Poll Everywhere pollev.com/cse333j

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, kindess), NOT a number or constant.

Systems Programming

Makefiles, C++ Preview

Instructors:

Justin Hsia Amber Hu

Teaching Assistants:

Ally Tribble	Blake Diaz	Connor Olson
Grace Zhou	Jackson Kent	Janani Raghavan
Jen Xu	Jessie Sun	Jonathan Nister
Mendel Carroll	Rose Maresh	Violet Monserate

Relevant Course Information

- No more leniency with assignment submission – messed up tag or file locations = no submission
- Exercise 7 posted Wednesday, due Monday
  - Read a directory and open/copy text files found there
- Homework 1 due last night (1/22)
  - 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
  - See Ed post #236 for partner sign-up & matching forms
  - Builds on top of Homework 1 data structures to create search engine!

Lecture Outline (1/4)

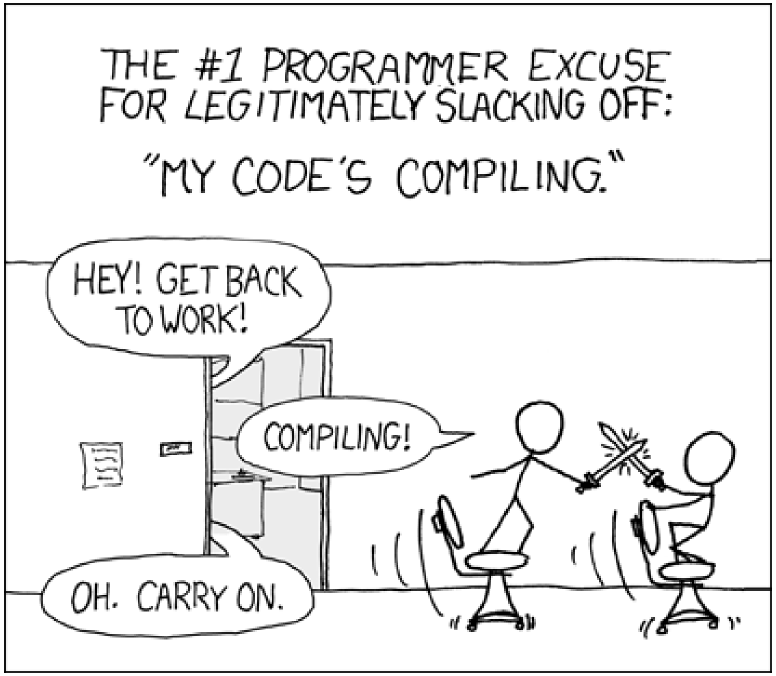
- **Make and Build Tools**
- Makefile Basics
- C History
- C++ Preview

`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 (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 script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

