CHAPTER 8

Structuring Your Programs

I mentioned in Chapter 1 that breaking up a program into reasonably self-contained units is basic to the development of any program of a practical nature. When confronted with a big task, the most sensible thing to do is break it up into manageable chunks. You can then deal with each small chunk fairly easily and be reasonably sure that you've done it properly. If you design the chunks of code carefully, you may be able to reuse some of them in other programs.

One of the key ideas in the C language is that every program should be segmented into functions that are relatively short. Even with the examples you've seen so far that were written as a single function, `main()`, other functions are also involved because you've still used a variety of standard library functions for input and output, for mathematical operations, and for handling strings.

In this chapter, you'll look at how you can make your programs more effective and easier to develop by introducing more functions of your own.

In this chapter you'll learn:

- How data is passed to a function
- How to return results from your functions
- How to define your own functions
- The advantages of pointers as arguments to functions

Program Structure

As I said right at the outset, a C program consists of one or more functions, the most important of which is the function `main()` where execution starts. When you use library functions such as `printf()` or `scanf()`, you see how one function is able to call up another function in order to carry out some particular task and then continue execution back in the calling function when the task is complete. Except for side effects on data stored at global scope, each function in a program is a self-contained unit that carries out a particular operation. When a function is called, the code within the body of that function is executed, and when the function has finished executing, control returns to the point at which that function was called. This is illustrated in Figure 8-1, where you can see an idealized representation of a C program structured as five functions. It doesn't show any details of the statements involved—just the sequence of execution.

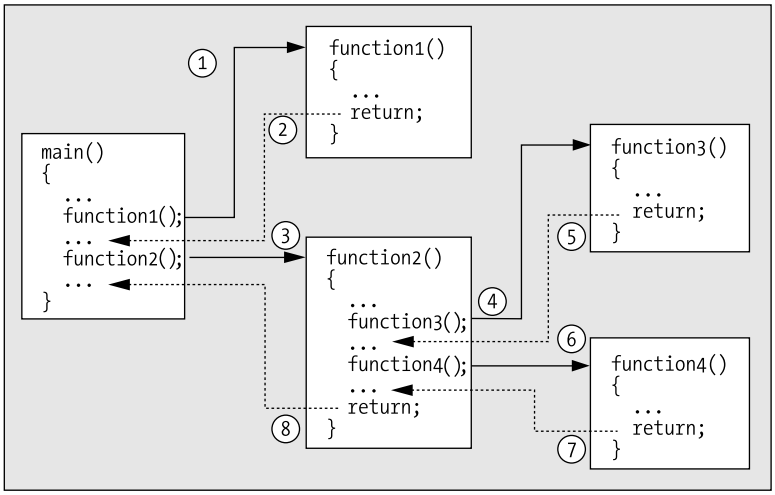

Figure 8-1. *Execution of a program made up of several functions*

The program steps through the statements in sequence in the normal way until it comes across a call to a particular function. At that point, execution moves to the start of that function—that is, the first statement in the body of the function. Execution of the program continues through the function statements until it hits a `return` statement or reaches the closing brace marking the end of the function body. This signals that execution should go back to the point immediately after where the function was originally called.

The set of functions that make up a program link together through the function calls and their `return` statements to perform the various tasks necessary for the program to achieve its purpose. Figure 8-1 shows each function in the program executed just once. In practice, each function can be executed many times and can be called from several points within a program. You've already seen this in the examples that called the `printf()` and `scanf()` functions several times.

Before you look in more detail at how to define your own functions, I need to explain a particular aspect of the way variables behave that I've glossed over so far.

Variable Scope and Lifetime

In all the examples up to now, you've declared the variables for the program at the beginning of the block that defines the body of the function `main()`. But you can actually define variables at the beginning of *any* block. Does this make a difference? "It most certainly does, Stanley," as Ollie would have said. Variables exist only within the block in which they're defined. They're created when they are declared, and they cease to exist at the next closing brace.

This is also true of variables that you declare within blocks that are inside other blocks. The variables declared at the beginning of an outer block also exist in the inner block. These variables are freely accessible, as long as there are no other variables with the same name in the inner block, as you'll see.

