# Intro, C refresher

## CSE 333 Spring 2019

**Instructor:** Justin Hsia

**Teaching Assistants:**

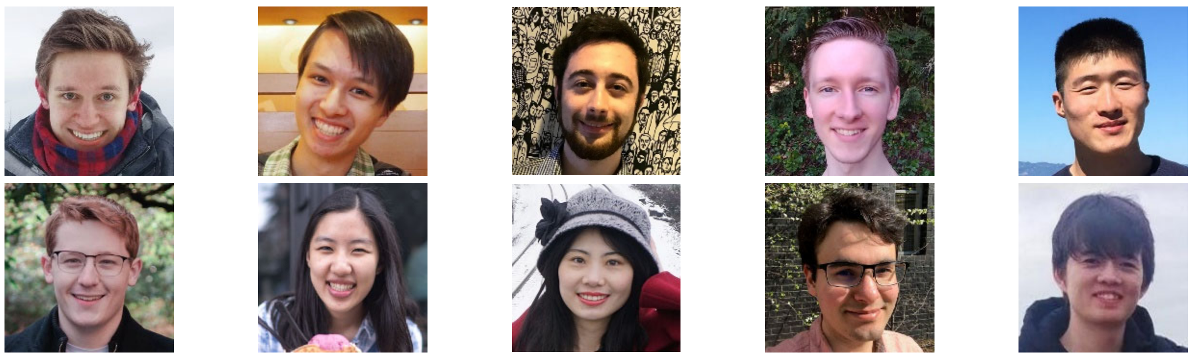
| | | |
|---|---|---|
| Aaron Johnston | Andrew Hu | Daniel Snitkovskiy |
| Forrest Timour | Kevin Bi | Kory Watson |
| Pat Kosakanchit | Renshu Gu | Tarkan Al-Kazily |
| Travis McGaha | | |

# Lecture Outline

- **Course Introduction**
- Course Policies
  - https://courses.cs.washington.edu/courses/cse333/19sp/syllabus/
- C Intro

# Introductions: Course Staff

- Your Instructor: just call me Justin
  - From California (UC Berkeley and the Bay Area)
  - I like: teaching, the outdoors, board games, and ultimate
  - Excited to be teaching this course for the 2nd time!
- TAs:



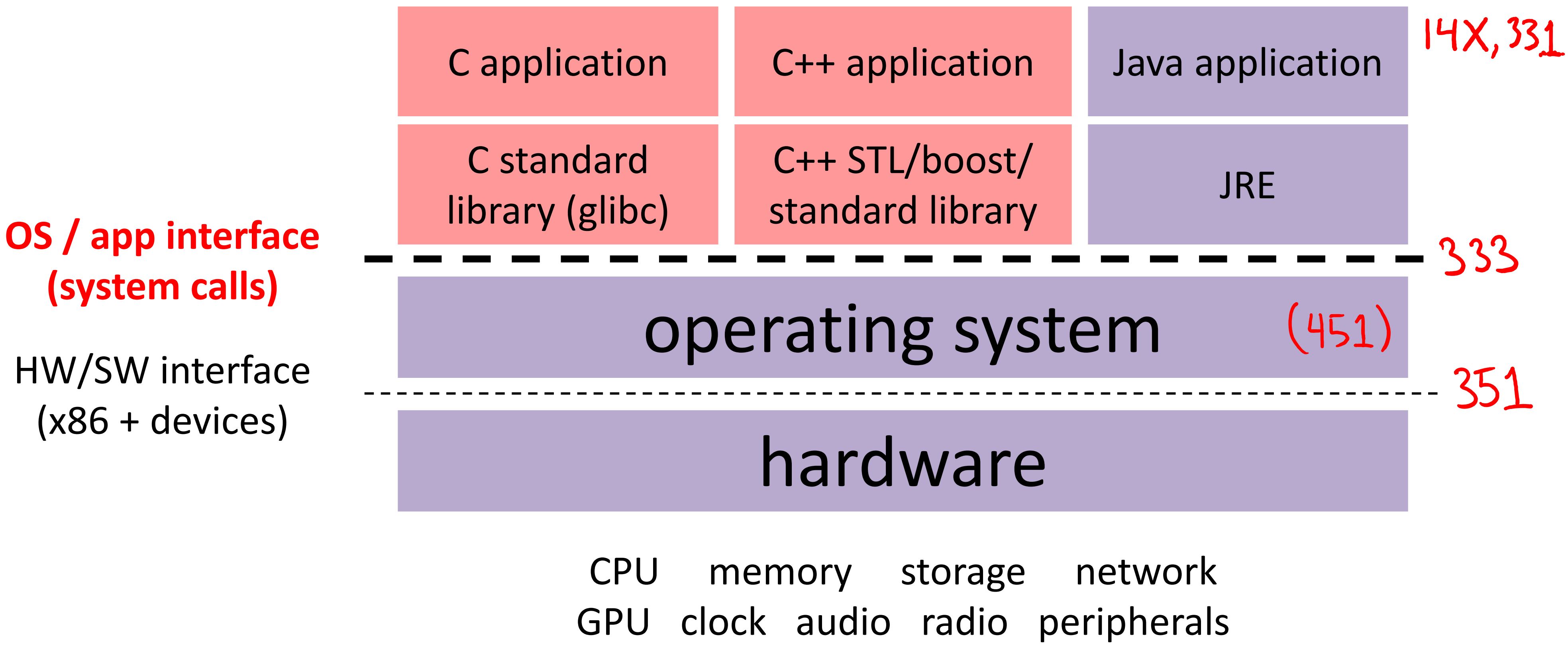
  - Available in section, office hours, and discussion group
  - An invaluable source of information and help
- Get to know us
  - We are here to help you succeed!

