make

Statistical Computing & Programming

Shawn Santo

Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Additional resources

- minimal make by Karl Broman
- Why Use Make by Mike Bostock
- GNU make manual
- Make for Windows

`make`

- Automatically build software / libraries / documents by specifying dependencies via a file named `Makefile`
    - provide instructions for what you want to build and how it can be built
- Originally created by Stuart Feldman in 1976 at Bell Labs
- Almost universally available (all flavors of UNIX / Linux / OSX)

Check for `make` with

```
make --version
```

```
#> GNU Make 3.81
#> Copyright (C) 2006  Free Software Foundation, Inc.
#> This is free software; see the source for copying conditions.
#> There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
#> PARTICULAR PURPOSE.
#>
#> This program built for i386-apple-darwin11.3.0
```

`Makefile` structure

```
target: prerequisite_1 prerequisite_2 ...
  recipe
  ...
  ...
```

- `target` is the file you want to generate
- `prerequisite_*` are the files the target file depends on
- a recipe is an action that `make` carries out, commands you run in the terminal

Alternatively,

```
targetfile: sourcefile
  command
  ...
  ...
```

`Makefile` structure

A more realistic structure:

```
target: prerequisite_1 prerequisite_2 ...
  recipe
  ...
  ...

prerequisite_1: prerequisite_1a prerequisite_1b ...
  recipe
  ...
  ...

prerequisite_2: prerequisite_2a prerequisite_2b ...
  recipe
  ...
  ...
```

Example

```
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
```

What are the targets and dependencies?

The first target is the default goal of what `make` tries to create.

Another example

```
hd_cov_test_band.o: hd_cov_test_band.c
    export PKG_CFLAGS="-fopenmp"
    export PKG_LIBS="-lgomp"
    R CMD SHLIB hd_cov_test_band.c

clean:
    rm hd_cov_test_band.o
    rm hd_cov_test_band.so

.PHONY: clean
```

How `make` processes a `Makefile`

1. Once you have a `Makefile` written, type `make` in your terminal.

   ```
   make
   ```

2. `make` looks for files named `GNUmakefile`, `makefile`, or `Makefile`.
3. The `make` program uses the `Makefile` data base and last-modification times of the files to decide which of the files need to be updated.
4. For each file that needs to be updated, the recipes are executed.

```
hd_cov_test_band.o: hd_cov_test_band.c
    export PKG_CFLAGS="-fopenmp"
    export PKG_LIBS="-lgomp"
    R CMD SHLIB hd_cov_test_band.c
```

Understanding `make`

Consider the `Makefile` below. I run `make`. Later, I change some code in `Fig2/fig.R` and save the file. What is updated when I run `make` again?

```
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
```

What if I only change some text in `paper.Rmd` and then save the file?

`Makefile` **tips**

1. Name your file `Makefile`.
2. Use `tab` to add recipes.
3. Use `#` to add comments to your `Makefile`.
4. Split long lines with `\`.
5. Have one target precede each `:`.
6. Remember, recipes are meant to be interpreted by the shell and thus are written using shell syntax.
7. Use semicolons to specify a sequence of recipes to be executed in a single shell invocation.

`make` **demo**

Some advanced `make`

Variables

Like R, or other languages, we can define variables.

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

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

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

- Typically, we use uppercase letters for variable names.
- Refer to a variable's value by `${MY_VARIABLE}` or `$(MY_VARIABLE)`.
- Do not use `:`, #, =, or a white space in your variable's name.

Built-in variables

Variable	Description
`$@`	the file name of the target
`$<`	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. For these cases we can use `%` as a special wildcard character to match both targets and dependencies.

Rather than our `Makefile` be

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

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

we can use built-in variables and patterns to have

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

- `%` can match any nonempty substring.
- The substring that the `%` matches is called the stem.
- A prerequisite with `%` has the same stem that was matched by the `%` in the target.

Phony targets

A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request. There are two reasons to use a phony target: to avoid a conflict with a file of the same name, and to improve performance.

For example,

```
clean:
  rm *.log
```

would remove all `.log` files when `make clean` is run. However, a problem can arise if we ever have a file named `clean`.

To make this more robust we can configure it as

```
.PHONY: clean
clean:
  rm *.log
```

Command `make clean` will remove the log files regardless of whether a file named `clean` exists.

Another common phony target is `all`. Its prerequisites are all the individual programs we want to build. For example:

```
.PHONY: all
all: prog1 prog2 prog3

prog1: prog1.o utils.o
  cc -o prog1 prog1.o utils.o

prog2: prog2.o
  cc -o prog2 prog2.o

prog3: prog3.o sort.o utils.o
  cc -o prog3 prog3.o sort.o utils.o
```

Use `make` to build all the programs. Or build a subset by specifying each program's name: `make prog1 prog2`.

Fancy `Makefile`

Our original example:

```
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
```

```
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 paper.html
    rm -f Fig*/*.png

.PHONY: clean
```

Another fancier `Makefile`

```
SRC = $(wildcard *.Rmd)
TAR_PDF = $(SRC:.Rmd=.pdf)
TAR_HTML = $(SRC:.Rmd=.html)

all: $(TAR_PDF) $(TAR_HTML)

%.pdf: %.html
	Rscript -e "pagedown::chrome_print('$(<F)')"

%.html: %.Rmd
	Rscript -e "rmarkdown::render('$(<F)')"

clean:
	rm *.pdf
	rm *.html

.PHONY: clean all
```

Exercise

Create a `Makefile` for the R project in the learn_make repository on GitHub. The target goal should be `learn_make.html`. The below steps will help guide you in creating `Makefile`.

1. Diagram the dependency structure on paper.
2. First, create a `Makefile` that only knits the Rmd file and produces the `learn_make.html` file.
3. Next, add rules for the data dependencies.
4. Add phony `clean_html` and `clean_data` targets that delete the html file and delete the rds files in `data/`, respectively.
5. Revise your `Makefile` with built-in variables or other useful features.

References

1. Broman, K. (2020). minimal make. http://kbroman.org/minimal_make/.
2. GNU make. (2020). https://www.gnu.org/software/make/manual/make.html#toc-An-Introduction-to-Makefiles.