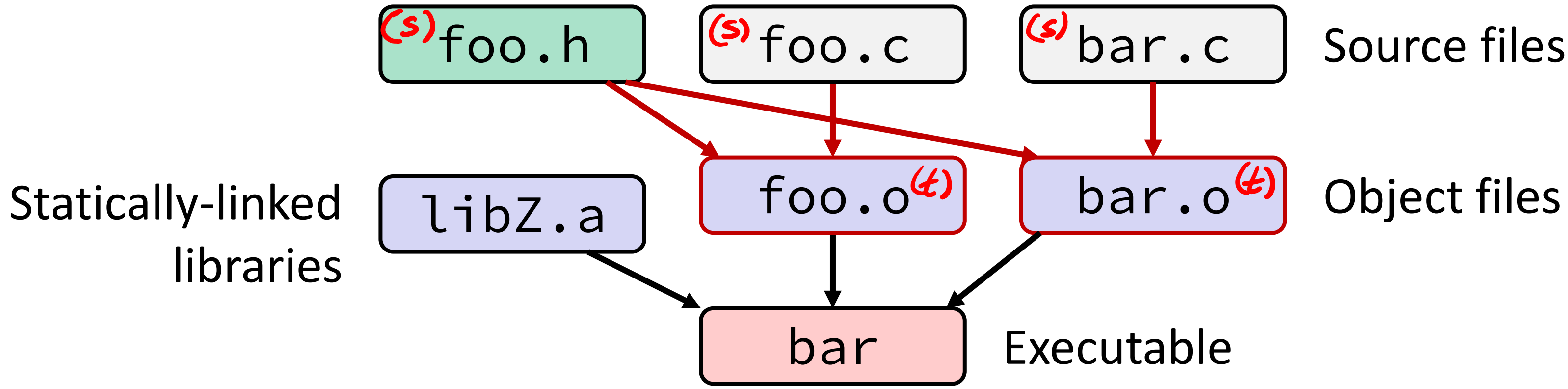
Recompilation Management

- The "theory" behind avoiding unnecessary compilation is a *dependency dag* (**d**irected, **a**cyclic **g**raph)

- To create a target ① t, you need sources ② $s_1, s_2, \ldots, s_n$ 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"!

(s) = source
(t) = target

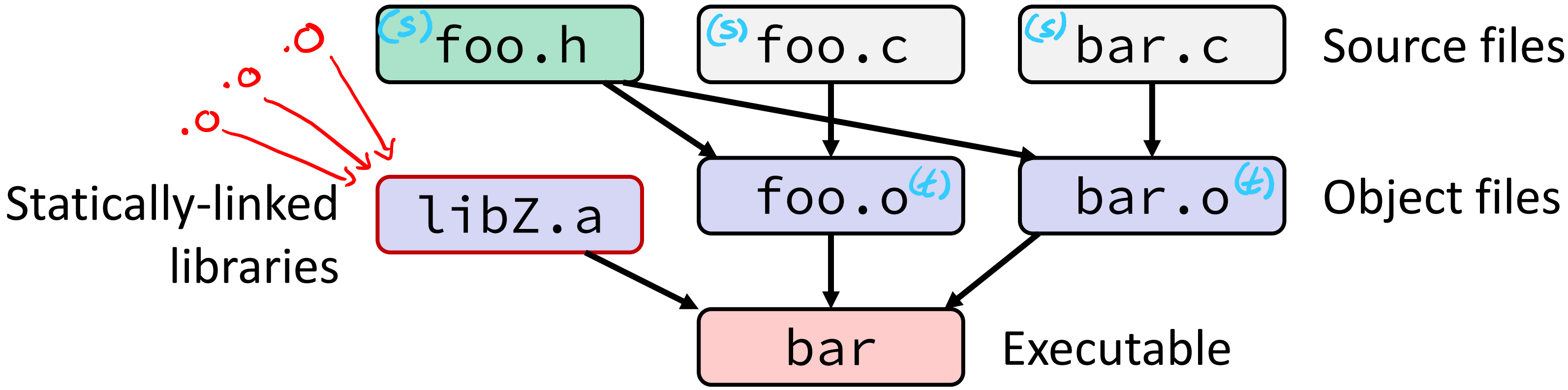
Theory Applied to C (1/4)



