Writing C++ Functions

Computers never become bored. They perform the same input, output, and computations your program requires—for as long as you want them to do it. You can take advantage of their repetitive natures by looking at your programs in a new way: as a series of small routines that execute whenever you need them, however many times you require.

This chapter approaches its subject a little differently than the previous chapters do. It concentrates on teaching you to write your own *functions*, which are *modules* of code that you execute and control from the `main()` function. So far, the programs in this book have consisted of a single long function called `main()`. As you learn here, the `main()` function's primary purpose is to control the execution of other functions that follow it.

This chapter introduces the following:

- The need for functions
- How to trace functions
- How to write functions
- How to call and return from functions

This chapter stresses the use of *structured programming*, sometimes called *modular programming*. C++ was designed in a way that the programmer can write programs in several modules rather than in one long block. By breaking the program into several smaller routines (*functions*), you can isolate problems, write correct programs faster, and produce programs that are easier to maintain.

Function Basics

When you approach an application that has to be programmed, it is best not to sit down at the keyboard and start typing. Rather, first *think* about the program and what it is supposed to do. One of the best ways to attack a program is to start with the overall goal, then divide this goal into several smaller tasks. You should never lose sight of the overall goal, but think also of how individual pieces can fit together to accomplish such a goal.

When you finally do sit down to begin coding the problem, continue to think in terms of those pieces fitting together. Don't approach a program as if it were one giant problem; rather, continue to write those small pieces individually.

This does not mean you must write separate programs to do everything. You can keep individual pieces of the overall program together—if you know how to write functions. Then you can use the same functions in many different programs.

C++ programs should consist of many small functions.

C++ programs are not like BASIC or FORTRAN programs. C++ was designed to force you to think in a modular, or subroutine-like, functional style. Good C++ programmers write programs that consist of many small functions, even if their programs execute one or more of these functions only once. Those functions work together to produce a program quicker and easier than if the program had to be written from scratch.

TIP: Rather than code one long program, write several smaller routines, called functions. One of those functions must be called `main()`. The `main()` function is always the first to execute. It doesn't have to be first in a program, but it usually is.

Breaking Down Problems

If your program does very much, break it into several functions. Each function should do only *one* primary task. For example, if you were writing a C++ program to retrieve a list of characters from the keyboard, alphabetize them, then print them to the screen, you could—but shouldn't—write all these instructions in one big `main()` function, as the following C++ *skeleton* (program outline) shows:

```
main()
{
   // :
   // C++ code to retrieve a list of characters.
   // :
   // C++ code to alphabetize the characters.
   // :
   // C++ code to print the alphabetized list on-screen.
   // :
   return 0;
}
```

This skeleton is *not* a good way to write this program. Even though you can type this program in only a few lines of code, it is much better to begin breaking every program into distinct tasks so this process becomes a habit to you. You should not use `main()` to do everything—in fact, use `main()` to do very little except call each of the functions that does the actual work.

A better way to organize this program is to write a separate function for each task the program is supposed to do. This doesn't mean that each function has to be only one line long. Rather, it means you make every function a building block that performs only one distinct task in the program.

The following program outline shows you a better way to write the program just described:


```
main()
{
   getletters();    // Calls a function to retrieve the numbers.
   alphabetize();   // Calls a function to alphabetize
                    // letters.
```

```
   printletters(); // Calls a function to print letters
                   // on-screen.
   return 0;       // Returns to the operating system.
}

getletters()
{
   // :
   // C++ code to get a list of characters.
   // :
   return 0;   // Returns to main().
}

alphabetize()
{
   // :
   // C++ code to alphabetize the characters
   // :
   return 0;   // Returns to main().
}

printletters()
{
   // :
   // C++ code to print the alphabetized list on-screen
   // :
   return 0;   // Returns to main().
}
```

The program outline shows you a much better way of writing this program. It takes longer to type, but it's much more organized. The only action the `main()` function takes is to control the other functions by calling them in a certain order. Each separate function executes its instructions, then returns to `main()`, whereupon `main()` calls the next function until no more functions remain. The `main()` function then returns control of the computer to the operating system.

Do not be too concerned about the `0` that follows the `return` statement. C++ functions return values. So far, the functions you've seen have returned zero, and that return value has been ignored.

Chapter 19, "Function Return Values and Prototypes," describes how you can use the return value for programming power.

TIP: A good rule of thumb is that a function should not be more than one screen in length. If it is longer, you are probably doing too much in one function and should therefore break it into two or more functions.

