CITS4407 Open Source Tools and Scripting
Build management

Unit coordinator: Arran Stewart

Overview

This week:

- `make` and Makefiles

make and Makefiles

The make tool is used for *building* things on Unix-like systems, using "recipes" contained in *Makefiles*.

`make` and Makefiles

The `make` tool is used for *building* things on Unix-like systems, using "recipes" contained in *Makefiles*.

What sort of "things"?

They could be:

- PDF documents reporting on the results of experiments
- complex programs, built from multiple files or packages
- web sites (the CITS4407 site is tested using a Makefile)
- Docker images, like the one used for Assignment 1
- virtual machines

or almost any other sort of complex software artifact.

Makefiles

The recipes are contained in plain text files, conventionally named `Makefile`, and at their simplest look something like this:

Makefile

```
workshop09.pdf: workshop09.md
	pandoc --from markdown --to html \
	  --pdf-engine=weasyprint \
	  --output=workshop09.pdf workshop09.md
```

Makefile rules

```
workshop09.pdf: workshop09.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=workshop09.pdf workshop09.md
```

Makefile contain what are called “rules” for building things; their syntax is

```
output: ingredient1 ingredient2 ...
→command1
→command2
→command3
→...
```

Makefile rules

```
output: ingredient1 ingredient2 ...
→command1
→command2
→command3
→...
```

The rule specifies:

- something we want to make (the output);
- what ingredients we need to make it; and
- what commands we need to run, to turn those ingredients into the output.

The arrows indicate “tab” characters. In a Makefile, the instructions for *how* to build the output always start with a tab character. (But we will not show them from now on.)

Makefile rules

```
workshop09.pdf: workshop09.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=workshop09.pdf workshop09.md
```

Here, `workshop09.pdf` is the thing we want to build; `workshop09.md` is the ingredient we need in order to build it (a Markdown file); and we use the pandoc tool to convert from Markdown format to PDF.

Running make

Makefile

```
workshop09.pdf: workshop09.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=workshop09.pdf workshop09.md
```

To run make, we simply give the name of some output we want to build – in this case,

```
$ make workshop09.pdf
```

Build management tools

`make` is an example of a *build-management* (or *build automation*) tool.

Many others exist, but `make` is one of the oldest (originally created by Stuart Feldman in April 1976 at Bell Labs)[1] and most widely used.

[1] https://en.wikipedia.org/wiki/Make_(software)#Origin

Dependencies

Makefile

```
workshop09.pdf: workshop09.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=workshop09.pdf workshop09.md
```

In build automation terminology, we say that `workshop09.md` is a *dependency* of `workshop09.pdf`, and that `workshop09.pdf` is a *target*.

Dependencies

If all `make` did was run a set of commands to build something, there would be no great advantage to using it over a Bash script.

However, it *also* tracks whether any of the dependencies (and *their* dependencies, and so on – there could be hundreds, in a large Makefile) have changed and are *newer* than the the output.

If so, it knows the recipe needs to be re-run; if not, it knows the output is up to date.

Dependencies

Makefile

```
workshop09.pdf: workshop09.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=workshop09.pdf workshop09.md
```

```
$ make workshop09.pdf
make: 'workshop09.pdf' is up to date.
$ touch workshop09.md
$ make workshop09.pdf
pandoc --from markdown --to html --pdf-engine=weasyprint
```

Dependencies

If we are building a very large software program, or a complex output of some other sort, `make` ensures that

- we don't re-build things if we don't have to – this can save a great deal of time
- we *do* re-build things when their dependencies have changed

Generic recipes

What if have multiple files, that are all built in the same general way?

For instance,

- `workshop06.pdf`
- `workshop07.pdf`
- `workshop08.pdf`

. . . and so on.

Generic recipes

If using pandoc to turn Markdown files into PDFs, we'll always need to run a command of the form

```
pandoc --from markdown --to html \
     --pdf-engine=weasyprint \
     --output=output.pdf input.md
```

but the files *output.pdf* and *input.md* will vary.

Generic recipes

`make` lets us write *generic* recipes, which use the following “special variables”:

- `$@` – the current target file
- `$^` – all dependencies listed for the current target
- `$<` – the first (left-most) dependencies for the current target

(There are many more special variables, but these are the most common.)

Generic recipes

Using special variables, we can create a rule that looks like this:

Makefile

```
%.pdf: %.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=$@ $<
```

We can read this as saying:
"To build a PDF file, look for a Markdown file of the same base name, but ending in" `.md`".
Run the `pandoc` command specified in the recipe, but after `--output` put the name of the output file we're producing, and as a final argument, put the name of the left-most dependency."

Generic recipes

Makefile

```
%.pdf: %.md
  pandoc --from markdown --to html \
    --pdf-engine=weasyprint \
    --output=$@ $<
```

(In this case, there's only one dependency – but we could have more, for instance, if our Markdown document refers to images.)

Makefile variables

The term “special variables” suggests make has the concept of *ordinary* variables as well – and it does.

Let’s see how we might use them.

We might want to say, in order to run pandoc, you first have to fetch the pandoc binary from the Internet.

Makefile

```
pandoc:
  wget https://github.com/jgm/pandoc/releases/download/2.13/pandoc-2.13-linux-amd6
  tar xf pandoc-2.13-linux-amd64.tar.gz
  mv ./pandoc-2.13/bin/pandoc .
```

The URL where the pandoc binary is located is lengthy, and also might change if we start to use a new version.

Makefile variables

So make lets us put fragments of text in *variables*;
once they are defined, we can *expand* them using the syntax `$(`*var*`)`.

(This is similar to, but not quite the same as, the expansion syntax Bash uses. Why might we not want to use the same syntax?)

Makefile

```
PANDOC_URL=https://github.com/jgm/pandoc/releases/download/2.13/pandoc-2.13-linux-a

pandoc:
  wget $(PANDOC_URL)
  tar xf pandoc-2.13-linux-amd64.tar.gz
  mv ./pandoc-2.13/bin/pandoc .
```

Writing your own Makefiles

In the lab/workshops, there will be opportunity to write your own Makefiles – and we will use them in Assignment 2.