CSE333 Systems Programming

Intro, Getting Started in C

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Introductions: Course Instructors

- Professor Naomi :)
  - Electrical engineer by training
  - Bopped around Silicon Valley hacking on everything from OS kernels to internet backbone routers to LIDAR firmware to mobile graphics libraries
  - Discovered in industry that computers are boring
    - But *people* on the other hand…
  - Proud cat mom to Danni (aka Her Royal Majesty, Queen Baby)

Course Staff: Teaching Assistants

- Learn more about us on the course website!

- More than anything, we want you to feel…
  - ✓ Comfortable and welcome in this space
  - ✓ Able to learn and succeed in this course
  - ✓ Comfortable reaching out if you need help or want change

Introductions: Students

- ~200 students registered

- Expected background
  - **Prereq:** CSE 351 – C, pointers, memory model, linker, system calls
  - **Indirect Prereq**: CSE 123 – Classes, Inheritance, Basic Data structures, and general good style practices
  - CSE 391 or Linux skills needed for CSE 351 assumed

- Get to know each other! Help each other out!
  - Working well with others is a valuable life skill

Lecture Outline (1/3)

- **Course Policies**
  - https://courses.cs.washington.edu/courses/cse333/26sp/syllabus.html
  - Summary here, but you ***must*** read the full details online
- Course Introduction
- Getting Started in C
  - What do you need to write a C program from scratch?

Staff-Student Communication

- **Website:** http://cs.uw.edu/333
  - Schedule, policies, materials, assignments, etc.
- **Discussion:** https://edstem.org/us/courses/97181/
  - Announcements made here
  - Ask and answer questions – staff will monitor and contribute
- **Office Hours:** In-person OH, Google Sheet queue (CSE login)
- **1-on-1 Meetings:** can request a limited number of appointments via Google Form (CSE login)
- **Anonymous feedback**

Office Hours

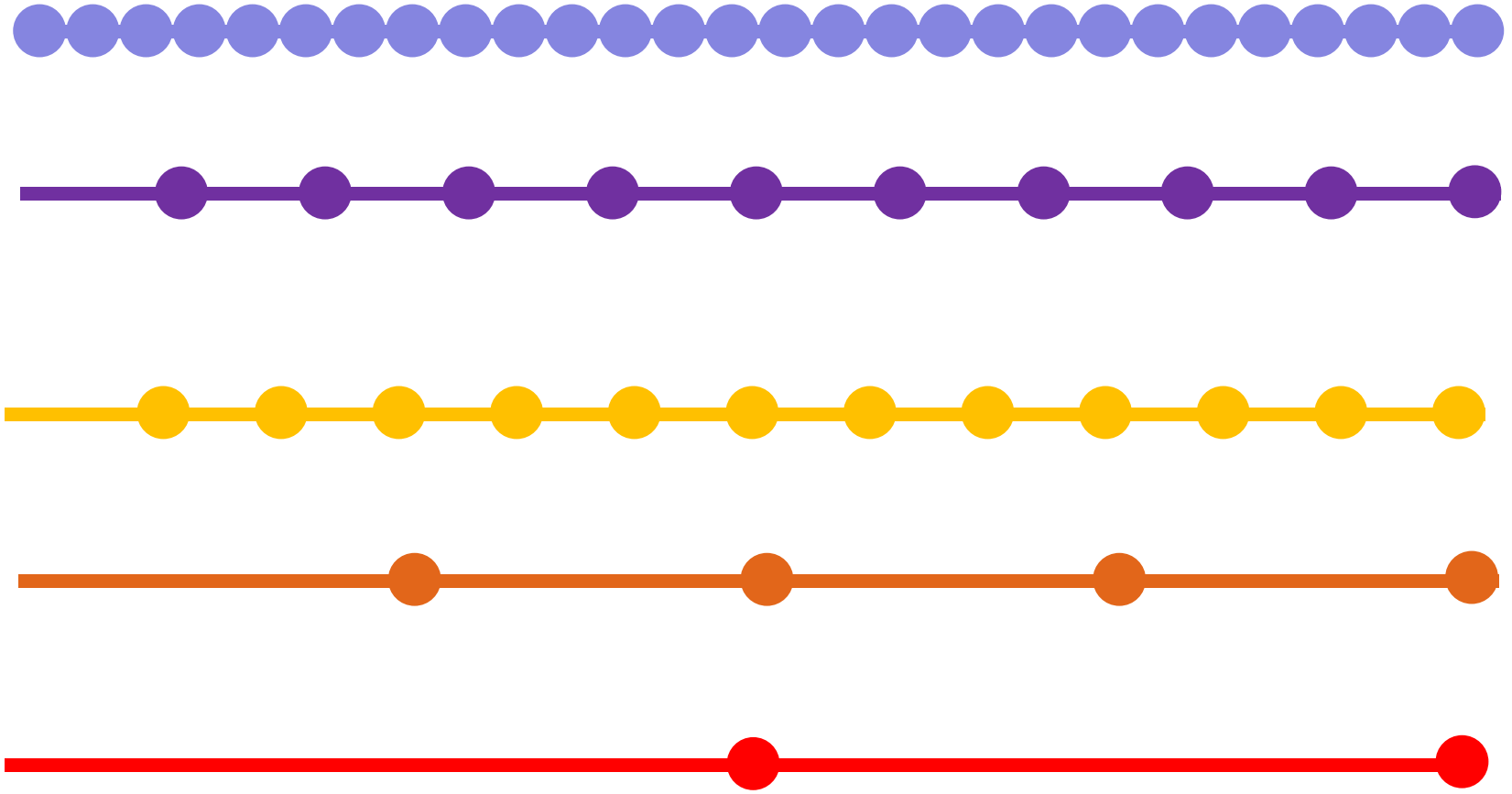
- Check Weekly Calendar for scheduled office hours:

Mon 3/30	Tue 3/31	Wed 4/1	Thu 4/2	Fri 4/3
9a - 10a Office Hours Irene CSE 153	11:30a - 12:30p Office Hours Angela Allen 4th Floor Breakout	11:30a - 12:20p Lecture A C: Memory, Data, Parameters CSE2 G20	Section C, Pointers, Gitlab	Ex-1 Due
11:30a - 12:20p Lecture A Introduction, Getting Started in C CSE2 G20	1p - 2p Office Hours Jonathan Allen 218	12:30p - 1:30p Office Hours Jen TBD	4:30p - 5:30p Office Hours Hannah and Advay TBD	Pre-Survey Due Link
12:30p - 1:30p Office Hours Ann CSE 121	3:30p - 4:30p Office Hours Mendel Allen 4th Floor Breakout	2p - 3p Office Hours Camden Allen 2nd Floor Breakout		10:30a - 11:30a Office Hours Barbora CSE 121
4p - 5p Office Hours Advay CSE 121		3p - 4p Office Hours Rishabh TBD		11:30a - 12:20p Lecture A C: Pointers CSE2 G20
				1:30p - 2:30p Office Hours Aviel Allen 2nd Floor Breakout

- We encourage you to chat with other students if the TAs are busy!
  - Keep it high level, no sharing code or answers
  - Trade debugging strategies, tips for using course tools

