the product of the two values. Each function should perform the appropriate computation and display the results. The source code file provided for this lab includes the variable declarations and the input statements. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `Computation.cpp` using Notepad or the text editor of your choice.
2. Write the C++ statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `Computation.cpp`.
5. Execute the program.

Writing Functions that Return a Value

Thus far in this book, none of the functions you have written (except for the `main()` function) return a value. The header for each of these functions includes the keyword `void`, as in `void nameAndAddress()` indicating that the function does not return a value. However, as a programmer, you will often find that you need to write functions that do return a value. In C++, a function can only return a single value; when you write the code for the function, you must indicate the data type of the value you want returned. This is often referred to as the function's **return type**. The return type can be any of the built-in data types found in C++, as well as a class type, such as `string`. You will learn more about classes in Chapter 10 of this book. For now, you will focus on returning values of the built-in types and `string` objects.

In Chapter 9 of *Programming Logic and Design*, you studied the design for a program that includes a function named `getHoursWorked()`. This function is designed to prompt a user for the number of hours an employee has worked, retrieve the value, and then return that value to the location in the program where the function was called. The C++ code that implements this design is shown in Figure 9-7.

```
1   // GrossPay.cpp - This program computes an employee's gross pay.
2   // Input:  Interactive
3   // Output:  The employee's hours worked and their gross pay
4
5   #include <iostream>
6   #include <string>
7   using namespace std;
8   double getHoursWorked();
9
10  int main()
11  {
12     double hours;
13     const double PAY_RATE = 12.00;
14     double gross;
15
16     hours = getHoursWorked();
17     gross = hours * PAY_RATE;
18
19     cout << "Hours worked: " << hours << endl;
20     cout << "Gross pay is: " << gross << endl;
21
22     return 0;
23  } // End of main() function
24
25  double getHoursWorked()
26  {
27     double workHours;
28
29     cout << "Please enter hours worked: ";
30     cin >> workHours;
31
32     return workHours;
33  } // End of getHoursWorked function
```

Figure 9-7 C++ code for a program that includes the `getHoursWorked()` function

The C++ program shown in Figure 9-7 declares local constants and variables `hours`, `PAY_RATE`, and `gross` on lines 12, 13, and 14 in the `main()` function. The next statement (line 16), shown below, is an assignment statement:

```
hours = getHoursWorked();
```

This assignment statement includes a call to the function named `getHoursWorked()`. As with all assignment statements, the expression on the right side of the assignment operator (=) is evaluated, and then the result is assigned to the variable named on the left side of the assignment operator (=). In this example, the expression on the right is a call to the `getHoursWorked()` function.

When the `getHoursWorked()` function is called, program control is transferred to the function. Notice that the header (line 25) for this function is written as follows:

```
double getHoursWorked()
```

The keyword `double` is used in the header to specify that a value of data type `double` is returned by this function. Also, notice on line 8 in the program, the function declaration agrees with the function header. That is, the function declaration specifies that the function returns a `double`.

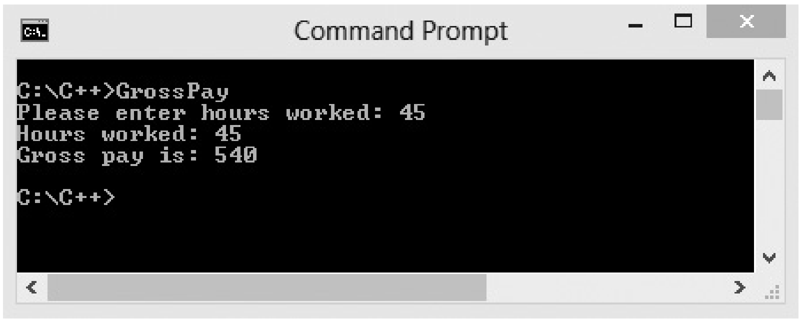
A local, `double` variable named `workHours` is then declared on line 27. On line 29, the user is asked to enter the number of hours worked, and on line 30 the value is retrieved and stored in the local variable named `workHours`. The `return` statement that follows on line 32 returns a copy of the value stored in `workHours` (data type `double`) to the location in the calling function where `getHoursWorked()` is called, which is the right side of the assignment statement on line 16.

The value returned to the right side of the assignment statement is then assigned to the variable named `hours` (data type `double`) in the `main()` function. Next, the gross pay is calculated on line 17, followed by the `cout` statements on lines 19 and 20 that display the value of the local variables, `hours` and `gross`, which contain the number of hours worked and the calculated gross pay.

You can also use a function's return value directly rather than store it in a variable. The two C++ statements that follow make calls to the same `getHoursWorked()` function shown in Figure 9-7, but in these statements, the returned value is used directly in the statement that calculates gross pay and in the statement that prints the returned value:

```
gross = getHoursWorked() * PAY_RATE;
cout << "Hours worked are " << getHoursWorked() << endl;
```

When the input to this program is `45`, the output is shown in Figure 9-8.



```
C:\C++>GrossPay
Please enter hours worked: 45
Hours worked: 45
Gross pay is: 540

C:\C++>
```

Figure 9-8 Output from a program that includes the `getHoursWorked()` function

Exercise 9-4: Writing Functions that Return a Value

In this exercise, you use what you have learned about writing functions that return a value to answer Questions 1–2.

