Intro, C refresher

CSE 333 Winter 2021

Welcome – please set up your Zoom session. We'll start the actual class meeting at 11:30 am pdt

Instructor: John Zahorjan

Teaching Assistants:

Matthew Arnold
Nonthakit Chaiwong
Jacob Cohen
Elizabeth Haker
Henry Hung
Chase Lee
Leo Liao
Tim Mandzyuk
Benjamin Shmidt
Guramrit Singh

Lecture Outline

- **Course Introduction**
- Course Policies
  - https://cs.washington.edu/333 / Syllabus tab
- Course Map
- C Intro

To get started…

- It's all remote, all the time this quarter
- Core infrastructure is same as always (Gradescope, Gitlab, web, discussion board)
- Lectures, sections, office hours – Zoom
- Stay healthy in every way

Lectures

- Classes will be mostly lectures – more interaction in sections
  - Varied experiences so far. Let us know where we could do better!
- Conventions (from page on our web site)
  - Lecture will be recorded and available to class members (only)
  - Recordings are kept only a short while past the end of the quarter
  - If you have a question, (a) thank you!, and (b) type it in the chat window or speak up
  - We intend to post lecture slides in advance

Online 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; details TBA
    - Those will be available online
  - Slides and any sample code, worksheets, etc. posted as always

Online Everything Else

- Office hours: also Zoom; combination of group gatherings, breakouts, waiting rooms, sign-up sheets to organize – all as needed
  - 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
- You will be bombarded with email as we add these things to Canvas/Zoom. Feel free to file away for future reference. ☺

Stay in Touch – Speak up

- This is a strange world we're in and there's a lot of stress for many people
- Please speak up if things aren't (or are!) going well
  - We can often help if we know about things, so stay in touch with TAs, instructor, advising, friends and peers, others
- We're all in this together but not all in the same way, so please show understanding and help us understand

Introductions: Course Staff

- John Zahorjan (instructor)
  - Long-time CSE faculty member and CSE 333 veteran
- TAs:
  - Matthew Arnold, Nonthakit Chaiwong, Jacob Cohen, Elizabeth Haker, Henry Hung, Chase Lee, Leo Liao, Tim Mandzyuk, Benjamin Shmidt, Guramrit Singh
  - 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

- ~135 students this quarter
  - There are no overload forms or waiting lists for CSE courses
- Time zones?
- Expected background
  - **Prereq:** CSE 351 – a little bit of hardware architecture (registers, memory, instructions), a little bit of C (notion of types, pointers, procedure call convention, compile/link build)
  - CSE 391 or Linux skills needed for CSE 351 helpful, but not having that isn't a show stopper

Lecture Outline

- Course Introduction
- **Course Policies**
  - https://courses.cs.washington.edu/courses/cse333/21wi/syllabus/
  - Summary here, but you ***must*** read the full details online
- Course Map
- C Intro

Communication

- Website: 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** Google identity (not cse)
  - Ask and answer questions – staff will monitor and contribute
- Staff mailing list: cse333-staff@cs for things not appropriate for Ed group
  - Mail is sent to instructor and all TAs
- Course mailing list: for announcements from staff
  - Registered students automatically subscribed with your @uw email
- Office Hours: spread throughout the week
  - Schedule posted shortly and will start right away
  - Can also e-mail to staff list to make individual appointments
  - Will try to consider time zones when scheduling

Course Components

- Lectures (28)
  - Introduce the concepts
- Sections (10)
  - Applied concepts, important tools and skills for assignments, clarification of lectures, exam review and preparation
- Programming Exercises (~17)
  - 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
- No traditional exams, but hoping to do ~4 "recap/review" assignments for things traditionally covered on exams

Grading (tentative)

- **Exercises:** ~35%
  - Submitted via GradeScope (account info mailed this morning)
  - Graded on correctness and beauty by TAs
- **Projects:** ~45% total
  - Submitted via GitLab; must tag commit that you want graded
  - Binaries provided if you didn't get previous part working

- Note the lack of quiz and exam points

Deadlines and Student Conduct

- Official late policy
  - Exercises: no late submissions accepted, due 10 am
  - Projects: 4 late days for entire quarter, max 2 per project
  - Need to get things done on time – difficult to catch up!
    - But given remote world, we'll work with you if things come 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

And off we go…

- 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
- So:
  - First exercise out today, due Friday morning **10 am**
  - **First homework out Wednesday, due next Monday**
  - Warmup/logistics in sections Thursday
    - HW0 (the warmup project) published this afternoon and gitlab repo's created.

