Programming, D1

Introduction

Programming a computer is simply telling it what to do. Understand that a computer can only do one thing, move data around and change it. From this simple behaviour arises mechanics complex enough to simulate whole universes.

When you're browsing, watching videos, playing games or whatever the fuck on your computer, the computer doesn't know that you're doing this. The computer knows that you're telling it to move this certain block of data to a certain place and change it in a certain way. Your output devices, speakers, monitors etc., interpret these data and produce a corresponding output.

Now for the fun stuff

Go to cs50.io. You'll be brought to your coding environment. Set up a GitHub account, after which it will take some time to set up. Yours truly will show you how to navigate this environment.

Some commands to note[1]:

- `touch <filename>` creates a file with name `<filename>`.
- `rm <filename>` deletes a file with name `<filename>`.
- `mkdir <name>` creates a folder (directory) with name `<name>`.
- `rm -r <name>` removes a directory with name `<name>`[2].
- `cd <name>` changes current directory into directory with name `<name>`.
- `code <filename>` opens the file named `<filename>` into the editor.

A programming language

Remember my little computer science note about what binary is and why it is? Well the problem with that is computers understand only binary, nothing other than it. Us humans don't understand binary, so there must be an intermediary[3]. In comes the programming language, where we write instructions in a language that is very close to English.

Yet, the code we write still is not binary. To convert these arbitrary, abstract instructions to something computers can understand, we must *translate* it to an *executable binary file* using special software called a *compiler*. Note that code that computers can understand is called machine code.

[1] Everything in between angled brackets, `<>`, is a placeholder and something else may be in that place.

[2] The `-r` in this command stands for "recursively", which means to do it repeatedly until it's done.

[3] Fun fact: People actually used to write binary on things called "punchcards". Back then, the field was dominated by women.

We will learn the C programming language, a fan favourite since the 1970s and a very good starting point. The compiler for C is called `gcc`. We feed to `gcc` the code we have written, and the name of the file into which we would like it to barf out the binary code the machine can read, and it obeys.

The primary command involving `gcc` which we will use is:

```
gcc <codefile> -o <outfile>
```

The above tells `gcc` to compile `<codefile>` and output the resulting machine code in `<outfile>`.

Some boilerplate

Before we start, we must first conform to some norms. Firstly, we must "include" the "standard i/o library". In other words, we must bring in input/output functionality into our program, doing which by hand is a momentous task. Fortunately, this has been done for us by some very smart people five decades ago and all this functionality was put into a file called `stdio.h` (read: standard i-o dot h). To do so we add the following line:

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

to the very top of our code file.

Next, we must tell `gcc` from where to start reading and translating the instructions we have written:

```
int main(void)
{
  // our code
}
```

`gcc` looks for this exact line: `int main(void)`, from after which it will start translating our written code.

This brings our boilerplate code file to being:

```
#include <stdio.h>

int main(void)
{
  // code
}
```

Note that, `gcc` ignores anything written after a pair of forward slashes, `//`. Such a line is called a comment. In the above, `// code` is an example.

For real this time

Creating data

To keep track of data we created and are using, we use the concept of *variables*. Every variable has a type of data it can store, a name, called its identifier, and the data being stored itself. To tell the computer that we wouldl like some data allocated for our program, we use the following line:

```
<datatype> <identifier>;
```

where `<datatype>` is the type of data being declared and `<identifier>` is the name of the variable itself. Note the use of a semicolon to denote end of line.

Output

When outputting in C, you must mention the type of data being outputted alongside the data being outputted itself. Every data type has a *format specifier* with which you can specify the data type being outputted. A similar thing happens for input. The function `printf()` is used to output a formatted variable. What functions are, we will get into later, but know that they take in some things and spit out some others depending on what you fed it. The things being fed are called *arguments*.

The `printf()` function takes in two arguments, `printf(<format>, <var>)`.