

The Compiler and Toolchain

CSE 220: Systems Programming

Ethan Blanton & Carl Alphonc

Department of Computer Science and Engineering

University at Buffalo



The C Toolchain

The **C compiler** as we know it is actually **many tools**.

This is due to:

- C's particular history
- Common compiler design
- The specific design goal of compilation in parts

What we actually invoke is the **compiler driver**.

The **compiler** is only a single step of the multi-step process!

Compiling a C Program

A C program consists of one or more source files.

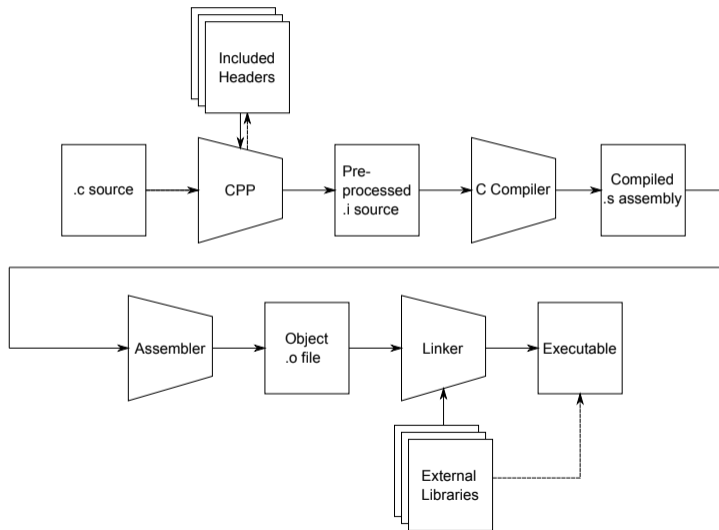
The C compiler driver passes the source code through several stages to translate it into machine code.

A source file¹ is sometimes called a translation unit.

Each stage may be invoked individually ... more later.

¹Plus some other stuff

The Complete Toolchain



Development: Writing the Code

After your planning, **start writing code**.

Don't plan forever!

Find something that you can write **right now**:

- Consult your notes on the documentation
- Look through your pseudocode

One line of code is all it takes to get started!

Spend no more than about **10% of your time** planning up front.

The C Compiler Driver

First, we will ignore most stages of compilation.

The C compiler driver can take a .c source file and produce an executable directly.

We'll look at that with Hello World:

```
#include <stdio.h>
```

```
int main(int argc, char *argv[]) {  
    printf("Hello, world!\n");  
    return 0;  
}
```

Compiling Hello World

We compile Hello World as follows:

```
gcc -Wall -Werror -O2 -g -std=c99 -o helloworld helloworld.c
```

This command says:

- `-Wall`: Turn on all warnings
- `-Werror`: Treat all warnings as errors
- `-O2`: Turn on moderate optimization
- `-g`: Include debugging information
- `-std=c99`: Use the 1999 ISO C Standard
- `-o helloworld`: Call the output helloworld
- `helloworld.c`: Compile the file helloworld.c

Compiling Hello World II

The C compiler driver ran **all of the steps necessary to build an executable** for us.

- The **C preprocessor** handled including a header
- The **compiler** produced assembly
- The **assembler** produced object code
- The **linker** produced helloworld

```
[elb@westruun]~/.../posix$ ./helloworld  
Hello, world!
```


Compiling in Steps

The compiler driver can be used to invoke **each step** of the compilation individually.

It can also be used to invoke **up to** a step.

The **starting step** is determined by the **input filename**.

The **ending step** is determined by **compiler options**.

We will explore each step in some detail.

The C Preprocessor

The **preprocessor** does just what it sounds like.

It performs certain **source code** transformations **before the C is processed by the compiler**.

It **doesn't understand C**, and can be used for other things!

Functions of the Preprocessor

The C preprocessor applies **preprocessor directives** and **macros** to a source file, and removes **comments**.

Directives **begin with #**.

- **#include**: (Preprocess and) insert another file
- **#define**: Define a symbol or macro
- **#ifdef/#endif**: Include the enclosed block only if a symbol is defined
- **#if/#endif**: Include only if a condition is true
- ...

Preprocessor directives **end with the current line** (not a semicolon).

Including headers

The `#include` directive is primarily used to incorporate headers.

There are two syntaxes for inclusion:

- `#include <file>`
Include a file from the system include path (defined by the toolchain)
- `#include "file"`
Include a file from the current directory

Defining Symbols and Macros

The `#define` directive defines a symbol or macro:

```
#define PI 3.14159
```

```
#define PLUSONE(x) (x + 1)
```

```
PLUSONE(PI) /* Becomes (3.14159 + 1) */
```

Macros are **expanded**, not calculated!

