

Systems Programming

Intro, Getting Started in C

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

Introductions: Instructor

❖ Justin Hsia (he/him)

- Engineering background (EE, MechE, some Synth Bio), but a teacher at heart
 - Love helping students pursue teaching careers
- Raising a kindergartener and an infant/toddler:
 - Will be sick 🤒 and tired 😩 at various points this quarter
- Interests include ultimate, a cappella, board games, hiking, anime, and more!
 - Learn more about me through the off-topic questions in the lecture posts



Introductions: 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

- ❖ ~240 students registered, split across two lectures
- ❖ 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!
 - College is a unique time to get to know lots of different people
 - Forming bonds with classmates makes class more enjoyable and can help with learning

Lecture Outline (1/3)

❖ Course Policies

- <https://courses.cs.washington.edu/courses/cse333/26au/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?

Staff-Student Communication

- ❖ **Website:** <http://cs.uw.edu/333>

- Schedule, policies, materials, assignments, etc.

- ❖ **Discussion:** <https://edstem.org/us/courses/101845/>

- Announcements made here
- Ask and answer questions – staff will monitor and contribute

- ❖ **Office Hours:** Google Sheet queue (UW login) for both in-person and virtual OHs

- ❖ **1-on-1 Meetings:** can request a limited number of appointments via Google Form (UW login)

- ❖ **Anonymous feedback**

Office Hours

❖ Check Weekly Calendar for scheduled office hours:

- Zoom meeting links are found in the Zoom tab within Canvas

❖ All office hours will use a Google Sheets queue

- Fill out the first 3 columns to enter the queue:

❖ 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

Weekly Calendar ([Subscribe](#))

< Prev Next > Oct 11 – 17, 2026 Compact Week List

Sun 10/11	Mon 10/12	Tue 10/13	Wed 10/14	Thu 10/15	Fri 10/16	Sat 10/17
	11:30a - 12:20p Lecture A C: Linking, File I/O JHN 102	10a - 11a Office Hours Megan CSE1 Third Floor Breakout	Ex-3 Due	HW1 Due	MeetStf Due	11:30a - 1:30p Office Hours Shrihan CSE1 Third Floor Breakout
	12:30p - 1:30p Office Hours Justin CSE 544	1:30p - 2:30p Office Hours Aadi & Camden CSE2 121	10a - 11a Office Hours Justin Zoom and CSE 544	Section POSIX I/O	10a - 11:30a Office Hours Yusong CSE2 121	1:30p - 2:30p Office Hours Andrew Zoom
	1:30p - 2:20p Lecture B C: Linking, File I/O CSE2 G10	2:30p - 3:30p Office Hours Aadi & Aviel CSE2 121	11:30a - 12:20p Lecture A Buffering, POSIX I/O, System Calls JHN 102	1:30p - 2:30p Office Hours Laura & Yusong CSE2 121	11:30a - 12:20p Lecture A Makefiles, C++ Preview JHN 102	

Full Name(s)	Category	Description	Time Queued	Staff	Status
Example 1	Concept	What are the advantages and disadvantages of buffering?		Justin	Done
Example 2	Debugging	HW1: valgrind memory leak in Part B from HashTable.		Justin	Done
Example 3	Spec	EX3: what is required for the testing code in main?		Justin	Done
Example 4	Tools	GDB: how do I examine memory on the stack?		Justin	Done

Course Components

- ❖ Lectures (28)
 - Introduce the concepts; take notes!!!
- ❖ Sections (10)
 - Applied concepts, important tools and skills for assignments, exam preparation
- ❖ Exercises (12)
 - Small problems to familiarize you with C/C++ fundamentals; submitted via GitLab
- ❖ Homework (4)
 - Web search server built in progressive steps; submitted via GitLab
- ❖ In-Person Exams (2)
 - **Midterm:** Monday, November 2 from 5:30–6:40 PM
 - **Final:** Wednesday, December 16 from 12:30–2:20 PM

Grading

❖ **Exercises:** 15% total

- Graded on a 9-point scale for correctness and style by autograders and TAs
- Lowest exercise score dropped
- No late work accepted (solutions are released immediately)

❖ **Homework:** 30% total

- Binaries provided if you didn't get the previous part working
- Graded on test suite, manual tests, and style
- Late day system + 6 tokens to avoid penalties

❖ **Exams:** Midterm (25%) and Final (25%)

