**Your First C Program**

#include<stdio.h>

int main()

{  
 printf(“Hello World\n”);  
 return 0;  
}

Save the program with the name: hello.c

## #include<stdio.h>

With this line of code we **include** a file called stdio.h. (**St**andar**d** **I**nput/**O**utput **h**eader file). This file lets us use certain commands for input or output which we can use in our program. (Look at it as lines of code commands that have been written for us by someone else). For instance it has commands for input like reading input from the keyboard and output commands like printing things on the screen.

## int main()

The int is what is called the return value (in this case of the type **int**eger). Where it used for will be explained further down. Every program must have a main(). It is the starting point of every program. The round brackets are there for a reason, in a later session it will be explained, but for now it is enough to know that they have to be there.

## { …. }

The two curly brackets (one in the beginning and one at the end) are used to group all commands together. In this case all the commands between the two curly brackets belong to main(). The curly brackets are often used in the C language to group commands together. (To mark the beginning and end of a group or function.).

## printf(“Hello World\n”);

The printf is used for **print**ing things on the screen, in this case the words: **Hello World**. As you can see the data that is to be printed is put inside round brackets. The words Hello World are inside inverted commas, because they are what is called a string. (A single letter is called a character and a series of characters is called a string). Strings must always be put between inverted double commas. The ‘\n’ is called an escape sequence. In this case it represents a newline character. After printing something to the screen you usually want to print something on the next line. If there is no \n then a next printf command will print its string on the same line.

Commonly used escape sequences are:

* \n (newline)
* \t (tab)
* \v (vertical tab)
* \f (new page)
* \b (backspace)
* \r (carriage return)

After the last round bracket there must be a semi-colon. The semi-colon shows that it is the end of the command, just like full-stops in English and bar in Hindi language. (So in the future, don’t forget to put a semi-colon if a command ended).

## return 0;

When we wrote the first line **int main()** , we declared that main must **return** an integer int main(). (**int** is short for integer which is another word for number). With the command return 0; we can return the value null to the operating system. When you return with a zero you tell the operating system that there were no errors while running the program.

## Compilation

Compilation refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file. This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled. For instance, if you compile (but don't link) three separate files, you will have three object files created as output, each with the name **<filename>.o** or **<filename>.obj** (the extension will depend on your compiler). Each of these files contains a translation of your source code file into a machine language file -- but you can't run them yet! You need to turn them into executables your operating system can use. That's where the linker comes in.

## Linking

Linking refers to the creation of a single executable file from multiple object files. In this step, it is common that the linker will complain about undefined functions (commonly, main itself). During compilation, if the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file. If this isn't the case, there's no way the compiler would know -- it doesn't look at the contents of more than one file at a time. The linker, on the other hand, may look at multiple files and try to find references for the functions that weren't mentioned.

**Possible Errors during programming**

|  |  |
| --- | --- |
| **COMPILER** | **LINKER** |
| Compiler errors are usually syntactic in nature -- a missing semicolon, an extra parenthesis. | Linking errors usually have to do with missing or multiple definitions. |

**Advantage of Compilation -> Linking procedure**

1. It's easier to implement things that way.
2. The compiler does its thing, and the linker does its thing -- by keeping the functions separate, the complexity of the program is reduced.
3. This allows the creation of large programs without having to redo the compilation step every time a file is changed. Instead, using so called "conditional compilation", it is necessary to compile only those source files that have changed; for the rest, the object files are sufficient input for the linker.
4. This makes it simple to implement libraries of pre-compiled code: just create object files and link them just like any other object file.

P.S:  If you're working with an integrated development environment (IDE) it may already take care of this for you. If you're using command line tools, there's a nifty utility called [make](http://www.gnu.org/software/make/make.html) that comes with most \*nix distributions.

**Loader:**

A loader is a special type of a program that copies programs from a storage device to the main memory, where they can be executed. Most loaders are transparent to the users.

**Interpreter :**

like the compiler , the interpreter also executes instructions written in a high-level language. Basically, a program written in a high-level language can be executed in any of the two ways-by compiling the program or by passing the program through an interpreter.

The compiler translates instructions written in a high-level programming language directly in to machine language; the interpreter, on the other hand, translate the instructions into an intermediate form, which it then executes.

|  |  |
| --- | --- |
| Interpreter | Compiler |
| Translates program one statement at a time. | Scans the entire program and translates it as a whole into machine code. |
| It takes less amount of time to analyze the source code but the overall execution time is slower. | It takes large amount of time to analyze the source code but the overall execution time is comparatively faster. |
| No intermediate object code is generated, hence are memory efficient. | Generates intermediate object code which further requires linking, hence requires more memory. |
| Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy. | It generates the error message only after scanning the whole program. Hence debugging is comparatively hard. |
| Programming language like Python, Ruby use interpreters. | Programming language like C, C++ use compilers. |

The Hello World program is a small program that is easy to understand. But a program can contain thousands of lines of code and can be so complex that it is hard for us to understand. To make our lives easier it is possible to write an explanation or comment in a program. This makes it much easier to understand the code. (Even if you did not look at the code for years). These comments will be ignored by the compiler at compilation time.

Comments have to be put after

**//** *A Single Line Comment*

or be placed between

**/\***

***A really***

***Really***

***Long comment* \*/** .

Here is an example of how to comment the Hello World source code :

/\* Description : Print Hello World on the screen

Author : Your name

Date : 01/01/2007 \*/ #include<stdio.h>

int main()  
 {  
 //Print something and then newline  
 printf(“Hello World\n”);  
 return 0;  
 }

## Indentation

As you can see the **printf** and **return** statements have been indented or moved to the right side. This is  
done to make the code more readable. In a program as Hello World, it seems a stupid thing to do. But as the programs become more complex, you will see that it makes the code more readable.

So, always use indentations and comments to make the code more readable. It will make your life much easier if the code becomes more complex.