1. Given the following variable declarations and function calls, write the function's header:

   a.
   ```
   double price, percent, newPrice;
   newPrice = calculateNewPrice(price, percent);
   ```
   ____

   b.
   ```
   double area, one_length, two_length;
   area = calcArea(one_length, two_length);
   ```
   ____

   c.
   ```
   string lowerCase, upperCase;
   upperCase = changeCase(lowerCase);
   ```
   ____

2. Given the following function headers and variable declarations, write a function call:

   a.
   ```
   int itemID = 1234;
   string itemName;
   string findItem(int itemNumber)
   ```
   ____

   b.
   ```
   int val, cube;
   int cubed(int num1)
   ```
   ____

   c.
   ```
   int number = 3, exponent = 4, result;
   int power(int num, int exp)
   ```
   ____

Lab 9-4: Writing Functions that Return a Value

In this lab, you complete a partially written C++ program that includes a function that returns a value. The program is a simple calculator that prompts the user for two numbers and an operator (+, -, *, or /). The two numbers and the operator are passed to the function where the appropriate arithmetic operation is performed. The result is then returned to the `main()` function where the arithmetic operation and result are displayed. For example, if the user enters 3, 4, and *, the following is displayed:

```
3 * 4 = 12
```

The source code file provided for this lab includes the necessary variable declarations and input and output statements. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `Calculator.cpp` using Notepad or the text editor of your choice.
2. Write the C++ statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `Calculator.cpp`.
5. Execute the program.

Passing an Array and an Array Element to a Function

As a C++ programmer, there are times when you will want to write a function to perform a task on all of the elements you have stored in an array. For example, in Chapter 9 of *Programming Logic and Design,* you saw a design for a program that used a function to quadruple all of the values stored in an array. This design is translated into C++ code in Figure 9-9.

```
// PassEntireArray.cpp - This program quadruples the values stored in an array.
// Input:  None
// Output:  The original values and the quadrupled values

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

void quadrupleTheValues(int[]);
int main()
{
   const int LENGTH = 4;
   int someNums[] = {10, 12, 22, 35};
   int x;

   cout << "At beginning of main function. . . " << endl;
   x = 0;
   while (x < LENGTH)
   {
      cout << someNums[x] << endl;
      x++;
   }
```

Figure 9-9 C++ code for Pass Entire Array program *(continues)*

(continued)

```
23     quadrupleTheValues(someNums);
24     cout << "At the end of main function. . . " << endl;
25     x = 0;
26     while (x < LENGTH)
27     {
28        cout << someNums[x] << endl;
29        x++;
30     }
31     return 0;
32  } // End of main() function
33
34  void quadrupleTheValues(int vals[])
35  {
36     const int LENGTH = 4;
37     int x;
38
39     x = 0;
40     while(x < LENGTH)
41     {
42        cout << " In quadrupleTheValues() function, value is " << vals[x] << endl;
43        x++;
44     }
45     x = 0;
46     while(x < LENGTH)
47     {
48        vals[x] = vals[x] * 4;
49        x++;
50     }
51     x = 0;
52     while(x < LENGTH)
53     {
54        cout << "  After change, value is " << vals[x] << endl;
55        x++;
56     }
57     return;
58  } // End of quadrupleTheValues function
```

Figure 9-9 C++ code for Pass Entire Array program

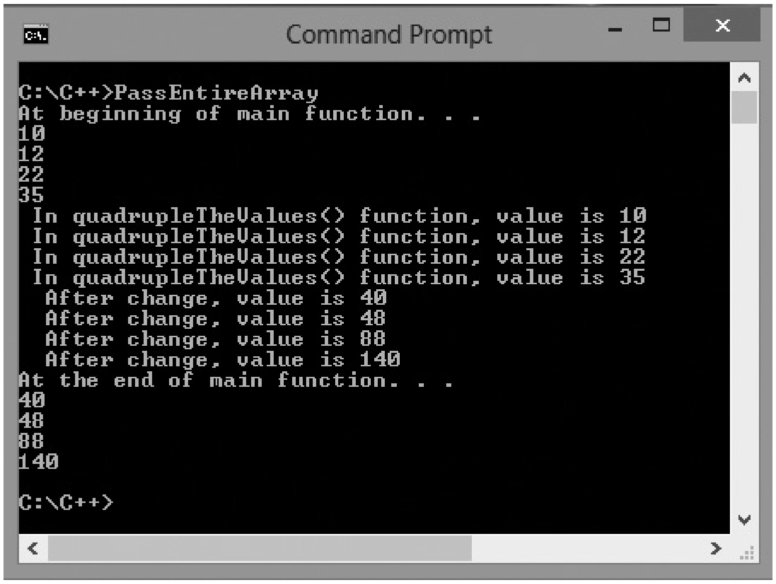
The `main()` function begins on line 10 and proceeds with the declaration and initialization of the constant named `LENGTH` (line 12) and the array of `ints` named `someNums` (line 13), followed by the declaration of the variable named `x` (line 14), which is used as a loop control variable. The first `while` loop in the program on lines 18 through 22 is responsible for printing the values stored in the array at the beginning of the program. Next, on line 23, the function named `quadrupleTheValues()` is called. The array named `someNums` is passed as an argument. Notice that when an entire array is passed to a function, the square brackets and the size are not included. Also note that when you pass an entire array to a function, the beginning memory address of the array is passed by value. This means instead of a copy of the array being passed, the memory address of the array is passed. Although you cannot change the beginning memory address of the array, this does give the function access to that memory location; the function can then change the values stored in the array if necessary.

