

pollev.com/cse333

Name a *value* that you feel is embedded in the C language.

(open-ended survey question)

By “value” we mean an adjective describing the relative worth, merit, or importance of something (*e.g.*, loyalty, kindness), NOT a number or constant.

Systems Programming

Makefiles, C++ Preview

Instructor: Justin Hsia

Teaching Assistants:

Aadi Adgaonkar

Advaith Vijayakumar

Alice Fu

Andrew Vlahovich

Aviel Wood

Camden Harris

Esha Gore

Hannah Hempstead

Helena Wang

Laura Wang

Megan Frisella

Nina Tharamal

Shrihun Sankepally

Shriya Sateesh

Yusong Yan

Relevant Course Information

- ❖ Exercise 4 posted Wednesday, due next Wednesday
- ❖ Today is the last chance to do your Meet the Staff check-in
- ❖ Homework 1 due last night (10/15)
 - Check for HW upload error email – time to fix during late window
 - Late days: can still tag a commit made until the end of Sunday
- ❖ Homework 2 is released today
 - Builds on top of Homework 1 data structures to create a file search engine!
 - “Check-In 1” activity

Lecture Outline (1/4)

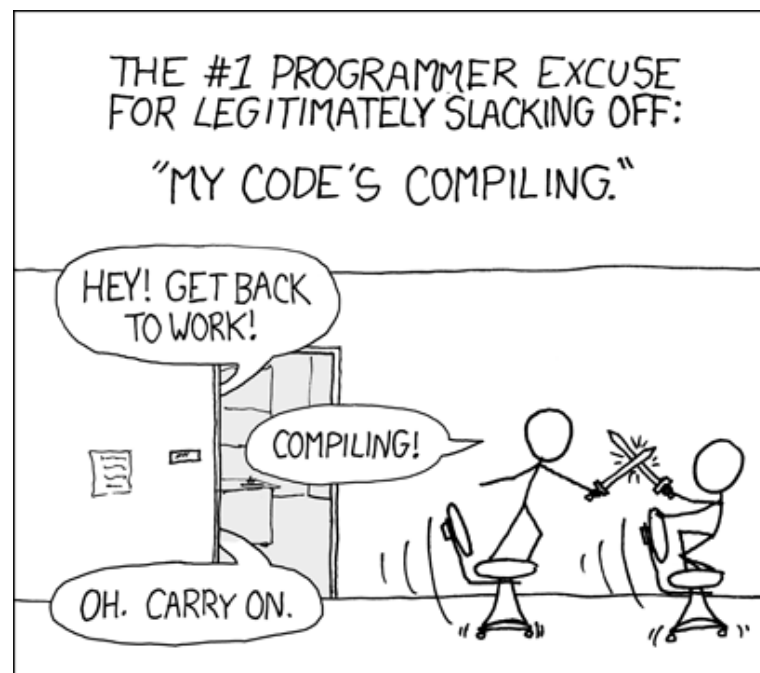
- ❖ **Make and Build Tools**
- ❖ Makefile Basics
- ❖ C History
- ❖ C++ Preview

make

- ❖ make was the first coordinated software **build system** – a program for controlling what gets (re)compiled and how
 - Many other such programs now exist (*e.g.*, ant, maven, IDE “projects”)
 - But make is the original and still in heavy use today
- ❖ make has tons of fancy features, but only two basic ideas:
 - 1) Be a **centralized space** that holds the commands used to compile our project
 - 2) Express **dependencies** for those commands so they only run when the source code they process changes (avoid unnecessary work)
- ❖ To avoid “just teaching make features” (boring and narrow), let’s focus more on the concepts...

