Intro, C refresher

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Justin Tysdal

Sayuj Shahi

Nicholas Batchelder

Leanna Mi Nguyen

Lecture Outline

- **Course Introduction**
- Course Policies
  - https://courses.cs.washington.edu/courses/cse333/24su/syllabus.html
- C Intro

- Staff
- Students
- Code Quality
- Topic

Introductions: Course Staff

- Instructor: Alex Sanchez-Stern (asnchstr@cs)

- 5 TAs:
  - Justin Tysdal, Sayuj Shahi, Nicholas Batchelder, and Leanna Mi Nguyen
  - 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!

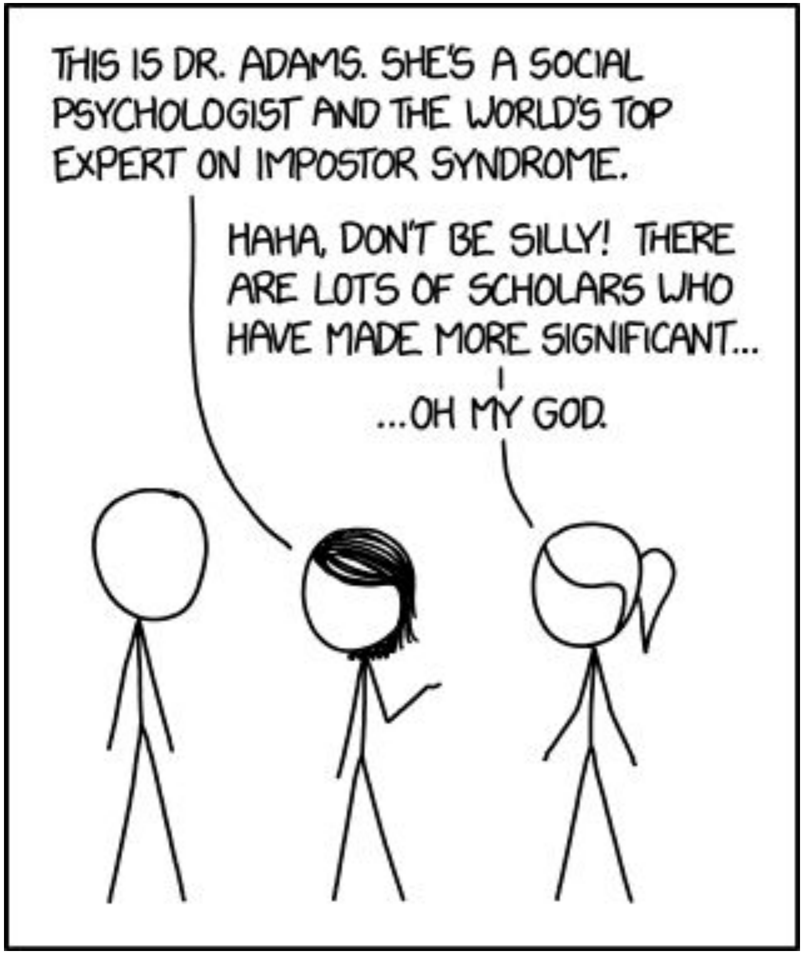
Introductions: Students

- ~75 students this quarter

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

Introductions: Students

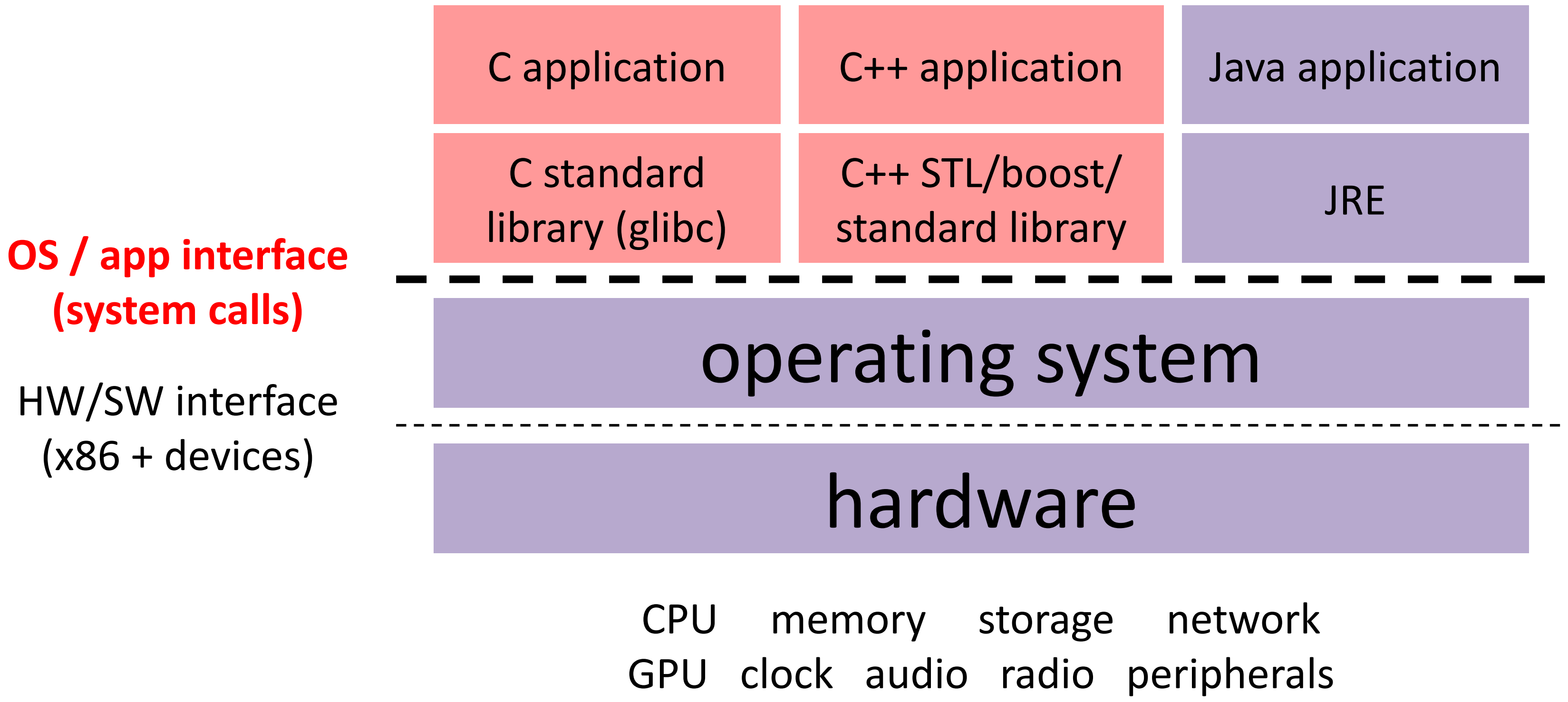
- ~70 students this quarter
  - Easier to feel lost, as if everyone is "better" than you
- "Nearly 70% of individuals will experience signs and symptoms of impostor phenomenon at least once in their life."
  - https://en.wikipedia.org/wiki/Impostor_syndrome



Code Quality

- Good code quality will help you in the long run
  - Systems code is complex!
  - Complexity is tamed by good habits and good abstractions
  - Easy to understand code now will help you later.
- So use these:
  - Coding style conventions
  - Unit testing, code coverage testing, regression testing
  - Documentation (code comments, design docs)
  - Code reviews
- Learning to writing clean code is a lifelong process

Course Map: 100,000 foot view



Systems Programming

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

Lecture Outline

- Course Introduction
- **Course Policies**
  - https://courses.cs.washington.edu/courses/cse333/24su/syllabus.html
- C Intro

This is Only an Overview!

- This is just the summary/highlights
  - … but you must read the full details online!

https://courses.cs.washington.edu/courses/cse333/24su/syllabus.html

- Course Components
- Grading
- Deadlines and Student Conduct
- Communication

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
- Final exam and midterm
  - Goal is to revisit and internalize concepts

Course Components

- Programming Exercises (~18)
  - Roughly one per lecture, due the morning before the next lecture
  - Coarse-grained grading (check plus/check/check minus = 0, 1, 2, or 3)
- Programming Projects (0+4)
  - Warmup, then 4 “homeworks” that build on each other
  - Individual work
- Lecture Activities (huge variance but can assume >50)
  - In-class polls graded on *completion* not *correctness*

