A Makefile defines a graph of rules for creating a target (or targets). Its purpose is to do the minimum amount of work needed to update a target to the most recent version of the source. Famously written over a weekend by Stuart Feldman in 1976, it is still widely used (particularly on Unix) despite many competitors and criticisms.

There are many varieties of make in existance, this article assumes that we are using GNU make which is the standard on Linux.

```
# Comments can be written like this.

# Files should be named Makefile and then be can run as `make <target>`.
# Otherwise we use `make -f "filename" <target>`.

# Warning - only use TABS to indent in Makefiles, never spaces!

#-----------------------------------------------------------------------
# Basics
#-----------------------------------------------------------------------

# A rule - this rule will only run if file0.txt doesn't exist.
file0.txt:
    echo "foo" > file0.txt
    # Even comments in these 'recipe' sections get passed to the shell.
    # Try `make file0.txt` or simply `make` - first rule is the default.


# This rule will only run if file0.txt is newer than file1.txt.
file1.txt: file0.txt
    cat file0.txt > file1.txt
    # use the same quoting rules as in the shell.
    @cat file0.txt >> file1.txt
    # @ stops the command from being echoed to stdout.
    -@echo 'hello'
    # - means that make will keep going in the case of an error.
    # Try `make file1.txt` on the commandline.

# A rule can have multiple targets and multiple prerequisites
file2.txt file3.txt: file0.txt file1.txt
    touch file2.txt
    touch file3.txt

# Make will complain about multiple recipes for the same rule. Empty
# recipes don't count though and can be used to add new dependencies.

#-----------------------------------------------------------------------
# Phony Targets
#-----------------------------------------------------------------------

# A phony target. Any target that isn't a file.
# It will never be up to date so make will always try to run it.
all: maker process

# We can declare things out of order.
maker:
```

```
    touch ex0.txt ex1.txt

# Can avoid phony rules breaking when a real file has the same name by
.PHONY: all maker process
# This is a special target. There are several others.

# A rule with a dependency on a phony target will always run
ex0.txt ex1.txt: maker

# Common phony targets are: all make clean install ...

#-----------------------------------------------------------------------
# Automatic Variables & Wildcards
#-----------------------------------------------------------------------

process: file*.txt  #using a wildcard to match filenames
    @echo $^    # $^ is a variable containing the list of prerequisites
    @echo $@    # prints the target name
    #(for multiple target rules, $@ is whichever caused the rule to run)
    @echo $<    # the first prerequisite listed
    @echo $?    # only the dependencies that are out of date
    @echo $+    # all dependencies including duplicates (unlike normal)
    #@echo $|   # all of the 'order only' prerequisites

# Even if we split up the rule dependency definitions, $^ will find them
process: ex1.txt file0.txt
# ex1.txt will be found but file0.txt will be deduplicated.

#-----------------------------------------------------------------------
# Patterns
#-----------------------------------------------------------------------

# Can teach make how to convert certain files into other files.

%.png: %.svg
    inkscape --export-png $^

# Pattern rules will only do anything if make decides to create the \
target.

# Directory paths are normally ignored when matching pattern rules. But
# make will try to use the most appropriate rule available.
small/%.png: %.svg
    inkscape --export-png --export-dpi 30 $^

# make will use the last version for a pattern rule that it finds.
%.png: %.svg
    @echo this rule is chosen

# However make will use the first pattern rule that can make the target
%.png: %.ps
    @echo this rule is not chosen if *.svg and *.ps are both present

# make already has some pattern rules built-in. For instance, it knows
```

```
# how to turn *.c files into *.o files.

# Older makefiles might use suffix rules instead of pattern rules
.png.ps:
    @echo this rule is similar to a pattern rule.

# Tell make about the suffix rule
.SUFFIXES: .png

#-----------------------------------------------------------------------
# Variables
#-----------------------------------------------------------------------
# aka. macros

# Variables are basically all string types

name = Ted
name2="Sarah"

echo:
    @echo $(name)
    @echo ${name2}
    @echo $name    # This won't work, treated as $(n)ame.
    @echo $(name3) # Unknown variables are treated as empty strings.

# There are 4 places to set variables.
# In order of priority from highest to lowest:
# 1: commandline arguments
# 2: Makefile
# 3: shell enviroment variables - make imports these automatically.
# 4: make has some predefined variables

name4 ?= Jean
# Only set the variable if enviroment variable is not already defined.

override name5 = David
# Stops commandline arguments from changing this variable.

name4 +=grey
# Append values to variable (includes a space).

# Pattern-specific variable values (GNU extension).
echo: name2 = Sara # True within the matching rule
    # and also within its remade recursive dependencies
    # (except it can break when your graph gets too complicated!)

# Some variables defined automatically by make.
echo_inbuilt:
    echo $(CC)
    echo ${CXX)}
    echo $(FC)
    echo ${CFLAGS)}
    echo $(CPPFLAGS)
    echo ${CXXFLAGS}
```

```
    echo $(LDFLAGS)
    echo ${LDLIBS}

#---------------------------------------------------------------------
# Variables 2
#---------------------------------------------------------------------

# The first type of variables are evaluated each time they are used.
# This can be expensive, so a second type of variable exists which is
# only evaluated once. (This is a GNU make extension)

var := hello
var2 ::=  $(var) hello
#:= and ::= are equivalent.

# These variables are evaluated procedurely (in the order that they
# appear), thus breaking with the rest of the language !

# This doesn't work
var3 ::= $(var4) and good luck
var4 ::= good night

#---------------------------------------------------------------------
# Functions
#---------------------------------------------------------------------

# make has lots of functions available.

sourcefiles = $(wildcard *.c */*.c)
objectfiles = $(patsubst %.c,%.o,$(sourcefiles))

# Format is $(func arg0,arg1,arg2...)

# Some examples
ls: * src/*
    @echo $(filter %.txt, $^)
    @echo $(notdir $^)
    @echo $(join $(dir $^),$(notdir $^))

#---------------------------------------------------------------------
# Directives
#---------------------------------------------------------------------

# Include other makefiles, useful for platform specific code
include foo.mk

sport = tennis
# Conditional compilation
report:
ifeq ($(sport),tennis)
    @echo 'game, set, match'
else
    @echo "They think it's all over; it is now"
endif
```

```
# There are also ifneq, ifdef, ifndef

foo = true

ifdef $(foo)
bar = 'hello'
endif
```

More Resources

- gnu make documentation
- software carpentry tutorial
- learn C the hard way ex2 ex28