

Intro, Getting Started in C

CSE 333 Summer 2023

Instructor: Timmy Yang

Teaching Assistants:

Jennifer Xu

Leanna Nguyen

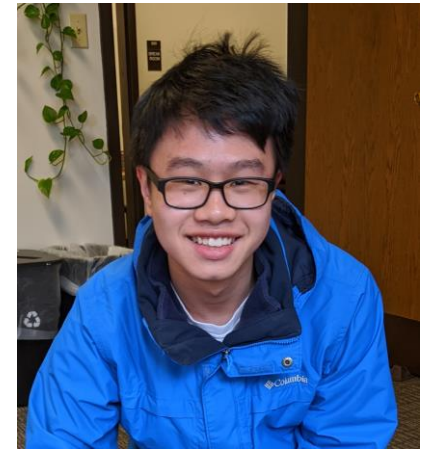
Pedro Amarante

Sara Deutscher

Tanmay Shah

Introductions: Course Staff

- ❖ Your Instructor: just call me Timmy
 - Part-Time Lecturer
 - First-time lecturing! Learning along with you 😊



- ❖ TAs:



Jennifer



Leanna



Pedro



Sara



Tanmay

- Available in section, office hours, and discussion board
- ❖ 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

- ❖ ~60 students registered
- ❖ 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 or 123
 - 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
 - Take advantage of partner work, where permissible, to *learn*, not just get a grade
 - Good chance to learn collaboration tools and tricks

Lecture Outline

❖ Course Policies

- <https://courses.cs.washington.edu/courses/cse333/23su/syllabus.html>
- Digest here, but you *must* read the full details online

❖ Course Introduction

❖ Getting Started in C

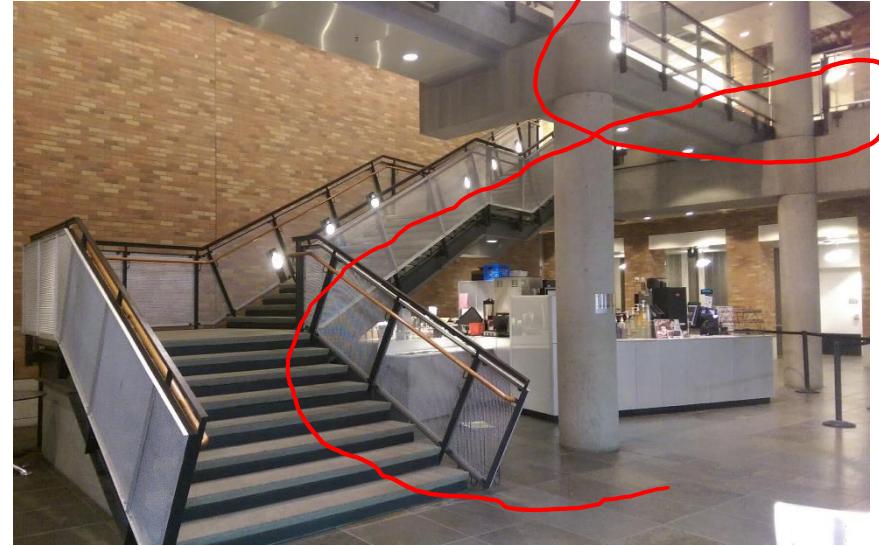
- What do you need to write a C program from scratch?

Communication

- ❖ **Website:** <http://cs.uw.edu/333>
 - Schedule, policies, materials, assignments, etc.
- ❖ **Discussion:** <https://edstem.org/us/courses/40647>
 - Announcements made here
 - Ask and answer questions – staff will monitor and contribute
- ❖ **Office Hours:** Google Sheet queue for both in-person and virtual OHs, which are spread throughout the week
- ❖ **1-on-1 Meetings:** can request a limited number of appointments via Google Form
- ❖ **Anonymous feedback**

In-Person Office Hours

- ❖ Ex: Allen 3rd floor breakout
 - Up the stairs in the CSE Atrium (Allen Center, not Gates)
 - At the top of two flights, the open area with the whiteboard wall is the 3rd floor breakout!