Grading

- **Exercises:** ~30%
  - Submitted via Gradescope
  - Evaluated on correctness and code quality; drop the lowest score
- **Homeworks:** ~30%
  - Submitted via GitLab; must tag commit that you want graded
  - "Does it work?" and code quality both matter, roughly equally
  - Binaries provided if you didn't get previous part working or prefer to start with a known good solution to previous parts
- **Lecture Activities:** ~15%
- **Midterm**: ~10%
- **Final:** ~15%

Deadlines and Student Conduct

- Late policies
  - Exercises: no late submissions accepted, due 10 am before class
  - Projects: 4 late days for entire quarter, max 2 per project
  - Need to get things done on time – difficult to catch up!
    - But we will work with you if unusual circumstances / problems

- Academic Integrity (**read** the full policy on the web)
  - 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

Gadgets

Please:

- No laptops in class unless you're taking notes
- The only app you should be using on your phone is PollEverywhere

Communication

- Website: http://cs.uw.edu/333
  - Schedule, policies, materials, assignments, etc.
- Office Hours: spread throughout the week
  - Schedule posted shortly and will start as soon as we can
- One-on-ones: by appointment
  - Send us a message with your availability in the next 3 days
  - Do **not** expect a response in less than 24 hours!

Communication

- Messages to staff: things unsuitable for Ed chat or Gradescope regrade requests
  - Please send email to cse333-staff@cs.uw.edu. Reaches all staff so the right person can help out quickly, and helps follow up until resolved
  - (***don't*** email to instructor or individual TAs if possible – we can get quick answers for you and coordinate better if it goes to the staff
- Discussion: Ed group linked to course home page
  - Ask and answer questions – staff will monitor and contribute
  - Use private messages for questions about detailed code, etc.
- Announcements: will use broadcast Ed messages to send "things everyone must read and know"

Starting…. NOW!

- First exercise out today, due Thursday morning **10am** before class
- HW0 (the warmup project) published wednesday,
  due next Monday

- Goal is to figure out setup and computing infrastructure right away so we don't put that off and then have a crunch later in the quarter

- Logistics for larger projects explained in sections Thursday
  - It's okay to ignore the homework details until section on Thursday, but try to start the setup
  - Bring a laptop to sections! We may have time to go through some of the initial configuration parts for hw0.

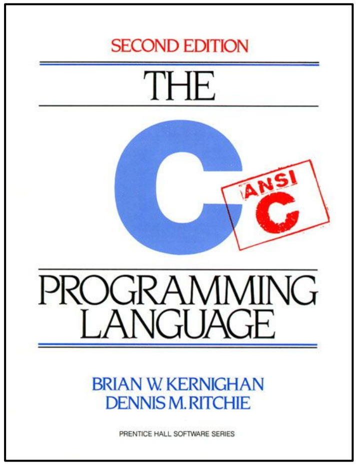
Deep Breath….

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

Lecture Outline

- Course Introduction
- Course Policies
  - https://courses.cs.washington.edu/courses/cse333/24su/syllabus.html
- **C Intro**
  - **Workflow, Variables, Functions**

C



- Created in 1972 by Dennis Ritchie
  - Designed for creating system software
  - Portable across machine architectures
  - More recently updated in 1999 (C99) and 2011 (C11) and 2017 (C17)
- 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 (often necessary to bypass the type system)
  - Small standard 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 & stru

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

/* define other functions */
```

We'll cover this stuff late next week

We'll cover this stuff today

C Syntax: `main`

- All programs start with `main`:

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

- What do the arguments 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 arrays and pointers later).
- Example: `$ ./foo hello 87`
  - `argc = 3`
  - `argv[0]="./foo", argv[1]="hello", argv[2]="87"`

When Things Go Wrong…

- Processes return an "exit code" when they terminate
  - Can be read and used by parent process (shell or other)
    - In main: return EXIT_SUCCESS; or return EXIT_FAILURE; (e.g., 0 or 1)
- In C, functions do the same!
  - C does not have exception handling (no `try`/`catch`)
  - Errors are returned as integer error codes from functions
  - Because of this, it's easy to miss an important error
- 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')

Language Feature	S/D	Differences in C
Control structures	S	
Primitive datatypes	S/D	Similar but sizes can differ (char, esp.), unsigned, no boolean, uninitialized data, …
Operators	S	Java has >>>, C has ->
Casting	D	Java enforces type safety, C does not
Arrays	D	Not objects, don't know their own length, no bounds checking
Memory management	D	Manual (malloc/free), no garbage collection

Primitive Types in C

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

No standard size!
Can depend on architecture, compiler, etc.

Size technically also unspecified, but pretty much always the same

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
  ...
```