Program control is then transferred to the `quadrupleTheValues()` function. The header for the function on line 34 includes one parameter, `int vals[]`. The syntax for declaring an array as a formal parameter includes the parameter's data type, followed by a local name for the array, followed by empty square brackets. Note that a size is not included within the square brackets.

On line 9, you see the function declaration for the `quadrupleTheValues()` function. The function declaration includes the return type of the function (`void`), the name of the function (`quadrupleTheValues()`), the data type of the parameter(s) (`int`), and empty square brackets (`[]`) that specify the parameter is an array.

In the `quadrupleTheValues()` function, the first `while` loop on lines 40 through 44 prints the values stored in the array, and the second `while` loop on lines 46 through 50 accesses each element in the array, quadruples the value, and then stores the quadrupled values in the array at their same location. The third `while` loop on lines 52 through 56 prints the changed values now stored in the array. Program control is then returned to the location in the `main()` function where the function was called.

When program control returns to the `main()` function, the next statements to execute (lines 24 through 30) are responsible for printing out the values stored in the array once more. The output from this program is displayed in Figure 9-10.



```
C:\C++>PassEntireArray
At beginning of main function. . .
10
12
22
35
 In quadrupleTheValues() function, value is 10
 In quadrupleTheValues() function, value is 12
 In quadrupleTheValues() function, value is 22
 In quadrupleTheValues() function, value is 35
  After change, value is 40
  After change, value is 48
  After change, value is 88
  After change, value is 140
At the end of main function. . .
40
48
88
140

C:\C++>
```

Figure 9-10 Output from Pass Entire Array program

As shown in Figure 9-10, the array values printed at the beginning of the `main()` function (lines 18 through 22) are the values with which the array was initialized. Next, the `quadrupleTheValues()` function prints the array values (lines 40 through 44) again before they are changed. As shown, the values remain the same as the initialized values. The `quadrupleTheValues()` function then prints the array values again after the values are quadrupled (lines 52 through 56). Finally, after the call to `quadrupleTheValues()`, the `main()` function prints the array values one last time (lines 26 through 30). These are the quadrupled values, indicating that the `quadrupleTheValues()` function has access to the memory location where the array is stored and is able to permanently change the values stored there.

You can also pass a single array element to a function, just as you pass a variable or constant. The following C++ code initializes an array named `someNums`, declares a variable named `newNum`, and passes one element of the array to a function named `tripleTheNumber()`. The array element is passed by value in this example, which means a copy of the value stored in the array is passed to the `tripleTheNumber()` function:

```
int someNums[]= {10, 12, 22, 35};
int newNum;
newNum = tripleTheNumber(someNums[1]);
```

The following C++ code includes the header for the function named `tripleTheNumber()` along with the code that triples the value passed to it:

```
int tripleTheNumber(int num)
{
   int result;
   result = num * 3;
   return result;
}
```

Exercise 9-5: Passing Arrays to Functions

In this exercise, you use what you have learned about passing arrays and array elements to functions to answer Questions 1–3.

1. Given the following function calls, write the function's header:

   a. `int marchBirthdays [] = {17, 24, 21, 14, 28, 9};`
      `printBirthdays(marchBirthdays);`

   ____

   b. `double octoberInvoices [] = {100.00, 200.00, 55.00, 230.00};`
      `double total;`
      `total = monthlyIncome(octoberInvoices);`

   ____

   c. `double balance[] = {34.56, 33.22, 65.77, 89.99};`
      `printBill(balance[1]);`

   ____

2. Given the following function headers and variable declarations, write a function call:

   a.
   ```
   string names[] = {"Jones", "Smith", "Brown", "Perez"};
   double grades[] = {95, 76, 88, 72};
   void midtermGrades(string names[], double grades[])
   ```

   ____

   b.
   ```
   int numbers[] = {1, 4, 6, 8, 3, 7};
   int result;
   int printAverage(int nums[])
   ```

   ____

3. Given the following function header (in which `sal` is one element of an array of `double`s), write a function call that passes the last value in the array named `salaries`:

   a.
   ```
   double salaries[] = {45000, 23000, 35000};
   void bonus(double sal)
   ```

   ____

Lab 9-5: Passing Arrays to Functions

In this lab, you complete a partially written C++ program that reverses the order of five numbers stored in an array. The program should first print the five numbers stored in the array. Next, the program passes the array to a function where the numbers are reversed. The program then prints the reversed numbers in the main program.

The source code file provided for this lab includes the necessary variable declarations. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `Reverse.cpp` using Notepad or the text editor of your choice.
2. Write the C++ statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `Reverse.cpp`.
5. Execute the program.

Passing Arguments by Reference and by Address

As a C++ programmer, you will sometimes want to pass one or more arguments to a function and allow the function to permanently change the value of that argument. In the previous section, you learned when you pass an array to a function, it is actually the beginning memory address of the array that is passed. Because the array's address is passed to a function, the function has access to that memory address and can therefore change the value (or values) that are stored at the memory location.

In C++, there are two techniques you can use to pass the memory address of variables that are not arrays to functions. The two techniques are referred to as **pass by reference** and **pass by address**. You begin with a discussion of pass by reference.

Pass by Reference

Passing an argument to a function by reference allows the called function to access a variable in the calling function. In other words, the called function gains the ability to change the value of a local variable in the calling function. The argument in the called function becomes an **alias** (another name) for the variable in the calling function.