Course Components

- ❖ Lectures (26)
 - Introduce the concepts; take notes
- ❖ Sections (9)
 - Applied concepts, important tools and skills for assignments, clarification of lectures, quiz review and preparation
- ❖ Programming Exercises (12)
 - One due roughly every 4-5 days
 - We are checking for: **correctness, memory issues, code style/quality**
- ❖ Programming Project (0+4)
 - Warm-up, then 4 “homework” that build on each other
- ❖ Take-home Quizzes (4)
 - Initial dates on website under Quizzes, encourage review of course content

Grading

- ❖ **Exercises: 30% total**
 - Submitted via Gradescope (under your UW email)
 - Graded on correctness and style by autograders and TAs
- ❖ **Projects: 43% total**
 - Submitted via GitLab; must tag commit that you want graded
 - Binaries provided if you didn't get previous part working
 - Graded on test suite, manual tests, and style
- ❖ **Quizzes: 24% total (~8% each)**
 - Take-home; short answer questions based on assignments
- ❖ **Effort, Participation, and Altruism: 3%**
 - Many ways to earn credit here, relatively lenient on this

Academic Integrity and Student Conduct

- ❖ 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 or copy work that is supposed to be yours
 - Partners allowed this quarter on programming assignments!
- ❖ If you find yourself in a situation where you are tempted to perform academic misconduct, please reach out to Timmy to explain your situation instead
 - See the Extenuating Circumstances section of the syllabus

Lecture Outline

❖ Course Policies

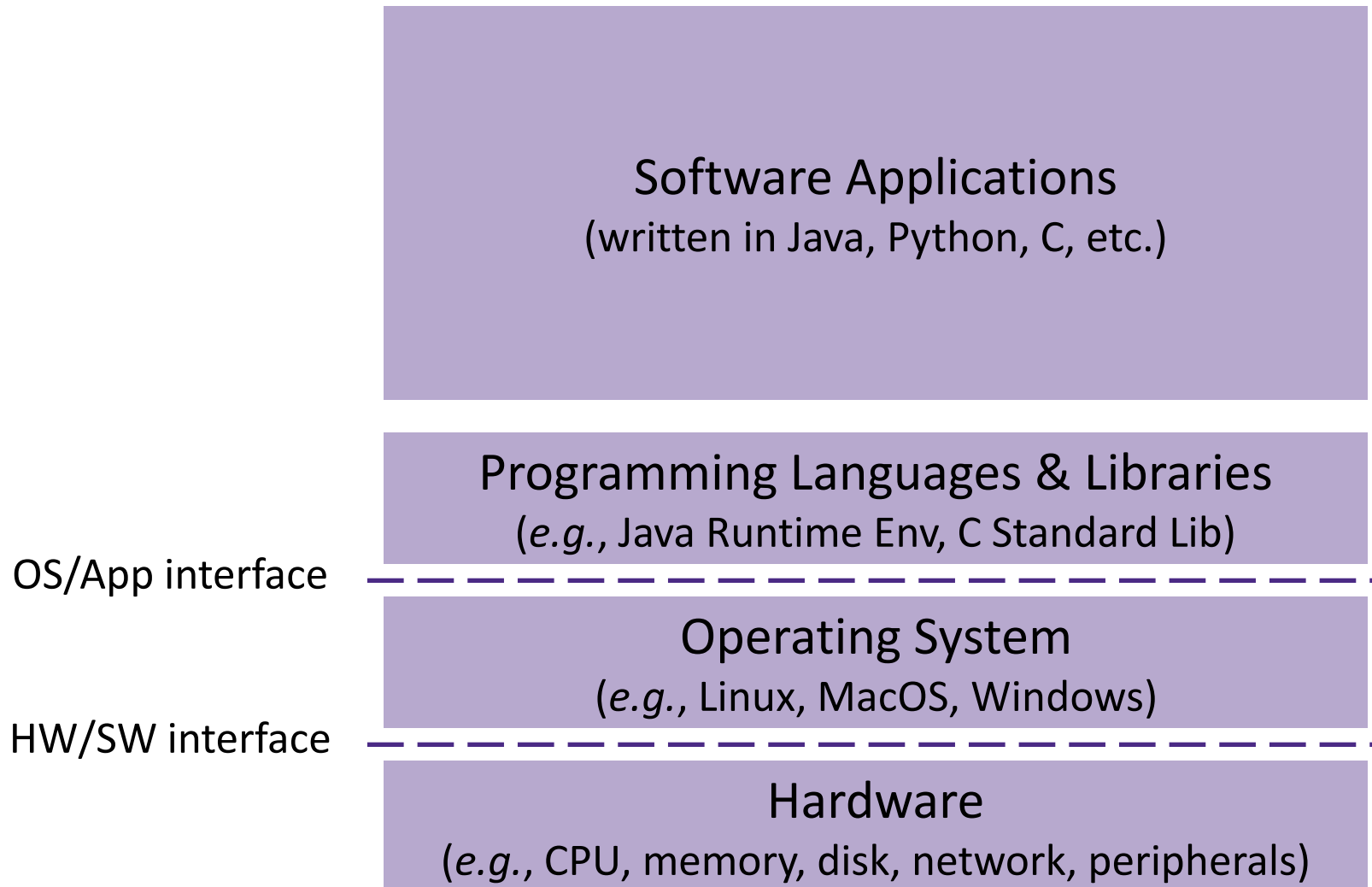
- <https://courses.cs.washington.edu/courses/cse333/23su/syllabus/>
- Summary here, but you *must* read the full details online

