CHAPTER 9

More on Functions

Now that you've completed Chapter 8, you have a good grounding in the essentials of creating and using functions. In this chapter you'll build on that foundation by exploring how functions can be used and manipulated; in particular, you'll investigate how you can access a function through a pointer. You'll also be working with some more flexible methods of communicating between functions.

In this chapter you'll learn the following:

- What pointers to functions are and how you use them
- How to use static variables in functions
- How to share variables between functions
- How functions can call themselves without resulting in an indefinite loop
- How to write an Othello-type game (also known as Reversi)

Pointers to Functions

Up to now, you've considered pointers as an exceptionally useful device for manipulating data and variables that contain data. It's a bit like handling things with a pair of tongs. You can manipulate a whole range of hot items with just one pair of tongs. However, you can also use pointers to handle *functions* at a distance. Because a function has an address in memory where it starts execution (i.e., its starting address), the basic information to be stored in a pointer to a function is going to be that address.

If you think about it, though, this isn't going to be enough. If a function is going to be called through a pointer, information also has to be available about the number and type of the arguments to be supplied, and the type of return value to be expected. The compiler can't deduce these just from the address of the function. This means that declaring a pointer to a function is going to be a little more complicated than declaring a pointer to a data type. Just as a pointer holds an address and must also define a type, a function pointer holds an address and must also define a prototype.

Declaring a Pointer to a Function

The declaration for a pointer to a function looks a little strange and can be confusing, so let's start with a simple example:

```
int (*pfunction) (int);
```

This declares a pointer to a function. It doesn't point to anything—yet; this statement just defines the pointer variable. The name of the pointer is `pfunction`, and it's intended to point to functions that have one parameter of type `int` and that return a value of type `int` to the calling program. Furthermore, you can *only* use this particular pointer to point to functions with these characteristics. If you want a pointer to functions that accept a `float` argument and return `float` values, you need to declare another pointer with the required characteristics. The components of the declaration are illustrated in Figure 9-1.

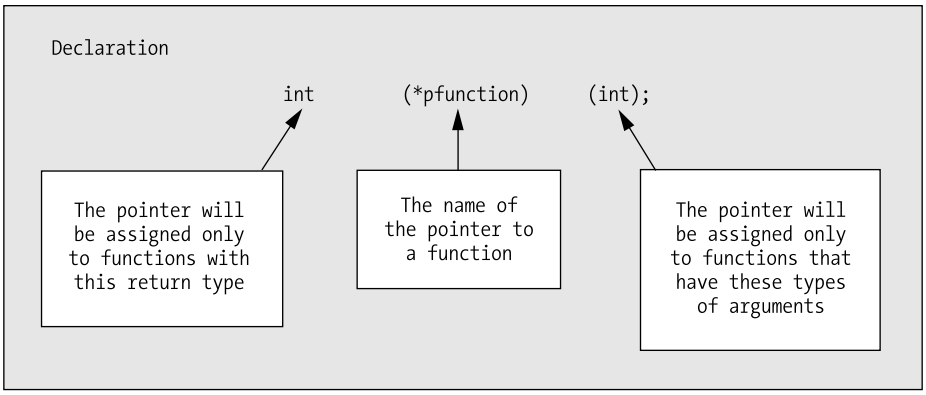

Figure 9-1. *Declaring a pointer to a function*

There are a lot of parentheses in a "pointer to function" declaration. In this example, the `*pfunction` part of the declaration must be between parentheses.

■ **Note** If you omit the parentheses, you'll have a declaration for a function called `pfunction()` that returns a value that's a pointer to `int`, which isn't what you want.

With "pointer to function" declarations, you must always put the * plus the pointer name between parentheses. The second pair of parentheses just encloses the parameter list in the same way as it does with a standard function declaration.

Because a pointer to a function can point only to functions with a given return type and a given number of parameters of given types, the variation in what it can point to is just the function name.

Calling a Function Through a Function Pointer

Suppose that you define a function that has the following prototype:

```
int sum(int a, int b);                /* Calculates a+b */
```

This function has two parameters of type `int` and returns a value of type `int` so you could store its address in a function pointer that you declare like this:

```
int (*pfun)(int, int) = sum;
```

This declares a function pointer with the name `pfun` that will store addresses of functions with two parameters of type `int` and a return value of type `int`. The statement also initializes `pfun` with the

address of the function sum(). To supply an initial value you just use the name of a function that has the required prototype.

You can now call sum() through the function pointer like this:

```
int result = pfun(45, 55);
```

This statement calls the sum() function through the pfun pointer with argument values of 45 and 55. You use the value returned by sum() as the initial value for the variable result so result will be 100. Note that you just use the function pointer name just like a function name to call the function that it points to; no dereference operator is required.

Suppose you define another function that has the following prototype:

```
int product(int a, int b);              /* Calculates a*b */
```

You can store the address of product() in pfun with the following statement:

```
pfun = product;
```

With pfun containing the address of product(), you can call product through the pointer

```
result = pfun(5, 12);
```

After executing this statement, result will contain the value 60.

Let's try a simple example and see how it works.