# Introductions: Students

- ~160 students registered, split across two lectures
  - There are no overload forms or waiting lists for CSE courses
    - Majors must add using the UW system as space becomes available
    - Non-majors should work with undergraduate advisors to handle enrollment details (over in the *new* Gates Center!)

- Expected background
  - **Prereq:** CSE 351 – C, pointers, memory model, linker, system calls
  - CSE 391 or Linux skills needed for CSE 351 assumed

# Course Map: 100,000 foot view



# 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
  - 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

# Lecture Outline

- Course Introduction
- **Course Policies**
  - https://courses.cs.washington.edu/courses/cse333/19sp/syllabus/
  - Digest here, but you ***must*** read the full details online
- C Intro

# Communication

- Website: http://cs.uw.edu/333
  - Schedule, policies, materials, assignments, etc.
- Discussion: http://piazza.com/washington/spring2019/cse333
  - Announcements made here
  - Ask and answer questions – staff will monitor and contribute
- Office Hours: spread throughout the week
  - Can e-mail/private Piazza post to make individual appointments
- Anonymous feedback:
  - Comments about anything related to the course where you would feel better not attaching your name

# Course Components

- Lectures (28)
  - Introduce the concepts; take notes!!!
- Sections (10)
  - Applied concepts, important tools and skills for assignments, clarification of lectures, exam review and preparation
- Programming Exercises (19)
  - One for most lectures, due the morning before the next lecture
  - New grading scheme (correctness, tools check, code style/quality)
- Programming Projects (0+4)
  - Warm-up, then 4 “homework” that build on each other
- Exams (2)
  - **Midterm:** Friday, May 10, 5:00-6:10 [joint]
  - **Final:** Wednesday, June 12, 12:30-2:20 [joint]

# Grading

- **Exercises:** 20% total
  - Submitted via GradeScope (account info mailed later today)
  - Graded on correctness and style by TAs
- **Projects:** 40% total
  - Submitted via GitLab; must tag commit that you want graded
  - Binaries provided if you didn't get previous part working
- **Exams:** Midterm (15%) and Final (20%)
  - Several old exams on course website
- **EPA:** Effort, Participation, and Altruism (5%)

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

# Deadlines and Student Conduct

- Late policies
  - Exercises: no late submissions accepted, due 11 am
  - Projects: 4 late day "tokens" for quarter, max 2 per homework
  - 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 or copy work that is supposed to be yours

# Hooked on Gadgets

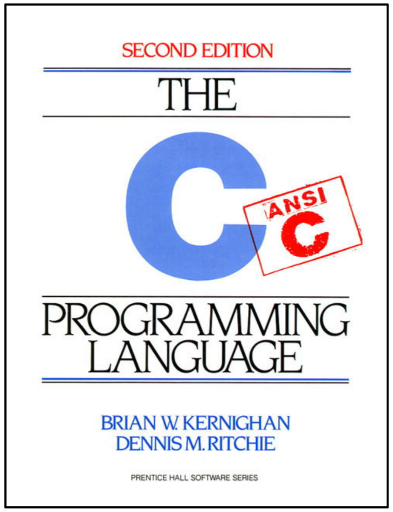
- Gadgets reduce focus and learning
  - Bursts of info (*e.g.* emails, IMs, etc.) are *addictive*
  - Heavy multitaskers have more trouble focusing and shutting out irrelevant information
    - http://www.npr.org/2016/04/17/474525392/attention-students-put-your-laptops-away
  - Seriously, you will learn more if you use **paper** instead!!!

