`make` it Rain

Sumner Evans

March 21, 2019

Mines Linux Users Group

Introduction

What is GNU `make`?

GNU `make` is a tool which controls the generation of executables and other non-source files of a program from the program's source files.

GNU `make` gets its knowledge of how to build your program from a file called the *makefile*, which lists each of the non-source files and how to compute it from other files.

(The GNU `make` Project Page[1])

[1] `https://www.gnu.org/software/make/`

Basic Usage of GNU `make`

```
$ make
```

Advanced Usage of GNU `make`

To run `make`, there must be a file called `GNUmakefile`, `makefile`, or `Makefile` in the current working directory. Otherwise, it will return an error.

The `make` program takes an arbitrary number of parameters. Each of the parameters must be a *target*[2].

If no parameter is specified, the `all` target is assumed.

Examples:

```
$ make all
$ make a.out README.pdf
$ make makefiles.pdf
$ make profit
```

[2] More on targets later

Advanced Usage of GNU `make`

To run `make`, there must be a file called `GNUmakefile`, `makefile`, or `Makefile` in the current working directory. Otherwise, it will return an error.

The `make` program takes an arbitrary number of parameters. Each of the parameters must be a *target*[2].

If no parameter is specified, the `all` target is assumed.

Examples:

```
$ make all
$ make a.out README.pdf
$ make makefiles.pdf
$ make profit
```

[2] More on targets later

How to Make a `Makefile`

Understanding the Building Blocks of `Makefiles`

The main building block of a `Makefile` is the *rule*. Every `Makefile` contains a set of rules.

All rules are of the form

```
target: dependencies ...
        commands
        ...
```

Note: commands must be indented with the `TAB` character.

- The `target` is the file to be created by the rule.
- The `dependencies` (also known as prerequisites) are a list of the targets/files which need to exist for the `target` to be created.
- The `commands` are a list of the commands to create the `target` from the `dependencies`.

Understanding the Building Blocks of `Makefiles`

The main building block of a `Makefile` is the *rule*. Every `Makefile` contains a set of rules.

All rules are of the form

```
target: dependencies ...
        commands
        ...
```

Note: commands must be indented with the `TAB` character.

- The `target` is the file to be created by the rule.
- The `dependencies` (also known as prerequisites) are a list of the targets/files which need to exist for the `target` to be created.
- The `commands` are a list of the commands to create the `target` from the `dependencies`.

Understanding the Building Blocks of `Makefiles`

The main building block of a `Makefile` is the *rule*. Every `Makefile` contains a set of rules.

All rules are of the form

```
target: dependencies ...
        commands
        ...
```

Note: commands must be indented with the `TAB` character.

- The `target` is the file to be created by the rule.
- The `dependencies` (also known as prerequisites) are a list of the targets/files which need to exist for the `target` to be created.
- The `commands` are a list of the commands to create the `target` from the `dependencies`.

A Very Simple Example

```
all: helloworld

helloworld: helloworld.c
        gcc -o helloworld helloworld.c
```

In this example, to build `helloworld`, GNU `make` will ensure that the `helloworld.c` file exists.

Why is this Better than a Bash Script?

But wait, I can do that with a BASH script:

```
[[ -f helloworld.c ]] && gcc -o helloworld helloworld.c
```

why do I need a `Makefile`?

GNU `make` only executes a target's commands if its *dependencies* have been updated since the last time `make` was called.

Why is this Better than a Bash Script?

But wait, I can do that with a BASH script:

```
[[ -f helloworld.c ]] && gcc -o helloworld helloworld.c
```

why do I need a `Makefile`?

GNU `make` only executes a target's commands if its *dependencies* have been updated since the last time `make` was called.

Dependency Management

Dependency management is one of the best features of GNU `make`. Dependencies can even be chained!

```
all: presentation.pdf

presentation.pdf: presentation.tex
	xelatex -shell-escape presentation.tex

presentation.tex: presentation.rst xelatex.tex
	rst2beamer --template=xelatex.tex \
	    presentation.rst > presentation.tex
```

.PHONY

GNU `make` expects all targets to be *files or directories*. So if you have a file called `all`, you may run into problems.

To avoid this, you can define targets as being *phony*. This will basically make it so that those targets are perpetually out of date and will always be recomputed, regardless of whether or not they exist on the filesystem.

```
all: foo bar baz

.PHONY: all
```

A `Makefile` is not a Bash Script!

Do **not** write a `Makefile` that looks like this:

```
all: documentation
	gcc helloworld.c

documentation:
	xelatex -shell-escape doc.tex
	biber doc
	xelatex -shell-escape doc.tex
```

This `Makefile` will cause the program and the documentation to be compiled *regardless* of whether or not you've updated the corresponding source files (`doc.tex` and `helloworld.c`).

This is very bad if you are compiling a sizeable program.

Automatic Variables and Implicit Rules i

Often, you may have many files which have the same basic rule. For example you may want to compile a `.o` file for every `.cpp` file in a C++ project. Luckily, there's a syntax for that:

```
%.o: %.cpp
	g++ -c $< -o $@
```

- `%.o` as the *target* is a wildcard for all `*.o` targets.
- `%.cpp` means that if the target is `foo.o`, then it depends on `foo.cpp`.
- `$@` is the file name of the *target* of the rule.
- `$<` is the file name of the first prerequisite.

