CS102: Functions

A **function** is named, defined sequence of statements that performs a specific task. Once a function is written, it can be called from elsewhere in a program. Most functions accept one or more **parameters** (values passed into variables) as input, and many functions will return a **return value** to its caller.

We've already seen several examples of functions that are provided for programmers in libraries, such as "printf" and "scanf". Now, we are going to cover how a programmer creates new functions. A function that is written by the programmer is called a **user-defined function**.

Like variables, functions can be declared and defined. A function declaration is done using a **prototype statement**. The prototype statement lists the return type of the function, the name of the function, and then the types of parameters that the function accepts separated by commas and enclosed in parenthesis. A prototype statement must be followed by a semicolon.

The first function we will examine is a function that accepts two integers, the second of which is assumed to be positive. The function will compute the first integer raised to the power of the second, and return the result to the caller. The function can be declared with the following prototype statement:

```
int power(int, int);
```

It is also allowable to give "suggested" variable names for each parameter. For example:

```
int power(int base, int power);
```

However, the actual variable names that the function uses need not match the ones in the function prototype.

If a function does not take any parameters, you can include the word "void" between parenthesis or include nothing between the parenthesis. If a function does not return any value, you must use the word "void" as its return type.

A **function definition** consists of a **function header** and a **function body**. The function header is similar to a prototype statement. The general structure of a function header is:

```
return_type function_name (parameter list)
```

The parameter list consists of the variable types and names that will be used to store the values passed to the function. The variable types must match those stated in the prototype statement for the function or else the code will not compile. A function header does NOT end in a semicolon.

The function body is a compound statement that follows the function header. Like any compound statement, it is enclosed in curly braces and consists of (optional) local declarations followed by statements.

Here is a complete program that defines and uses the function called "power".

```
#include <stdio.h>

int power(int, int);

int main(void)
{
  int x;

  x = power(2, 5);
  printf("2 to the power of 5 = %d.\n", x);
  printf("-3 to the power of 4 = %d.\n", power(-3, 4));
  printf("10 to the power of 3 = %d.\n", power(10, 3));

  return 0;
}

int power(int base, int exponent)
{
  int x;

  for (x = 1; exponent > 0; exponent--)
    x = x * base;

  return x;
}
```

This program will print to standard output:

```
2 to the power of 5 = 32.
-3 to the power of 4 = 81.
10 to the power of 3 = 1000.
```

It is customary to list the prototype statements for all functions (other than "main" which doesn't need one), at the top of the program under the #include and any #define statements. In fact, in order to call a function, the prototype statement for that function, or the function definition itself, must appear above the call to the function. Actually, if the code for a function appears above all calls to that function, you don't need to have a prototype statement for that function, and leaving out prototype statements when they are not needed is common in practice. For example, in the above code, if you listed the code for the function "power" above the code for "main", you would not really need the prototype statement declaring function "power". However, it is recommendable for beginners to always include them.

You are now ready to understand the line "#include <stdio.h>" a little better! What this is actually doing is taking a file called "stdio.h" that contains a list of prototype statements declaring the functions in the standard input and standard output library, and it is including these prototype statements at the top of your program. This is necessary for you to call these functions after linking to the appropriate library.

Now, let's talk about the parameters of "power". We have named them "base" and "exponent". They are both integers, matching the prototype statement for the function.

The function starts off assigning "x" to 1. We continue to loop as long as "exponent" is greater than 0, and for each iteration of the "for" loop, we multiply "x" by "base". For example, let's say we pass in 2 for the "base" and 5 for the "exponent". The loop will iterate 5 times, and each time, "x", which starts as 1, will be multiplied by 2. Therefore, after the "for" loop ends, "x" will have the value of 2^5 which equals 32. The "return" statement will return this value of "x". When a "return" statement is executed, regardless of whether or not it is located at the end of the function, the function ends and control is returned to the caller.

Notice that this program has two variables named "x"! One is declared and used in the function "main", and the other is declared and used in the function "power". When we went over compound statements, we defined the notion of **scope**. The scope of a variable is the portion of the code in which that variable can be used. When a function has a local declaration, the declared variables are referred to as **local variables**, or locals, and the scope of such a variable is the function in which it is declared. So when we refer to "x" in "power" we are referring to one variable, and when we refer to "x" in "main" we are referring to another variable!

Now let's look at the calls to "power". The values passed to a function are referred to as **arguments**. The parameters of the called function can be thought of as declared local variables that get initialized with the values of the passed arguments. Some sources use the terms formal parameters and actual parameters instead of parameters and arguments, respectively.

The first call to "power" in the above program passes in the values of 2 and 5 as arguments. The value 32 is returned by the function, and this value is assigned to "x"; that is, to the local variable "x" declared in "main". This value will be printed by the first call to "printf".

