make

Lecture 16

Dr. Colin Rundel

make

- build tool for the creation of software / libraries / documents by specifying dependencies

    - Almost any process that has files as input and outputs can be automated via make

- Originally created by Stuart Feldman in 1976 at Bell Labs

- Almost universally available (all flavors of unix / linux / MacOS / Windows via RTools)

- Dependencies are specified using a text-based `Makefile` with a simple syntax

Makefile

A `Makefile` provides a list of target files along, their dependencies, and the steps necessary to generate each of the targets from the dependencies.

```
1  target1: depend1 depend2 depend3 ...
2      step1
3      step2
4      step3
5      ...
6
7  depend1: depend4 depend5
8      step1
9      step2
10     ...
```

In the above example `target*` and `depend*` are all just files (given by a relative or absolute path).

Makefile (basic example)

```
1  paper.html: paper.Rmd fig1/fig.png fig2/fig.png
2      Rscript -e "rmarkdown::render('paper.Rmd')"
3
4  fig1/fig.png: fig1/fig.R
5      Rscript fig1/fig.R
6
7  fig2/fig.png: fig2/fig.R
8      Rscript fig2/fig.R
```

Smart Building

Because the `Makefile` specifies the dependency structure `make` knows when a file has changed (by examining the file's modification timestamp) and only runs the steps that depend on the file(s) that have changed.

- After running `make` the first time, I edit `paper.Rmd`, what steps run if I run `make` again?

- What about editing `fig1/fig.R`?

Variables

Like R or other language we can define variables

```
1  R_OPTS=--no-save --no-restore --no-site-file --no-init-file --no-environ
2
3  fig1/fig.png: fig1/fig.R
4      cd fig1;Rscript $(R_OPTS) fig.R
```

Special Targets

By default if you run `make` without arguments it will attempt to build the *first* target in the `Makefile` (whose name does not start with a `.`). By convention we often include an `all` target which explicitly specifies how to build everything within the project.

`all` is an example of what is called a phony target - because there is no file named `all` in the directory. Other common phony targets:

- `clean` - remove any files created by the Makefile, restores to the original state

- `install` - for software packages, installs the compiled programs / libraries / header files

Optionally, we specify all phony targets by including a line with `.PHONY` as the target and the phony targets as dependencies, i.e.:

```
1   .PHONY: all clean install
```

Builtin / Automatic Variables

- `$@` - the file name of the target

- `$<` - the name of the first dependency

- `$^` - the names of all dependencies

- `$(@D)` - the directory part of the target

- `$(@F)` - the file part of the target

- `$(<D)` - the directory part of the first dependency

- `$(<F)` - the file part of the first dependency

Pattern Rules

Often we want to build several files in the same way, in these cases we can use `%` as a special wildcard character to match both targets and dependencies.

So we can go from

```
1   fig1/fig.png: fig1/fig.R
2       cd fig1;Rscript fig.R
3
4   fig2/fig.png: fig2/fig.R
5       cd fig2;Rscript fig.R
```

to

```
1   fig%/fig.png: fig%/fig.R
2       cd $(<D);Rscript $(<F)
```

Makefile (fancier example)

```
1  all: paper.html
2
3  paper.html: paper.Rmd fig1/fig.png fig2/fig.png
4      Rscript -e "library(rmarkdown);render('paper.Rmd')"
5
6  Fig%/fig.png: Fig%/fig.R
7      cd $(<D);Rscript $(<F)
8
9  clean:
10     rm -f paper.html
11     rm -f Fig*/*.png
12
13 .PHONY: all clean
```

Live Demo
HW4 Makefile