❖ **Effort, Participation, and Altruism:** 5%

- Many ways to earn credit here; relatively lenient on this

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
- ❖ We are using the Allen School's [“AI as a Reference Tool” policy](#)
 - AI may help with general questions and ungraded work, but not graded assignments or assessments
 - More explanation for why this class still focuses on programming by hand can be found on our [Generative AI webpage](#)
- ❖ If you find yourself in a situation where you are tempted to perform academic misconduct, please reach out to the instructor to explain your situation instead
 - See the [Extenuating Circumstances](#) section of the syllabus



Lecture Outline (2/3)

❖ Course Policies

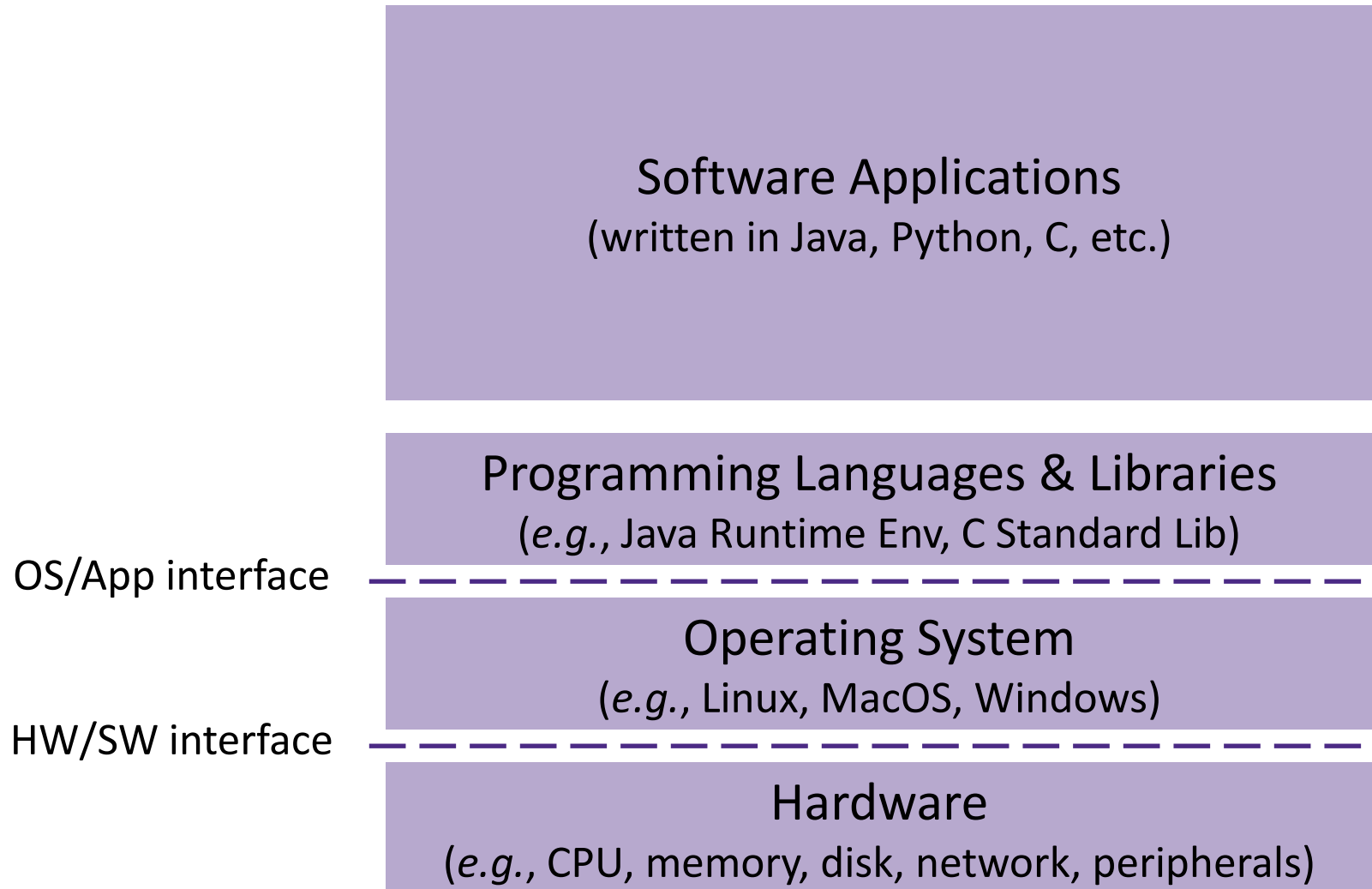
- <https://courses.cs.washington.edu/courses/cse333/26au/syllabus.html>
- Summary here, but you ***must*** read the full details online

