**The Four Stages of Compiling a C Program**

by **[Calle Erlandsson](https://www.calleerlandsson.com/)** · August 7, 2015   
Tagged c

Knowing how compilation works can be very helpful both when writing code and debugging.

Compiling a C program is a multi-stage process. At an overview level, the process can be split into four separate stages: [Preprocessing](https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/#preprocessing), [compilation](https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/#compilation), [assembly](https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/#assembly), and [linking](https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/#linking). Traditional C compilers orchestrate this process by invoking other programs to handle each stage.

In this post, I’ll walk through each of the four stages of compiling the following C program:

*/\**

*\* "Hello, World!": A classic.*

*\*/*

#include <stdio.h>

int

main(void)

{

puts("Hello, World!");

**return** 0;

}

**preprocessing**

The first stage of compilation is called preprocessing. In this stage, lines starting with a # character are interpreted by the *preprocessor* as *preprocessor commands*. These commands form a simple macro language with its own syntax and semantics. This language is used to reduce repetition in source code by providing functionality to inline files, define macros and to conditionally omit code.

Before interpreting commands, the preprocessor does some initial processing. This includes joining continued lines (lines ending with a \) and stripping comments.

To print the result of the preprocessing stage, pass the -E option to cc:

cc -E hello\_world.c

Given the “Hello, World!” example above, the preprocessor will produce the contents of the stdio.h header file joined with the contents of thehello\_world.c file, stripped free from its leading comment:

[lines omitted for brevity]

extern int \_\_vsnprintf\_chk (char \* restrict, size\_t,

int, size\_t, const char \* restrict, va\_list);

# 493 "/usr/include/stdio.h" 2 3 4

# 2 "hello\_world.c" 2

int

main(void) {

puts("Hello, World!");

return 0;

}

**compilation**

The second stage of compilation is confusingly enough called compilation. In this stage, the preprocessed code is translated to *assembly instructions* specific to the target processor architecture. These form an intermediate human readable language.

The existence of this step allows for C code to contain inline assembly instructions and for different *assemblers* to be used.

Some compilers also supports the use of an integrated assembler, in which the compilation stage generates *machine code* directly, avoiding the overhead of generating the intermediate assembly instructions and invoking the assembler.

To save the result of the compilation stage, pass the -S option to cc:

cc -S hello\_world.c

This will create a file named hello\_world.s, containing the generated assembly instructions. On Mac OS 10.10.4, where cc is an alias for clang, the following output is generated:

.section \_\_TEXT,\_\_text,regular,pure\_instructions

.macosx\_version\_min 10, 10

.globl \_main

.align 4, 0x90

\_main: ## @main

.cfi\_startproc

## BB#0:

pushq %rbp

Ltmp0:

.cfi\_def\_cfa\_offset 16

Ltmp1:

.cfi\_offset %rbp, -16

movq %rsp, %rbp

Ltmp2:

.cfi\_def\_cfa\_register %rbp

subq $16, %rsp

leaq L\_.str(%rip), %rdi

movl $0, -4(%rbp)

callq \_puts

xorl %ecx, %ecx

movl %eax, -8(%rbp) ## 4-byte Spill

movl %ecx, %eax

addq $16, %rsp

popq %rbp

retq

.cfi\_endproc

.section \_\_TEXT,\_\_cstring,cstring\_literals

L\_.str: ## @.str

.asciz "Hello, World!"

.subsections\_via\_symbols

**assembly**

During the assembly stage, an assembler is used to translate the assembly instructions to machine code, or *object code*. The output consists of actual instructions to be run by the target processor.

To save the result of the assembly stage, pass the -c option to cc:

cc -c hello\_world.c

Running the above command will create a file named hello\_world.o, containing the object code of the program. The contents of this file is in a binary format and can be inspected using hexdump or od by running either one of the following commands:

hexdump hello\_world.o

od -c hello\_world.o

**linking**

The object code generated in the assembly stage is composed of machine instructions that the processor understands but some pieces of the program are out of order or missing. To produce an executable program, the existing pieces have to be rearranged and the missing ones filled in. This process is called linking.

The *linker* will arrange the pieces of object code so that functions in some pieces can successfully call functions in other pieces. It will also add pieces containing the instructions for library functions used by the program. In the case of the “Hello, World!” program, the linker will add the object code for the puts function.

The result of this stage is the final executable program. When run without options, cc will name this file a.out. To name the file something else, pass the -ooption to cc:

cc -o hello\_world hello\_world.c