# Build_Process and C_Basics

## 1. Text editor

Some popular text editors used in the Unix/Linux OS environment are the following:

1. vim
2. emacs
3. gedit

Use the text editor of your choice to create the file **hello.c**

## 2. Build Process

The purpose of this part of the project is to understand the different steps involved in building an executable file from a C program. Task is to use the gcc C compiler to generate the following files as explained below.

### 2.1. Preprocessing Phase

Build the intermediate file after preprocessing and store it in a file named **hello.i**. Command used to build the intermediate file after preprocessing is:

```
gcc -E -o hello.i hello.c -m64 -Wall
```

1. The command to see the man page for gcc is: **`man gcc`**. Manual pages are useful to understand how to use linux commands.
2. **gcc**: - this is the command to invoke the GCC (GNU Compiler Collection) compiler.
3. **-E: -** this option tells the compiler to perform only preprocessing and stop before compilation. It processes the **#include** and **#define** directives in the source code and generate the preprocessed output.

4. **-o hello.i:** - this specifies the output file name as hello.i. Which stores the preprocessed code.
5. **hello.c:** - input source file.
6. The option `-m64` is used to generate code for a 64-bit environment
7. `-Wall:-` instructs the compiler to display the warning messages for common programming mistakes.

8. Open the file **hello.i** in the text editor and see how much extra code has been added.

9. the declaration of the function printf used in our C program. The declaration look something similar as shown below:

   ```
   extern int printf (__const char *__restrict __format, ...);
   ```

10. The reason for including the file **stdio.h** is mainly to let the compiler know that the function **printf()** that we are using in our program **hello.c** is actually declared in the file **stdio.h**.
11. The definition of **printf()** can be seen in the GNU C Library (glibc).

12. A ***declaration*** introduces an identifier and describes its type, be it a type, object, or function. A declaration is ***what the compiler needs*** to accept references to that identifier. These are declarations:

    ```
    extern int bar;
    extern int g(int, int);
    double f(int, double); // extern can be omitted for function declarations
    class foo; // no extern allowed for type declarations
    ```

13. A ***definition*** actually implements this identifier. It's ***what the linker needs*** in order to link references to those entities. These are definitions corresponding to the above declarations:

```
int bar;
int g(int lhs, int rhs) {
    return lhs*rhs;
}
double f(int i, double d) {
     return i+d;
}
class foo {};
```

