Welcome – please set up your Zoom session. We’ll start the actual class meeting at 10:50 am pdt
Lecture Outline

❖ **Course Policies**
  ▪ [https://courses.cs.washington.edu/courses/cse333/20su/syllabus/](https://courses.cs.washington.edu/courses/cse333/20su/syllabus/)
  ▪ Summary here, but you **must** read the full details online

❖ **Course Introduction**

❖ **C Intro**
But first...

❖ It’s all virtual, all the time this quarter

❖ Core infrastructure is same as usual (Gradescope, Gitlab, web, discussion board) except that lab machines are remote login only all quarter

❖ But lectures, sections, office hours – Zoom

❖ Most important: stay healthy, keep your (physical) distance from others, help others both in and out of class
Virtual Lectures

❖ Classes are going to be mostly lectures. Will have some student participation with Poll Everywhere.

❖ Conventions (from page on our web site)
  ▪ Lecture will be recorded and archived – available to class only
  ▪ If you have a question, type “hand” or “question” in Zoom chat window
    • If needed, indicate if we should pause recording while you’re talking
  ▪ Please keep your microphone muted during class unless you’re using it for a question or during breakout room discussions
  ▪ Lecture slides will be posted in advance along with “virtual handouts” for some lectures
Virtual Sections

❖ Sections: more Zoom
  ▪ Not normally recorded so we can have open discussions and group work without people being too self-conscious
  ▪ We’re going to try to produce videos for things that would normally be done as demos or presentations in sections; details tba
    • Those will be available online via canvas
  ▪ Slides and any sample code, worksheets, etc. posted on website

Note that there are now 4 sections. Please register for the one you plan to show up to And even out the distribution. Smaller sections means TA’s can interact and answer questions more easily.
Virtual Everything Else

- Office hours: also Zoom; Will make use of a queue system (more info on website)
  - Not recorded or archived
  - Once gitlab repos are set up, if your question concerns your code (exercises, projects), please push latest code to the repo before meeting with TA to save some time

- We are also offering the chance to ask for 1-on-1 meetings with a staff member. This could help alleviate time zone differences and busy OH’s.

- You will be bombarded with email as we add these things to Canvas/Zoom. Feel free to ignore. 😊
Introductions: Course Staff

❖ Travis McGaha(instructor)
  ▪ First-time Instructor, given lectures previously and a CSE 333 veteran TA.

❖ TAs:

Ramya Challa  Ian Hsiao  Allen Jung

Jeter Arellano  Sylvia Wang  Kyrie Dowling

❖ Get to know us
  ▪ We are here to help you succeed!
Introductions: Students

❖ ~75 students this quarter
  ▪ There are no overload forms or waiting lists for CSE courses

❖ Expected background
  ▪ **Prereq:** CSE 351 – C, pointers, memory model, linker, system calls
  ▪ **Indirect Prereq:** CSE 143 – Classes, Inheritance, Basic Data structures, and general good style practices.
  ▪ CSE 391 or Linux skills needed for CSE 351 assumed
Assignments

❖ Explore the website *thoroughly*: [http://cs.uw.edu/333](http://cs.uw.edu/333)

❖ Computer setup: CSE remote lab, attu, or CSE Linux VM

❖ **Exercise 0 is due 10:30 am Wednesday before class***
  - Find exercise spec on website, submit via Gradescope
  - Sample solution will be posted Friday after class
  - Give it your best shot to get it done more-or-less on time***
    *but we’ll figure out how to work around late exercises for this week…*

❖ **Pre-Quarter survey up on canvas. Due Friday @11:59 pm**
  - Answers are anonymous. Will help us figure out how to make course as great as possible
Communication

❖ **Website:** [http://cs.uw.edu/333](http://cs.uw.edu/333)
  - Schedule, policies, materials, assignments, etc.

❖ **Discussion:** Ed group linked to course home page
  - Must log in using your [@uw.edu](mailto:@uw.edu) Google identity (not cse)
  - Ask and answer questions – staff will monitor and contribute
  - Can post private questions, but students can also help. It is probably worthwhile posting anonymously instead of privately (unless you intend to show your code)

❖ **Staff mailing list:** cse333-staff@cs for urgent things not appropriate for discussion group.