TRY IT OUT: USING POINTERS TO FUNCTIONS

In this example, you'll define three functions that have the same parameter and return types and use a pointer to a function to call each of them in turn.

```
/* Program 9.1 Pointing to functions */
#include <stdio.h>

/* Function prototypes */
int sum(int, int);
int product(int, int);
int difference(int, int);

int main(void)
{
  int a = 10;                          /* Initial value for a              */
  int b = 5;                           /* Initial value for b              */
  int result = 0;                      /* Storage for results              */
  int (*pfun)(int, int);               /* Function pointer declaration     */

  pfun = sum;                          /* Points to function sum()         */
  result = pfun(a, b);                 /* Call sum() through pointer       */
  printf("\npfun = sum            result = %d", result);

  pfun = product;                      /* Points to function product()     */
  result = pfun(a, b);                 /* Call product() through pointer   */
  printf("\npfun = product        result = %d", result);
```

```
  pfun = difference;                /* Points to function difference()   */
  result = pfun(a, b);              /* Call difference() through pointer */
  printf("\npfun = difference      result = %d\n", result);
  return 0;
}

int sum(int x, int y)
{
  return x + y;
}

int product(int x, int y)
{
  return x * y;
}

int difference(int x, int y)
{
  return x - y;
}
```

The output from this program looks like this:

```
pfun = sum             result = 15
pfun = product         result = 50
pfun = difference      result = 5
```

How It Works

You declare and define three different functions to return the sum, the product, and the difference between two integer arguments. Within `main()`, you declare a pointer to a function with this statement:

```
int (*pfun)(int, int);              /* Function pointer declaration      */
```

This pointer can be assigned to point to any function that accepts two `int` arguments and also returns a value of type `int`. Notice the way you assign a value to the pointer:

```
pfun = sum;                         /* Points to function sum()          */
```

You just use a regular assignment statement that has the name of the function, completely unadorned, on the right side! You don't need to put in the parameter list or anything. If you did, it would be wrong, because it would then be a function call, not an address, and the compiler would complain. A function is very much like an array in its usage here. If you want the address of an array, you just use the name by itself, and if you want the address of a function you also use the name by itself.

In `main()`, you assign the address of each function, in turn, to the function pointer `pfun`. You then call each function using the pointer `pfun` and display the result. You can see how to call a function using the pointer in this statement:

```
result = pfun(a, b);                /* Call sum() through pointer        */
```

You just use the name of the pointer as though it were a function name, followed by the argument list between parentheses. Here, you're using the "pointer to function" variable name as though it were the original function name, so the argument list must correspond with the parameters in the function header for the function you're calling. This is illustrated in Figure 9-2.

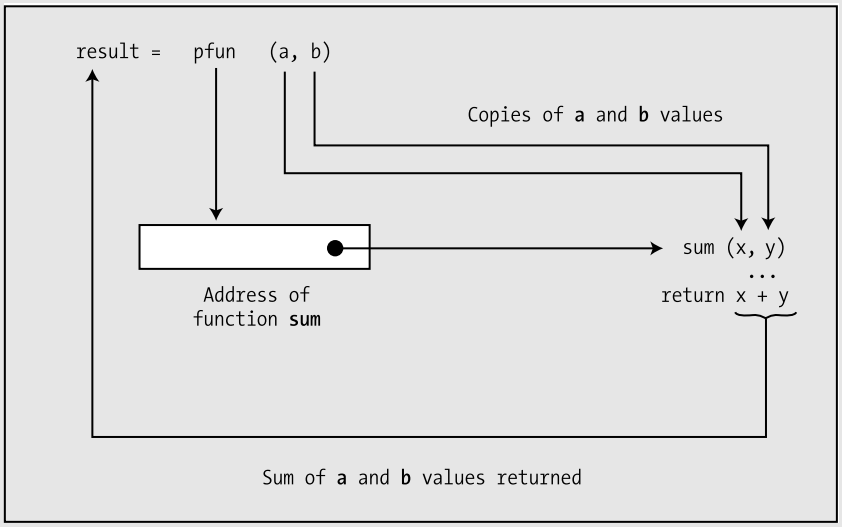

Figure 9-2. *Calling a function through a pointer*

Arrays of Pointers to Functions

Of course, a function pointer is a variable like any other. You can therefore create an array of pointers to functions.

To declare an array of function pointers, you just put the array dimension after the function pointer array name, for instance

```
int (*pfunctions[10]) (int);
```

This declares an array, `pfunctions`, with ten elements. Each element in this array can store the address of a function with a return type of `int` and two parameters of type `int`. Let's see how this would work in practice.

TRY IT OUT: ARRAYS OF POINTERS TO FUNCTIONS

You can demonstrate how you can use an array of pointers to functions with a variation on the last example:

```
/* Program 9.2 Arrays of Pointers to functions   */
#include <stdio.h>

/* Function prototypes   */
int sum(int, int);
int product(int, int);
int difference(int, int);
```

```
int main(void)
{
  int a = 10;                          /* Initial value for a               */
  int b = 5;                           /* Initial value for b               */
  int result = 0;                      /* Storage for results               */
  int (*pfun[3])(int, int);            /* Function pointer array declaration */

  /* Initialize pointers */
  pfun[0] = sum;
  pfun[1] = product;
  pfun[2] = difference;

  /* Execute each function pointed to */
  for(int i = 0 ; i < 3 ; i++)
  {
    result = pfun[i](a, b);            /* Call the function through a pointer */
    printf("\nresult = %d", result);  /* Display the result                  */
  }

  /* Call all three functions through pointers in an expression */
  result = pfun[1](pfun[0](a, b), pfun[2](a, b));
  printf("\n\nThe product of the sum and the difference = %d\n",
                                                          result);
  return 0;
}

/* Definitions of sum(), product() and difference() as before… */
```

The output from this program is as follows:

```
result = 15
result = 50
result = 5

The product of the sum and the difference = 75
```

How It Works

The major difference between this and the last example is the pointer array, which you declare as follows:

```
int (*pfun[3])(int, int);            /* Function pointer array declaration */
```

This is similar to the previous declaration for a single pointer variable, but with the addition of the array dimension in square brackets following the pointer name. If you want a two-dimensional array, two sets of square brackets would have to appear here, just like declarations for ordinary array types. You still enclose the parameter list between parentheses, as you did in the declaration of a single pointer. Again, in parallel with what happens for ordinary arrays, all the elements of the array of "pointers to functions" are of the same type and will accept only the argument list specified. So, in this example, they can all only point to functions that take two arguments of type `int` and return an `int` value.

