makefiles

A guide to Makefiles. Designed to be useful for students I TA at the University of British Columbia (UBC), but also hopefully more broadly applicable to other developers. (I've read a lot of Makefile tutorials, and most of them are a bit confusing. So, hopefully this rendition is a bit clearly for undergraduates.) The content here is mostly just a more accessible version of Chapter 2 of the `make` manual, "An Introduction to Makefiles," with some extra tidbits here and there.

Now, without further ado,

A guide to GNU make

Table of Contents

Motivations

`make` is a command line program that, in tandem with a `Makefile`, allows one to 'compile' and 'link' small, medium, and large programs. If you've ever compiled a program, like

```
gcc main.c -o main; ./main
```

or perhaps

```
javac Main.java; java -cp . Main
```

or even

```
clang++ main.cpp -o main; ./main
```

you'll benefit from `make`. `make` can be used to add "shortcuts" for various common commands as well, such as cleaning up undesired files from your project. It also *only recompiles what is necessary* which, while never particularly relevant for assignments consisting of less than 1,000 lines of code, becomes extraordinarily relevant in codebases of 1,000,000+ lines.

Makefile Basics

Firstly, make sure you have `make` actually installed. Note this is not the same thing as `cmake`. (I'm not proud to admit it, but I thought these were equivalents for a very long time.) You should also have `gcc` working (and ideally some kind

of Java installed). Again, new-to-C Windows users might have issues here; I recommend using Chocolatey to fix those issues (or better yet, WSL).

> Aside: `cmake` is a meta-`make` (that is, like `make` for `make`). It builds makefiles for various systems simultaneously. It has 'profiles' for different compilers and systems, and will automatically generate the correct flags, et cetera, on the fly. In short, it's for *huge* projects, and I won't be covering it in this guide.

To check if you have `make` installed, run `make --version`. I see `GNU Make 4.2.1 [...]` on my machine. If you see an error like,

> " 'make' is not recognized as an internal or external command, operable program or batch file" (from PowerShell, on Windows.)

or

> "make": command not found (from Linux, I believe.)

you need to install the software. Follow these steps; on Windows, `choco` is your friend here.

Try running `make` with no arguments. You should see the following.

```
$ make
> make: *** No targets specified and no makefile found.  Stop.
```

`make` is looking for a corresponding `Makefile` and didn't find one. Let's start by creating one!

Makefiles are essential a set of different commands or "rules." They have **no** file extension, and are just written in plaintext. Each rule has roughly the following structure.

```
target ...: prerequisites ...
    recipe
    ...
```

The **target** is the name of the file generated by the program. A **prerequisite** is dependency for the target, which should be another target itself. The **recipe** is the actual set of actions carried out. The target and recipe are required, but prerequisites are of course optional.

WARNING: Makefiles are picky about spacing, like Python. Each line in a recipe must be prefixed by a `TAB` character. Four spaces don't work; it really needs to be tabs. Just a note.

Let's go back to the basic `gcc main.c -o main` we all know and love. If you check out `examples/basic`, we can see what this looks like in a Makefile. Begin by trying

```
$ gcc main.c -o main; ./main
> H3llo, w0rld!
```

which works as normal. Now run

```
$ make; ./main
> H3llo, w0rld!
```

our first Makefile! And it's really just one line:

```
main: main.c
```

kind-of cool isn't it? Though it's a bit unorthodox, we could even do this for a Java file. Running the `examples/java` file in VS Code gave some horrible command like

```
$ /usr/bin/env /home/mdema/.sdkman/candidates/java/current/bin/java \
    --enable-preview \
    -XX:+ShowCodeDetailsInExceptionMessages \
    -Dfile.encoding=UTF-8 \
    -cp /home/mdema/.vscode-server/data/[...]/bin \
    examples.java.Main
> Hello, world!
```

Imagine trying to memorize that! Instead, try

```
$ make
> Hello, world!
```

(Forgive the `cd`s in the Makefile; this is just for demonstration.) Clearly, `make` *can* be used to run any set of commands. Rules are at their core, just a set of instructions, a "recipe", that are just commands you would've had to run by hand in the shell.

More on Makefiles

Oftentimes, our programs are not just single files. What if we want to compile a bunch of `.c` files and `.h` files into a single executable? This is where object files (`.o`) can truly benefit us. We can create rules to construct intermediate `.o` files, and then create a "master" rule that combines (or better yet, links) a bunch of these object files together.

In sum, the template is something like this

```
main: something.o something_else.o something_else_other.o
    gcc -o main something.o \
      something_else.o \
      something_else_other.o

something.o: something.c something.h defs.h
    gcc -c something.c

something_else.o: something_else.c defs.h
    gcc -c something_else.c
```

```
something_else_other.o: something_else_other.c constants.h defs.h
    gcc -c something_else_other.c
```

To create `main`, you would type `make` (or `make main`). Notice how `main` is just that, and not suffixed with a `.o`. We also use `-o` instead of `-c` to create an object file instead of an object file. This example nicely demonstrates the importance of prerequisites; if we edit `constants.h`, recompiling main will only recompile `something_else_other.o` before linkage, saving some computation time.

The first rule given is used by default, so it should be the "main" goal or executable produced by your program.