❖ **Course mailing list:** for announcements from staff
  - Registered students automatically subscribed with your @uw email

❖ **Office Hours:** spread throughout the week
  - Schedule & OH queue posted on website. Zoom links are on canvas.
  - Can also e-mail to staff list to make individual appointments
Course Components

❖ Lectures (~26)
  ▪ Introduce the concepts; take notes!!!

❖ Sections (9)
  ▪ Applied concepts, important tools and skills for assignments, and clarification of lectures

❖ Programming Exercises (~20)
  ▪ Roughly one per lecture, due the morning before the next lecture
  ▪ Coarse-grained grading (0, 1, 2, or 3)

❖ Programming Projects (0+4)
  ▪ Warm-up, then 4 “homeworks” that build on each other

❖ Exams: nothing traditional; maybe 1-2 online quizzes
  ▪ Stay tuned, still working on that
Grading (tentative)

- **Exercises**: 30% total
  - Submitted via GradeScope (account info mailed yesterday)
  - Graded on correctness and style by TAs
- **Projects**: 50% total
  - Submitted via GitLab; must tag commit that you want graded
  - Binaries provided if you didn’t get previous part working
- **Quizzes**: ~15%, if we have them
- **Participation**: ~5%
  - Many ways to earn it, as detailed on syllabus. Will be relatively lenient on this.

- More details on course website
  - You **must** read the syllabus there – you are responsible for it
Deadlines and Student Conduct

❖ Late policies (standard quarters)
  ▪ **Exercises:** no late submissions accepted, due 10:30 am
  ▪ **Projects:** 4 late days for entire quarter, max 2 per project
  ▪ Need to get things done on time – difficult to catch up!

❖ Academic Integrity (**read** the full policy on the web)
  ▪ I 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
  ▪ This does **not** mean suffer in silence – learn from the course staff and peers, talk, share ideas; **but** don’t share your work or copy someone else’s work.
Deadlines (this quarter)

- We’re hoping to stay close to a normal schedule to make progress, but...
  - It is an unusual quarter (understatement)
  - We’ll be quite flexible depending on circumstances

- We’re going to start exercises right away
  - Need to discover how to get compute cycles now; no point in putting it off
  - We will be pretty lenient on the exercise grading this quarter.
Deep Breath....

- Any questions, comments, observations, before we go on to, uh, some technical stuff?
Lecture Outline

❖ Course Policies
  ▪ https://courses.cs.washington.edu/courses/cse333/20su/syllabus/

❖ Course Introduction

❖ C Intro
Course Map: 100,000 foot view

OS / app interface (system calls)

HW/SW interface (x86 + devices)

operating system

hardware

C application

C++ application

Java application

C standard library (glibc)

C++ STL/boost/standard library

JRE

CPU memory storage network

GPU clock audio radio peripherals
What is Systems Programming?

❖ The programming skills, engineering discipline, and knowledge you need to build a system

- **Programming**: Usually C / C++

- **Discipline**: testing, debugging, following good practices, and light performance analysis

- **Knowledge**: long list of interesting topics
  - Concurrency, OS interfaces and semantics, techniques for consistent data management, networks, ...
  - 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
  - Documentation (code comments, design docs)
  - Code reviews

- Will take you a lifetime to learn
  - But oh-so-important, especially for systems code
Lecture Outline

❖ Course Policies
  ▪ https://courses.cs.washington.edu/courses/cse333/20su/syllabus/

❖ Course Introduction

❖ C Intro
  ▪ Workflow, Variables, Functions
C

- Created in 1972 by Dennis Ritchie
  - Designed for creating system software
  - Portable across machine architectures
  - Most recent notable updates in 1999 (C99) and 2011 (C11)

- Characteristics
  - “Low-level” language that allows us to exploit underlying features of the architecture – but easy to fail spectacularly (!)
  - Procedural (not object-oriented)
  - Typed but unsafe (possible to bypass the type system)
  - Small, basic library compared to Java, C++, most others….
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`

