9 Functions

OBJECTIVES

- *Learn how to build structured programs that are divided into functions.*
- *Understand what is meant by the phrase "scope of variables."*
- *Understand how data is passed to functions.*
- *Learn how to use the library functions that are included with the compiler.*

Overview

Up to now you have written programs with only a main function. In Chapter 3 you learned that programs are often divided into more than one function. Programs are divided up so that the functions perform a specific task or *function*. You will learn how to create functions in this chapter as well as how to use the pre-written functions that are included with your compiler.

CHAPTER 9, SECTION 1

How to Build Programs with Functions

Examine the source code in Figure 9-1. The program consists of one function, **`main()`**. You may have difficulty, however, quickly determining what the program accomplishes.

When the program is run, the user is prompted from a menu to choose to view a series of numbers. Depending upon the user's choice, the program displays a series of odd numbers, even numbers, or all integers from 1 to 30.

Let's run the program to see its output.

EXERCISE 9-1 SINGLE-FUNCTION PROGRAM

1. Retrieve the source file *SERIES.CPP*.
2. Compile and run the program to see the program's output.
3. Close the source code file.

The program you just executed could have been better built using more than one function. The diagram in Figure 9-2, known as a ***Visual Table of Contents (VTOC)***, illustrates the point. The lines represent connections between functions. Each function can be accessed by the function above it as long as they are connected by a line.

In this case, the main function "calls" the **`get_choice`** function to ask the user to choose the series to display. Next, the **`handle_choice`** function is called to direct the program flow to one of the three functions under it—one for each series. The source code for Figure 9-2 is presented later in this chapter.

GUIDELINES FOR BUILDING A PROGRAM WITH FUNCTIONS

You may recall from Chapter 2 that good programmers follow the five basic steps shown below when developing programs.

1. Define the problem.
2. Develop an algorithm.
3. Code the program.

```
#include<iostream.h>

main()
{
 int choice;  // variable for user input
 int i;       // variable for loops and output

 do  // loop until a valid choice is entered
  {
   cout << "Which series do you wish to display?\n";
   cout << "1 - Odd numbers from 1 to 30\n";
   cout << "2 - Even numbers from 1 to 30\n";
   cout << "3 - All numbers from 1 to 30\n";
   cin >> choice;  // get choice from user
   if ((choice < 1) || (choice > 3))  // if invalid entry, give message
    {
     cout << "Choice must be 1, 2, or 3\n";
    }
  } while ((choice < 1) || (choice > 3));

  switch (choice)
    {
     case 1:
       for (i = 1; i <= 30; i = i + 2)
       cout << i << ' ';
       cout << endl;
       break;
     case 2:
       for (i = 2; i <= 30; i = i + 2)
       cout << i << ' ';
       cout << endl;
       break;
     case 3:
       for (i = 1; i <= 30; i++)
       cout << i << ' ';
       cout << endl;
       break;
    }
  return 0;
}
```

FIGURE 9-1
This entire program is in the main function.

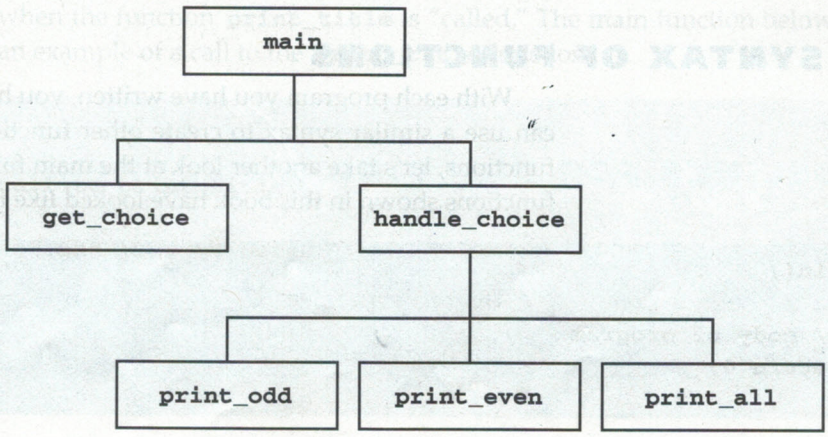

FIGURE 9-2
A diagram that shows the functions of a program is sometimes called a Visual Table of Contents.

4. Test and debug the program.
5. Document and maintain the program.

Using functions helps the programmer develop programs that can be easily coded, debugged, and maintained. Keep the following guidelines in mind when building programs of more than one function.

1. **Organization.** A large program is easier to read and modify if it is logically organized into functions. It is easier to work with a program in parts, rather than one large chunk. A well-organized program, consisting of multiple functions, is easier to read and debug. Once a single function is tested and performs properly, you can set it aside and concentrate on problem areas.
2. **Autonomy**. Programs should be designed so that they consist mainly of stand-alone functions or modules. Each function is autonomous, meaning the function does not depend on data or code outside the function any more than necessary.
3. **Encapsulation**. The term ***encapsulation*** refers to enclosing the details of a function within the function itself, so that those details do not have to be known in order to use the function.
4. **Reusability**. Because functions typically perform a single and well-defined task, they may be reused in the same program or even in other programs.

Functions may be written for any purpose. For example, you could create a function that converts Fahrenheit temperatures to Celsius or a function that gets input from the user. A function can also be a go-between for other parts of the program, as illustrated in the **`handle_choice`** function of Figure 9-2.

On the Net

For more information about program design and coding standards, see http://www.ProgramCPP.com. See topic 9.1.1.

