UNIVERSITY of York

Introduction to Makefiles

Peter Hill

Introduction to makefiles

What are we trying to solve?

- Building software which consists of more than n files rapidly becomes tedious to do by hand
- Building software which includes any nature of logic in determining what/how to build rapidly becomes very complex
- Automation beats manual fiddling 9 times out of 10

Important corollary

- All build systems are terrible, but some also work

Simple example

Small C project

```
src/
 |- program.c
 |- foo.c
 \- foo.h
```

- C program with a “main” source file, and header/implementation files
- Easy to compile by hand:

  ```
  gcc -o program program.c foo.c
  ```

- Gets trickier with more files, more flags, compilation order, etc.

First thing

Shell script

```
#!/bin/bash
gcc -o program program.c foo.c
```

- Write a shell script with our compile commands in
- Works ok for simple projects
- Has some downsides:
  - Change one file and we have to recompile everything
  - Update how to compile one file and we need to recompile everything
  - Adding more files means either copy+pasting (with associated mistakes) or adding complex logic
  - We have all those CPU cores sitting idle while we compile one file at a time...

make

- Like most veteran programming tools, `make` was created in the 1970s
- Standard implementation on Linux is GNU `make`
- `make` is a *declarative* language, as opposed to *imperative*
- You tell `make` *how* to do something, and it figures *what* to do
- Basic form of a *rule* is:

```
target: prerequisites
    recipe
```

- Note: that's a literal **tab** before recipe

Advantages of `make`

- `make` will figure out what order things need doing in
- `make` will figure out what things actually need doing
- Change a file, and `make` will rebuild that file and everything that depends on, and no more
- These things are true of other build systems too!
  - Second most popular is probably CMake
  - Also whatever web developers are using this week

Simple example

Makefile

```
all:
    gcc -o program program.c foo.c
```

- Now we can just run `make` and build our program!
- By default, `make` builds the first target in the makefile
- By convention, the `all` target builds the whole program

Simple example

Makefile

- Let's get fancier:

```
all: program
program: program.o foo.o
    gcc -o program program.o foo.o
program.o: program.c
    gcc -c program.c
foo.o: foo.c
    gcc -c foo.c
```

- Now if we only change `program.c`, `foo.c` doesn't need to be recompiled
- `make program.o` will build just `program.o` and its prerequisites

Variables

Compile time configuration

- What if we sometimes want to compile with a different compiler?

```
# CC is the default name for the C compiler
CC = gcc
all: program
program: program.o foo.o
    $(CC) -o program program.o foo.o
program.o: program.c
    $(CC) -c program.c
foo.o: foo.c
    $(CC) -c foo.c
```

- Note: the default/conventional names for the C++ compiler is `CXX`, and `FC` for the Fortran compiler

Variables

Flags and libraries

```
CC = gcc
# CFLAGS for C compile flags, CXXFLAGS for C++, FFLAGS for Fortran
CFLAGS = -g -Wall
# LDLIBS for libraries, LDFLAGS for linker flags (i.e. -L)
LDLIBS = -lm
all: program
program: program.o foo.o
    $(CC) $(CFLAGS) -o program program.o foo.o $(LDLIBS)
program.o: program.c
    $(CC) $(CFLAGS) -c program.c
foo.o: foo.c
    $(CC) $(CFLAGS) -c foo.c
```

Cleaning up

The clean rule

- We often want to compile from a clean start
- The conventional target for this is `clean`:

```
.PHONY: clean
clean:
    rm -fv *.o *~
```

- We use `-f` so that `rm` doesn't error if a file doesn't exist (more important if you use a variable here)
- The `.PHONY` rule tells make that `clean` doesn't produce a file named `clean`

Adding more files

Patterns and automatic variables

- We're repeating ourselves a lot, specifying the source file in both the prerequisite and the actual recipe
- Also, all the recipes are identical, save for the filenames
- This is where *patterns* and *automatic variables* come in:

```
%.o: %.c
    $(CC) $(CFLAGS) -o $@ $^
```

- The % is a pattern: `foo.o` matches `%.o`
  - The part matching % is called the *stem* and gets expanded on both sides
  - If `foo.o` matches the target, then `%.c` expands to `foo.c`

Adding more files

Patterns and automatic variables

- We're repeating ourselves a lot, specifying the source file in both the prerequisite and the actual recipe
- Also, all the recipes are identical, save for the filenames
- This is where *patterns* and *automatic variables* come in:

```
%.o: %.c
	$(CC) $(CFLAGS) -o $@ $^
```

- `$@` and `$^` are automatic variables:
  - `$@` expands to the name of the *target*
  - `$^` expands to the names of all the *prerequisites*, separated by spaces

Adding more files

Back to our example

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

all: program
program: program.o foo.o
    $(CC) $(CFLAGS) -o $@ $^ $(LDLIBS)
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $^

clean:
    rm -fv *.o *~
```

Adding more files

- Now when we want to add a new source file to the program, it's a very simple change
- Let's also list all the files in a variable

Adding more files

Back to our example

```
CC = gcc
CFLAGS = -g -Wall
LDLIBS = -lm
OBJECTS = program.o foo.o bar.o

all: program
program: $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $^ $(LDLIBS)
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $^

clean:
    rm -fv *.o *~
```

Going further

Useful flags

- `--jobs=N` tries to run *N* recipes at once – very useful for larger projects!
- `--load-average=N` doesn't start new jobs if the load is more than *N*
- `--keep-going` tells `make` to keep going even if there are errors – useful for finding as many errors as possible in one go
- `--file=FILE` use *FILE* instead of the default `makefile`
- `--directory=DIRECTORY` change to *DIRECTORY* before doing anything
- `--dry-run` just print the recipes, instead of running them

Going further

Fancier features

- `make` is the standard *nix build system for projects large and small, and hence has *lots* of features
- More complicated features not covered here:
  - Implicit rules and variables
  - Searching other directories for prerequisites using `VPATH`, useful to place compilation artefacts in a different directory to the source
  - Lots of functions for transforming text
  - Conditionals:

```
# ARCHER specific compiler
ifeq ($(machine),archer)
FC = ftn
endif
```

Going further

Implicit rules and variables

- `make` already knows how to build certain types of files from other ones
- Also has lots of "implicit" variables
- Our example makefile could be as simple as:

```
program: program.o foo.o
    $(CC) $(CFLAGS) -o $@ $^ $(LDLIBS)
clean:
    rm -fv *.o *~
```

Going further

Automatically making Fortran dependencies

https://github.com/ZedThree/fort_depend.py

```
pip install --user fortdepend
```

```
all: $(actual_executable) my_project.dep

.PHONY: depend
depend: my_project.dep

my_project.dep: $(OBJECTS)
    fortdepend -w -o my_project.dep -f $(OBJECTS)

include my_project.dep
```

Going further

Automatically creating directories

```
OBJDIR := objdir
OBJS := $(addprefix $(OBJDIR)/,foo.o bar.o baz.o)

$(OBJDIR)/%.o : %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<

all: $(OBJS)
# The | signifies an "order-only" prerequisite
$(OBJS): | $(OBJDIR)

$(OBJDIR):
        mkdir $(OBJDIR)
```