

Build Tools (make)

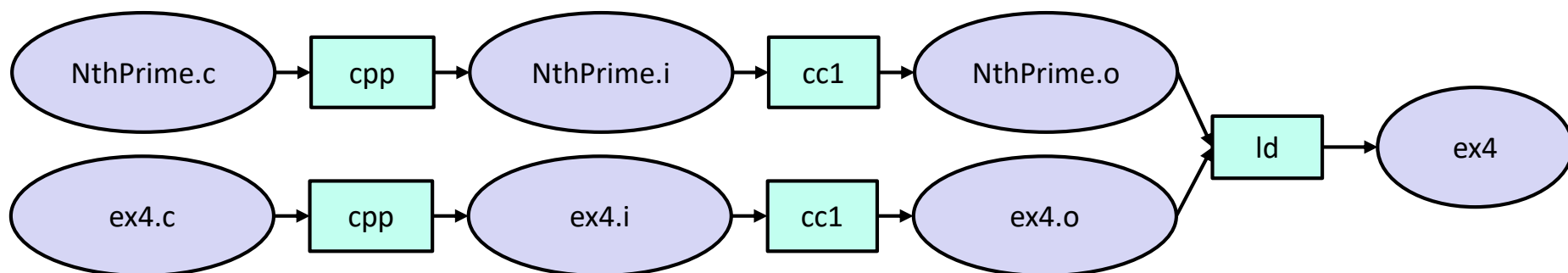
CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:

Deeksha Vatwani	Hannah Jiang	Jen Xu
Justin Tysdal	Leanna Nguyen	Sayuj Shahi
Wei Wu	Yiqing Wang	Youssef Ben Taleb

- ❖ Assume `NthPrime.h`, `NthPrime.c`, and `ex4.c`, as in our most-recently completed exercise



- ❖ ... except we are now precalculating the first 20 primes. What are the contents of each `.o`?

```
#define NUM_PRECALC 20
int16_t kPrecalculated[NUM_PRECALC] =
    {2, 3, 5, 7, /* etc */};

int16_t NthPrime(int16_t n);
```

NthPrime.h

Lecture Outline

- ❖ **A Few Words about Code Quality**
- ❖ Make and Build Tools

Code Quality

- ❖ Code quality **really** matters – and not just for homework
- ❖ The quality rules we follow are distilled from almost 50 years of bug(fixe)s
 - Bad casts = data loss or data corruption
 - Memory leaks = unexpected crashes or data corruption
 - Non-standard input/output parameters = incorrect usage by callers
 - We saw that on Ed this weekend!
 - And so so so so so much more ...

HashTable_Insert create new pointer before pushing #244

 **Anonymous Mouse**
19 hours ago in **Homeworks - HW1 (Data Structures)**

 PIN  STAR  WATCH **187** VIEWS

 4

Thought this might be helpful for other people who might have struggled with the same issue, when pushing the newkeyvalue to the chain in hashtable_insert, **DO NOT** directly push into the linkedlist with &newkeyvalue, instead create a new pointer and dereference the pointer to copy newkeyvalue then pushing the newly created pointer instead into the linkedlist.

Hope this will help someone.

Code Quality Rules

- ❖ Rule 0: The reader's time is *much* more important than the writer's
 - Clarity/understandability is critical
 - *(yes, we're talking about code comments)*
 - What does the reader need to know to understand / modify / use the code, that can't be discovered by reading the code itself?
 - Good comments ultimately save the writer's time, too!

HashTable_Insert create new pointer before pushing #244

 **Anonymous Mouse**
19 hours ago in [Homeworks - HW1 \(Data Structures\)](#)

 PIN  STAR  WATCH 187 VIEWS

 4

Thought this might be helpful for other people who might have struggled with the same issue, when pushing the newkeyvalue to the chain in hashtable_insert, **DO NOT** directly push into the linkedlist with &newkeyvalue, instead create a new pointer and dereference the pointer to copy newkeyvalue then pushing the newly created pointer instead into the linkedlist.

Hope this will help someone.