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

(s) = source
(t) = target

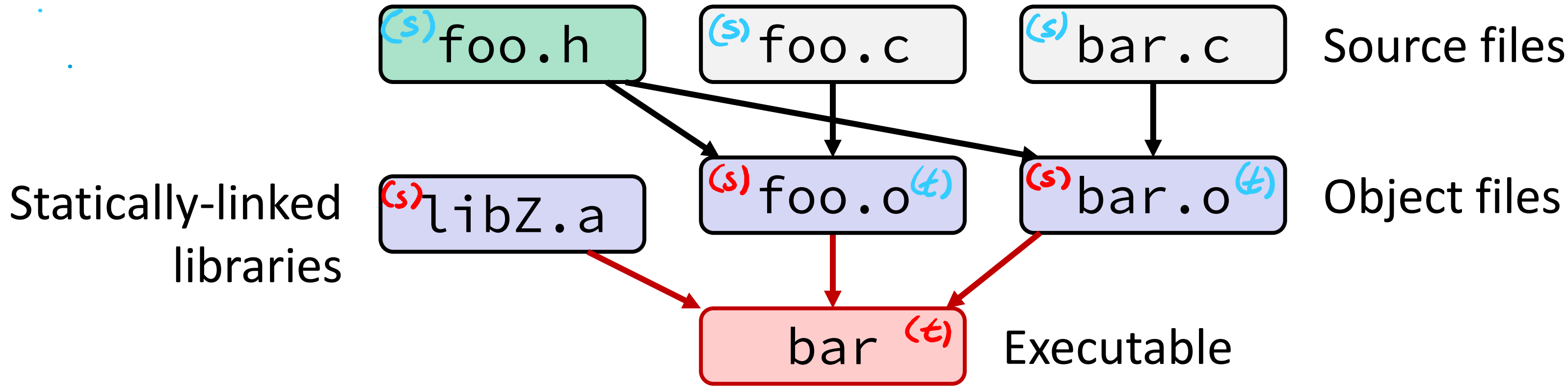
Theory Applied to C (2/4)



- 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

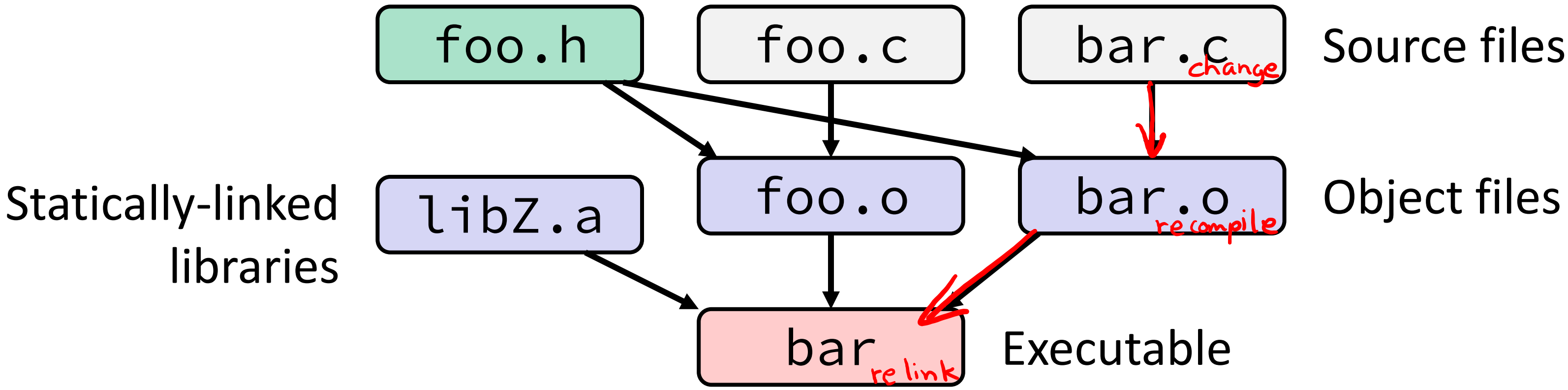
(s) = source
(t) = target

Theory Applied to C (3/4)



- 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!