Variables that are created when they're declared and destroyed at the end of a block are called **automatic variables**, because they're automatically created and destroyed. The extent within the program code where a given variable is visible and can be referenced is called the variable's **scope**. When you use a variable within its scope, everything is OK. But if you try to reference a variable outside its scope, you'll get an error message when you compile the program because the variable doesn't exist outside of its scope. The general idea is illustrated in the following code fragment:

```
{
  int a = 0;                                    /* Create a        */
  /* Reference to a is OK here       */
  /* Reference to b is an error here */
  {
      int b = 10;                               /* Create b        */
      /* Reference to a and b is OK here */
  }                                             /* b dies here     */
  /* Reference to b is an error here */
  /* Reference to a is OK here       */
}                                               /* a dies here     */
```

All the variables that are declared within a block die and no longer exist after the closing brace of the block. The variable a is visible within both the inner and outer blocks because it's declared in the outer block. The variable b is visible only within the inner block because it's declared within that block.

While your program is executing, a variable is created and memory is allocated for it. At some point, which for automatic variables is the end of the block in which the variable is declared, the memory that the variable occupies is returned back to the system. Of course, while functions called within the block are executing, the variable continues to exist; it is only destroyed when execution reaches the end of the block in which it was created. The time period during which a variable is in existence is referred to as the **lifetime** of the variable.

Let's explore the implications of a variable's scope through an example.

TRY IT OUT: UNDERSTANDING SCOPE

Let's take a simple example that involves a nested block that happens to be the body of a loop:

```
/* Program 8.1 A microscopic program about scope */
#include <stdio.h>
int main(void)
{
  int count1 = 1;                               /* Declared in outer block */

  do
  {
    int count2 = 0;                             /* Declared in inner block */
    ++count2;
    printf("\ncount1 = %d     count2 = %d", count1,count2);
  } while( ++count1 <= 8 );

  /* count2 no longer exists */

  printf("\ncount1 = %d\n", count1);
  return 0;
}
```

You will get the following output from this program:

```
count1 = 1     count2 = 1
count1 = 2     count2 = 1
count1 = 3     count2 = 1
count1 = 4     count2 = 1
count1 = 5     count2 = 1
count1 = 6     count2 = 1
count1 = 7     count2 = 1
count1 = 8     count2 = 1
count1 = 9
```

How It Works

The block that encloses the body of `main()` contains an inner block that is the `do-while` loop. You declare and define `count2` inside the loop block:

```
do
{
  int count2 = 0; /* Declared in inner block */
  ++count2;
  printf("\ncount1 = %d     count2 = %d", count1,count2);
} while( ++count1 <= 8 );
```

As a result, the value of `count2` is never more than 1. During each iteration of the loop, the variable `count2` is created, initialized, incremented, and destroyed. It only exists from the statement that declares it down to the closing brace for the loop. The variable `count1`, on the other hand, exists at the `main()` block level. It continues to exist while it is incremented, so the last `printf()` produces the value 9.

Try modifying the program to make the last `printf()` output the value of `count2`. It won't compile. You'll get an error because, at the point where the last `printf()` is, `count2` no longer exists. From this you may guess, correctly, that failing to initialize automatic variables before you use them can cause untold chaos, because the memory that they occupy may be reallocated to something else at the end of their existence. As a consequence, next time around, your uninitialized variables may contain anything but what you expect.

TRY IT OUT: MORE ABOUT SCOPE

Let's try a slight modification of the last example:

```
/* Program 8.2 More scope in this one */
#include <stdio.h>
int main(void)
{
  int count = 0;                                /* Declared in outer block */
  do
  {
    int count = 0;               /* This is another variable called count */
    ++count;                     /* this applies to inner count           */
    printf("\ncount = %d ", count);
  }
  while( ++count <= 8 );                   /* This works with outer count */
```

```
  /* Inner count is dead, this is outer */
  printf("\ncount = %d\n", count);
  return 0;
}
```

Now you've used the same variable name, `count`, at the `main()` block level and in the loop block. Observe what happens when you compile and run this:

```
count = 1
count = 1
count = 1
count = 1
count = 1
count = 1
count = 1
count = 1
count = 9
```

How It Works

The output is boring, but interesting at the same time. You actually have two variables called `count`, but inside the loop block the local variable will "hide" the version of `count` that exists at the `main()` block level. The compiler will assume that when you use the name `count`, you mean the one that was declared in the current block. Inside the `while` loop, only the local version of `count` can be reached, so that is the variable being incremented. The `printf()` inside the loop block displays the local `count` value, which is always 1, for the reasons given previously. As soon as you exit from the loop, the outer `count` variable becomes visible, and the last `printf()` displays its exit value from the loop as 9.

Clearly, the variable that is controlling the loop is the one declared at the beginning of `main()`. This little example demonstrates why it isn't a good idea to use the same variable name for two different variables in a function, even though it's legal. At best, it's most confusing. At worst, you'll be thinking "that's another fine mess I've gotten myself into."