Course Components

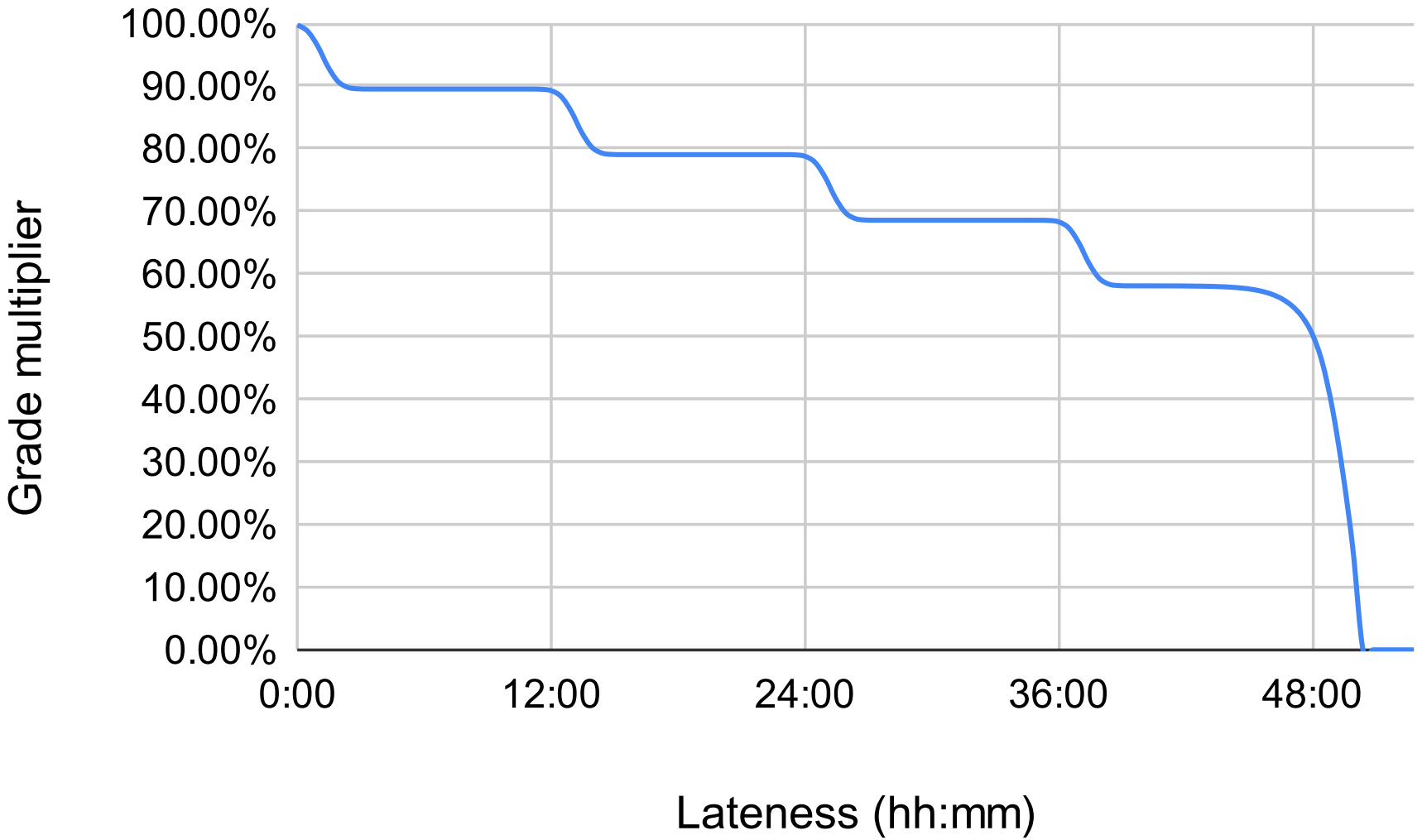
- Lectures (28)
- Sections (10)
- Exercises (12)
- Homeworks (4)
- In-Person Exams (2)
  - **Midterm:** Mon, May 4 in class (11:30a – 12:20p)
  - **Final:** Wed, June 10 (2:30a – 4:20p)



Grading

- **Exercises:** 30% total
  - Each graded on 8pt scale for correctness and style by autograders and TAs
  - Lowest two exercise scores dropped
  - No late work accepted (solutions are released immediately)
- **Homeworks:** 40% total
  - Binaries provided if you didn't get previous part working
  - Graded on test suite, manual tests, and style
  - Late work is penalized by a smooth decay curve (no "tokens")
- **Exams:** Midterm (14%) and Final (14%)
- **Effort, Participation, and Altruism:** 2%
  - Many ways to earn credit here, relatively lenient on this

The Soft Stair (to sadness)



Academic Integrity and Student Conduct

- We trust you implicitly and will follow up if that trust is violated
  - In short: don't attempt to gain credit for something you didn't do and don't help others do so, either
- If you find yourself in a situation where you are tempted to perform academic misconduct, please reach out to the instructors to explain your situation instead
  - See the Extenuating Circumstances section of the syllabus

