Compiling C files with gcc, step by step

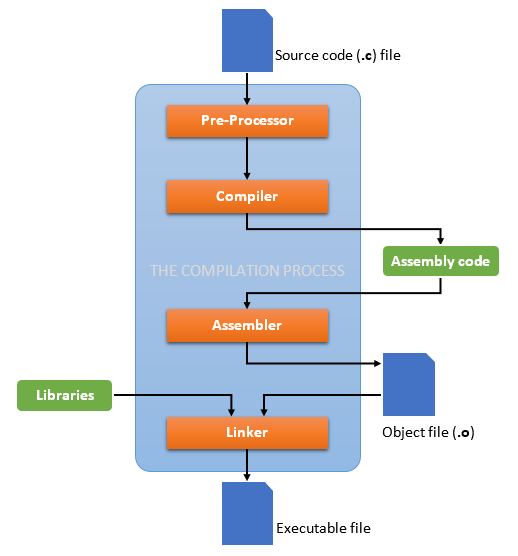
[](https://medium.com/@laura.derohan?source=post_page-----8e78318052----------------------)

[Laura Roudge](https://medium.com/@laura.derohan?source=post_page-----8e78318052----------------------)

Follow

[Feb 6, 2019](https://medium.com/@laura.derohan/compiling-c-files-with-gcc-step-by-step-8e78318052?source=post_page-----8e78318052----------------------) · 7 min read





In order to explain all the steps of compilation, we need to clarify a few programming concepts beforehand. In this article, we will cover what the C language is, how to compile it with a tool like gcc, and what happens when we compile it.

**The C programming language**

All the softwares, programs, websites and apps are written in a certain programming language. Basically, everything we see on the screen of our computers or smartphones are just a lot of code written in different languages and assembled together in a certain way. Every programming language has a different use, and today we are going to focus on C.

C is a programming language invented by Dennis Ritchie that first appeared in 1972. It’s what we call a low-level language, meaning there is only little abstraction between C and machine language, so it can be considered to be closer to the computer’s hardware. C is also a **compiled** language, as opposed to interpreted, meaning the source files written in C should be compiled in order for them to be executable.

**The tools**

Before anything else, let’s talk about the tools we’ll be using in our example. We will be working on a Unix-like operating system, so the examples may vary from Windows. We need to have access to the *shell*, which is “a program that takes commands from the keyboard and gives them to the operating system to perform” according to [http://linuxcommand.org](http://linuxcommand.org/). For this, we need a *terminal*, or terminal emulator, which is just a window that lets us interact with the shell. Inside the terminal, we should see the shell prompt, that contains your user name and the name of the machine, followed by the PS1 environment variable that is often the character “$”. We are able to input commands after this character in what we call the *command line.* We also need a text editor, like vi or emacs, to create a source file.





**Compilation**

Compilation is the translation of source code (the code we write) into object code (sequence of statements in machine language) by a **compiler.**

The compilation process has four different steps:

* The preprocessing
* The compiling
* The assembling
* The linking

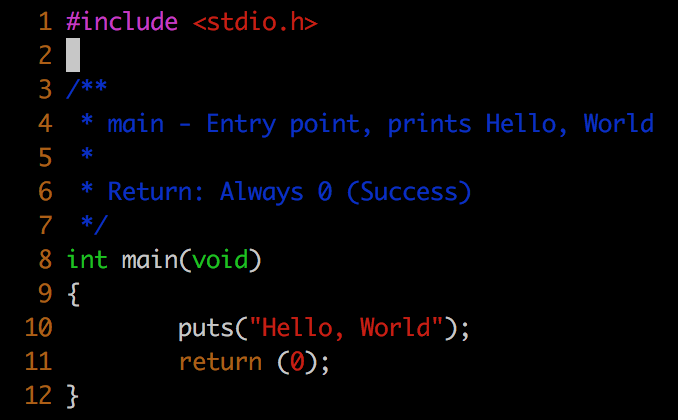
The compiler we will be using as an example is **gcc**which stands for**GNU Compiler Collection**. The GNU project is an free-software and mass-collaboration project launched by Richard Stallman in 1983, allowing developers to have access to powerful tools for free.

Gcc supports various programming languages, including C, is completely free and is the go-to compiler for most Unix-like operating systems. In order to use it, we should make sur we install it on our computer, if it’s not already there.

**The source code**

For our example, let’s take a look at a source code inside a file called “main.c”, where “.c” is a file extension that usually means the file is written in C. This picture is inside the text editor vi:





main.c

In pink is the preprocessor directive #include that tells the compiler to include the stdio.h header file, but we will come back to it later.

In blue are comments about the code, these are useful for remembering what your code actually does months after having creating it. We don’t really need them in such a small program, but it’s good practice to put them.