- Non-disruptive use okay
  - NO audio allowed (mute phones & computers)
  - Stick to side and back seats
  - Stop/move if asked by fellow student

# Lecture Outline

- Course Introduction
- Course Policies
  - https://courses.cs.washington.edu/courses/cse333/19sp/syllabus/
- **C Intro**
  - **Workflow, Variables, Functions**

# C




- Created in 1972 by Dennis Ritchie
  - Designed for creating system software
  - Portable across machine architectures
  - Most recently updated 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)
  - "Weakly-typed" or "type-unsafe"
  - Small, basic library compared to Java, C++, most others….

# 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`

Advantages: ① easy – keyboard chars passed as chars
② flexible – any number

Disadvantages: ① input checking – prevent user misuse (usage messages)
② data conversion – if not intended to be chars

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

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

instead of: int main()

char* argv[] same as char** argv

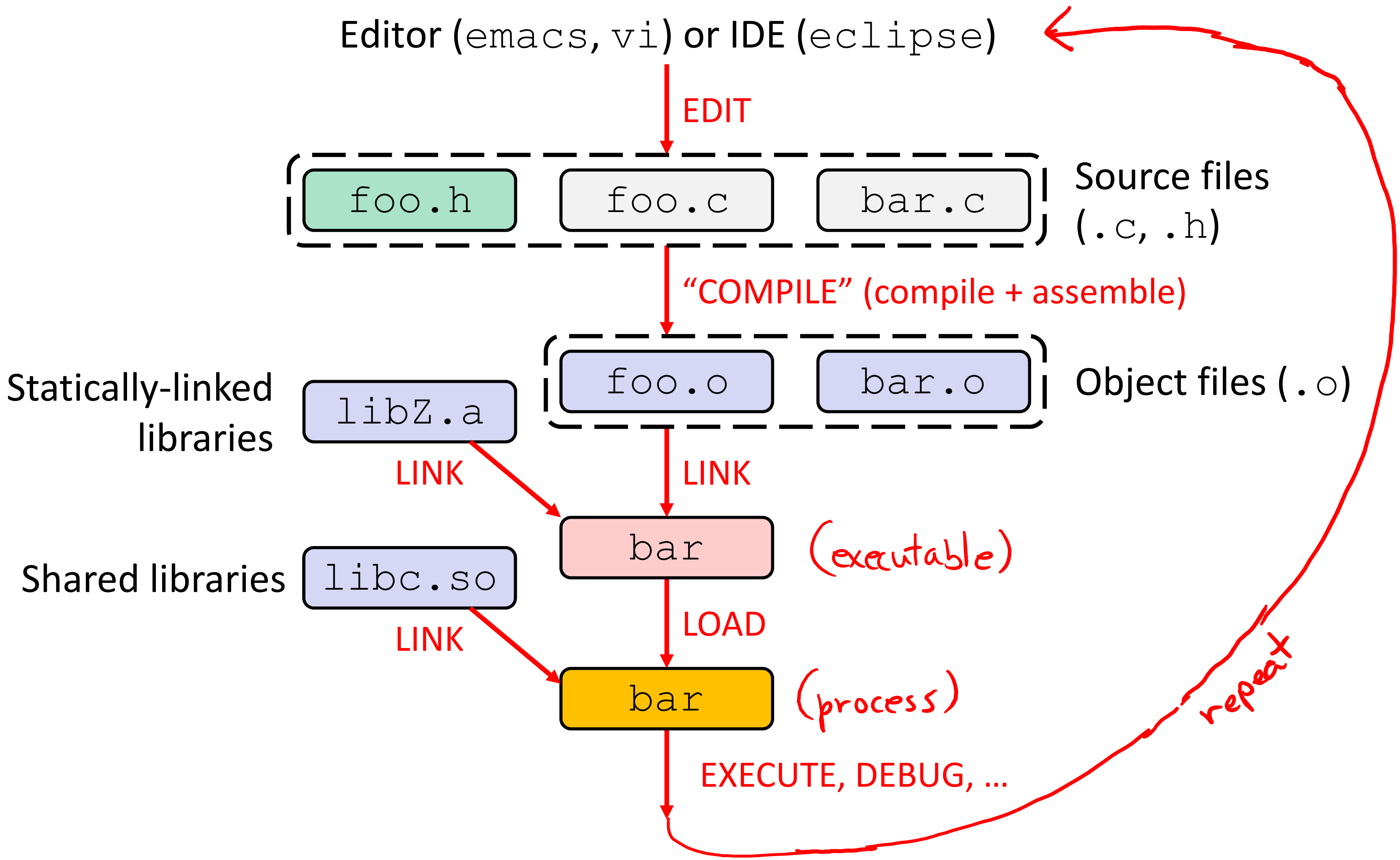
- What does this mean?
  - `argc` contains the number of strings on the command line (the executable name counts as one, plus one for each argument).
  - `argv` is an array containing *pointers* to the arguments as strings (more on pointers later)