Code Quality Rules

❖ Rule 1: Match existing code

- Do output parameters go at the end of the param list? The beginning?
- Yes, whitespace does matter!
 - `char** argv` vs `char **argv` "reads differently" to novice programmers!
 - There's a reason Google's style guide is so pedantic about whitespace

❖ Rule 2: Make use of the tools provided to you

- Compiler: fix the warnings!
- Valgrind: fix all of them unless you know why it's *not* an error
- style checkers: fix most things; be sure you understand anything you don't fix and can justify it (use of long, types in sizeof(), readdir, not much else)

Lecture Outline

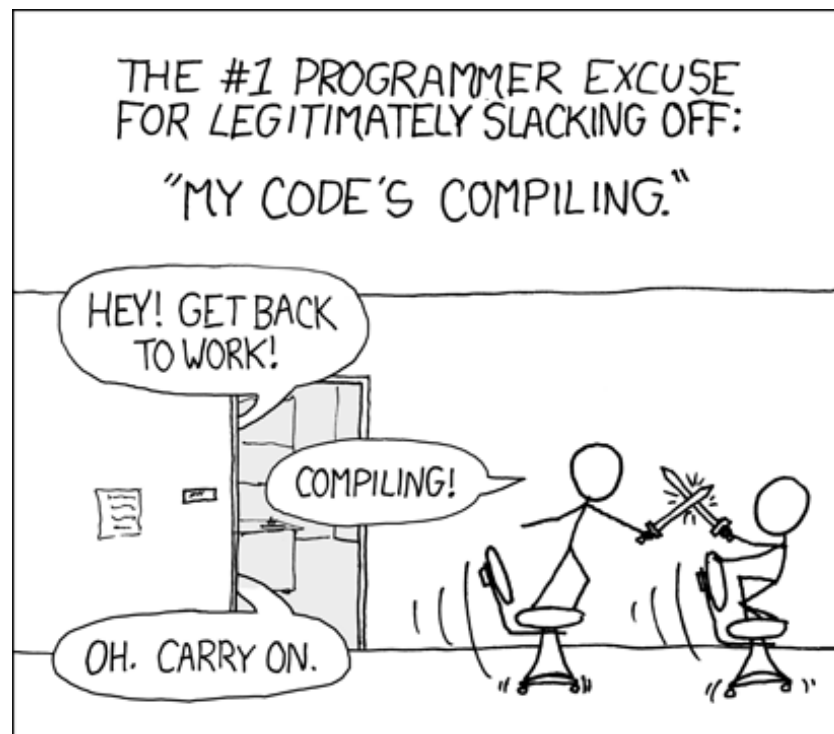
- ❖ A Few Words about Code Quality
- ❖ **Make and Build Tools**

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



<https://xkcd.com/303/>

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=c17 -o widget foo.c bar.c baz.c`
 - Retype this every time: 
 - Use up-arrow or history:  (still retype after logout)
 - Have an alias or bash script: 
 - Have a Makefile:  (you're ahead of us)

“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:
 - 1) 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 for each output command
 - 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 10^5 - 10^7 files of source code)
- ❖ A script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

An Example

- ❖ We have a small program that is split into multiple tiny modules (code on the web linked to this lecture):

