Chapter 5 Functions: The Basics

5-1. What Is a Function?

This chapter approaches the question "What is a function?" in two ways. First, it tells you what functions are, and then it shows you how they're used.

5-1-1. A Function Defined

First the definition: A *function* is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program. Now let's look at the parts of this definition:

- *A function is named*. Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as *calling* the function. A function can be called from within another function.
- *A function is independent*. A function can perform its task without interference from or interfering with other parts of the program.
- *A function performs a specific task*. This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root.
- *A function can return a value to the calling program*. When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program.

That's all there is to the "telling" part. Keep the previous definition in mind as you look at the next section.

5-1-2. A Function Illustrated

Listing 5.1 contains a user-defined function.

Listing 5.1. A program that uses a function to calculate the cube of a number.

```
1:   /* Demonstrates a simple function */
2:   #include <stdio.h>
3:
4:   long cube(long x);
5:
6:   long input, answer;
7:
8:   main()
9:   {
10:     printf("Enter an integer value: ");
11:     scanf("%d", &input);
12:     answer = cube(input);
13:     /* Note: %ld is the conversion specifier for */
14:     /* a long integer */
15:     printf("\nThe cube of %ld is %ld.\n", input, answer);
16:
17:     return 0;
18:  }
19:
20:  /* Function: cube() - Calculates the cubed value of a variable */
```

```
21:  long cube(long x)
22:  {
23:     long x_cubed;
24:
25:     x_cubed = x * x * x;
26:     return x_cubed;
27:  }
Enter an integer value: 100
The cube of 100 is 1000000.
Enter an integer value: 9
The cube of 9 is 729.
Enter an integer value: 3
The cube of 3 is 27.
```

NOTE: The following analysis focuses on the components of the program that relate directly to the function rather than explaining the entire program.

ANALYSIS: Line 4 contains the *function prototype,* a model for a function that will appear later in the program. A function's prototype contains the name of the function, a list of variables that must be passed to it, and the type of variable it returns, if any. Looking at line 4, you can tell that the function is named cube, that it requires a variable of the type long, and that it will return a value of type long. The variables to be passed to the function are called *arguments,* and they are enclosed in parentheses following the function's name. In this example, the function's argument is long x. The keyword before the name of the function indicates the type of variable the function returns. In this case, a type long variable is returned.

Line 12 calls the function cube and passes the variable input to it as the function's argument. The function's return value is assigned to the variable answer. Notice that both input and answer are declared on line 6 as long variables, in keeping with the function prototype on line 4.

The function itself is called the *function definition*. In this case, it's called cube and is contained in lines 21 through 27. Like the prototype, the function definition has several parts. The function starts out with a function header on line 21. The *function header* is at the start of a function, and it gives the function's name (in this case, the name is cube). The header also gives the function's return type and describes its arguments. Note that the function header is identical to the function prototype (minus the semicolon).

The body of the function, lines 22 through 27, is enclosed in braces. The body contains statements, such as on line 25, that are executed whenever the function is called. Line 23 is a variable declaration that looks like the declarations you have seen before, with one difference: it's local. *Local* variables are declared within a function body. (Local declarations are discussed further on Day 12, "Understanding Variable Scope.") Finally, the function concludes with a return statement on line 26, which signals the end of the function. A return statement also passes a value back to the calling program. In this case, the value of the variable x_cubed is returned.

If you compare the structure of the cube() function with that of the main() function, you'll see that they are the same. main() is also a function. Other functions that you already have used are printf() and scanf(). Although printf() and scanf() are library functions (as opposed to user-defined functions), they are functions that can take arguments and return values just like the functions you create.

5-2. How a Function Works

A C program doesn't execute the statements in a function until the function is called by another part of the program. When a function is called, the program can send the function information in the form of one or more arguments. An *argument* is program data needed by the function to perform its task. The statements in the function then execute,

performing whatever task each was designed to do. When the function's statements have finished, execution passes back to the same location in the program that called the function. Functions can send information back to the program in the form of a return value.

Figure 5.1 shows a program with three functions, each of which is called once. Each time a function is called, execution passes to that function. When the function is finished, execution passes back to the place from which the function was called. A function can be called as many times as needed, and functions can be called in any order.

You now know what a function is and the importance of functions. Lessons on how to create and use your own functions follow.