When you want to assign a value to a pointer within the array, you write it in the same way as an element of any other array:

```
pfun[0] = sum;
```

Apart from the function name on the right of the equal sign, this could be a normal data array. It's used in exactly the same way. You could have chosen to initialize all the elements of the array of pointers within the declaration itself:

```
int (*pfun[3])(int, int) = { sum, product, difference };
```

This would have initialized all three elements and would have eliminated the need for the assignment statements that perform the initialization. In fact, you could have left out the array dimension, too, and gotten it by default:

```
int (*pfun[])(int, int) = { sum, product, difference };
```

The number of initializing values between the braces would determine the number of elements in the array. Thus an initialization list for an array of function pointers works in exactly the same way as an initialization list for any other type of array.

When it comes to calling a function that an array element points to, you write it as follows:

```
result = pfun[i](a, b);  /* Call the function through a pointer */
```

This, again, is much like the previous example, with just the addition of the index value in square brackets that follow the pointer name. You index this array with the loop variable `i` as you've done many times before with ordinary data arrays.

Look at the output. The first three lines are generated in the `for` loop, where the functions `sum()`, `product()`, and `difference()` are each called in turn through the corresponding element of the pointer array. The last line of output is produced using the value `result` from the following statement:

```
result = pfun[1](pfun[0](a, b), pfun[2](a, b));
```

This statement shows that you can incorporate function calls through pointers into expressions, in the same way that you might use a normal function call. Here, you call two of the functions through pointers, and their results are used as arguments to a third function that's called through a pointer. Because the elements of the array correspond to the functions `sum()`, `product()`, and `difference()` in sequence, this statement is equivalent to the following:

```
result = product(sum(a, b), difference(a, b));
```

The sequence of events in this statement is as follows:

1. Execute `sum(a, b)` and `difference(a, b)` and save the return values.
2. Execute the function `product()` with the returned values from step 1 as arguments, and save the value returned.
3. Store the value obtained from step 2 in the variable `result`.

Pointers to Functions As Arguments

You can also pass a pointer to a function as an argument. This allows a different function to be called, depending on which function is addressed by the pointer that's passed as the argument.

TRY IT OUT: POINTERS TO FUNCTIONS AS ARGUMENTS

You could produce a variant of the last example that will pass a pointer to a function as an argument to a function.

```
/* Program 9.3 Passing a Pointer to a function   */
#include <stdio.h>

/* Function prototypes */
int sum(int,int);
int product(int,int);
int difference(int,int);
int any_function(int(*pfun)(int, int), int x, int y);

int main(void)
{
  int a = 10;                          /* Initial value for a */
  int b = 5;                           /* Initial value for b */
  int result = 0;                      /* Storage for results */
  int (*pf)(int, int) = sum;           /* Pointer to function */

  /* Passing a pointer to a function */
  result = any_function(pf, a, b);

  printf("\nresult = %d", result );

  /* Passing the address of a function      */
  result = any_function(product,a, b);

  printf("\nresult = %d", result );

  printf("\nresult = %d\n", any_function(difference, a, b));
  return 0;
}

/* Definition of a function to call a function */
int any_function(int(*pfun)(int, int), int x, int y)
{
  return pfun(x, y);
}

/* Definition of the function sum        */
int sum(int x, int y)
{
  return x + y;
}

/* Definition of the function product    */
int product(int x, int y)
{
  return x * y;
}
```

```
/* Definition of the function difference */
int difference(int x, int y)
{
  return x - y;
}
```

The output looks like this:

```
result = 15
result = 50
result = 5
```

How It Works

The function that will accept a "pointer to a function" as an argument is `any_function()`. The prototype for this function is the following:

```
int any_function(int(*pfun)(int, int), int x, int y);
```

The function named `any_function()` has three parameters. The first parameter type is a pointer to a function that accepts two integer arguments and returns an integer. The last two parameters are integers that will be used in the call of the function specified by the first parameter. The function `any_function()` itself returns an integer value that will be the value obtained by calling the function indicated by the first argument.

Within the definition of `any_function()`, the function specified by the pointer argument is called in the `return` statement:

```
int any_function(int(*pfun)(int, int), int x, int y)
{
  return pfun(x, y);
}
```

The name of the pointer `pfun` is used, followed by the other two parameters as arguments to the function to be called. The value of `pfun` and the values of the other two parameters `x` and `y` all originate in `main()`.

Notice how you initialize the function pointer `pf` that you declared in `main()`:

```
  int (*pf)(int, int) = sum;   /* Pointer to function */
```

You place the name of the function `sum()` as the initializer after the equal sign. As you saw earlier, you can initialize function pointers to the addresses of specific functions just by putting the function name as an initializing value.

