Hello.c

Writing and compiling your first C program

The title is so chosen because that is the name you might want to choose for your first program that you write in C. Yes, like I promised this time we are going to write our first C program and delve into its "analysis". Let's just see the following piece of code and try to understand how it works and start our journey from there.

```c
#include <stdio.h>
/* Main function */
int main (){
    printf("Hello World");
    return 0;
}
```

We'll start from the first line. What the heck does "`#include <stdio.h>`" mean? It is called the #include directive. Would it make sense if I told you this line tells the preprocessor to include the contents of the header file stdio.h into the compiled code of the program? Probably not. So many fancy terms, ehh? I will break it down later in this tutorial I promise. For now it will suffice to understand that the #include directive tells the compiler about what some of the following lines (line 5, in this case) are going to mean. What I mean is by using this line we are telling the compiler about what printf on the 5th line is going to do.

On the 3rd line we have a comment. Comments are useless for the compiler. They are only for the programmers. It is a good practice to write comments whenever you write a program. We'll learn about comments a bit later in this tutorial itself. And then from line 4 through 7, we have the heart of our program, the main function. The essence of the whole program lies in this block of code only. The program won't run without a main function. But what is a main function, what is its "function" and why is it so important after all? To begin with, what is a function? So many questions! I will try to answer them as soon as I can. But first things first, where do we write the program and how do we compile it? Let me answer that first.

- **Windows users** can easily write and compile a C program in IDEs like **TurboC++** or **DevC++**. **IDE or Integrated Development Environment** is a piece of software which consists of both a text-editor and a compiler so that unrelated tasks (writing the code and compiling it) can be conveniently performed. After writing the code, we have to save it with a .c extension and then run it.

- **Linux users** have to first use a **text-editor like Vim or Atom** to write the code and save it with a**.c extension**. Let's say we save it as file_name.c. Next step will be to open the **terminal** and type "`gcc file_name.c -o output`" without the quotes and press Enter. This will generate an executable called "output" which can be run by typing "`./output`" without the quotes and pressing Enter.

Having answered that question, it is now time for some definitions. I have been constantly mentioning the term "compile" all over the place, never defining it though. I will do it now but that will require some more definitions. Let's see what I mean.

High-level and Low-level or Assembly level language: High-level language is one which we humans understand. The code we wrote above is in high-level language. You can pretty much guess what is happening around. But computers are really dumb. They don't understand it. They only understand 1's and 0's. This is called machine-level language. Low-level language is one which is very closer to machine code where as high-level language is one closer to humans. So the closer a language gets to the machine the harder it becomes for humans to understand that and vice-versa.

Compiler, Interpreter and Assembler: Compilers are programs which convert high-level language into machine language which can be easily executed. Interpreters do the same. So what's the difference? The differnce is in the way they do it. Compilers take the whole program as its input at one go while interpreters do the conversion line-by-line. This makes compilers faster than interpreters. However, error detection is super easy when we are using interpreters since the execution stops at the line which has an error but in case of compiler it is difficult to debug the program (make the program free of errors) since we only get to know that there is an error but don't know where. Assemblers are special compilers which convert only low-level language into machine code. C uses a compiler.

Now let's understand the compilation process of C in further details. There are four stages of compilation of a C program:

1. **Pre-processing:** The pre-processor is a separate program invoked by the compiler as the first step of translation. It essentially does the following 3 things:
    - **Removal of comments:** All unnecessary lines (at least for the compiler) are changed into whitespaces.
    - **Macro expansion:** All macros are expanded (we'll learn about macros later, leave it for now).
    - **Header file expansion or Directive Handling:** Header files are necessary files which contain information about all the predefined functions we use (like scanf, printf, etc). In the pre-processing stage, the line `#include <stdio.h>` is replaced by the corresponding code present in **stdio.h** (which itself is a C program).

