Build Tools 1: Make (1)

lots of stuff we do on a computer is **reformatting documents or code**, and then turning them into something else or with code, compiling it.

- for source code → compilation
- take folders of coursework and compress them into zip files
- turning things into PDF etc

So the translation from current file format into target file format can be automated

You could use a shell script for this and stick all the tasks in one place, however there are better methods...

- say if our coursework has changed by our source code hasn't?
  - it would therefore be unnecessary to keep recompiling it

Can we make **build patterns** that correctly identify that not all tasks need to be run together all the time?

Make

- Make is the solution
- developed in 1976 by *Stuart Feldman* for automating builds
- **takes rules which tells you how to build files**

    - i.e. how to translate files into different formats
- Make then simply follows these rules

Two main versions of it:

1. **BSD Make** (old fashioned, POSIX compatiable, Mac OS comes with this as default)
2. **GNU Make** (more features, default for LINUX os)
    a. basically everyone uses GNU Make in practise however
    b. use `gmake` to use GNU Make

Makefiles

- rules for Make are placed into a Makefile:

```
hello: hello.c library.o
	cc -o hello hello.c library.o

library.o: library.c
	cc -c -o library.o library.c

coursework.zip: coursework
	zip -r coursework.zip coursework

flowchart.pdf: flowchart.dot
	dot -Tpdf flowchart.dot -O flowchart.pdf
```

- to build “hello” you need hello.c and library.o
    - to get hello from hello.c and library.o you run: `cc -o hello hello.c library.o`
- so how do you get library.o?
    - you need library.c and then run this command: `cc -c -o library.o library.c`

- for coursework.zip, if you have the folder “coursework” then you can get the corresponding zip file from it by running: `zip -r coursework.zip coursework`
- so on and so forth

When you instruct Make to build something it will look through your Makefile rules and dependancies, and it will do what it needs to do to build those files

- in the above example, compile the libaray first, and then link the library with hello.c and build the final output file:

```
make hello
cc -c -o library.o library.c
cc -o hello hello.c library.0
```

Making changes

- if you alter files, Make knows to only rerun the steps you need
- e.g. if you edit hello.c and rebuild:

```
$ make hello
cc -o hello hello.c library.o
```

- it that library.o comes from library.c, if library.o is *newer* than library.c then it knows that library.c couldnt of changed.
- however if library.c is new than library.o, i.e. the source file is newer than the output file, then Make will know it needs to redo the build with up to date files as the souce code has changed
- if you change hello.c but dont touch library.c then Make will know it doesnt need to rebuild library.o as it is up to date

---

Phony Targets

- as well as having rules on how to make files, you can also have **phony targets** which don't depend on files but instead **just tell Make what to do when they're run**
- Phony targets are not actual files; instead, they are just names for a recipe to be executed when explicitly requested.
- Makefiles often include a phoney:
- in Makefile you often have three rules:
    1. `all` - typically the first rule in a file, tells it to go and build **everything** (e.g. `make all`)
    2. `clean` - deletes all generated files (`make clean`)
    3. `install` - installs the program (`make install`)

Purpose: The main purpose of phony targets is to avoid conflicts with actual files of the same name and to ensure that the commands associated with them are always executed, regardless of the state of the filesystem. For instance, the `clean` command to remove all the object files and executables that the build process creates. If, by accident, a file named `clean` is created in the directory, `make` won't run the `clean` recipe unless it is declared as a phony target.

Declaring phony targets

```
.PHONY: all clean

all: hello coursework.zip flowchart.pdf

clean:
	git clean -dfx

hello: hello.c library.o
	cc -o hello hello.c library.o

library.o: library.c
	cc -c -o library.o library.c

coursework.zip: coursework
	zip -r coursework.zip coursework

flowchart.pdf: flowchart.dot
	dot -Tpdf flowchart.dot -O flowchart.
```

- this declares `all` and `clean` as phony targets
- in the above example, the `all` command just says that those output files are the only important ones
- `clean` using the command `git clean -dfx` just gets rid of files and directories (with the -dfx flag)

Pattern Rules

- the previous examples of file compilation have relied on the same number of files and libraries, say if we wanted to compile an extra library into our hello program
- We could update the Makefile, but it is better to **generalise it** in the first place
- these are where **pattern rules** come into play

```
CC=clang
CFLAGS=-Wall -O3

.PHONY: all clean

all: hello coursework.zip flowchart.pdf
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

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

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

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

Pattern rule syntax

in the above example:

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

- the `%.o` isbasically saying: to create a file that is followed by .o : you need a something followed by .c (`%.c`)
- using those, you can say use what ever is stored in the CC variable `$(CC)` (i.e. the C compiler `CC=clang`) with the CFLAGS `$(CFLAGS)` and the two extra variables: `$@ $<`
  - $@ is the **target,** i.e what ever you are trying to build in the `%.o`

- $< is **all the dependancies** specified earlier

So now instead of having to specify individual dependancies, you are are telling Make to go and figure out what all the dependancies are

- e.g. hello now depends on hello.c library.o extra-library.o

```
hello: hello.c library.o extra-library.o
```

Implicit Pattern Rules

- due to Make's age, it comes with some rules about compiling C code already (and other old langauges)
- you dont need to set the `CC` and `CFLAGS` because they're already built in
- this reduces the Makefile even more:

```
.PHONY: all clean

all: hello coursework.zip flowchart.pdf
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

Generalising the Makefile further

- say you want to add more figures, you could add dependancies on `all` to built them, or instead:

```
$(wildcard *.dot)
```

- this is saying go and find everything with a .dot at the end, then do some pattern substitution `patsubst` change all the .dot files to .pdf
- you store this rule inside the `figures` variable

```
.PHONY: all clean
figures=$(patsubst .dot,.pdf,$(wildcard *.dot))

all: hello coursework.zip ${figures}
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```