



Compiling a C program:- Behind the Scenes

C is a high level language and it needs a compiler to convert it into an executable code so that the program can be run on our machine.

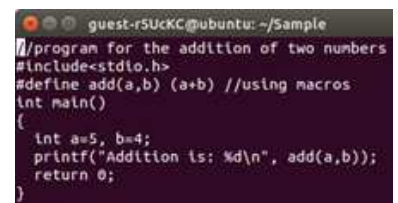
How do we compile and run a C program?

Below are the steps we use on an Ubuntu machine with gcc compiler.

- We first create a C program using an editor and save the file as filename.c

```
$ vi filename.c
```

The diagram on right shows a simple program to add two numbers.



```
guest-r5UcKC@ubuntu: ~/Sample
//program for the addition of two numbers
#include<stdio.h>
#define add(a,b) (a+b) //using macros
int main()
{
    int a=5, b=4;
    printf("Addition is: %d\n", add(a,b));
    return 0;
}
```

- Then compile it using below command.

```
$ gcc -Wall filename.c -o filename
```

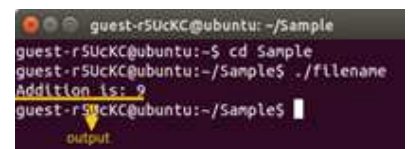
The option -Wall enables all compiler's warning messages. This option is recommended to generate better code.

The option -o is used to specify output file name. If we do not use this option, then an output file with name a.out is generated.



- After compilation executable is generated and we run the generated executable using below command.

```
$ ./filename
```



```
guest-r5UcKC@ubuntu: ~/Sample
guest-r5UcKC@ubuntu:~$ cd Sample
guest-r5UcKC@ubuntu:~/Sample$ ./filename
Addition is: 9
guest-r5UcKC@ubuntu:~/Sample$
```

What goes inside the compilation process?

Compiler converts a C program into an executable. There are four phases for a C program to become an executable:

1. Pre-processing

2. Compilation
3. Assembly
4. Linking

By executing below command, We get the all intermediate files in the current directory along with the executable.

```
$gcc -Wall -save-temps filename.c -o filename
```

The following screenshot shows all generated intermediate files.



Let us one by one see what these intermediate files contain.

Pre-processing

This is the first phase through which source code is passed. This phase include:

- Removal of Comments
- Expansion of Macros
- Expansion of the included files.

The preprocessed output is stored in the **filename.i**. Let's see what's inside filename.i: using **\$vi filename.i**

```
extern char *ctermid (char * _s) __attribute__ ((__nothrow__ , __leaf__));
# 913 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE * _stream) __attribute__ ((__nothrow__ , __leaf__));

extern int ftrylockfile (FILE * _stream) __attribute__ ((__nothrow__ , __leaf__));
} 1;

extern void funlockfile (FILE * _stream) __attribute__ ((__nothrow__ , __leaf__));
} 2;
# 943 "/usr/include/stdio.h" 3 4
# 3 "filename.c" 2
int main()
{
    int a=5, b=4;
    printf("Addition is: %d\n",a+b);
    return 0;
}
```

```
extern int fprintf (FILE * __restrict __stream,
    const char * __restrict __format, ...);

keyword extern tells that it is not defined here. It is external to this file.
extern int printf (const char * __restrict __format, ...);
extern int sprintf (char * __restrict __s,
    const char * __restrict __format, ...) __attribute__ ((__nothrow__));

extern int vfprintf (FILE * __restrict __s, const char * __restrict __format,
    __gnuc_va_list __arg);
```

In the above output, source file is filled with lots and lots of info, but at the end our code is preserved.

Analysis:

- printf contains now a + b rather than add(a, b) that's because macros have expanded.
- Comments are stripped off.
- **#include<stdio.h>** is missing instead we see lots of code. So header files has been expanded and included in our source file.

Compiling

The next step is to compile filename.i and produce an; intermediate compiled output file **filename.s**. This file is in assembly level instructions. Let's see through this file using **\$vi filename.s**

```
guest-wifk80@ubuntu: ~/Sample
file "filename.c"
.section .rodata


.LC0:
.string "Addition is: %d\n"
.text
.globl main
.type main, @function

main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
movl $5, -8(%rbp)
movl $4, -4(%rbp)
movl -4(%rbp), %eax
movl -8(%rbp), %edx
addl %edx, %eax
movl %eax, %esi
movl $.LC0, %edi
```

The snapshot shows that it is in assembly language, which assembler can understand.

Assembly.

In this phase the filename.s is taken as input and turned into **filename.o** by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved. Let's view this file using **\$vi filename.o**



```

guest-wifi@ubuntu: ~/Sample
CLF:~/Sample$ ls
ls: cannot access 'ls': No such file or directory

```

Linking

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using **\$size filename.o** and **\$size filename**. Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.

```

guest-wifk80@ubuntu: ~/Sample
guest-wifk80@ubuntu:~$ cd Sample
guest-wifk80@ubuntu:~/Sample$ size filename.o
text    data    bss     dec      hex filename.o
127      0         0      127      7f filename.o
guest-wifk80@ubuntu:~/Sample$ size filename
text    data    bss     dec      hex filename
1253    560     8      1821     71d filename
guest-wifk80@ubuntu:~/Sample$

```

Note that GCC by default does dynamic linking, so printf() is dynamically linked in above program. Refer [this](#), [this](#) and [this](#) for more details on static and dynamic linkings.

This article is contributed by **Vikash Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Article Tags : [C Quiz](#) [C Basics](#) [C++ Basics](#) [system-programming](#)

[Login to Improve this Article](#)

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Recommended Posts:

[How are variables scoped in C – Static or Dynamic?](#)
[Interesting Facts about Macros and Preprocessors in C](#)
[C | C Quiz – 113 | Question 1](#)
[Understanding “extern” keyword in C](#)
[Difference between “int main\(\)” and “int main\(void\)” in C/C++?](#)
[Output of C programs | Set 64 \(Pointers\)](#)
[C | Macro & Preprocessor | Question 15](#)
[C Quiz – 112 | Question 5](#)
[C Quiz – 112 | Question 4](#)
[C Quiz – 112 | Question 3](#)

(Login to Rate)

2.3 Average Difficulty : **2.3/5.0**
Based on **126** vote(s)

☐ Add to TODO List
☐ Mark as DONE

[Feedback](#)

[Add Notes](#)

Basic

Easy

Medium

Hard

Expert

[Improve Article](#)

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

[Load Comments](#)

[Share this post!](#)

A computer science portal for geeks

710-B, Advant Navis Business Park,
Sector-142, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org

COMPANY

About Us
Careers
Privacy Policy
Contact Us

PRACTICE

Company-wise
Topic-wise
Contests
Subjective Questions

LEARN

Algorithms
Data Structures
Languages
CS Subjects
Video Tutorials

CONTRIBUTE

Write an Article
Write Interview Experience
Internships
Videos

@geeksforgeeks, Some rights reserved