Variable Scope and Functions

The last point to note, before I get into the detail of creating functions, is that the body of every function is a block (which may contain other blocks, of course). As a result, the automatic variables that you declare within a function are local to the function and don't exist elsewhere. Therefore, the variables declared within one function are quite independent of those declared in another function or in a nested block. There's nothing to prevent you from using the same name for variables in different functions; they will remain quite separate.

This becomes more significant when you're dealing with large programs in which the problem of ensuring unique variables can become a little inconvenient. It's still a good idea to avoid any unnecessary or misleading overlapping of variable names in your various functions and, of course, you should try to use names that are meaningful to make your programs easy to follow. You'll see more about this as you explore functions in C more deeply.

Functions

You've already used built-in functions such as `printf()` or `strcpy()` quite extensively in your programs. You've seen how these built-in functions are executed when you reference them by name and how

you are able to transfer information to a function by means of arguments between parentheses following the function name. With the `printf()` function, for instance, the first argument is usually a string literal, and the succeeding arguments (of which there may be none) are a series of variables or expressions whose values are to be displayed.

You've also seen how you can receive information back from a function in two ways. The first way is through one of the function arguments in parentheses. You provide an address of a variable through an argument to a function, and the function places a value in that variable. When you use `scanf()` to read data from the keyboard, for instance, the input is stored in an address that you supply as an argument. The second way is that you can receive information back from a function as a **return value**. With the `strlen()` function, for instance, the length of the string that you supply as the argument appears in the program code in the position where the function call is made. Thus, if `str` is the string `"example"`, in the expression `2*strlen(str)`, the value 7, which the function returns, replaces the function call in the expression. The expression will therefore amount to `2*7`. Where a function returns a value of a given type, the function call can appear as part of any expression where a variable of the same type could be used.

Of necessity you've written the function `main()` in all your programs, so you already have the basic knowledge of how a function is constructed. So let's look at what makes up a function in more detail.

Defining a Function

When you create a function, you need to specify the **function header** as the first line of the function definition, followed by the executable code for the function enclosed between braces. The block of code between braces following the function header is called the **function body**.

- The function header defines the name of the function, the function parameters (in other words, what types of values are passed to the function when it's called), and the type for the value that the function returns.
- The function body determines what calculations the function performs on the values that are passed to it.

The general form of a function is essentially the same as you've been using for `main()`, and it looks like this:

```
Return_type  Function_name( Parameters - separated by commas )
{
   Statements;
}
```

The statements in the function body can be absent, but the braces must be present. If there are no statements in the body of a function, the return type must be `void`, and the function will have no effect. You'll recall that I said that the type `void` means "absence of any type," so here it means that the function doesn't return a value. A function that does not return a value must also have the return type specified as `void`. Conversely, for a function that does not have a `void` return type, every return statement in the function body must return a value of the specified return type.

Although it may not be immediately apparent, presenting a function with a content-free body is often useful during the testing phase of a complicated program. This allows you to run the program with only selected functions actually doing something; you can then add the function bodies, step by step, until the whole thing works.

The parameters between parentheses are placeholders for the argument values that you must specify when you call a function. The term **parameter** refers to a placeholder in the function definition that specifies the type of value that should be passed to the function when it is called. A parameter consists of the type followed by the parameter name that is used within the body of the function to

refer to that value when the function executes. The term **argument** refers to the value that you supply corresponding to a parameter when you call a function. I'll explain parameters in more detail in the "Function Parameters" section later in this chapter.

■ **Note** The statements in the body of a function can also contain nested blocks of statements. But you can't define a function inside the body of another function.

The general form for calling a function is the following expression:

```
Function_name(List of Arguments - separated by commas)
```

You simply use the function's name followed by a list of arguments separated by commas in parentheses, just as you've been doing with functions such as `printf()` and `scanf()`. A function call can appear as a statement on a line by itself, like this:

```
printf("A fool and your money are soon partners,");
```

A function that's called like this can be a function that returns a value. In this case the value that's returned is simply discarded. A function that has been defined with a return type of `void` can *only* be called like this.

A function that returns a value can, and usually does, participate in an expression. For example

```
result = 2.0*sqrt(2.0);
```

Here, the value that's returned by the `sqrt()` function (declared in the `<math.h>` header file) is multiplied by 2.0 and the result is stored in the variable `result`. Obviously, because a function with a `void` return type doesn't return anything, it can't possibly be part of an expression.