`main.c``spea.k.h``spea.k.c``shout.h``shout.c`

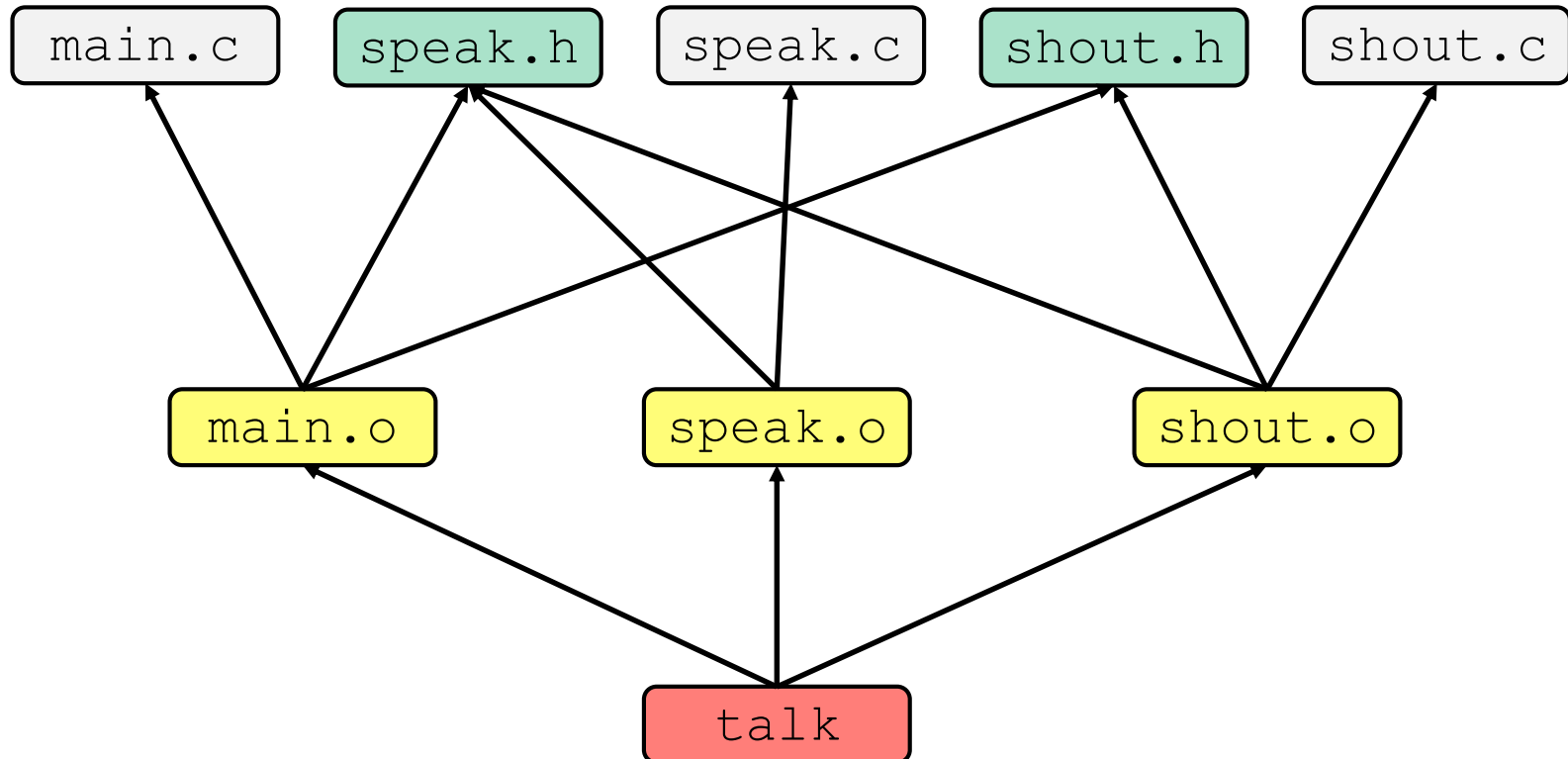
- ❖ Modules:
 - `spea.k.h/spea.k.c`: write a string to stdout
 - `shout.h/shout.c`: write a string to stdout LOUDLY
 - `main.c`: client program
- ❖ Demo: build this program incrementally, and recompile only necessary parts when something changes
- ❖ How do we automate this “minimal rebuild”?

Recompilation Management

- ❖ The “theory” behind avoiding unnecessary compilation is a *dependency DAG* (**d**irected, **a**cyclic **g**raph)
- ❖ To create a target t , you need sources s_1, s_2, \dots, s_n 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_i is itself a target for some other sources, see if it needs to be rebuilt...
 - Cycles “make no sense”!

Theory Applied to Our Example

- ❖ What are the dependencies between built and source files?
- ❖ What needs to be rebuilt if something changes?



- ❖ Draw the dependency graph for `example_program_ll.c` and `example_program_ht.c`
 - *(ignore the existence of `CSE333.h`, `libhw1.a`, and the `_priv.h`'s)*

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 space
- Multiple commands for same target are executed *in order*
 - Can split commands over multiple lines by ending lines with ‘\’

- ❖ Example:

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

- ❖ Demo: look at Makefile for our example program

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

- ❖ Example:

```
CC = gcc
CFLAGS = -Wall -std=c17
foo.o: foo.c foo.h bar.h
        $(CC) $(CFLAGS) -o foo.o -c foo.c
```

- ❖ Advantages:

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

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 *~
```

"Phony" Targets

- ❖ 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
      gcc -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
          gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

Revenge of the Funny Characters

- ❖ Special variables:
 - `$$` for target name
 - `$$^` for all sources
 - `$$<` for left-most source
 - Lots more! – see the documentation

- ❖ Examples:

```
# CC and CFLAGS defined above  
widget: foo.o bar.o  
          $(CC) $(CFLAGS) -o $$ $^  
foo.o: foo.c foo.h bar.h  
        $(CC) $(CFLAGS) -c $$<
```

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
```
- ❖ 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