Building Software (1/2)

- ❖ 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 (2/2)

❖ 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

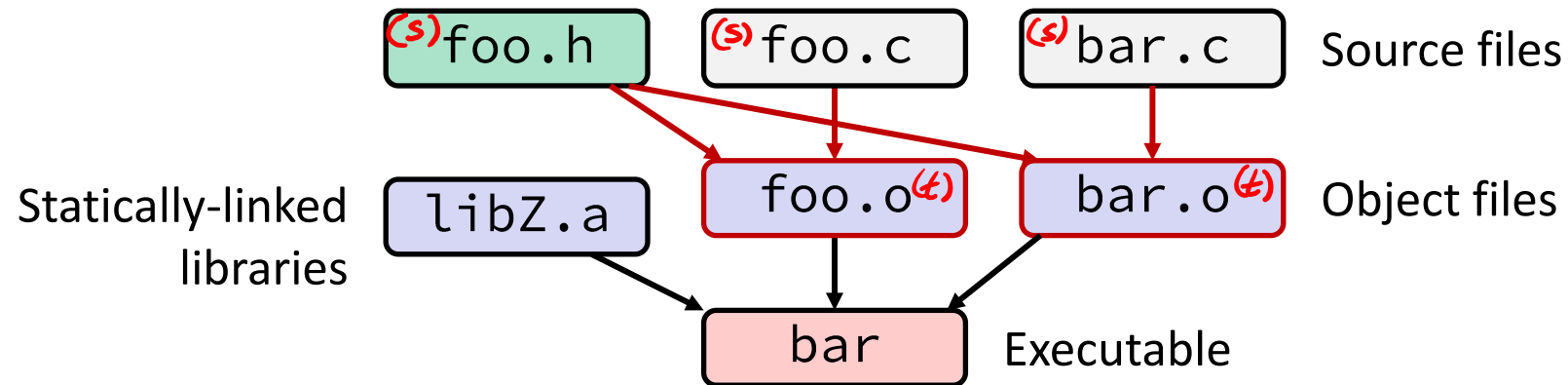
- ❖ On larger projects, you can't or don't want to have one big (set of) command(s) that are all run every time you change anything. To do things “smarter,” consider:
 - 1) It could be worse: If gcc didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
 - 2) Source files could have multiple outputs (*e.g.*, javadoc). You may 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; make it relatively simple for others to build
 - ★ 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 traditional shell script can handle 1-3, but 4 is trickier

Recompilation Management

- ❖ The “theory” behind avoiding unnecessary compilation is a *dependency dag* (**d**irected, **a**cyclic **g**raph)
- ❖ To create a target ^① t , you need sources ^② $s_1, s_2, \dots, s_n$ and a command ^③ 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 C (1/4)

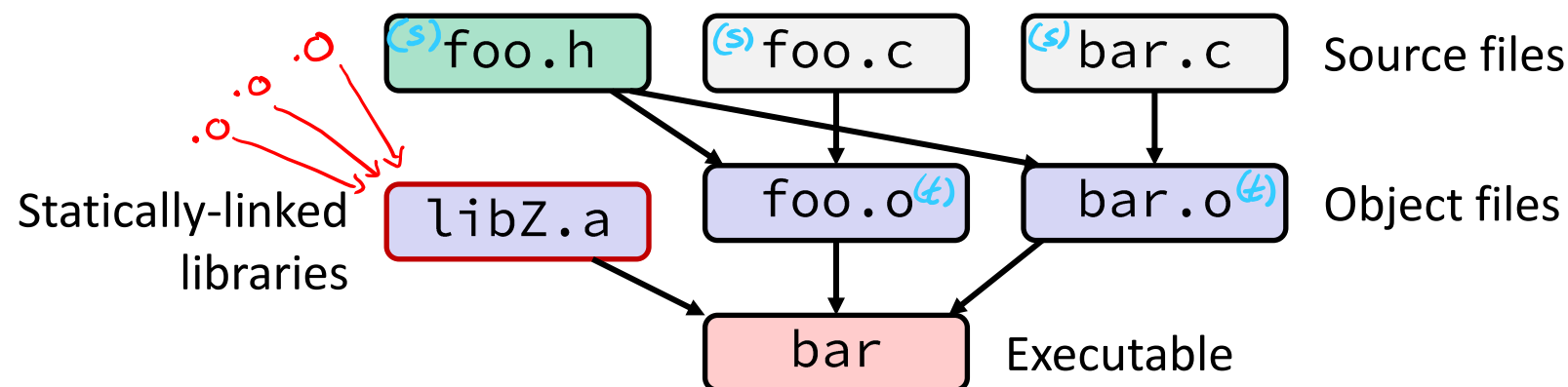
(s) = source
(t) = target



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

Theory Applied to C (2/4)