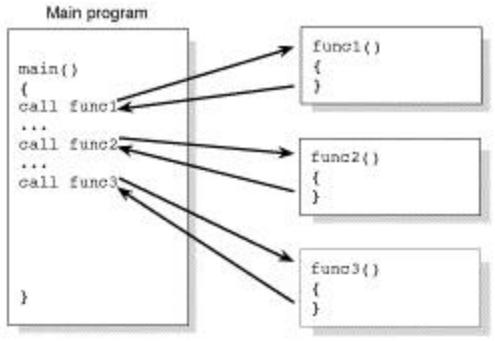

Figure 5-1. When a program calls a function, execution passes to the function and then back to the calling program.

Functions

Function Prototype

```
return_type function_name( arg-type name-1,...,arg-type name-n);
```

Function Definition

```
return_type function_name( arg-type name-1,...,arg-type name-n)
{
    /* statements; */
}
```

A *function prototype* provides the compiler with a description of a function that will be defined at a later point in the program. The prototype includes a return type indicating the type of variable that the function will return. It also includes the function name, which should describe what the function does. The prototype also contains the variable types of the arguments (arg type) that will be passed to the function. Optionally, it can contain the names of the variables that will be passed. A prototype should always end with a semicolon.

A *function definition* is the actual function. The definition contains the code that will be executed. The first line of a function definition, called the *function header,* should be identical to the function prototype, with the exception of the semicolon. A function header shouldn't end with a semicolon. In addition, although the argument variable names were optional in the prototype, they must be included in the function header. Following the header is the function body, containing the statements that the function will perform. The function body should start with an opening bracket and end with a closing bracket. If the function return type is anything other than void, a return statement should be included, returning a value matching the return type.

Function Prototype Examples

```
double squared( double number );
void print_report( int report_number );
int get_menu_choice( void );
```

Function Definition Examples

```
double squared( double number )          /* function header */
{                                        /* opening bracket */
```

```
    return(  number * number );          /* function body   */
}                                        /* closing bracket */
void print_report( int report_number )
{
    if( report_number == 1 )
        puts( "Printing Report 1" );
    else
        puts( "Not printing Report 1" );
}
```

5-3. Functions and Structured Programming

By using functions in your C programs, you can practice *structured programming,* in which individual program tasks are performed by independent sections of program code. "Independent sections of program code" sounds just like part of the definition of functions given earlier, doesn't it? Functions and structured programming are closely related.

5-3-1. The Advantages of Structured Programming

Why is structured programming so great? There are two important reasons:

- It's easier to write a structured program, because complex programming problems are broken into a number of smaller, simpler tasks. Each task is performed by a function in which code and variables are isolated from the rest of the program. You can progress more quickly by dealing with these relatively simple tasks one at a time.
- It's easier to debug a structured program. If your program has a *bug* (something that causes it to work improperly), a structured design makes it easy to isolate the problem to a specific section of code (a specific function).

A related advantage of structured programming is the time you can save. If you write a function to perform a certain task in one program, you can quickly and easily use it in another program that needs to execute the same task. Even if the new program needs to accomplish a slightly different task, you'll often find that modifying a function you created earlier is easier than writing a new one from scratch. Consider how much you've used the two functions printf() and scanf() even though you probably haven't seen the code they contain. If your functions have been created to perform a single task, using them in other programs is much easier.

5-3-2. Planning a Structured Program

If you're going to write a structured program, you need to do some planning first. This planning should take place before you write a single line of code, and it usually can be done with nothing more than pencil and paper. Your plan should be a list of the specific tasks your program performs. Begin with a global idea of the program's function. If you were planning a program to manage your name and address list, what would you want the program to do? Here are some obvious things:

- Enter new names and addresses.
- Modify existing entries.
- Sort entries by last name.
- Print mailing labels.

With this list, you've divided the program into four main tasks, each of which can be assigned to a function. Now you can go a step further, dividing these tasks into subtasks. For example, the "Enter new names and addresses" task can be subdivided into these subtasks:

- Read the existing address list from disk.
- Prompt the user for one or more new entries.
- Add the new data to the list.

- Save the updated list to disk.

Likewise, the "Modify existing entries" task can be subdivided as follows:

- Read the existing address list from disk.
- Modify one or more entries.
- Save the updated list to disk.

You might have noticed that these two lists have two subtasks in common--the ones dealing with reading from and saving to disk. You can write one function to "Read the existing address list from disk," and that function can be called by both the "Enter new names and addresses" function and the "Modify existing entries" function. The same is true for "Save the updated list to disk."