To specify that an argument is passed by reference, you include the ampersand (`&`) operator in the function declaration and in the function header. You do not use the `&` operator in the function call. Figure 9-11 shows a C++ program named `IncreaseSalary.cpp` that includes a function named `increase()`. The `increase()` function calculates an employee's new salary based on a current salary and a percentage that represents an employee's raise.

```
// IncreaseSalary.cpp - This program demonstrates pass by reference.
// Input:  None
// Output:  Beginning salary along with new salary
#include <iostream>
using namespace std;

void increase(double, double, double&);
int main()
{
   double salary = 50000.00;
   double raise = .15;
   double new_salary;

   cout << "Salary is: " << salary << endl;

   // Call the increase function; the variables salary and raise
   // are passed by value, the variable new_salary is passed by
   // reference.
   increase(salary, raise, new_salary);

   cout << "New salary is: " << new_salary << endl;
   return 0;
} // End of main() function

void increase(double amt, double pcnt, double& new_sal)
{
   // Changes the value of new_salary in the main function
   new_sal = amt * (1 + pcnt);
   return;
}
```

Figure 9-11 Pass by reference example

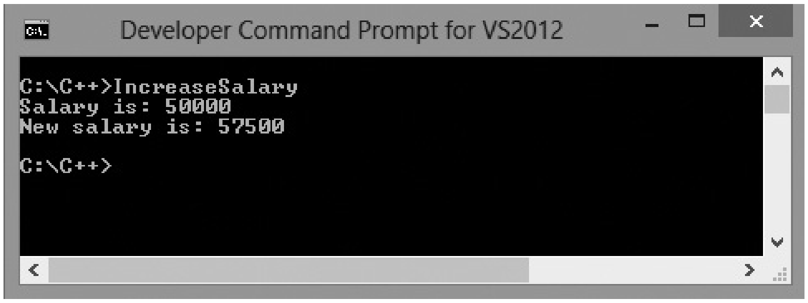
On line 7 in Figure 9-11, you see the function declaration for the function named `increase()`. The function declaration includes the data types for three arguments that are passed to the `increase()` function. The first two arguments are `double`s, and the third argument is a reference to a `double` (`double&`). The function declaration on line 7 has been shaded in Figure 9-11.

In Chapter 3 of this book, you learned that local variables are not available to other functions in the program.

On line 19, you see the function call to the `increase()` function in the `main()` function. The `increase()` function is also shaded so you can easily see that it passes three arguments (`salary`, `raise`, and `new_salary`) that are all local variables declared in the `main()` function.

On line 25, also shaded, you see the header for the `increase()` function that includes three parameters, `double amt`, `double pcnt`, and `double& new_sal`. The first two arguments are passed by value, which means that `amt` contains a copy of the value stored in `salary`, and `pcnt` contains a copy of the value stored in `raise`. The third argument is passed by reference, which means that `new_sal` is actually an alias for the variable `new_salary`. Because `new_sal` is an alias for `new_salary`, when the value of `new_sal` is changed on line 28, the value of `new_salary`, which is declared in the `main()` function, is also changed.

You can compile and execute this program to verify that the `increase()` function changes the value of the variable named `new_salary`. The output from this program is shown in Figure 9-12. The program, named `IncreaseSalary.cpp`, is included with the data files provided with this book.



```
C:\C++>IncreaseSalary
Salary is: 50000
New salary is: 57500

C:\C++>
```

Figure 9-12 Output from pass by reference example

Pass by Address

The second technique you can use to pass the memory address of variables (that are not arrays) to functions is called *pass by address*. Passing an argument by address allows the called function to change the value of an argument.

Before you can understand passing arguments by address, you must learn about pointers. A **pointer** is a variable that stores a memory address as its value. You are accustomed to using a variable name to directly access a value stored at a memory address associated with a variable.

You can also use a pointer variable and the memory address stored in the pointer variable to indirectly access a value stored at a memory address.

The first thing you need to learn in order to use pointers is how to use the address operator. You use the C++ address operator (&) to obtain the memory address associated with a variable. The following C++ code prints the value of a variable named num and also uses the address operator (&) to print the memory address associated with the variable named num:

```
int num = 5;
cout << "Value of num is: " << num << endl;
cout << "Address of num is: " << &num << endl;
```

As you can see in the preceding, highlighted line of code, the address operator (&) is placed before the name of the variable (num) to obtain the address of num.

Next, you must learn to use the asterisk (*) symbol to declare a pointer variable. When you declare a pointer, a specific data type must be explicitly associated with the pointer. The C++ code that follows declares an int variable named age and ptr_age as a pointer to an int:

```
int age = 21;
int* ptr_age = &age;
```

Figure 9-13 shows memory after the preceding code executes. The variable named age is associated with memory location 2000, where the value 21 is stored. The variable named ptr_age is associated with memory location 2004, where the value 2000 (the address of age) is stored. Because ptr_age contains the address of age, you can say that ptr_age is a pointer to an int and that it points to age.