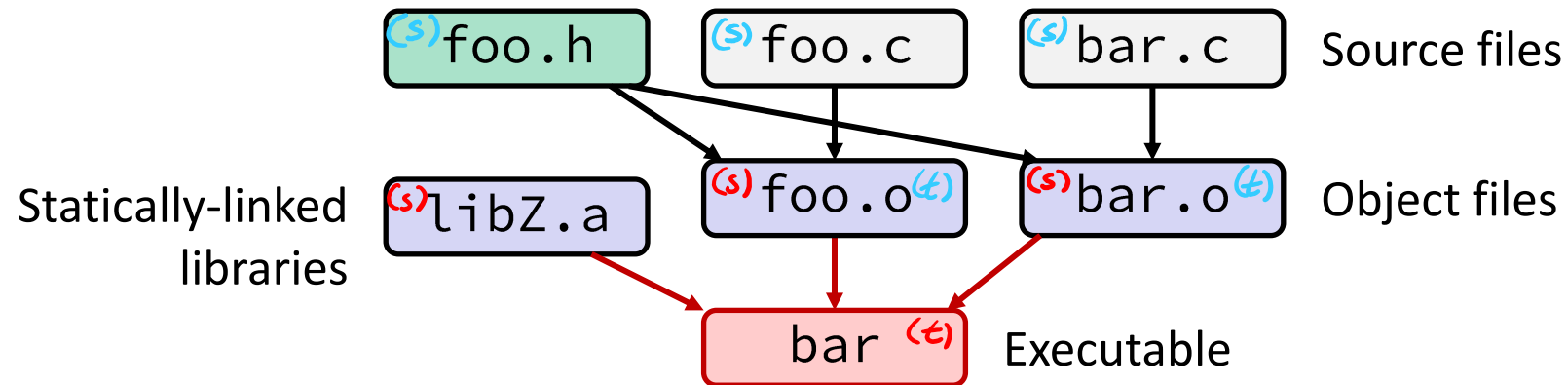
(s) = source
(t) = target



- ❖ 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

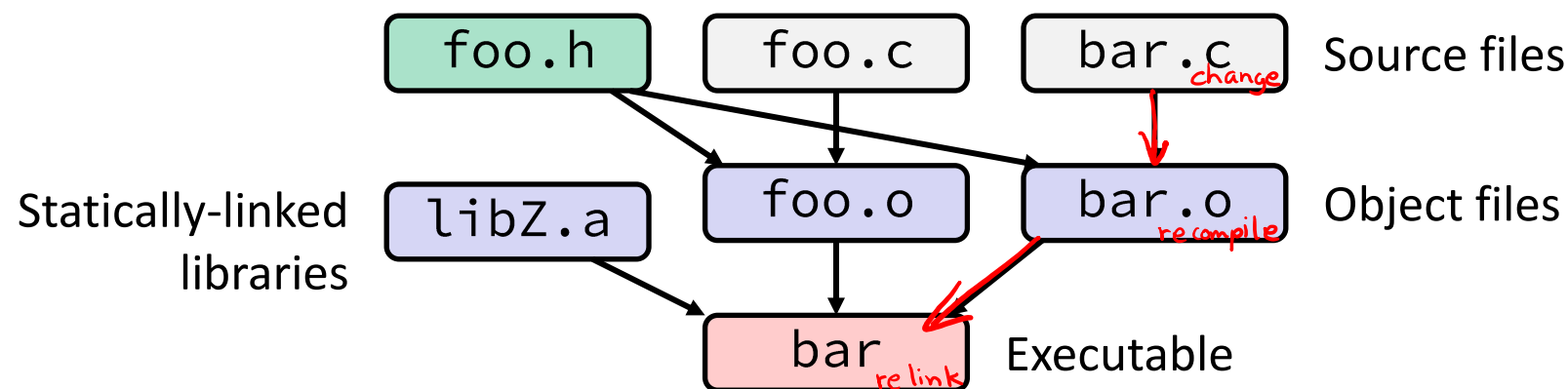
Theory Applied to C (3/4)

(s) = source
(t) = target



- ❖ 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)

Theory Applied to C (4/4)