And off we go…

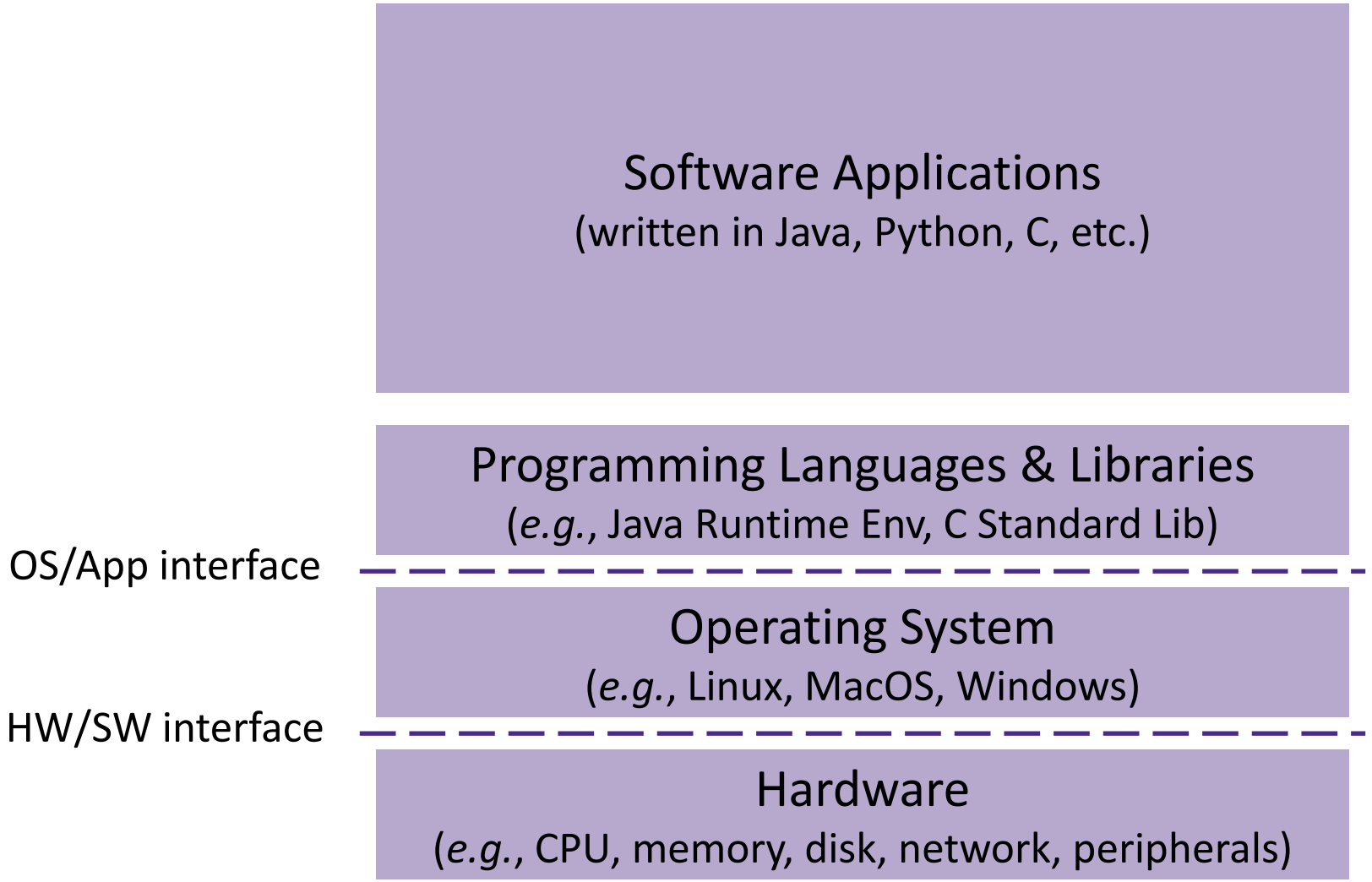
- This week: Goal is to figure out setup and course infrastructure right away

- So:
  - First exercise is out today, due Friday morning **11 am** before class
  - We'll create repos on GitLab for you. Do your work in these repos and "submit" your work by tagging a commit.
  - For help doing this:
    - Written guide here
    - Go to section on Thursday!! We will walk you through exactly how to do it
  - Poll Everywhere polls starting next lecture