Okay, but…

- CSE 333 has a tightly integrated set of projects, exercises, and lectures
- Okay, but…
- I think things are hard enough, and I'm naturally inclined to think that having to worry about the details makes it harder to reflect on anything larger
- So… the mantra for this quarter is "flexibility"
  - Grading
  - Due dates
  - Programming style rules

Something old

- Same homeworks…
  - Students who have done them are generally enthusiastic about them
- Mainly most same exercises
  - This is just fear – I don't know at this point how tightly integrated the exercises, lectures, and homeworks are
  - I endorse the idea of frequent, small exercises, though
- This quarters slides will be derived from "the standard slides for CSE 333"

Something new

- Today's class isn't exactly what usually happens
- My intention is for this class to be "gentle"
  - Carrots
- My intention is to help you learn more than you would if you were left to learn this material on your own
- I've monkeyed with hw0 / ex00
- I've made some policy changes
  - Meet A TA sessions
  - Homeworks
    - Homeworks 0 – 2 (in C) done individually
    - Homework 3 (C++) done in pairs
    - Home 4 (C++) done in different pairs
    - We suggest "pair programming"

Meet A TA Session

- Brief…
- Your part starts with:

  “"Hi, _____, my name is _____. I'm a fresh/soph/junior/senior majoring in _____. My favorite CSE course so far was _____. The thing I liked about it was _____. My favorite course ever, anywhere, was _____. The thing I liked about it was _____. I'm [really/kind of] looking forward to _____ in CSE 333 this quarter. I'm worried about _____ in CSE 333 this quarter. [*Optional: Something I'd like you to know about me is* _____.] I have this question: _____.”

Team Programming

- Traditionally, all work done individually
  - All the infrastructure seems to be affected by that
- We're going where no CSE 333 has gone before
  - Who knows what can go wrong!
- *Aside: I've changed the version of C/C++ we're using from c11/c++11 to c17/c++17, but it turns out there's a course tool that might depend on c11/c++11…*
- This quarter we'll do some projects individually, some in pairs
- Why?
  - Programming alone is an anachronism
  - Every classmate knows something you don't, and vice versa
  - C is pretty tame – questions tend to be simpler, and web searches more effective
  - C++ is a monster – I think it's very likely having someone to talk over details of some implementation issue will be good for almost everyone