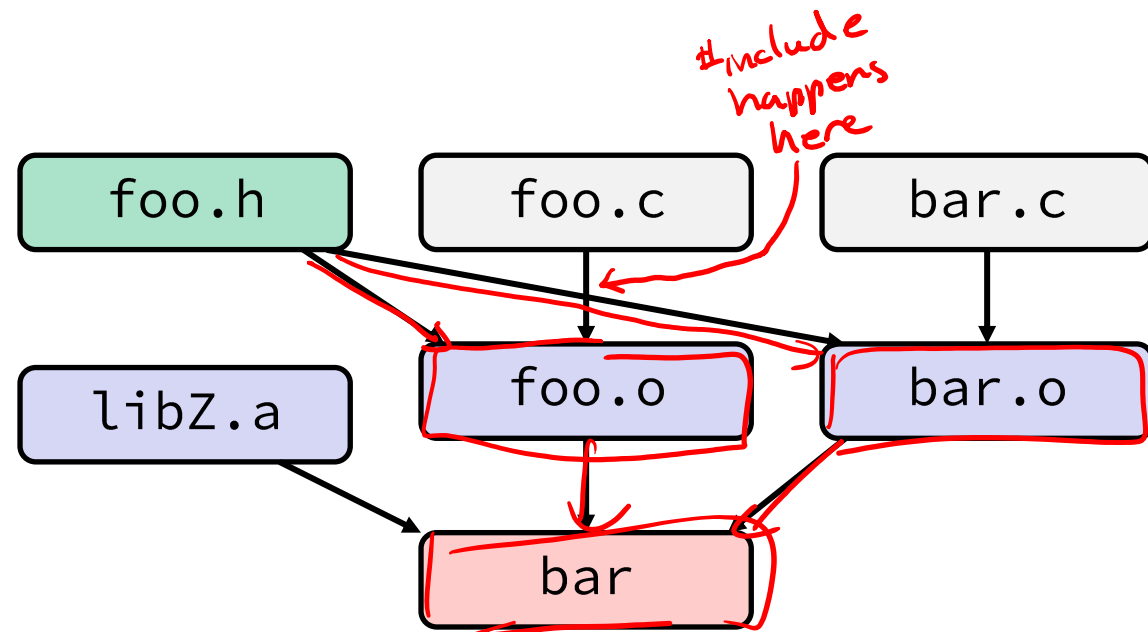
❖ Effects of code changes:

- 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!

pollev.com/cse333

If `foo.h` is updated, which files get re-built?

- A. `foo.o, bar.o`
- B. `foo.c, foo.o`
- C. `foo.o, bar.o, bar`
- D. `foo.c, foo.o, bar`
- E. None of them
- F. We're lost...



Lecture Outline (2/4)

- ❖ Make and Build Tools
- ❖ **Makefile Basics**
- ❖ C History
- ❖ C++ Preview

make Basics

❖ A makefile contains a bunch of **triples**:

① **target:** *sources* ②
← **Tab** → *command* ③

- 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 ‘\’

❖ Example: **foo.o:** *foo.c foo.h bar.h*
gcc -Wall -o foo.o -c foo.c

Using make

❖ **\$ make -f <makefileName> target**

- Defaults: *\$ make*
 - If no `-f` specified, use the file `Makefile` in current directory
 - If no `target` specified, will use the first one in the file
 - Will interpret commands in your default shell unless you set `SHELL` variable in makefile
- Target execution performed by checking 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)

“Phony” Targets

- ❖ A make target whose command does not create a file with the target’s name (*i.e.*, a “recipe”)
 - As long as the target file doesn’t exist, the command(s) will be executed because the target must be “remade”
 - Example: Target `clean` is a convention to remove generated files to “start over” from just the source:

```
clean:  
    rm foo.o bar.o baz.o widget *~
```
 - Example: Target `all` is a convention to build all “final products” in the makefile by listing all of the “final products” as sources

“all” Example (make or make all)

```
1 all: 5 prog B.class someLib.a 6  
2 # notice no commands this time  
3 prog: 4 foo.o bar.o main.o  
4 gcc -o prog foo.o bar.o main.o  
5 B.class: B.java  
6 javac B.java  
7 someLib.a: foo.o baz.o 8  
8 ar r foo.o baz.o  
9 foo.o: foo.c foo.h header1.h header2.h  
10 gcc -c -Wall foo.c  
11 # similar targets for bar.o, main.o, baz.o, etc...
```

