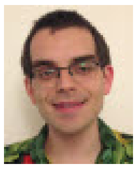
Lesson 4: Functions in C


By Alex Allain

Now that you should have learned about variables, loops, and conditional statements it is time to learn about functions. You should have an idea of their uses as we have already used them and defined one in the guise of main. Getchar is another example of a function. In general, functions are blocks of code that perform a number of pre-defined commands to accomplish something productive. You can either use the built-in library functions or you can create your own functions.

Functions that a programmer writes will generally require a prototype. Just like a blueprint, the prototype gives basic structural information: it tells the compiler what the function will return, what the function will be called, as well as what arguments the function can be passed. When I say that the function returns a value, I mean that the function can be used in the same manner as a variable would be. For example, a variable can be set equal to a function that returns a value between zero and four.

For example:

```
#include <stdlib.h>   /* Include rand() */

int a = rand(); /* rand is a standard function that all compilers have */
```

Do not think that 'a' will change at random, it will be set to the value returned when the function is called, but it will not change again.

The general format for a prototype is simple:

```
return-type function_name ( arg_type arg1, ..., arg_type argN );
```

arg_type just means the type for each argument -- for instance, an int, a float, or a char. It's exactly the same thing as what you would put if you were declaring a variable.

There can be more than one argument passed to a function or none at all (where the parentheses are empty), and it does not have to return a value. Functions that do not return values have a return type of void. Let's look at a function prototype:

```
int mult ( int x, int y );
```

This prototype specifies that the function mult will accept two arguments, both integers, and that it will return an integer. Do not forget the trailing semi-colon. Without it, the compiler will probably think that you are trying to write the actual definition of the function.

When the programmer actually defines the function, it will begin with the prototype, minus the semi-colon. Then there should always be a block (surrounded by curly braces) with the code that the function is to execute, just as you would write it for the main function. Any of the arguments passed to the function can be used as if they were declared in the block. Finally, end it all with a cherry and a closing brace. Okay, maybe not a cherry.

Let's look at an example program:

```
#include <stdio.h>

int mult ( int x, int y );

int main()
{
  int x;
  int y;

  printf( "Please input two numbers to be multiplied: " );
  scanf( "%d", &x );
  scanf( "%d", &y );
  printf( "The product of your two numbers is %d\n", mult( x, y ) );
  getchar();
}
```

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

This program begins with the only necessary include file. Next is the prototype of the function. Notice that it has the final semi-colon! The main function returns an integer, which you should always have to conform to the standard. You should not have trouble understanding the input and output functions if you've followed the previous tutorials.

Notice how printf actually takes the value of what appears to be the mult function. What is really happening is printf is accepting the value returned by mult, not mult itself. The result would be the same as if we had use this print instead

```
printf( "The product of your two numbers is %d\n", x * y );
```

The mult function is actually defined below main. Because its prototype is above main, the compiler still recognizes it as being declared, and so the compiler will not give an error about mult being undeclared. As long as the prototype is present, a function can be used even if there is no definition. However, the code cannot be run without a definition even though it will compile.

Prototypes are declarations of the function, but they are only necessary to alert the compiler about the existence of a function if we don't want to go ahead and fully define the function. If mult were defined before it is used, we could do away with the prototype--the definition basically acts as a prototype as well.

Return is the keyword used to force the function to return a value. Note that it is possible to have a function that returns no value. If a function returns void, the return statement is valid, but only if it does not have an expression. In other words, for a function that returns void, the statement "return;" is legal, but usually redundant. (It can be used to exit the function before the end of the function.)

The most important functional (pun semi-intended) question is why do we need a function? Functions have many uses. For example, a programmer may have a block of code that he has repeated forty times throughout the program. A function to execute that code would save a great deal of space, and it would also make the program more readable. Also, having only one copy of the code makes it easier to make changes. Would you rather make forty little changes scattered all throughout a potentially large program, or one change to the function body? So would I.

Another reason for functions is to break down a complex program into logical parts. For example, take a menu program that runs complex code when a menu choice is selected. The program would probably best be served by making functions for each of the actual menu choices, and then breaking down the complex tasks into smaller, more manageable tasks, which could be in their own functions. In this way, a program can be designed that makes sense when read. And has a structure that is easier to understand quickly. The worst programs usually only have the required function, main, and fill it with pages of jumbled code.

Still not getting it? Ask an expert!
Quiz yourself
Previous: Loops
Next: Switch/case
Back to C Tutorial Index