The first call to `any_function()` involves passing the value of the pointer `pf` and the values of the variables `a` and `b` to `any_function()`:

```
  result = any_function(pf, a, b);
```

The pointer is used as an argument in the usual way, and the value returned by `any_function()` is stored in the variable `result`. Because of the initial value of `pf`, the function `sum()` will be called in `any_function()`, so the returned value will be the sum of the values of `a` and `b`.

The next call to `any_function()` is in this statement:

```
  result = any_function(product,a, b);
```

Here, you explicitly enter the name of a function, `product`, as the first argument, so within `any_function()` the function `product` will be called with the values of `a` and `b` as arguments. In this case, you're effectively persuading the compiler to create an internal pointer to the function `product` and passing it to `any_function()`.

The final call of `any_function()` takes place in the argument to the `printf()` function call:

```
printf("\nresult = %d\n", any_function(difference, a, b));
```

In this case, you're also explicitly specifying the name of a function, `difference`, as an argument to `any_function()`. The compiler knows from the prototype of `any_function()` that the first argument should be a pointer to a function. Because you specify the function name, `difference`, explicitly as an argument, the compiler will generate a pointer to this function for you and pass that pointer to `any_function()`. Lastly, the value returned by `any_function()` is passed as an argument to the function `printf()`. When all this unwinds, you eventually get the difference between the values of `a` and `b` displayed.

Take care not to confuse the idea of passing an address of a function as an argument to a function, such as in this expression,

```
any_function(product, a, b)
```

with the idea of passing a value that is returned from a function, as in this statement,

```
printf("\n%d", product(a, b));
```

In the former case, you're passing the address of the function `product()` as an argument, and if and when it gets called depends on what goes on inside the body of the function `any_function()`. In the latter case, however, you're calling the function `product()` before you call `printf()` and passing the result obtained as an argument to `printf()`.

Variables in Functions

Structuring a program into functions not only simplifies the process of developing the program, but also extends the power of the language to solve problems. Carefully designed functions can often be reused making the development of new applications faster and easier. The standard library illustrates the power of reusable functions. The power of the language is further enhanced by the properties of variables within a function and some extra capabilities that C provides in declaring variables. Let's take a look at some of these now.

Static Variables: Keeping Track Within a Function

So far, all the variables you've used have gone out of scope at the end of the block in which they were defined, and their memory on the stack then becomes free for use by another function. These are called **automatic variables** because they're automatically created at the point where they're declared, and they're automatically destroyed when program execution leaves the block in which they were declared. This is a very efficient process because the memory containing data in a function is only retained for as long as you're executing statements within the function in which the variable is declared.

However, there are some circumstances in which you might want to retain information from one function call to the next within a program. You may wish to maintain a count of something within a function, such as the number of times the function has been called or the number of lines of output that have been written. With just automatic variables you have no way of doing this.

However, C does provide you with a way to do this with **static variables**. You could declare a `static` variable `count`, for example, with this declaration:

```
static int count = 0;
```

The word `static` in this statement is a keyword in C. The variable declared in this statement differs from an automatic variable in two ways. First of all, despite the fact that it may be defined within the scope of a function, this `static` variable doesn't get destroyed when execution leaves the function. Second, whereas an automatic variable is initialized each time its scope is entered, the initialization of a variable declared as `static` occurs only once, right at the beginning of the program. Although a static variable is visible only within the function that contains its declaration, it is essentially a global variable and therefore treated in the same way.

■ **Note** You can make any type of variable within a function a `static` variable.

TRY IT OUT: USING STATIC VARIABLES

You can see static variables in action in the following very simple example:

```
/* Program 9.4 Static versus automatic variables */
#include <stdio.h>

/* Function prototypes */
void test1(void);
void test2(void);

int main(void)
{
  for(int i = 0; i < 5; i++ )
  {
    test1();
    test2();
  }
  return 0;
}

/* Function test1 with an automatic variable */
void test1(void)
{
  int count = 0;
  printf("\ntest1   count = %d ", ++count );
}

/* Function test2 with a static variable */
void test2(void)
{
  static int count = 0;
  printf("\ntest2   count = %d ", ++count );
}
```

This produces the following output:

```
test1   count = 1
test2   count = 1
test1   count = 1
test2   count = 2
test1   count = 1
test2   count = 3
test1   count = 1
test2   count = 4
test1   count = 1
test2   count = 5
```

How It Works

As you can see, the two variables called `count` are quite separate. The changes in the values of each show clearly that they're independent of one another. The `static` variable `count` is declared in the function `test2()`:

```
static int count = 0;
```

Although you specify an initial value, here the variable would have been initialized to 0 anyway because you declared it as static.

Note All static variables are initialized to 0 unless you initialize them with some other value.

The `static` variable `count` is used to count the number of times the function is called. This is initialized when program execution starts, and its current value when the function is exited is maintained. It isn't reinitialized on subsequent calls to the function. Because the variable has been declared as `static`, the compiler arranges things so that the variable will be initialized only once. Because initialization occurs before program startup, you can always be sure a static variable has been initialized when you use it.

The automatic variable `count` in the function `test1()` is declared as follows:

```
int count = 0;
```

Because this is an automatic variable, it isn't initialized by default, and if you don't specify an initial value, it will contain a junk value. This variable gets reinitialized to 0 at each entry to the function, and it's discarded on exit from `test1()`; therefore, it never reaches a value higher than 1.