- **To get command-line arguments in `main`, use:**
  ```c
  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`
C Workflow

Editor (emacs, vi) or IDE (eclipse)

Source files (.c, .h)

Object files (.o)

“COMPILE” (compile + assemble)

Statically-linked libraries

Shared libraries

LINK

LOAD

EXECUTE, DEBUG, ...

EDIT
### C to Machine Code

```c
void sumstore(int x, int y, int* dest) {
    *dest = x + y;
}
```

**C source file**
(sumstore.c)

**C compiler** (`gcc -S`)

**sumstore:***
- `addl %edi, %esi`
- `movl %esi, (%rdx)`
- `ret`

**Assembly file**
(sumstore.s)

**Assembler** (`gcc -c or as`)

**Machine code**
(sumstore.o)

```
400575: 01 fe 89 32 c3
```
When Things Go South...

❖ Errors and Exceptions
  ▪ C does not have exception handling (no \texttt{try/catch})
  ▪ Errors are returned as integer error codes from functions
    • Standard codes found in \texttt{stdlib.h}:
      \texttt{EXIT\_SUCCESS} (usually 0) and \texttt{EXIT\_FAILURE} (non-zero)
    • Return value from \texttt{main} is a status code
  ▪ Because of this, error handling is ugly and inelegant

❖ Crashes
  ▪ If you do something bad, you hope to get a “segmentation fault”
    (believe it or not, this is the “good” option)
Java vs. C (351 refresher)

- Are Java and C mostly similar (S) or significantly different (D) in the following categories?
  - List any differences you can recall (even if you put ‘S’)

<table>
<thead>
<tr>
<th>Language Feature</th>
<th>S/D</th>
<th>Differences in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control structures</td>
<td>S</td>
<td>C has goto (which we will not use)</td>
</tr>
<tr>
<td>Primitive datatypes</td>
<td>S/D</td>
<td>Similar but sizes can differ (char, esp.), unsigned, no boolean, uninitialized data, ...</td>
</tr>
<tr>
<td>Operators</td>
<td>S</td>
<td>Java has &gt;&gt;&gt;&gt;, C has -&gt;</td>
</tr>
<tr>
<td>Casting</td>
<td>D</td>
<td>Java enforces type safety, C does not</td>
</tr>
<tr>
<td>Arrays</td>
<td>D</td>
<td>Not objects, don’t know their own length, no bounds checking</td>
</tr>
<tr>
<td>Memory management</td>
<td>D</td>
<td>Manual (malloc/free), no garbage collection</td>
</tr>
</tbody>
</table>
Primitive Types in C

- **Integer types**
  - `char, int`

- **Floating point**
  - `float, double`

- **Modifiers**
  - `short [int]`
  - `long [int, double]`
  - `signed [char, int]`
  - `unsigned [char, int]`

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>32-bit</th>
<th>64-bit</th>
<th>printf</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char</code></td>
<td>1</td>
<td>1</td>
<td><code>%c</code></td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
<td><code>%hd</code></td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2</td>
<td>2</td>
<td><code>%hu</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td>4</td>
<td>4</td>
<td><code>%d/%i</code></td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>4</td>
<td><code>%u</code></td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
<td><code>%ld</code></td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
<td><code>%lld</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td>4</td>
<td>4</td>
<td><code>%f</code></td>
</tr>
<tr>
<td><code>double</code></td>
<td>8</td>
<td>8</td>
<td><code>%lf</code></td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16</td>
<td><code>%Lf</code></td>
</tr>
<tr>
<td><code>pointer</code></td>
<td>4</td>
<td>8</td>
<td><code>%p</code></td>
</tr>
</tbody>
</table>

Typical sizes – see `sizeofs.c`
C99 Extended Integer Types

- Solves the conundrum of “how big is an `long int`?”

```c
#include <stdint.h>

void foo(void) {
    int8_t a;  // exactly 8 bits, signed
    int16_t b; // exactly 16 bits, signed
    int32_t c; // exactly 32 bits, signed
    int64_t d; // exactly 64 bits, signed
    uint8_t w; // exactly 8 bits, unsigned
    ...
}
```

When byte size matters, use extended integer types.

