Make

Make is a build tool which allows you to automate formal shifting. We used Make abt in C & now we'll learn abt of the nuts & bolts.

You could write a shell script to automate this process but this will often lead to duplicated effort w.r to lines of code run. Say for example you want to compile some source code & zip the executables. If you make a modification to a single source script, a shell script will have to recompile all the source code to get to the new output.

Make gives us a way to generalise build patterns & track files to prevent duplicated effort. The version of Make most commonly used is GNU Make.

Rules for Make are kept in the Makefile. Let's look at an example:

Rules for Make are placed into a *Makefile* and look like the following:

```
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
```

folder

the 'hello' rule has the dependancies 'hello.c' & 'library.o', which in turn has the dependancy 'library.c'

If you ask make to build `hello` it will figure out what it needs to do:

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

Makefile executes what needs to be done according to the rule.

Here having 'library.o' as a dependancy triggers the 'library.o' rule.

Make also tracks whether or not the dependancies are up to date. So if you ran `make hello` twice, it wouldn't run the second time. The hierarchy of dependancies also applies here.

So say you already had the 'hello' output & changed 'library.c' & re ran `make hello`, Make would expand the 'library.o' dependancy & see that the dependancy library.c' has a more recent version on after ∴ it would re run both rules.

BUT

If you only changed 'hello.c', the 'library.o' dependancy would be unchanged & so only the command in the `make hello` rule would run. As the lecture explains:

> If you alter files... Make is smart enough to only rerun the steps you need:
> For example if you edit `hello.c` and rebuild:
>
> ```
> $ make hello
> cc -o hello hello.c library.o
> ```
>
> But if you edit `library.c` it can figure out it needs to rebuild *everything*
>
> ```
> $ make hello
> cc -c -o library.o library.c
> cc -o hello hello.c library.o
> ```

Phony Targets

These rules that don't depend on specific files but just tell Make what to do when they're run. Examples of this are all, clean & install.

Make all usually just has a few other rules as its dependancies (& it'll only update the ones that need it), Make clean just deletes all generated files & Make install installs things...

These rules (need) to be declared as .PHONY.
it isn't strictly necessary but can turn up some errors. It's good practice.

```
.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.
```

Pattern Rules

These provide a good way to generalise Make rules (rather than listing dependancies individually)
To do this you use the % operator.

```
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 $@
```

you can also have string variables to save you writing things out multiple times.

— you call these variables using standard bash notation. *

$@ is the target (whatever you're trying to build in the rule name)

$< is all the dependancies

This rule generates no '.o' suffix

The % is the wildcard operator of Make. So '%.c' represents any file ending in .c

* $() doesn't mean variable, it just means run it as a command & sub-in the output.

More Generalisation Variables

Make has a function called patsubst. This finds white-space seperated words in TEXT, that match a PATTERN & replaces them with REPLACEMENT. PATTERN may contain a '%' which acts as a wildcard, matching any number of charecters in a word. Usage:

```
$(patsubst PATTERN, REPLACEMENT, TEXT)
```

We can use this function to generate a list of dependancies:

```
.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 $@
```

The wildcard variable is specific to Make. It is used to find all filenames that match a specific pattern.

The figures variable takes in all the .dot files as text input, changes the file extension to .pdf & returns it as text output.

This can now be called as a dependancy in all, which in turn means, the .pdf rule for all PDFs in the directory are dependancies for all

Summary of Perks:

I love Make...

- I abuse it for compiling everything
- For distributing reproducible science studies
- For building and deploying websites

Pattern rules and the advanced stuff is neat...

- ...but if you never use it I won't be offended
- Make is one of those tools that you'll come back to *again and again* over your careers.
- ...and there's a *bunch* of tricks I haven't shown you ;-)

Go and read the GNU *Make Manual*

- Its pretty good for a technical document

Language Specific Build Tools.

One thing Make is not good at is tracking external libraries. It doesn't know how to fetch dependencies. It doesn't track versions beyond if the source is newer than the object functionality we saw last video. This means it struggles with collecting 3rd party packages.

Unfortunately, modern programming is reliant on importing 3rd party packages.

You used to have to download all the dependacies by hand, compile, install them yourself. This process has now been automated.

Now, almost every language comes with its own library management tooling which lets developers specify dependencies (which libraries they're using, from where), it tells the compiler how to rebuild your project. Unfortunately that means there is a seperate build tool, each with different syntax, for each language. :(

Some include:

Commonlisp	ASDF and Quicklisp
Go	Gobuild
Haskell	Cabal
Java	Ant, Maven, Gradle...
JavaScript	NPM
Perl	CPAN
Python	Distutils and `requirements.txt`
R	CRAN
Ruby	Gem
Rust	Cargo
LaTeX	CTAN and TeXlive

...and *many* more.

Comprehensive .... Archive Network

C

C primarily just uses Make