FUNCTIONS: regular functions, methods and the main

Functions are just like recipes: they take ingredients (arguments) and use them to give you delicious dishes (results).

To write a function, you have to say what type of data are you getting back, the name of the function and its arguments.

```
int addToNumber ( int number, int add ) {
      /*
        * Prepare data
        * Steps and operations
      */
      return addedNumber;
}
```

```
dish fryMyEgg (egg chickenEgg, oil oliveOil) {
      /*
        * Prepare ingredients
        * Steps to preparing the dish
      */
      return friedEgg;
}
```

Regular functions are just that. You give them something and get something in return.

Methods don't return something, but rather may or may not change something in the data provided. Would be like adding a spice to a dish; you give them the spice and the dish, and then the dish has changed, but you already had it so it is not returned.

In C, the only way to distinguish a method from a function is that the returning type is ***void***.

```
void addToNumber ( int *number, int add ) {
      /*
        * Prepare data
        * Steps and operations
      */
}
```

```
nothing addSpice (dish *friedEgg, spice salt) {
      /*
        * Prepare ingredients
        * Steps to changing the dish
      */
}
```

We consider a program to have one or more functions/methods. Programs do things, that's why. You cannot make a dish without a recipe. In C, a program will always have a main.

The main is the function the computer executes when a C program is called, and it works just like a function. The integer it returns is for telling if there were any problems during execution (0 means all is OK).

```
int main(int argc, char *argv[]) {
      /* Bad things happen here,
      muahaha */
      return 0;
}
```

It has two arguments:

- **Argv** stands for **Arg**ument **V**ector. It holds all the arguments passed to this program. The first one (position 0) always holds the name of the program.
- **Argc** stands for **Arg**ument **C**ount. It stores how many elements are there in the argument vector. It is always more or equal to one, as argv always holds the name of the program in the first position.

Oh, and the most important thing: calling the functions. Functions/methods can be called from inside other function/methods!

NOTES, TIPS AND IMPORTANT THINGS TO REMEMBER:

- Functions/methods are written **OUTSIDE** other functions/methods, except in given occasions. **DON'T** write them inside the main.
- Write the functions/methods **ABOVE** the main. The computer needs to process them to know they exist before being able to use them. This is a very common error.
- Functions/methods are meant to be as generic as possible. They are meant to save you time and lines of code, and are usually used several times in one program.
- This means that if you need to change something that you wrote on your code several times, you won't have to go through it changing those same lines a bunch of times. This is what is called maintenance and modularity, and it's really important.

VARIABLES

Variables are used to store data. They are like a's and b's in basic algebra, only with bigger names and different types.

To create a variable, we need to write its type and its name.

```
type name;
```

```
int number;
```

To store something inside, all we need is to use the = symbol, just like in any maths problem.

```
number = 5;
```

And these two things can be done at the same time, like this:

```
type name = value;
```

```
int number = 5;
```

Then, variables can be used for a lot of things: adding, substracting, multiplying, dividing... Seriously, mostly anything.

Here, read this link with all the operators and things neatly explained:

http://www.tutorialspoint.com/cprogramming/c_operators.htm

Also, a small table of equivalences you can expand:

`n = n + 1;`	`n += 1;`	`n++;`
`n = n - 1;`	`n -= 1;`	`n--;`

Also:

Multiplying	*
Dividing	/
Modulus	%

Modulus is SUPER useful. It tells you what is left out of a division.

For example, number % 2 will give you a 0 if the number was even, and 1 if it is odd.

TYPES

The type says what kind of data are you working with. Just that.

Here is the complete list: https://en.wikipedia.org/wiki/C_data_types, but I'm also giving you a small table with the basic ones.

Integer (32 bits)	• int • It's a plain integer, including negative and positive values. • Value range: [-32767, 32767]
Float	• float • For real numbers and fractions.
Char	• char • Smallest integer, represents a character. • Written between '.
String	• [Not in C as a type] • Is a sequence of chars. • They are written between ". For example, in a print. • To actually work with strings in C, a char array can be used.
Boolean	• bool • Values: true or false. • Actually, it is sort of a bit, where 0 would be false and anything else would be converted to a 1, true. • They have their own set of operators (see operators in the variables section, in the link provided).

CONDITIONAL STATEMENTS

A conditional statement is meant to execute a chunk of code (a bunch of code lines) if something happens.

```
if (condition) {
	/* Stuff */
}
```

That is the simple if structure. This is the complete one:

```
if (condition) {
	/* Condition happened */

} else if (condition) {
	/* First condition didn't happen, but
second did. */

} else if (condition) {
	/* Yet another condition */

} ... {
	/* Seriously, you can do as many else
ifs as you want */

} else {
	/* No condition happened, so do this
other stuff */
}
```

Else ifs and the else part of the statement are completely optional.

The **condition** part is whatever you need to happen, and it must always return a boolean.

You can concatenate conditions using ands (&&) and ors (||), like:

```
condition && condition && condition ...
```

```
condition || condition || condition ...
```

It works like any logic operations.

- If you have an &&, both conditions must be true to execute the chunk of code.
- If you have an ||, either one can be true, but at least one must be true in order to execute.

For example:

If the number is in the range (0, 10] ...	If the number is out of the range [0, 10) ...

`if ( n > 0 && n <= 10 ) {`
` /* Stuff */`
`}`
	`if ( n < 0 || n >= 10 ) {`
	` /* Stuff */`
	`}`

LOOPS: FORs AND WHILEs

Loops are statements that repeat the same block of code a number of times. They stop once a condition is met. Loops need a **control variable** (could be more than one) that takes care of stopping the loop once it changes. There are two types of loops: **for** and **while** (the do while is just another way of writing the while loop).