Already you should see at least one advantage of structured programming. By carefully dividing the program into tasks, you can identify parts of the program that share common tasks. You can write "double-duty" disk access functions, saving yourself time and making your program smaller and more efficient.

This method of programming results in a *hierarchical,* or layered, program structure. Figure 5.2 illustrates hierarchical programming for the address list program.

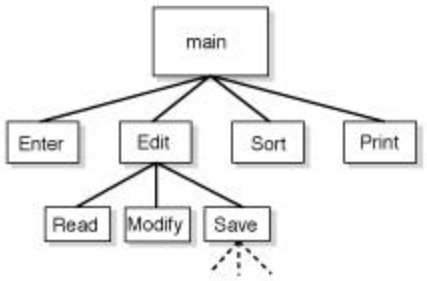

Figure 5-2. A structured program is organized hierarchically.

When you follow this planned approach, you quickly make a list of discrete tasks that your program needs to perform. Then you can tackle the tasks one at a time, giving all your attention to one relatively simple task. When that function is written and working properly, you can move on to the next task. Before you know it, your program starts to take shape.

5-3-3. The Top-Down Approach

By using structured programming, C programmers take the *top-down approach*. You saw this illustrated in Figure 5.2, where the program's structure resembles an inverted tree. Many times, most of the real work of the program is performed by the functions at the tips of the "branches." The functions closer to the "trunk" primarily direct program execution among these functions.

As a result, many C programs have a small amount of code in the main body of the program--that is, in main(). The bulk of the program's code is found in functions. In main(), all you might find are a few dozen lines of code that direct program execution among the functions. Often, a menu is presented to the person using the program. Program execution is branched according to the user's choices. Each branch of the menu uses a different function.

This is a good approach to program design. Day 13, "Advanced Program Control," shows how you can use the switch statement to create a versatile menu-driven system.

Now that you know what functions are and why they're so important, the time has come for you to learn how to write your own.

DO plan before starting to code. By determining your program's structure ahead of time, you can save time writing the code and debugging it.

DON'T try to do everything in one function. A single function should perform a single task, such as reading information from a file.

5-4. Writing a Function

The first step in writing a function is knowing what you want the function to do. Once you know that, the actual mechanics of writing the function aren't particularly difficult.

5-4-1. The Function Header

The first line of every function is the function header, which has three components, each serving a specific function. They are shown in Figure 5.3 and explained in the following sections.

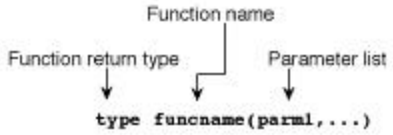

Figure 5-3. The three components of a function header.

The Function Return Type

The function return type specifies the data type that the function returns to the calling program. The return type can be any of C's data types: char, int, long, float, or double. You can also define a function that doesn't return a value by using a return type of void. Here are some examples:

```
int func1(...)          /* Returns a type int.   */
float func2(...)        /* Returns a type float. */
void func3(...)         /* Returns nothing.      */
```

The Function Name

You can name a function anything you like, as long as you follow the rules for C variable names (given in Day 3, "Storing Data: Variables and Constants"). A function name must be unique (not assigned to any other function or variable). It's a good idea to assign a name that reflects what the function does.

The Parameter List

Many functions use *arguments*, which are values passed to the function when it's called. A function needs to know what kinds of arguments to expect--the data type of each argument. You can pass a function any of C's data types. Argument type information is provided in the function header by the parameter list.

For each argument that is passed to the function, the parameter list must contain one entry. This entry specifies the data type and the name of the parameter. For example, here's the header from the function in Listing 5.1:

```
long cube(long x)
```

The parameter list consists of long x, specifying that this function takes one type long argument, represented by the parameter x. If there is more than one parameter, each must be separated by a comma. The function header

```
void func1(int x, float y, char z)
```

specifies a function with three arguments: a type int named x, a type float named y, and a type char named z. Some functions take no arguments, in which case the parameter list should consist of void, like this:

```
void func2(void)
```

NOTE: You do not place a semicolon at the end of a function header. If you mistakenly include one, the compiler will generate an error message.

Sometimes confusion arises about the distinction between a parameter and an argument. A *parameter* is an entry in a function header; it serves as a "placeholder" for an argument. A function's parameters are fixed; they do not change during program execution.