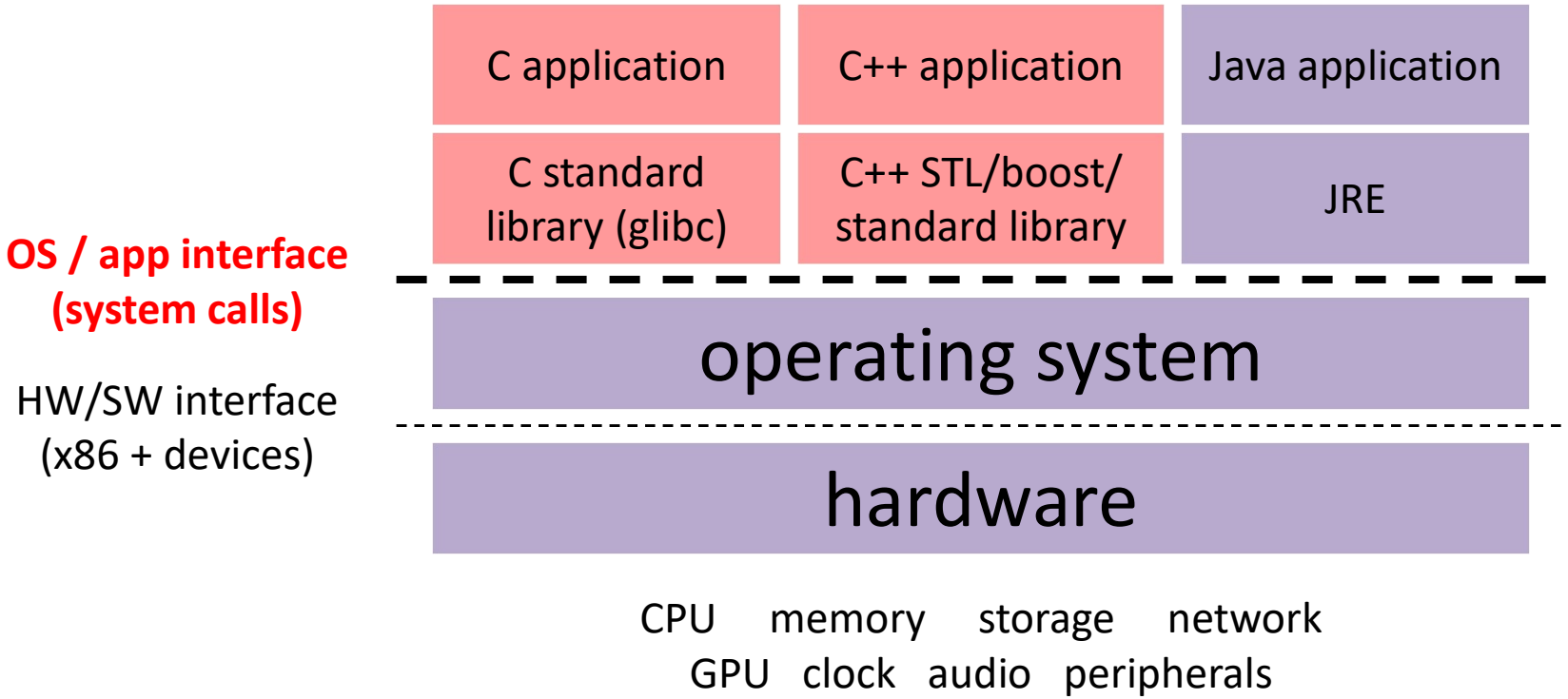
Changing Teams Mid-Stream

- For all we know, this could be a great idea…
- But homeworks are cumulative…
- Your goal:
  - Be a teammate everyone would want to work with
  - Learn something
- Not your goal:
  - Producing the finest project cse333 has ever seen
- If you're capable of doing a better implementation on your own than your team can produce, then set your goal on figuring out how to make it so you aren't
- Grades…
  - Effort counts