The `main()` function is usually a calling function that controls the remainder of the program.

The first function called `main()` is what you previously used to hold the entire program. From this point, in all but the smallest of programs, `main()` simply controls other functions that do the work.

These listings are not examples of real C++ programs; instead, they are skeletons, or outlines, of programs. From these outlines, it is easier to develop the actual full program. Before going to the keyboard to write a program such as this, know that there are four distinct sections: a primary function-calling `main()` function, a keyboard data-entry function, an alphabetizing function, and a printing function.

Never lose sight of the original programming problem. (Using the approach just described, you never will!) Look again at the `main()` calling routine in the preceding program. Notice that you can glance at `main()` and get a feel for the overall program, without the remaining statements getting in the way. This is a good example of structured, modular programming. A large programming problem is broken into distinct, separate modules called functions, and each function performs one primary job in a few C++ statements.

More Function Basics

Little has been said about naming and writing functions, but you probably understand much of the goals of the previous listing already. C++ functions generally adhere to the following rules:

1. Every function must have a name.
2. Function names are made up and assigned by the programmer (you!) following the same rules that apply to naming

variables: They can contain up to 32 characters, they must begin with a letter, and they can consist of letters, numbers, and the underscore (_) character.

3. All function names have one set of parentheses immediately following them. This helps you (and C++) differentiate them from variables. The parentheses may or may not contain something. So far, all such parentheses in this book have been empty (you learn more about functions in Chapter 18, "Passing Values").
4. The body of each function, starting immediately after the closing parenthesis of the function name, must be enclosed by braces. This means a block containing one or more statements makes up the body of each function.

TIP: Use meaningful function names. `Calc_balance()` is more descriptive than `xy3()`.

Although the outline shown in the previous listing is a good example of structured code, it can be improved by using the underscore character (_) in the function names. Do you see how `get_letters()` and `print_letters()` are much easier to read than are `getletters()` and `printletters()`?

CAUTION: Be sure to use the underscore character (_) and not the hyphen (-) when naming functions and variables. If you use a hyphen, C++ produces misleading error messages.

All programs must have a `main()` function.

The following listing shows you an example of a C++ function. You can already tell quite a bit about this function. You know, for instance, that it isn't a complete program because it has no `main()` function. (All programs must have a `main()` function.) You know also that the function name is `calc_it` because parentheses follow this name. These parentheses happen to have something in them (you learn more about this in Chapter 18). You know also that the body of the function is enclosed in a block of braces. Inside that block is a

smaller block, the body of a `while` loop. Finally, you recognize that the `return` statement is the last line of the function.

```
calc_it(int n)
{
   // Function to print the square of a number.
   int square;

   while (square <= 250)
     { square = n * n;
       cout << "The square of " << n <<
               " is " << square << "\n";
       n++; }     // A block in the function.

   return 0;
}
```

TIP: Not all functions require a `return` statement for their last line, but it is recommended that you always include one because it helps to show your intention to return to the calling function at that point. Later in the book, you learn that the `return` is required in certain instances. For now, develop the habit of including a `return` statement.

Calling and Returning Functions

You have been reading much about "function calling" and "returning control." Although you might already understand these phrases from their context, you can probably learn them better through an illustration of what is meant by a function call.

A function call is like a temporary program detour.

A function call in C++ is like a detour on a highway. Imagine you are traveling along the "road" of the primary function called `main()` and then run into a function-calling statement. You must temporarily leave the `main()` function and execute the function that was called. After that function finishes (its `return` statement is

reached), program control reverts to `main()`. In other words, when you finish a detour, you return to the "main" route and continue the trip. Control continues as `main()` calls other functions.

> **NOTE:** Generally, the primary function that controls function calls and their order is called a *calling function*. Functions controlled by the calling function are called the *called functions*.

A complete C++ program, with functions, will make this concept clear. The following program prints several messages to the screen. Each message printed is determined by the order of the functions. Before worrying too much about what this program does, take a little time to study its structure. Notice that there are three functions defined in the program: `main()`, `next_fun()`, and `third_fun()`. A fourth function is used also, but it is the built-in C++ `printf()` function. The three defined functions appear sequentially. The body of each is enclosed in braces, and each has a `return` statement at its end.