Naming a Function

The name of a function can be any legal name in C that isn't a reserved word (such as `int`, `double`, `sizeof`, and so on) and isn't the same as the name of another function in your program. You should take care not to use the same names as any of the standard library functions because this would prevent you from using the library function of the same name, and it would also be very confusing.

One way of differentiating your function names from those in the standard library is to start them with a capital letter, although some programmers find this rather restricting. A legal name has the same form as that of a variable: a sequence of letters and digits, the first of which must be a letter. As with variable names, the underline character counts as a letter. Other than that, the name of a function can be anything you like, but ideally the name that you choose should give some clue as to what the function does. Examples of valid function names that you might create are as follows:

```
cube_root    FindLast    Explosion    Back2Front
```

You'll often want to define function names (and variable names, too) that consist of more than one word. There are two common approaches you can adopt that happen to be illustrated in the first two examples:

- Separate each of the words in a function name with an underline character.
- Capitalize the first letter of each word.

Both approaches work well. Which one you choose is up to you, but it's a good idea to pick an approach and stick to it. You can, of course, use one approach for functions and the other for variables. Within this book, both approaches have been sprinkled around to give you a feel for how they

look. By the time you reach the end of the book, you'll probably have formed your own opinion as to which approach is best.

Function Parameters

Function parameters are defined within the function header and are placeholders for the arguments that need to be specified when the function is called. The parameters for a function are a list of parameter names and their types, and successive parameters are separated by commas. The entire list of parameters is enclosed between the parentheses that follow the function name.

Parameters provide the means by which you pass information *from* the calling function *into* the function that is called. These parameter names are local to the function, and the values that are assigned to them when the function is called are referred to as arguments. The computation in the body of the function is then written using these parameter names, which will have the values of the arguments when the function executes. Of course, a function may also have locally defined automatic variables declared within the body of the function. Finally, when the computation has finished, the function will exit and return an appropriate value back to the original calling statement.

Some examples of typical function headers are shown in Table 8-1.

Table 8-1. *Examples of Function Headers*

Function Header	Description
`bool SendMessage(char *text)`	This function has one parameter, `text`, which is of type "pointer to `char`," and it returns a value of type `bool`.
`void PrintData(int count, double *data)`	This function has two parameters, one of type `int` and the other of type "pointer to `double`." The function does not return a value.
`char message GetMessage(void)`	This function has no parameters and returns a pointer of type `char`.

You call a function by using the function name followed by the arguments to the function between parentheses. When you actually call the function by referencing it in some part of your program, the arguments that you specify in the call will replace the parameters in the function. As a result, when the function executes, the computation will proceed using the values that you supplied as arguments. The arguments that you specify when you call a function need to agree in type, number, and sequence with the parameters that are specified in the function header. The relationship and information passing between the calling and called function is illustrated in Figure 8-2.

If the type of an argument to a function does not match the type of the corresponding parameter, the compiler will insert a conversion of the argument value to the parameter type where this is possible. This may result in truncation of the argument value, when you pass a value of type `double` for a parameter of type `int`, for example, so this is a dangerous practice. If the compiler cannot convert an argument to the required type, you will get an error message.

If there are no parameters to a function, you can specify the parameter list as `void`, as you have been doing in the case of the `main()` function.

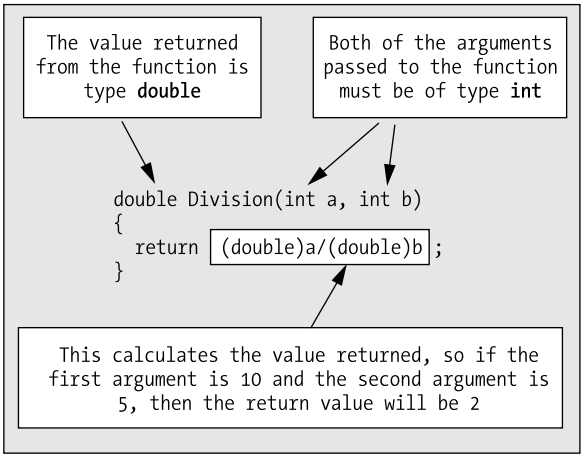

Figure 8-2. *Passing arguments to a function*

Specifying the Return Value Type

Let's take another look at the general form of a function:

```
Return_type  Function_name(List of Parameters - separated by commas)
{
   Statements;
}
```

The `Return_type` specifies the type of the value returned by the function. If the function is used in an expression or as the right side of an assignment statement, the return value supplied by the function will effectively be substituted for the function in its position. The type of value to be returned by a function can be specified as any of the legal types in C, including pointers. The type can also be specified as `void`, meaning that no value is returned. As noted earlier, a function with a `void` return type can't be used in an expression or anywhere in an assignment statement.