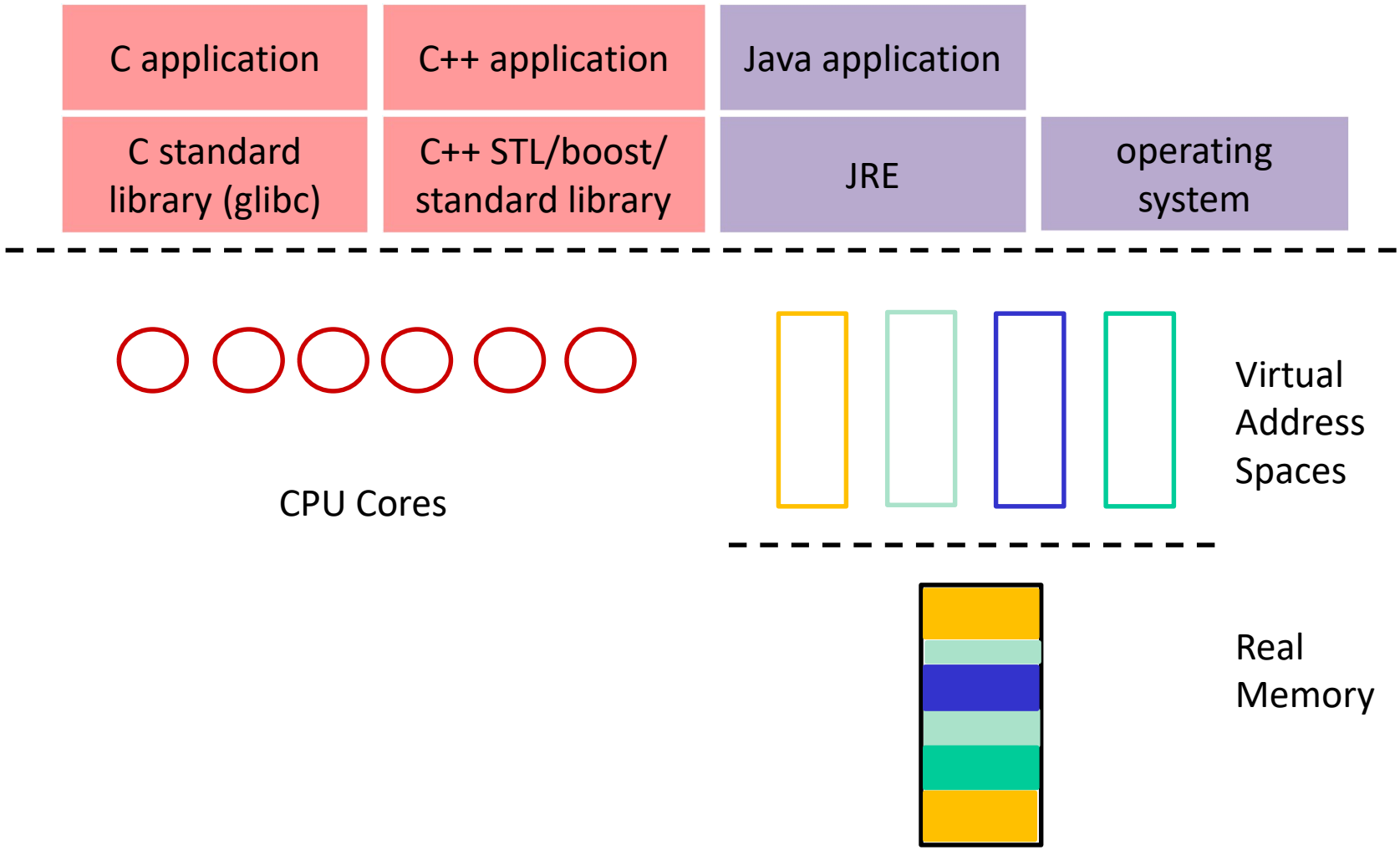
Lecture Outline

- Course Introduction
- Course Policies
  - https://cs.washington.edu/333 / Syllabus tab
- **Course Map**
- C Intro

