## Build Tools (make) CSE 333 Autumn 2018

### **Instructor:** Hal Perkins

#### **Teaching Assistants:**

| Tarkan Al-Kazily | Renshu Gu |
|------------------|-----------|
| Harshita Neti    | Thai Pham |
| Soumya Vasisht   | Yifan Xu  |

Travis McGaha Forrest Timour

## **Lecture Outline**

\* Make and Build Tools

### make

- make is a classic program for controlling what gets (re)compiled and how
  - Many other such programs exist (e.g. ant, maven, IDE "projects")
- make has tons of fancy features, but only two basic ideas:
  - 1) Scripts for executing commands
  - 2) Dependencies for avoiding unnecessary work
- To avoid "just teaching make features" (boring and narrow), let's focus more on the concepts...

# **Building Software**

- Programmers spend a lot of time "building"
  - Creating programs from source code
  - Both programs that they write and other people write
- Programmers like to automate repetitive tasks
  - Repetitive: gcc -Wall -g -std=c11 -o widget foo.c bar.c baz.c
    - Retype this every time:
    - Use up-arrow or history:
    - Have an alias or bash script:
    - Have a Makefile:



# "Real" Build Process

- On larger projects, you can't or don't want to have one big (set of) command(s) that redoes everything every time you change anything:
  - If gcc didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
  - 2) If source files have multiple outputs (*e.g.* javadoc), you'd have to type out the source file name(s) multiple times
  - 3) You don't want to have to document the build logic when you distribute source code
  - 4) You don't want to recompile everything every time you change something (especially if you have 10<sup>5</sup>-10<sup>7</sup> files of source code)
- A script can handle 1-3 (use a variable for filenames for 2), but
   4 is trickier

## **Recompilation Management**

- The "theory" behind avoiding unnecessary compilation is a *dependency dag* (directed, acyclic graph)
- To create a target t, you need sources s<sub>1</sub>, s<sub>2</sub>, ..., s<sub>n</sub> and a command c that directly or indirectly uses the sources
  - It t is newer than every source (file-modification times), assume there is no reason to rebuild it
  - Recursive building: if some source s<sub>i</sub> is itself a target for some other sources, see if it needs to be rebuilt...
  - Cycles "make no sense"!



Compiling a .c creates a .o – the .o depends on the .c and all included files (.h, recursively/transitively)



- Compiling a .c creates a .o the .o depends on the .c and all included files (.h, recursively/transitively)
- An archive (library, .a) depends on included .o files



- ★ Compiling a .c creates a .o the .o depends on the .c
   and all included files (.h, recursively/transitively)
- An archive (library, .a) depends on included .o files
- Creating an executable ("linking") depends on . o files and archives
  - Archives linked by -L<path> -l<name>
     (e.g. -L. -lfoo to get libfoo.a from current directory)



- If one .c file changes, just need to recreate one .o file, maybe a library, and re-link
- If a . h file changes, may need to rebuild more
- Many more possibilities!

### make Basics

A makefile contains a bunch of triples:

target: sources
 ← Tab → command

- Colon after target is *required*
- Command lines must start with a TAB, NOT SPACES
- Multiple commands for same target are executed in order
  - Can split commands over multiple lines by ending lines with '  $\setminus$  '
- Example:

foo.o: foo.c foo.h bar.h
gcc -Wall -o foo.o -c foo.c

## Using make

### bash% make -f <makefileName> target

- Defaults:
  - If no -f specified, use a file named Makefile
  - If no target specified, will use the first one in the file
  - Will interpret commands in your default shell
    - Set SHELL variable in makefile to ensure
- Target execution:
  - Check each source in the source list:
    - If the source is a target in the Makefile, then process it recursively
    - If some source does not exist, then error
    - If any source is newer than the target (or target does not exist), run command (presumably to update the target)

## make Variables

- You can define variables in a makefile:
  - All values are strings of text, no "types"
  - Variable names are case-sensitive and can't contain ':', '#', '=', or whitespace

### Advantages:

- Easy to change things (especially in multiple commands)
- Can also specify on the command line (CC=clang FLAGS=-g)

C

## **More Variables**

It's common to use variables to hold list of filenames:

```
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
    gcc -o widget $(OBJFILES)
clean:
    rm $(OBJFILES) widget *~
```

- \* clean is a convention
  - Remove generated files to "start over" from just the source
  - It's "funny" because the target doesn't exist and there are no sources, but it works because:
    - The target doesn't exist, so it must be "remade" by running the command
    - These "phony" targets have several uses, such as "all"...

## "all" Example

```
all: prog B.class someLib.a
      # notice no commands this time
prog: foo.o bar.o main.o
      qcc -o prog foo.o bar.o main.o
B.class: B.java
      javac B.java
someLib.a: foo.o baz.o
      ar r foo.o baz.o
foo.o: foo.c foo.h header1.h header2.h
      qcc -c -Wall foo.c
# similar targets for bar.o, main.o, baz.o, etc...
```

# Makefile Example

"talk" program (find files on web with lecture slides)



speak.h

speak.c

shout.h



# **Revenge of the Funny Characters**

- Special variables:
  - \$@ for target name
  - \$^ for all sources
  - \$< for left-most source</p>
  - Lots more! see the documentation

Examples:

# CC and CFLAGS defined above
widget: foo.o bar.o
 \$(CC) \$(CFLAGS) -o \$@ \$^
foo.o: foo.c foo.h bar.h
 \$(CC) \$(CFLAGS) -c \$<</pre>

## And more...

- There are a lot of "built-in" rules see documentation
- There are "suffix" rules and "pattern" rules
  - Example: %.class: %.java
    javac \$< # we need the \$< here</pre>
- Remember that you can put *any* shell command even whole scripts!
- You can repeat target names to add more dependencies
- Often this stuff is more useful for reading makefiles than writing your own (until some day...)

## Extra Exercise #1

- Modify the linked list code from Lecture 5 Extra Exercise #1
  - Add static declarations to any internal functions you implemented in linkedlist.h
  - Add a header guard to the header file
  - Write a Makefile
    - Use Google to figure out how to add rules to the Makefile to produce a library (liblinkedlist.a) that contains the linked list code