```c
void sumstore(int x, int y, int* dest) {
    ...
}

void sumstore(int32_t x, int32_t y, int32_t* dest) {
    ...
}
```
CSE333, Summer 2020

L01: Intro, C

Basic Data Structures

❖ C does not support objects!!!

❖ Arrays are contiguous chunks of memory
  ▪ Arrays have no methods and do not know their own length
  ▪ Can easily run off ends of arrays in C – security bugs!!

❖ Strings are null-terminated char arrays
  ▪ Strings have no methods, but string.h has helpful utilities

```
char* x = "hello\n";
```

❖ Structs are the most object-like feature, but are just collections of fields – no “methods” or functions
Function Definitions

- Generic format:

```c
returnType fname(type param1, ..., type paramN) {
    // statements
}
```

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;

    for (i = 1; i <= max; i++) {
        sum += i;
    }

    return sum;
}
Function Ordering

- You *shouldn’t* call a function that hasn’t been declared yet

```c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += i;
  }

  return sum;
}
```
Solution 1: Reverse Ordering

- Simple solution; however, imposes ordering restriction on writing functions (who-calls-what?)

```c
#include <stdio.h>

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}
```
Solution 2: Function Declaration

- Teaches the compiler arguments and return types; function definitions can then be in a logical order

```c
#include <stdio.h>

int sumTo(int); // func prototype

int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```

Hint: code examples from slides are on the course web for you to experiment with
Function Declaration vs. Definition

❖ C/C++ make a careful distinction between these two

❖ **Definition:** the thing itself
  - e.g. code for the function, variable definition that creates storage
  - Must be **exactly one** definition of each thing (no duplicates)

❖ **Declaration:** description of a thing
  - e.g. function prototype, external variable declaration
    - Often in header files and incorporated via `#include`
    - Should also `#include` declaration in the file with the actual definition to check for consistency
  - Needs to appear in **all files** that use that thing
    - Should appear before first use
Multi-file C Programs

C source file 1
(sumstore.c)

```c
void sumstore(int x, int y, int* dest) {
    *dest = x + y;
}
```

C source file 2
(sumnum.c)

```c
#include <stdio.h>

void sumstore(int x, int y, int* dest);

int main(int argc, char** argv) {
    int z, x = 351, y = 333;
    sumstore(x, y, &z);
    printf("%d + %d = %d\n", x, y, z);
    return 0;
}
```

Note that some of lecture code has bad style to demo things. (Such as this code)

Compile together:
```bash
$ gcc -o sumnum sumnum.c sumstore.c
```
Compiling Multi-file Programs

- The **linker** combines multiple object files plus statically-linked libraries to produce an executable
  - Includes many standard libraries (e.g. libc, crt1)
    - A library is just a pre-assembled collection of .o files

```
sumstore.c  gcc -c  sumstore.o

sumnum.c    gcc -c  sumnum.o

libraries (e.g. libc)

ld or gcc

sumnum
```
Peer Instruction Question

❖ Which of the following statements is FALSE?

- Vote at http://PollEv.com/cse33320su

A. With the standard `main()` syntax, it is always safe to use `argv[0]`.

B. We can’t use `uint64_t` on a 32-bit machine because there isn’t a C integer primitive of that length. Note that instructions performed with `int64_t` are more complicated since registers are only 32 bits, but it does work.

C. Using function declarations is beneficial to both single- and multi-file C programs.

D. When compiling multi-file programs, not all linking is done by the Linker.

E. We’re lost...
To-do List

❖ Explore the website *thoroughly*: [http://cs.uw.edu/333](http://cs.uw.edu/333)

❖ Computer setup: CSE remote lab, attu, or CSE Linux VM

❖ **Exercise 0 is due 10:30 am Wednesday before class**
  - Find exercise spec on website, submit via Gradescope
  - Sample solution will be posted Friday after class
  - Give it your best shot to get it done more-or-less on time*
    *but we’ll figure out how to work around late exercises for this week…

❖ Pre-Quarter survey up on canvas. Due Friday @11:59 pm
  - Answers are anonymous. Will help us figure out how to make course as great as possible

❖ Gradescope accounts created just before class
  - Userid is your uw.edu email address
  - Exercise submission: find CSE 333 20su, click on the exercise, drag-n-drop file(s)! That’s it!! Ignore any messages about autograding not using this quarter