❖ Course Introduction

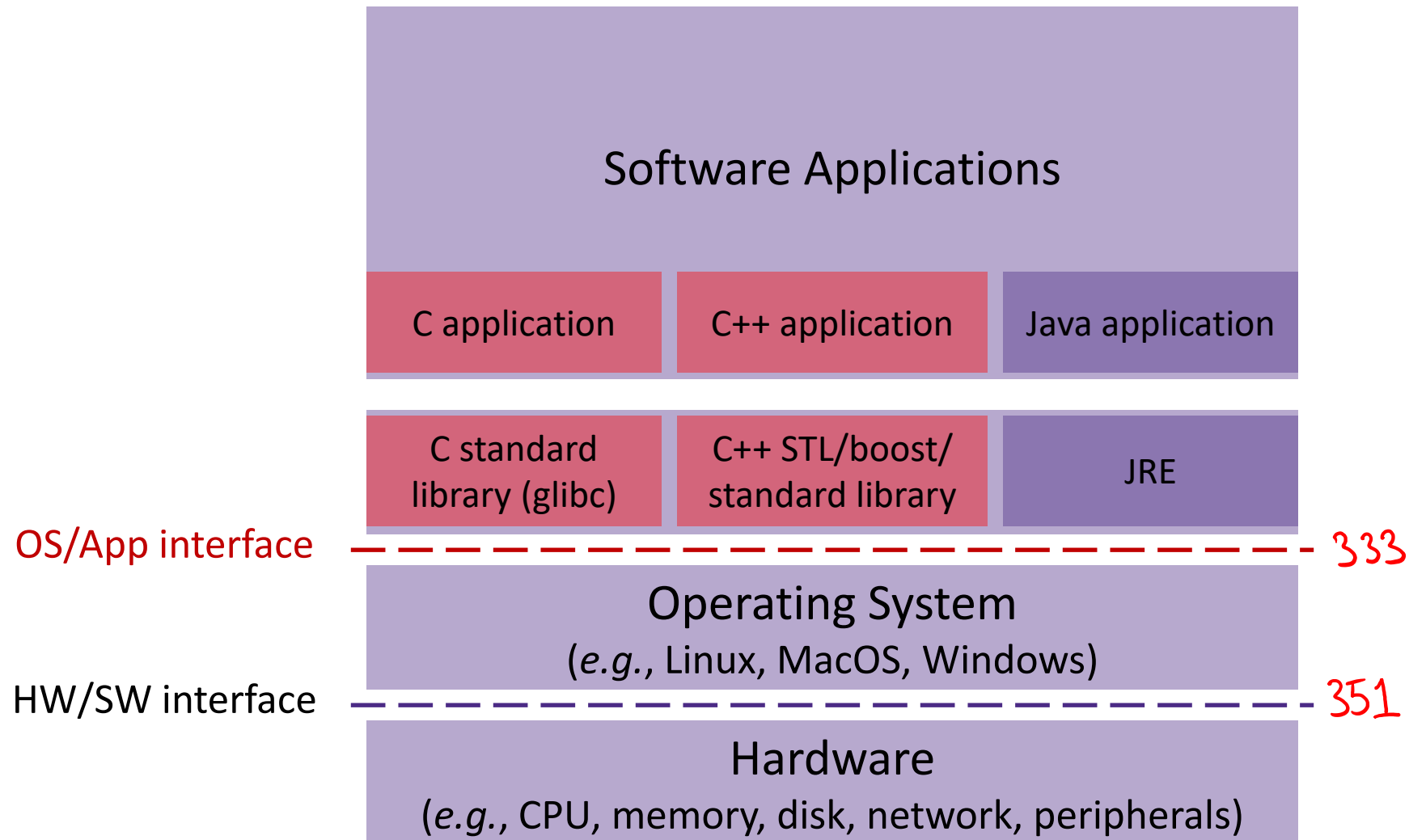
❖ Getting Started in C

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

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, networking, 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 (3/3)

❖ Course Policies

- <https://courses.cs.washington.edu/courses/cse333/26au/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?

