make and build helper/automation tools

PCSE, Spring 2018

Building programs

- You *could* write a shell script to manage your own build process
  - would become unwieldy for larger projects with many source files, libraries and dependencies
  - would need to track what needs to be rebuilt (when there are changes)
  - would need to consider how to make it portable

make

- make is a program for building programs
- generally targeted at codes with multiple source files, library dependencies, etc.
- strives to recompile only things that have changed
- portable (available on most every system)
- we will focus on GNU make
  http://savannah.gnu.org/projects/make/
  (there is also BSD make
  https://stackoverflow.com/questions/1194957/what-is-the-difference-between-gmake-and-make)

make

- make works by reading a Makefile that describes
  - files to be created
  - their dependencies
  - instructions to create the specified files

- Makefiles describe a Directed Acyclic Graph DAG of the dependencies
  - circular dependencies are dealt with by dropping the most recently discovered dependency and forging ahead
  - make then works its way (using post-order traversal) up the dependency graph building files until the goal file is up-to-date
  - only builds files whose dependencies are newer than the goal file itself

Basic Usage

- In the directory that contains your source files
- create a file called: Makefile or makefile
  - uppercase is preferred (convention) so that it comes near the top of a directory listing, but it's not required
- put the instructions for building your code in the Makefile
- Type make to build your program, i.e.

  ```
  $ make
  ```

Example

```
foo.c:
#include "bar.h"
int c=3;
int d=4;
int main()
{
   int a=2;
   return(bar(a*c*d));
}
```

```
bar.c:
#include "bar.h"
int bar(int a)
{
   int b=10;
   return(b*a);
}
```

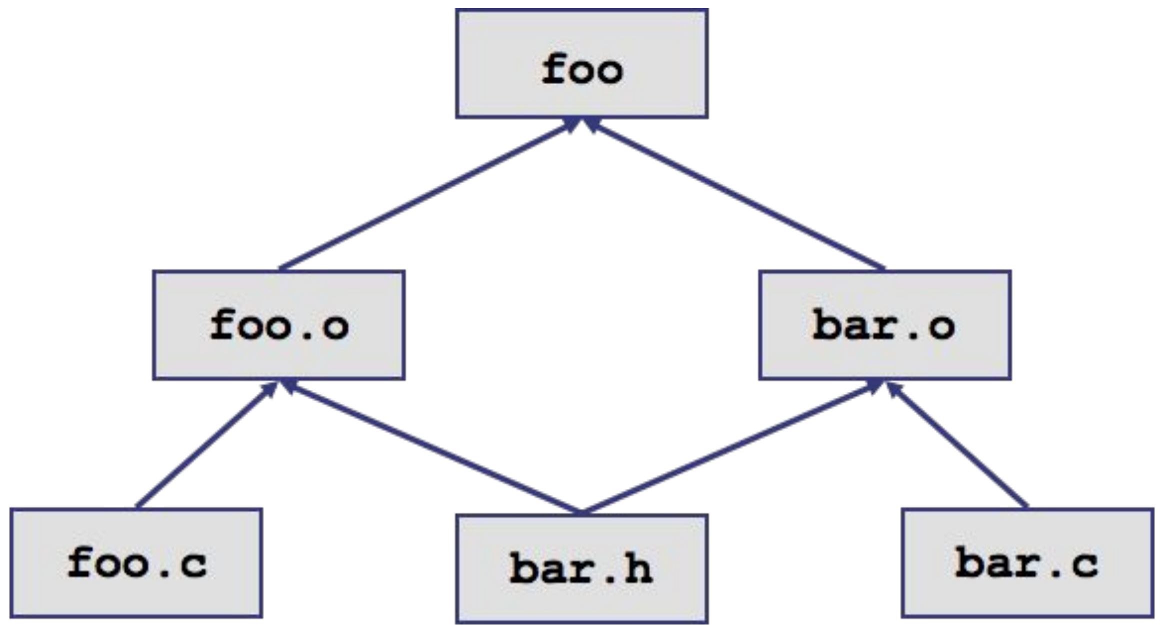
```
bar.h:
int bar(int);
```

Example

- What would the commands be to build these files?
- to build this without make, you might execute:
- `$ gcc -c foo.c` (compile without linking)
  `$ gcc -c bar.c` (compile without linking)
  `$ gcc -o foo foo.o bar.o` (Link multiple object files into one executable and name the output)

Example, dependencies and the DAG

- bar.c + bar.h -> bar.o
  foo.c + bar.h -> foo.o
  foo.o + bar.o -> foo (the executable)