❖ Course Introduction

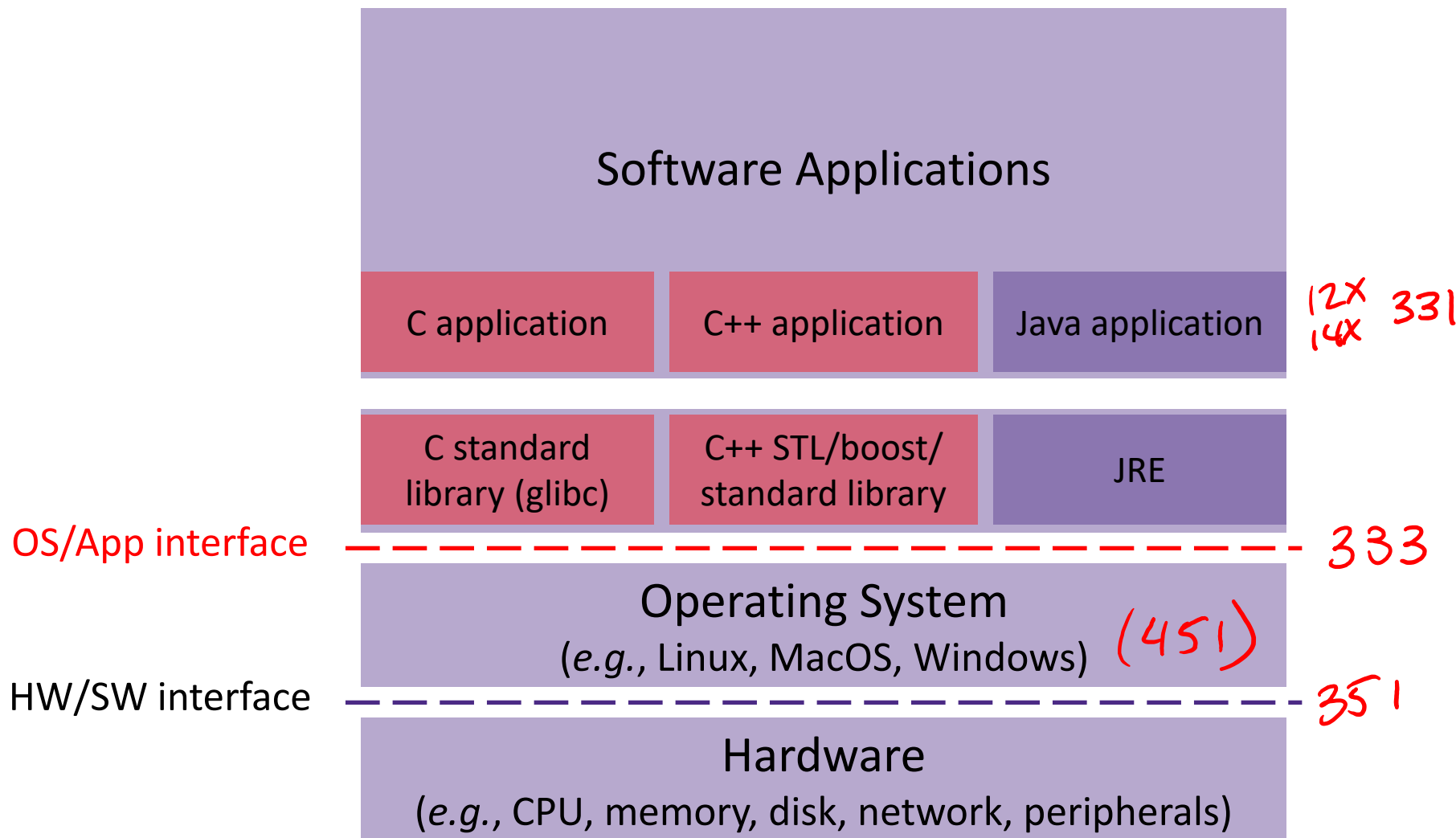
❖ Getting Started in C

- What do you need to write a C program from scratch?

Layers of Computing Below Programming



Layers of Computing Below Programming



Systems Programming

- ❖ **The programming skills, engineering discipline, and knowledge you need to build a system**
 - **Programming:** C / C++
 - **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

Lecture Outline

❖ Course Policies

- <https://courses.cs.washington.edu/courses/cse333/23su/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";` X 'h'|'i'|'\n'|'\0'
 - `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

```
#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:

```
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`

- ❖ 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!