The return type can also be type `void *`, which is a "pointer to `void`." The value returned in this case is an address value but with no specified type. This type is used when you want the flexibility to be able to return a pointer that may be used for a variety of purposes, as in the case of the `malloc()` function for allocating memory. The most common return types are shown in Table 8-2.

Table 8-2. *Common Return Types*

Type	Meaning	Type	Meaning
`int`	Integer, 2 or 4 bytes	`int *`	Pointer to `int`
`short`	Integer, 2 bytes	`short *`	Pointer to `short`
`long`	Integer, 4 bytes	`long *`	Pointer to `long`
`long long`	Integer, 8 bytes	`long long *`	Pointer to `long`
`char`	Character, 1 byte	`char *`	Pointer to `char`
`float`	Floating-point, 4 bytes	`float *`	Pointer to `float`

Table 8-2. *Common Return Types (Continued)*

Type	Meaning	Type	Meaning
`double`	Floating-point, 8 bytes	`double *`	Pointer to `double`
`long double`	Floating-point, 12 bytes	`long double *`	Pointer to `long double`
`void`	No value	`void *`	Pointer to undefined type

Of course, you can also specify the return type to a function to be an unsigned integer type, or a pointer to an unsigned integer type. A return type can also be an enumeration type or a pointer to an enumeration type. If a function has its return type specified as other than `void`, it must return a value. This is achieved by executing a `return` statement to return the value.

■ **Note** In Chapter 11 you will be introduced to objects called `struct`s that provide a way to work with aggregates of several data items as a single unit. A function can have parameters that are `struct`s to pointers to a `struct` type and can also return a `struct` or a pointer to a `struct`.

The return Statement

The `return` statement provides the means of exiting from a function and resuming execution of the calling function at the point from which the call occurred. In its simplest form, the return statement is just this:

```
return;
```

In this form, the `return` statement is being used in a function where the return type has been declared as `void`. It doesn't return any value. However, the more general form of the `return` statement is this:

```
return expression;
```

This form of `return` statement must be used when the return value type for the function has been declared as some type other than `void`. The value that's returned to the calling program is the value that results when `expression` is evaluated.

■ **Caution** You'll get an error message if you compile a program that contains a function defined with a `void` return type that tries to return a value. You'll get an error message from the compiler if you use a bare `return` in a function where the return type was specified to be other than `void`.

The return `expression` can be any expression, but it should result in a value that's the same type as that declared for the return value in the function header. If it isn't of the same type, the compiler will insert a conversion from the type of the `return expression` to the one required where this is possible. The compiler will produce an error message if the conversion isn't possible.

There can be more than one `return` statement in a function, but each `return` statement must supply a value that is convertible to the type specified in the function header for the return value.

Note The calling function doesn't have to recognize or process the value returned from a called function. It's up to you how you use any values returned from function calls.

TRY IT OUT: USING FUNCTIONS

It's always easier to understand new concepts with an example, so let's start with a trivial illustration of a program that consists of two functions. You'll write a function that will compute the average of two floating-point variables, and you'll call that function from `main()`. This is more to illustrate the mechanism of writing and calling functions than to present a good example of their practical use.

```
/* Program 8.3 Average of two float values */
#include <stdio.h>

/* Definition of the function to calculate an average */
float average(float x, float y)
{
   return (x + y)/2.0f;
}

/* main program - execution always starts here */
int main(void)
{
  float value1 = 0.0F;
  float value2 = 0.0F;
  float value3 = 0.0F;

  printf("Enter two floating-point values separated by blanks: ");
  scanf("%f %f", &value1, &value2);
  value3 = average(value1, value2);
  printf("\nThe average is: %f\n",  value3);
  return 0;
}
```

Typical output of this program would be the folllowing:

```
Enter two floating-point values separated by blanks: 2.34 4.567

The average is: 3.453500
```

How It Works

Let's go through this example step by step. Figure 8-3 describes the order of execution in the example.