Now let's look at the other two calls. In each case, two arguments are passed to "power", and an integer is returned. Rather than assigning these returned values to a variable, they are getting passed as arguments to "printf"! Remember, the arguments to "printf", and to any function for that matter, can be expressions, and expressions can contain calls to functions if those functions return a numeric value. The following is a valid statement:

```
x = power(power(2, 5), 2);
```

The inner call to "power" returns 32, which is then used as an argument for the outer call to "power". This call to "power" passes 32 and 2 as arguments, and "power" will return the value 1024 which gets assigned to "x".

Also, even when a function does return a value, it is possible to ignore it (i.e., to not do anything with it). With the "power" function, if we ignore its return value, the function becomes useless, since nothing is printed or changed within the function. However, we have seen many examples of "scanf", which does return the number of variables given values, in which we have ignored the return value and still made use of the function.

Of course, variables can be used as arguments to functions, or as part of expressions whose values get passed to functions. We have already seen examples of this with "printf".

It is also possible to have a return statement return a value that depends on a function call. For instance, let's say we wanted to write another function specifically for computing powers of 2 because we think this will be very common. If the function "power" above is already written, we could write a new function as follows:

```
int power_of_two(int exponent)
{
     return power(2, exponent);
}
```

The prototype statement for this function would be:

```
int power_of_two(int);
```

When we looked at "if" statements and loops, if a single simple statement followed the "if" or loop, we had the option of not making it part of a compound statement. With functions, however, the function header must be followed by a compound statement (enclosed in curly braces) even if it consists of just one statement like "power_of_two".

If a function is going to be called from only one other function, it is allowable to include the prototype statement for the called function along with the local declarations of the calling function. For example, in the above code, we could have listed the prototype statement declaring the function "power" along with the local declarations of "main".

As we have it written above, the function "power" will not work correctly for negative exponents. Which is to say it will not fit the mathematical definition of power for negative exponents; it will still run and return a value. In fact, since the exponent starts less than 0, there will be no iteration of the for loop, but the initialization still takes place regardless, and "x" will remain as 1, so this value will get returned.

Now we will look at a function that takes no parameters and returns no value. We have already seen several ways to display "Hello World!" three times to the screen. We've

done it with three calls to "printf", one call to "printf" that includes the string three times, "while" loops, "do…while" loops, and "for" loops. Here's how to do it with a function:

```
#include <stdio.h>

void hello(void);

int main(void)
{
  hello();
  hello();
  hello();

  return 0;
}

void hello(void)
{
  printf("Hello World!\n");
  return;
}
```

Actually, when a function takes no parameters, the inclusion of the word "void" inside the parentheses is optional, since it is the default. When a function returns no value, however, it is required to include "void" as the function type, since the default is "int". If you are writing a function that returns an "int", technically speaking you could leave out the type, but this makes for confusing code, and you should always include it.

One thing to notice here is that even when there are no parameters, you need to include left and right parentheses after the name of the function when you call it. If you leave them out, the code will still compile, but the function never actually gets called. What happens is that C interprets a function name without parentheses as the memory address where the function is stored, and it is actually OK to have a number by itself as a statement. The statement is useless, but valid.

Also, note that when a function doesn't return a value, the "return" statement is not followed by an expression, just a semicolon right away. Actually, if there is not a "return" statement at the end of a function, and execution gets to the end of the function, a "return" statement is assumed and control goes back to the caller. In practice, you will often see this for functions that don't return a value, but for beginners it a good idea to put in the "return" statement anyway.

Earlier, we sketched out the general format of a C program as follows:

```
Preprocessor Directives

Global Declarations
```

```
int main (void)
{
     Local Declarations

     Statements
}

Optionally other functions with similar structure to "main"
```

Global declarations can declare **global variables**, or globals, that can be seen by all functions. Below is a simple example that should help distinguish between global variables and local variables.

```
#include <stdio.h>

int x = 5;

void do_stuff(int);

int main(void)
{
  int y = 5;
  printf("In function main, x = %d, y = %d.\n", x, y);
  do_stuff(y);
  printf("Now in function main, x = %d, y = %d.\n", x, y);

  return 0;
}

void do_stuff(int y)
{
  x = x + 1;
  y = y + 1;
  printf("In function do_stuff, x = %d, y = %d.\n", x, y);
  return;
}
```

This program prints to standard output:

```
In function main, x = 5, y = 5.
In function do_stuff, x = 6, y = 6.
Now in function main, x = 6, y = 5.
```

Why? There is just one variable "x". It is declared with a global declaration, so all functions share it. As with other declarations, you can initialize the variable as you declare it. This is what we did here, initializing "x" to 5.

In main, we define a local variable "y". This variable can only be seen within "main". We have another variable named "y" in function "do_stuff". Do not be tricked by the fact that it has the same name; they are different variables! When we call a function, we pass only a value; for this reason, C is called a **pass-by-value** language. This value gets assigned to the parameter of the function being called.

