The Compiler and Toolchain

CSE 220: Systems Programming

Ethan Blanton & Carl Alphonce

Department of Computer Science and Engineering

University at Buffalo

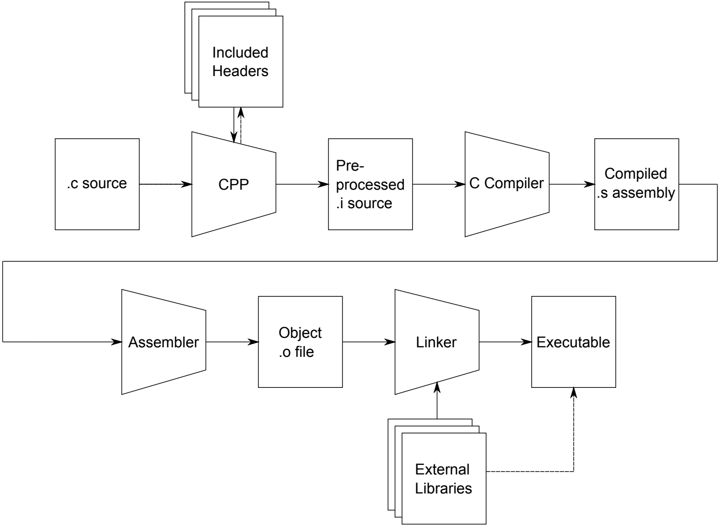
The C Toolchain

The C compiler as we know it is actually a driver for a chain of tools; it is sometimes referred to as a compiler driver, which invokes the following tools:

- The preprocessor transforms the source code into C code
- The compiler turns C code into assembly code
- The assembler turns assembly code into machine code in object files
- The linker links object files into an executable file

Notice that the compiler is only a single step of the multi-step process!

The Complete Toolchain



An example

We’ll explore the compilation process using Hello World as an example:

```
#include <stdio.h>

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

Compiling Hello World, part I

Consider the following gcc invocation to compile Hello World:

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

This command passes many command-line arguments to gcc:

- -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, part 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`, an executable file

```
$ ./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 C preprocessor applies preprocessor directives and macros to a source file, and removes comments. The output of the preprocessor is valid C code, and is the input to the actual C compiler.

Preprocessor 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
- ...

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

Using the Preprocessor

The preprocessor can be invoked as gcc -E.

Using the preprocessor correctly and safely is tricky.

In the make lab we showed you how to use it for debugging.

The C Compiler

The compiler transforms C code into assembly code.

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 language.

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

The semantics of C code are preserved in the translation to assembly code.

The structure of the assembly code may be vastly different from that of the original C code!

Compiling to Assembly

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

```
$ gcc -fno-asynchronous-unwind-tables -std=c99 -S helloworld.c
```

The `-fno-asynchronous-unwind-tables` option excludes some meta information from the output.

Excluding this information makes the assembly code easier to read.

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

helloworld.s I

```
        .file   "helloworld.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello, world!"
        .text
        .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
- The `.globl` and `.type` directives declare that we're defining a global function named `main`

helloworld.s II

```
main:
        endbr64
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $0, %eax
        leave
        ret
```

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.

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`, as in:

```
$ gcc -Wall -Werror -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 -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`:

```
$ 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

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