Lecture Outline (2/4)

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

make Basics

- A makefile contains a bunch of triples:

```
①target: 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 a file named `Makefile` in current dir
  - 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)

"Phony" Targets

- A make target whose command does not create a file of the target's name (*i.e.*, a "recipe")
  - As long as target file doesn't exist, the command(s) will be executed because the target must be "remade"

- *e.g.*, 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 *~
```

- *e.g.*, target `all` is a convention to build all "final products" in the makefile
  - Lists all of the "final products" as sources

"all" Example (make or make all)

```
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...
```

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)
    - It's common to use variables to hold lists of filenames
  - Can also specify/overwrite variables on the command line: (*e.g.*, `make CC=clang CFLAGS=-g`)

Makefile Writing Tips

STYLE TIP

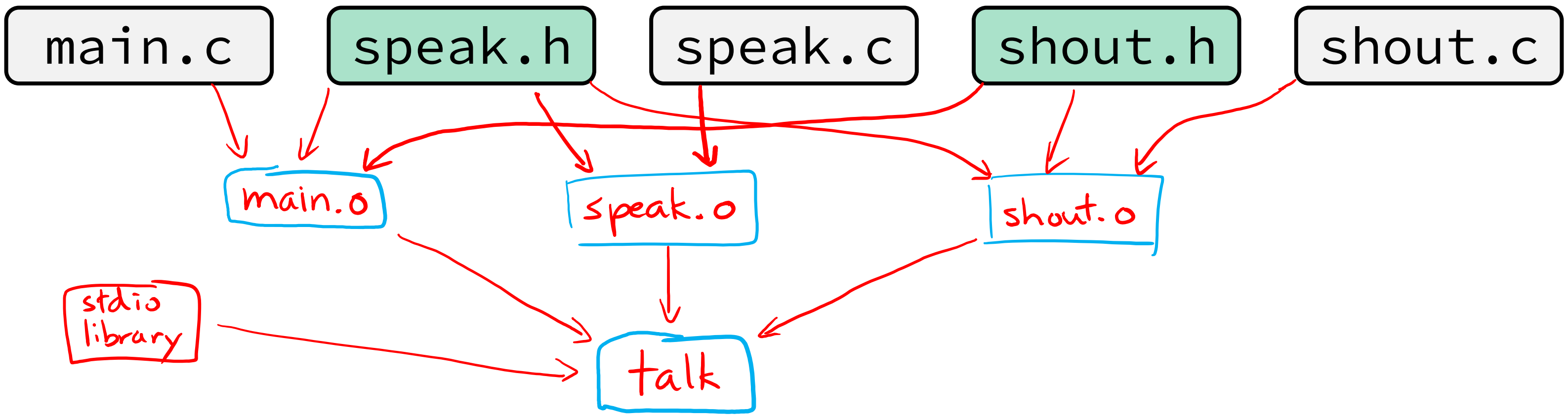
- *When creating a Makefile, first draw the dependencies!!!!*

- 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 web with lecture slides)