An *argument* is an actual value passed to the function by the calling program. Each time a function is called, it can be passed different arguments. In C, a function must be passed the same number and type of arguments each time it's called, but the argument values can be different. In the function, the argument is accessed by using the corresponding parameter name.

An example will make this clearer. Listing 5.2 presents a very simple program with one function that is called twice.

Listing 5.2. The difference between arguments and parameters.

```
1:   /* Illustrates the difference between arguments and parameters.
*/
2:
3:   #include <stdio.h>
4:
5:   float x = 3.5, y = 65.11, z;
6:
7:   float half_of(float k);
8:
9:   main()
10:  {
11:      /* In this call, x is the argument to half_of(). */
12:      z = half_of(x);
13:      printf("The value of z = %f\n", z);
14:
15:      /* In this call, y is the argument to half_of(). */
16:      z = half_of(y);
17:      printf("The value of z = %f\n", z);
18:
19:      return 0;
20:  }
21:
22:  float half_of(float k)
23:  {
24:      /* k is the parameter. Each time half_of() is called, */
25:      /* k has the value that was passed as an argument. */
26:
27:      return (k/2);
28:  }
The value of z = 1.750000
The value of z = 32.555000
```

Figure 5.4 shows the relationship between arguments and parameters.

ANALYSIS: Looking at Listing 5.2, you can see that the half_of() function prototype is declared on line 7. Lines 12 and 16 call half_of(), and lines 22 through 28 contain the actual function. Lines 12 and 16 each send a different argument to half_of(). Line 12 sends x, which contains a value of 3.5, and line 16 sends y, which contains a value of 65.11. When the program runs, it prints the correct number for each. The values in x and y are passed into the

argument k of half_of(). This is like copying the values from x to k, and then from y to k. half_of() then returns this value after dividing it by 2 (line 27).

DO use a function name that describes the purpose of the function.

DON'T pass values to a function that it doesn't need.

DON'T try to pass fewer (or more) arguments to a function than there are parameters. In C programs, the number of arguments passed must match the number of parameters.

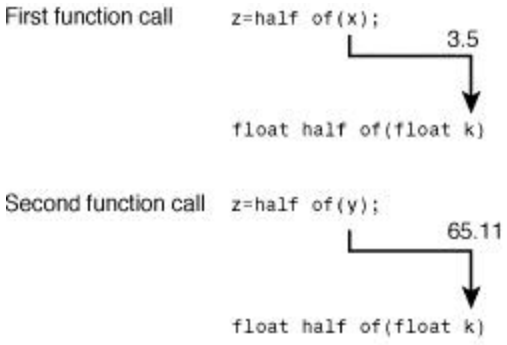

Figure 5-4. Each time a function is called, the arguments are passed to the function's parameters.

5-4-2. The Function Body

The *function body* is enclosed in braces, and it immediately follows the function header. It's here that the real work is done. When a function is called, execution begins at the start of the function body and terminates (returns to the calling program) when a return statement is encountered or when execution reaches the closing brace.

Local Variables

You can declare variables within the body of a function. Variables declared in a function are called *local variables*. The term *local* means that the variables are private to that particular function and are distinct from other variables of the same name declared elsewhere in the program. This will be explained shortly; for now, you should learn how to declare local variables.

A local variable is declared like any other variable, using the same variable types and rules for names that you learned on Day 3. Local variables can also be initialized when they are declared. You can declare any of C's variable types in a function. Here is an example of four local variables being declared within a function:

```
int func1(int y)
{
    int a, b = 10;
    float rate;
    double cost = 12.55;
    /* function code goes here... */
}
```

The preceding declarations create the local variables a, b, rate, and cost, which can be used by the code in the function. Note that the function parameters are considered to be variable declarations, so the variables, if any, in the function's parameter list also are available.

When you declare and use a variable in a function, it is totally separate and distinct from any other variables that are declared elsewhere in the program. This is true even if the variables have the same name. Listing 5.3 demonstrates this independence.