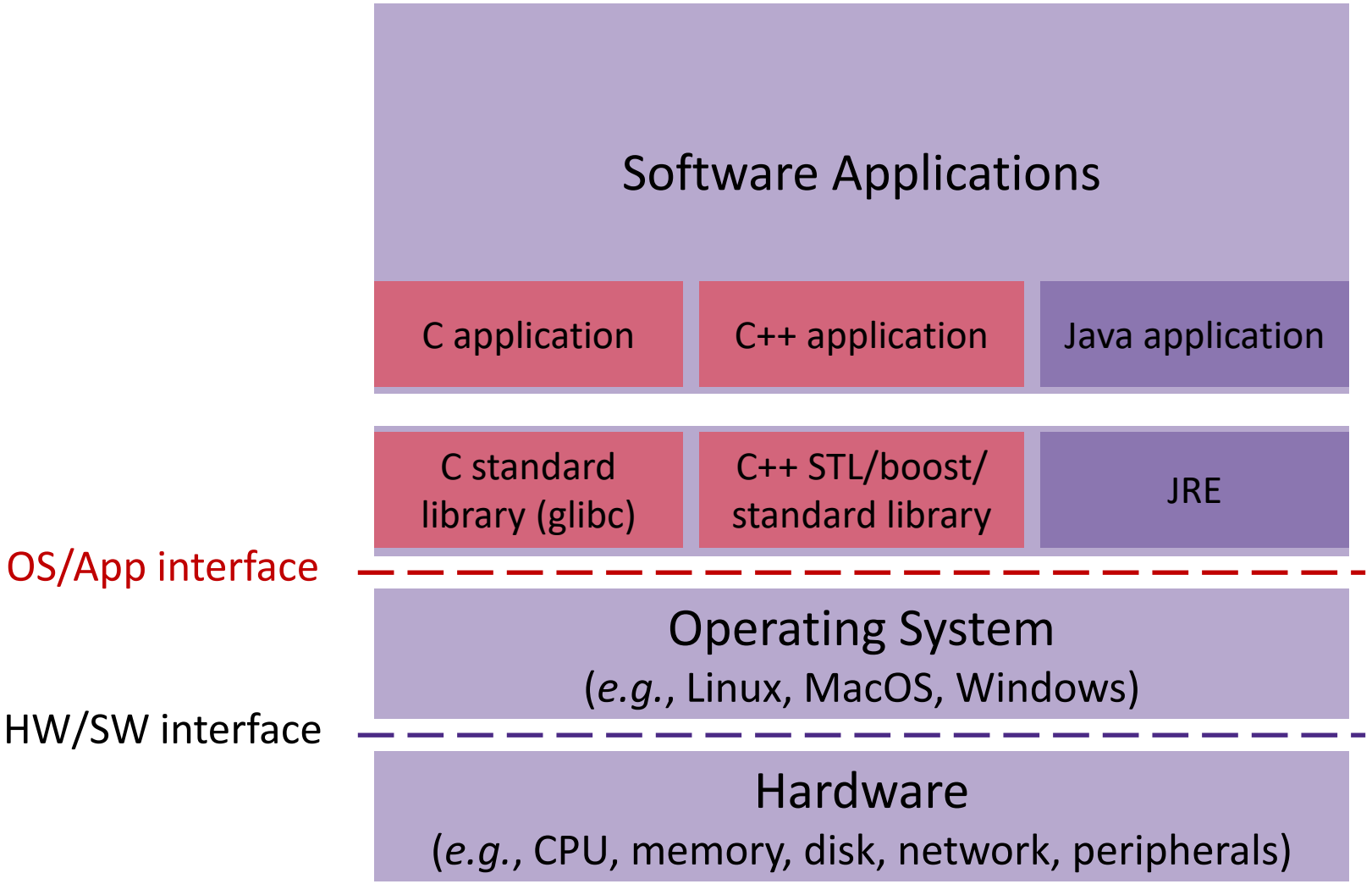
Lecture Outline (2/3)

- Course Policies
  - https://courses.cs.washington.edu/courses/cse333/26sp/syllabus/
  - Summary here, but you ***must*** read the full details online
- **Course Introduction**
- Getting Started in C
  - What do you need to write a C program from scratch?

Lower Computing Layers (1/2)



Lower Computing Layers (2/2)



Systems Programming

- **The programming skills, engineering discipline, and knowledge you need to build…a……system**
  - **Programming:** C / C++ / Rust / Zig / …
  - **Discipline:** testing, debugging, performance analysis
  - **Knowledge:** long list of interesting topics
    - Concurrency, OS interfaces and semantics, techniques for consistent data management, distributed systems algorithms, …
    - Most important: a deep(er) understanding of the "layer below"

Discipline?!?

- Cultivate good habits, encourage clean code
  - Coding style conventions
  - Unit testing, code coverage testing, regression testing
  - Reading/writing documentation (code comments, design docs)
  - Code reviews

- Will take you a lifetime to learn, but oh-so-important, especially for systems code
  - Avoid write-once, read-never code
  - Treat assignment submissions in this class as production code
    - Comments must be updated, no commented-out code, no extra (debugging) output