Program Design

*There are two popular methods of designing programs. The first method, called **top-down design,** begins with the functions at the top of the VTOC and works toward the functions at the bottom of the VTOC. In other words, the general organization and flow of the program is decided before the details are coded.*

***Bottom-up design** involves beginning with the bottom of the VTOC and working your way up. Some programmers prefer to work out the details of how the program will perform specific tasks and then bring the details together to create the overall organization and flow.*

Whether you use top-down or bottom-up design, it is important to take an organized approach to writing a multi-function program.

THE SYNTAX OF FUNCTIONS

With each program you have written, you have created a main function. You can use a similar syntax to create other functions. But before we look at other functions, let's take another look at the main function. Up to this point, the main functions shown in this book have looked like the one below.

```
main()
{
 // body of program
 return 0;
}
```

When the program reaches the **return 0;** statement, the value zero is returned to the operating system. This value tells the operating system that the program ended normally. The value returned is a standard integer because we did not specify otherwise. The **int** type is assumed because we did not specify another type. To be more explicit, programs are often written using a main function like the one below.

```
int main()
{
 // body of program
 return 0;
}
```

To prevent a value from being returned, the **void** keyword is used in place of a data type. You may have seen programs with a main function like the one below.

```
void main()
{
 // body of program
}
```

In a **void** function, no value is returned, therefore no **return** statement is included. Newer operating systems are more likely to take advantage of the value returned by the main function. Therefore, you should get into the habit of creating main functions which return a zero when they terminate normally. The **void** main functions are used less frequently now than in the past.

As mentioned earlier, creating other functions in C++ programs is similar to creating the main function. Let's begin by looking at a simple function that prints a message to the screen.

```
void print_title()
 {
  cout << "Tennis Tournament Scheduler Program\n";
  cout << "By Jennifer Baker\n";
 }
```

The name of the function is **print_title**. The **void** keyword indicates that no value is returned. The parentheses after the name let the compiler know that **print_title** is a function. The statements between the braces are executed when the function **print_title** is "called." The main function below includes an example of a call to the **print_title** function.

```
int main()
 {
  print_title(); // call to print_title

  // insert the rest of the program here

  return 0;
 }
```

FUNCTIONS AND PROGRAM FLOW

In the example above, the main function, called the **print_title** function is executed, then the remainder of the program is executed. When a function is called, the computer executes the statements in the function beginning with the first statement. When the end of the function is reached, program execution resumes with the statement that follows the call to the function. Figure 9-3 shows the sequence of execution in a simple three-function example.

The program begins with the first statement of the main function (1), which is a call to the **print_title** function. The flow of logic goes to the **print_title** function which includes two statements (2 and 3) and they are executed next. When the last statement (3) in the **print_title** function is executed, the flow of logic returns to the statement (4) that follows the previous function call in **main()**. The statement in the **print_goodbye** function (5) is then executed. The flow of logic then returns to **main()** where the program ends (6).

FUNCTION PROTOTYPES

There is one more thing you have to do to make your own functions work. At the top of your program, you must tell the compiler that your function exists. You do this by creating a *prototype*. Basically, a prototype defines the function for the compiler. Figure 9-4 shows the functions from Figure 9-3 assembled into a working program.

```
                int main()
Program
Begins  ------>{
            (1) print_title();        // call to print_title

                // insert the rest of the program here

            (4) print_goodbye();

                return 0;
Program
Ends    ------> (6) } // end of main function


               // Function to print program title to screen.
               void print_title()
                {
            (2)  cout << "Tennis Tournament Scheduler Program\n";
            (3)  cout << "By Jennifer Baker\n";
                }

               // Function to print closing message to screen.
               void print_goodbye()
                {
            (5)  cout << "Thank you for using the Tennis Tournament Scheduler.\n";
                }
```

FIGURE 9-3
The numbers next to the statements show the order of execution.

```
#include<iostream.h>

void print_title();     // prototype for print_title function
void print_goodbye();   // prototype for print_goodbye function


int main()
 {
  print_title();       // call to print_title

  // insert the rest of the program here

  print_goodbye();
  return 0;

 } // end of main function


// Function to print program title to screen.
void print_title()
 {
  cout << "Tennis Tournament Scheduler Program\n";
  cout << "By Jennifer Baker\n";
 }

// Function to print closing message to screen.
void print_goodbye()
 {
  cout << "Thank you for using the Tennis Tournament Scheduler.\n";
 }
```

FIGURE 9-4
Functions you create must be prototyped at the beginning of the program.

PITFALLS

Notice that a function's prototype is just like the first line of the function, except for the semicolon. An error will result if you forget to include the semicolon at the end of a prototype.

On the Net

C++ programs begin with a main function. The order that other functions are placed in the program is not important because the function prototype has defined the function for the compiler. Therefore, when a call is made to one of your functions, the compiler knows that the function being called exists, even if it exists below the place where it is called.

Some languages (such as Pascal) end with the main part of the program and place the other functions above the main function (called a procedure in Pascal). In Pascal, you must include a special declaration called a forward declaration if you plan to call a function from code which is above the function. The concept is similar to the function prototypes used in C++.

For more information about the order of functions and the effects of the order of functions, see http://www.ProgramCPP.com. See topic 9.1.2.

EXERCISE 9-2 MAKING A FUNCTION

1. Open *1STFUNCT.CPP*. The program from Figure 9-4 appears.
2. Compile, link, and run the program.
3. When you have run the program successfully, close the source code file.

