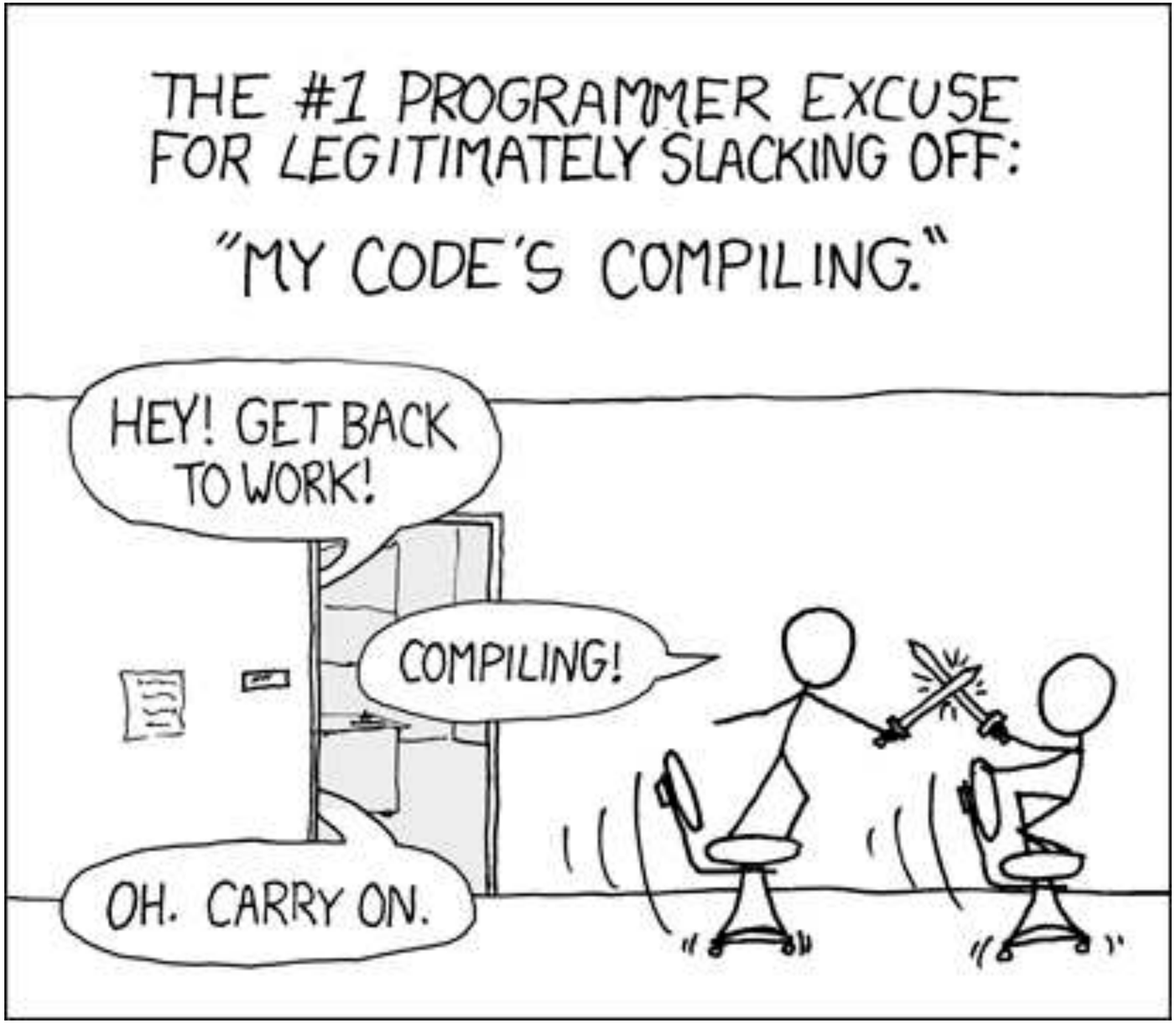
CS 241: Systems Programming
Lecture 11. Make and Compiling

Fall 2019

Prof. Stephen Checkoway



https://xkcd.com/303/

Only compile what you need

All at once

- ~~`$ clang -std=c11 -Wall -o program *.c`~~

One file at a time with separate linking step

- ```
  $ clang -std=c11 -Wall -c -o foo.o foo.c
  $ clang -std=c11 -Wall -c -o bar.o bar.c
  $ clang -std=c11 -Wall -c -o qux.o qux.c
  $ clang -o program foo.o bar.o qux.o
  ```
  ```

One option: Make a script

Are there any problems with this approach?