The expansion will be given directly to the compiler.

Conditional Compilation

The various `#if` directives control [conditional compilation](#).

```
#ifdef ARGUMENT
```

```
/* This code will be included only if ARGUMENT is  
   a symbol defined by the preprocessor --  
   regardless of its expansion */
```

```
#endif
```

The `#ifndef` directive requires ARGUMENT to be [undefined](#).

The `#if` directive requires ARGUMENT to [evaluate to true](#).

Using the Preprocessor

The preprocessor can be invoked as `gcc -E`.

Using the preprocessor `correctly` and `safely` is tricky.

For now, it is best to limit your use of the preprocessor.

`Do` use it for `debugging`, though!

The C Compiler

The **compiler** transforms C into machine-dependent **assembly code**.

It produces an **object file** via **the assembler**.

The compiler is the **only part** of the toolchain that **understands C**.

It understands:

- The **semantics of C**
- The **capabilities of the machine**

It uses these things to **transform C into assembly**.

Assembly Language

Assembly language is **machine-specific**, but **human-readable**.

Assembly language contains:

- Descriptions of **machine instructions**
- Descriptions of **data**
- **Address labels** marking variables and functions (**symbols**)
- Metadata about the code and **compiler transformations**

All of the **semantics** of the C program are in the assembly.

The **structure** of the assembly may be very different!

Compiling to Assembly

Let's compile **to assembly** using `-S`:

```
$ gcc -Wall -Werror -O2 -std=c99 -S helloworld.c
```

On the next slides, we'll examine the output from `helloworld.s`.

helloworld.s |

```
.file "helloworld.c"
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "Hello, world!"
.section .text.startup,"ax",@progbits
.p2align 4,,15
.globl main
.type main, @function
```

We'll get to the details later, but for now notice:

- `.LC0:` is a **local label**
- `.string` declares a **string constant** (no newline!)
- The `.globl` and `.type` directives declare that we're defining a **global function** named `main`

helloworld.s II

```
main:
.LFB11:
    .cfi_startproc
    leaq    .LC0(%rip), %rdi
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    call    puts@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
```

We'll [skip the postamble](#), for now.

The Generated Code

First of all, **you aren't expected to understand the assembly.**

```
leaq    .LC0(%rip), %rdi
```

This code **loads the string constant's address** (from .LC0).

Then, later:

```
call    puts@PLT
```

...it calls puts() to output the string.

Note that the C compiler:

- Noticed we were outputting a **static string**
- Noticed it ended in a newline
- Replaced printf() with puts() *and a modified string*

The Assembler

The **assembler** transforms **assembly language** into **machine code**.

Machine code is **binary instructions understood by the processor**.

The output of the assembler is **object files**.

An **object file** contains:

- Machine code
- Data
- Metadata about the **structure** of the code and data

Compiling to an Object File

You may wish to compile [to an object file](#).

This is used when [multiple source files](#) will be linked.

In this case, use `-c`:

```
$ gcc -Wall -Werror -O2 -std=c99 -c helloworld.c
```

This will produce `helloworld.o`.

The Linker

The **linker** turns one or more **object files** into an **executable**.

An **executable** is:

- The **machine code and data** from object files
- Metadata used by the OS to run a complete program

An executable's metadata includes:

- The platform on which it runs
- The **entry point** (where it should start execution)
- Anything it requires from libraries, *etc.*

Linking

Compiling **any input files** without an **explicit output stage** will invoke the linker.

```
gcc -Wall -Werror -O2 -std=c99 -o helloworld helloworld.o
```

This command will **link** `helloworld.o` with **the system libraries** to produce `helloworld`.

You can **view the linkage** with `ldd`:

```
[elb@westruun]~/../posix$ ldd helloworld
linux-vdso.so.1 (0x00007ffe34d1a000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f24dacbb
/lib64/ld-linux-x86-64.so.2 (0x00007f24db25c000)
```

Summary

- The “C compiler” is actually a **chain of tools**
 - We invoke the **compiler driver**
 - The **preprocessor** transforms the **source code**
 - The **compiler** turns C into **assembly language**
 - The **assembler** turns assembly language into **machine code** in **object files**
 - The **linker** links object files into an **executable**

References I

Required Readings

- [1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 1: Intro, 1.1-1.4. Pearson, 2016.

License

Copyright 2019, 2020, 2021, 2022 Ethan Blanton, All Rights Reserved.

Copyright 2022 Carl Alphonse, All Rights Reserved.

Copyright 2019 Karthik Dantu, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.