

Make all the things

Canton Linux Enthusiasts

Michael Meffie

April 27, 2023

About me

Michael Meffie

Software Engineer

Sine Nomine Associates

Canton Linux Enthusiast

Introduction to Make

Gentle introduction to Make and how it can be helpful and/or harmful.

Out of scope

- C development
- Autotools
- Most of GNU Make

*Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, cc *.o was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. – Stuart Feldman*

The tab blunder

Why the tab in column 1? Yacc was new, Lex was brand new. I hadn't tried either, so I figured this would be a good excuse to learn. After getting myself snarled up with my first stab at Lex, I just did something simple with the pattern newline-tab. It worked, it stayed. And then a few weeks later I had a user population of about a dozen, most of them friends, and I didn't want to screw up my embedded base. The rest, sadly, is history. – Stuart Feldman

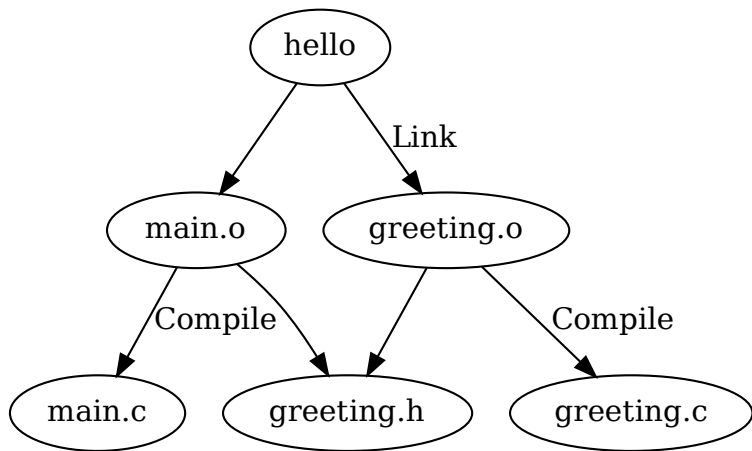
*Non-file productions were intentional and in there from day one.
'Make all' and 'clean' were my own conventions from earliest days.
– Stuart Feldman*



Running make

```
$ make [options] [target [target ...]]
```


- make looks for Makefile in the current directory
- make evaluates the given targets
- If not targets are specified, make evaluates the first rule found in the Makefile

Dependencies



target  *prerequisites* 

```
target: depend1 depend2
<tab> shell command
<tab> shell command
```

recipe 

- A recipe is a list of commands (not a script)
- Each command is run in a separate shell
- Make exits if a command fails (exits with non-zero)
- Compound commands can be used

- Think Dependencies **Not Sequence**
- Dependencies must be defined before Make runs
- Dependencies allow for parallelization
- File modify time checks (not content)
- Static filenames
- Non-existing files are always out of date (key to task running)

- Basic templating using string replacement
- Evaluated recursively on use (not definition)
- Override values on the command line
- Fallback to environment (by default)

Makefile:

```
COMMAND = rpmbuild
FLAGS = -ba --rcfile $(RPMRC)
RPMRC = $(HOME)/myrpmrc
SPEC = foo.spec
```

```
build:
    $(COMMAND) $(FLAGS) $(SPEC)
```

Command line:

```
$ make SPEC=bar.spec
```

Expands to:

```
rpmbuild -ba --rcfile /home/mmeffie/myrpmrc bar.spec
```

Phony Targets

- Since non-existing files are always out of date, they can be used to run tasks
- The special `.PHONY:` directive tell make to ignore the target file if one exists
- Phony targets can have prereqs
- Phony targets can be prereqs for other phony targets (but not real targets)

Makefile:

```
.PHONY: clean
clean:
    rm -f *.html
```

Example:

```
$ touch clean
$ make clean
rm -f foo.html
```

Conventional target names

- all - first target
- help - list targets
- install - build then install
- check, test - run tests
- doc, docs - generate documentation
- clean, distclean - cleanup

Pattern Rules

- Basic rules templating using file name extensions
- Special % pattern matching character
- Special \$@ target name macro
- Special \$< prereq name macro

Makefile:

```
html: foo.html bar.html baz.html

%.html: %.html.j2 data.json
    jinja2 --outfile $@ $< data.json
```


Tip: Marker targets

- A marker target (or sentinel target) is a filename used to represent state
- Useful if a rule generates multiple targets (batch mode)
- Useful if the generated filename is not known in advance

Makefile:

```
TEMPLATES = foo.html.j2 bar.html.j2 baz.html.j2
```

```
.PHONY: html
```

```
html: .html
```

```
.html: $(TEMPLATES)
```

```
    generate-html $?
```

```
    touch .html
```

Tip: Line continuation

Use `\` to wrap long lines

Makefile:

```
TEMPLATES = foo.html.j2 \  
            bar.html.j2 \  
            baz.html.j2
```

```
a_long_command:  
    for j in $(TEMPLATES); do \  
        check-template $$j; \  
    done
```

- Non-portable but common
- Parallel option (-j, --jobs)
- Variables
- Builtin functions
- Conditionals

```
foo := this value $(expands_now)
bar := $(shell gather-data)
ifndef baz
baz := "conditionals"
endif
```

```
$ make --debug --no-builtin-rules --no-builtin-variables
```