As you will see from the program, there is something new following the `#include` directive. The first line of every function that `main()` calls is listed here and also appears above the actual function. C++ requires these *prototypes*. For now, just ignore them and study the overall format of multiple-function programs. Chapter 19, "Function Return Values and Prototypes, " explains prototypes.

```
// C16FUN1.CPP
// The following program illustrates function calls.
#include <stdio.h>
next_fun();  // Prototypes.
third_fun();

main()  // main() is always the first C++ function executed.
{
   printf("First function called main() \n");
   next_fun();          // Second function is called here.
   third_fun();           // This function is called here.
   printf("main() is completed \n");      // All control
                                          // returns here.
```

```
   return 0;                        // Control is returned to
                                   //the operating system.
}                                // This brace concludes main().

next_fun()                                  // Second function.
                                 // Parentheses always required.
{
   printf("Inside next_fun() \n");     // No variables are
                                     // defined in the program.
   return 0;             // Control is now returned to main().
}

third_fun()                 // Last function in the program.
{
   printf("Inside third_fun() \n");
   return 0;               // Always return from all functions.
}
```

The output of this program follows:

```
First function called main()
Inside next_fun()
Inside third_fun()
main() is completed
```

Figure 16.1 shows a tracing of this program's execution. Notice that `main()` controls which of the other functions is called, as well as the order of the calling. Control *always* returns to the calling function after the called function finishes.

To call a function, simply type its name—including the parentheses—and follow it with a semicolon. Remember that semicolons follow all executable statements in C++, and a function call (sometimes called a *function invocation*) is an executable statement. The execution is the function's code being called. Any function can call any other function. In the previous program, `main()` is the only function that calls other functions.

Now you can tell that the following statement is a function call:

```
print_total();
```

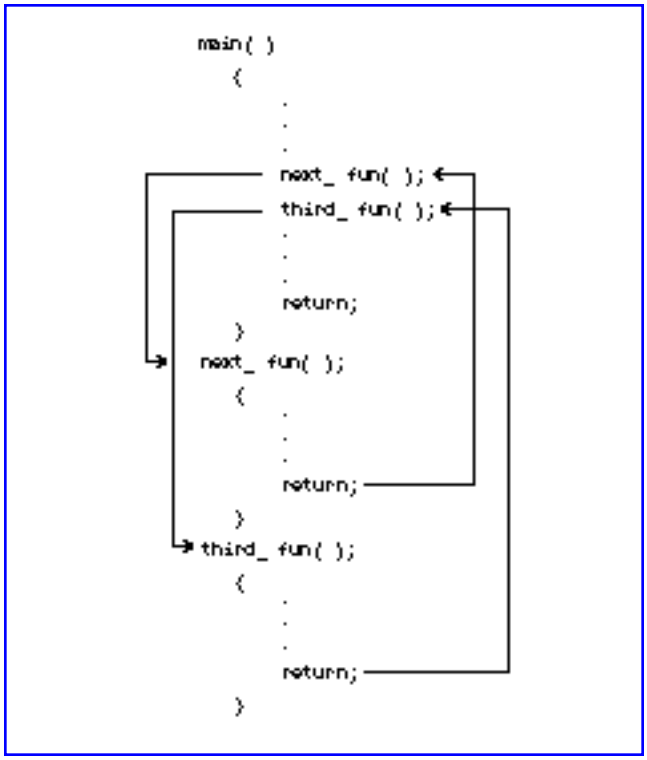


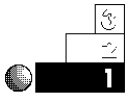
Figure 16.1. Tracing function calls.

Because `print_total` is not a C++ command or built-in function name, it must be a variable or a written function's name. Only function names end with the parentheses, so it must be a function call or the start of a function's code. Of the last two possibilities, it must be a call to a function because it ends with a semicolon. If it didn't have a semicolon, it would have to be the start of a function definition.

When you define a function (by typing the function name and its subsequent code inside braces), you *never* follow the name with a semicolon. Notice in the previous program that `main()`, `next_fun()`, and `third_fun()` have no semicolons when they appear in the body of the program. A semicolon follows their names only in `main()`, where these functions are called.

CAUTION: Never define a function in another function. All function code must be listed sequentially, throughout the program. A function's closing brace *must* appear before another function's code can be listed.

Examples



1. Suppose you are writing a program that does the following. First, it asks users for their departments. Then, if they are in accounting, they receive the accounting department's report. If they are in engineering, they receive the engineering department's report. Finally, if they are in marketing, they receive the marketing department's report.