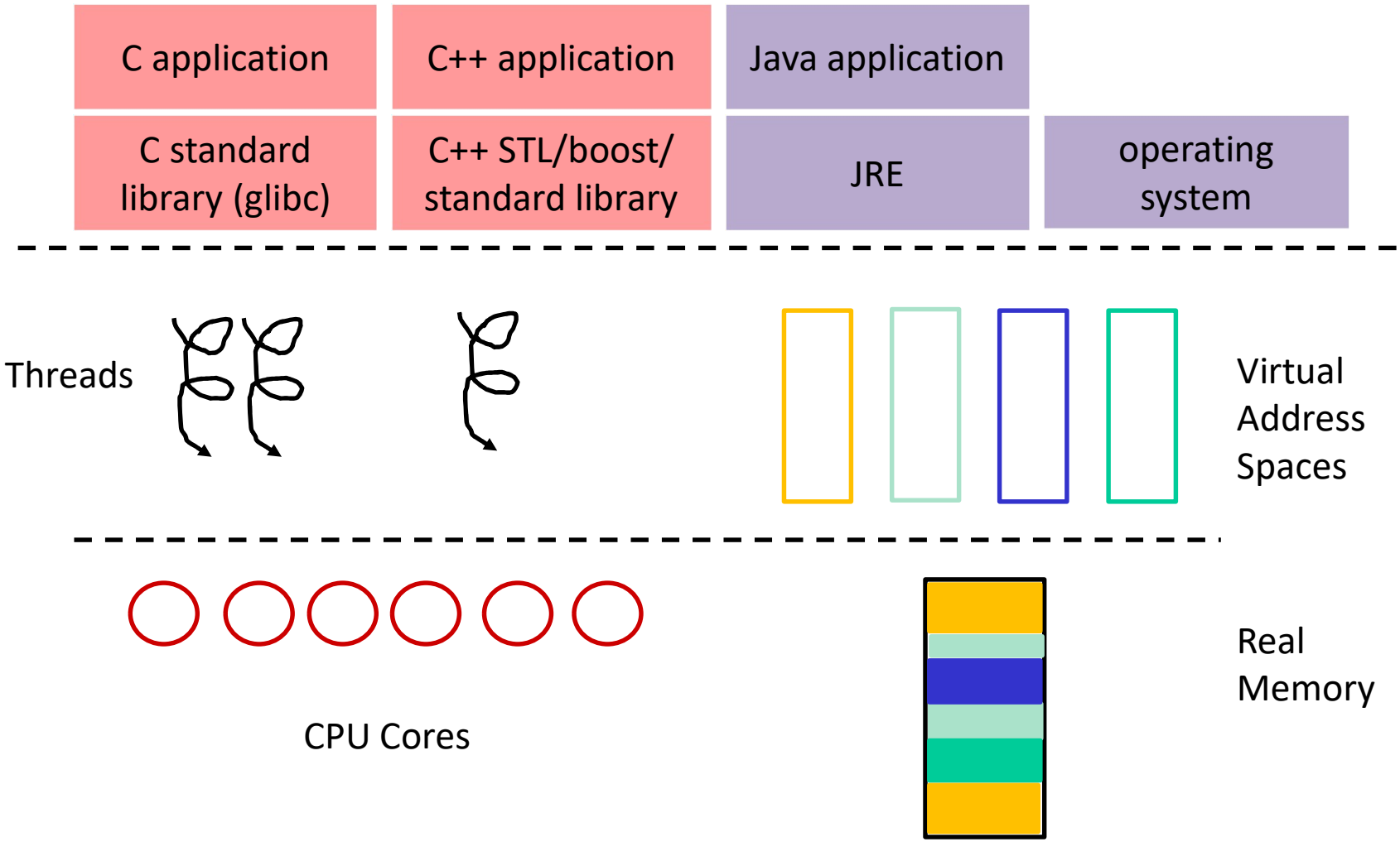
Course Map: 100,000 foot view



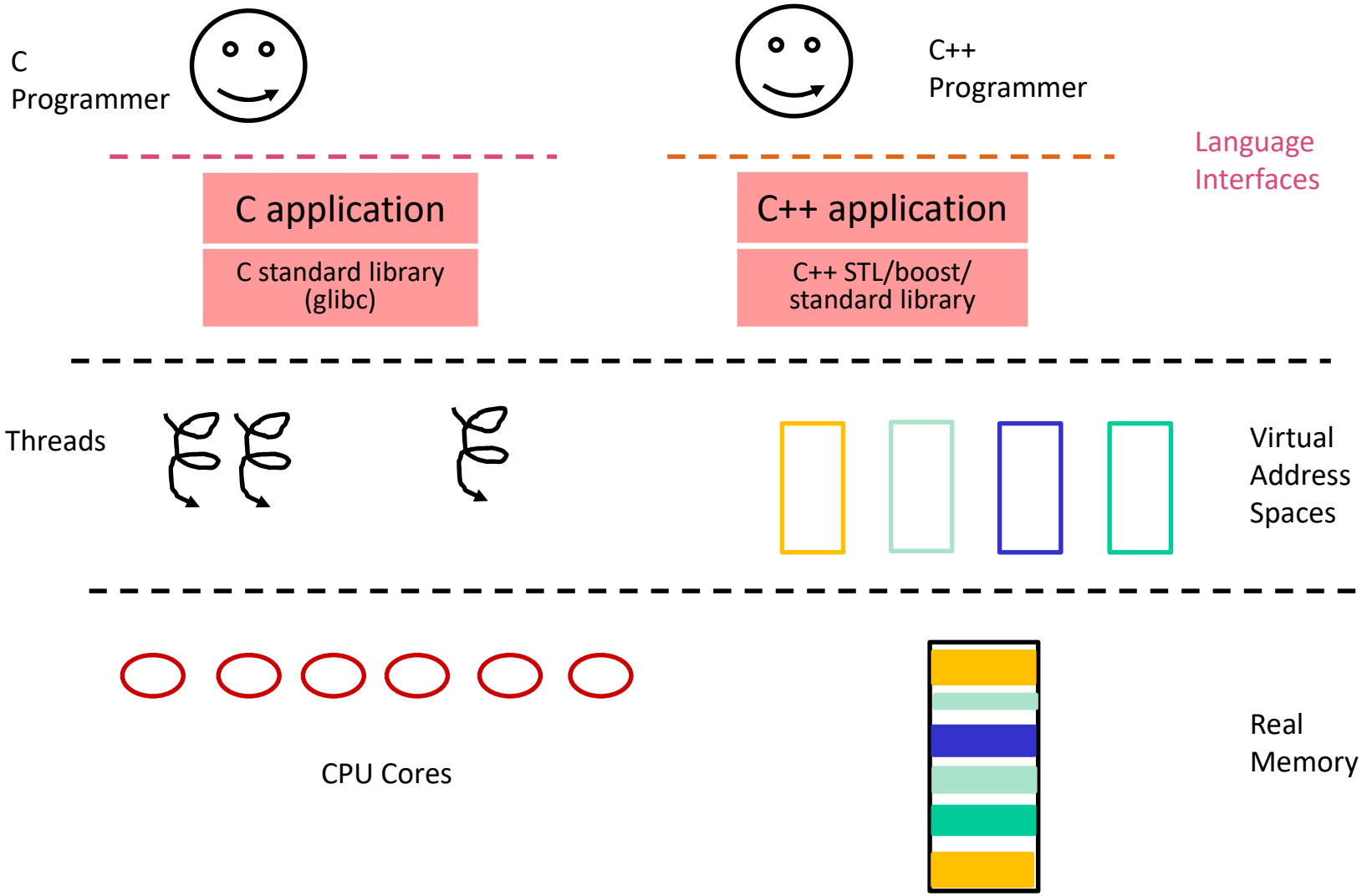
Course Map Picture Revisited I



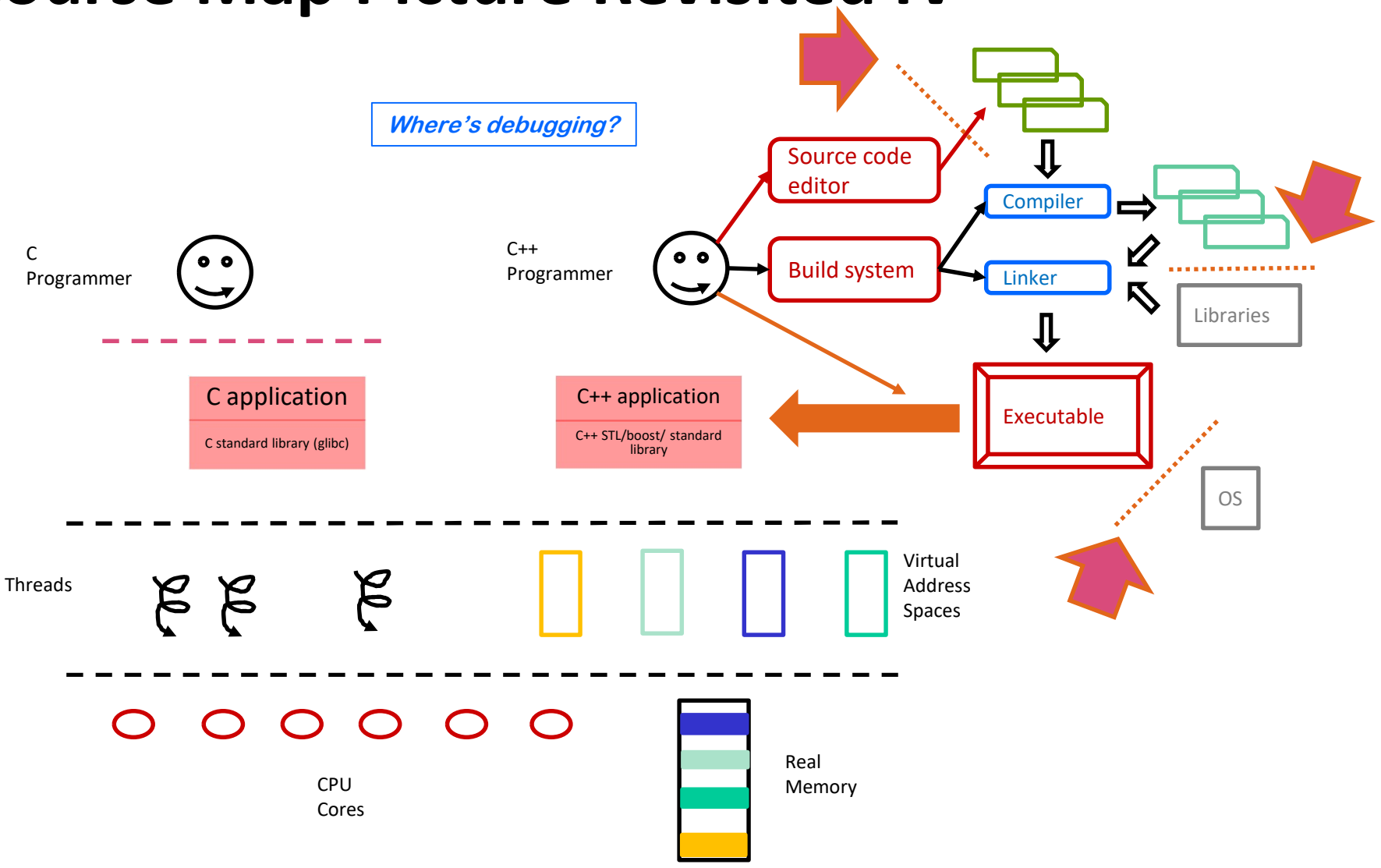
Course Map Picture Revisited II



Course Map Picture Revisited III



Course Map Picture Revisited IV



CSE 333 21wi Tools

Static

- OS
  - CentOS 8.2 (attu.cs.washington.edu)
- Compilers / libraries
  - gcc 9.2.1 / g++ 9.2.1
- Build
  - GNU make 4.2.1
- Language versions
  - C 2017 / C++ 2017
- Editors
  - vscode
  - vim
  - emacs
- Source Control
  - git / gitlab

Dynamic

- Debugger
  - gdb
- Unit Test
  - Gtest
- Other Tests
  - valgrind

Systems Programming

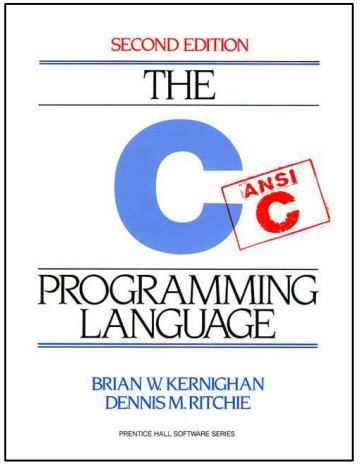
- "Systems"
  - Not an application
  - Or perhaps an application that facilitates building other applications

- "Programming"
  - In CSE 333, in C/C++ -- systems languages
  - In CSE 333, "programming" includes the tools/procedures to go from source code to debugged execution
    - The tools are often language specific
    - Many of the concepts are not