```
#!/bin/bash

changed=no
if [[ ! -f foo.o || foo.c -nt foo.o ]]; then
  clang -std=c11 -Wall -c -o foo.o foo.c
  changed=yes
fi
if [[ ! -f bar.o || bar.c -nt bar.o ]]; then
  clang -std=c11 -Wall -c -o bar.o bar.c
  changed=yes
fi
if [[ ${changed} = yes ]]; then
  clang -o myprogram foo.o bar.o
fi
```

Potential problems

Lots of very similar code (imagine many different files)

Didn't mention header files! Need to recompile if any included header files change

Enter Make

A `Makefile` consists of a set of rules like

```
myprogram: main.o foo.o bar.o
        clang -o myprogram main.o foo.o bar.o -lsomelibrary
```

The target (myprogram) names the file to be created

The prerequisites (main.o foo.o bar.o) name the files that are required to exist or be built before the target can be made

The recipe (clang ...) specifies a sequence of shell commands to run

- Each line is its own command
- Each line must start with a literal tab character, spaces won't work

Making things

At the command line

- `$ make target1 ... targetn`
- `$ make # makes the first target in the Makefile`

Make creates a list of the prerequisites of each target and their prerequisites and their prerequisites and…

For each item in the list, make decides if it is up to date (newer than each of its prerequisites) or if it needs to run a recipe to remake it

Make then runs the necessary rules to make all the targets

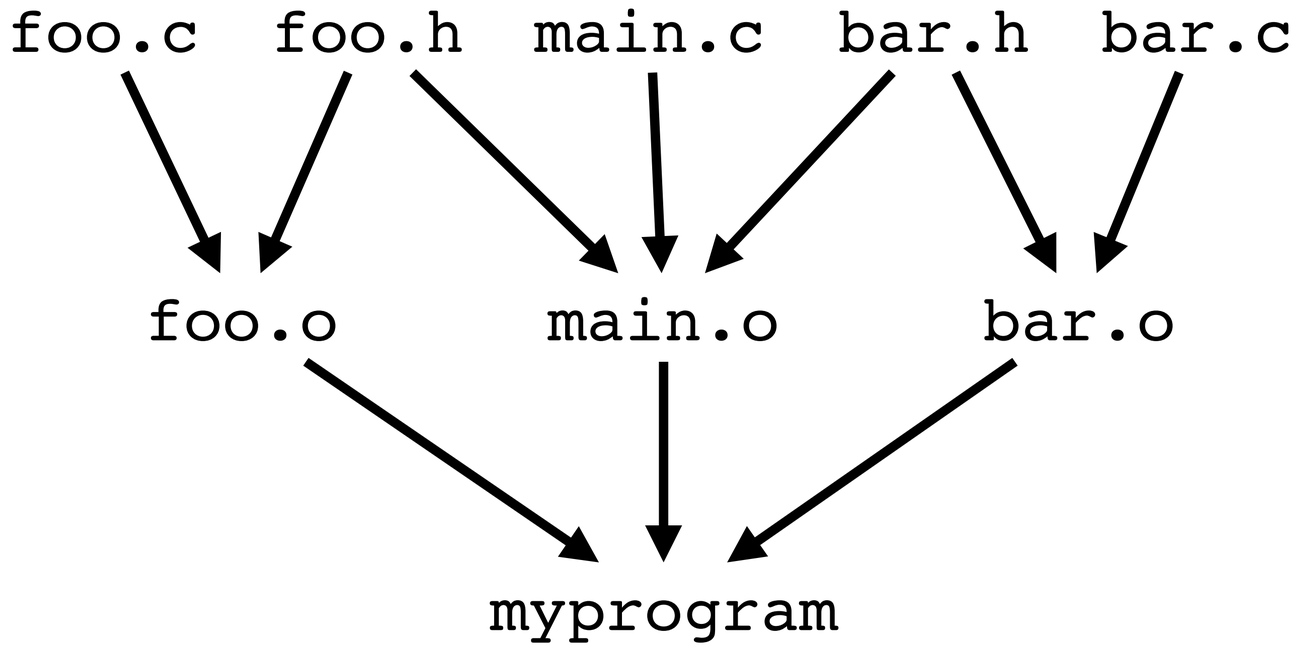
Prerequisite example

`myprogram` depends on `main.o`, `foo.o`, and `bar.o`

`bar.o` depends on `bar.c` and `bar.h`

`foo.o` depends on `foo.c` and `foo.h`

`main.o` depends on `main.c`, `foo.h` and `bar.h`



If foo.c changes, which files need to be remade?

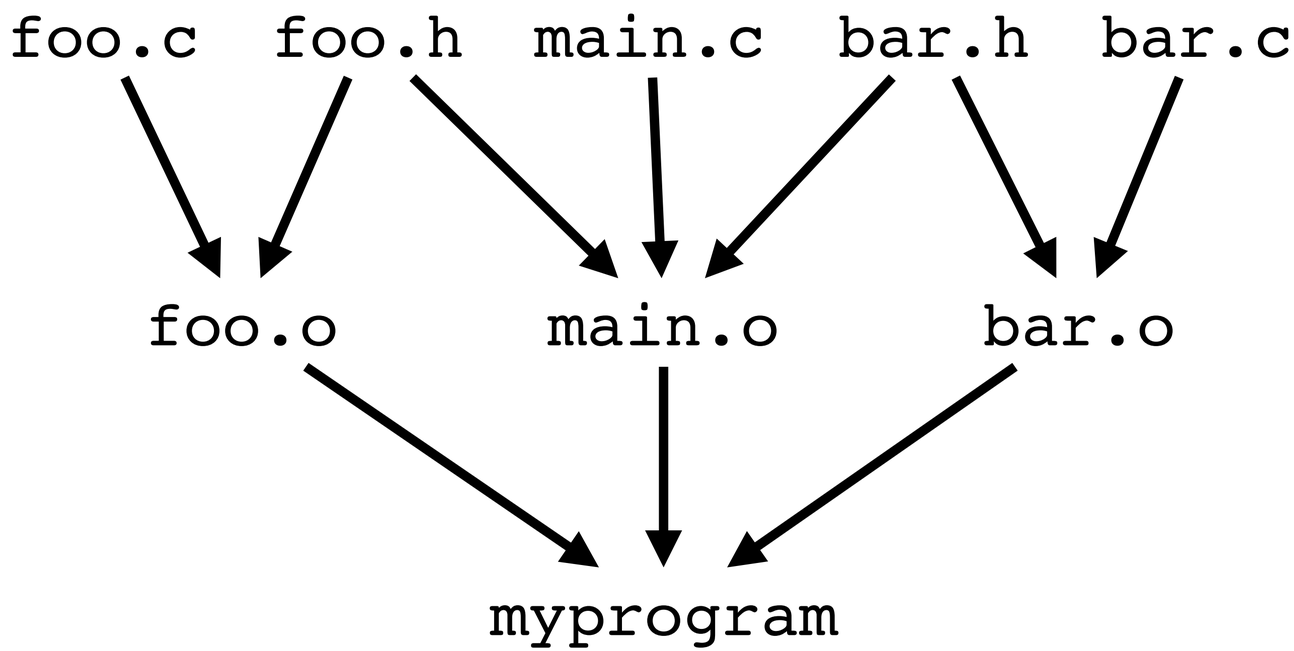
A. foo.o

B. foo.o and myprogram

C. foo.o, main.o, and myprogram

D. foo.o, main.o, bar.o, and myprogram

E. None



If foo.h changes, which files need to be remade?

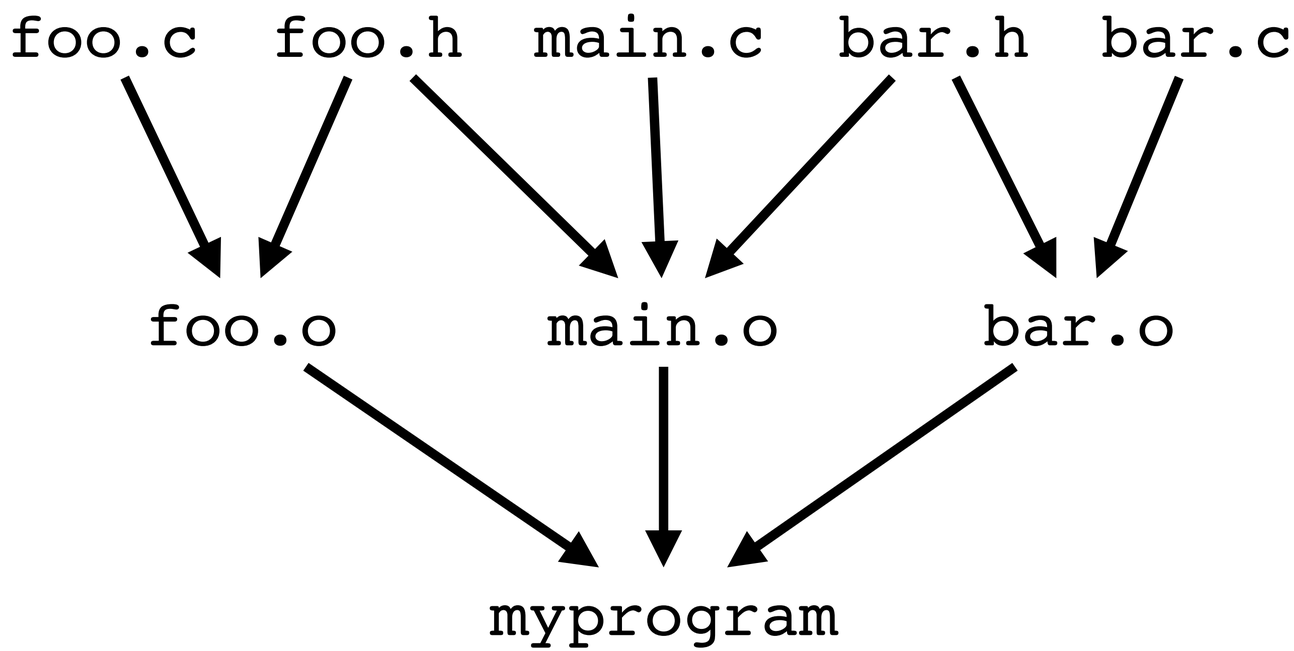
A. foo.o

B. foo.o and myprogram

C. foo.o, main.o, and myprogram

D. foo.o, main.o, bar.o, and myprogram

E. None



A complete Makefile