Although a `static` variable will persist for as long as the program is running, it will be visible only within the scope in which it is declared, and it can't be referenced outside of that original scope.

Sharing Variables Between Functions

You also have a way of sharing variables between all your functions. In the same way that you can declare constants at the beginning of a program file, so that they're outside the scope of the functions that make up the program, you can also declare variables like this. These are called **global variables** because they're accessible anywhere. Global variables are declared in the normal way; it's the position of the declaration that's significant and determines whether they're global.

TRY IT OUT: USING GLOBAL VARIABLES

By way of a demonstration, you can modify the previous example to share the `count` variable between the functions as follows:

```
/* Program 9.5 Global variables */
#include <stdio.h>

int count = 0;                          /* Declare a global variable   */

/* Function prototypes */
void test1(void);
void test2(void);

int main(void)
{
  int count = 0;                       /* This hides the global count */

  for( ; count < 5; count++)
  {
    test1();
    test2();
  }
  return 0;
}

/* Function test1 using the global variable   */
void test1(void)
{
  printf("\ntest1   count = %d ", ++count);
}

/* Function test2 using a static variable */
void test2(void)
{
  static int count;                    /* This hides the global count */
  printf("\ntest2   count = %d ", ++count);
}
```

The output will be this:

```
test1   count = 1
test2   count = 1
test1   count = 2
test2   count = 2
test1   count = 3
test2   count = 3
test1   count = 4
test2   count = 4
test1   count = 5
test2   count = 5
```

How It Works

In this example you have three separate variables called `count`. The first of these is the global variable `count` that's declared at the beginning of the file:

```
#include <stdio.h>

int count = 0;
```

This isn't a `static` variable (although you could make it `static` if you wanted to), but because it is global it will be initialized by default to 0 if you don't initialize it. It's potentially accessible in any function from the point where it's declared to the end of the file, so it's accessible in any of the functions here.

The second variable is an automatic variable `count` that's declared in `main()`:

```
  int count = 0;                        /* This hides the global count */
```

Because it has the same name as the global variable, the global variable `count` can't be accessed from `main()`. Any use of the name `count` in `main()` will refer to the variable declared within the body of `main()`. The local variable `count` *hides* the global variable.

The third variable is a static variable `count` that's declared in the function `test2()`:

```
  static int count;                     /* This hides the global count */
```

Because this is a `static` variable, it will be initialized to 0 by default. This variable also hides the global variable of the same name, so only the static variable `count` is accessible in `test2()`.

The function `test1()` works using the global `count`. The functions `main()` and `test2()` use their local versions of `count`, because the local declaration hides the global variable of the same name.

Clearly, the `count` variable in `main()` is incremented from 0 to 4, because you have five calls to each of the functions `test1()` and `test2()`. This has to be different from the `count` variables in either of the called functions; otherwise, they couldn't have the values 1 to 5 that are displayed in the output.

You can further demonstrate that this is indeed the case by simply removing the declaration for `count` in `test2()` as a `static` variable. You'll then have made `test1()` and `test2()` share the global `count`, and the values displayed will run from 1 to 10. If you then put a declaration back in `test2()` for `count` as an initialized automatic variable with the statement

```
  int count = 0;
```

the output from `test1()` will run from 1 to 5, and the output from `test2()` will remain at 1, because the variable is now automatic and will be reinitialized on each entry to the function.

Global variables can replace the need for function arguments and return values. They look very tempting as the complete alternative to automatic variables. However, you should use global variables sparingly. They can be a major asset in simplifying and shortening some programs, but using them excessively will make your programs prone to errors. It's very easy to modify a global variable and forget what consequences it might have throughout your program. The bigger the program, the more difficult it becomes to avoid erroneous references to global variables. The use of local variables provides very effective insulation for each function against the possibility of interference from the activities of other functions. You could try removing the local variable `count` from `main()` in Program 9.5 to see the effect of such an oversight on the output.

■ **Caution** As a rule, it's unwise to use the same names in C for local and global variables. There's no particular advantage to be gained, other than to demonstrate the effect, as I've done in the example.

Functions That Call Themselves: Recursion

It's possible for a function to call itself. This is termed **recursion**. You're unlikely to come across a need for recursion very often, so I won't dwell on it, but it can be a very effective technique in some contexts, providing considerable simplification of the code needed to solve particular problems. There are a few bad jokes based on the notion of recursion, but we won't dwell on those either.

Obviously, when a function calls itself there's the immediate problem of how the process stops. Here's a trivial example of a function that obviously results in an indefinite loop:

```
void Looper(void)
{
  printf("\nLooper function called.");
  Looper();                          /* Recursive call to Looper() */
}
```

Calling this function would result in an indefinite number of lines of output because after executing the `printf()` call, the function calls itself. There is no mechanism in the code that will stop the process.

This is similar to the problem you have with an indefinite loop, and the answer is similar too: a function that calls itself must also contain the means of stopping the process. Let's see how it works in practice.

