CS104

CS104

make

https://makefiletutorial.com/

```
target: prerequisites
    command
    command
```

1. typically a *target* is a *file* produced (unless `.PHONY` is used)

2. order of searching for makefile in current directory:

   ```
   GNUmakefile
   makefile
   Makefile
   ```

3. run as `make target` or `make` for default target

4. first target is default

5. commands are only run if

   1. the target file does not exist
   2. a *file prerequisite* has been **modified after target**
   3. a *target prerequisite* needs to be run

6. first step is recursive in nature

7. the essence is: *if any prereq has mtime later than target then run the commands*

8. it uses `mtime` to decide

```
blah: blah.o
    cc blah.o -o blah # Runs third

blah.o: blah.c
    cc -c blah.c -o blah.o # Runs second

# Typically blah.c would already exist, but I want to limit any additional required files
blah.c:
    echo "int main() { return 0; }" > blah.c # Runs first
```

9. `blah.c` deleted ⇒ all 3 run
10. `touch blah.c` (thus mtime blah.c > blah.o) ⇒ first 2 run
11. `touch blah.o` (thus mtime blah.o > blah) ⇒ first 1 runs
12. nothing ⇒ nothing runs

```
some_file:
    touch some_file

clean:
    rm -f some_file
```

13. `clean` is not a keyword. so must
    1. *NOT be a file* in order to run
    2. *NOT be first target*
    3. *NOT be a prereq*
    4. OR just make it phony

Variables

```
files := file1 file2
some_file: $(files)
    echo "Look at this variable: " $(files)
    touch some_file

file1:
    touch file1
file2:
    touch file2

clean:
    rm -f file1 file2 some_file
```

14. variables are literally the string following `:=` so quotes are not special, tho they are for bash
15. preferred:
    1. `msg := hello world`
    2. `printf '$(msg)'` (bcz bash printf requires quoted input)
16. reference using `$(...)` or `${...}`

```
all: one two three

one:
    touch one
two:
    touch two
three:
    touch three

clean:
    rm -f one two three
```

`make` runs all which runs all targets since its default

```
all: f1.o f2.o

f1.o f2.o:
    echo $@
# Equivalent to:
# f1.o:
#    echo f1.o
# f2.o:
#    echo f2.o
```

- in case of multiple targets, the rule is defined for all targets

Wildcards

asterisk

- does file name matching like `*.o`
- DANGER 1: does not expand in variable definition usage :(
- DANGER 2: if written as it is, and there are no matches, remains as it is :/
- REC: `$(wildcard *.o)` to explicitly expand using wildcard function

```
thing_wrong := *.o # Don't do this! '*' will not get expanded
thing_right := $(wildcard *.o)

all: one two three four

# Fails, because $(thing_wrong) is the string "*.o"
one: $(thing_wrong)

# Stays as *.o if there are no files that match this pattern :(
two: *.o

# Works as you would expect! In this case, it does nothing.
three: $(thing_right)

# Same as rule three
four: $(wildcard *.o)
```

percent sign

- used in "match and replace" substitution in Static Pattern Rules

- used to define general pattern based rules in Pattern Rules

Automatic Variables

`$@` = target name

`$?` = prereqs later than the target

`$^` = all prereqs

`$<` = the first prereq

skibidi rules

Implicit Rules

these rules are added implicitly if you do not specify any commands

```
CC = gcc     # Flag for implicit rules : C compiler
CFLAGS = -g  # Flag for implicit rules. Turn on debug info

# Implicit rule #1: blah is built via the C linker implicit rule
# Implicit rule #2: blah.o is built via the C compilation implicit rule, because blah.c exists
blah: blah.o

blah.c:
	echo "int main() { return 0; }" > blah.c

clean:
	rm -f blah*
```

The important variables used by implicit rules are:

- `CC` : Program for compiling C programs; default `cc`
- `CXX` : Program for compiling C++ programs; default `g++`
- `CFLAGS` : Extra flags to give to the C compiler
- `CXXFLAGS` : Extra flags to give to the C++ compiler
- `CPPFLAGS` : Extra flags to give to the C preprocessor
- `LDFLAGS` : Extra flags to give to compilers when they are supposed to invoke the linker
- Compiling a C program:

```
%.o: %.c
    $(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@
```

- Compiling a C++ program:

```
%.o: %.cpp OR %.cc
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@
```

- Another *odd implicit rule* Linking a single object file

```
%: %.o
    $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@
```

so if you add `all: $(objects)`, in case `all.o` is not in `objects`, make will ADD IT!!!

Static Pattern Rules

Usage:
`targets... : target-pattern : prereq-patterns ...`
it will match targets with target-pattern containing `%` as the wildcard. the matched text *stem* is used as the replacement for `%` in prereq-patterns

```
objects = foo.o bar.o all.o
all: $(objects)
    $(CC) $^ -o all

# In the case of the first target, foo.o, the target-pattern matches foo.o and sets the "stem" to be "foo".
# It then replaces the '%' in prereq-patterns with that stem
$(objects): %.o: %.c
    $(CC) -c $^ -o $@

# equivalent to
# foo.o: foo.c
#     $(CC) -c $^ -o $@
# bar.o: bar.c
#     $(CC) -c $^ -o $@
# all.o: all.c
#     $(CC) -c $^ -o $@

all.c:
    echo "int main() { return 0; }" > all.c

# Note: all.c does not use this rule because Make prioritizes more specific matches when there is more than one match.
%.c:
    touch $@

clean:
    rm -f *.c *.o all
```

