Tools of the Trade – Part I: Compiler, Linker, Build System

Computational Science II (CAAM 520)

Christopher Thiele

Rice University, Spring 2020

Overview

Now that we can write C code, how do we transform it into a program?

- Compiler: Translates C code to machine code.
- Linker: Transforms machine code to a binary, e.g., for Linux.
- Build system: Manages the compiler and linker (among other things).

Compiler

While there are countless C compilers available, we will focus on GCC.

- Can be considered the default on GNU/Linux.
- Most other compilers for Linux/Unix are largely compatible (e.g., Clang, Intel Compiler).

Compiler

To compile a single source file `test.c` in to an ***executable*** or ***binary*** called `test`, we simply run

```
gcc -o test test.c
```

Compiler

Use the compiler to your advantage!

The compiler can help you with a lot of warnings if you write code that is not robust and potentially faulty:

```
gcc -Wall -Wextra -Wpedantic -o test test.c
```

Compiler

So far we compiled and linked in one step! Let’s look at both steps separately.

If we only want to compile, i.e., translate C code into machine code, we run

```
gcc -c -o test.o test.c
```

$\longrightarrow$ Produces an ***object file***.

Linker

To create an executable from one or many object files, we run

```
gcc -o test test1.o test2.o
```

$\rightarrow$ Linking

Linker

The linker also includes ***libraries*** and third-party code.

```
gcc -o test test.o -lm
```

→ The flag `-lm` tells the linker to link (l) against the math library (m), which contains functions like `exp()`.

Linker

Why would we want to separate compilation and linking?

Compiling everything at once might be undesirable or impossible:

- If we have 100 source files and there is an error in one of them, the compilation of all 100 has to be repeated.
- If we compile files separately, we can do so in parallel.

Build system

Who has 100 source files?

→ Linux kernel has more than 10,000,000 lines of code.

Does Linus Torvalds type

```
gcc -c -o file.o file.c
```

hundreds of times?

→ No, he uses a build system.

Build system

We will use GNU Make as a build system.

It allows us to compile and link an application accorting to a recipe, a so-called ***Makefile***:

```
# Simplest structure of a target in a Makefile:
target: dependencies
  things_to_do
```

Build system

A Makefile to compile a program from two source files:

```
my_application: file1.c file2.c
# Simplest structure of a target in a Makefile:
  # Indent with *tabs*, not with spaces!
  gcc -o my_application file1.c file2.c
```

Build system

Makefiles can automate much more complex build processes:

```
SRC := $(wildcard *.c)
OBJ := $(patsubst %.c, %.o, $(SRC))

my_application: $(OBJ)
  gcc -o $@ $^

%.o: %.c
  gcc -c -o $@ $<
```

→ Unfortunately, the syntax is horrible.

Build system

Why do we need Makefiles in class?

To ensure that your code can be built without knowing the details!

```
# Student 1
hw1: file.c
  gcc -o hw1 file.c
```

```
# Student 2
hw1: file1.cc file2.cc
  g++ -std=c++14 -o hw1 file1.cc file2.cc -lm
```

Build system

Note that (GNU) Make has its limitations:

- Horrible syntax
- First appeared in 1976
- Targets Unix/Linux

$\longrightarrow$ For more complex tasks, consider using a meta build system like CMake.