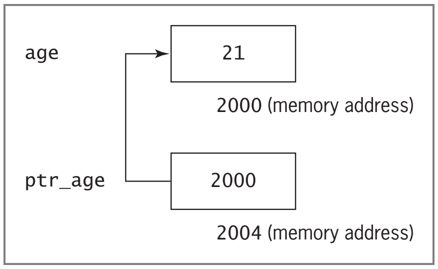


Figure 9-13 Memory contents for age and ptr_age

The (*) symbol also serves as the **indirection operator**, which means that it can be used to indirectly access the value a pointer points to. Because the (*) symbol is used in many ways in C++, the C++ compiler must use context to determine the appropriate meaning. The C++ code that follows shows the use of the (*) symbol to carry out multiplication, to declare a pointer variable, and to function as the indirection operator:

```
int y = 5;
int x;
int* ptr_x; // Pointer variable
x = y * 10; // Multiplication; x is assigned the value 50
ptr_x = &x; // ptr_x is assigned the memory address of x
cout << *ptr_x; // Indirection operator; prints 50
```

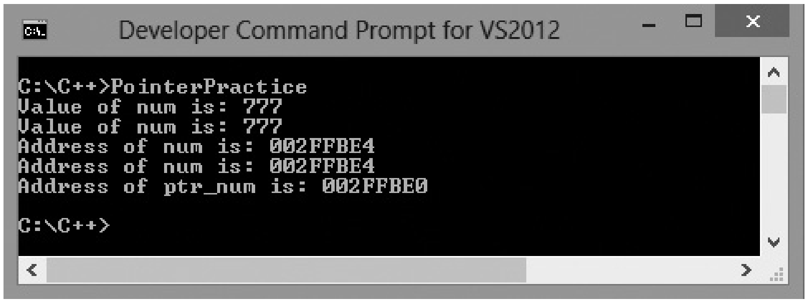
The C++ program shown in Figure 9-14 shows additional examples when the (*) symbol is used to declare a pointer variable and also when it is used as the indirection operator.

```
 1  // PointerPractice.cpp - This program demonstrates pointers.
 2  // Input:  None
 3  // Output:  Values of variables
 4  #include <iostream>
 5  using namespace std;
 6
 7  int main()
 8  {
 9     int num = 777;
10     int* ptr_num;
11
12     ptr_num = &num;  // Assigns address of num to ptr_num
13
14     cout << "Value of num is: " << num << endl;
15
16     cout << "Value of num is: " << *ptr_num << endl;
17
18     cout << "Address of num is: " << &num << endl;
19
20     cout << "Address of num is: " << ptr_num << endl;
21
22     cout << "Address of ptr_num is: " << &ptr_num << endl;
23
24     return 0;
25  } // End of main() function
```

Figure 9-14 C++ program that uses pointer variables and indirection operator

On line 9, the variable `num` is declared as an `int` and initialized with the value `777`, and on line 10, the variable named `ptr_num` is declared as a pointer to an `int`. Line 12 assigns the address of the variable `num` to `ptr_num`, and then on line 14, the `cout` statement prints the value of `num` by using the name of the variable to directly access its value. The `cout` statement on line 16 also prints the value of `num`, but this time the indirection operator is used along with the pointer variable (`*ptr_num`) to indirectly access the value of `num`. Next, the address of `num` is printed on line 18 using the address operator and the name of the variable (`&num`) to obtain the address of the variable. On line 20, the address of `num` is printed again, this time using the value stored in the pointer variable `ptr_num`. The `cout` statement on line 22 prints the address of `ptr_num` using the address operator (`&ptr_num`).

The output of this program is shown is Figure 9-15. Keep in mind that actual memory addresses will vary from one system to another and even from one run of the program to another.



```
C:\C++>PointerPractice
Value of num is: 777
Value of num is: 777
Address of num is: 002FFBE4
Address of num is: 002FFBE4
Address of ptr_num is: 002FFBE0

C:\C++>
```

Figure 9-15 Output of Pointer Practice program

You can compile and execute this program to see the output values on your system. The program file, named `PointerPractice.cpp`, is included with the data files provided with this book.

To pass an argument(s) to a function using pass by address, the memory address of a local variable(s) in the calling function is passed to the called function. You use the (&) operator (address operator) when you call the function. The parameter(s) must be declared as a pointer in the function declaration and in the function's header. You use the (*) symbol to declare the pointer parameter(s). You also must use the * symbol (indirection operator) to dereference the pointer(s) in the body of the function. **Dereferencing** a pointer means that you use the indirection operator to indirectly gain access to the value of a variable.

The C++ program shown in Figure 9-16 illustrates the use of pass by address.

```
// PassByAddress.cpp - This program demonstrates pass by address.
// Input:  None
// Output:  Values of variables
#include <iostream>
using namespace std;

void swap(int*, int*);

int main()
{
   int num1 = 5;
   int num2 = 10;

   cout << "Value of num1 is: " << num1 << endl;

   cout << "Value of num2 is: " << num2 << endl;

   swap(&num1, &num2);

   cout << "Value of num1 is: " << num1 << endl;

   cout << "Value of num2 is: " << num2 << endl;

```

Figure 9-16 C++ program that uses pass by address *(continues)*

(continued)

```
24      return 0;
25   } // End of main() function
26
27   void swap(int* val1, int* val2)
28   {
29      int temp;
30      temp = *val1;
31      *val1 = *val2;
32      *val2 = temp;
33   }
```

Figure 9-16 C++ program that uses pass by address

The function declaration for the `swap()` function is shown on line 7. Notice the `swap()` function does not return a value (`void`); instead, it specifies two parameters, both pointers to `int`s (`int*, int*`).