Use extended types in most cse333 code

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

But `int` is usually fine for simple counters

```
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";
```

x	h	e	l	l	o	\n	\0

- **Structs** are the most object-like feature, but are just collections of fields – no "methods" or functions
  - (but can contain pointers to 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
- This is because C compilers used to be single-pass

sum_badorder.c

```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?)

sum_betterorder.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, and call each other without restriction

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;
}
```

Code examples from slides are on the course web for you to experiment with!

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 defined elsewhere
  - *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 the thing
    - Should appear before first use

Multi-file C Programs

C source file 1 (sumstore.c)

definition

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

C source file 2 (sumnum.c)

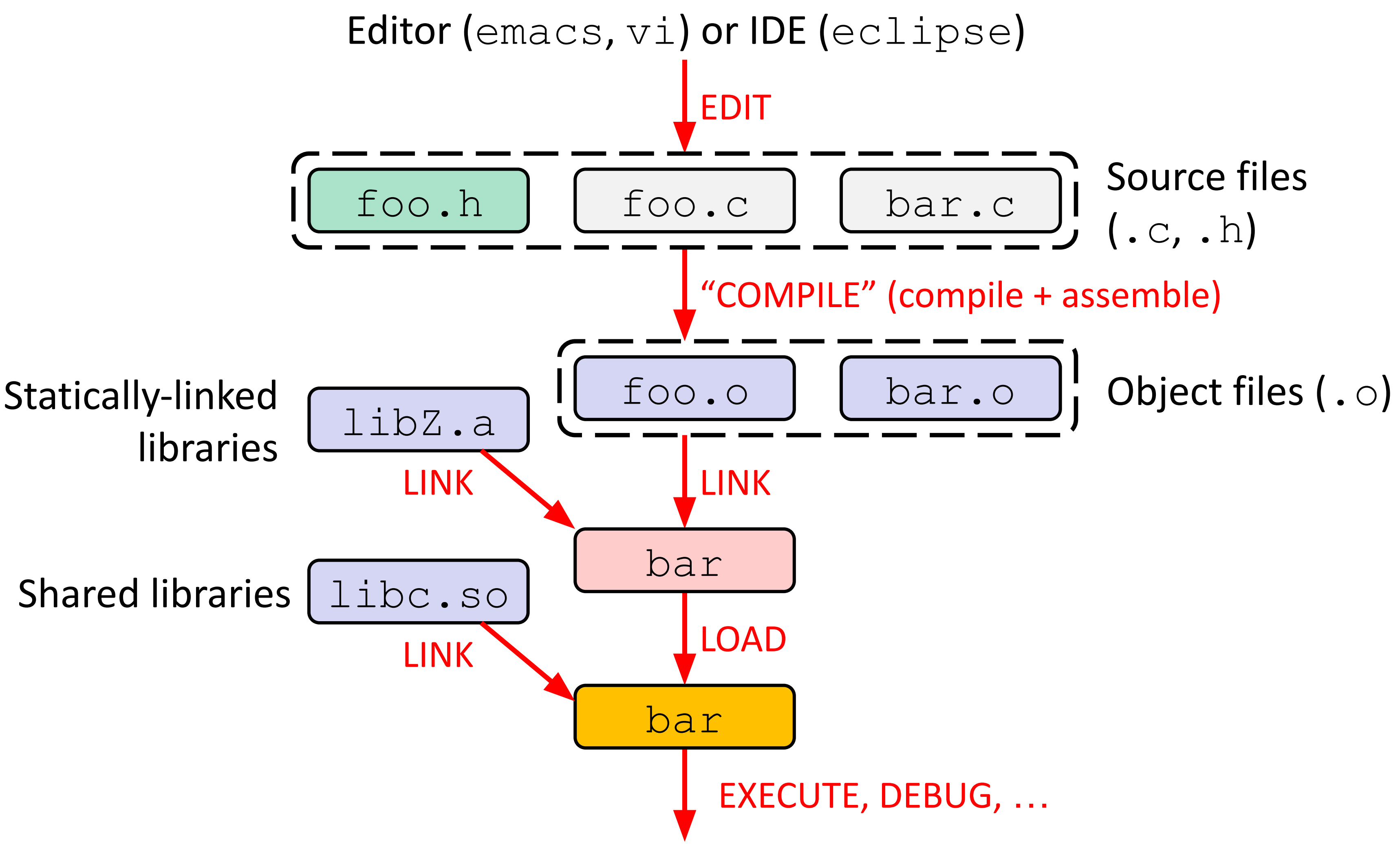
```
#include <stdio.h>

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;
}
```