Fors are loops meant to iterate through a range and always end. They are fixed. You always iterate the number of times you set for it.

C only has the **classical** for-loop: you say which is the control variable and what value does it have, what is the value it will move to, and how big are the steps it is taking.

Increasing control variable	Decreasing control variable

`int i;`
`for ( i = 0; i < N; i++ ) {`
`  /* Stuff */`
`}`
	`int i;`
	`for ( i = N-1; i >= 0; i-- ) {`
	`  /* Stuff */`
	`}`

Steps don't have to be one by one. You could use `i=i+2` instead of `i++`.

Whiles, however, are used for traversing and stopping before actually reaching the end, because a **condition** was met. The main difference with the for is that you have to take care of the control variable manually and see where it changes.

`while ( condition ) {`
`  /* Stuff */`
`}`
	`do {`
	`  /* Stuff */`
	`} while ( condition )`

`int i = 0;`
`while ( i < N ) {`
`  /* Stuff */`
`  i++;`
`}`
	`int i = 0;`
	`do {`
	`  /* Stuff */`
	`  i++;`
	`} while ( i < N )`

This is an example. The control variable can change anywhere in a while loop.

NOTES, TIPS AND IMPORTANT THINGS TO REMEMBER:

- You DON'T break loops — *any loop*. DON'T use *breaks* or *returns* inside of *any* kind of loop. It is okay for debugging purposes, but *never ever* leave them in your actual code. That's just ugly and plain bad programming for a number of reasons (whatever other people say), and there is always a better way of doing it, wether it is with a while loop or modifying the loop condition.

ARRAYS

If variables are like boxes, then arrays are like shelves full of boxes where each box has a number and a position. Arrays are used for storing related data.

Each position holds a value, and every value has the same type. So we have arrays of integers, chars or booleans, but never mixed (except in some advanced types of arrays where that is posible).

To create an array, we need to write its type and its name.

```
type name[<size>];
```

```
int numbers[10];
```

Notice that the [<size>] is what indicates that our variable is an array. Size is fixed for this type of arrays, and it can be substituted by a variable (although the compiler may ask the variable to be constant).

To store something inside, all we need is to address the position of the array and use the = symbol, just like before. Bear in mind that in C the first position is 0 and the last one is the size of the array minus one. That is: [0, size).

```
numbers[0] = -40; /* --> first position of the array */
numbers[9] = 5;   /* --> tenth and last position of the array */

numbers[4] += 100;   /* --> use an array as any other variable */
```

To traverse an array, we have to use a loop. We will use the control variable as the index of the array, and it is also possible to traverse them in reverse or in bigger steps.

Full traverse with for	Look for a sentinel (0)

`int i;`
`for ( i = 0 = 0; i < sizeof(numbers); i++)`
`{`
`  printf(numbers[i]);`
`}`
	`int i = 0;`
	`int size = sizeof(numbers)/ sizeof(int);`
	
	`while (i < size && numbers[i] != 0) {`
	`  i++;`
	`}`
	
	`if (i < size) { // We found a 0`
	`  /* Stuff */`
	`}`

Would we want to have more elements in the array than its size, then we would have to create a new array with the space we need, and copy the first array in the second one.

NOTES, TIPS AND IMPORTANT THINGS TO REMEMBER:

- `sizeof(<array>)` returns the number of bytes of the array. That's why we need to divide by the size in bytes of its type; to get how many values are stored inside.
- Careful with the range. If you try to access a position that does not exist for the array, a exception will be thrown and, unless catched, the program will not be able to continue.
- The positions you haven't accessed yet are full of garbage values. Sentinels are useful for that. You can keep track of how full the array is by using a token of your chosing (0, in the example above).

POINTERS

(WIP, continue with FILES in the next page.)

FILES

So, we want to work with files.

C opens files by having a pointer pointing to them. That can be achieved with the function fopen:

```
FILE *f = fopen("myCoolFile.txt", 'r');
```

The ***r*** stands for reading and ***w*** can be used instead for writing. There are other modes for opening a file. You can check them here: http://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm

There are several functions you can use for reading data out of a file:

fscanf(f, "%d", &number **);**	• Scans the first integer found in the file and stores it in number. • %d can be exchanged for %s or any other % we would normally use in a scanf.
fgetc(f **);**	• Returns a pointer for the first read char. • If you use the function again, it will give you the second char, then the third char, and so on until end of file is reached.
fgetline(f, size **);**	• Returns a pointer to a char that contains the first line. • Size is a *size_t* pointer (declared like *size_t *size*) that will be modified upon calling the function to contain the size of the line read. • If you use the function again, it will give you the second line, then the third line, and so on until end of file is reached.

There are more functions for reading, and very similar, like *getchar* or *getline*, that are worth a look.

For reading a file we will need to use a while-loop, like this:

```
FILE *f = fopen("myCoolFile.txt", 'r');
while ( (number = <reading function>(f)) != EOF) {
      /* Stuff */
}
```

Every line in a file ends with ***\n***, which is the character for *new line*. When the end of a file has been reached, that is, when we try to read after the last line, we get the *End of File* (EOF) value. It can be used as a constant global variable, as it is programmed to contain the value returned when the end of a file is reached. The function *feof* is also available for checking the EOF of a file, returning true if so.

To write to a file, just use:

```
f.write(<insert data here>);
```

And to close it:

```
f.close;
```

NOTES, TIPS AND IMPORTANT THINGS TO REMEMBER:

- We *always* open the file before reading/writing it and close it after we are done.
- *Always* close the file when you are done.
- When you open a file in C, you are actually opening a pipe for the data to go through. That is why you can use fscanf to read, using the file pointer instead of *stdin* (which, by the way, stands for standard input). You need to close that gate when you're done, or else bad things could happen.
- Beware of the EOF.