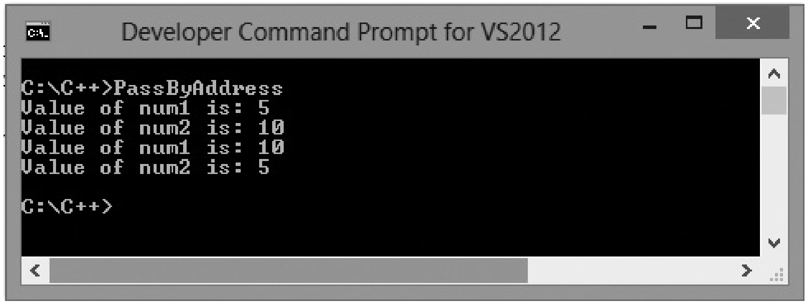
In the `main()` function on lines 11 and 12, two `int` variables, `num1` and `num2`, are declared and initialized to the values 5 and 10, respectively. Both of these variables, `num1` and `num2`, are local to the `main()` function, which means that the `swap()` function does not have direct access to them. On lines 14 and 16, the values of `num1` (5) and `num2` (10) are printed.

Next, on line 18, the `swap()` function is called, passing the address of `num1` (`&num1`) and the address of `num2` (`&num2`) to the function. Because the memory addresses of `num1` and `num2` are passed, the `swap()` function has access to the values stored at those memory addresses and will be able to change the values stored there.

Program control is now passed to the `swap()` function. The header for the `swap()` function on line 27 specifies that the function does not return a value and accepts two parameters, `val1` and `val2`. The two parameters are declared as pointers to `int`s (`int* val1, int* val2`). The parameter `val1` contains the address of `num1`, and the parameter `val2` contains the address of `num2`. On line 29, the `int` variable named `temp` is declared. Because `val1` and `val2` are both pointers, the indirection operator (`*`) is used on lines 30, 31, and 32 to gain access to the values stored in memory that are pointed to by `val1` and `val2`. On line 30, the value pointed to by `val1` (the value of `num1`) is accessed indirectly and then assigned to `temp`. On line 31, the value pointed to by `val2` (the value of `num2`) is accessed indirectly then assigned to what `val1` points to, which is `num1`. This assignment statement changes the value of `num1` in the `main()` function. Line 32 assigns the value stored in `temp` to what `val2` points to, which is `num2`. This assignment changes the value of `num2` in the `main()` function.

When the `swap()` function returns control to the `main()` function, the values of `num1` and `num2` are displayed again on lines 20 and 22. This time, the value of `num1` is 10 and the value of `num2` is 5, illustrating that the local variables `num1` and `num2` have been changed by the `swap()` function.

The output from this program is shown in Figure 9-17. You can compile and execute this program to see the output values on your system. The program, named `PassByAddress.cpp`, is included with the data files provided with this book.



```
C:\C++>PassByAddress
Value of num1 is: 5
Value of num2 is: 10
Value of num1 is: 10
Value of num2 is: 5

C:\C++>
```

Figure 9-17 Output from C++ program that uses pass by address

Exercise 9-6: Pass by Reference and Pass by Address

In this exercise, you use what you have learned about passing arguments by reference and by address to functions to answer Questions 1–2.

1. Given the following variable and function declarations, write the function call and the function's header:

   a. `double price = 22.95, increase = .10;`
      `void changePrice(double&, double);`

      ____

      ____

   b. `double price = 22.95, increase = .10;`
      `void changePrice(double* , double);`

      ____

      ____

   c. `int age = 23;`
      `void changeAge(int&);`

      ____

      ____

   d. `int age = 23;`
      `void changeAge(int*);`

      ____

      ____

2. Given the following function headers and variable declarations, write a function call:

   a.
   ```
   custNames[] = {"Perez", "Smith", "Patel", "Shaw"};
   balances[] = {34.00, 21.00, 45.50, 67.00};
   void cust(string name[], double bal[])
   ```
   ____

   b.
   ```
   int values[] = {1, 77, 89, 321, -2, 34};
   void printSum(int nums[])
   ```
   ____

Lab 9-6: Pass by Reference and Pass by Address

In this lab, you complete a partially written C++ program that includes a function named `multiplyNumbers()` that multiplies two `int` values to find their product. Three `int`s should be passed to the `multiplyNumbers()` function, the two numbers to be multiplied (`num1` and `num2`) should be passed by value, and another `int` (`product`) to hold the product of the two numbers should be passed by reference, enabling the `multiplyNumbers()` function to change its value.

The source code file provided for this lab includes the necessary variable declarations and input and output statements. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `MultiplyTwo.cpp` using Notepad or the text editor of your choice.
2. Write the `multiplyNumbers()` function, the function declaration, and the function call as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `MultiplyTwo.cpp`.
5. Execute the program.
6. Rewrite the `multiplyNumbers()` function to pass the two numbers (`num1` and `num2`) by value and to pass `product` by address.
7. Save this program as `MultiplyTwo2.cpp`.
8. Compile `MultiplyTwo2.cpp`.
9. Execute the program. Both programs (`MultiplyTwo.cpp` and `MultiplyTwo2.cpp`) should generate the same output.

Overloading Functions

You can **overload** functions by giving the same name to more than one function. Overloading functions is useful when you need to perform the same action on different types of inputs. For example, you may want to write multiple versions of an `add()` function—one that can add two integers, another that can add two doubles, another that can add three integers, and another that can add two integers and a double. Overloaded functions have the same name, but they must either have a different number of arguments or the arguments must be of a different data type. C++ figures out which function to call based on the function's name and its arguments, the combination of which is known as the function's **signature**. The signature of an overloaded function consists of the function's name and its argument list; it does not include the function's return type.