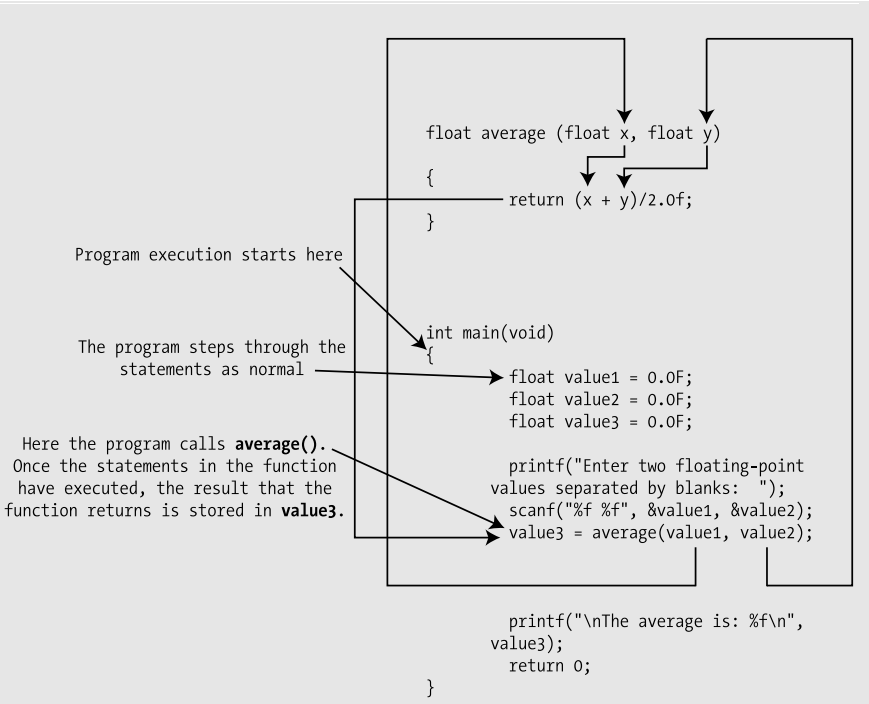

Figure 8-3. *Order of execution*

As you already know, execution begins at the first executable statement of the function `main()`. The first statement in the body of `main()` is the following:

```
printf("Enter two floating-point values separated by blanks: ");
```

There's nothing new here—you simply make a call to the function `printf()` with one argument, which happens to be a string. The actual value transferred to `printf()` will be a pointer containing the address of the beginning of the string that you've specified as the argument. The `printf()` function will then display the string that you've supplied as that argument.

The next statement is also a familiar one:

```
scanf("%f %f",&value1, &value2);
```

This calls the input function `scanf()`. There are three arguments: a string, which is therefore effectively a pointer, as in the previous statement; the address of the first variable, which again is effectively a pointer; and the address of the second variable—again, effectively a pointer. As I've discussed, `scanf()` must have the addresses for those last two arguments to allow the input data to be stored in them. You'll see why a little later in this chapter.

Once you've read in the two values, the assignment statement is executed:

```
value3 = average(value1, value2);
```

This calls the `average()`function, which expects two values of type `float` as arguments, and you've correctly supplied `value1` and `value2`, which are both of type `float`.

The first executable statement that actually does something is this:

```
printf("Enter two floating-point values separated by blanks: ");
```

There's nothing new here—you simply make a call to the function `printf()` with one argument, which happens to be a string. The actual value transferred to `printf()` will be a pointer containing the address of the beginning of the string that you've specified as the argument. The `printf()` function will then display the string that you've supplied as that argument.

The next statement is also a familiar one:

```
scanf("%f %f",&value1, &value2);
```

This calls the input function `scanf()`. There are three arguments: a string, which is therefore effectively a pointer, as in the previous statement; the address of the first variable, which again is effectively a pointer; and the address of the second variable—again, effectively a pointer. As I've discussed, `scanf()` must have the addresses for those last two arguments to allow the input data to be stored in them. You'll see why a little later in this chapter.

Once you've read in the two values, the assignment statement is executed:

```
value3 = average(value1, value2);
```

This calls the `average()`function, which expects two values of type `float` as arguments, and you've correctly supplied `value1` and `value2`, which are both of type `float`. These two values are accessed as `x` and `y` in the body of `average()` when it executes. The result that is returned by the function is then stored in `value3`. The last `printf()` call then outputs the value of `value3`.

The Pass-By-Value Mechanism

There's a very important point to be made in Program 8.3, which is illustrated in Figure 8-4.