main.c

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

int main(int argc, char** argv) {…
```

speak.c

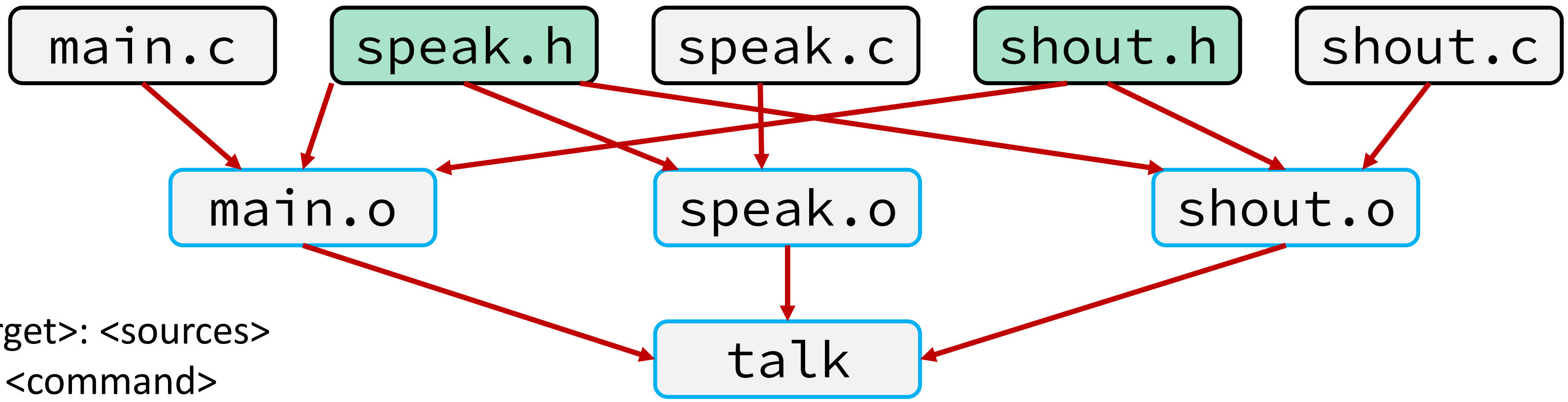
```
#include "speak.h"
...
```

shout.c

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

Writing a Makefile Example: Makefile

- "`talk`" program (find files on web 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.

0

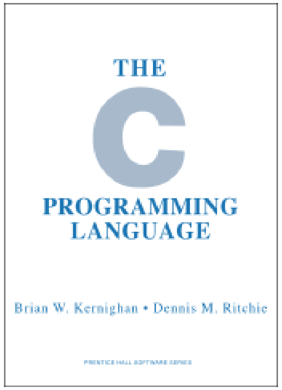
Nobody has responded yet.

Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Development of the C Language (1/3)

- Created in 1972
  - BCPL → B → C
  - Designed specifically as a system programming language for Unix
    - 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



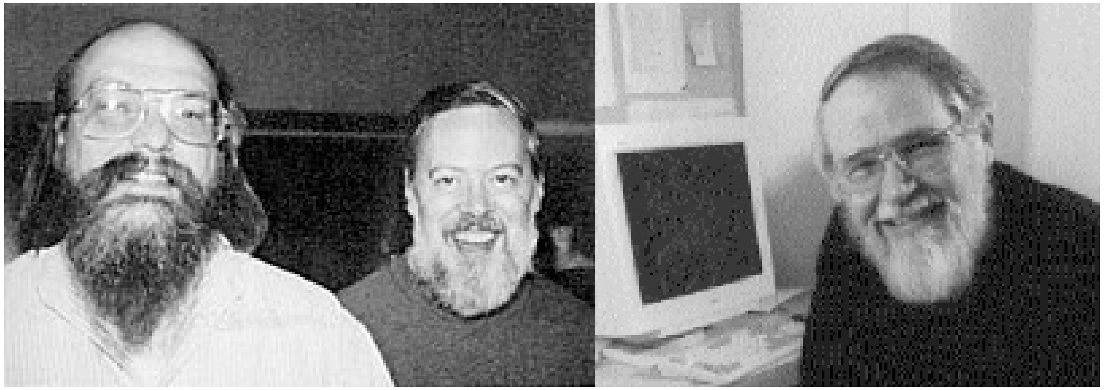
- Formal standardization via American National Standards Institute (ANSI) in 1989 and International Organziation for Standardization (ISO) in 1990
  - Non-portable portion of the Unix C library was the basis for the POSIX standard via IEEE

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 addressibility
  - 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
- "The development of the C language" (https://dl.acm.org/doi/10.1145/155360.155580)



Ken Thompson | Dennis Ritchie | Brian Kernighan

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."

Minimalistic (first four items)

Rugged (items 4–6)

Individualistic (items 6–7)

Lecture Outline (4/4)

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

Programming Terminology Review

- **Encapsulation and Abstraction:** Hiding implementation details (restricting access) and associating behaviors (methods) with data