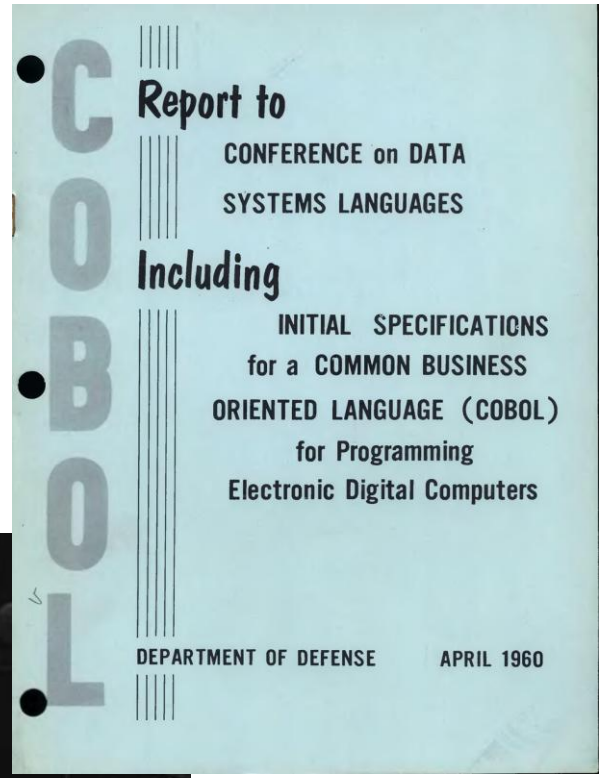
Style Grading in 333

- A **style guide** is a "set of standards for the writing, formatting, and design of documents" – in this case, code
- No style guide is perfect
  - Inherently limiting to coding as a form of expression/art
  - Rules should be motivated (*e.g.*, consistency, performance, safety, readability), even if not everyone agrees
- In 333, we will use a subset of the Google C++ Style Guide
  - Want you to experience adhering to a style guide
  - Hope you view these more as *design decisions* to be considered rather than rules to follow to get a grade
  - We acknowledge that judgments of language implicitly encode certain values and not others

History

1956
IBM keeps releasing new computers and the assembly keeps getting weirder. They develop **FORTRAN** to avoid it.

1959
A consortium of US government and business representatives (including Grace Hopper ☺) are tired of porting code from one machine to another. They invent **COBOL**.

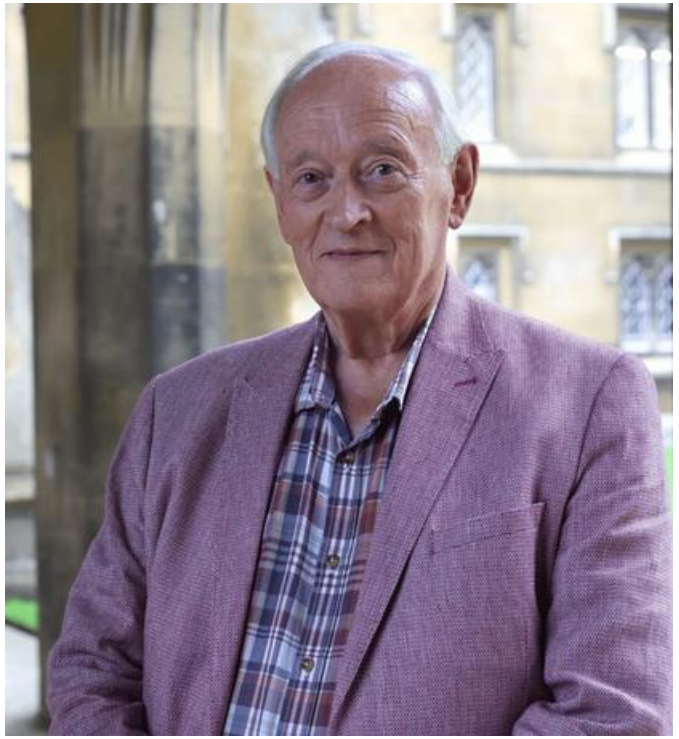


History

1950s: COBOL, FORTRAN

1967:
It's too hard to compile **COBOL** and **FORTRAN**. We don't have enough RAM and we're angry and tired.

Academics at the University of Cambridge invent a language that's harder to use but easier to compile called **BCPL**.