TRY IT OUT: RECURSION

The primary uses of recursion tend to arise in complicated problems, so it's hard to come up with original but simple examples to show how it works. Therefore, I'll follow the crowd and use the standard illustration: the calculation of the **factorial** of an integer. A factorial of any integer is the product of all the integers from 1 up to the integer itself. So here you go:

```
/* Program 9.6 Calculating factorials using recursion */
#include <stdio.h>

unsigned long factorial(unsigned long);

int main(void)
{
 unsigned long number = 0L;
  printf("\nEnter an integer value: ");
  scanf(" %lu", &number);
  printf("\nThe factorial of %lu is %lu\n", number, factorial(number));
  return 0;
}

/* Our recursive factorial function */
unsigned long factorial(unsigned long n)
{
  if(n < 2L)
    return n;

  return n*factorial(n - 1);
}
```

Typical output from the program looks like this:

```
Enter an integer value: 4

The factorial of 4 is 24
```

How It Works

This is very simple once you get your mind around what's happening. Let's go through a concrete example of how it works. Assume you enter the value 4. Figure 9-3 shows the sequence of events.

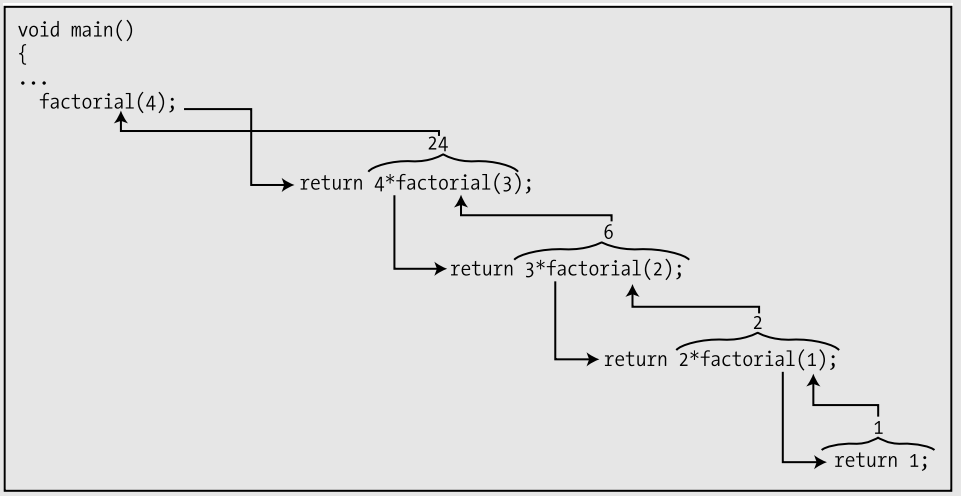

Figure 9-3. *Recursive function calls*

Within the statement

```
printf("\nThe factorial of %lu is %lu", number, factorial(number));
```

the function `factorial()` gets called from `main()` with `number` having the value 4 as the argument.

Within the `factorial()` function itself, because the argument is greater than 1, the statement executed is

```
return n*factorial(n - 1L);
```

This is the second `return` statement in the function, and it calls `factorial()` again with the argument value 3 from within the arithmetic expression. This expression can't be evaluated, and the `return` can't be completed until the value is returned from this call to the function `factorial()` with the argument 3.

This continues, as shown in Figure 9-3, until the argument in the last call of the `factorial()` function is 1. In this case, the first `return` statement

```
return n;
```

is executed and the value 1 is returned to the previous call point. This call point is, in fact, inside the second `return` in the `factorial()` function, which can now calculate `2 * 1` and return to the previous call.

In this way, the whole process unwinds, ending up with the value required being returned to `main()` where it's displayed. So for any given number `n`, you'll have `n` calls to the function `factorial()`. For each call, a copy of the argument is created, and the location to be returned to is stored. This can get expensive as far as memory is concerned if there are many levels of recursion. A loop to do the same thing would be cheaper and faster. If you do need or want to use recursion, the most important thing to remember is that there has to be a way to end the process. In other words, there must be a mechanism for *not* repeating the recursive call. In this example the check for whether the argument is 1 provides the way for the sequence of recursive calls of the `factorial()` function to end.

Note that factorial values grow very quickly. With quite modest input values, you'll exceed the capacity of an `unsigned long` integer and start getting the wrong results.

Functions with a Variable Number of Arguments

It can't have escaped your notice that some functions in the standard libraries accept a variable number of arguments. The functions `printf()` and `scanf()` are obvious examples. You may come up with a need to do this yourself from time to time, so the standard library `<stdarg.h>` provides you with routines to write some of your own.

The immediately obvious problem with writing a function with a variable number of parameters is how to specify its prototype. Suppose you're going to produce a function to calculate the average of two or more values of type `double`. Clearly, calculating the average of fewer than two values wouldn't make much sense. The prototype would be written as follows:

```
double average(double v1, double v2, ...);
```

The **ellipsis** (that's the fancy name for the three periods after the second parameter type) indicates that a variable number of arguments may follow the first two fixed arguments. You must have at least one fixed argument. The remaining specifications are as you would usually find with a function prototype. The first two arguments are of type `double`, and the function returns a `double` result.

The second problem with variable argument lists that hits you between the eyes next is how you reference the arguments when writing the function. Because you don't know how many there are, you can't possibly give them names. The only conceivable way to do this is indirectly, through pointers. The `<stdarg.h>` library header provides you with routines that are usually implemented as macros to help with this, but they look and operate like functions, so I'll discuss them as though they were. You need to use three of these when implementing your own function with a variable number of arguments. They are called `va_start()`, `va_arg()`, and `va_end()`. The first of these has the following form:

```
void va_start(va_list parg, last_fixed_arg);
```

The name, `va_start`, is obtained from **variable argument start**. This function accepts two arguments: a pointer `parg` of type `va_list`, and the name of the last fixed parameter that you specified for the function you're writing. The `va_list` type is a type that is also defined in `<stdarg.h>` and is designed to store information required by the routines provided that support variable argument lists.