SECTION 9.1 QUESTIONS

1. Describe an advantage of dividing a program into multiple functions.
2. What is the name of the diagram that shows the functions that make up a program?
3. List two guidelines that you can use to decide what code is a good candidate for being made into a function.
4. Briefly describe top-down and bottom-up design.
5. Write the prototype for the function below.

```
void print_warning()
 {
  cout << "WARNING: Calculations indicate that you will run\n";
  cout << "out of fuel before the next available gas station.\n";
 }
```

6. Write an if structure to be added to the main function below that calls the **print_warning** function above if **fuel_miles < miles_remaining**.

```
int main()
 {
  float miles_remaining, fuel_remaining, mpg, fuel_miles;

  cout << "How many miles are you from the next gas station? ";
  cin >> miles_remaining;
  cout << "How much fuel do you have left (in gallons)? ";
  cin >> fuel_remaining;
  cout << "What is your average miles to the gallon? ";
  cin >> mpg;

  fuel_miles = fuel_remaining * mpg;


  return 0;
 } // end of main function
```

PROBLEM 9.1.1

Write a complete program based on the functions in questions 5 and 6 above. Add another function called **print_okay** that tells the user that he or she will make it to the next gas station. Convert the if structure to an if/else and include the call to the **print_okay** function in the else clause. Save the source code as *GASCHECK.CPP*.

PROBLEM 9.1.2

Write a program that asks the user for an integer. The program should call one of three functions, based on the value entered. If the value is negative, call a function that prints a message indicating that the value is negative. Create similar functions to call when the value is zero and positive. Save the source code as *VALTEST.CPP*.

Data and Functions

When building a program that consists of functions, you must be concerned with how data is made available to the functions. In this section, you will learn about the accessibility of variables in functions and how to get data to and from functions.

SCOPE OF VARIABLES

You have been working mostly with programs that have one function: **main()**. Within **main()**, you declared variables. These variables, however, would be inaccessible outside of **main()**. The "availability" of a variable is known as its *scope*. While this may sound difficult, in C++ the scope of variables is easy to understand.

Note

Local variables are sometimes called **automatic variables** *and global variables are sometimes called* **external variables.**

Variables in C++ can either be local or global. A ***local variable*** is a variable declared within a function and is accessible only within that function. A ***global variable*** is a variable declared before the main function. Global variables are accessible by any function.

Consider the program in Figure 9-5. One variable (**i**) is declared before the main function, making it a global variable. Because **j** and **k** are declared in the main function, they are local to the main function. Therefore, **j** and **k** cannot be used outside of the main function. Within the function named **myfunction**, the variable **l** is declared. It too is local, and accessible only within **myfunction**. After the last statement in **myfunction** is executed, the variable **l** is gone from memory.

If a statement were added to **myfunction** that attempted to access the variable **k**, located in main, an error would result. The variable **k** is accessible only from within the main function. In a similar manner, the variable **l** is inaccessible outside of **myfunction** because it is local to **myfunction**.

EXERCISE 9-3 SCOPE OF VARIABLES

1. Open *SCOPE.CPP*. The program from Figure 9-5 appears on your screen.
2. Compile and run the program as it appears. Study the source code to get clear in your mind where each variable is available.
3. Enter the following statement at the end of **myfunction**.

```
k = i + j;
```

4. Compile the program to see the errors the new statement generates. Your compiler will probably generate an error telling you that the variable **j** and **k** are not defined. The error is generated because **j** and **k** are available only in the main function.
5. Delete the erring statement and close the source code file.

Why have local variables if they are inaccessible to other parts of the program? One reason is that they exist only while the function is executing and

```
#include<iostream.h>

int i = 3;     // global variable

void myfunction();

int main()
 {
  int j,k;     // variables local to the main function
               // j and k are not accessible outside of main
  j = 2;
  k = i + j;
  cout << "j = " << j << " and k = " << k << '\n';
  cout << "i = " << i << " before the call to myfunction.\n";
  myfunction(); // call to myfunction
  cout << "i = " << i << " after the call to myfunction.\n";
  return 0;
 }

void myfunction()
 {
  int l;       // local variable
  l = ++i;     // the variable i is accessible because i is global
               // the variable i is changed globally
  cout << "l = " << l << '\n';
  cout << "The variable l is lost as soon as myfunction exits.\n";
 }
```

FIGURE 9-5
This simple program illustrates the difference between local and global variables.

memory is released when the function terminates. If a variable is needed only within a particular function, you save memory by creating and disposing of the variable within the function.

Using local variables could limit the number of errors that occur in a program. If all variables were global, an error made in a variable and used by various functions could cause multiple errors. However, if you use local variables, any errors are limited to the function in which the variable is declared.

Use local variables whenever possible. Even a large program should have very few global variables. Using local variables keeps a tighter control over your program's data, resulting in fewer bugs and programs which are easier to maintain.

You may be wondering how data can get to other functions if everything is local. As you will learn next, when a function is created, you can choose what data you want to send to the function.

GETTING DATA TO AND FROM FUNCTIONS

You have learned that the parentheses following a function's name lets the compiler know that it is a function. The parentheses can serve another purpose as well. That is, parentheses can be used to pass data to a function and in some cases to return data from a function.

When a function is called, the data in the parentheses (called the ***argument***) is ***passed*** into the receiving function. There are three ways to pass data to functions: passing by value, passing by reference, and passing by address.