Listing 5.3. A demonstration of local variables.

```
1:   /* Demonstrates local variables. */
2:
```

```
3:   #include <stdio.h>
4:
5:   int x = 1, y = 2;
6:
7:   void demo(void);
8:
9:   main()
10:  {
11:    printf("\nBefore calling demo(), x = %d and y = %d.", x, y);
12:    demo();
13:    printf("\nAfter calling demo(), x = %d and y = %d\n.", x, y);
14:
15:    return 0;
16:  }
17:
18:  void demo(void)
19:  {
20:      /* Declare and initialize two local variables. */
21:
22:      int x = 88, y = 99;
23:
24:      /* Display their values. */
25:
26:      printf("\nWithin demo(), x = %d and y = %d.", x, y);
27:  }
Before calling demo(), x = 1 and y = 2.
Within demo(), x = 88 and y = 99.
After calling demo(), x = 1 and y = 2.
```

ANALYSIS: Listing 5.3 is similar to the previous programs in this chapter. Line 5 declares variables x and y. These are declared outside of any functions and therefore are considered global. Line 7 contains the prototype for our demonstration function, named demo(). It doesn't take any parameters, so it has void in the prototype. It also doesn't return any values, giving it a type of void. Line 9 starts our main() function, which is very simple. First, printf() is called on line 11 to display the values of x and y, and then the demo() function is called. Notice that demo() declares its own local versions of x and y on line 22. Line 26 shows that the local variables take precedence over any others. After the demo function is called, line 13 again prints the values of x and y. Because you are no longer in demo(), the original global values are printed.

As you can see, local variables x and y in the function are totally independent from the global variables x and y declared outside the function. Three rules govern the use of variables in functions:

- To use a variable in a function, you must declare it in the function header or the function body (except for global variables, which are covered on Day 12.
- In order for a function to obtain a value from the calling program, the value must be passed as an argument.
- In order for a calling program to obtain a value from a function, the value must be explicitly returned from the function.

To be honest, these "rules" are not strictly applied, because you'll learn how to get around them later in this book. However, follow these rules for now, and you should stay out of trouble.

Keeping the function's variables separate from other program variables is one way in which functions are independent. A function can perform any sort of data manipulation you want, using its own set of local variables. There's no worry that these manipulations will have an unintended effect on another part of the program.

Function Statements

There is essentially no limitation on the statements that can be included within a function. The only thing you can't do inside a function is define another function. You can, however, use all other C statements, including loops (these are covered on Day 6, "Basic Program Control"), if statements, and assignment statements. You can call library functions and other user-defined functions.

What about function length? C places no length restriction on functions, but as a matter of practicality, you should keep your functions relatively short. Remember that in structured programming, each function is supposed to perform a relatively simple task. If you find that a function is getting long, perhaps you're trying to perform a task too complex for one function alone. It probably can be broken into two or more smaller functions.

How long is too long? There's no definite answer to that question, but in practical experience it's rare to find a function longer than 25 or 30 lines of code. You have to use your own judgment. Some programming tasks require longer functions, whereas many functions are only a few lines long. As you gain programming experience, you will become more adept at determining what should and shouldn't be broken into smaller functions.

Returning a Value

To return a value from a function, you use the return keyword, followed by a C expression. When execution reaches a return statement, the expression is evaluated, and execution passes the value back to the calling program. The return value of the function is the value of the expression. Consider this function:

```
int func1(int var)
{
    int x;
    /* Function code goes here... */
    return x;
}
```

When this function is called, the statements in the function body execute up to the return statement. The return terminates the function and returns the value of x to the calling program. The expression that follows the return keyword can be any valid C expression.

A function can contain multiple return statements. The first return executed is the only one that has any effect. Multiple return statements can be an efficient way to return different values from a function, as demonstrated in Listing 5.4.

Listing 5.4. Using multiple return statements in a function.

```
1:   /* Demonstrates using multiple return statements in a function.
*/
2:
3:   #include <stdio.h>
4:
5:   int x, y, z;
6:
7:   int larger_of( int , int );
8:
9:   main()
10:  {
11:      puts("Enter two different integer values: ");
12:      scanf("%d%d", &x, &y);
13:
```

```
14:      z = larger_of(x,y);
15:
16:      printf("\nThe larger value is %d.", z);
17:
18:      return 0;
19:  }
20:
21:  int larger_of( int a, int b)
22:  {
23:      if (a > b)
24:          return a;
25:      else
26:          return b;
27:  }
Enter two different integer values:
200 300
The larger value is 300.
Enter two different integer values:
300
200
The larger value is 300.
```

ANALLYSIS: As in other examples, Listing 5.4 starts with a comment to describe what the program does (line 1). The STDIO.H header file is included for the standard input/output functions that allow the program to display information to the screen and get user input. Line 7 is the function prototype for larger_of(). Notice that it takes two int variables for parameters and returns an int. Line 14 calls larger_of() with x and y. The function larger_of() contains the multiple return statements. Using an if statement, the function checks to see whether a is bigger than b on line 23. If it is, line 24 executes a return statement, and the function immediately ends. Lines 25 and 26 are ignored in this case. If a isn't bigger than b, line 24 is skipped, the else clause is instigated, and the return on line 26 executes. You should be able to see that, depending on the arguments passed to the function larger_of(), either the first or the second return statement is executed, and the appropriate value is passed back to the calling function.