So in main, the first time we reach the "printf", "x" will be 5 and "y" will be 5.

When we get into "do_stuff", the global variable "x" has its value increased to 6, and the parameter "y" local to function "do_stuff" has its value increased to 6. These are the values printed by the call to "printf" in function "do_stuff".

Then, we return to "main". The variable "y" that is local to "main" has not been changed! The global "x" has been changed, since it is shared by all functions. So when the last "printf" is reached, "x" and "y" have values of 6 and 5 respectively.

In fact, we had already seen an example of a global declaration before. Remember, when we list a prototype statement for a function at the top of the code (outside any function), this function can be called from anywhere. That's because the prototype statement is a global declaration of the function in this case! If the prototype statement for a function is inside another function, it is a local declaration of the function to be called.

We are now going to return again to the notion of scope. The scope is the portion of a program in which a variable can be seen. The scope of a global variable is the entire program. You can use or manipulate a global variable from any function. The scope of a local variable defined at the top of a function is that function. From within the same function, you can access the variable, but you cannot access the variable from other functions. As we pointed out with compound statements, it is also possible to declare variables at the beginning of any compound statement, in which case the scope of the variable is just that compound statement. (Modern C compilers also allow you to declare local variables in the middle of a function or other compound statement; then the scope of the variable is the point at which it is declared up until the end of the inner-most compound statement in which it is declared.)

It is possible to use the same name for a global variable and for a local variable within a function. When this happens, the function with the local variable uses the local variable when referring to that name, and other functions use the global. This can get very confusing, and you should always try to avoid it.

It should be pointed out that many programmers think that global variables should be avoided whenever possible. It is true that, in practice, some programmers, especially beginner programmers, do tend to overuse them. A program with too many global variables tends to be harder to read and harder to expand (i.e., if you want to add to the functionality of the program later). However, if used properly, making certain key constructs of a program global can simplify the code by avoiding the need to constantly pass data that is crucial to an entire program between many functions.

We've seen that when you pass the value of a typical variable to a function, a copy of that value gets assigned to the parameter. Changing the value of the parameter within the called function does not affect the value of the local variable in the calling function. Things are different when you pass arrays. What you are actually passing is the memory address of the array (this may seem more clear after we learn about pointers), and if the called function changes specific entries in the array, these entries remain changed when control gets back to the calling function. Part of the reason for designing C this way is so that you can pass huge arrays to functions without requiring enough memory to make a copy of the array. You do not need to specify the size of the array in either the prototype statement of the called function or in the function header of the called function. The caller needs only specify the name of the array. Here is an example:

```
#include <stdio.h>

void change(int []);

int main(void)
{
  int arr[3] = {1, 2, 3};

  change(arr);
  printf("Elements are %d, %d, and %d.\n", arr[0], arr[1],
arr[2]);

  return 0;
}

void change(int my_array[])
{
  my_array[0] = 10;
  my_array[2] = 20;

  return;
}
```

This program will print "Elements are 10, 2, and 20." to the screen.

There are actually several reasons why functions are useful and important!

One has to do with **code reuse**. Imagine you are writing a large program, and there is a specific task you need to do over and over again in different parts of your program. Let's say that this task requires several lines of code. You have two choices; one is to repeat these lines of code over and over again in all the appropriate places. The other is to create a function to perform the task and call it from all the appropriate places. This second option is better for two reasons. The first is that it leads to less code, which is generally more readable and means you are less likely to make a mistake. The second is that if you

later realize there is a mistake in the code (for instance, a logical mistake which leads to incorrect behavior for specific data), you only have to fix it in one place instead of everywhere.

Another reason that functions are useful deals with **code sharing**. Let's say that someone has already written a function to perform a specific task. You can include that function in your own program and use it. You only have to know what the function takes as parameters and what it returns; there is no need to understand how it works. For instance, when we use functions like "printf" from the standard input and output library, or "sqrt" from the math library, we know what to pass and we know what we will get back, but we don't know how the function has been implemented and it doesn't matter.

Yet another reason that functions are useful deals with **managing data**. If you were writing a large program with just one function, that function would need so many variables that it would be hard to keep track of them all! You may accidentally use one variable for more than one purpose and cause an error. When you have many reasonably-sized functions, each can only see its own local variables and globals (and there are generally only a few globals, if any, which store very important and commonly used information), so it is much simpler to keep track of variables and determine what code is actually doing.

One crucial reason that functions are so important deals with the notions of **top-down design** and **structured programming**. Top-down design is the act of breaking down a task into manageable parts such that each individual part is relatively simple and the parts can be combined without much difficulty to achieve the desired task. Structured programming refers to the implementation of the resulting design. Structured programming takes planning and organization, but a good design will often save much time and effort when it comes to actual implementation, and the resulting code will be more elegant and readable.