PASSING BY VALUE

When you pass a variable to a function by value, a copy of the value in the variable is given to the function for it to use. If the variable is changed within the function, the original copy of the variable in the calling function remains the same. Shown below is an example of a function that passes data to a function using the *passing by value* technique.

```
void print_true_or_false(int True_False)
 {
  if True_False          // If True_False is not equal to zero,
   {                     // display the word TRUE.
    cout << "TRUE\n";
   }
  else                   // If the True_False is equal to zero,
   {                     // display the word FALSE.
    cout << "FALSE\n";
   }
 }
```

A value comes into the function through the parentheses and the copy of the value will be placed in the variable **True_False**. The variable **True_False** is called a *parameter.*

Arguments and Parameters

Many people use the terms argument *and* parameter *interchangeably, but there is a difference. An argument is a value or expression passed to a function through the parentheses when a function is called. A parameter is the variable that receives the value or any other identifier in the parentheses of the function declaration. In other words, an argument is passed to a function, but once in the function, the argument is a parameter.*

When you write a call to a function, you can put any variable or literal in the parentheses to be passed to the function as long as the data types do not conflict. For example, the statements below are all legal calls to the **print_true_or_false** function.

```
print_true_or_false(complete);          // passes a variable
print_true_or_false(1);                 // passes a literal
print_true_or_false(j == 3 && k == 2);  // passes the result of an
                                        // expression
```

The program in Figure 9-6 illustrates how a value passed to the function named **print_value** does not pass back to the main function. Notice that the **print_value** function uses a variable named **j**, even though the main function passes a variable named **i**. The data types must match, but the names are often different.

```
#include<iostream.h>

void print_value(int j);  // function prototype

int main()
 {
  int i = 2;
  cout << "The value before the function is " << i << endl;
  print_value(i);
  cout << "The value after the function exits is " << i << endl;
  return 0;
 }

void print_value(int j)
 {
  cout << "The value passed to the function is " << j << endl;
  j = j * 2; // the value in the variable i is doubled
  cout << "The value at the end of the function is " << j << endl;
 }
```

FIGURE 9-6
This program uses passing by value to pass an integer to the **print_value** function. The argument **i** which appears in the parentheses in the function call is passed to a parameter named **j** in the receiving function.

EXERCISE 9-4 PASSING BY VALUE

1. Enter the program shown in Figure 9-6. Save the source code as *PASSVAL.CPP*.
2. Compile and run the program to see that the value passed to the **print_value** function is not passed back to the main function.
3. Leave the source code file open for the next exercise.

PASSING BY REFERENCE

Functions that pass variables by reference will pass any changes you make to the variables back to the calling function. For example, suppose you need a function that gets input from the user. The function below uses ***passing by reference*** to get two values from the user and pass them back through parentheses.

```
void get_values(float &income, float &expense)
 {
  cout << "Enter this month's income amount: $";
  cin >> income;
  cout << "Enter this month's expense amount: $";
  cin >> expense;
 }
```

To pass a variable by reference, simply precede the variable name with an ampersand (&) in the function definition. But even though it is easy to pass by

reference, you should do so sparingly. You should write functions that pass variables by value whenever possible. The reason is because passing variables by value is safer. When you pass a variable by value, you know it cannot be changed by the function you call. When you pass a variable by reference, a programming error in the function could cause a problem throughout the program.

As a general rule, you should use passing by reference only when data needs to be passed back to the calling function. In the preceding example, the data entered by the user must be passed back to the calling function.

The program you ran in the last exercise passed a variable by value. Let's modify the program to make it pass the variable by reference.

EXERCISE 9-5 PASSING BY REFERENCE

1. Add an ampersand (&) before the identifier **j** in both the prototype and the function declaration. Save the source code as *PASSREF.CPP*.
2. Compile and run the program again to see the difference passing by reference makes.
3. Close the source code file.

PASSING BY ADDRESS

When you pass an array (such as a character array), the syntax looks just like passing by value. However, what C++ passes to the function is the address of the location in memory where the first element of the array resides. So even though it may seem like you are passing by value, you are actually using a technique called ***passing by address.*** Because of the way arrays are passed, any changes made to an array in a function remain when the function is complete.

EXERCISE 9-6 PASSING ARRAYS

1. Open *PASSARRY.CPP*.
2. Study the source code to see how the array is passed.
3. Compile, link, and run the program.
4. Close the source code file.

RETURNING VALUES USING RETURN

As you learned earlier in this chapter, unless a function is declared with the keyword **void**, the function will return a value. In the case of the main function, it returns a value to the operating system. Other functions, however, return a value to the calling function. The value to be returned is specified using the **return** statement.

The function below is an example of a function that returns a value of type **float**. The temperature in Celsius is passed into the function by value and the temperature in Fahrenheit is returned using the **return** statement.

```
float celsius_to_fahrenheit(float celsius)
 {
  float fahr;  // local variable for calculation
  fahr = celsius * (9.0/5.0) + 32.0;
  return(fahr);
 }
```

Any function which is not declared as void should include a **return** statement. The value or expression in the return statement is that which is returned to the calling function. In the **celsius_to_fahrenheit** function, the value stored in **fahr** is returned to the calling function.

The program below shows how you can use this function.

```
#include<iostream.h>
int main()
 {
  float fahrenheit;
  float celsius = 22.5;

  fahrenheit = celsius_to_fahrenheit(celsius);

  cout << celsius << " C = " << fahrenheit << " F\n";
  return 0;
 }
```

The statement **fahrenheit = celsius_to_fahrenheit(celsius);** calls the **celsius_to_fahrenheit** function and passes the value in the variable **celsius** to the function. The function returns the temperature in Fahrenheit degrees and the calling statement assigns the Fahrenheit temperature to the variable **fahrenheit**.

The **celsius_to_fahrenheit** function could be rewritten to include only one statement and return the same result. Any valid expression can appear in the parentheses of the return statement. In this case, the local variable **fahr** is eliminated by performing the calculation in the return statement.

Note

You can use any data type when declaring a function.

```
float celsius_to_fahrenheit(float celsius)
 {
  return(celsius * (9.0/5.0) + 32.0);
 }
```

On the Net

Important Points Regarding Return

1. *The return statement does not require that the value being returned be placed in parentheses. You may, however, want to get into the habit of placing variables and expressions in parentheses to make the code more readable.*
2. *A function can return only one value using return. Use passing by reference to return multiple values from a function.*
3. *When a return statement is encountered, the function will exit and return the value specified, even if other program lines exist below the return.*
4. *A function can have more than one return statement to help simplify an algorithm.*
5. *The calling function is not required to use or even to capture the value returned from a function it calls.*

To learn more about the ways return can be used, and to see examples of the points listed above, see http://www.ProgramCPP.com. See topic 9.2.1.