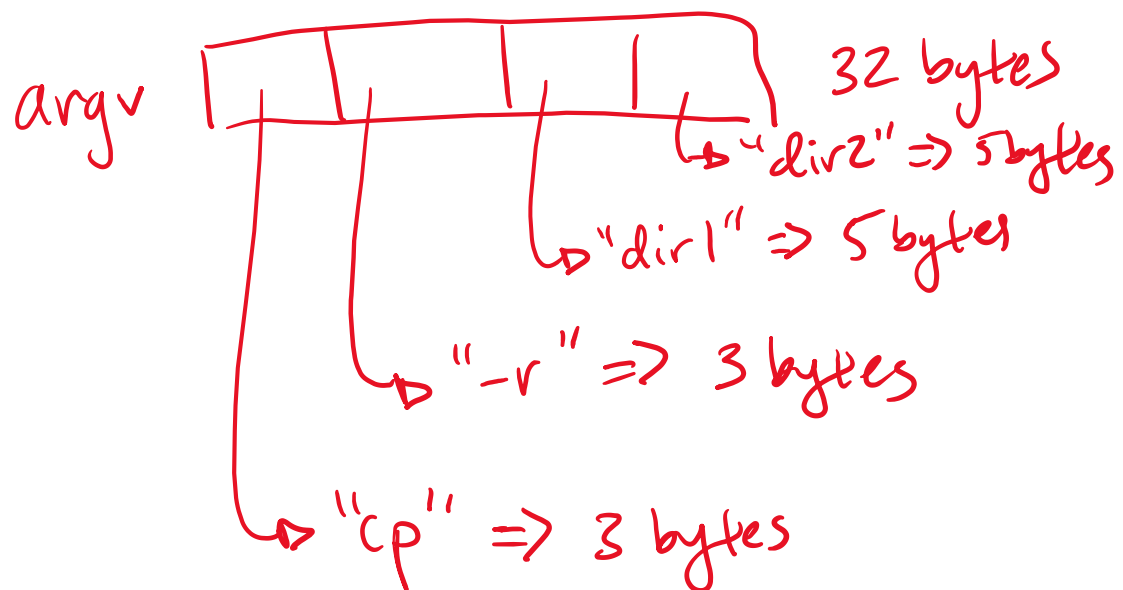
Poll Everywhere

pollev.com/cse333

How much memory would you expect to be allocated for `argv` & all of its pointed-to arrays?

```
$ cp -r dir1 dir2
```

- A. 44 bytes
- B. 48 bytes**
- C. 52 bytes
- D. 56 bytes
- E. We're lost...



Printing in C

❖ `int printf(const char* format, ...);`

- Can check documentation to learn about (1) parameters, (2) the return value, and (3) *error handling*
 - <https://www.cplusplus.com/reference/cstdio/printf/>
- Very important to use correct format specifier for the value you want to print, otherwise implicit casting will occur

specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000

ex.
use %d
3.33 → 3



Error Handling

❖ Errors and Exceptions

- C does not have exception handling (no `try/catch`)
- Errors are returned as **integer error codes** from functions
 - Because of this, error handling is ugly and inelegant
 - For readability, `CONSTANT_NAMES` are defined to abstract away the actual integer values – need to look up in documentation
- Global variable **`errno`** holds value of last system error

❖ Status codes and signals

- Processes exit (e.g., `return` from `main`) with status code
 - Standard codes found in `stdlib.h`:
`EXIT_SUCCESS` (usually 0) and `EXIT_FAILURE` (non-zero)
- “Crashes” trigger signals from OS (e.g., `SIGSEGV` for segfault)

→ Autograders check for this

usu. 1

Function Definitions

❖ Generic format:

```
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

[Note](#): code examples from slides are posted on the course website for you to experiment with!

sum_badorder.c

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

// 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?)

sum_betterorder.c

```
// 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 EXIT_SUCCESS;
}
```