The diagram illustrates the dependencies for the 'all' target. Red arrows show the following relationships:

- Target **all** (1) depends on **prog** (5), **B.class** (5), and **someLib.a** (6).
- Target **prog** (2) depends on **foo.o** (4), **bar.o** (4), and **main.o** (4).
- Target **B.class** (5) depends on **B.java** (6).
- Target **someLib.a** (7) depends on **foo.o** (8) and **baz.o** (8).
- Target **foo.o** (9) depends on **foo.c** (10), **foo.h** (10), **header1.h** (10), and **header2.h** (10).

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
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
        $(CC) $(CFLAGS) -o widget $(OBJFILES)
```

❖ Advantages:

- Easy to change things (especially in multiple commands), *e.g.*, lists of filenames
- Can also specify/overwrite variables on the command line, *e.g.*,

```
$ make CC=clang CFLAGS=-g
```



Makefile Writing Tips

❖ *When creating a Makefile, first draw the dependencies!!!!*

❖ C Dependency Rules:

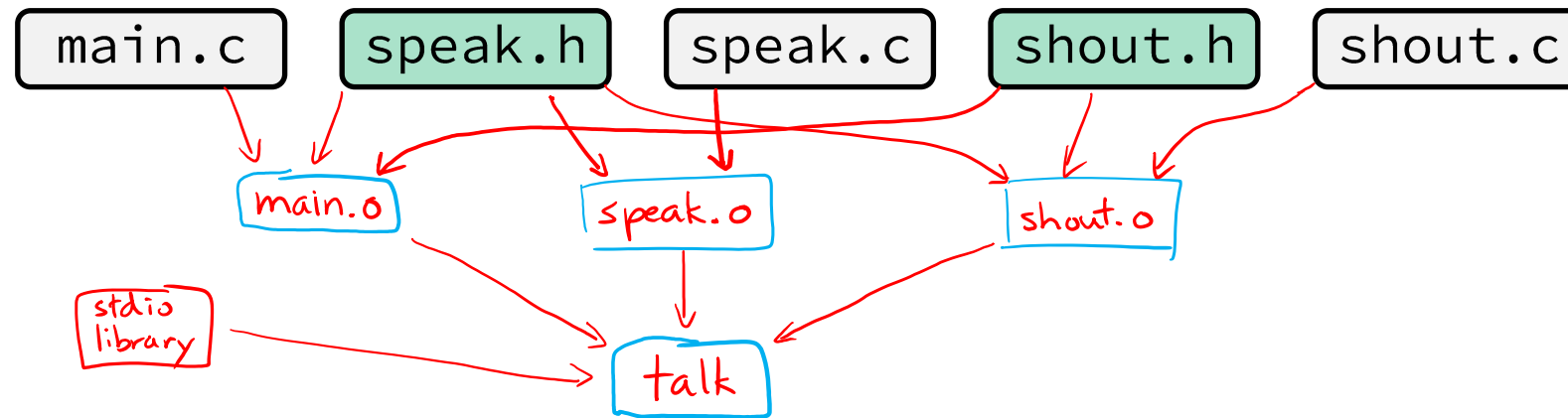
- .c and .h files are never targets, only sources
- Each .c file will be compiled into a corresponding .o file
 - Header files will be implicitly used via `#include`
- Executables will typically be built from one or more .o file

❖ Good Conventions:

- Include a `clean` rule
- If you have more than one “final target,” include an `all` rule
- The first/top target should be your singular “final target” or `all`

Writing a Makefile Example: DAG

- ❖ “talk” program (find files on website schedule with lecture slides)



main.c