So using the function `average()` as an illustration, you can start to write the function as follows:

```
double average(double v1, double v2,...)
{
  va_list parg;              /* Pointer for variable argument list */
  /* More code to go here... */
  va_start( parg, v2);
  /* More code to go her. . . */
}
```

You first declare the variable parg of type va_list. You then call va_start() with this as the first argument and specify the last fixed parameter v2 as the second argument. The effect of the call to va_start() is to set the variable parg to point to the first variable argument that is passed to the function when it is called. You still don't know what type of value this represents, and the standard library is no further help in this context. You must determine the type of each variable argument, either implicitly—all variable arguments assumed to be of a given type, for instance—or by deducing the type of each argument from information contained within one of the fixed arguments.

The average() function deals with arguments of type double, so the type isn't a problem. You now need to know how to access the value of each of the variable arguments, so let's see how this is done by completing the function average():

```
/* Function to calculate the average of a variable number of arguments */
double average( double v1, double v2,...)
{
  va_list parg;              /* Pointer for variable argument list */
  double sum = v1+v2;        /* Accumulate sum of the arguments    */
  double value = 0;          /* Argument value                     */
  int count = 2;             /* Count of number of arguments       */

  va_start(parg,v2);         /* Initialize argument pointer        */
  while((value = va_arg(parg, double)) != 0.0)
  {
    sum += value;
    count++;
  }
  va_end(parg);              /* End variable argument process      */
  return sum/count;
}
```

You can work your way through this step by step. After declaring parg, you declare the variable sum as double and as being initialized with the sum of the first two fixed arguments, v1 and v2. You'll accumulate the sum of all the argument values in sum. The next variable, value, declared as double will be used to store the values of the variable arguments as you obtain them one by one. You then declare a counter, count, for the total number of arguments and initialize this with the value 2 because you know you have at least that many values from the fixed arguments. After you call va_start() to initialize parg, most of the action takes place within the while loop. Look at the loop condition:

```
while((value = va_arg(parg, double)) != 0.0)
```

The loop condition calls another function from stdarg.h, va_arg(). The first argument to va_arg() is the variable parg you initialized through the call to va_start(). The second argument is a specification of the type of the argument you expect to find. The function va_arg() returns the value of the current argument specified by parg, and this is stored in value. It also updates the pointer parg to point to the next argument in the list, based on the type you specified in the call. It's essential to have some means of determining the types of the variable arguments. If you don't specify the correct type, you won't be able to obtain the next argument correctly. In this case, the function is written assuming the arguments are all double. Another assumption you're making is that all the arguments will be nonzero except for the last. This is reflected in the condition for continuing the loop, being that value isn't equal to 0. Within the loop you have familiar statements for accumulating the sum in sum and for incrementing count.

When an argument value obtained is 0, the loop ends and you execute the statement

```
  va_end(parg);              /* End variable argument process      */
```

The call to va_end() is essential to tidy up loose ends left by the process. It resets the parg point to NULL. If you omit this call, your program may not work properly. Once the tidying up is complete, you can return the required result with the statement

```
return sum/count;
```

TRY IT OUT: USING VARIABLE ARGUMENT LISTS

After you've written the function average(), it would be a good idea to exercise it in a little program to make sure it works:

```
/* Program 9.7 Calculating an average using variable argument lists */
#include <stdio.h>
#include <stdarg.h>

double average(double v1 , double v2,...);       /* Function prototype */

int main(void)
{
  double Val1 = 10.5, Val2 = 2.5;
  int num1 = 6, num2 = 5;
  long num3 = 12, num4 = 20;

  printf("\n Average = %lf", average(Val1, 3.5, Val2, 4.5, 0.0));
  printf("\n Average = %lf", average(1.0, 2.0, 0.0));
  printf("\n Average = %lf\n", average( (double)num2, Val2,(double)num1,
                                    (double)num4,(double)num3, 0.0));
  return 0;
}

/* Function to calculate the average of a variable number of arguments */
double average( double v1, double v2,...)
{
  va_list parg;              /* Pointer for variable argument list */
  double sum = v1+v2;        /* Accumulate sum of the arguments    */
  double value = 0;          /* Argument value                     */
  int count = 2;             /* Count of number of arguments       */

  va_start(parg,v2);         /* Initialize argument pointer        */

  while((value = va_arg(parg, double)) != 0.0)
  {
    sum += value;
    count++;
  }
  va_end(parg);              /* End variable argument process      */
  return sum/count;
}
```

If you compile and run this, you should get the following output:

```
Average = 5.250000
Average = 1.500000
Average = 9.100000
```

How It Works

This output is as a result of three calls to `average` with different numbers of arguments. Remember, you need to ensure that you cast the variable arguments to the type `double`, because this is the argument type assumed by the function `average()`. You can call the `average()` function with as many arguments as you like as long as the last one is 0.

You might be wondering how `printf()` manages to handle a mix of types. Well, remember the first argument is a control string with format specifiers. This supplies the information necessary to determine the types of the arguments that follow, as well as how many there are. The number of arguments following the first must match the number of format specifiers in the control string. The type of each argument after the first must match the type implied by the corresponding format specifier. You've seen how things don't work out right if you specify the wrong format for the type of variable you want to output.