```
myprogram: main.o foo.o bar.o
        clang -o myprogram main.o foo.o bar.o

main.o: main.c foo.h bar.h
        clang -std=c11 -Wall -c -o main.o main.c

foo.o: foo.c foo.h
        clang -std=c11 -Wall -c -o foo.o foo.c

bar.o: bar.c bar.h
        clang -std=c11 -Wall -c -o bar.o bar.c
```

DRY: Don't Repeat Yourself

```
CC := clang
CFLAGS := -std=c11 -Wall
objs := main.o foo.o bar.o

myprogram: $(objs)
	$(CC) -o myprogram $(objs)

main.o: main.c foo.h bar.h
	$(CC) $(CFLAGS) -c -o main.o main.c

foo.o: foo.c foo.h
	$(CC) $(CFLAGS) -c -o foo.o foo.c

bar.o: bar.c bar.h
	$(CC) $(CFLAGS) -c -o bar.o bar.c
```

Pattern rules and automatic vars

Pattern rule

- target contains one % which matches any nonempty sequence of chars
- prerequisites can also (and usually do) contain % which match

Automatic variables (the two most useful)

- `$@` is set to the rule's target
- `$<` is set to the first prerequisite

Example rule to compile .c files to .o files

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

DRYer

```
CC := clang
CFLAGS := -std=c11 -Wall
objs := main.o foo.o bar.o

myprogram: $(objs)
	$(CC) -o $@ $(objs)

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

But wait, we forgot something!

Makefile: complete

```
CC := clang
CFLAGS := -std=c11 -Wall
objs := main.o foo.o bar.o

myprogram: $(objs)
	$(CC) -o $@ $(objs)

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

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

Phony targets

Phony targets don't correspond to actual files

Phony targets' recipes always run if the target is specified (or is a prerequisite of a target to be built)

Common phony targets

- all: Usually the first target, depends on the program(s) to be built
- clean: Removes all of the files built by make

```
.PHONY: all clean
all: myprogram

clean:
	$(RM) myprogram $(objs)
```

Generating Makefiles (opinion!)

GNU Automake (together with GNU Autoconf) generate Makefiles

- Really great to use as a user, you run
  $ ./configure
  $ make
  $ make install
- Horrible for developers (need to know m4, autoconf, automake, make, sh)

Cmake

- Horrible for users, a huge hassle for developers
- Can generate Makefiles, Ninja files (Ninja is great!), or IDE configurations

Make Makefile (mkmf): The less said about it the better

Generating dependencies

makedepend

- Pretty simple to use to generate the dependencies
- Modifies the Makefile itself!
- Doesn't play nice with version control

The compiler knows what files it needs to compile each source file, you can ask it to dump out the dependency information in Makefile format (usually .d files) and the include that information

- Complicated to get right
- Fantastic when it works
- http://make.mad-scientist.net/papers/advanced-auto-dependency-generation/#include

In-class exercise

https://checkoway.net/teaching/cs241/2019-fall/exercises/Lecture-11.html

Grab a laptop and a partner and try to get as much of that done as you can!