Next we have our entry point, the main() function. It means the program will start by executing the statements that are inside this function’s block, that is between the curly brackets. Here, there are only two statements: one that will print the sentence “Hello, World” on the terminal, and another one that tells the program to “return” 0 if it exited, or ended, correctly. So once we compiled it, if we run this program we will only see the phrase “Hello, World” appearing.

In order for our main.c code to be executable, we need to enter the command “gcc main.c”, and the compiling process will go through all of the four steps it contains. Of course gcc has options that allow us to stop the compiling process after each step. Let’s take a look at them.

**The steps**

**1. The preprocessor**

The preprocessor has several roles:

* it gets rid of all the *comments* in the source file(s)
* it includes the code of the *header file(s)*, which is a file with extension .h which contains C function declarations and macro definitions
* it replaces all of the *macros* (fragments of code which have been given a name) by their values

The output of this step will be stored in a file with a “.i” extension, so here it will be in main.i.

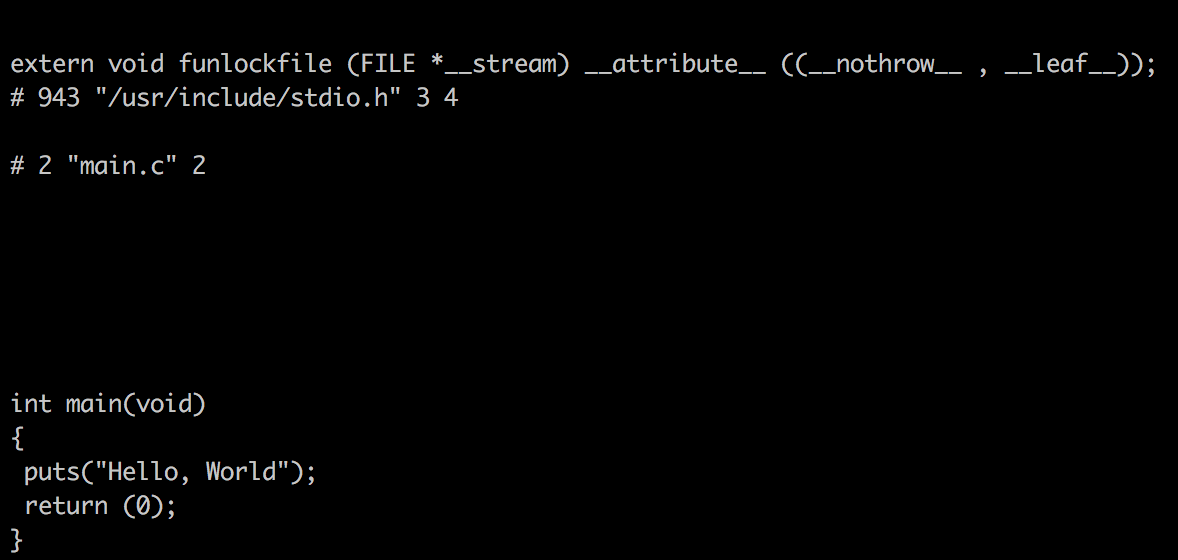
In order to stop the compilation right after this step, we can use the option “-E” with the gcc command on the source file, and press enter.





This is what the end of the main.i file should look like:





**2. The compiler**

The compiler will take the preprocessed file and generate IR code (Intermediate Representation), so this will produce a “.s” file. That being said, other compilers might produce assembly code at this step of compilation.

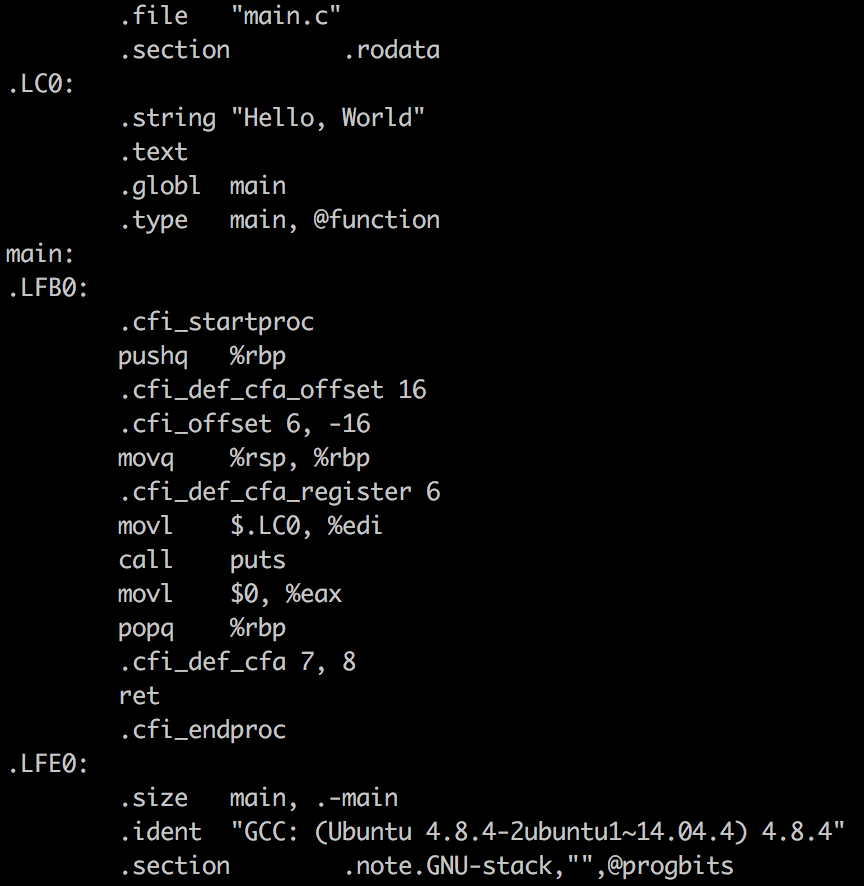
We can stop after this step with the “-S” option on the gcc command, and press enter.