PITFALLS

When the last line of a function is reached, or when a **`return()`** *statement is executed, the function ends and the program returns to the calling function and begins executing statements from where it left. Do not end functions with a call back to the original function or the function will not terminate properly. Continually calling functions without returning from them will eventually cause the program to crash.*

If you call the main function at the end of a function you wrote, the main function will begin with the first statement, rather than beginning with the statement following the call to your function.

EXERCISE 9-7 USING RETURN

1. Open *CTOF.CPP*. The complete Celsius to Fahrenheit program appears.
2. Compile, link, and run the program. After you have seen it work, close the source code file.

DIVIDING THE SERIES PROGRAM INTO FUNCTIONS

Now that you have practiced creating functions and moving data to and from them, let's take another look at the program from Exercise 9-1. Earlier in this chapter, you studied a VTOC of the program divided into functions. Refer back to the first few pages of this chapter if you need to review the program or VTOC.

EXERCISE 9-8 MULTI-FUNCTION PROGRAM

1. Retrieve the source code file *SERIES2.CPP*. Analyze the source code to see that the program is divided into functions.
2. Compile and run the program to see that it has the same result as the single-function version you ran in Exercise 9-1.
3. Close the source code file.

More About Function Prototypes

A function prototype consists of the function's return type, name and argument list. In this chapter, the function prototypes specified the parameter names in the argument list. However, this is not necessary as long as the type is specified. For example, the prototype for the **`celsius_to_fahrenheit`** *function could be written as:*

```
float celsius_to_fahrenheit(float);
```

The prototype for the **`get_values`** *function could be written as:*

```
void get_values(float &, float &);
```

SECTION 9.2 QUESTIONS

1. What is meant by the scope of a variable?
2. Describe one advantage to using local variables.
3. Where must a variable be declared in order for it to be local?
4. Where must a variable be declared in order for it to be global?
5. Describe what happens when a variable is passed by value?
6. When should you pass variables by reference?
7. What method is used to pass character arrays to functions?

PROBLEM 9.2.1

Modify *SERIES2.CPP* so that the calls to **get_choice** and **handle_choice** are in a do while loop. Add an item to the menu numbered 0 (zero) that exits the program. Have the loop continually redisplay the menu until zero is chosen. *Note:* Make sure you change the do while loop in the **get_choice** function so that zero is a valid input. Save the source code as *SERIES3.CPP*.

PROBLEM 9.2.2

Add a function to the *CTOF.CPP* program that asks the user for the temperature in Celsius and uses passing by reference to return the temperature to the main function. Save the source code as *CTOF2.CPP*.

CHAPTER 9, SECTION 3

Library Functions

C++ compilers include pre-written, ready-to-use functions to make programming easier. The number and type of functions available to you will vary depending on your compiler. The functions that come with your compiler are called ***library functions***. This section shows you how to use some of the more common library functions.

USING LIBRARY FUNCTIONS

Library functions are just like functions you create and may be used in the same way. The difference is that the source code for library functions does not appear in your program. The prototypes for library functions are provided to your program using the **#include** compiler directive.

Let's examine a common C++ library function, **pow()**, which is used to raise a value (x) by the power (y). The **pow()** function prototype is shown below.

```
double pow(double x, double y);
```

The function pow receives two values or expressions of type double and returns the result as a double. Below is an example of a call to the pow function.

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

In order to use the pow function, you must include the **math.h** header file using the compiler directive below. A ***header file*** is a text file that provides the prototypes of a group of library functions. The linker uses the information in the header file to properly link your program with the function you want to use.

```
#include<math.h>
```

EXERCISE 9-9 USING LIBRARY FUNCTIONS

1. Write a program that prompts the user for two values of type double and uses the **pow** function to calculate x^y. Be sure to include the header files for **iostream.h** and **math.h**.
2. Add appropriate comments to your program to describe the parts of your program.
3. Save the program as *POWER.CPP*, compile, run, and test your program.
4. When you have completed the program and verified that it is working properly, close the source code file.

POPULAR MATH FUNCTIONS

Many C++ compilers provide basic math functions, such as the one you used to calculate x^y. Table 9-1 describes some basic math functions, and shows their prototypes and their purpose.

Note

All of the functions in Table 9-1 require that **math.h** *be included in the calling program.*

FUNCTION	PROTOTYPE	DESCRIPTION
abs	int abs(int x);	Returns the absolute value of an integer
labs	long int labs(long int x);	Returns the absolute value of a long integer
fabs	double fabs(double x);	Returns the absolute value of a floating-point number
ceil	double ceil(double x);	Rounds up to a whole number
floor	double floor(double x);	Rounds down to a whole number
hypot	double hypot(double a, double b);	Calculates the hypotenuse (c) of a right triangle where $c^2 = a^2 + b^2$
pow	double pow(double x, double y);	Calculates x to the power of y
pow10	double pow10(int x);	Calculates 10 to the power of x
sqrt	double sqrt(double x);	Calculates the positive square root of x

TABLE 9-1

EXERCISE 9-10 USING sqrt

1. Write a program that uses the **sqrt** function to calculate the circumference of a circle, given its area. Use the formula $2\sqrt{\pi * \text{area}}$.
2. Save the program as *CIRC.CPP* and close the source code when you have completed the exercise.

TRIGONOMETRIC AND LOGARITHMIC FUNCTIONS

Included with many C++ compilers are also trigonometric and logarithmic library functions. These functions, shown in Table 9-2, also require that **math.h** be included in the calling program.