declaration

Compile together:

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

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
Shared libraries
libc.so
LINK
LOAD
bar
EXECUTE, DEBUG, …

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)

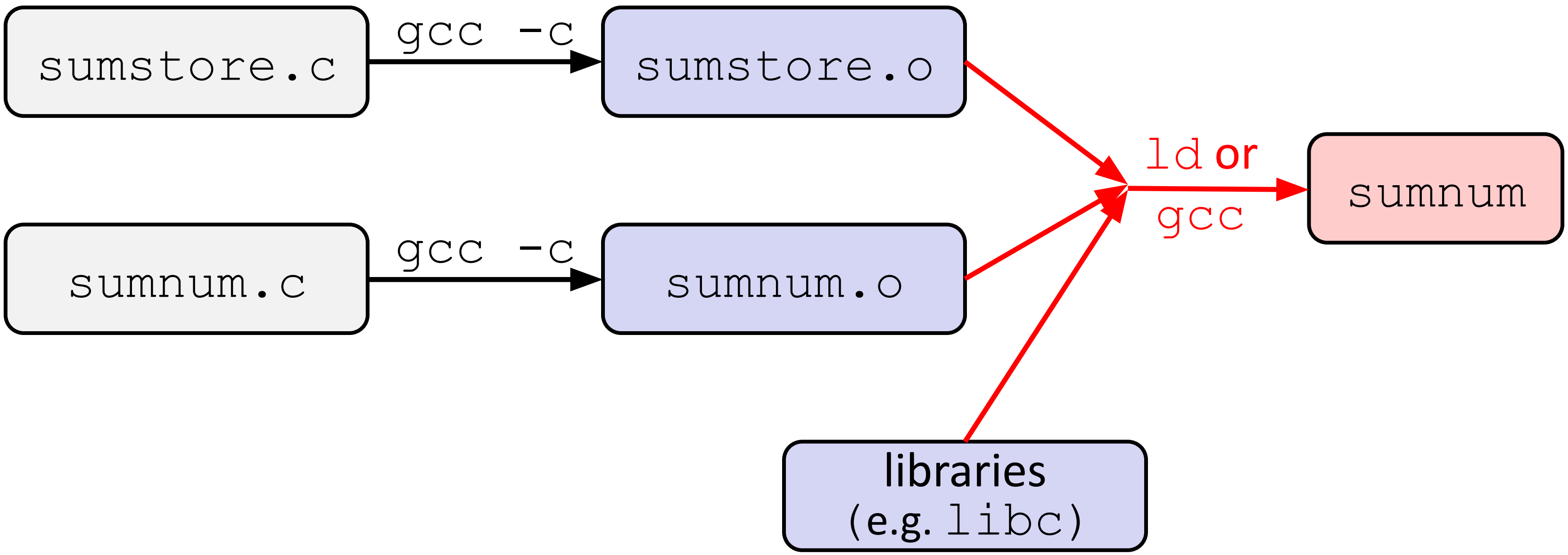
```
400575: 01 fe
        89 32
        c3
```

Machine code (sumstore.o)

C compiler (gcc -c)

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



To-do List

- Explore the website *thoroughly*: http://cs.uw.edu/333

- Computer setup: CSE labs, attu, or CSE Linux VM

- Exercise 0 is due 10 am **sharp** on Thursday
  - Find exercise spec on website, submit via Gradescope
  - Sample solution will be posted later that day
  - Give it your best shot

- Project repos created and hw0 out Wednesday
  - Ask questions on Ed!
  - More questions? Bring them (and your laptop) to section