Static Pattern Rules and Filter

WEIRD++ so not covering cuz not in class but here it is

Pattern Rules

Think as a way to define your own implicit rules
Helps remind one of the oddities of implicit rules in general :)

```
%.o: %.c
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

- used to compile all `.o` files from corresponding `.c` files
- `%` matches any nonempty *stem* and same stem is subst. into `%.c`
- *note that* make prefers more specific matchings first so if `all.o: ...` is defined then it will be run when called, instead of `%.o` being run

Misc

- Prepend `@` to stop command from being printed to terminal
- Each command on a newline (which can be escaped using `\`) is exec in new shell
- So use, `;` on same line or `\ \n` to achieve effect of same shell for two commands
- To use bash variables or bash command substitution use `$$(...)`
- Add `.PHONY: clean` to indicate the target does not have file associated with it. this will also prevent from adding implicit rules for the above
- Makefile variable pattern substitution:

```
SRCS := main.cpp foo.cpp bar.cpp
OBJS := $(SRCS:.cpp=.o)
```

Error Handling

- Add `-k` when running make to continue running even in the face of errors. Helpful if you want to see all the errors of Make at once.
    - executes as many targets as possible without the risk of erroneous actions
- Add a `-` before a command to suppress the error
- Add `-i` to make to suppress errors for every command
    - different from `-k` since it will forcefully run every single command irrespective of errors
- `make -f makefile_path`

gdb

Viewing errors

- `2>&1` means send fd2 to wherever fd1 went
- `program ... | tee out.txt` will both print and write to out.txt the stdout of program (ie piped output)

Basic debugging

- use `#ifdef DEBUG` macros
- use assert: `<cassert> (C++)` or `<assert.h> (C)`

  - controlled by `NDEBUG` directive
- add `-g` flag to enable debug via gdb

gdb commands

Name	Usage	Explanation
`help`	-	-
`run`	`run [arglist]`	run prog from start with arglist

`break`	`b [file:]line`
	`b [file:]func`
		add breakpt at line or func (puts it at entry)

	`b ... if expr`	conditional breakpt when expr evals to True
`info`	`info breakpoints|break`	list current breakpoints
	`info args`	args to current function
	`info locals`	print local vars
`delete`	`delete [n]`	delete breakpoints (or n-th)
`continue`	`c`	continue to next bkpt or termination / signal
`print`	`p [/f] [expr]`	print val of expr acc to format f (or last value $)

	`p buffer[0]='Z'`
	`p strlen(buffer)`
		can be used to eval exprs/functions *but not set vars*

`set`	`set var=expr`	set var mid execution
`display`	`display [/f] expr`	display expr whenever bkpt is reached or we step through
`list`	`list`	print next 10 lines
	`list -`	prev 10 lines

	`list [file:]num`
	`list [file:]func`
		print lines surrounding given line/func in given file

`next`	`n`	next line, stepping over function calls
`step`	`s`	next line, stepping into function calls
`backtrace`	`bt`	print trace of all frames; also prints args to funcs on call stack
`up`	`up [n]`	go up n frames (caller)
`down`	`down [n]`	go down n frames (callee)

Notes:

- `s` **steps into any code that has debugging symbols enabled.**
  for eg stdlib functions like `printf` do not so `s` on them will just step over. but if `custom_print` is used which is in file `my_lib.c` and has been compiled with `-g` flag then it will step in.

Profiler

https://www.thegeekstuff.com/2012/08/gprof-tutorial/

- compile with `-pg`
- generates `gmon.out` (OVERWRITES!)
- run `gprof ./program gmon.out > analysis.txt`
- **Flat profile:** shows total time spent per func call
- **Call graph:** shows graph of time spent per function and in the functions called by it

```
Flat profile:

Each sample counts as 0.01 seconds.
%    cumulative self          self   total
time seconds    seconds calls s/call s/call name
33.86 15.52     15.52    1    15.52  15.52  func2
33.82 31.02     15.50    1    15.50  15.50  new_func1
33.29 46.27     15.26    1    15.26  30.75  func1
0.07  46.30     0.03                        main

% the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.

total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
function is profiled, else blank.

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in
the gprof listing if it were to be printed.

Call graph (explanation follows)

```

```
granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds

index % time self children called name

[1]   100.0  0.03  46.27          main [1]
             15.26 15.50    1/1      func1 [2]
             15.52 0.00     1/1      func2 [3]
-----------------------------------------------
             15.26 15.50    1/1      main [1]
[2]   66.4   15.26 15.50    1     func1 [2]
             15.50 0.00     1/1      new_func1 [4]
-----------------------------------------------
             15.52 0.00     1/1      main [1]
[3]   33.5   15.52 0.00     1     func2 [3]
-----------------------------------------------
             15.50 0.00     1/1      func1 [2]
[4] 33.5     15.50 0.00     1     new_func1 [4]
-----------------------------------------------


This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:
index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function in the table.

% time This is the percentage of the `total' time that was spent
in this function and its children. Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

```

```
children This is the total amount of time propagated into this
function by its children.

called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.

name The name of the current function. The index number is
printed after it. If the function is a member of a
cycle, the cycle number is printed between the
function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the function into this parent.

children This is the amount of time that was propagated from
the function's children into this parent.

called This is the number of times this parent called the
function `/' the total number of times the function
was called. Recursive calls to the function are not
included in the number after the `/'.

name This is the name of the parent. The parent's index
number is printed after it. If the parent is a
member of a cycle, the cycle number is printed between
the name and the index number.

If the parents of the function cannot be determined, the word
`' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the child into the function.

```

```
children This is the amount of time that was propagated from the
child's children to the function.

called This is the number of times the function called
this child `/' the total number of times the child
was called. Recursive calls by the child are not
listed in the number after the `/'.

name This is the name of the child. The child's index
number is printed after it. If the child is a
member of a cycle, the cycle number is printed
between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole. This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The `+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.

Index by function name

[2] func1 [1] main
```