One final note on this program. Line 11 is a new function that you haven't seen before. puts()--meaning *put string*--is a simple function that displays a string to the standard output, usually the computer screen. (Strings are covered on Day 10, "Characters and Strings." For now, know that they are just quoted text.)

Remember that a function's return value has a type that is specified in the function header and function prototype. The value returned by the function must be of the same type, or the compiler generates an error message.

NOTE: Structured programming suggests that you have only one entry and one exit in a function. This means that you should try to have only one return statement within your function. At times, however, a program might be much easier to read and maintain with more than one return statement. In such cases, maintainability should take precedence.

5-4-3. The Function Prototype

A program must include a prototype for each function it uses. You saw an example of a function prototype on line 4 of Listing 5.1, and there have been function prototypes in the other listings as well. What is a function prototype, and why is it needed?

You can see from the earlier examples that the prototype for a function is identical to the function header, with a semicolon added at the end. Like the function header, the function prototype includes information about the function's return type, name, and parameters. The prototype's job is to tell the compiler about the function's return type, name, and parameters. With this information, the compiler can check every time your source code calls the function and verify that you're passing the correct number and type of arguments to the function and using the return value correctly. If there's a mismatch, the compiler generates an error message.

Strictly speaking, a function prototype doesn't need to exactly match the function header. The parameter names can be different, as long as they are the same type, number, and in the same order. There's no reason for the header and prototype not to match; having them identical makes source code easier to understand. Matching the two also makes writing a program easier. When you complete a function definition, use your editor's cut-and-paste feature to copy the function header and create the prototype. Be sure to add a semicolon at the end.

Where should function prototypes be placed in your source code? They should be placed before the start of main() or before the first function is defined. For readability, it's best to group all prototypes in one location.

DON'T try to return a value that has a type different from the function's type.

DO use local variables whenever possible.

DON'T let functions get too long. If a function starts getting long, try to break it into separate, smaller tasks.

DO limit each function to a single task.

DON'T have multiple return statements if they aren't needed. You should try to have one return when possible; however, sometimes having multiple return statements is easier and clearer.

5-5. Passing Arguments to a Function

To pass arguments to a function, you list them in parentheses following the function name. The number of arguments and the type of each argument must match the parameters in the function header and prototype. For example, if a function is defined to take two type int arguments, you must pass it exactly two int arguments--no more, no less--and no other type. If you try to pass a function an incorrect number and/or type of argument, the compiler will detect it, based on the information in the function prototype.

If the function takes multiple arguments, the arguments listed in the function call are assigned to the function parameters in order: the first argument to the first parameter, the second argument to the second parameter, and so on, as shown in Figure 5.5. *Multiple arguments are assigned to function parameters in order.*

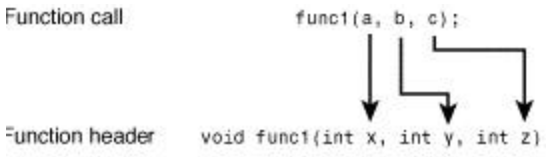

Figure 5-5. Multiple arguments are assigned to function parameters in order.

Each argument can be any valid C expression: a constant, a variable, a mathematical or logical expression, or even another function (one with a return value). For example, if half(), square(), and third() are all functions with return values, you could write

```
x = half(third(square(half(y))));
```

The program first calls half(), passing it y as an argument. When execution returns from half(), the program calls square(), passing half()'s return value as an argument. Next, third() is called with square()'s return value as the

argument. Then, half() is called again, this time with third()'s return value as an argument. Finally, half()'s return value is assigned to the variable x. The following is an equivalent piece of code:

```
a = half(y);
b = square(a);
c = third(b);
x = half(c);
```

5-6. Calling Functions

There are two ways to call a function. Any function can be called by simply using its name and argument list alone in a statement, as in the following example. If the function has a return value, it is discarded.

```
wait(12);
```

The second method can be used only with functions that have a return value. Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used. You've already seen an expression with a return value used as the right side of an assignment statement. Here are some more examples.