14. Read this Stack Overflow question for details.

## 2.2. Compilation Phase

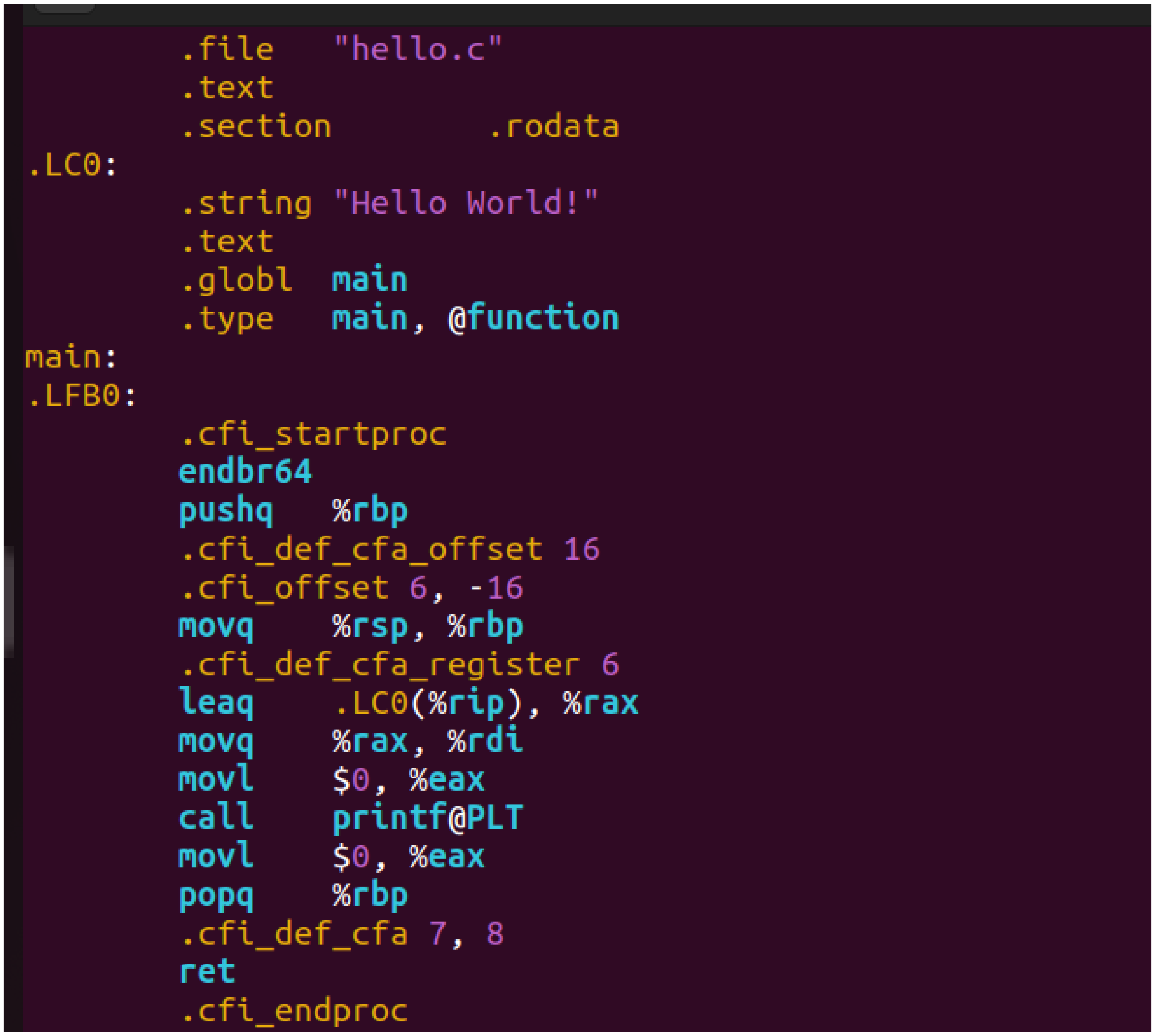
stop the compilation after the compiler generates the assembly file. The option to let gcc know that it should stop the build process after the compilation phase (named as "compilation proper" in the man page). The output of the compilation phase is a file named **hello.s**.
The following is the steps to generate the assembly code:

1. Run the following command at the command prompt:

```
gcc -S hello.c -m64 -Wall (OR)
gcc -S hello.i -m64 -Wall
```

1. give the file **hello.c** or the file **hello.i** as an input to gcc to generate the file **hello.s**.
2. Open the generated **hello.s** file in a text editor. Note that only a part of this file is shown here.

3. Just check if you can see the string “hello, world” and the label “main:” (as shown below) in the **hello.s** file.



```
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

## 2.3. Assembling Phase

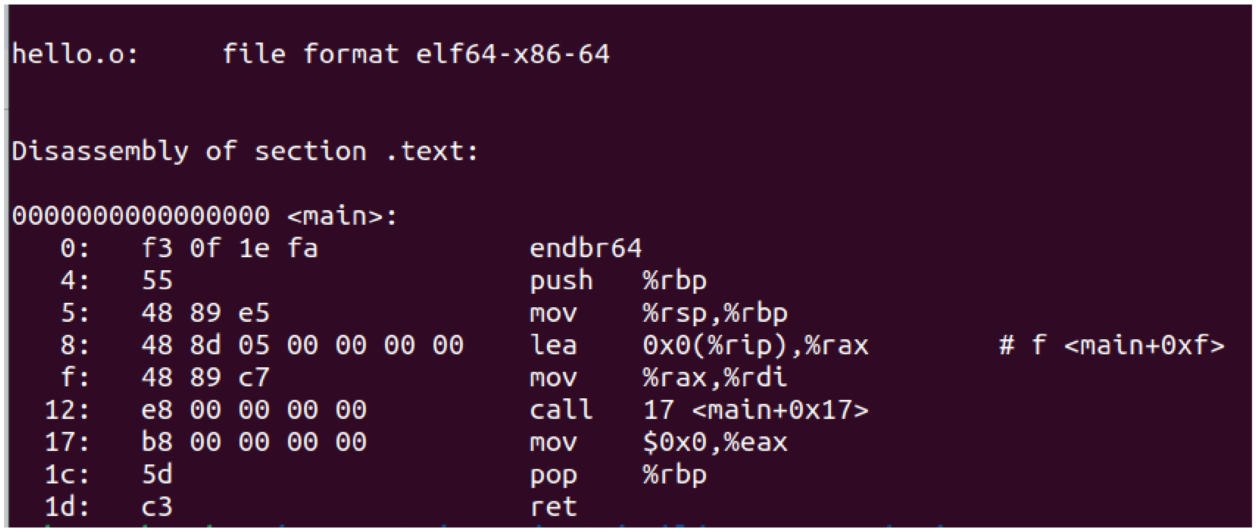
stop the build process after the assembling phase and before the linking phase. The output generated will be the object file for the C source file hello.o.

- Execute the following command to create the object file **hello.o**:

  **`gcc -c hello.c -m64 -Wall`** (OR) **`gcc -c hello.s -m64 -Wall`**

  The input to gcc can either be the C source file (hello.c) or the Assembly Code file (hello.s) that was generated from the previous step.

1. View the contents of the object file (hello.o) using a tool named **objdump** (object dump) as shown below. objdump is a disassembler which converts the machine code (in binary) to assembly code (human readable mnemonic form).
2. A disassembler does the inverse operation of an assembler (which converts assembly code into machine code).



