Functions

By: **Annamalai A**

A function is basically a set of tasks executed in sequence to accomplish a certain goal, or maybe like a sub-program. For example, our morning routines during a working day usually involves something like this:

- Wake up at 6 in the morning
- Brush teeth
- Take a shower
- Eat breakfast
- Walk / Cycle to college

This is a set of tasks that is always executed in sequence to accomplish a specific goal, which in this case is to get ready and go to college. So this is a function. We follow many more routines like this in our daily lives. Can you think of those?

In programming, a function is a set of statements that are executed in sequence. This block is given a certain name, and it can accessed anywhere in the program by calling it's name. Basically, the same block need not be written everywhere it is required. But it can written once as a function and it's name can be called any number of times to run those set of statements. That is the main advantage of functions!

For example, let's say we want to count till 5 and then till 10, here is how we usually do it:

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        printf("%d\t", i); // \t means tab (multiple spaces)
    }
    printf("\n");

    for (int i = 1; i <= 10; i++) {
```

```
        printf("%d\t", i);
    }
    printf("\n");
}
```

You'll soon find out how to do this using functions. Let's compare how functions are so much more better after that.

In `C`, there two kinds of functions:

- Library or built-in functions, and
- User defined functions

Built-in Functions

Built-in functions are those functions which are defined in the standard libraries of the program. For example, `printf` and `scanf` are two functions present in the library `stdio`. Note that they both are just names used to do a specific task, but they consist of many lines of code and are written somewhere else.

Note: How do identify functions? A function has a unique name followed by a paranthesis, which may or may not consist of values, variables or expressions.

The `math` Library

The `math` library consists of many useful functions that do scientific calculations.

Function	Purpose
`sqrt(x)`	Square root
`pow(x, y)`	`x` to the power of `y`
`ceil(x)`, `floor(x)`	Ceiling (Smallest integer greater than `x`) and Floor (Greatest integer lesser than `x`)
`fabs(x)`	Absolute value or modulus
`log(x)`, `log10(x)`	Logarithm (base e and base 10)
`exp(x)`	e to the power of `x`
`sin(x)`, `cos(x)`, `tan(x)`	Trigonometric functions

All math functions work with `double` precisions, meaning, they will convert inputs to type `double` and returns a value of type `double` as well.

Trigonometric functions take inputs in radians. If you have an input in degrees, convert it to radians using the formula:

$$\text{radians} = \text{degrees} \cdot \frac{\pi}{180}$$

User Defined Functions

These are functions defined and written by the programmers. Let's see how to write functions and also, write our first function in this language!

There are three important parts of a function:

- Declaration — Define the name of function and inputs required
- Definition — Define what the function does, and
- Calling — Use the function name and provide necassery inputs to execute the set of statements.

Declaration

Now, let's dive into how function declaration is done. Declaring a function is done in one line. While declaring, it is important to define the **return type**, **function name** and the **list of inputs** or **parameters**. The structure for declaring functions look like this:

```
return_type function_name(type parameter_1, type parameter_2, ..., type pa
rameter_n) {
    // Code
}
```

For example, consider a function that adds two integers and returns the result. The function declaration looks like this:

```
int add(int a, int b) {
    // Code
}
```

Here, the return type is `int`, meaning the functions returns an integer value to the compiler after executing the code inside. The name of the function is `add` and it accepts two inputs `a` and `b` of type `int`.

Definition

Cool! So now we know how to declare a new function! Now let's see how to define what the function does. The set of statements to be executed is written inside the block. In the above code, it is where `// Code` is present. In our case, we want to add two numbers. So it looks something like this:

```
int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

The first statement creates a variable `sum` of type `int` and stores the value of `a + b`. The second statement *returns* the sum to the compiler.

Now, what is the `return` statement? It simply returns or tells the compiler what is return after it. It could be just a value, a variable, or an expression. In case of a vairable, it tells the compiler what value is present in the vairable, and in case of an expression, it tells the compiler the result of the expression.

Also, the compiler exits the function upon recieving the value from `return`. Hence it must be placed at the end of functions or inside decision blocks where termination might be required. Any code after the `return` statement in the same block becomes *unreachable* and will never be executed.

Cool! Now try writing the same code without using the variable `sum`.

Calling

This part is where we pass necassery inputs and call the function to execute a block of code. It is simple! Calling a function follows the following structure:

```
function_name(argument_1, argument_2, ..., argument_n);
```

In our example, we could call our function like this:

```
add(5, 6);
```

Not that the two arguments passed are of type `int`. Also, the number arguments passed must be equal to the number of parameters present.