Example *Makefile* (/home/adb/make_examples/Makefile)

```
target: prerequisite
<tab>  command
```

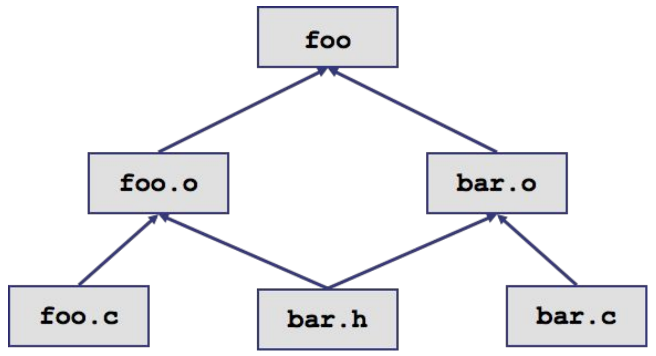
- Targets are files to be created/updated
- Prerequisites are files which must be up-to-date before the target can be update
- Commands are shell scripts (**`/bin/sh`**, preferably) used to update the target

bar.c + bar.h -> bar.o
foo.c + bar.h -> foo.o
foo.o + bar.o -> foo

```
Makefile:
foo: foo.o bar.o
<tab>gcc -o foo foo.o bar.o

foo.o: foo.c bar.h
<tab>gcc -c foo.c

bar.o: bar.c bar.h
<tab>gcc -c bar.c
```



Running make

-
  ```
  $ make
  gcc -c foo.c
  gcc -c bar.c
  gcc -o foo foo.o bar.o
  ```
- Compiles foo.c and bar.c to foo.o and bar.o
- Links foo.o and bar.o into foo, the executable
- Echoes each command as it goes
  (prefix the command with an @ to suppress this e.g.
  `@gcc -o foo foo.o bar.o`)

Makefile syntax

```
target: prerequisite
<tab> command
```

- make (generally) assumes that a line describes a target and its dependencies (and starts a rule description) unless the line begins with a TAB
- Lines that begin with a TAB are considered commands belonging to the most recent rule definition
- each line is a separate command invoked in a separate shell unless you use '\' shell continuation to tell make otherwise

```
#Makefile:
foo: foo.o bar.o
     gcc -o foo \
     foo.o bar.o
```

- If make gets confused (e.g. it encounters an extra space), it stops and prints an error message like:

```
$ make
Makefile:14: *** missing separator.  Stop.
```

- Many editors (vi and emacs) can tell when you're editing a Makefile and know to use an actual TAB character rather than expanding it to SPACEs.

Makefile Processing

- 2 phases:
    - Read in the Makefile
        - internalize variables and rules
        - construct the DAG
- Use the above to start work on dependencies
- make uses the target of the first rule it finds as the goal of the entire process if no goal is specified on the command line

improved example:

```
#Makefile

CC := gcc
foo: foo.o bar.o
   $(CC) -o $@ $^
foo.o: foo.c bar.h
   $(CC) -c $<
bar.o: bar.c bar.h
   $(CC) -c $<
```

variable

automatic variable

```
old Makefile:
foo: foo.o bar.o
   gcc -o foo foo.o bar.o
foo.o: foo.c bar.h
   gcc -c foo.c
bar.o: bar.c bar.h
   gcc -c bar.c
```

Automatic Variables:
`$@` is the target of the current rule (foo)
`$^` is all the prerequisites of the current rule (foo.o, bar.o)
`$<` is the first prerequisite of the current rule, the input file

`CC := gcc`

- Assigns a variable with the compiler name (used in the compile and linking commands) (Note: different syntax than the shell)
- Note: commands get expanded by make first then passed to the shell
- to get to a shell variable, you must escape the \$ by using \$\$ (as long as variable is not set in makefile) e.g. if you had a shell variable called \$mylibs foo: foo.o
- `$(CC) -o foo foo.o -l$$mylibs`

reference: https://gist.github.com/isaacs/62a2d1825d04437c6f08

```
$@  The file that is being made right now by this rule (aka the "target")
#     You can remember this because it's like the "$@" list in a
#     shell script.  @ is like a letter "a" for "arguments.
#     When you type "make foo", then "foo" is the argument.
# $<  The input file (that is, the first prerequisite in the list)
#     You can remember this because the < is like a file input
#     pipe in bash.  `head <foo.txt` is using the contents of
#     foo.txt as the input.  Also the < points INto the $
# $^  This is the list of ALL input files, not just the first one.
#     You can remember it because it's like $<, but turned up a notch.
#     If a file shows up more than once in the input list for some reason,
#     it's still only going to show one time in $^.
```