> A few asides from the above example. Firstly, note that `make` itself is not picky about what goes in a recipe, and it's your task as the programmer to devise a recipe that really works. Second, `gcc` is the GNU C compiler; you can see a brief guide to different compilers here. `cc` is a more generally applicable option.

It can be a bit tedious re-writing all these file names. Instead, it's common practice to throw them into a variable. The above example becomes.

```
OBJECTS = something.o something_else.o something_else_other.o

main: $(OBJECTS)
    gcc -o main $(OBJECTS)

# etc.
```

You'll also notice that we're manually stating `gcc -o main ...` in each of these, but we did not do that in our basic example. In fact, `make` has an implicit rule to use `cc -c` to compile object files from `.c` files. It's pretty nifty. Likewise, `make` also allows us to skip adding the `.c` file itself to its object file prerequisite. The above example simplifies to

```
OBJECTS = something.o something_else.o something_else_other.o

main: $(OBJECTS)
    gcc -o main $(OBJECTS)

something.o: something.h defs.h
something_else.o: defs.h
something_else_other.o: constants.h defs.h
```

Notice how `defs.h` is a pre-requisite to all of our object files? There's more room for improvement here. We can group entries by their pre-requisites instead. This yields

```
OBJECTS = something.o something_else.o something_else_other.o
```

```
main: $(OBJECTS)
    gcc -o main $(OBJECTS)

$(OBJECTS): defs.h
something.o: something.h
something_else_other.o: constants.h
```

Last thing before we wrap up—running these commands will leave a lot of unwanted files hanging around. To fix this, let's add a "phony" command called `clean` that will delete junk files. The reason it is called "phony" is that we don't have a file `clean.c`, but rather we're looking to just automate some command line task.

We get

```
OBJECTS = something.o something_else.o something_else_other.o

main: $(OBJECTS)
    gcc -o main $(OBJECTS)

$(OBJECTS): defs.h
something.o: something.h
something_else_other.o: constants.h

.PHONY: clean

clean:
    rm main $(OBJECTS)
```

Fancy Makefiles

What if we want to add some command line options to our `gcc` call? For example, running `gcc main.c -std=gnu11 -Wall -o main`? To do this, we just parameterize the `gcc` part of a recipe. Continuing our earlier example,

```
CC = gcc
CFLAGS = -I . -std=gnu11 -Wall

objects = something.o something_else.o something_else_other.o

main: $(objects)
    $(CC) -o main $(objects)

# etc.
```

Notice that we must specify the `$(CC)` and the `CFLAGS` is added automatically by the compiler.

Remember the earlier example where we wanted to pull out `defs.h` from each

of our pre-requisites? There's a different, arguably better way of doing this that involves using macros. We can also create "generic" rules that apply to globs (e.g., all `*.c` files). This example demonstrates this

```
CC = gcc
CFLAGS = -I . -std=gnu11 -Wall

OBJECTS = something.o something_else.o something_else_other.o
DEPS = defs.h

%.o: %.c $(DEPS)
    $(CC) -c $(CFLAGS)

main: $(OBJECTS)
    $(CC) -o main $(OBJECTS)

# etc.
```

This fancy `%.o` rule says all object files depend on their respective `.c` file along with all of the `DEPS`.

For a final step, let's add directories. What if our `.h` files are in an `include/` directory, our source code lives in a `src/` folder, we have local libraries in a `lib/` folder, and we want to stuff our object files into an output directory. Here's what that would look like.

```
IDIR = ../include
ODIR = obj
LDIR = ../lib

CC = gcc
CFLAGS = -I $(IDIR) -std=gnu11 -Wall

LIBS = -lm

_DEPS = main.h
DEPS = $(patsubst %, $(IDIR)/%, $(_DEPS))

_OBJ = main.o other.o
OBJ = $(patsubst %, $(ODIR)/%, $(_OBJ))


$(ODIR)/%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

main: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS) $(LIBS)
```

```
.PHONY: clean

clean: # core is for core dumps!
    rm -f $(ODIR)/*.o *~ core $(IDIR)/*~
```

There are a few extra oddities in this snippet. There are a few automatic variables being used. `$@` is the name specified on the left of the colon. `$<` is the first item in the dependencies list. `$^` is the names of all of the pre-requisites.

`patsubst` is a helper function to replace words matching a specified pattern with some text. "%" acts as a wildcard character.

With the above information and the knowledge in the rest of this guide, try to break down what happens above for yourself as an exercise!

This file comes courtesy of Bruce Maxwell of Colby College. If you try to re-use it, you may need to tweak the file paths a bit!

That's all for this Makefiles guide. Hope you enjoyed!

Glossary

- **compile**, meaning to transform source code into object files (e.g. running `gcc main.c`)
- **executable file**, a file that when run causes a computer to perform a set of instructions (e.g., the `main` in `gcc main.c -o main`)
- **link**, meaning to transform many object files (i.e., the things made in compilation) into a single executable file
- **machine code**, a (super) low-level programming language that controls a CPU
- **object files**, a file that contain object code, which is machine code that is output from a compiler or assembler (e.g., our `.o` files); they cannot be run in the command line

Further Reading

- GNU Make Homepage
- A Simple Makefile Tutorial
- isaacs/Makefile
- makefiletutorial.com
- Using make and writing Makefiles
- Tutorial on writing makefiles
- GNU `make`