```
#include "speak.h"
#include "shout.h"

int main(int argc, char* argv[]) {...
```

speak.c

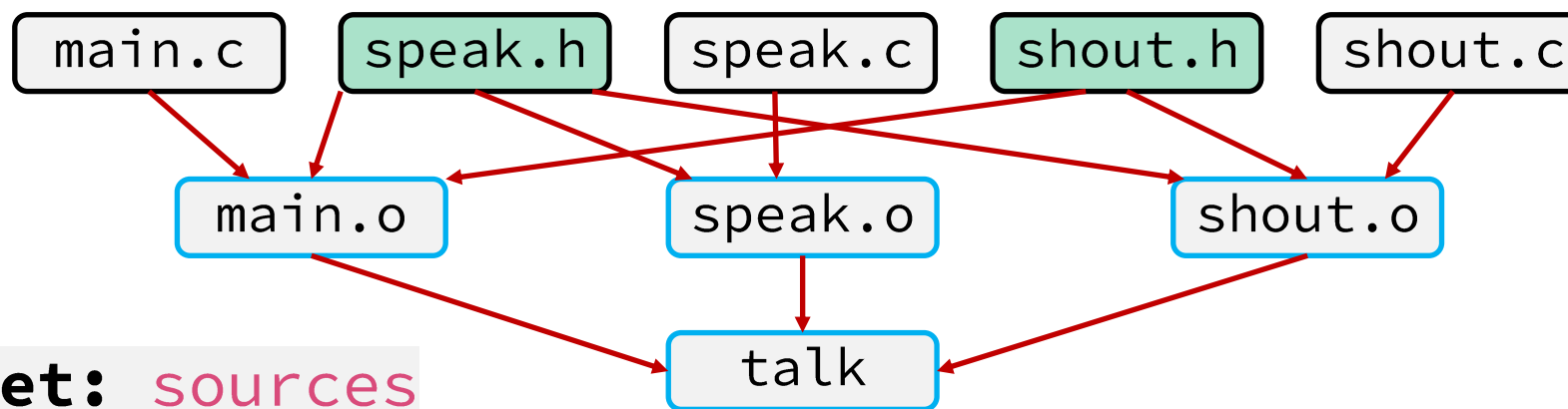
```
#include "speak.h"
...
```

shout.c

```
#include "speak.h"
#include "shout.h"
...
```

Writing a Makefile Example: Makefile

- ❖ “talk” program (find files on website schedule with lecture slides)



- **target:** sources
command

```
talk: main.o speak.o shout.o
gcc $(CFLAGS) -o talk main.o speak.o shout.o
```

