Functions

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.

- *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. Thiss 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.

A Function Illustrated

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

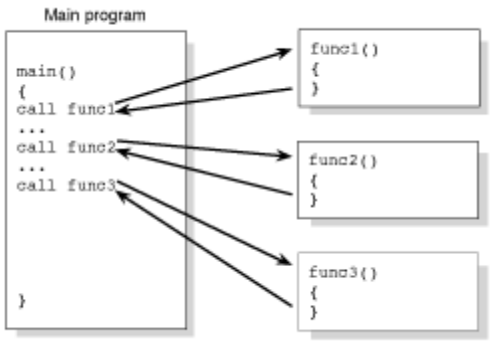
```
1:   /* Demonstrates a simple function */
2:   #include <stdio.h>
3:
4:   long cube(long x);
5:
6:   long input, answer;
7:
8:   main()
9:   {
10:     printf("Enter an integer value: ");
11:     scanf("%d", &input);
12:     answer = cube(input);
13:     /* Note: %ld is the conversion specifier for */
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.
```

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.



.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" );
}
```

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.

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.

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.

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.

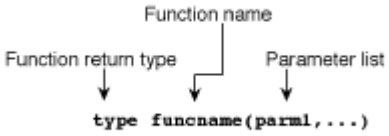


Figure 5.3. *The three components of a function header.*

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

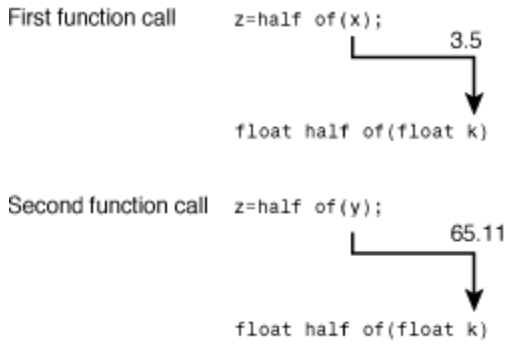
be different. In the function, the argument is accessed by using the corresponding parameter name.

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
```

The relationship between arguments and parameters.



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

-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..

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.
```

.

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
- 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.

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.

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.

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.
```

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 below 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.

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.