- **Polymorphism:** The provision of a single interface to entities of different types

- **Generics:** Algorithms written in terms of types *to-be-specified-later*

Encapsulation and Abstraction (C)

- Used header file conventions and the `static` specifier to separate "private" functions, definitions, and constants from "public"

- Used forward-declared `structs` and opaque pointers (*i.e.*, `void*`) to hide implementation-specific details

- Can't associate behaviors with encapsulated state
  - Functions that operate on a `LinkedList` not actually tied to the struct

Really difficult to mimic – implemented primarily via coding conventions

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

Generics (C)

- Generic linked list and hash table by using `void*` payload
- Function pointers to generalize different behavior for data structures
  - Comparisons, deallocation, pickling up state, etc.

Emulated generic data structures primarily by disabling type system

Generics (C++)

- Templates facilitate generic data types
  - *Parametric polymorphism:* same idea as Java generics, but different in details, particularly implementation
    - A vector of `int`s: `vector<int> x;`
    - A vector of `float`s: `vector<float> x;`
    - A vector of (vectors of `float`s): `vector<vector<float>> x;`
- Specialized casts to increase type safety

Namespaces (C)

- Names are global and visible everywhere
  - Can use `static` to prevent a name from being visible outside a source file (as close as C gets to "private")

- Naming conventions help avoid collisions in the global namespace
  - *e.g.*, `LinkedList_Allocate`, `HTIterator_Next`, etc.