```
main.o: main.c speak.h shout.h
gcc $(CFLAGS) -c main.c
```

```
speak.o: speak.c speak.h
gcc $(CFLAGS) -c speak.c
```

```
shout.o: shout.c speak.h shout.h
gcc $(CFLAGS) -c shout.c
```

```
clean:
rm talk *.o
```

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...)

Lecture Outline (3/4)

- ❖ Make and Build Tools
- ❖ Makefile Basics
- ❖ **C History**
- ❖ C++ Preview



Name a value that you feel is embedded in the C language.

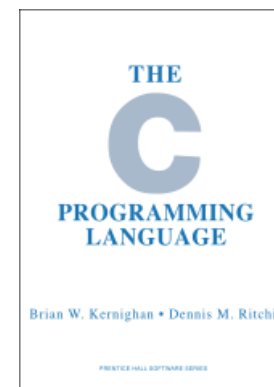


Nobody has responded yet.

Hang tight! Responses are coming in.

Development of the C Language (1/3)

- ❖ Created in 1972 as a system programming language for Unix
 - Lineage: BCPL → B → C
 - Unix was rewritten entirely in C (Version 4 in 1973)
- ❖ “Standardized” in 1978 with release of K&R Ed. 1:
 - From initial creation, developed in terms of portability and type safety
- ❖ Formal standardization via American National Standards Institute (ANSI) in 1989 and International Organization for Standardization (ISO) in 1990
 - Non-portable portion of the Unix C library was the basis for the POSIX standard via IEEE



Development of the C Language (2/3)

❖ Development Context:

- Developed for the PDP-7/PDP-11
 - Very limited memory available for program
- Improvements over B: data typing, performance, byte addressability
 - B was a “typeless language” (computer word size)
- Developed in the context of operating system innovations (Multics, Unix)
 - “Particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers.”
 - “By design, C provides constructs that map efficiently to typical machine instructions. It has found lasting use in applications previously coded in assembly language.”

❖ Who used computers and programming at the time?

Development of the C Language (3/3)

❖ Credits:

- **Dennis Ritchie** designed C
- **Ken Thompson** designed B and, with Ritchie, were the primary architects of UNIX (in assembly)
- **Brian Kernighan** helped Ritchie write K&R, the first “standardization” of the C language



Ken
Thompson

Dennis
Ritchie

Brian
Kernighan

- ❖ Source: “The development of the C language”
(<https://dl.acm.org/doi/10.1145/155360.155580>)

Principles of C

- ❖ Some commonly-held contemporary views:
 - “Since C is relatively small, it can be described in small space and learned quickly.”
 - “Shows what’s really happening.”
 - “Close to the machine/hardware.”
 - “Only the bare essentials.”
 - “No one to help you.”
 - “You’re on your own.”
 - “I know what I’m doing, get out of my way.”

Principles of C – Embedded Values?

❖ Some commonly-held contemporary views:

- “Since C is relatively small, it can be described in small space and learned quickly.”
- “Shows what’s really happening.”
- “Close to the machine/hardware.”
- “Only the bare essentials.”
- “No one to help you.”
- “You’re on your own.”
- “I know what I’m doing, get out of my way.”

} **Rugged**

} **Minimalistic**

} **Individualistic**

Lecture Outline (4/4)

- ❖ Make and Build Tools
- ❖ Makefile Basics
- ❖ C History
- ❖ **C++ Preview**

Encapsulation and Abstraction (C++)

❖ Support for classes and objects!

- Public, private, and protected access specifiers
- **Methods** and **instance variables** ("this")
- (Multiple!) inheritance

❖ Polymorphism

- *Static polymorphism*: multiple functions or methods with the same name, but different argument types (overloading)
