# Introduction to `make`

## Reproducible Computing

## @ JSM 2019

Colin Rundel

July 27, 2019

# make

- Automatically build software / libraries / documents by specifying dependencies
- Originally created by Stuart Feldman in 1976 at Bell Labs
- Almost universally available (all flavors of unix / linux / osx)

# Makefile basics

A `Makefile` provides a list of targets and their dependencies. For each target you then specify the steps necessary to generate the target using the dependencies.

```
target1: depend1 depend2 depend3 ...
    step1
    step2
    step3
    ...

depend1: other_depend
    step1
    step2
```

The targets and their dependencies must form a directed acyclic graph - which is how `make` evaluates what steps to run and in what order.

# Steps

Steps are just one or more shell commands to be executed that will eventually generate the target.

Some important features / requirements:

- Steps must be prefixed with a tab character (not spaces)
- Each step executes in its own shell, therefore commands that change state / environment (e.g. `cd`) will not necessarily persist.
    - The solution is to string commands together into a single step using `;` or `&&`.
- To stop a step from echoing its command when running prefix it with a `@`.

# Example 1 - Dependencies

```
a: b c
    @printf "Building a\n"

b:
    @printf "Building b\n"

c:
    @printf "Building c\n"
```

# Example 2 - Paper

```
paper.html: paper.Rmd Fig1/fig.png Fig2/fig.png
    Rscript -e "library(rmarkdown);render('paper.Rmd')"

Fig1/fig.png: Fig1/fig.R
    cd Fig1;Rscript fig.R

Fig2/fig.png: Fig2/fig.R
    cd Fig2;Rscript fig.R
```

# Smart Execution

Because the `Makefile` specifies the dependency structure and `make` knows when a file has changed (by examining the file's modification timestamp) it 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 if I edit `Fig1/fig.R`?
- What if I rename `paper.html` to `paper2.html`

# Variables

Like shell (or R) we can define variables

```
R_OPTS=--no-save --no-restore --no-site-file

Fig1/fig.png: Fig1/fig.R
  cd Fig1;Rscript $(R_OPTS) fig.R
```

# Special Targets

By default when running `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 as this first 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 `all` file 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 / headers

Any phony targets in a Makefile can be listed using the `.PHONY` special built-in target name,

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

# Example 3 - Phony

```
.PHONY: c

a: b c
    @printf "Building a\n"

b:
    @printf "Building b\n"

c:
    @printf "Building c\n"
```

# Builtin 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

```
Fig1/fig.png: Fig1/fig.R
    cd R;Rscript fig.R

Figs2/fig.png: Fig1/fig.R
    cd R;Rscript fig.R
```

to

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

# Example 4 - Paper (Fancy)

```
all: paper.html

paper.html: paper.Rmd Fig1/fig.png Fig2/fig.png
    Rscript -e "library(rmarkdown);render('paper.Rmd')"

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

clean:
    rm -f paper.html
    rm -f Fig*/*.png

.PHONY: all clean
```

# Further Reading / Reference

- Mike Bostock - Why use make
- Karl Broman - minimal make
- GNU Manual
- GitHub Code Search - `filename:Makefile`