Overloading functions allows a C++ programmer to choose meaningful names for functions, and it also permits the use of polymorphic code. **Polymorphic** code is code that acts appropriately depending on the context. (The word *polymorphic* is derived from the Greek words *poly*, meaning "many," and *morph*, meaning "form.") Polymorphic functions in C++ can take many forms. You will learn more about polymorphism in other C++ courses, when you learn more about object-oriented programming. For now, you can use overloading to write functions that perform the same task but with different data types.

In Chapter 9 of *Programming Logic and Design,* you studied the design for an overloaded function named `printBill()`. One version of the function includes a numeric parameter, a second version includes two numeric parameters, a third version includes a numeric parameter and a `string` parameter, and a fourth version includes two numeric parameters and a `string` parameter. All versions of the `printBill()` function have the same name with a different signature; therefore, it is an overloaded function. In Figure 9-18, you see a C++ program that includes the four versions of the `printBill()` function.

```
// Overloaded.cpp - This program illustrates overloaded functions.
// Input:  None
// Output:  Bill printed in various ways

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

void printBill(double);
void printBill(double, double);
void printBill(double, string);
void printBill(double, double, string);

int main()
{
   double bal = 250.00, discountRate = .05;
   string msg = "Due in 10 days.";

```

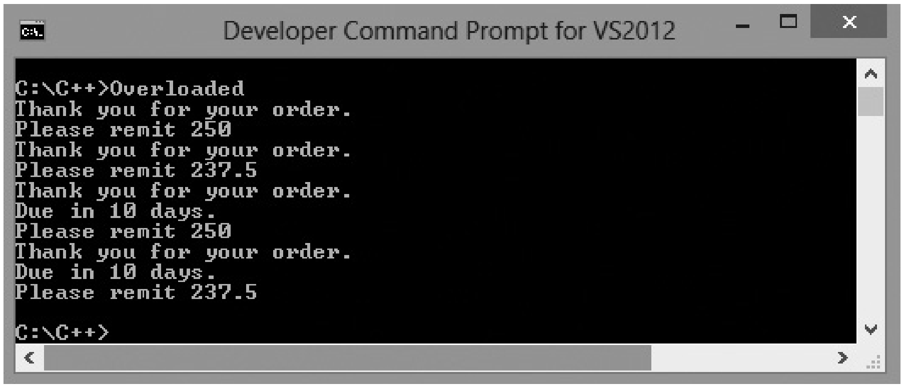
Figure 9-18 Program that uses overloaded `printBill()` functions *(continues)*

(continued)

```
19      printBill(bal);                    // Call version #1
20      printBill(bal, discountRate);      // Call version #2
21      printBill(bal, msg);               // Call version #3
22      printBill(bal, discountRate, msg); // Call version #4
23
24      return 0;
25   } // End of main() function
26
27   // printBill() function #1
28   void printBill(double balance)
29   {
30      cout << "Thank you for your order." << endl;
31      cout << "Please remit " << balance << endl;
32   } // End of printBill #1 function
33
34   // printBill() function #2
35   void printBill(double balance, double discount)
36   {
37      double newBal;
38      newBal = balance - (balance * discount);
39      cout << "Thank you for your order." << endl;
40      cout << "Please remit " << newBal << endl;
41   } // End of printBill #2 function
42
43   // printBill() function #3
44   void printBill(double balance, string message)
45   {
46      cout << "Thank you for your order." << endl;
47      cout << message << endl;
48      cout << "Please remit " << balance << endl;
49   } // End of printBill #3 function
50
51   // printBill() function #4
52   void printBill(double balance, double discount, string message)
53   {
54      double  newBal;
55      newBal = balance - (balance * discount);
56      cout << "Thank you for your order." << endl;
57      cout << message << endl;
58      cout << "Please remit " << newBal << endl;
59   } // End of printBill #4 function
```

Figure 9-18 Program that uses overloaded `printBill()` functions

On line 19, the first call to the `printBill()` function passes one argument, the variable named `bal`, which is declared as a `double`. This causes the runtime system to find and execute the `printBill()` function that is written to accept one `double` as an argument (line 28). The third call to the `printBill()` function (line 21) passes two arguments, a `double` and a `string`. This causes the runtime system to find and execute the `printBill()` function that is written to accept a `double` and a `string` as arguments (line 44). You can compile and execute this program if you would like to verify that a different version of the `printBill()` function is called when a different number of arguments are passed or arguments of different data types are passed. The output from this program is shown in Figure 9-19. The program, named `Overloaded.cpp`, is included with the data files provided with this book.



```
C:\C++>Overloaded
Thank you for your order.
Please remit 250
Thank you for your order.
Please remit 237.5
Thank you for your order.
Due in 10 days.
Please remit 250
Thank you for your order.
Due in 10 days.
Please remit 237.5

C:\C++>
```

Figure 9-19 Output from program that uses overloaded `printBill()` functions

Exercise 9-7: Overloading Functions

In this exercise, you use what you have learned about overloading functions to answer Question 1.