Note

The trigonometric functions work with angles in radians rather than degrees.

FUNCTION	PROTOTYPE	DESCRIPTION
cos	double cos(double x);	Calculates the cosine of x
sin	double sin(double x);	Calculates the sine of x
tan	double tan(double x);	Calculates the tangent of x
acos	double acos(double x);	Calculates the arc cosine of x
asin	double asin(double x);	Calculates the arc sine of x
atan	double atan(double x);	Calculates the arc tangent of x
atan2	double atan2(double y, double x);	Calculates the arc tangent of y/x
cosh	double cosh(double x);	Calculates the hyperbolic cosine of x
sinh	double sinh(double x);	Calculates the hyperbolic sine of x
tanh	double tanh(double x);	Calculates the hyperbolic tangent of x
exp	double exp(double x);	Calculates the exponential function e^x
log	double log(double x);	Calculates the natural logarithm of x
log10	double log10(double x);	Calculates the base 10 logarithm of x

TABLE 9-2

EXERCISE 9-11 TRIG AND LOG FUNCTIONS

1. Write a program that prompts the user for a variable of type double and returns the cosine, sine, tangent, and natural logarithm of the value entered.
2. Save the source code as *TRIGLOG.CPP* and close.

FUNCTIONS FOR WORKING WITH CHARACTERS

C++ compilers also include many functions for analyzing and changing characters. The header file **ctype.h** must be included for a calling program to use the functions listed in Table 9-3. The conditional functions in the table return a non-zero integer if the condition is true and zero if the condition is false.

FUNCTION	PROTOTYPE	DESCRIPTION
isupper	int isupper(int c);	Determines if a character is upper case
islower	int islower(int c);	Determines if a character is lower case
isalpha	int isalpha(int c);	Determines if a character is a letter (a–z, A–Z)
isdigit	int isdigit(int c);	Determines if a character is a digit (0 – 9)
toupper	int toupper(int c);	Convert a character to uppercase
tolower	int tolower(int c);	Convert a character to lowercase

TABLE 9-3

EXERCISE 9-12 CHARACTER FUNCTIONS

1. Write a program that accepts a single character as input and uses the character functions to report back to the user the following information:
   a. Whether the character is a letter, a digit, or some other type of character.
   b. If the character is a letter, tell the user whether the letter is uppercase or lowercase.
2. Save the source code as *CHARFUN.CPP* and close.

SECTION 9.3 QUESTIONS

1. What is the term for functions that come with your compiler?
2. What do you do to provide prototypes for functions that come with your compiler?
3. What does the **pow10** function do?
4. What type of variable is passed to the **sqrt** function?
5. Write a function call that returns the tangent of 1.0.
6. What library function can be used to convert a character to uppercase?

KEY TERMS

argument	parameter
automatic variable	pass
bottom-up design	passing by address
encapsulation	passing by reference
external variable	passing by value
global variable	prototype
header file	scope
library functions	top-down design
local variable	Visual Table of Contents (VTOC)

SUMMARY

- Designing a program that consists of functions results in code that is better organized, reusable, and easier to debug.
- The syntax of functions you create is very similar to that of the main function. The parentheses after the function name tell the compiler that you are defining a function.
- You must create a prototype for your functions to let the compiler know your function exists. Prototypes are placed at the top of the program.

- A local variable is created within a function and is accessible only from within that function. A global variable is declared outside of all functions and is accessible from any function.
- Data can be passed to functions by value, by reference, or by address. When possible, you should pass by value. Passing by value passes a copy of the data, and the original data cannot be changed from within the function. Data passed by reference brings back changes made to it within a function. Arrays are always passed by address, which means the memory address of the array is passed to the function. Any changes to an array made in a function affect the array throughout the program.
- Functions that come with your compiler are called library functions.

PROJECTS

PROJECT 9-1 • SOFTWARE DEVELOPMENT

Write a program source code template that you can use as a starting point for programs you create in the future. Include comments at the top that give your name and provide places for you to fill in the date and description of the program. Set aside a place for **`#include`** directives, prototypes, constants, and global variables. Create an empty main function. Save the source code file as *NEWPROG.CPP* and close the file.

PROJECT 9-2 • MATHEMATICS

Write a program that returns the square root of a number as a multiplier and a square root. For example, given the value 45, the program should output that the square root of 45 is equal to 3 times the square root of 5.

PROJECT 9-3 • POINT-OF-SALE SYSTEM

Use the template you created in Project 9-1 as a starting point to write a program that will function as a point-of-sale system at a rodeo snack bar. The snack bar sells only six different items: a sandwich, chips, pickle, brownie, regular drink, and a large drink. All items are subject to sales tax. Set prices for the products.

The program should repeatedly display the menu below until the sale is totaled. The program should keep a running total of the amount of the sale based on costs that you place in constants for each of the food items. The running total should be displayed somewhere on the screen each time the menu is displayed again.

S - Sandwich

C - Chips

B - Brownie

R - Regular Drink

L - Large Drink

X - Cancel sale and start over

T - Total the sale

If the sale is canceled, clear your running total and display the menu again. When the sale is totaled, calculate the sales tax based on your local tax rate (use 6% if you have no sales tax in your area). Print the final total due on the screen.

You can use your own functions to design a solution to the problem. You are required to use a function to calculate the sales tax. Other use of functions is up to you.

PROJECT 9-4 • TRAFFIC CONTROL

Write a program that simulates the operation of traffic lights at an intersection. Use a *for loop* to create a delay or refer to your compiler's manual for a function such as **`sleep()`** to create a delay. The program should display information about the traffic lights of all four directions. Include a left-turn arrow, and don't forget the yellow lights.