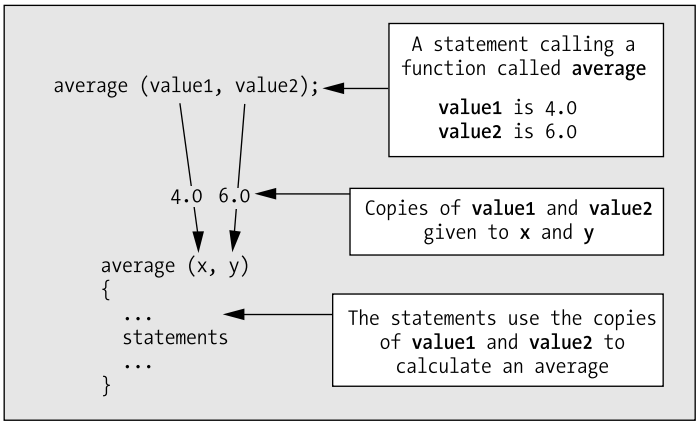

Figure 8-4. *Passing arguments to a function*

The important point is this: *copies* of the values of `value1` and `value2` are transferred to the function as arguments, *not the variables themselves*. This means that the function can't change the values

stored in value1 or value2. For instance, if you input the values 4.0 and 6.0 for the two variables, the compiler will create separate copies of these two values on the stack, and the average() function will have access to these copies when it's called. This mechanism is how all argument values are passed to functions in C, and it's termed the **pass-by-value mechanism**.

The only way that a called function can change a variable belonging to the calling function is by receiving an argument value that's the address of the variable. When you pass an address as an argument to a function, it's still only a copy of the address that's actually passed to the function, not the original. However, the copy is still the address of the original variable. I'll come back to this point later in the chapter in the section "Pointers As Arguments and Return Values."

The average() function is executed with the values from value1 and value2 substituted for the parameter names in the function, which are x and y, respectively. The function is defined by the following statements:

```
float average(float x, float y)
{
  return (x + y)/2.0f;
}
```

The function body has only one statement, which is the return statement. But this return statement does all the work you need. It contains an expression for the value to be returned that works out the average of x and y. This value then appears in place of the function in the assignment statement back in main(). So, in effect, a function that returns a value acts like a variable of the same type as the return value.

Note that without the f on the constant 2.0, this number would be of type double and the whole expression would be evaluated as type double. Because the compiler would then arrange to cast this to type float for the return value, you would get a warning message during compilation because data could be lost when the value is converted from type double to type float before it is returned.

Instead of assigning the result of the function to value3, you could have written the last printf() statement as follows:

```
printf("\nThe average is: %f", average(value1, value2));
```

Here the function average() would receive copies of the values of the arguments. The return value from the function would be supplied as an argument to printf() directly, even though you haven't explicitly created a place to store it. The compiler would take care of allocating some memory to store the value returned from average(). It would also take the trouble to make a copy of this to hand over to the function printf() as an argument. Once the printf() had been executed, however, you would have had no means of accessing the value that was returned from the function average().

Another possible option is to use explicit values in a function call. What happens then? Let's stretch the printf() statement out of context now and rewrite it like this:

```
printf("\nThe average is: %f", average(4.0, 6.0));
```

In this case if the literal 4.0 and 6.0 don't already exist, the compiler would typically create a memory location to store each of the constant arguments to average(), and then supply copies of those values to the function—exactly as before. A copy of the result returned from the function average() would then be passed to printf().

Figure 8-5 illustrates the process of calling a function named useful() from main() and getting a value returned.

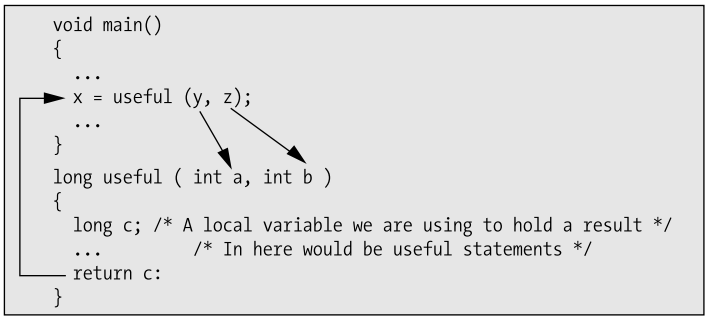

Figure 8-5. *Argument passing and the return value*

The arrows in the figure show how the values correspond between `main()` and `useful()`. Notice that the value returned from `useful()` is finally stored in the variable `x` in `main()`. The value that's returned from `useful()` is stored in the variable `c` that is local to the `useful()` function. This variable ceases to exist when execution of the `useful()` function ends, but the returned value is still safe because a copy of the value is returned, not the original value stored in `c`. Thus, returning a value from a function works in the same way as passing an argument value.