1. In Figure 9-20, which function header would the following function calls match? Use a line number as your answer.

```
1  // Function headers
2  int sum(int num1, int num2)
3  int sum(int num2, int num2, int num3)
4  double sum(double num1, double num2)
5  // Variable declarations
6  double number1 = 1.0, ans1;
7  int number2 = 5, ans2;
```

Figure 9-20 Function headers and variable declarations

a. `ans2 = sum(2, 7, 9);`

____

b. `ans1 = sum(number2, number2);`

____

c. `ans1 = sum(10.0, 7.0);`

____

d. `ans2 = sum(2, 8, number2);`

____

e. `ans2 = sum(3, 5);`

____

Lab 9-7: Overloading Functions

In this lab, you complete a partially written C++ program that computes hotel guest rates at Cornwall's Country Inn. The program is described in Chapter 9, Exercise 11, in *Programming Logic and Design*. In this program, you should include two overloaded functions named `computeRate()`. One version accepts a number of days and calculates the rate at $99.99 per day. The other accepts a number of days and a code for a meal plan. If the code is *A*, three meals per day are included, and the price is $169.00 per day. If the code is *C*, breakfast is included, and the price is $112.00 per day. Each function returns the rate to the calling program where it is displayed. The main program asks the user for the number of days in a stay and whether meals should be included; then, based on the user's response, the program either calls the first function or prompts for a meal plan code and calls the second function. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `Cornwall.cpp` using Notepad or the text editor of your choice.
2. Write the C++ statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `Cornwall.cpp`.
5. Execute the program.

Using C++ Built-in Functions

Throughout this book, you have been told that you do not know enough about the C++ programming language to be able to control the number of places displayed after the decimal point when you print a value of data type `double`. There are actually several ways to control the number of places displayed after a decimal point. In this section, you look at one approach: using a manipulator named `setprecision()`. A **manipulator** is a special C++ function that can control (manipulate) how data is displayed. Throughout this book, you have been using the `endl` manipulator to display a new line character.

In the C++ program that follows, you see how the `setprecision()` manipulator is used to control the number of digits displayed.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
```

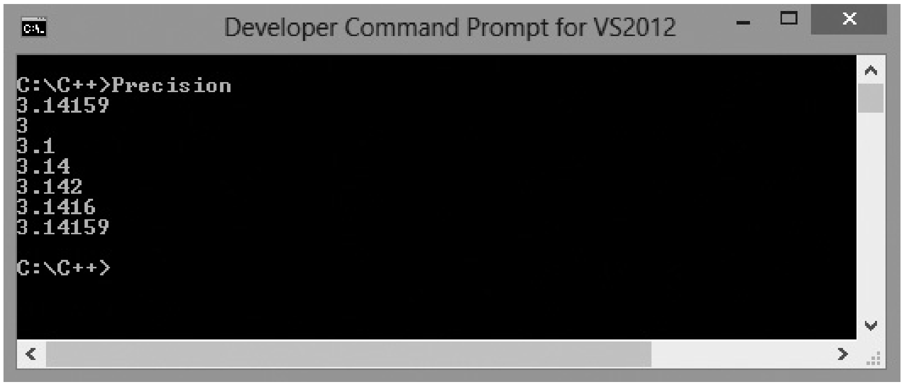
```
{
   double value = 3.14159;
   cout << value << endl;
   cout << setprecision(1) << value << endl;
   cout << setprecision(2) << value << endl;
   cout << setprecision(3) << value << endl;
   cout << setprecision(4) << value << endl;
   cout << setprecision(5) << value << endl;
   cout << setprecision(6) << value << endl;
   return 0;
}
```

In the preceding code sample, `setprecision()` is a manipulator that controls how the floating point value should be formatted. The `setprecision()` manipulator requires one integer argument that specifies the maximum number of digits to be displayed including both those before and those after the decimal point. For example, the C++ code

```
double value = 3.14159;
cout << setprecision(3) << value << endl;
```

produces the following output: `3.14`.

The output from this program is shown in Figure 9-21. You can compile and execute this program to see how the number of digits changes with different integer arguments provided to `setprecision`. The program, named `Precision.cpp`, is included with the data files provided with this book.



```
C:\C++>Precision
3.14159
3
3.1
3.14
3.142
3.1416
3.14159

C:\C++>
```

Figure 9-21 Output from Precision program

Exercise 9-8: Using C++ Built-in Functions

In this exercise, you use a browser, such as Google, to find information about a built-in function that belongs to the C++ function library to answer Questions 1–7.

1. What does the `pow()` function do?

   ____

2. What data type does the pow() function return?

   ____

3. Is the pow() function overloaded? How do you know?

   ____

4. How many arguments does the pow() function require?

   ____

5. What is the data type of the argument(s) that the pow() function requires?

   ____

6. What is the value of the variable named result?

   ```
   result = pow(2,4);
   ```

   ____

7. What is the value of the variable named result?

   ```
   result = pow(10, 2);
   ```

   ____

Lab 9-8: Using C++ Built-in Functions

In this lab, you complete a partially written C++ program that includes built-in functions that convert characters stored in a character array to all uppercase or all lowercase. The program prompts the user to enter nine characters. For each character in the array, you call the built-in functions tolower() and toupper(). Both of these functions return a character with the character changed to uppercase or lowercase. Here is an example:

```
char sample1 = 'A';
char sample2 = 'a';
char result1, result2;
result1 = tolower(sample1);
result2 = toupper(sample2);
```

The source code file provided for this lab includes the necessary variable declarations and the necessary input and output statements. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named ChangeCase.cpp using Notepad or the text editor of your choice.
2. Write the C++ statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file ChangeCase.cpp.
5. Execute the program with the following data:

   ```
   uppercase
   LOWERCASE
   ```