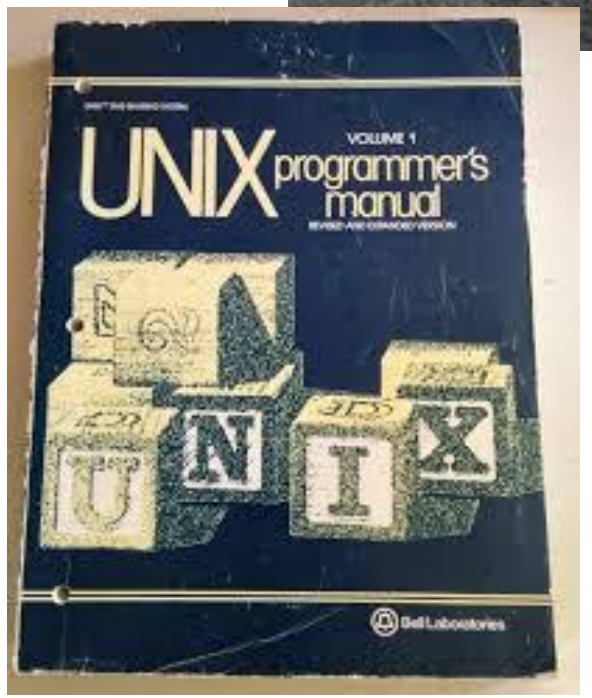
History

1950s: COBOL, FORTRAN

1960s: BCPL

1969:
Bell Labs is developing a new operating system called **UNIX**.

Dennis Ritchie wants to use **BCPL** to write "ls" and "cd" but he can't find a full copy of the language spec, so he vibes his own and calls it "B"



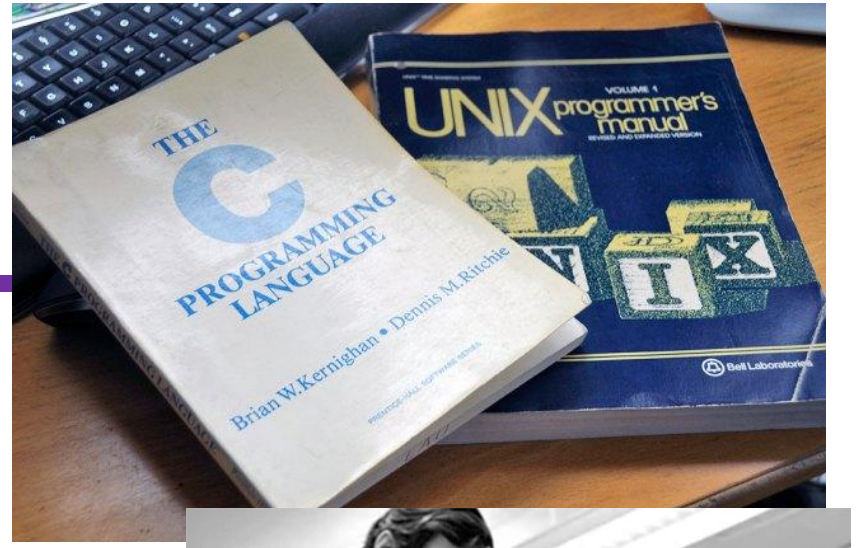
History

1950s: COBOL, FORTRAN

1960s: BCPL, B

1972:
Dennis didn't feel like he did a good job on **B**, so he tried again and called it **C**. **UNIX** ends up being written entirely in **C**.

1978:
Kernighan & Ritchie release the first public edition of the C to the world (the "white book", or "K&R")



