

Dr. Christopher S. Simmons (csim@ices.utexas.edu)

Make

CSE 380: Tools and Techniques
of Computational Science

Use Case

- ❖ Scientific codes tend towards the small end
 - ❖ not usually as long or as complex as OSes, word processors, and web browsers
 - ❖ usually have small development teams
- ❖ Dozens to hundreds of files
- ❖ $O(10K)$ – $O(100K)$ lines of code
- ❖ You could just write a shell script to manage your build process
 - ❖ probably more trouble than it's worth
 - ❖ make already exists
 - ❖ ...and it knows what to and not to build

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
- ❖ There are two common flavors of make
 - ❖ GNU
 - ❖ portable (available for just about any architecture including your toaster, certainly portable to anywhere GCC can be built)
 - ❖ default/only make on Linux
 - ❖ we'll concentrate on this
 - ❖ <http://savannah.gnu.org/projects/make/>
 - ❖ BSD
 - ❖ default on many other UNIXes
 - ❖ usually much less feature-rich than GNU make
 - ❖ gmake is GNU make's name on these systems

Make

- ❖ make works by reading a Makefile which describes
 - ❖ files to be created
 - ❖ their dependencies
 - ❖ instructions to create the specified files
- ❖ Makefiles describe a DAG of the dependencies
 - ❖ Directed Acyclic Graph
 - ❖ circular dependencies are dealt with by dropping the most recently discovered dependency and forging ahead (more later)
- ❖ 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 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.
`localhost$ make`

Example Makefile

- ❖ Recall the silly example code from last time
 - ❖ `bar.c + bar.h -> bar.o`
 - ❖ `foo.c + bar.h -> foo.o`
 - ❖ `foo.o + bar.o -> foo` (the executable)
- ❖ What does the DAG for this look like? (Hint: I just wrote a flattened version of it)
- ❖ What would a Makefile describing this DAG look like?
- ❖ What would the commands be to build these files?
 - ❖ to build this without make, you might execute:

```
localhost$ gcc -c foo.c
localhost$ gcc -c bar.c
localhost$ gcc -o foo foo.o bar.o
```

Silly Example

foo.c:

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

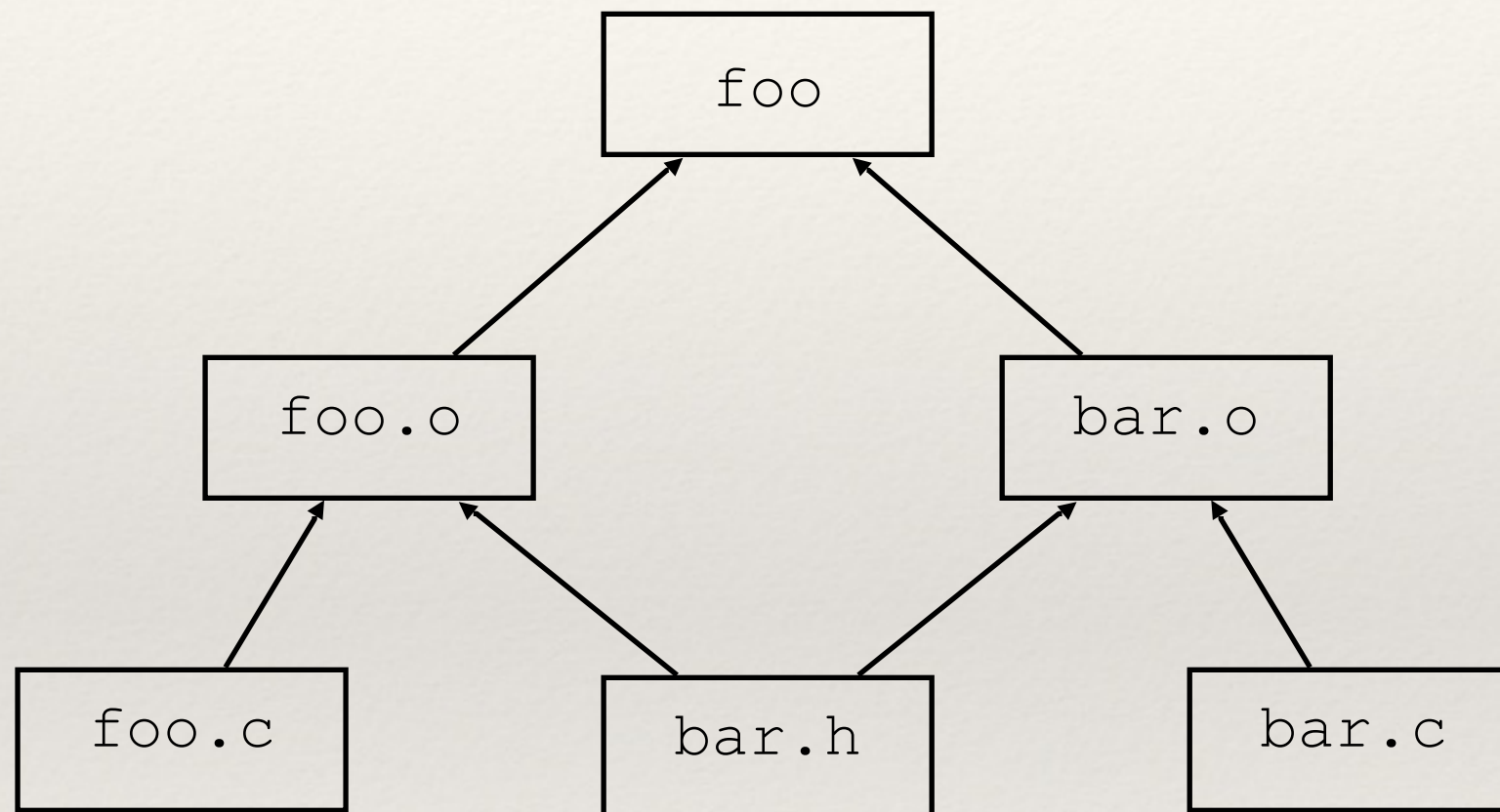
bar.c:

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

bar.h:

```
int bar(int);
```

Simple DAG



Example Makefile

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
```

Basic Makefile Rules

```
target: prerequisite  
command
```

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

More Syntax

- ❖ 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
- ❖ If make gets confused, it stops and prints an error message like:

```
localhost$ make  
Makefile:14: *** missing separator.  Stop.
```
- ❖ Most editors can tell when you’re editing a Makefile and know to use an actual TAB character rather than expanding it to SPACES.
 - ❖ vi (or it’s progeny, e.g. vim, gvim, etc.) does
 - ❖ emacs certainly does
 - ❖ YMMV with other editors (look for a “makefile” mode)

Makefile Processing

- ❖ 2 phases
 - ❖ Read in the Makefile
 - ❖ internalize variables and rules
 - ❖ construct the DAG
 - ❖ Use the above to start building 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

Running make

```
lslogin1$ 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

A Slightly 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 $<
```

A Slightly Improved Example

- ❖ Assigns a variable with the compiler name
 - ❖ used in the compile and linking commands
 - ❖ different syntax than the shell
 - ❖ commands get expanded by make first then passed to the shell
 - ❖ to get to a shell variable, you must escape the '\$' by using '\$\$'
 - ❖ `foo: foo.o`
 - ❖ `$(CC) -o foo foo.o -l$$mylibrary`
- ❖ Uses some Automatic Variables
 - ❖ `$@` is the target of the current rule
 - ❖ `^` is all the prerequisites of the current rule
 - ❖ `<` is the first prerequisite of the current rule

More Complex/Useful Makefile

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

```
# 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)
```

Comments

`# Files`

- ❖ 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`

Variables

`EXEC := foo`

- ❖ Defines the variable EXEC with the value foo, i.e. the name of the program we want to build
 - ❖ there's nothing special about the name EXEC
 - ❖ however, there are many special make variables
 - ❖ predefined values
 - ❖ predefined uses
- `:=` immediately evaluates the RHS expression and assigns its value to the LHS
 - ❖ almost always the better choice
 - ❖ unique to GNU make

Variables

- ❖ = 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
- ❖ SPACES before and after the assignment operators are ignored
 - ❖ However, SPACES after the value are kept
 - ❖ be careful not to get a variable whose value is: fooSPACE

Variables

- ❖ Variables can be defined in three places
 - ❖ in the Makefile, like the above example
 - ❖ in your shell environment

```
lslogin1$ export FOO="foo"
```

or

```
setenv FOO "foo"
```
 - ❖ on the command line

```
make FOO="foo"
```
- ❖ All variables used the same way
 - ❖ `$(FOO)`

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”

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'`
 - ❖ `foo.out` would not match the pattern
 - ❖ be careful with the spaces around the commas!
- ❖ Evaluates immediately
- ❖ Uses the value in SRC
 - ❖ notice the syntax (it's similar to, but different from, shell variable syntax)
- ❖ Sets OBJ to `“bar.o foo.o”`

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

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

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

Phony Targets

.PHONY: clobber clean neat echo

clobber: clean

\$(RM) \$(EXEC)

clean: neat

\$(RM) \$(OBJ)

neat:

\$(RM) *~ . *~

echo:

@echo \$(OBJ)

Phony Targets

- ❖ Targets listed as the dependencies of .PHONY do not reference files
 - ❖ Are always treated as out-of-date
- ```
localhost$ 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 fewer files
  - ❖ echo useful in debugging your Makefiles

---

# Running the Improved *Makefile*

---

```
lslogin1$ make
gcc -O3 -c bar.c
gcc -O3 -c foo.c
gcc -L/usr/lib -lm -o foo bar.o foo.o
```

❖ Now has our extra options

---

# Building Only Parts

---

```
lslogin1$ 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

---

```
lslogin1$ 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!)

---

# An Even Better Makefile

---

```
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)
```

---

# Helpful Predefined Variables

---

```
$ (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



---

# Finding the Built-in Stuff

---

```
lslogin1$ 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
- ❖ You can also look in the manual (more later)

---

# Alternate Makefile Names

---

- ❖ make looks for Makefile and makefile in the current directory by default
- ❖ You can give a file with any name you like using *make -f mymakefilename*

---

# Conditionals

---

```
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



---

# Conditionals

---

```
lslogin1$ make DEBUG_MODE=asdfs
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

---

- ❖ `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
```

```
lslogin1$ 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
lslogin1$ make DEBUG_MODE=yes
make: `foo' is up to date.
lslogin1$ make clobber
rm -f *~ .*~
rm -f bar.o foo.o
rm -f foo
lslogin1$ 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
```

---

# More info(1)

---

- ❖ info(1) is yet another set of manual pages available
- ❖ Generally associated with GNU packages
- ❖ GNU package man pages may be terse and then refer to the info page
  - ❖ lslogin1\$ info make
- ❖ Brings up the info pages for make
- ❖ Type 'h' to get a tutorial on how to use info