PROJECT 9-5 • PHYSICS

Einstein predicted that the length of an object gets smaller as the object moves more quickly. Use the formula **`length = (length_at_zero * sqrt(1 - ((velocity * velocity)/(C * C))))`**, where C equals the speed of light (3.00×10^8 m/sec), to calculate the new length when given the velocity and length at zero speed. Declare C as a constant. Lengths should be entered in meters.

Overview

In this case study, you will follow the steps of the programming process to develop a program that calculates the growth of money using compound interest. When money is placed in an interest-bearing account, the bank adds the interest earned at a regular interval, often monthly. Each month, you earn interest on a larger amount of money because you are also earning interest on the interest you earned in previous months.

Defining the Problem

We'll begin by defining the problem. As input, the program will ask for several pieces of data.

1. The amount of money placed in the savings account.
2. The interest rate the money will earn in the account.
3. The year that the money is placed in the account.
4. The month that the money is placed in the account.
5. The number of months the money is to remain in the account.

The output will be a table that shows the month and year and the amount of money to which the account has grown by that month.

Developing an Algorithm

The flow of the program is fairly simple. First, it must ask the user for the needed values. Then the program will use a loop as it calculates the new principal amount for each month. The flowchart in Figure II-1 illustrates the flow of logic necessary for the program.

The next step would be to develop a Visual Table of Contents (VTOC) to decide what functions must be written. Functions would need to be written that get the input from the user, that

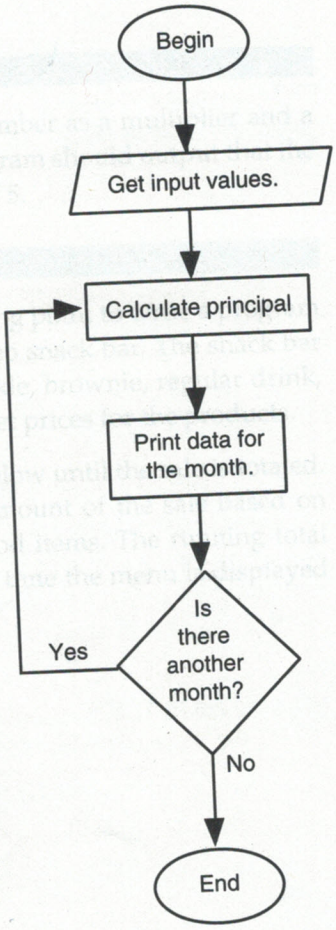

FIGURE II-1
A flowchart helps a programmer visualize the flow of logic in a program.

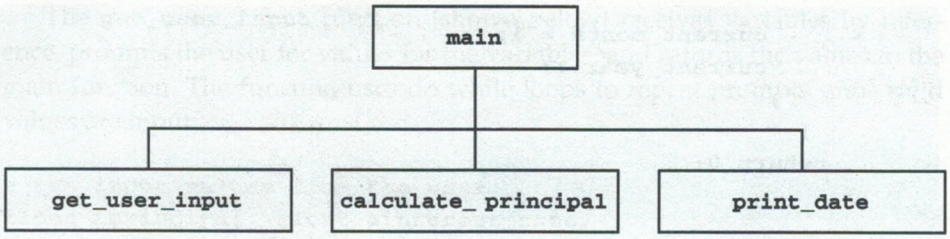

FIGURE II-2
A visual table of contents shows the functions of the program.

calculate the new account balance, and that print a line of the table, as shown in the VTOC in Figure II-2.

Coding the Program

A top-down approach to coding the program is followed. We begin by writing the main function, and then write the functions on which the main function depends. As our flowchart showed, the program must first ask the user to input values. But before we ask the user for those values, there must be variables in which to store the data. The code below shows the main function.

```
// main function
int main()
{
  float principal, interest_rate;
  int first_year, first_month,
      total_months, month_count,
      current_year, current_month;
  char WaitTemp[2];

  get_user_input(principal, interest_rate, first_year,
                 first_month, total_months);

  current_year = first_year;
  current_month = first_month;

  cout.setf(ios::fixed);      // prevent exponential notation
  cout.setf(ios::showpoint); // always show decimal point
  for (month_count = 0; month_count < total_months; month_count++)
   {
    calculate_principal(principal, interest_rate);
    print_date(current_year, current_month);
    cout << "$" << setprecision(2) << principal << '\n';
    if (current_month < 12)
     {                  // If not yet December, increment month
      current_month++;
     }
    else                // else stop output until Enter is pressed,
     {                  // set month back to January, and increment year
      cout << "\nPress Enter to Continue...\n";
      cin.get(WaitTemp,0);
      cin.ignore(80,'\n');
```

```
        current_month = 1;
        current_year++;
      }
    }
    return 0;
  }
```

After the variables are declared, the main function calls **get_user_input** to get the necessary values from the user. Once those values are returned to the main function, **current_year** and **current_month** are initialized to the initial values entered by the user.

Next, two format options are set to ensure that the output does not appear in exponential notation, and that the decimal point always appears. Now we're ready for the loop that prints the output.

We'll use a for loop to print the table because we know the number of times we need to iterate. A loop counter variable named **month_count** is used by the for loop.

The first statement in the loop calls a function named **calculate_principal** that takes the principal and interest rate and returns a new principal amount. The principal will grow with each iteration of the loop.

Next, the current year and current month are passed to a function called **print_date**, which prints the name of the month followed by the year. The principal is then printed on the same line of the screen.

The remainder of the loop is an if structure used to increment the month and, when necessary, the year. Each time the year is incremented, the output pauses to allow the user time to read the output.