More make options

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
CC      := gcc
CFLAGS  := -O3
LDFLAGS := -L/usr/lib
LDLIBS  := -lm
# Rules
$(EXEC): $(OBJ)
    $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c $<
foo.o bar.o: bar.h
# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
    $(RM) $(EXEC)
clean: neat
    $(RM) $(OBJ)
neat:
    $(RM) *~ .*~
echo:
    @echo $(OBJ)
```

- Good for lots of source files
- Puts the frequently modified parts at the top
- Defines some convenience targets
- Uses some make built-in features

Comments:

```
#This is a comment
```

- An unescaped # anywhere on a line denotes the start of a comment which continues until the end of the line (just like in shell scripts)

- If you need to actually use a # in your script somewhere, you must escape it:

```
echo:
  echo foo \# bar
```

More make options:

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
CC       := gcc
CFLAGS   := -O3
LDFLAGS := -L/usr/lib
LDLIBS   := -lm
# Rules
$(EXEC): $(OBJ)
    $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c $<
foo.o bar.o: bar.h
# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
    $(RM) $(EXEC)
clean: neat
    $(RM) $(OBJ)
neat:
    $(RM) *~ .*~
echo:
    @echo $(OBJ)
```

- Good for lots of source files
- Puts the frequently modified parts at the top
- Defines some convenience targets
- Uses some make built-in features

Variables:

`x := foo`

```
 x := foo
 y := $(x) bar
 x := later
.PHONY: echo
echo:
echo $(y)    #foo bar
```

:= immediately evaluates the RHS expression and assigns its value to the LHS

unique to GNU make almost always the better choice

`x = foo`

```
 x = foo
 y = $(x) bar
 x = later
.PHONY: echo
echo:
echo $(y)   #later bar
```

= assigns the unevaluated RHS to the LHS

the expression now stored in the variable is evaluated anew every time the variable is used

need to take care that you get what you're expecting

Variables:

- SPACEs before & after the assignment operators are ignored
- However, SPACEs **after** the value are kept
- be careful not to get a variable with value: fooSPACE

```
 x = foo<space>
 y = $(x) bar
 x = later
.PHONY: echo
echo:
echo $(y).   #later bar.
```

Functions – Wildcard

```
SRC  := $(wildcard *.c)
```

- Assigns to the variable SRC all of the files in the current directory matching the glob pattern *.c (man 7 glob, for more details)
- Evaluates the wildcard function now (as opposed to when SRC is used)
- Sets SRC to bar.c foo.c

`$(patsubst pattern,replacement,text)`

Finds whitespace-separated words in *text* that match *pattern* and replaces them with *replacement*.

Here *pattern* may contain a ‘`%`’ which acts as a wildcard,

matching any number of any characters within a word.

Functions – Pattern Substitution

```
OBJ:=$(patsubst %.c,%.o,$(SRC))
```

- Changes every space-separated thing in SRC that ends in '.c' to end in '.o'
- '%' is the pattern matching operator in make
- foo.o matches '%.o' with the "stem" foo and the remainder '.o' Note: foo.out would **not** match the pattern
- be careful with the spaces around the commas!
- Evaluates immediately
- Uses the value in SRC , Sets OBJ to bar.o foo.o

More make options:

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
CC       := gcc
CFLAGS   := -O3
LDFLAGS := -L/usr/lib
LDLIBS   := -lm
# Rules
$(EXEC): $(OBJ)
    $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c $<
foo.o bar.o: bar.h
# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
    $(RM) $(EXEC)
clean: neat
    $(RM) $(OBJ)
neat:
    $(RM) *~ .*~
echo:
    @echo $(OBJ)
```

- Good for lots of source files
- Puts the frequently modified parts at the top
- Defines some convenience targets
- Uses some make built-in features

Main Goal Rule with Variables

```
$(EXEC): $(OBJ)
	$(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
```

- Makes the value of EXEC depend on the value of OBJ
  - i.e. foo depends on foo.o and bar.o
- Uses a number of Automatic Variables to construct the command
  - CC points to the C compiler (cc by default)
- LDFLAGS and LDLIBS for library paths and libraries themselves
  - **`$@`** expands to the target of the rule
  - **`$^`** expands to the list of prerequisites

Automatic Variables:
`$@` is the target of the current rule (foo)
`$^` is all the prerequisites of the current rule (foo.o, bar.o)
`$<` is the first prerequisite of the current rule, the input file

More make options:

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
CC      := gcc
CFLAGS  := -O3
LDFLAGS := -L/usr/lib
LDLIBS  := -lm
# Rules
$(EXEC): $(OBJ)
    $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c $<
foo.o bar.o: bar.h
# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
    $(RM) $(EXEC)
clean: neat
    $(RM) $(OBJ)
neat:
    $(RM) *~ .*~
echo:
    @echo $(OBJ)
```