   The skeleton of such a program follows. The code for `main()` is shown in its entirety, but only a skeleton of the other functions is shown. The `switch` statement is a perfect function-calling statement for such multiple-choice selections.

```
// Skeleton of a departmental report program.
#include <iostream.h>
main()
{
   int choice;

   do
     { cout << "Choose your department from the " <<
               "following list\n";
       cout << "\t1. Accounting \n";
       cout << "\t2. Engineering \n";
       cout << "\t3. Marketing \n";
       cout << "What is your choice? ";
       cin >> choice;
      } while ((choice<1) || (choice>3));  // Ensure 1, 2,
                                           // or 3 is chosen.

   switch choice
   { case(1): { acct_report(); // Call accounting function.
```

```
                  break; }             // Don't fall through.
      case(2): { eng_report(); // Call engineering function.
                  break; }
      case(3): { mtg_report();   // Call marketing function.
                  break; }
   }
   return 0;  // Program returns to the operating
              // system when finished.
}

acct_report()
{
   // :
   // Accounting report code goes here.
   // :
   return 0;
}

eng_report()
{
   // :
   // Engineering report code goes here.
   // :
   return 0;
}

mtg_report()
{
   // :
   // Marketing report code goes here.
   // :
   return 0;
}
```

The bodies of `switch` statements normally contain function calls. You can tell that these `case` statements execute functions. For instance, `acct_report();` (which is the first line of the first `case`) is not a variable name or a C++ command. It is the name of a function defined later in the program. If users enter `1` at the menu, the function called `acct_report()` executes. When it finishes, control returns to the first `case`

body, and its `break` statement causes the `switch` statement to end. The `main()` function returns to DOS (or to your integrated C++ environment if you are using one) when its `return` statement executes.

2. In the previous example, the `main()` routine is not very modular. It displays the menu, but not in a separate function, as it should. Remember that `main()` does very little except control the other functions, which do all the work.

   Here is a rewrite of this sample program, with a fourth function to print the menu to the screen. This is truly a modular example, with each function performing a single task. Again, the last three functions are shown only as skeleton code because the goal here is simply to illustrate function calling and returning.

```
// Second skeleton of a departmental report program.
#include <iostream.h>
main()
{
   int choice;

   do
     {  menu_print();  // Call function to print the menu.
        cin >> choice;
     } while ((choice<1) || (choice>3));   // Ensure 1, 2,
                                           // or 3 is chosen.
   switch choice
   { case(1): { acct_report(); // Call accounting function.
                break; }             // Don't fall through.
     case(2): { eng_report(); // Call engineering function.
                break; }
     case(3): { mtg_report();   // Call marketing function.
                break; }
   }
   return 0;  // Program returns to the operating system
              // when finished.
}

menu_print()
{
```

```
    cout << "Choose your department from the following"
            "list\n";
    cout << "\t1. Accounting \n";
    cout << "\t2. Engineering \n";
    cout << "\t3. Marketing \n";
    cout << "What is your choice? ";
    return 0;   // Return to main().
}

acct_report()
{
    // :
    // Accounting report code goes here.
    // :
    return 0;
}

eng_report()
{
    // :
    // Engineering report code goes here.
    // :
    return 0;
}

mtg_report()
{
    // :
    // Marketing report code goes here.
    // :
    return 0;
}
```

The menu-printing function doesn't have to follow `main()`. Because it's the first function called, however, it seems best to define it there.

3. Readability is the key, so programs broken into separate functions result in better written code. You can write and test each function, one at a time. After you write a general outline of the program, you can list a bunch of function calls in `main()`, and define their skeletons after `main()`.

The body of each function initially should consist of a single `return` statement, so the program compiles in its skeleton format. As you complete each function, you can compile and test the program. This enables you to develop more accurate programs faster. The separate functions enable others (who might later modify your program) to find the particular function easily and without affecting the rest of the program.

Another useful habit, popular with many C++ programmers, is to separate functions from each other with a comment consisting of a line of asterisks (*) or dashes (-). This makes it easy, especially in longer programs, to see where a function begins and ends. What follows is another listing of the previous program, but now with its four functions more clearly separated by this type of comment line.

```
// Third skeleton of a departmental report program.
#include <iostream.h>
main()
{
   int choice;

   do
     {  menu_print();  // Call function to print the menu.
        cin >> choice;
     } while ((choice<1) || (choice>3));  // Ensure 1, 2,
                                          // or 3 is chosen.
   switch choice
   { case(1): { acct_report(); // Call accounting function.
                break; }             // Don't fall through.
     case(2): { eng_report(); // Call engineering function.
                break; }
     case(3): { mtg_report();   // Call marketing function.
                break; }
   }
   return 0;  // Program returns to the operating system
              // when finished.
}

//*******************************************************
menu_print()
```