Solution 2: Function Declaration

- ❖ Teaches the compiler the arguments and return types; function definitions can then be in a logical order
 - Function comment usually by the *prototype*

sum_declared.c

```
// sum of integers from 1 to max
int SumTo(int max); // func prototype

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

int SumTo(int max) {
    int i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```

Function Declaration vs. Definition

- ❖ C/C++ make a careful distinction between these two
- ❖ **Definition:** the thing itself
 - *e.g.*, code for 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

333 Workflow Aids/Upgrades

- ❖ See **Linux → Text Editors** on website for how to configure vim or VS Code for use in this class
 - From vi/vim, can compile and execute code without ever leaving the editor using "`: ! <cmd>`"
 - For VS Code, can connect to attu remotely and take advantage of the IDE features
 - From either text editor, you will want to get comfortable navigating and editing multiple files *simultaneously*
- ❖ We will learn the basics of Makefiles to simplify the compilation steps into the command `make`

To-do List

- ❖ Make sure you're registered on Canvas, Ed Discussion, Gradescope, and Poll Everywhere (all **uw.edu** email address)
- ❖ Explore the website *thoroughly*: <http://cs.uw.edu/333>
- ❖ Computer setup: CSE lab, attu, or 23su CSE Linux VM
- ❖ **Pre-Quarter Survey** (Canvas) due Friday @ 11:59 pm
- ❖ **Exercise 1** is due Friday @ 1 pm
 - Find exercise spec on website, submit via Gradescope
 - **Hint:** look at documentation for [stdlib.h](#), [string.h](#), and [inttypes.h](#)
- ❖ **Homework 0** (Gitlab) is due Monday @ 11:59 pm
 - Gitlab email sent when repos created – no action needed
 - **Make a private Ed post if you don't have a repo or the hw0 files**