EXERCISE II-1 CREATING THE main FUNCTION

1. Enter the following lines into a blank editor screen.

```
// COMPOUND.CPP
// By Jonathan Kleid
// Calculates future value of an amount of money placed in an interest
// bearing account over time.

#include <iostream.h>
#include <iomanip.h>

// function prototypes
void get_user_input(float &principal, float &interest_rate,
                    int &first_year, int &first_month, int &total_months);
void calculate_principal(float &principal, float interest_rate);
void print_date(int year, int month);
```

2. Add the main function at the bottom of the source code file.
3. Save the source code file as *COMPOUND.CPP* and leave it open for the next exercise.

Now that we have analyzed the main function, let's look at the functions required to finish the program.

The **get_user_input** function (shown below) receives variables by reference, prompts the user for values for the variables, and returns the values to the main function. The function uses do while loops to repeat prompts until valid values are input.

```
// Function that gets the input values from the user.
void get_user_input(float &principal, float &interest_rate,
                    int &first_year, int &first_month, int &total_months)
{
  cout << "Enter the starting principal: ";
  cin  >> principal;
  cout << "Enter the current interest rate (Ex. 0.09 for 9%): ";
  cin  >> interest_rate;
  do
   {
     cout << "Enter the first year: ";
     cin  >> first_year;
     if ((first_year < 1900) || (first_year > 2050))
      {
       cout << "Invalid year.\n";
      }
    } while((first_year < 1900) || (first_year > 2050));

   do
    {
     cout << "Enter the first month (1 for Jan., 2 for Feb., etc...): ";
     cin  >> first_month;
     if ((first_month < 1) || (first_month > 12))
      {
       cout << "Invalid month.\n";
      }
    } while((first_month < 1) || (first_month > 12));

   cout << "Enter the total number of months: ";
   cin  >> total_months;
   cin.ignore(80,'\n');
}
```

EXERCISE II-2 ADDING THE USER INPUT FUNCTION

1. Add the **get_user_input** function to the source code on your screen.
2. Leave the source code file open for the next exercise.

The **print_date** function (shown below) is primarily a switch structure that prints the name of the month based on the month number that is passed to the function. The year and month are passed by value because there is no need to return the values to the main function. After the switch structure prints the month, a statement prints the year.

```
// Function that prints the month and year to the screen.
void print_date(int year, int month)
{
  switch(month)
```

```
  {
    case 1:
      cout << "Jan";
      break;
    case 2:
      cout << "Feb";
      break;
    case 3:
      cout << "Mar";
      break;
    case 4:
      cout << "Apr";
      break;
    case 5:
      cout << "May";
      break;
    case 6:
      cout << "Jun";
      break;
    case 7:
      cout << "Jul";
      break;
    case 8:
      cout << "Aug";
      break;
    case 9:
      cout << "Sep";
      break;
    case 10:
      cout << "Oct";
      break;
    case 11:
      cout << "Nov";
      break;
    case 12:
      cout << "Dec";
      break;
  }
  cout << " " << year << ": ";
}
```

The final function required for the program is a simple function (shown below) that calculates the new principal amount. The variable **principal** is passed by reference because the new principal amount must be passed back to the main function. The interest rate is passed by value because there is no need to pass it back to the main function.

```
// Function that adds the interest earned and calculates the
// new principal amount.
void calculate_principal(float &principal, float interest_rate)
{
  principal = principal + (principal * (interest_rate / 12));
}
```

EXERCISE II-3 FINISHING THE CODING

1. Enter the **`print_date`** and **`calculate_principal`** functions.
2. Save the source code file and leave the source code file for the next exercise.

Testing and Debugging

Once the code for a program has been entered, the program must be tested and debugged before it can be used and distributed as a reliable program.

EXERCISE II-4 COMPILING THE PROGRAM

1. Issue the command to compile the program. If errors occur during the compilation, check your source code for syntax errors or other typographical errors.
2. When the program compiles and runs successfully, test it with the following input.

   Starting Principal: 1000.00
   Interest Rate: 0.08
   Year: 1996
   Month: 4 (April)
   Number of months: 12

3. Calculating by hand, we have determined that the input above should produce the output below. Compare the program's output to the values below to see if the program is working correctly. When the program is run, the values do not match the expected output below. Can you determine what is going wrong?

```
Apr 1996: $1000.00
May 1996: $1006.67
Jun 1996: $1013.38
Jul 1996: $1020.13
Aug 1996: $1026.93
Sep 1996: $1033.70
Oct 1996: $1040.67
Nov 1996: $1047.61
Dec 1996: $1054.59
Jan 1997: $1061.63
Feb 1997: $1068.70
Mar 1997: $1075.83
```

4. Your testing should have revealed that the output of the program is paying interest on the principal before a month has passed. The program should print the first month with the initial principal amount and add interest to the months that follow. The problem can be fixed by moving one line of the program to a different location in the program. Do you know what line to move and where to move it?

5. Study the source code to find the line that is out of place. Your instructor can help you locate the necessary fix.
6. After you make the correction, run the program again to see if the output is correct.
7. When the program functions properly, save the source code and close.

Documenting and Maintaining

The final step is to write documentation and maintain the program. Some of the lines in the program already contain comments. The analysis in this case study is similar to the external documentation written for some programs.

The user documentation for a program such as this should explain how the program expects the input to be entered. For example, the user should be shown the proper format for entering each of the required inputs.

EXERCISE II-5 DOCUMENTING THE PROGRAM

1. Write user documentation for the program. Begin with a paragraph describing the purpose of the program.
2. Next, describe the necessary inputs, and explain the formats required for each.
3. If you entered the documentation in a text editor or word processor, print it.

Modifying the Program

As an additional exercise, modify the program to display the interest earned each month, in addition to the new principal. You may also want to change the program so that a yearly total of interest earned is printed.