C-Refresher: Session 01
GNU GCC Compiler

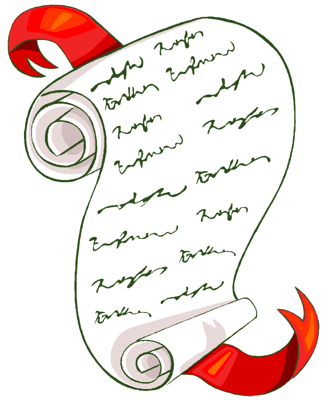
Arif Butt
Summer 2017

I am Thankful to my student Muhammad Zubair bcsf14m029@pucit.edu.pk for preparation of these slides in accordance with my video lectures at

http://www.arifbutt.me/category/c-behind-the-curtain/

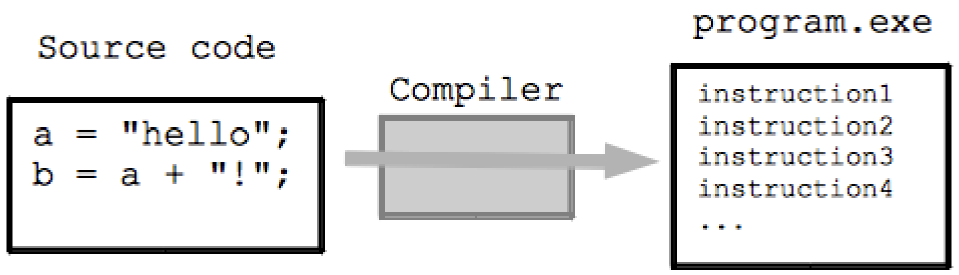
Today’s Agenda

- Brief Concept of GNU gcc Compiler
- Compilation Cycle of C-Programs
- Contents of Object File
- Multi-File Programs
- Linking Process
- Libraries



Compiler

Compiler is a program that transforms the source code of a high level language into underlying machine code.
Underlying machine can be x86 sparse, Linux, Motorola.



❑**Types of Compilers:**
gcc, clang, turbo C-compiler, visual C++…

GNU GCC Compiler

- GNU is an integrated distribution of compilers for C, C++, OBJC, OBJC++, JAVA, FORTRAN
- gcc can be used for cross compile

❑**Cross Compile:**

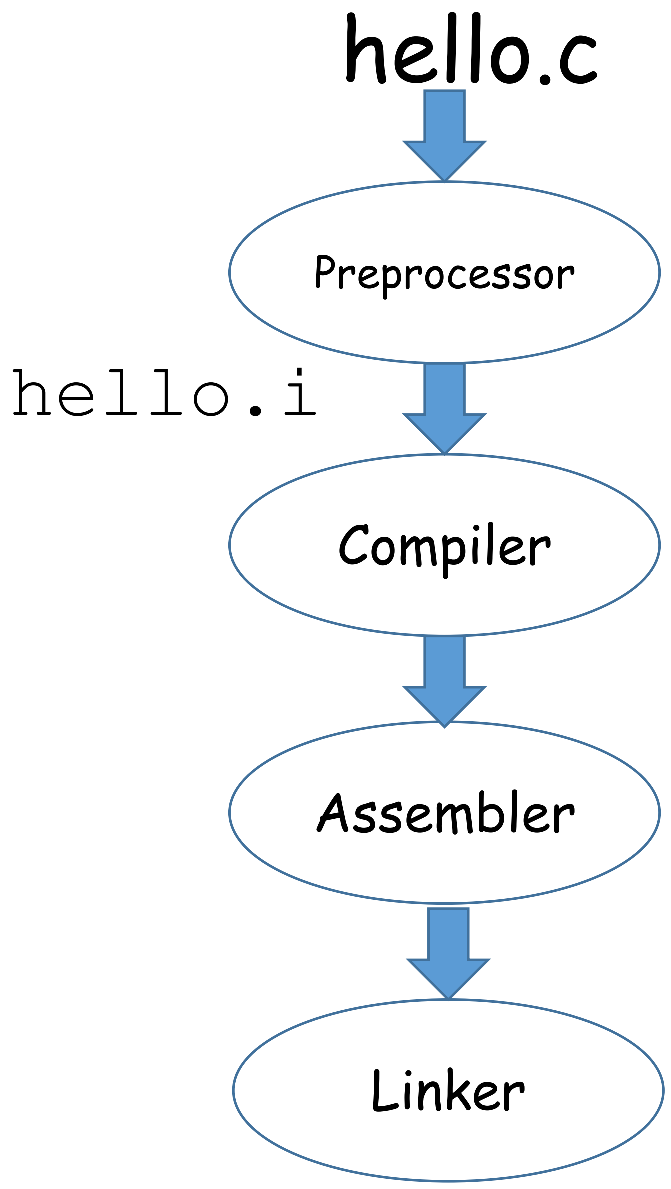
- Cross compile means to generate machine code for the platform other than the one in which it is running
- gcc uses tools like autoConf, automake and lib to generate codes for some other architecture

Compilation Process

❑ **Four stages of compilation:**

- Preprocessor
- Compiler
- Assembler
- Linker

Compilation Process(cont...)

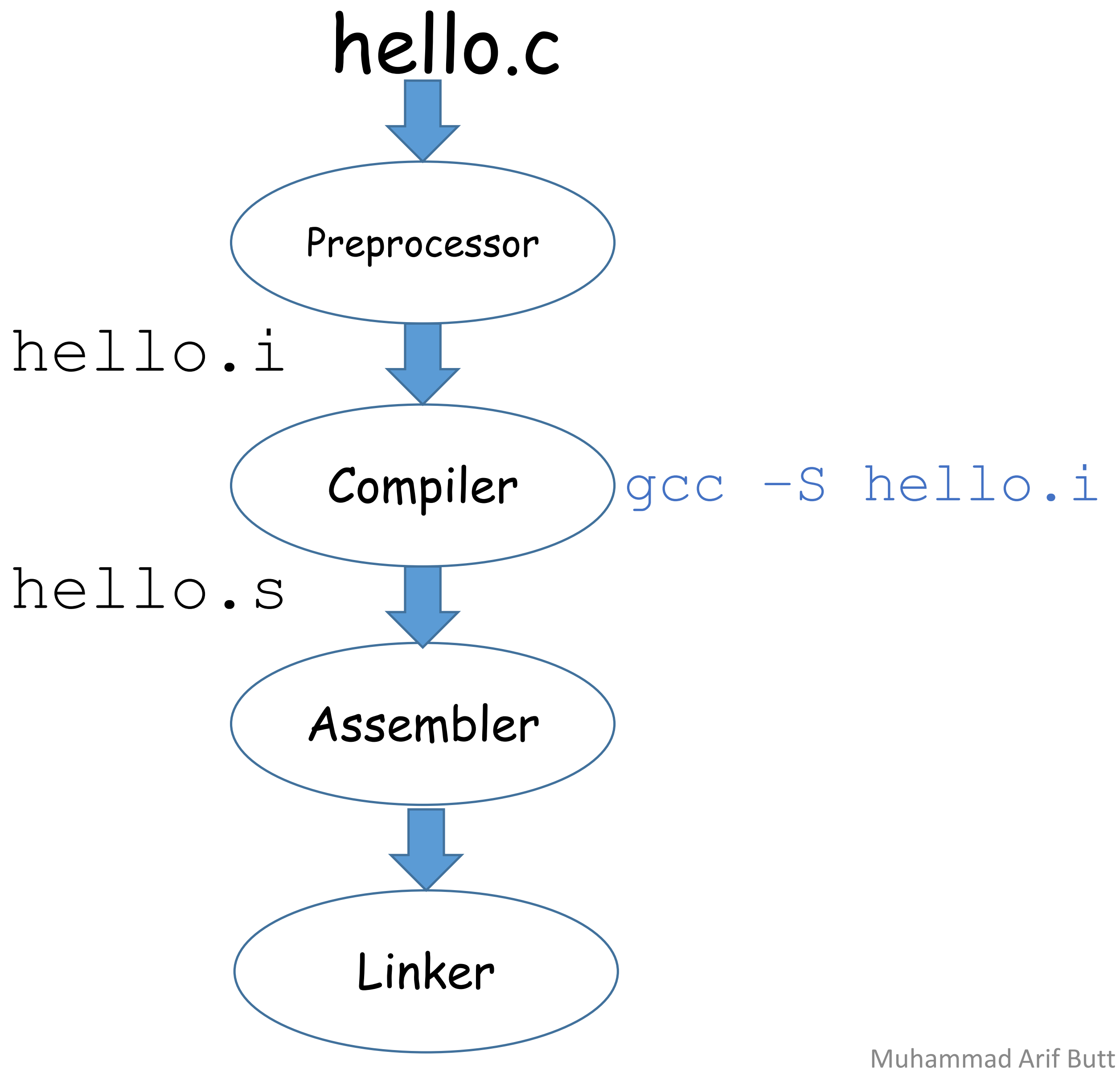


```
gcc -E hello.c 1>hello.i
```

1. Preprocessor

- Interpret Preprocessor directives
- Include header files
- Remove comments
- Expand headers

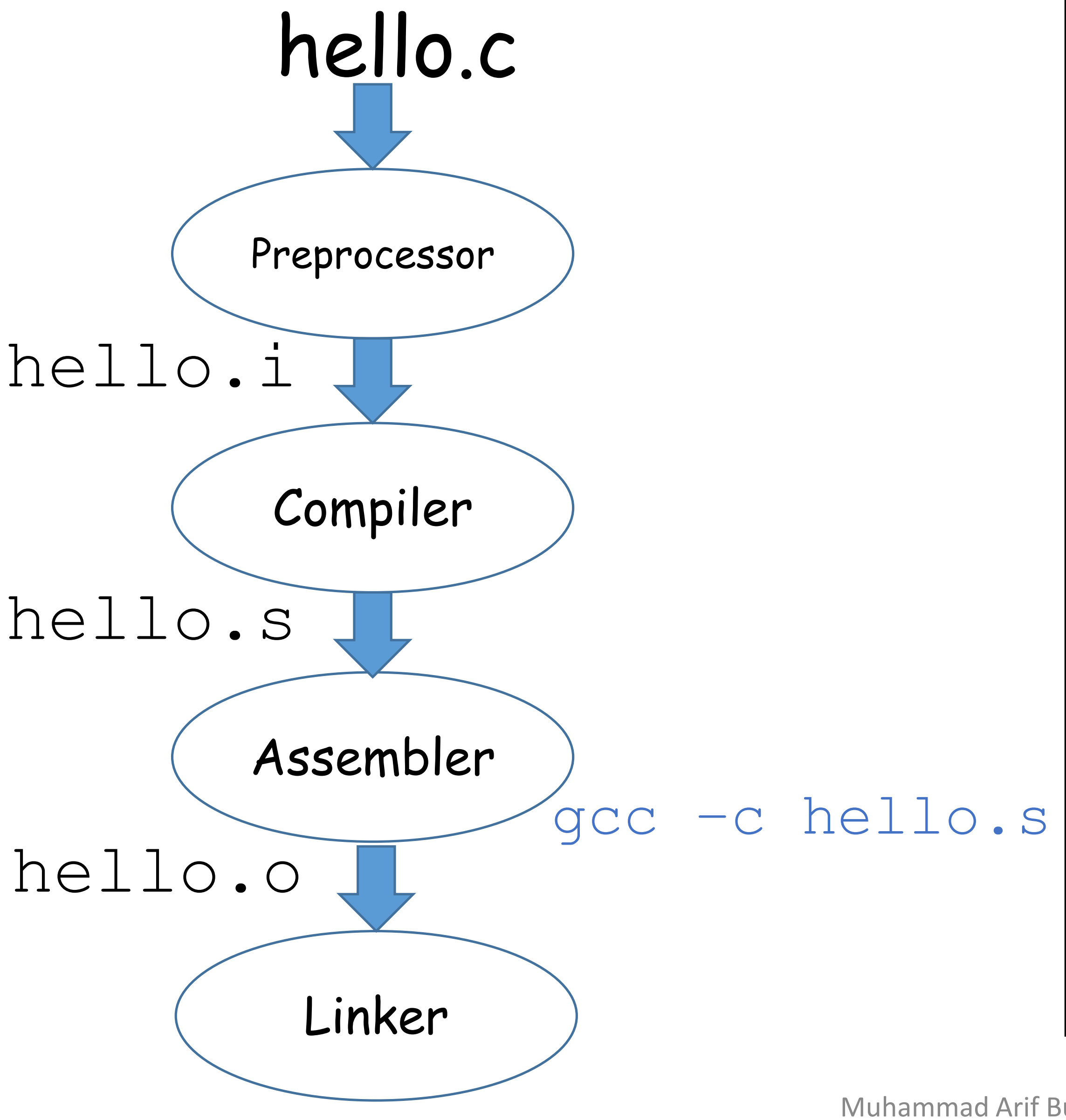
Compilation Process(cont...)



2. Compiler

- Check for syntax errors
- If no syntax error, the expanded code is converted to assembly code which is understood by the underlying processor, e.g. intel x86, AMD x86, SUN SPARC, ULTRA SPARC, ARM, Cell, Power PC, Motorola, MIPS

Compilation Process(cont...)

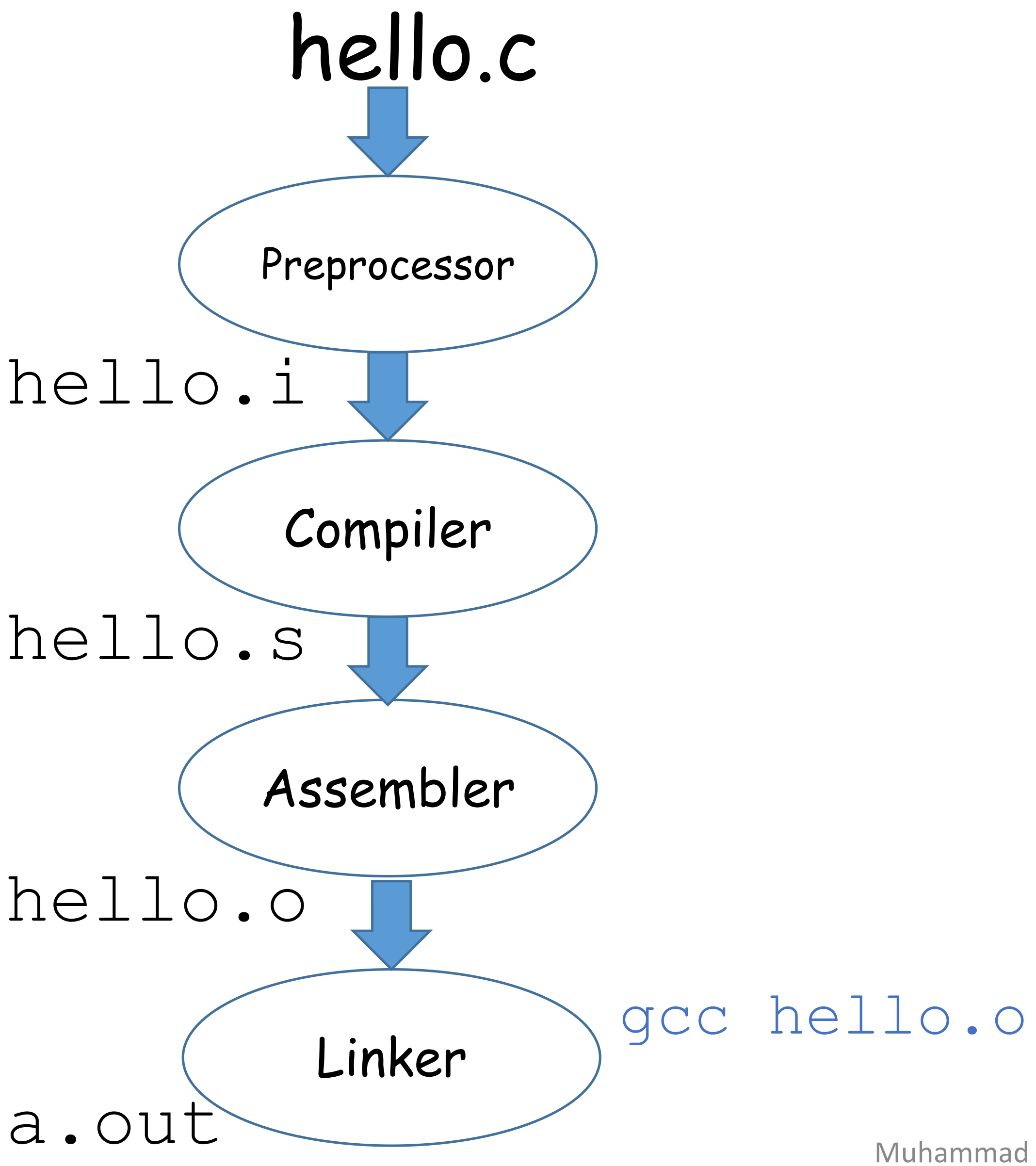


```
gcc -c hello.s
```

3. Assembler

- Assembler converts the assembly code to the machine dependent opcode
- Each object file contains a table known as symbol table which contains
  - Name, type and relative addresses of global variables
  - Name and relative addresses of functions defined in the program
  - Name of external functions like `printf()`

Compilation Process(cont...)

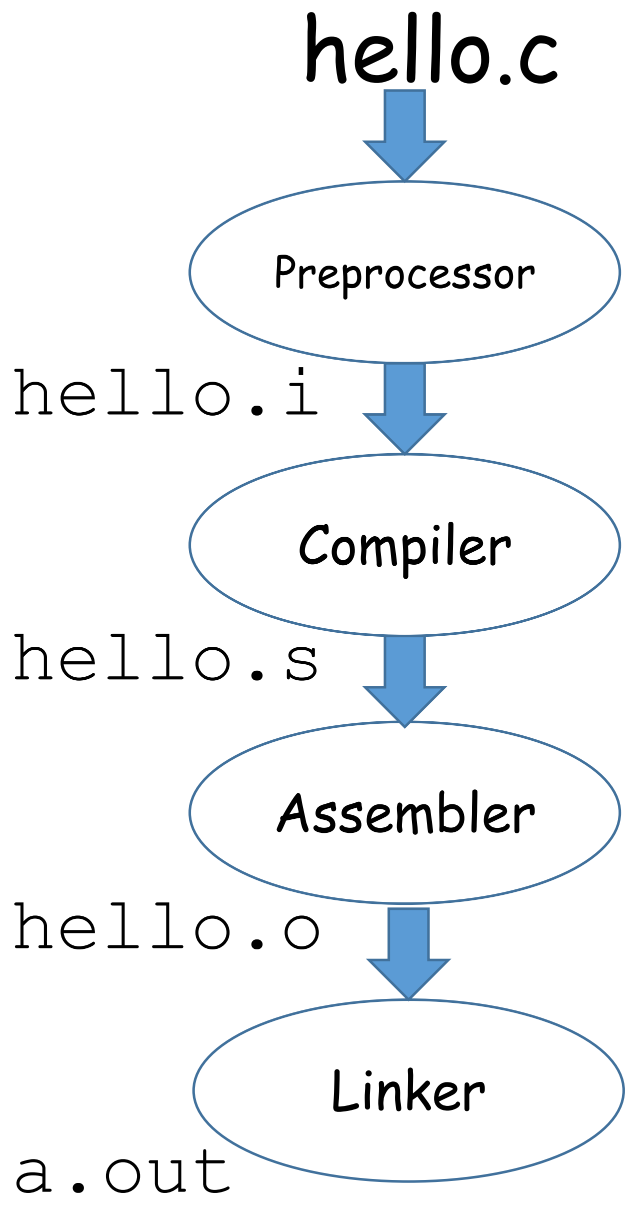


```
gcc hello.o
```

4. Linker

- Linker links a collection of object module(s) and libraries as input and combines them to produce a single executable
- It takes the symbol table of all `.o` files, that you have created, and combines them to create a global symbol table
- In case of a single source file, it links with appropriate functions of the standard C library implicitly

Compilation Process(cont…)



```
gcc -E hello.c 1>hello.i
gcc -S hello.i
gcc -c hello.s
gcc hello.o
```

Compilation Process(cont…)

- **Saving all files**

`gcc hello.c` => saves only the final `a.out` file while all the files created in between are deleted

`gcc -save-temps hello.c` =>saves all the files created in between, i.e. `hello.i, hello.s, hello.o` and finally `a.out`

- **Name of your choice**

`gcc hello.c -o myexe` => can be used to give a name of your choice other than `a.out` like `myexe` here

- ***C++ code compilation***

`g++ hello.cpp` => is used for compilation of C++ programs

Execution of C program

- **Command for execution of a C program**

`./a.out` (or if some other name of the executable file)

Why this ./ with the name of the executable file?

Execution of C program(cont...)

- **Answer:**

- The shell searches the executable file in the **`PATH`** variable (which contains the paths of different directories separated by column(:)) but the `PATH` variable generally doesn't contain the path of `PWD`(Present Working Directory)

- So there are two ways for Program Execution:

1. Add the `PWD` path in the `PATH` variable by using command

- `$export PATH=$PATH:$PWD`
- And then you can use only the name of the executable file to execute the program, like `a.out`

Execution of C program(cont...)

2. Use `./` before the name of the executable file. By doing this the shell looks for the executable file in `PWD`

Note: First way is generally not recommended.

Reading Object Files

- Object files **cannot** be read using `cat`, `more` or `less` commands
- The **reason** is, these files are not in text format
- They are in ELF format
- Note: `file` command can be used to know the format of a file
- **Syntax**
  - `file [filename]`

Reading Object Files(cont...)

- Commands used for reading files of `ELF` format are:
  - readelf
  - od
  - objdump

❑ **readelf**

- This command is used to read the contents of an `a.out` (executable) file or some other `ELF` file

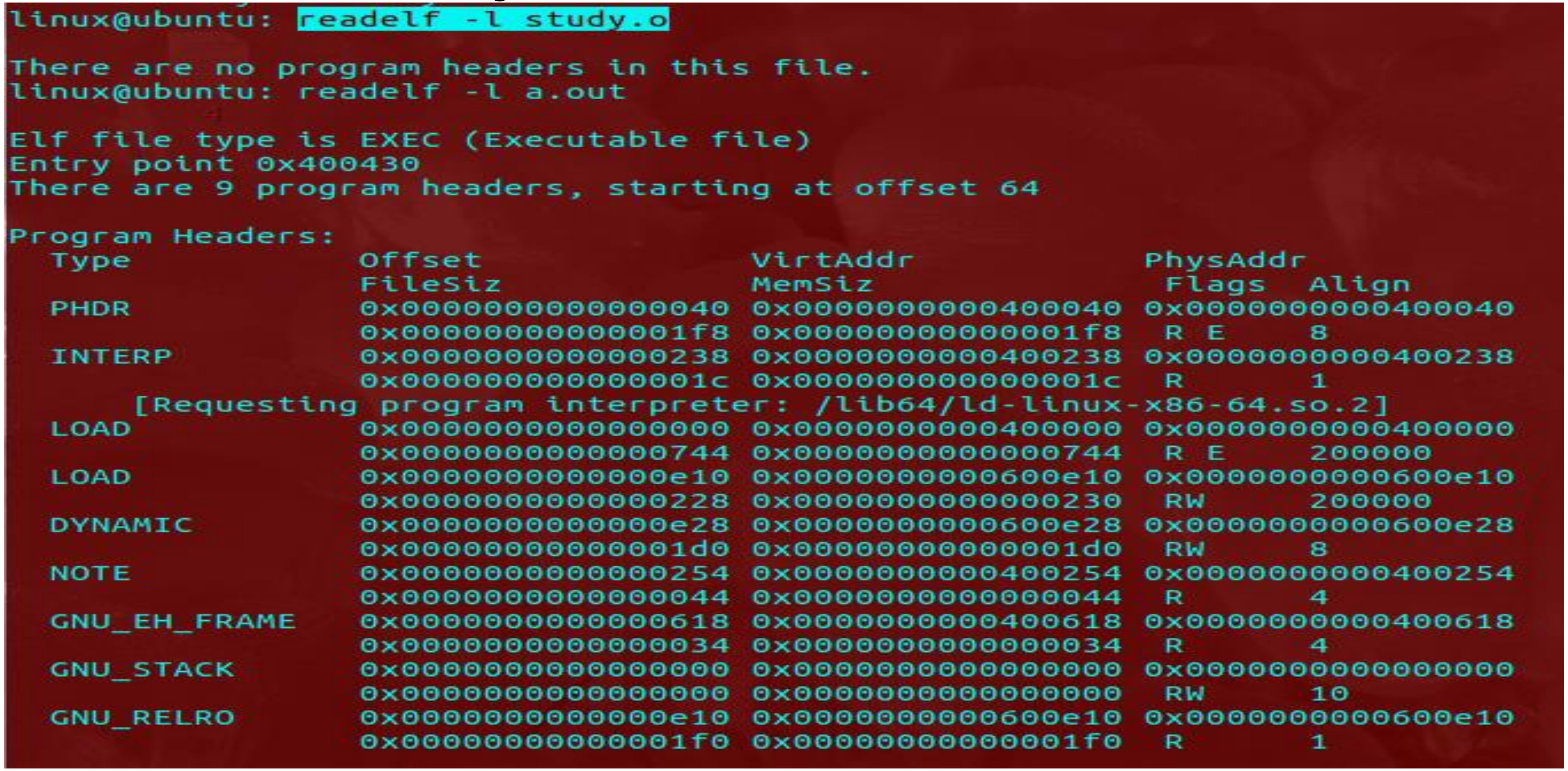
Syntax

```
readelf [option(s)] [argument(s)]
```

Some options are:

`readelf -l a.out` => to read program headers (program headers reside in only executable files not in `*.o` files)

Reading Object Files(cont...)