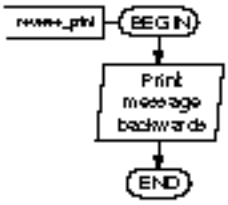
```
{
   cout << "Choose your department from the following"
           "list\n";
   cout << "\t1. Accounting \n";
   cout << "\t2. Engineering \n";
   cout << "\t3. Marketing \n";
   cout << "What is your choice? ";
   return 0;   // Return to main().
}

//*********************************************************
acct_report()
{
   // :
   // Accounting report code goes here.
   // :
   return 0;
}

//*********************************************************
eng_report()
{
   // :
   // Engineering report code goes here.
   // :
   return 0;
}

//*********************************************************
mtg_report()
{
   // :
   // Marketing report code goes here.
   // :
   return 0;
}
```

Due to space limitations, not all program listings in this book separate the functions in this manner. You might find, however, that your listings are easier to follow if you put these separating comments between your functions. The application in Appendix F, "The Mailing List Application,"

for example, uses these types of comments to separate its functions.

4. You can execute a function more than once simply by calling it from more than one place in a program. If you put a function call in the body of a loop, the function executes repeatedly until the loop finishes.

   The following program prints the message `C++ is Fun!` several times on-screen—forward and backward—using functions. Notice that `main()` does not make every function call. The second function, `name_print()`, calls the function named `reverse_print()`. Trace the execution of this program's `cout`s.




```
// Filename: C16FUN2.CPP
// Prints C++ is Fun! several times on-screen.
#include <iostream.h>
name_print();
reverse_print();
one_per_line();

main()
{
   int ctr;  // To control loops

   for (ctr=1; ctr<=5; ctr++)
     { name_print(); }          // Calls function five times.

   one_per_line();            // Calls the program's last
                              // function once.
   return 0;
}

//*********************************************************
name_print()
{
   // Prints C++ is Fun! across a line, separated by tabs.
   cout << "C++ is Fun!\tC++ is Fun!\tC++ is Fun!
            \tC++ is Fun!\n";
   cout << "C++  i s  F u n !\tC++  i s  F u n ! " <<
           "\tC++  i s  F u n !\n";
```

```
   reverse_print();      // Call next function from here.
   return 0;                         // Returns to main().
}

//**********************************************************
reverse_print()
{
   // Prints several C++ is Fun! messages,
   //    in reverse, separated by tabs.
   cout << "!nuF si ++C\t!nuF si ++C\t!nuF si ++C\t\n";

   return 0;                   // Returns to name_print().
}

//**********************************************************
one_per_line()
{
   // Prints C++ is Fun! down the screen.
   cout << "C++\n \ni\ns\n \nF\nu\nn\n!\n";
   return 0;   // Returns to main()
}
```

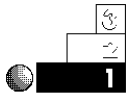
Here is the output from this program:

```
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++ is Fun!     C++ is Fun!     C++ is Fun!     C++ is Fun!
C++   i s   F u n !        C++   i s   F u n !   C++   i s   F u n !
!nuF si ++C     !nuF si ++C     !nuF si ++C
C++
```

```
i
s
F
u
n
!
```

Review Questions

The answers to the review questions are in Appendix B.



1. True or false: A function should always include a `return` statement as its last command.
2. What is the name of the first function executed in a C++ program?

3. Which is better: one long function or several smaller functions? Why?
4. How do function names differ from variable names?
5. How can you use comments to help visually separate functions?

6. What is wrong with the following program section?

```
calc_it()
{
   cout << "Getting ready to calculate the square of 25 \n";

   sq_25()
   {
      cout << "The square of 25 is " << (25*25);
      return 0;
   }

   cout << "That is a big number! \n";
   return 0;
}
```

7. Is the following a variable name, a function call, a function definition, or an expression?

```
scan_names();
```

8. True or false: The following line in a C++ program is a function call.

```
cout << "C++ is Fun! \n";
```

Summary

You have now been exposed to truly structured programs. Instead of typing a long program, you can break it into separate functions. This method isolates your routines so surrounding code doesn't clutter your program and add confusion.

Functions introduce just a little more complexity, involving the way variable values are recognized by the program's functions. The next chapter (Chapter 17, "Variable Scope") shows you how variables are handled between functions, and helps strengthen your structured programming skills.