CSE 1320 - Intermediate Programming

Make - A Build System

Alex Dillhoff

University of Texas at Arlington

Header Files

Building programs in C will inevitably result in many code files, modules, and subsystems. It is important to understand and

implement best practices when creating and organizing larger code bases.

Build Systems

As our C projects grow in scope, the rules and commands used to compile the program becomes complex.

This is especially true when we consider how the project's dependencies change based on the target platform.

Build Systems

For small projects, the compilation rules are relatively sample.

All of our examples in this course have been something along the lines of:

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

Build Systems

As our projects grow, so do the number of files, libraries, and other dependencies that it relies on.

There are a number of build systems to help organize C projects such as:

- Make
- Ninja
- CMake
- Bazel

Makefiles

In this course, we look at Make, a popular build system that comes pre-packaged on most UNIX systems.

The organization of the build is defined in what is called a **Makefile**.

Makefiles

What is a **Makefile**?

Essentially, it is a text file that defines how a project is configured, compiled, installed, and cleaned.

Makefile Basics

A Makefile is made up of rules in the following format:

```
targets: dependencies
    command1
    command2
    ...
    commandn
```

Makefile Basics

The `targets` specify a named rule.

For example, a rule to compile a basic program is written:

```
project:
    gcc project.c -o project
```

Makefile Basics

Multiple targets can be added depending on the requirements (testing, debugging, release).

```
project:
    gcc project.c -o project

test:
    gcc tests.c project.c -o test
```

Makefile Basics

It is common to add a `clean` target to remove unnecessary output.

```
project:
    gcc project.c -o project

clean:
    rm -f project
```

Dependencies

We can specify the rules for each file or group of files based on patterns.

We can also define the dependencies.

If a file is dependent on others, it will look for the rules of the dependencies and compile them first.

Dependencies

Consider a project with two C files:

- `program.c`
- `utils.c`

The `main` function is in `program.c` and uses functions defined in `utils.c`.

Dependencies

We can set up these dependencies with the following `make` targets.

```
all: program.o utils.o

program.o: program.c

utils.o: utils.c
```

Dependencies

```
utils.o: utils.c
```

First, `utils.c` will compile into an object file named `utils.o`.

Dependencies

```
program.o: program.c
```

Next, `program.c` will compile into an object file named `program.o`.

Dependencies

If the makefile were to be called again at this point, the object files would only recompile if the original C files have been modified.

This is extremely useful when dealing with large projects in C.

Dependencies

```
all: program.o utils.o
```

This is the final target, which is dependent on the object files defined previously.

If `make` is called in the terminal, this will execute and resolve all defined dependencies until the target is complete.

Example Version 1

Let's put this together into a slightly larger example.

`https://github.com/ajdillhoff/CSE1320-Examples/tree/main/make/user_db`

Example Version 2

This `Makefile` works, but it leaves a lot of clutter in our project directory.

We can organize this by having our `Makefile` include a build directory where all of the build objects will go.

PHONY targets

With the second version, we are able to build our project into a specific directory which keeps the project directory clean.

It is common to include a PHONY target that performs some command without creating a specific file.

PHONY targets

These are commonly used to include commands to `clean` our project directory and `run` our projects from scratch.

Example Version 3

Let's modify our `Makefile` to include both of these targets.

Streamlining the `Makefile`

Using the approaches we have looked at so far, we have to edit our `Makefile` every time we add more files.

As our project grows, this task becomes more tedious and error-prone.

Variables

It is possible to include variables in a Makefile.

This is helpful when organizing large projects with different groups of files.

```
CC = gcc
util_files = util1.c util2.c
project_files = project1.c project2.c
program = project_name

project: $(util_files) $(project_files)
    $(CC) $(utils_files) $(project_files) -o $(program)
```

Variables

Want to include all files of a certain type in your target? **Use a wildcard.**

```
# matches all files ending in .c
files = $(wildcard *.c)

project: $(files)
    gcc $(files)
```

Variables

CAUTION: Using `*.c` by itself is NOT the same as using the `wildcard` prefix.

If `*.c` alone does not match any files, it is left as-is.

```
# matches all files ending in .c
files = $(wildcard *.c)
maybe_files = *.c # not the same thing
```

Automatic Variables

Depending on your configuration, the target and prerequisite names might be variable. In that case, it is impossible to write them out directly.

Automatic variables are provided for each rule, based on the target and prerequisites.

Automatic Variables

Here are some of the more frequently used *automatic variables*.

- `$@` - Returns the file name of the target of the rule.
- `$^` - Returns the names of all prerequisites.
- `$?` - Returns the names of all prerequisites that are newer than the target.

Automatic Variables

There are a number of automatic variables provided for convenience.

A full list is available here:

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

The `all` target

It is possible to make all targets using the command `make all`.

It is common to place all targets under this singular target.

```
all: release debug test

release:
    gcc $(release_files) -o $(release_name)

debug:
    gcc $(debug_files) $(debug_flags)

test:
    gcc $(test_files)
```

Implicit Rules

Implicit rules are those that happen as part of an explicit rule.

For example, when compile a program, an implicit rule is that it is first converted to an object file.

There are some variables that can be modified which affect these implicit rules.

Implicit Rules

Here are the most important variables to consider for C programs:

- `CC` - The specific program for compiling C programs (default: `cc`).
- `CFLAGS` - Flags given to the C compiler.
- `CPPFLAGS` - Flags given to the C preprocessor.
- `LDFLAGS` - Linker flags.

Implicit Rules

Example of setting and using the flags implicitly.

```
CC = gcc
CFLAGS = -lm -g

# These will compile using implicit results,
# as well as the flags and compiler set above.
program: program.o
program.o: program.c utils.c
```

Make

There are many more features available as part of `Make`.

An excellent, extended tutorial can be found here:
https://makefiletutorial.com/

Example Version 4

Let's combine all of these tools to make a simple, yet effective, `Makefile`

Example: Build a project with `make`