History

1950s: COBOL, FORTRAN

1960s: BCPL, B

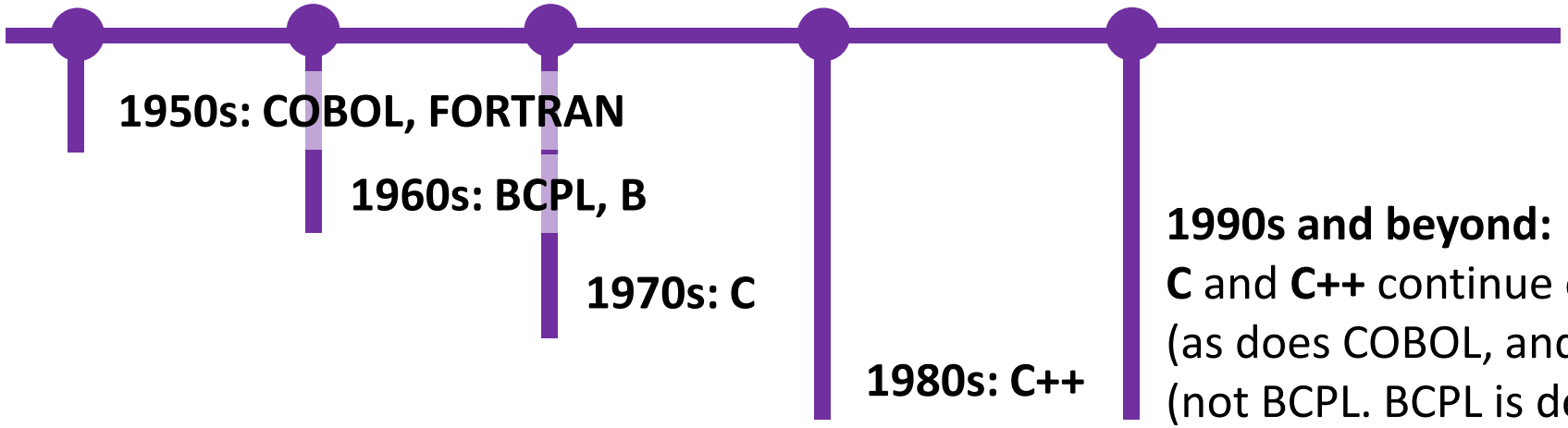
1970s: C

1985:
Computers are better now and compilers can be fancier. Bell Labs is embarrassed about how many features C doesn't have.

Bjarne Stroustrup adds classes to C and calls it C++ ("c plus one").

This is very hard work. He has to convince a lot of C programmers to not hunt him down and yell at him.

History


1950s: COBOL, FORTRAN
1960s: BCPL, B
1970s: C
1980s: C++
1990s and beyond:
C and C++ continue evolving
(as does COBOL, and FORTRAN)
(not BCPL. BCPL is dead.)

History (and a "WHY??!")

1950s: COBOL, FORTRAN

1960s: BCPL, B

1970s: C

1980s: C++

1990s and beyond:
C and **C++** continue evolving
(as does COBOL, and FORTRAN)
(not BCPL. BCPL is dead.)

Lecture Outline (3/3)

- Course Policies
  - https://courses.cs.washington.edu/courses/cse333/26wi/syllabus/
  - Summary here, but you ***must*** read the full details online
- Course Introduction
- **Getting Started in C**
  - **What do you need to write a C program from scratch?**

C Data Structures Review

- C does not support objects!
- **Arrays** are contiguous chunks of memory
  - No implicit initialization; declaration just gives you "mystery data"
  - Don't know their own length, so no bounds checking
- **C-strings** are null-terminated arrays of characters
  - Example: `char x[] = "hi\n";`
  - `string.h` has helpful library/utility functions
    - Documentation: http://www.cplusplus.com/reference/cstring/
- **Structs** are collections of fields (variables)
  - The most object-like, but no methods

Generic C Program Layout


```c
#include <system_files>
#include "local_files"

#define macro_name macro_expr

/* declare functions */
/* declare external variables & structs */

int main(int argc, char* argv[]) {
  /* the innards */
}

/* define other functions */
```

C Syntax: `main` (1/2)

- To get command-line arguments in `main`, use:
  - `int main(int argc, char* argv[])`

- What does this mean?
  - `argc` contains the number of strings on the command line (the executable name counts as one, plus one for each argument)
  - `argv` is an array containing *pointers* to the arguments as strings (more on pointers later)

- Example: `$ ./foo hello 87`
  - `argc = 3`
  - `argv[0]="./foo", argv[1]="hello", argv[2]="87"`

C Syntax: `main` (2/2)

- To get command-line arguments in `main`, use:
  - `int main(int argc, char* argv[])`

- Advantages:
  - Easy to implement – keyboard presses are passed as characters
  - Flexible – can handle any number of arguments

- Disadvantages:
  - Input checking needed by programmer – prevent user misuse
    - Common C idiom is to print back usage messages
  - Data conversion might be needed – if argument is not intended to be used as characters
    - See Exercise 1!

To-do List

- Make sure you're registered on Canvas, Ed Discussion, Gradescope, and Poll Everywhere
  - All user IDs should be your uw.edu email address
- Explore the website *thoroughly*: http://cs.uw.edu/333
- Computer setup: CSE lab or SSH into attu
  - Husky OnNet VPN when you're off campus (required as of 26wi)
- Exercise 1 is due at 11 AM on Friday
  - Find exercise spec on website, submit via GitLab (instructions during section)
  - Sample solution will be posted Friday afternoon
  - **Hint:** look at documentation for `stdlib.h`, `string.h`, and `inttypes.h`
- Check for exercise Gitlab repo tomorrow, then follow our guide
- Pre-Quarter Survey (Canvas) due Friday @ 11:59 PM