    needed because C doesn't track array lengths!

- Example: `$ foo hello 87`

  string or number?
  - `argc = 3`
  - `argv[0]="foo", argv[1]="hello", argv[2]="87"`

# C Workflow


Editor (emacs, vi) or IDE (eclipse)
EDIT
foo.h
foo.c
bar.c
Source files (.c, .h)
"COMPILE" (compile + assemble)
foo.o
bar.o
Object files (.o)
Statically-linked libraries
libZ.a
LINK
LINK
bar
(executable)
Shared libraries
libc.so
LINK
LOAD
bar
(process)
EXECUTE, DEBUG, …
repeat

# C to Machine Code

```
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)

```
400575: 01 fe
        89 32
        c3
```

Machine code (sumstore.o)

C compiler (gcc –c)

# When Things Go South…

- 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
- 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')

(these are not exhaustive)

| Language Feature | S/D | Differences in C |
|---|---|---|
| Control structures | S | no boolean → 0 is false, else is true; goto (don't use) |
| Primitive datatypes | S/D | yes pointers, no String, yes unsigned; different data widths (e.g. char) |
| Operators | S | Java has >>>; C has -> |
| Casting | D | C has no casting restrictions |
| Arrays | D | C has no length or bounds checking |
| Memory management | D | no garbage collection; explicit requests: malloc / free |

# Primitive Types in C

- Integer types
  - `char`, `int`
- Floating point
  - `float`, `double`
- Modifiers
  - `short` [int]
  - `long` [int, double]
  - `signed` [char, int]
  - `unsigned` [char, int]

| C Data Type | 32-bit | 64-bit | `printf` |
|---|---|---|---|
| **`char`** | 1 | 1 | `%c` |
| `short int` | 2 | 2 | `%hd` |
| `unsigned short int` | 2 | 2 | `%hu` |
| **`int`** | 4 | 4 | `%d` / `%i` |
| `unsigned int` | 4 | 4 | `%u` |
| `long int` | 4 | 8 | `%ld` |
| `long long int` | 8 | 8 | `%lld` |
| **`float`** | 4 | 4 | `%f` |
| **`double`** | 8 | 8 | `%lf` |
| `long double` | 12 | 16 | `%Lf` |
| **`pointer`** | 4 | 8 | `%p` |

Typical sizes – see `sizeofs.c`

# C99 Extended Integer Types

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

```
#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
  ...
}
```

fine for generic C code

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

needed for "system" code – please use in 333

```
void sumstore(int32_t x, int32_t y, int32_t* dest) {
```

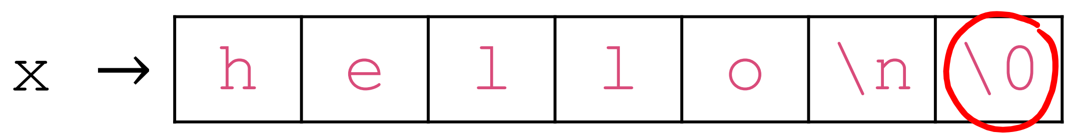
# Basic Data Structures

- 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:

```
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 compiler goes line-by-line:

sum_badorder.c

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

fix

```
#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;
}
```

??? (sumTo)

← defined here

# Solution 1: Reverse Ordering

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

sum_betterorder.c

```
#include <stdio.h>

// 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 0;
}
```