Copying a va_list

It is possible that you may need to process a variable argument list more than once. The `<stdarg.h>` header file defines a routine for copying an existing `va_list` for this purpose. Suppose you have created and initialized a `va_list` object, `parg`, within a function by using `va_start()`. You can now make a copy of `parg` like this:

```
va_list parg_copy;
copy(parg_copy, parg);
```

The first statement creates a new `va_list` variable, `parg_copy`. The next statement copies the contents of `parg` to `parg_copy`. You can then process `parg` and `parg_copy` independently to extract argument values using `va_arg()` and `va_end()`.

Note that the `copy()` routine copies the `va_list` object in whatever state it's in, so if you have executed `va_arg()` with `parg` to extract argument values from the list prior to using the `copy()` routine, `parg_copy` will be in an identical state to `parg` with some argument values already extracted. Note also that you must not use the `va_list` object `parg_copy` as the destination for another copy operation before you have executed `pa_end()` for `parg_copy`.

Basic Rules for Variable-Length Argument Lists

Here's a summary of the basic rules and requirements for writing functions to be called with a variable number of arguments:

- There needs to be at least one fixed argument in a function that accepts a variable number of arguments.
- You must call `va_start()` to initialize the value of the variable argument list pointer in your function. This pointer also needs to be declared as type `va_list`.
- There needs to be a mechanism to determine the type of each argument. Either there can be a default type assumed or there can be a parameter that allows the argument type to be determined. For example, in the function `average()`, you could have an extra fixed argument that would have the value 0 if the variable arguments were `double`, and 1 if they were `long`. If the argument type specified in the call to `va_arg()` isn't correct for the argument value specified when your function is called, your function won't work properly.

- You have to arrange for there to be some way to determine when the list of arguments is exhausted. For example, the last argument in the variable argument list could have a fixed value called a **sentinel** value that can be detected because it's different from all the others, or a fixed argument could contain a count of the number of arguments in total or in the variable part of the argument list.
- The second argument to `va_arg()` that specifies the type of the argument value to be retrieved must be such that the pointer to the type can be specified by appending * to the type name. Check the documentation for your compiler for other restrictions that may apply.
- You must call `va_end()` before you exit a function with a variable number of arguments. If you fail to do so, the function won't work properly.

You could try a few variations in Program 9.7 to understand this process better. Put some output in the function `average()` and see what happens if you change a few things. For example, you could display `value` and `count` in the loop in the function `average()`. You could then modify `main()` to use an argument that isn't `double`, or you could introduce a function call in which the last argument isn't 0.

The main() Function

You already know that the `main()` function is where execution starts. What I haven't discussed up to now is that `main()` can have a parameter list so that you can pass arguments to `main()` when you execute a program from the command line. You can write the `main()` function either with no parameters or with two parameters.

When you write `main()` with parameters, the first parameter is of type `int` and represents a count of the number of arguments that appear in the command that is used to execute `main()`, including the name of the program itself. Thus, if you add two arguments following the name of the program on the command line, the value of the first argument to `main()` will be 3. The second parameter to `main()` is an array of pointers to strings. The argument that will be passed when you write two arguments following the name of the program at the command line will be an array of three pointers. The first will point to the name of the program, and the second and third will point to the two arguments that you enter at the command line.

```
/* Program 9.8 A program to list the command line arguments */
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Program name: %s\n", argv[0]);
  for(int i = 1 ; i<argc ; i++)
    printf("\nArgument %d: %s", i, argv[i]);
  return 0;
}
```

The value of `argc` must be at least 1 because you can't execute a program without entering the program name. You therefore output `argv[0]` as the program name. Subsequent elements in the `argv` array will be the arguments that were entered at the command line, so you output these in sequence within the `for` loop.

My source file for this program had the name `Program9_08.c` so I entered the following command to execute it :

```
Program9_08   first   second_arg   "Third is this"
```

Note the use of double quotes to enclose an argument that includes spaces. This is because spaces are normally treated as delimiters. You can always enclose an argument between double quotes to ensure it will be treated as a single argument.

The program then produces the following output as a result of the preceding command:

```
Program name: Program9_08

Argument 1: first
Argument 2: second_arg
Argument 3: Third is this
```

As you can see, putting double quotes around the last argument ensures that it is read as a single argument and not as three arguments.

All command-line arguments will be read as strings, so when numerical values are entered at the command line, you'll need to convert the string containing the value to the appropriate numerical type. You can use one of the functions shown in Table 9-1 that are declared in `<stdlib.h>` to do this.

Table 9-1. *Functions That Convert Strings to Numerical Values*

Function	Description
`atof()`	Converts the string passed as an argument to type `double`
`atoi()`	Converts the string passed as an argument to type `int`
`atof()`	Converts the string passed as an argument to type `long`

For example, if you're expecting a command-line argument to be an integer, you might process it like this:

```
int arg_value = 0;           /* Stores value of command line argument */
if(argc>1)                   /* Verify we have at least one argument  */
  arg_value = atoi(argv[1]);
else
{
  printf("Command line argument missing.");
  return 1;
}
```

Note the check on the number of arguments. It's particularly important to include this before processing command-line arguments, as it's very easy to forget to enter them.

Ending a Program

There are several ways of ending a program. Falling off the end of the body of `main()` is equivalent to executing a `return` statement in `main()`, which will end the program. There are two standard library functions that you can call to end a program, both of which are declared in the `<stdlib.h>` header. Calling the `abort()` function terminates the program immediately and represents an abnormal end to program operations, so you shouldn't use this for a normal end to a program. You would call `abort()` to end a program like this: