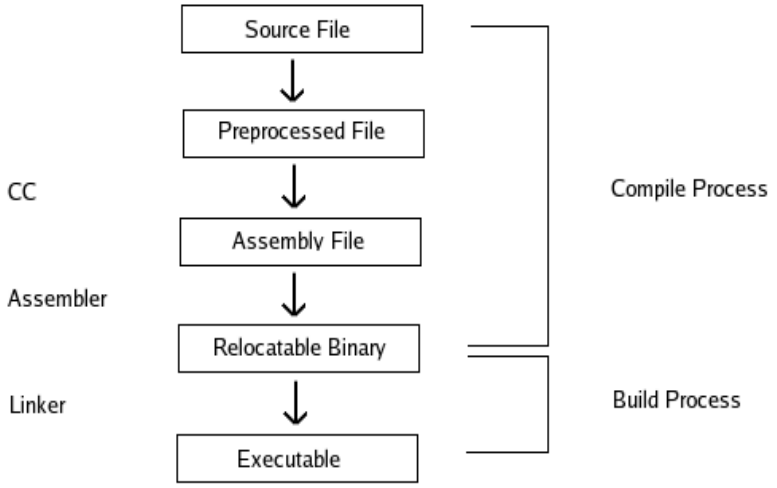
Compilation Stages of a C program – Class Notes



- The above diagram shows the four compilation stages through which a source code passes to become an executable.

In order to understand the function of each stage we will consider a sample C program **test.c**

$ **vim test.c**

```
#include<stdio.h>
int  main()
{
    printf("Welcome to Veda\n");
    return 0;
}
```

Now let's run gcc compiler to get the executable

$ **gcc test.c -o test**

$ **./test**

Welcome to Veda

- Now we have a basic idea about how gcc is used to convert a source code into binary. We'll review the 4 stages of a C program.

1) **Preprocessor:**

This is the very first stage through which a source code passes. To understand preprocessing better, you can compile the above '**test.c'** program using flag **-E**, which will generate the preprocessed output. Preprocessor converts our .c file into .i file.

$**gcc -E test.c -o test.i**

If we go through the **test.i** file we will notice that **stdio.h** header file will be replaced with its full code. Thus the preprocessor adds all the header files described in the source code.

In the above command if we add a flag -**v** i.e.,

$**gcc -v -E test.c -o test.i**

This will open the specs, in which the order of the execution of each tool and the command line arguments are specified. We can also see that gcc is using cc1 to compile.

2) **Compiler :**

The job of the compiler is to take the preprocessor output as an input, here it will take **test.i** and produce an assembler output. The output of the compiler is architecture dependent. The output file for this stage is '**test.s'**. The output present in **test.s** is assembly level instructions.

$gcc -S test.i -o test.s

If we go through the test.s file, we can find the machine level instructions which the assembler understands.

3) **Assembler:**

At this stage the **test.s** file is taken as an input and an intermediate file test.o is produced. This file is also known as the relocatable file. This file is produced by the assembler that understands

and converts a **'.s**' file with assembly instructions into a **'.o**' object file which contains machine level instructions.

$gcc -c test.s -o test.o

- Here if we give -**v** flag we can find that gcc is using '**as**' i.e., it is invoking assembler.
- Since the output of this stage is a machine level file (test.o). So we cannot view the content of it through editor. This can be viewed through a special tool called **objdump**

$objdump –D test.o | more

4) Linker:

This is the final stage at which all the linking of function calls with their definitions is done. That means it links all the libraries and object files into single executable file. The linker also does some extra work; it adds some extra code to our program that is required when the program starts and when the program ends. We call this code as **runtime code**. Thus it generates executable image from the relocatable file.

$gcc test.o -o test

DIFFERENCE BETWEEN RELOCATABLES AND EXECUTABLES

RELOCATABLES	EXECUTABLES
1. Instructions of relocatable binary are bound to offset address assigned as per the position of the instruction within the procedure Eg: 00000000<main> 0: 55 push %ebp	1. Executable binary contains instructions bound to platform specific load address. Eg: 080483e4<main> 80483e4: 55 push %ebp
2. Function call instructions in relocatable object files referred to called functions offset position Eg: call 12 <main+0x12>	2. Call instructions in executable binaries referred to functions base address Eg: call 8048300<printf@plt>
3. It contains compiled instructions that appear in the source file.	3. It contains functionality plus run time code.

Functionalities of Linker

- Instruction relocation
- Procedure relocation (assigning address to all the functions)
- Append runtime code into executable image.

Significances of Runtime code :

- Runtime code is responsible for allocating stack segment and configuring it on the process address space.

  When a program is loaded from the binary file the memory (address space) allocated for it comprises of code and data segments and a stack segment which is needed for the execution of the program.

- Linux run time code comprises of three key functions
  - Init: This function is responsible for allocation of stack and appending to address space and configuration of stack.
  - Start: This function is invoked after initialization of stack segment is carried out, and is responsible for invoking main function.
  - Fini : When main function returns, start invokes fini, it is responsible for releasing stack segment.
- Run time code invokes kernel system calls and is OS specific. So presence of runtime makes an application binary OS specific.