2. **Compilation:** The second stage of compilation is interestingly enough called compilation. In this step, the code is translated into assembly language which an assembler can understand.
3. **Assembly:** The assembly-code is then translated into actual machine code which the computer understands. The file at this stage is composed of binary data.
4. **Linking:** Though the target processor understands almost the whole of the file generated in step 3 but some pieces are still missing or out of order. A linker fills in these missing pieces and rearranges it in correct order (Linker sure knows how to do its job!). The linker will arrange the pieces of object code so that functions in some pieces can successfully call functions in other pieces. It will also add pieces containing the instructions for library functions (functions contained in header files) used by the program. For example, in our case, the linker puts in the oject code for printf in the required place. After all these, a successful executable code is generated.

The `main()` function

I think now you should have understood what function line 1 of our code performs and how it does so and also that line 3 has no significance except for us programmers (since comments are removed by preprocessor). The next portion is the main function. The "()" after main means main is a function. It is the entry point into your program. Whatever you want to do will be done here. So why is it special? Let's see.

1. The `main()` is just a function but is very much required. But what is a function? A function is just a block of code designed to do a specific thing. Every program of yours needs to have at least one function and that should be called `main()` and nothing else. Though you can have more than one function, but you can and need to have only one main. All that you need to do goes in the `main()`, enclosed in a pair of braces {}.
2. `main()` returns a value as in it is asked something and it replies. Who asks? The answer is your operating system. What does it ask? Let's not get into that. What `main()` replies? This one is interesting. In our case main returns an integer value, hence an "int" precedes `main()`. We are returning 0 in our line 6. A return value of 0 indicates success in most operating systems. If however `main()` fails to do its intended work (poor `main()`), we can return a non-zero number which would indicate failure. If we don't write a return statement after main, compiler automatically adds a return 0.
3. Some compilers like TurboC/C++ permit us to return nothing from main. In that case instead of int we write void. No return statement is required if we use void. Return types of functions is something with which we'll concern ourselves with later. Within main, there are just two lines of code. We already know about line 6. On line 5, we have a call to printf function, with argument '"Hello World!"'. Printf is a built-in function, that is, it knows how to do its job. Its job is to output something to the standard output or

screen. What does it output? Well, whatever we tell it to. So how do we tell it? We tell it what to output by passing it in as argument within the brackets following printf (If you don't understand arguments it is okay, we'll cover that later). Whatever we write within double inverted commas will be printed as is. We can also use format specifiers. We will play with the printf function in a coming lesson very soon. It will be used in almost all your programs and hence it is very important to know about it in great detail.

Syntax rules in C

Now that we have understood how our first C program worked, we need to know the syntax of the language in order to write any further code. Once you begin to see some more code and write yourself some code in C, you will get a hang of it. The following points will guide you:

1. Each instruction in a C program is written as a separate statement.
2. The statements are executed in the same order in which we write them, so the statements must be written in the same order in which we wish them to be executed.
3. Blank spaces between two words can be used to improve readability but no blank spaces are permitted within a variable, constant or keyword (these will be explained in due time).
4. All statements are to appear in small case. C is a case-sensitive language.
5. There is no specific rule for the position at which a statement is to be written on a given line and hence C is often called free-form language. But as a good programming practice, we must have a proper indentation formatting in our code.
6. Every C statement ends with a semi-colon ;.
7. Every block of code is written within a pair of braces {}.

Comments in C

We saw that comments are useless lines written within the code which are omitted when the program is translated into machine code. So why write it? I have said that it is a good programming practice to leave comments. Why? Though comments don't have any effect on the usual processing of the program, it helps a programmer to understand whatever they want to understand later when they revisit their old codes or see other people's code. In a long program, comments before every small section helps other people to easily understand the program and this increases the portability of the program. I will outline the rules and regulations neccessary to write comments in C:

1. Comments should be enclosed within /* */. Using this we can even write multi-line comments like

```
/* This is a  multi-line comment */
```

2. Single line comments are written after //. For example:

```
// This is a single-line comment
```

3. Since comments are only for programmers, we can type anything in here and get away with it. Normal syntax rules of C don't apply here. We can write in both lower and upper cases and even a combination of both within a comment.

With that I would like to put an end to this tutorial. Hope it helped you learn a great deal. This tutorial was yet almost theoretical with little practical knowledge about C (sadly, unlike I promised) but I guarantee you this was necessary for what we will learn next. We will carry our journey forward with some more C in coming tutorials.