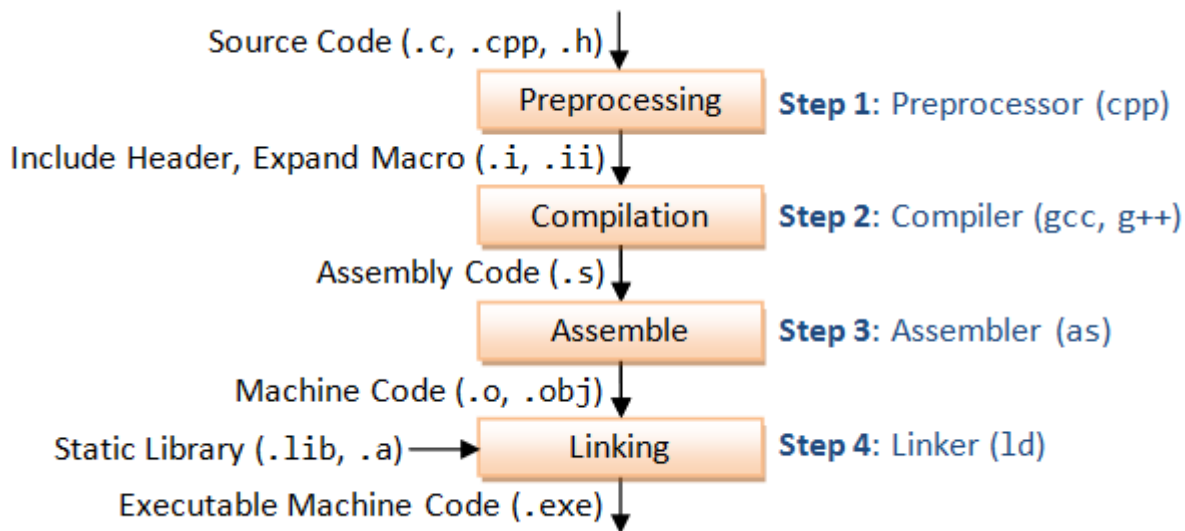


STAGES OF COMPILATION

Compiling a C program is a multi-stage process. At an overview level, the process can be split into four separate stages: *Preprocessing*, *compilation*, *assembly*, and *linking*. Traditional C compilers orchestrate this process by invoking other programs to handle each stage.



I will walk through each of the four stages of compiling the following C program:

Hello_world.c

```
#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 the hello_world.c file, stripped free from its leading comment:

```
extern int __vsprintf_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 support 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.

```
.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 content 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 must 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 -o option to cc:
cc -o hello_world hello_world.c

There are several sections that are common to all executable formats (may be named differently, depending on the compiler/linker) as listed below:

Segments in Executable File:

Section	Description
.text	This section contains the executable instruction codes and is shared among every process running the same binary. This section usually has READ and EXECUTE permissions only. This section is the one most affected by optimization.
.bss	BSS stands for ‘Block Started by Symbol’. It holds un-initialized global and static variables. Since the BSS only holds variables that do not have any values yet, it does not actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data section) does not take up any actual space in the object file.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.
.rdata	Also known as .rodata (read-only data) section. This contains constants and string literals.
.reloc	Stores the information required for relocating the image while loading.
Symbol table	A symbol is basically a name and an address. Symbol table holds information needed to locate and relocate a program’s symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The symbol table contains an array of symbol entries.

Relocation records	Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Re-locatable files must have relocation entries' which are necessary because they contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Simply said relocation records are information used by the linker to adjust section contents.
--------------------	---