Building C Programs with Makefiles

CS14 - S26

Building

The process of assembling all the dependencies of a project and compiling the final program(s) is called **building**.

In general, this is a complex process, and many different pieces of software have been written to try to automate the process for different programming languages and operating systems.

We will study a simple build automation process using **Makefiles** for C program compilation.

First, we need to understand how C programs are compiled.

Intro to C compilation

C programs are prepared to run in 4 or 5 steps.

1. Preprocessing
2. Compilation
3. Assembly
4. Static Linking
5. Runtime Dynamic Linking (if necessary)

1. Preprocessing

Directives, which are commands that start with a “#” are resolved and replaced with code if necessary. E.g.,

```
#include <stdio.h>

#define MAX 1000000

#pragma once
```

2. Compilation

The C code is converted to human readable assembly language, typically in a .s file.

The assembly code is platform dependent instructions for how a processor should execute a program.

Take CS31 for more details!

3. Assembly

The assembly code is “assembled” into binary machine code, typically in .o “object” files.

Take CS31 for more details!

4. Static Linking

All of the .o object files, along with any code in .a library files and .so shared object files is linked together to produce an executable file.

This process is called static because all of the behavior in the .o object files and .a library files is stored in the executable.

5. Runtime Dynamic Linking

.so shared object files are libraries that are loaded into memory once, but can be used simultaneously by multiple programs, as opposed to .a library files which are compiled into each program that uses them.

So, if you compiled an executable with a .so file, that file must be present when you execute the program and will be linked “dynamically” when the program is run.

Using .so files allows us to save memory for commonly used libraries.

C compilation with gcc

`gcc` is a commonly used C compiler.

The simplest way to use it is to just provide the name of a file containing a C main function.

E.g., `gcc foo.c` would produce an executable called `a.out`

Understanding the high level steps of compilation allows us to understand common `gcc` flags.

Common gcc flags

`-o` – specify the output file name

`-W` – specify a set of warnings that should be reported about possible issues with code (e.g. -Wall)

`-g` – include information for debuggers when compiling

`-I` – specify path(s) to search for header files to include (during the preprocessing step)

`-L` – specify path(s) to search for libraries to link (during the linking step)

`-l` – specify the name of a library to link

Example gcc invocation with the above options:

```
gcc  -I/home/me/include -o myprog myprog.c -L/home/me/lib -lexamplelib
```

`gcc` commands are long and error-prone to write, especially for larger projects, and especially if you have to write them multiple times.

`make` and `Makefile`s help solve this problem by automatically generating gcc commands based on predefined rules.

make and Makefiles

The `make` command attempts to interpret and run commands based on the `Makefile` in the directory where `make` is run.

A Makefile typically contains some variable definitions, followed by one or more “rules”.

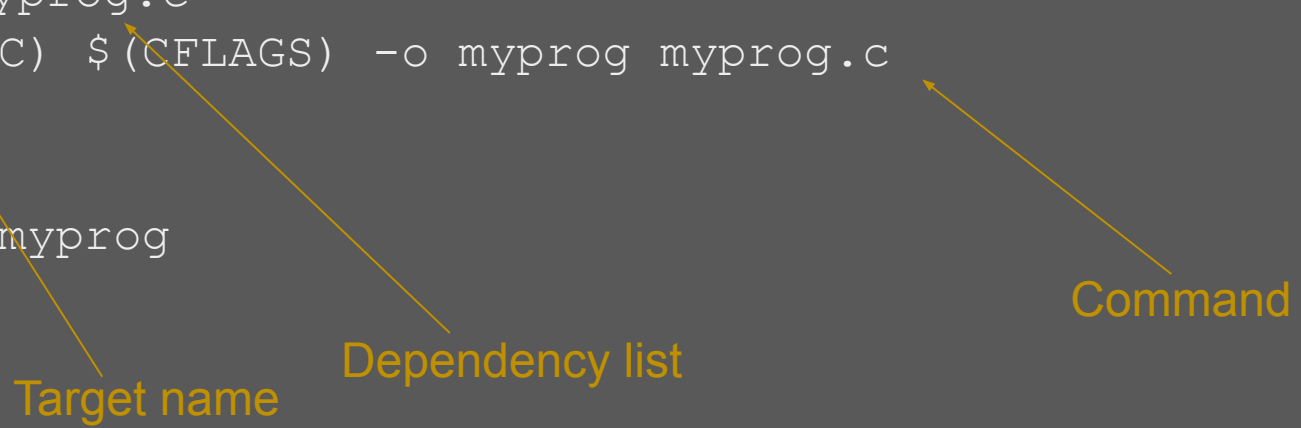
Rules are composed of a short “target” name, a command to be run, and what files that command depends on.

```
# Comments start with a # symbol

# Here are variable definitions
CC = gcc
CFLAGS = -g -Wall

# Here is a rule. The first target in a Makefile is what
# gets executed when you run make with no arguments.
all: myprog.c
    $(CC) $(CFLAGS) -o myprog myprog.c

clean:
    rm myprog
```



Notes on Makefiles

Whenever you run `make`, it will check if any files listed in target dependencies have been modified more recently than the target itself. If so, it will re-run those targets (and only those targets).

`Makefile` syntax is sensitive to whitespace. In particular, the command for a target must be indented with a tab. Check that your text editor inserts a literal tab rather than spaces!

It is often easiest to develop new `Makefile`s based on previous ones. Check Dive into Systems for good examples!

Metabuild systems

On large projects, Makefiles alone are usually difficult to manage.

More commonly, programs that *generate* Makefiles and configurations are used.

E.g.:

- Cmake
- Meson
- Ant (for Java)
- Maven (also typically for Java)