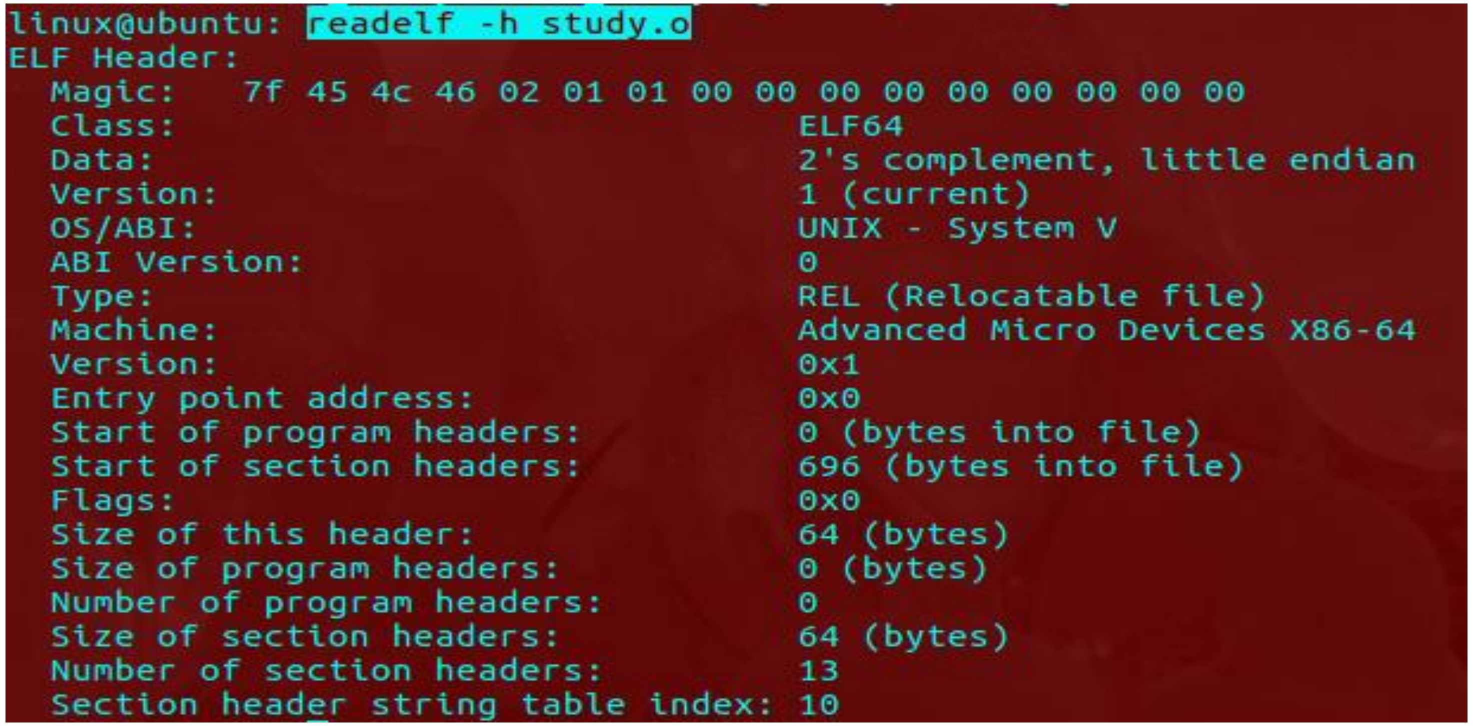
```
linux@ubuntu: readelf -l study.o

There are no program headers in this file.
linux@ubuntu: readelf -l a.out

Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000744 0x0000000000000744  R E    200000
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x0000000000000228 0x0000000000000230  RW     200000
  DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
                 0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000000618 0x0000000000400618 0x0000000000400618
                 0x0000000000000034 0x0000000000000034  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001f0 0x00000000000001f0  R      1
```

Example result of `readelf –l a.out`

Reading Object Files(cont...)

`readelf -l` is telling about the program type, entry point and is showing lots of information about program headers

`readelf -h hello.o` => shows file headers (file headers are in both `*.o` and `a.out` file)



```
linux@ubuntu: readelf -h study.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          696 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

Here it is showing file headers of a file namely `study.o` which including magic#, class, data, version and lots of other information

Reading Object Files(cont...)

- There are also other options available like
- `readelf -g` => Displays the information contained in the file's section groups, if it has any
- `readelf -S` => Displays the information contained in the file's section headers
- `readelf -t` => Displays the detailed section information
- and many more options

Reading Object Files(cont...)

❑od (Octal dump)

- `od` command dump files in octal and other formats
- **Syntax**
  - `od [option(s)] [argument(s)]`
- e.g.
- `od hello.o`

  //shows octal dump of `hello.o`
- `od –h hello.o`

  //shows hexa dump of `hello.o`

Reading Object Files(cont...)

❑ **objdump (object dump)**

- `objdump` is used to display information about object files
- e.g.
- `objdump –d hello.o`
- This command is used to disassemble
- Disassembling of section `.text` means showing the assembly code of underlying machine code

Multifile Programs

- Beginners write C-Programs in a single file containing the `main()` and zero or more functions. The source file may also contain Preprocessor directives, type and macro definitions, variables and function declarations
- But programs can be large! e.g., Linux-4.9 contains about 4.3M LOCs. So large C software needs to be divided into multiple source files
- Let's take a very simple example to understand this
- Suppose we are to write some basic math related functions and we want to write them in separate files

<u>Multifile Programs(cont…)</u>

- To accomplish this, we have to include their header file in the `main()` function file
- Let the header file name be **`mymath.h`**, now this file has to included in the `main()` function file
- <u>There are two ways of including our own header files</u>

1. include the file using

- `#include<mymath.h>`
- Here the file has been included using <> symbols so the compiler will search for the `mymath.h` file in **`/usr/include/`** directory

Multifile Programs(cont...)

- So now in order to compile the program we have to copy `mymath.h` file in **`/usr/include/`** directory **or** we have to compile the file using **`-I.`** option
- e.g. `gcc *.c -I.`
- This **-I.** option actually tells the compiler to search for the included file in `PWD`

2. **include the file using**

- `#include "mymath.h"`
- Now the compiler will look for the header file in `PWD`, as it has been included using "" symbols

Linking Process of libraries

- C-programs we write are often using libraries and these libraries have to be linked with the program for its successful compilation
- To understand this concept of linking of libraries let's start with an example program which uses some library functions and so needs the related library for its successful compilation

Linking process of Libraries(cont…)