Function Declarations

In a variation of Program 8.3, you could define the function `main()` first and then the function `average()`:

```
#include <stdio.h>

int main(void)
{
  /* Code in main() ... */
}

float average(float x, float y)
{
  return (x + y)/2.0;
}
```

As it stands this won't compile. When the compiler comes across the call to the `average()` function, it will have no idea what to do with it, because at that point the `average()` function has not been defined. For this to compile you must add something before the definition of `main()` that tells the compiler about the `average()` function.

A **function declaration** is a statement that defines the essential characteristics of a function. It defines its name, its return value type, and the type of each of its parameters. You can actually write it exactly the same as the function header and just add a semicolon at the end if you want. A function declaration is also called a **function prototype**, because it provides all the external specifications for the function. A function prototype enables the compiler to generate the appropriate instructions at each point where you use the function and to check that you use it correctly in each case. When you include a header file in a program, the header file adds the function prototypes for library functions to the program. For example, the header file `<stdio.h>` contains function prototypes for `printf()` and `scanf()`, among others.

To get the variation of Program 8.3 to compile, you just need to add the function prototype for `average()` before the definition of `main()`:

```
#include <stdio.h>
float average(float, float);                    /* Function prototype */

int main(void)
{
  /* Code in main() ... */
}

float average(float x, float y)
{
  return (x + y)/2.0;
}
```

Now the compiler can compile the call to `average()` in `main()` because it knows all its characteristics, its name, its parameter types, and its return type. Technically, you could put the declaration for the function `average()` within the body of `main()`, prior to the function call, but this is never done. Function prototypes generally appear at the beginning of a source file prior to the definitions of any of the functions. The function prototypes are then external to all of the functions in the source file, and their scopes extend to the end of the source file, thereby allowing any of the functions in the file to call any function regardless of where you've placed the definitions of the functions.

It's good practice to always include declarations for all of the functions in a program source file, regardless of where they're called. This approach will help your programs to be more consistent in design, and it will prevent any errors occurring if, at any stage, you choose to call a function from another part of your program. Of course, you never need a function prototype for `main()` because this function is only called by the host environment when execution of a program starts.

Pointers As Arguments and Return Values

You've already seen how it's possible to pass a pointer as an argument to a function. More than that, you've seen that this is essential if a function is to modify the value of a variable that's defined in the calling function. In fact, this is the only way it can be done. So let's explore how all this works out in practice with another example.

TRY IT OUT: FUNCTIONS USING ORDINARY VARIABLES

Let's first take an elementary example of a function that doesn't use a pointer argument. Here, you're going to try to change the contents of a variable by passing it as an argument to a function, changing it, and then returning it. You'll output its value both within the function and back in `main()` to see what the effect is.

```
/* Program 8.4 The change that doesn't  */
#include <stdio.h>

int change(int number);                 /* Function prototype             */

int main(void)
{
  int number = 10;                      /* Starting Value                 */
  int result = 0;                       /* Place to put the returned value */
```

```
  result = change(number);
  printf("\nIn main, result = %d\tnumber = %d", result, number);
  return 0;
}

/* Definition of the function change() */
int change(int number)
{
  number = 2 * number;
  printf("\nIn function change, number = %d\n", number);
  return number;
}
```

The output from this program is the following:

```
In function change, number = 20
In main, result = 20    number = 10
```

How It Works

This example demonstrates that you can't change the world without pointers. You can only change the values locally within the function.

The first thing of note in this example is that you put the prototype for the function `change()` outside of `main()` along with the `#include` directive:

```
#include <stdio.h>

int change(int number);                /* Function prototype              */
```

This makes the function declaration *global*, and if you had other functions in the example they would all be able to use this function.

In `main()` you set up an integer variable, `number`, with an initial value of 10. You also have a second variable, `result`, which you use to store the value that you get back from the function `change()`:

```
  int number = 10;                    /* Starting Value                  */
  int result = 0;                     /* Place to put the returned value */
```

You then call the function `change()` and pass the value of the variable `number` to it:

```
  result = change(number);
```

The function `change()` is defined as follows:

```
int change(int number)
{
  number = 2 * number;
  printf("\nIn function change, number = %d", number);
  return number;
}
```

Within the body of the function `change()`, the first statement doubles the value stored in the argument that has been passed to it from `main()`. You even use the same variable name in the function that you used in `main()` to reinforce the idea that you want to change the original value. The function `change()` displays the new value of `number` before it returns it to `main()` by means of the `return` statement.