In the following example, half_of() is a parameter of a function:

```
printf("Half of %d is %d.", x, half_of(x));
```

First, the function half_of() is called with the value of x, and then printf() is called using the values x and half_of(x).

In this second example, multiple functions are being used in an expression:

```
y = half_of(x) + half_of(z);
```

Although half_of() is used twice, the second call could have been any other function. The following code shows the same statement, but not all on one line:

```
a = half_of(x);
b = half_of(z);
y = a + b;
```

The final two examples show effective ways to use the return values of functions. Here, a function is being used with the if statement:

```
if ( half_of(x) > 10 )
{
    /* statements; */          /* these could be any statements! */
}
```

If the return value of the function meets the criteria (in this case, if half_of() returns a value greater than 10), the if statement is true, and its statements are executed. If the returned value doesn't meet the criteria, the if's statements are not executed.

The following example is even better:

```
if ( do_a_process() != OKAY )
{
    /* statements; */          /* do error routine */
```

```
}
```

Again, I haven't given the actual statements, nor is do_a_process() a real function; however, this is an important example that checks the return value of a process to see whether it ran all right. If it didn't, the statements take care of any error handling or cleanup. This is commonly used with accessing information in files, comparing values, and allocating memory.

If you try to use a function with a void return type as an expression, the compiler generates an error message.

DO pass parameters to functions in order to make the function generic and thus reusable.

DO take advantage of the ability to put functions into expressions.

DON'T make an individual statement confusing by putting a bunch of functions in it. You should put functions into your statements only if they don't make the code more confusing.

5-6-1. Recursion

The term *recursion* refers to a situation in which a function calls itself either directly or indirectly. *Indirect recursion* occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations.

For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is written x! and is calculated as follows:

```
x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1
```

However, you can also calculate x! like this:

```
x! = x * (x-1)!
```

Going one step further, you can calculate (x-1)! using the same procedure:

```
(x-1)! = (x-1) * (x-2)!
```

You can continue calculating recursively until you're down to a value of 1, in which case you're finished. The program in Listing 5.5 uses a recursive function to calculate factorials. Because the program uses unsigned integers, it's limited to an input value of 8; the factorial of 9 and larger values are outside the allowed range for integers.

Listing 5.5. Using a recursive function to calculate factorials.

```
1:   /* Demonstrates function recursion. Calculates the */
2:   /* factorial of a number. */
3:
4:   #include <stdio.h>
5:
6:   unsigned int f, x;
7:   unsigned int factorial(unsigned int a);
8:
```

```
9:   main()
10:  {
11:      puts("Enter an integer value between 1 and 8: ");
12:      scanf("%d", &x);
13:
14:      if( x > 8 || x < 1)
15:      {
16:          printf("Only values from 1 to 8 are acceptable!");
17:      }
18:      else
19:      {
20:          f = factorial(x);
21:          printf("%u factorial equals %u\n", x, f);
22:      }
23:
24:      return 0;
25:  }
26:
27:  unsigned int factorial(unsigned int a)
28:  {
29:      if (a == 1)
30:          return 1;
31:      else
32:      {
33:          a *= factorial(a-1);
34:          return a;
35:      }
36:  }
Enter an integer value between 1 and 8:
6
6 factorial equals 720
```

ANALYSIS: The first half of this program is like many of the other programs you have worked with so far. It starts with comments on lines 1 and 2. On line 4, the appropriate header file is included for the input/output routines. Line 6 declares a couple of unsigned integer values. Line 7 is a function prototype for the factorial function. Notice that it takes an unsigned int as its parameter and returns an unsigned int. Lines 9 through 25 are the main() function. Lines 11 and 12 print a message asking for a value from 1 to 8 and then accept an entered value.

Lines 14 through 22 show an interesting if statement. Because a value greater than 8 causes a problem, this if statement checks the value. If it's greater than 8, an error message is printed; otherwise, the program figures the factorial on line 20 and prints the result on line 21. When you know there could be a problem, such as a limit on the size of a number, add code to detect the problem and prevent it.

Our recursive function, factorial(), is located on lines 27 through 36. The value passed is assigned to a. On line 29, the value of a is checked. If it's 1, the program returns the value of 1. If the value isn't 1, a is set equal to itself times the value of factorial(a-1). The program calls the factorial function again, but this time the value of a is (a-1). If (a-1) isn't equal to 1, factorial() is called again with ((a-1)-1), which is the same as (a-2). This process continues until the if statement on line 29 is true. If the value of the factorial is 3, the factorial is evaluated to the following:

```
3 * (3-1) * ((3-1)-1)
```

DO understand and work with recursion before you use it.