```
//mymath.c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
int main(int argc,char* argv){
double x=atof(argv[1]);
double y=strtof(argv[2],NULL);
double and=pow(x,y);
printf("%lf^%lf=%lf"x,y,ans);
return 0;
}
```

Linking process of Libraries(cont…)

- The above code used `atof()`, `strtof()` and `pow()` functions
- Now `atof()` and `strtof()` are present in the standard C library while the `pow()` function is present there in the math library so our program needs both these libraries
- C-standard libraries are located in **`/usr/lib/x86_64-linux-gnu/`**
- **`libc.so`** & **`libc.a`** are both the standard C libraries with the difference that **`libc.so`** is the dynamic version while **`libc.a`** is the static version

Linking process of Libraries(cont…)

- `gcc` automatically links with dynamic version, when a library is imported, and if it is not available, only then it goes for the static version
- Similarly, math libraries are available by the names **`libm.so`** and **`libm.a`**
- So to compile the above **`mymath.c`** program the command used is

`gcc mymath.c -lc -lm` or `gcc  mymath.c -lm` (as **`lc`** is automatically linked by `gcc`)

- Note: To import a library use the starting character **`l`** and then the character(s) after **`lib`**

e.g.
libm

Linking process of Libraries(cont…)

- There are two ways of linking of libraries:

Dynamic linking:

- Only a reference to the linked libraries is placed in the object file
- `gcc mymath.c –lm`
- Smaller size of object file
- e.g. size of object file of `mymath.c` comes out to be 8.6K
- Default method

Static Linking:

- The whole library code is placed in the object file
- `gcc --static mymath.c –lm`
- Larger size of object file
- e.g. size of object file of `mymath.c` comes out to be 1.1M
- Not a default method has to be explicitly specified

Linking process of Libraries(cont…)

- There are other commands available which can help us linking of libraries

❑ **ldd:**

  - `ldd` is a command used to print shared object dependencies

- e.g. for above `mymath.c` program
- `gcc mymath.c -o dynamicM -lm` /*will produce dynamically linked executable file named `dynamicM`*/

Linking process of Libraries(cont…)

- `ldd dynamicM` //shows the following result
  - `linux-v dso.so.1` => (0x00007ffc329a0000)

    //linked with `vdso`
  - `libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fab2e66d000)`

    //linked with libm.so
  - `libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6` (0x00007fab2e2a4000)

    //linked with `libc.so`
  - `/lib64/ld-linux-x86-64.so.2 (0x000055d653ab1000)`

    //linked with `ld`

Linking process of Libraries(cont…)

- **vdso**
  - The `vdso` (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the `vDSO` is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take care of using any functionality that is available via the `vDSO`
- **ld**
  - `ld` is the basic gnu linker. So `ld` can also be used in place of `gcc` for compilation of a program with few differences

Linking process of Libraries(cont…)

- **Difference of `ld` from `gcc`**
  - In `ld`, all the libraries, including standard C libraries, have to be linked explicitly while in `gcc` standard C libraries are automatically linked
  - e.g. `ld mymath.h –lm`

    /*this will give an error message showing that undefined reference to `strtof()` and `atof()`*/

    So for successful compilation we have to use the command

    ```
    ld mymath.h –lc –lm
    ```
  - `ld` looks for the `_start` symbol to start the execution of the program while `gcc` looks for the `main()` symbol

Linking process of Libraries(cont…)

- nm
  - `nm` is a GNU command
  - `nm` lists the symbols from object files.

    e.g. `nm mymath.o`   //shows the following result

    ```
    U atof
    T main
    U pow
    U printf
    U strtof
    ```
  - If no object files are listed as arguments, `nm` assumes the file `a.out`
  - e.g. `nm` and `nm a.out` both give the same result

SUMMARY