Prototyping

There are two ways to declare and define functions, one is this way, and the other is something called **prototyping**. It looks something like this:

```
int add(int a, int b);

// Some code

int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

Here, we have first declared the function, and not defined what the function is supposed to do. Basically, we have created a *prototype* of the function. Later in the code we have written the same declaration again and defined what the function does.

Now you might be wondering “Hey, why would I want to define functions like this? I have to write the declaration twice. Isn’t the other way better anyway?” And it makes sense.

For this, let’s see how the compiler actually goes through our code. For that purpose, let’s look the complete program:

```
#include <stdio.h>

int add(int a, int b) {
    int sum = a + b;
    return sum;
}

int main() {
    printf("%d", add(5, 6));
```

```
    return 0;
}
```

The compiler first reads the name of the function and what it does. By the time it comes to `main()` it will know what functions have been declared and what has been defined in it.

But the problem is, the `main()` function which is the most important one goes at the end of the program, which is not really great, because the function definitions are fixed and only written when declaring them for the first time. They are rarely modified. But the code inside `main()` will be modified each time and it will great to keep it at first.

So why not do that?

```
#include <stdio.h>

int main() {
    printf("%d", add(5, 6));
    return 0;
}

int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

Great! But the proble now is, the compiler first goes through `main()` and finds `add()` there. But it does not what `add()` is because it has not gone through it yet. Hence, it throws an error.

So we need a way so that we *declare* our functions first, write the `main()` function and then *define* it later, so that the `main()` still stays on top, and the compiler does not throw an error. This is where prototyping becomes useful. So with function prototyping our porgram looks like this:

```
#include <stdio.h>

int add(int a, int b);
```

```
int main() {
    printf("%d", add(5, 6));
    return 0;
}

int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

However, prototyping is not mandatory for all programs, especially if it small like in our case.

Techniques of Passing Arguments

There are two ways of creating parameters and passing arguments to a function. They are:

- Call by value
- Call by reference

Call by Value

The parameters created accepts values from the user of the specified type, and the passed arguments are values of some variables.

The compiler, in this case *creates* a copy of the passed variable in a different memory address and processes on the copied variable. Hence, after the function finishes processing, the variable still holds the *same* information. Try out this code using a compiler:

```
void add(int a, int b, int s) {
    s = a + b;
}

int main() {
    s = 0; // define s = 0
    add(2, 5, s); // Inside the block, s = 7
    printf("%d", s); // Prints 0 because still, s = 0
```

```
    return 1;
}
```

Call by value works great when:

- the return type is *not* `void` and we are not redefining the value of a variable inside the function.
- We do not want the original value of a variable passed into the function being affected.

Call by Reference

The parameters created accepts the address of variables from the user of the specified type, and the passed arguments are the address of some variables. It is done using referencing and dereferencing.

The compiler, in this case *uses* the passed variable itself for processiong. Hence, after the function finishes processing, the variable holds the information that is obtained after processing. Try out this code using a compiler:

```
void add(int a, int b, int *s) {
    *s = a + b;
}

int main() {
    s = 0; // define s = 0
    add(2, 5, &s); // Inside the block, s = 7
    printf("%d", s); // Prints 0 because still, s = 0

    return 1;
}
```

Call by reference works great when:

- the return type is `void` and we are not redefining the value of a variable inside the function.
- We want the original value of a variable passed into the function being affected.

Recursive Functions

This function is all about looping of function calls. A function is recursive when it calls itself within it's definition, and it is good counterpart of writing loops in functions.

```
return_type function_name(parameters) {
    // Base condition (stopping condition)
    if (termination_condition)
        return value;
    // Recursive call
    else
        return function_name(modified_parameters);
}
```

It is important to return a value to continue processing, and mention function calls inside a decision block in order to prevent infinite function calls. Also, it is important to define a condition that terminates function calls at some point, so that the loop actually breaks. Moreover, the call inside must happen with the updated value, so that the function does not repeat doing the same thing (both process and call) again and again.

Here is an example where we are finding the factorial of a number. First lets do it using loops, then recursion.

```
// Iterative
#include <stdio.h>

int factorial(int n) {
    int fact = 1;
    for (int i = 2; i <= n; i++) {
        fact *= 1;
    }
    return fact;
}

int main() {
    printf("%d", fact(5));
    return 0;
}
```

```
// Recursive
#include <stdio.h>

int factorial(int n) {
    if (n == 1 || n == 0)
        return 1
    return n * factorial(n - 1);
}

int main() {
    printf("%d", fact(5));
    return 0;
}
```

Although recursion is doing the same thing as iteration, it can only be done with functions. Unlike iteration, where we play with time complexity, in recursion we handle memory because each call and return statement is stored in the memory for being executed later. Hence, it must be carefully used so that it does not occupy a lot of memory.