- Good for lots of source files
- Puts the frequently modified parts at the top
- Defines some convenience targets
- Uses some make built-in features

Pattern-based rules

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

- Creates a rule for creating object files from source files with a similar name
- Uses the automatic variables again
- CFLAGS contains any compiler options needed at compile time
- $< expands to the first prerequisite

More make options (see /home/adb/make_examples/Makefile3):

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
CC      := gcc
CFLAGS  := -O3
LDFLAGS := -L/usr/lib
LDLIBS  := -lm
# Rules
$(EXEC): $(OBJ)
	$(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
%.o: %.c
	$(CC) $(CFLAGS) -c $<
foo.o bar.o: bar.h
# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
	$(RM) $(EXEC)
clean: neat
	$(RM) $(OBJ)
neat:
	$(RM) *~ .*~
echo:
	@echo $(OBJ)
```

- Good for lots of source files
- Puts the frequently modified parts at the top
- Defines some convenience targets
- Uses some make built-in features

Rules With No Commands

```
foo.o bar.o: bar.h
```

- Adds a dependency for foo.o and bar.o on bar.h
- Especially useful for C/C++ header files and Fortran includes

More make options (see /home/adb/make_examples/Makefile3):

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
CC      := gcc
CFLAGS  := -O3
LDFLAGS := -L/usr/lib
LDLIBS  := -lm
# Rules
$(EXEC): $(OBJ)
    $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c $<
foo.o bar.o: bar.h
# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
    $(RM) $(EXEC)
clean: neat
    $(RM) $(OBJ)
neat:
    $(RM) *~ .*~
echo:
    @echo $(OBJ)
```

- Good for lots of source files
- Puts the frequently modified parts at the top
- Defines some convenience targets
- Uses some make built-in features

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.

https://www.gnu.org/software/make/manual/html_node/Phony-Targets.htm
l

```
.PHONY: clobber clean neat echo
clobber: clean
   $(RM) $(EXEC)
clean: neat
   $(RM) $(OBJ)
neat:
   $(RM) *~ .*~
echo:
   @echo $(OBJ)
```

- Targets listed as the dependencies of .PHONY do not reference files
- Are always treated as out-of-date

```
$ make clobber
```

- Invokes the commands for neat and clean and then its own command

useful for cleaning up a directory

- clean or neat could also be invoked to clean up less files
- echo useful in debugging your Makefiles
  (you've probably encountered make install)

Running the improved Makefile:

```
make
gcc -O3 -c bar.c
gcc -O3 -c foo.c
gcc -L/usr/lib -lm -o foo bar.o foo.o
```

- Order is different
- Now has our extra options

Building only parts

```
$ make bar.o
gcc -O3   -c -o bar.o bar.c
```

- Can be used with any target (or list of targets)
- Useful when you just want to check syntax, etc. while you’re developing

Running the improved Makefile:

```
$ make clobber
rm -f *~ .*~
rm -f bar.o foo.o
rm -f foo
```

- Good for cleaning up and starting over
- Tilde files (*~, .*~) are usually backup files from your text editor (be careful there aren't any spaces in there!)

Adding helpful predefined variables:

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
CC      := gcc
CFLAGS  := -O3
LDFLAGS := -L/usr/lib
LDLIBS  := -lm
# Rules
$(EXEC): $(OBJ)
    $(LINK.o) $(LDLIBS) -o $@ $^
%.o: %.c
    $(COMPILE.c) $<
foo.o bar.o: bar.h
# Useful phony targets
.PHONY: clobber clean neat echo
clobber: clean
    $(RM) $(EXEC)
clean: neat
    $(RM) $(OBJ)
neat:
    $(RM) *~ .*~
echo:
    @echo $(OBJ)
```

```
$(LINK.o)=$(CC) $(LDFLAGS) $(TARGET_ARCH)
$(COMPILE.c)=$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
```

- Note the use of `=` rather than :=
- Allows you to change each of the internal variables (like $(CC)) before you use it
- $(TARGET_ARCH) is empty by default (could add flags specific to your platform)

Finding the built-in stuff:

```
$ make -n -p | more
```

- -n tells make to do a dry-run
- prints the commands that it would run,
- ... but doesn’t actually do anything
- -p tells make to print out
- all its rules
- all its variables
- Searching through the output of this can tell you lots of useful things

Conditionals: example DEBUG mode, Makefile3

```
ifdef DEBUG_MODE
CFLAGS  := -g
else
CFLAGS  := -O3
endif
```

- Looks to see if DEBUG_MODE is defined (i.e., has any value)
- Sets CFLAGS accordingly
- Space between ifdef and its arguments is important

```
$ make DEBUG_MODE=asdfsa
gcc -g   -c -o foo.o foo.c
gcc -g   -c -o bar.o bar.c
gcc -L/usr/lib  foo.o bar.o  -lm -o foo

$ export DEBUG_MODE=asdfsa
make
gcc -g   -c -o foo.o foo.c
gcc -g   -c -o bar.o bar.c
gcc -L/usr/lib  foo.o bar.o  -lm -o foo
```

Conditionals Syntax

```
ifeq (ARG1,ARG2)
ifneq (ARG1,ARG2)
ifdef VARIABLE-NAME
ifndef
  VARIABLE-NAME
```

```
ifndef DEBUG_MODE
  CFLAGS  := -O3
else
  CFLAGS  := -g
endif
```

Conditionals

```
ifneq ($(DEBUG_MODE),yes)
  CFLAGS  := -O3
else
  CFLAGS  := -g
endif
```

```
$ make DEBUG_MODE=asdf
gcc -O3   -c -o foo.o foo.c
gcc -O3   -c -o bar.o bar.c
gcc -L/usr/lib  foo.o bar.o  -lm -o foo
$ make DEBUG_MODE=yes
make: `foo' is up to date.
$ make clobber
rm -f *~ .*~
rm -f bar.o foo.o
rm -f foo
$ make DEBUG_MODE=yes
gcc -g   -c -o foo.o foo.c
gcc -g   -c -o bar.o bar.c
gcc -L/usr/lib  foo.o bar.o  -lm -o foo
```

Includes

- make can include pieces of Makefiles to build up the Makefile it is working with

Makefile:

```
# Files
EXEC := foo
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
include Makefile_options.inc
# Rules
$(EXEC): $(OBJ)
$(OBJ): bar.h
include Makefile_phonies.inc
```

Makefile_options.inc:

```
CC      := gcc
ifneq ($(DEBUG_MODE),yes)
  CFLAGS  := -O3
else
  CFLAGS  := -g
endif
LDFLAGS := -L/usr/lib
LDLIBS  := -lm
```

make as a general tool

- Not restricted to compiling source codes
- Can be used to manage any project which requires tracking dependencies
- The command portion can be used to run any commands (shell scripts, mail, etc.)

e.g. book, rendering POV files:

```
image:   image.pov
         povray -W720 -H480 image.pov

big:     image.pov
         povray -W1920 -H1080 -D image.pov

image.pov:   header.pov C.pov H.pov S.pov
         cat header.pov C.pov H.pov S.pov > image.pov

C.pov:   T150m.xfg
         cat T150m.xfg | grep "C" | sed -e "s/C/.19\t1.000000/" | gfg2pov -no_header      \
          -color Grey -phong 1 > C.pov

H.pov:   T150m.xfg
         cat T150m.xfg | grep "H" | sed -e "s/H/.07\t1.000000/" | gfg2pov -no_header  \
          -color White -phong 1 > H.pov

S.pov:       T150m.xfg
         cat T150m.xfg | grep "S" | sed -e "s/S/.17\t1.000000/" | gfg2pov -no_header  \
         -color Yellow -phong 1 > S.pov

clean:
         rm C.pov H.pov S.pov image.pov
```

https://github.com/waveform80/flight-render/blob/master/Makefile

build helpers

- gnu autotools
  https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html
  https://en.wikipedia.org/wiki/GNU_Build_System

- cmake
  https://cmake.org/

- scons
  http://scons.org/

(will go over gnu auto tools and cmake briefly today and in more detail when we need it for the software we are using)

cmake

- Define “targets”
  – Binary executable programs
  – Libraries
  – Custom targets

- Define “dependencies”.
  – How binaries depend on source files
  – How binaries depend on libraries.
  – How libraries depend on source.

- Definitions in CMakeLists.txt
  – Macro style of syntax.

demo:

- Cmake is used by many open source projects for build system management.
- Cmake is a powerful means of generating build systems.
- Edit the CMakeCache.txt file to change default settings (or erase it and start again)
- The makefile created by Cmake is not really meant to be edited by hand.