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> } pull in declarations
#include "local_files"

#define macro_name macro_expr

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

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

/* define other functions */
```

often moved into other files

get input from terminal command line

C Syntax: `main` (1/2)

- ❖ To get command-line arguments in `main`, use:

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

instead of: `int main()`
same as `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) *needed because C doesn't track array lengths!*
- `argv` is an array containing *pointers* to the arguments as strings, "null-terminated" with a NULL pointer (in modern C standards)

v for "vector"

- ❖ Example: `$ ./foo hello 87`

String or number?

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

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 the programmer to prevent user misuse
 - Common C idiom is to print back usage messages
- Data conversion might be needed if argument is not intended 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
```

argc 4

A. 48 bytes

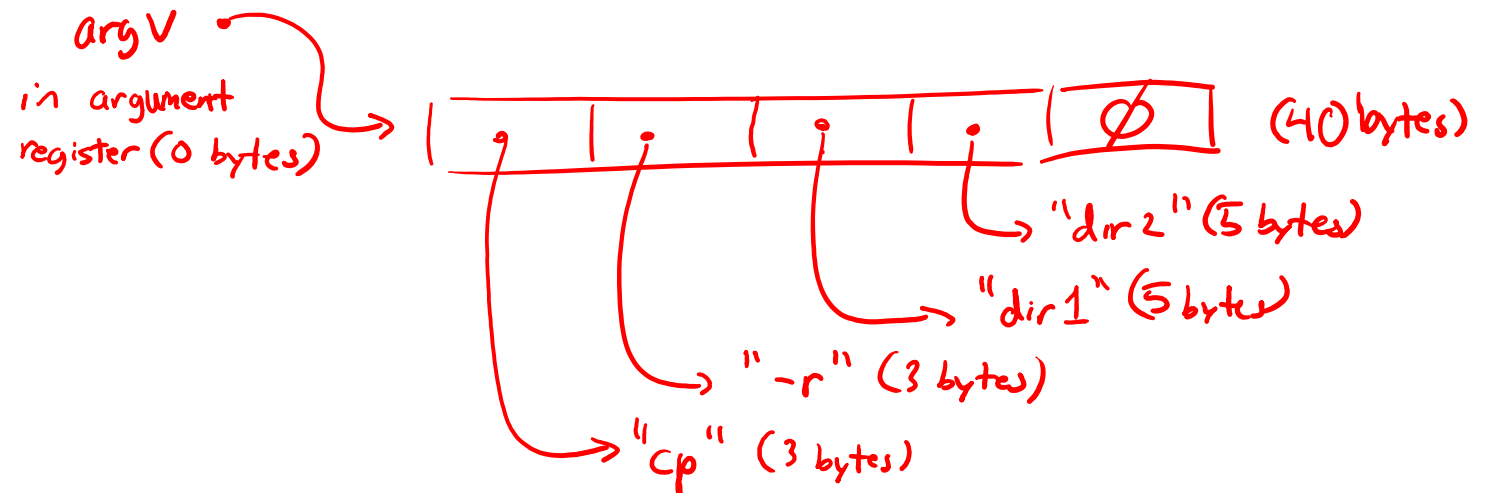
B. 52 bytes

C. 56 bytes

D. 60 bytes

E. 64 bytes

F. 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 the 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 |

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 ← our autograders check for these!

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

Function Definitions

❖ Generic format:

```
ReturnType FuncName(type param1, ..., type paramN) {  
    // statements  
}
```

■ Example:

```
// 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 compiler goes line-by-line:

■ Example:

```
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;  
}
```

fix

← defined here

???

Note: code examples from slides are posted on the course website for you to experiment with!

sum_badorder.c

Solution 1: Reverse Ordering

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

- Example:

```
// sum of integers from 1 to max
int sumTo(int max) { ← defined first ✓
    int i, sum = 0;

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

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

sum_betterorder.c

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*

■ Example:

declared here

```
// 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;  
}
```

has seen already

defined here

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

parameter names optional,
but strongly recommended

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
- more on this in Lecture 5*

To-do List (1/2)

- ❖ Check access: Canvas, Ed Discussion, Gradescope, Poll Everywhere
 - All user IDs should be your **uw.edu** email address
- ❖ Explore the website *thoroughly*: <http://cs.uw.edu/333>
 - In particular, the syllabus and generative AI policy
- ❖ Computer setup: CSE lab or SSH into attu
 - [Husky OnNet](#) VPN when you're off campus (required as of 26wi)
 - Check for GitLab repo NOW, then follow [the GitLab guide](#) to clone and add SSH key *before* Section 1 tomorrow
 - See the [text editors page](#) for how to configure vim or VS Code for use in this class
 - You will want to get comfortable navigating and editing multiple files *simultaneously*

To-do List (2/2)

- ❖ Pre-Quarter Survey (Canvas) due Friday @ 11:59 PM
- ❖ **Exercise 1 is due at 11 AM on Monday**
 - Find exercise spec on website, submit via GitLab (demo during Section 1)
 - Sample solution will be posted Monday afternoon
 - **Hint:** look at documentation for [stdlib.h](#), [string.h](#), and [inttypes.h](#)