 - Works for all functions, not just class members
- *Dynamic (subtype) polymorphism*: derived classes can override methods of parents, and methods will be dispatched correctly

Namespaces (C++)

❖ Explicit namespaces!

- The linked list module could define an “LL” namespace while the hash table module could define an “HT” namespace
- Both modules could define an Iterator class
 - One would be globally named `LL::Iterator` and the other would be globally named `HT::Iterator`

❖ Classes also allow duplicate names without collisions

- Classes can also define their own pseudo-namespace, very similar to Java static inner classes

No longer have to rely so much on coding conventions in C.

Generics (C++)

❖ **Templates** facilitate generic data types

- *Parametric polymorphism*: same idea as Java generics, but different in details, particularly implementation
 - A vector of `ints`: `vector<int> x;`
 - A vector of `floats`: `vector<float> x;`
 - A vector of (vectors of `floats`): `vector<vector<float>> x;`

❖ Specialized casts to increase type safety

Can avoid the messiness of using `void*` and function pointers in C.

Standard Library (C++)

- ❖ **Generic containers:** bitset, queue, list, associative array (including hash table), deque, set, stack, and vector
 - And iterators for most of these
- ❖ **A `string` class:** hides the implementation of strings
- ❖ **Streams:** allows you to stream data to and from objects, consoles, files, strings, and so on
- ❖ **Generic algorithms:** sort, filter, remove duplicates, etc.

Don't have to implement everything yourself!

Error Handling (C++)

❖ Supports exceptions!

- try / throw / catch
- If used with discipline, can simplify error processing
- If used carelessly, can complicate memory management
 - Consider: `a()` calls `b()`, which calls `c()`
 - If `c()` throws an exception that `b()` doesn't catch, you might not get a chance to clean up resources allocated inside `b()`
- Unfortunately, we will avoid exceptions in 333
 - You still benefit from having more interpretable errors!
 - But much C++ code still needs to work with C & old C++ libraries, so still uses return codes, `exit()`, etc.

Error handling is still a pain, unfortunately. 😞

Some Tasks Still Hurt in C++ (1/2)

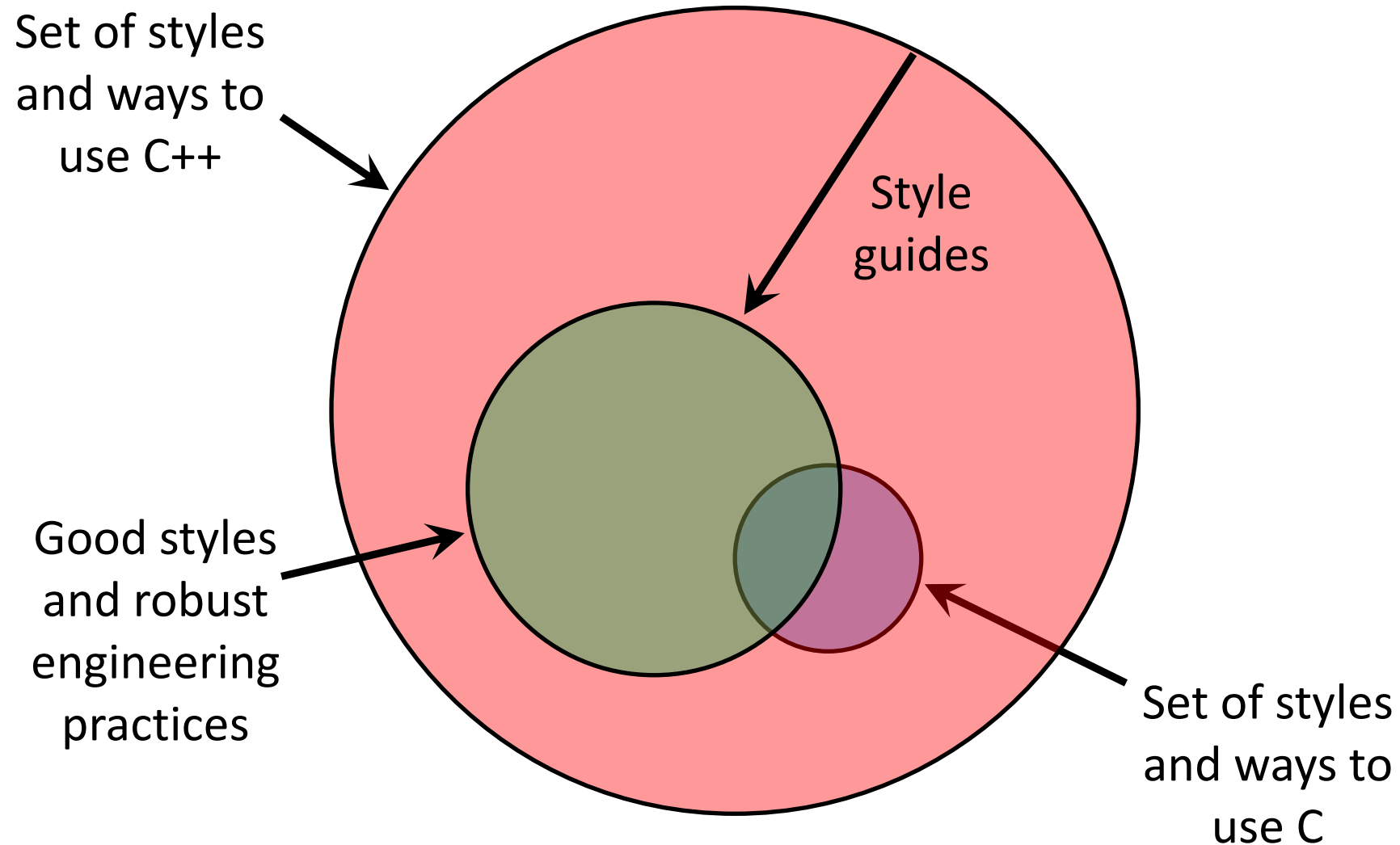
❖ Memory management

- C++ has no garbage collector
 - You still have to manage memory allocation & deallocation and track
 - It's still possible to have leaks, double frees, and so on
- But there are some things that help
 - “Smart pointers”
 - Classes that encapsulate pointers and track reference counts
 - Deallocate memory when the reference count goes to zero
 - C++'s constructors and destructors permit a pattern known as “Resource Allocation Is Initialization” (RAII)
 - Useful for releasing memory, locks, database transactions, etc.

Some Tasks Still Hurt in C++ (2/2)

- ❖ C++ doesn't guarantee type or memory safety
 - You can still:
 - Forcibly cast pointers between incompatible types
 - Walk off the end of an array and smash memory
 - Have dangling pointers
 - Conjure up a pointer to an arbitrary address of your choosing

How to Think About C++



Or...

In the hands of a
disciplined programmer,
C++ is a powerful tool



But if you're not so
disciplined about how you
use C++...