Avoid collisions primarily via coding conventions

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

Standard Library (C)

- C does not provide any standard data structures
  - We had to implement our own linked list and hash table
- Hopefully, you can use somebody else's libraries
  - But C's lack of abstraction, encapsulation, and generics means you'll probably end up tweak them or tweak your code to use them

YOU implement the data structures that you need

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.

Error Handling (C)

- Error handling is a pain
- Define error codes and return them
  - Either directly return or via a "global" like `errno`
  - No type checking: does `1` mean `EXIT_FAILURE` or `true`?

- Customers and implementors need to constantly test return values
  - *e.g.*, if `a()` calls `b()`, which calls `c()`
    - `a` depends on `b` to propagate an error in `c` back to it

Error handling is a pain – mixture of coding conventions and discipline

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()`

- We will largely avoid 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.

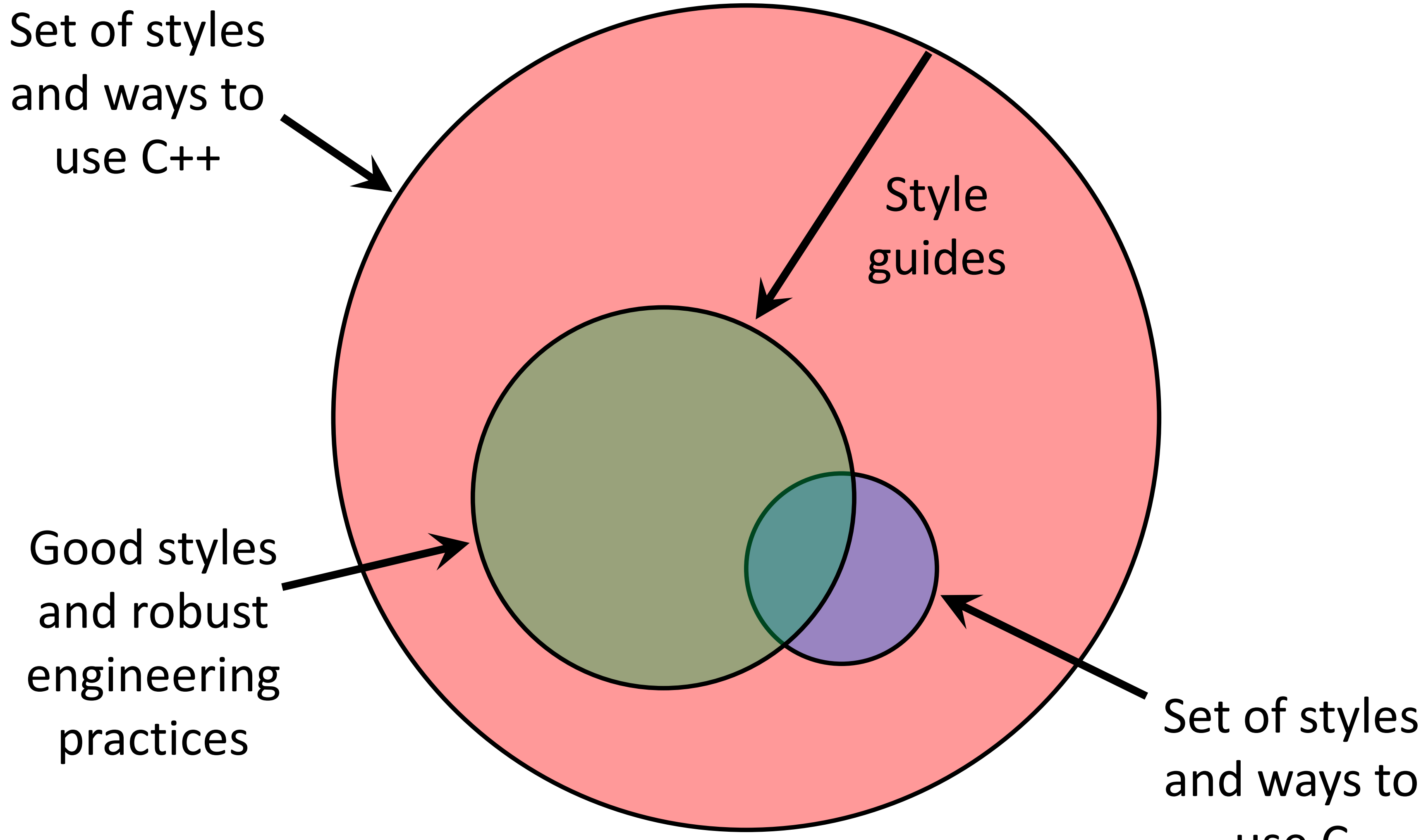
Some Tasks Still Hurt in C++ (1/2)

- 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)

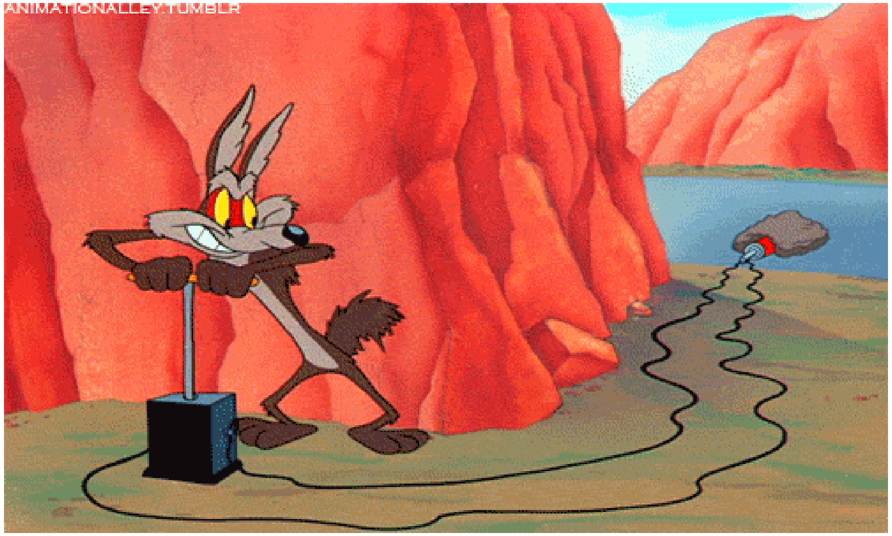
- C++ doesn't guarantee type or memory safety
  - You can still:
    - Forcibly cast pointers between incompatible types
    - Walk off the end of an array and smash memory
    - Have dangling pointers
    - Conjure up a pointer to an arbitrary address of your choosing

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++…