This is what the main.s file should look like:





**3. The assembler**

The assembler takes the IR code and transforms it into object code, that is code in machine language (i.e. binary). This will produce a file ending in “.o”.

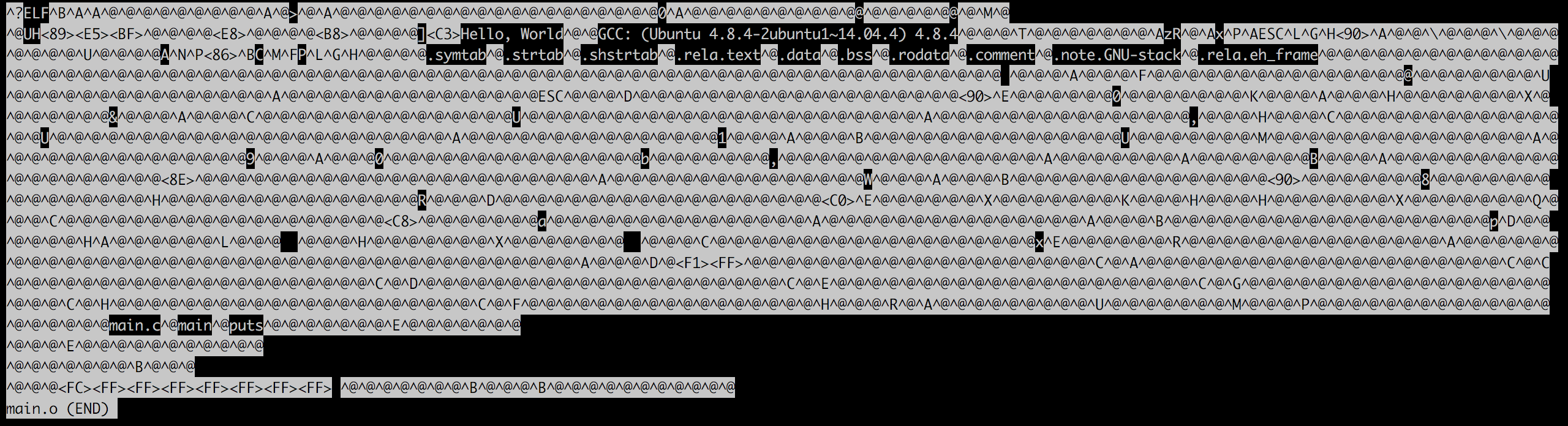
We can stop the compilation process after this step by using the option “-c” with the gcc command, and pressing enter.





Our main.o file should look like this (no, it’s not human readable):





**4. The linker**

The linker creates the final executable, in binary, and can play two roles:

* linking all the source files together, that is all the other object codes in the project. For example, if I want to compile main.c with another file called secondary.c and make them into one single program, this is the step where the object code of secondary.c (that is secondary.o) will be linked to the main.c object code (main.o).
* linking function calls with their definitions. The linker knows where to look for the function definitions in the *static libraries*or*dynamic libraries.*Static libraries are “the result of the linker making copy of all used library functions to the executable file”, according to geeksforgeeks.org, and dynamic libraries “don’t require the code to be copied, it is done by just placing the name of the library in the binary file”. Note that gcc uses by default dynamic libraries. In our example this is when the linker will find the definition of our “puts” function, and link it.

By default, after this fourth and last step, that is when you type the whole “gcc main.c” command without any options, the compiler will create an executable program called **a.out,**that we can run by typing **“./a.out”**in the command line.

We can also choose to create an executable program with the name we want, by adding the “-o” option to the gcc command, placed *after* the name of the file or files we are compiling, and pressing enter:





So now we could either type “./a.out” if you didn’t use the -o option or “./my\_program” to execute the compiled code, the output will be “Hello, World”, and following it the shell prompt will appear again.





The GNU Compiler Collection offers many more great tools to compile and operate our programs that would deserve to write about, but this article was about the basic steps of compilation. Maybe next time!