# Solution 2: Function Declaration

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

sum_declared.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;
}
```

parameter names optional

declared here

has seen already

defined here

# 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

# Multi-file C Programs

C source file 1 (sumstore.c)

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

← defined here

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;
}
```

optional (parameter names x, y, dest)
← declared here
← used here

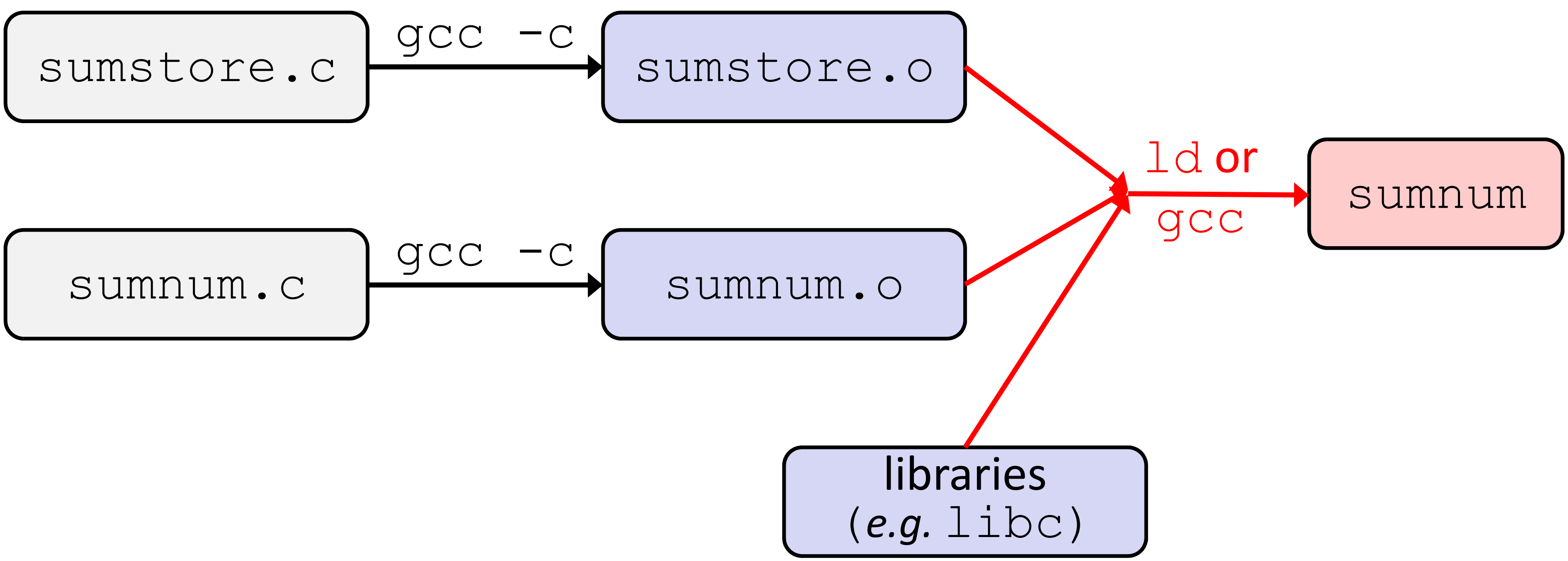
Compile together:

```
$ gcc -o sumnum sumnum.c sumstore.c
```

both files included during compilation

# 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



# Review Question

- Which of the following statements is FALSE?
  - Vote at http://PollEv.com/justinh

A. With the standard `main()` syntax, It is always safe to use `argv[0]`. ← will be the name of the executable

B. We can't use `uint64_t` on a 32-bit machine because there isn't a C integer primitive of that length. there is → long long int

C. Using function declarations is beneficial to both single- and multi-file C programs.
single : flexible ordering of functions
multi : use definitions in other files

D. When compiling multi-file programs, not all linking is done by the Linker.
Loader does some linking (shared libraries)

E. We're lost…

# To-do List

- Make sure you're registered on Canvas, Piazza, 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, attu, or CSE Linux VM
- Exercise 0 is due 11 am on Wednesday
  - Find exercise spec on website, submit via Gradescope
    - Course "CSE 333 Spring 19", Assignment "ex0 - Exercise 0", then drag-n-drop file(s)! Ignore any messages about autograding.
  - Sample solution will be posted Wednesday afternoon