```
hello.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf>
   f:   48 89 c7                mov    %rax,%rdi
  12:   e8 00 00 00 00          call   17 <main+0x17>
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   5d                      pop    %rbp
  1d:   c3                      ret
```

3. The use of the command **objdump** and the meaning of the option "**-d**" can be found by looking at the man page for objdump or typing "**objdump --help**" at the terminal.

   `objdump`: The main command for running the `objdump` tool.

   **-d, --disassemble        Display assembler contents of executable sections**

   When the **-d** option is passed to **objdump**, it will disassemble the machine code in the object file and present it in human-readable assembly language format.
   Disassembler determines the assembly code representation based purely on the byte sequences in the machine code file.

4. Save the disassembled output (shown in the image above) of the object file **hello.o** in a file named **objfile_contents.txt**. One easy way to do this is to redirect the output of the command "objdump -d hello.o" to a file named objfile_contents.txt as follows:

   **`objdump -d hello.o > objfile_contents.txt`** (The symbol '>' writes the output of the objdump command to the specified text file)

One other way to do this is to simply copy the contents of the output from the terminal and paste it in your text file.

## 2.4. Linking Phase

This is the final phase of the build process where the object file will be linked with some files in the standard C library to create the final executable file. steps to create the final executable file.

1. Execute the following command to complete the build process:

```
gcc hello.c -o hello -m64 -Wall (OR)
gcc hello.o -o hello -m64 -Wall
```

2. Execute the generated executable file (hello) using the following command in the terminal. You should be able to see the string "hello, world" printed on your screen.

```
./hello
```

3. Use objdump to view the disassembled contents of the executable file **hello**.
4. Redirect the disassembled output that you got from step 3 to a file named **exefile_contents.txt**. This file should be much larger than the disassembled output of the **hello.o** file since hello is an executable file which has information combined from **hello.o** and **printf** (binary file having the definition of the function printf).

Note: Even though we have seen each and every intermediate files that are created while we try to compile a C program, the two files that you'll most often use in your real life are the following:

1. C Source File (hello.c)
2. Executable File (hello)

Although we have multiple variations of using gcc, the most commonly used command in real life to compile the C program and to build the executable file are the following:

```
gcc hello.c -o hello
```
(in real life for a 64-bit machine or a 32-bit machine)

## 3. Writing a simple C program

write a C program named **integer.c**. Given an integer in the range (1, 100)  - 1 and 100 not included, program should print the following:

1. Hexadecimal representation of the integer
2. Octal representation of the integer
3. Whether the integer is a prime number or a composite number (prime vs. composite)

If the integer is not within the valid range, the program should print “not in range”. assume that no other types of input (e.g., double, string, etc) will be given to the program and no need to worry about any other input validation.

**Sample Output:**

```
$ ./integer
Enter an integer between 1 to 100 (1 and 100 excluded):
1
not in range

$ ./integer
Enter an integer between 1 to 100 (1 and 100 excluded):
2
0x2
02
Prime

$ ./integer
Enter an integer between 1 to 100 (1 and 100 excluded):
17
0x11
021
Prime
```

```
$ ./integer
Enter an integer between 1 to 100 (1 and 100 excluded):
50
0x32
062
Composite

$ ./integer
Enter an integer between 1 to 100 (1 and 100 excluded):
100
not in range
```

**HINT:** You may want to check the man page for printf to find out how to print the hexadecimal and octal versions of an integer.