DON'T use recursion if there will be several iterations. (An iteration is the repetition of a program statement.) Recursion uses many resources, because the function has to remember where it is.

5-7. Where the Functions Belong

You might be wondering where in your source code you should place your function definitions. For now, they should go in the same source code file as main() and after the end of main(). Figure 5.6 shows the basic structure of a program that uses functions.

You can keep your user-defined functions in a separate source-code file, apart from main(). This technique is useful with large programs and when you want to use the same set of functions in more than one program. This technique is discussed on Day 21, "Advanced Compiler Use."

```
/*  start of source code *
    ...
    prototypes here
    ...
    main()
    {
       ...
       ...
    }
    func1()
    {
       ...
    }
    func2()
    {
       ...
    }
/*  end of source code *
```

Figure 5-6. Place your function prototypes before main() and your function definitions after main().

5-8. Summary

This chapter introduced you to functions, an important part of C programming. Functions are independent sections of code that perform specific tasks. When your program needs a task performed, it calls the function that performs that task. The use of functions is essential for structured programming--a method of program design that emphasizes a modular, top-down approach. Structured programming creates more efficient programs and also is much easier for you, the programmer, to use.

You also learned that a function consists of a header and a body. The header includes information about the function's return type, name, and parameters. The body contains local variable declarations and the C statements that are executed when the function is called. Finally, you saw that local variables--those declared within a function--are totally independent of any other program variables declared elsewhere.

Q&A

Q What if I need to return more than one value from a function?
A Many times you will need to return more than one value from a function, or, more commonly, you will want to change a value you send to the function and keep the change after the function ends. This process is covered on Day 18, "Getting More from Functions."
Q How do I know what a good function name is?
A A good function name describes as specifically as possible what the function does.
Q When variables are declared at the top of the listing, before main(), they can be used anywhere, but local variables can be used only in the specific function. Why not just declare everything before main()?
A Variable scope is discussed in more detail on Day 12.
Q What other ways are there to use recursion?
A The factorial function is a prime example of using recursion. The factorial number is needed in many statistical calculations. Recursion is just a loop; however, it has one difference from other loops. With recursion, each time a recursive function is called, a new set of variables is created. This is not true of the other loops that you will learn about in the next chapter.
Q Does main() have to be the first function in a program?
A No. It is a standard in C that the main() function is the first function to execute; however, it can be placed anywhere in your source file. Most people place it first so that it's easy to locate.
Q What are member functions?

A Member functions are special functions used in C++ and Java. They are part of a class--which is a special type of structure used in C++ and Java.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. Will you use structured programming when writing your C programs?
2. How does structured programming work?
3. How do C functions fit into structured programming?
4. What must be the first line of a function definition, and what information does it contain?
5. How many values can a function return?
6. If a function doesn't return a value, what type should it be declared?
7. What's the difference between a function definition and a function prototype?
8. What is a local variable?
9. How are local variables special?
10. Where should the main() function be placed?

Exercises

1. Write a header for a function named do_it() that takes three type char arguments and returns a type float to the calling program.
2. Write a header for a function named print_a_number() that takes a single type int argument and doesn't return anything to the calling program.
3. What type value do the following functions return?
a. int print_error(float err_nbr);
b. long read_record(int rec_nbr, int size);
4. BUG BUSTER: What's wrong with the following listing?

```
#include <stdio.h>
void print_msg( void );
main()
{
    print_msg( "This is a message to print" );
    return 0;
}
void print_msg( void )
{
    puts( "This is a message to print" );
    return 0;
}
```

5. BUG BUSTER: What's wrong with the following function definition?

```
int twice(int y);
{
    return (2 * y);
}
```

6. Rewrite Listing 5.4 so that it needs only one return statement in the larger_of() function.
7. Write a function that receives two numbers as arguments and returns the value of their product.
8. Write a function that receives two numbers as arguments. The function should divide the first number by the second. Don't divide by the second number if it's zero. (Hint: Use an if statement.)
9. Write a function that calls the functions in exercises 7 and 8.
10. Write a program that uses a function to find the average of five type float values entered by the user.
11. Write a recursive function to take the value 3 to the power of another number. For example, if 4 is passed, the function will return 81.