Automatic Variables and Implicit Rules ii

There are a lot of other useful automatic variables.

- `$@` is the file name of the *target* of the rule.
- `$<` is the file name of the first prerequisite.
- `$?` is the file names of all the prerequisites that are newer than the target, with spaces between them.
- `$^` is the names of all the prerequisites, with spaces between them.
- A bunch more... See `https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html` for a list.

Variables

If there are *automatic* variables, then there must be normal variables too, right? Yes!

You can define variables with the following syntax:

```
CXX=clang
OUTFILES=foo.o bar.o baz.o
```

and you can use them like this (*note, parentheses are required*):

```
all: $(OUTFILES)

%.o: %.c
	$(CXX) $< -o $@
```

Wildcards and Pattern Substitutions

Often it is useful to have glob-like wildcards in variables. To do this, you can use the `wildcard` function:

```
TEXFILES=$(wildcard *.tex)
```

You may then want to use all of those filenames to determine the targets to compute. To do this, you can use the `patsubst` (**pat**tern **subst**itution) function:

```
PDFFILES=$(patsubst %.tex,out/%.pdf,$(TEXFILES))
```

This will result in `TEXFILES` containing all of the `.tex` files in the directory, and `PDFFILES` containing all of those file names, but with `.pdf` instead of `.tex`.

Wildcards and Pattern Substitutions

Often it is useful to have glob-like wildcards in variables. To do this, you can use the `wildcard` function:

```
TEXFILES=$(wildcard *.tex)
```

You may then want to use all of those filenames to determine the targets to compute. To do this, you can use the `patsubst` (**pat**tern **subst**itution) function:

```
PDFFILES=$(patsubst %.tex,out/%.pdf,$(TEXFILES))
```

This will result in `TEXFILES` containing all of the `.tex` files in the directory, and `PDFFILES` containing all of those file names, but with `.pdf` instead of `.tex`.

Wildcards and Pattern Substitutions

Often it is useful to have glob-like wildcards in variables. To do this, you can use the `wildcard` function:

```
TEXFILES=$(wildcard *.tex)
```

You may then want to use all of those filenames to determine the targets to compute. To do this, you can use the `patsubst` (**pat**tern **subst**itution) function:

```
PDFFILES=$(patsubst %.tex,out/%.pdf,$(TEXFILES))
```

This will result in `TEXFILES` containing all of the `.tex` files in the directory, and `PDFFILES` containing all of those file names, but with `.pdf` instead of `.tex`.

Managing TAR Files

You can use GNU `make` to manage TAR archives. This will create a `.tar.gz` containing all of the `.c` files in the directory.

```
CFILES=$(wildcard *.c)

all: test($(CFILES))

test(%.c):
        ar cr $@ $%
```

Note here that `$%` is another automatic variable containing the target member name. It is empty if the target is not an archive member.

Examples

Compiling a Presentation

```
FILENAME=makefiles
LATEX_COMPILER=xelatex -shell-escape

all: $(FILENAME).pdf

examples/%.pdf: examples/%.tex
	$(LATEX_COMPILER) -output-directory=examples $<

%.pdf: %.tex lug.cls
	$(LATEX_COMPILER) $<

.PHONY: all
```

Compiling a reStructuredText Presentation

```
RSTFILES=$(wildcard *.rst)
PDFFILES=$(patsubst %.rst,out/%.pdf,$(RSTFILES))

.PHONY: all
all: $(PDFFILES)

out:
	mkdir out

out/%.pdf: %.tex beamerthemecsam.sty csam.pdf | out
	xelatex -shell-escape $<
	mv $(patsubst out/%.pdf,%.pdf,$@) out

%.tex: %.rst xelatex.tex
	rst2beamer --template=xelatex.tex \
	    --theme=csam $< > $@
```

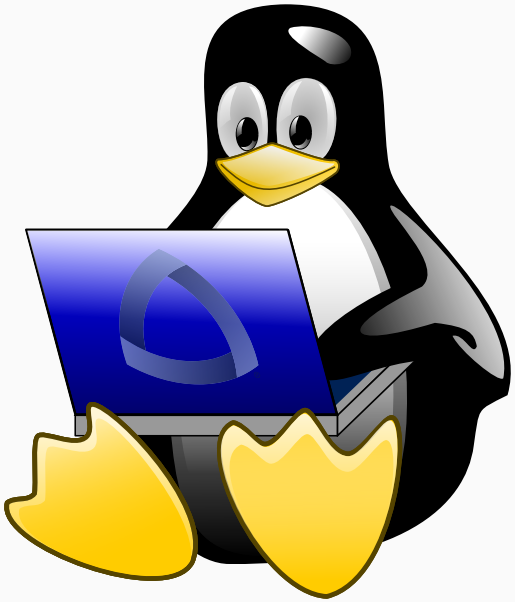
Additional Resources

- The GNU `make` Manual: `https://www.gnu.org/software/make/manual/html_node/index.html`
- Unix Makefile Tutorial: `https://www.tutorialspoint.com/makefile/`

Copyright Notice

This presentation was from the **Mines Linux Users Group**. A mostly-complete archive of our presentations can be found online at `https://lug.mines.edu`.

Individual authors may have certain copyright or licensing restrictions on their presentations. Please be certain to contact the original author to obtain permission to reuse or distribute these slides.



Colorado School of Mines
Linux Users Group