Lecture Outline

- Course Introduction
- Course Policies
  - https://courses.cs.washington.edu/courses/cse333/18sp/syllabus/
- Course Map
- **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) and 2017 (C17)
    - Currently working on C 2x
- 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….

Understanding C

- Assembler gives access to "everything" on the processor
- Assembler programs mainly involve:
  - Deciding how to use memory
    - How much memory is required to store a value
    - What values live where
    - How does program address those values at run time
    - Example: a logical array of values stored in consecutive memory locations
  - Deciding what values to cache in registers, when
  - Writing instructions that operate on values
    - Example: expression evaluation
  - Implementing control flow
    - Examples: loops; procedure call and return

- Don't think of assembler/machine code as how C progams are realized, think of C as a more convenient way to express an assembler program

What Does the C Compiler Do?

- Certain tasks the assembler programmer would have to do can be automated
  - Names, not addresses
    - myInt, not 0(eax)
  - Memory size
    - How much space is needed for an int? a float? a char?
  - Memory layout
    - If a procedure has three integer local variables, what are their offsets from the frame pointer?
  - Control flow instructions
    - for/while loops
    - Procedure call/return
  - Expression evaluation
    - Example: 7+x*(y-sub(z))

Compiling C vs. Assembling Assembly

- It seems natural to think of programs as simply a specification of an execution
  - That is, we make little mental distinction between the static code file and the execution of the program
- In assembler, nothing very interesting happens at assembly time
  - mov 0x0(%rip),%rax => 48 8b 05 00 00 00 00
- In contrast, what does the C compiler have to do with these?
  - z = "One" + (long int)"Two";
    y = x * Five – process(val);
    y /= x unless x==0; // Note: This isn't C… But why not?

C++ vs. C
C17 vs. C99

- In a sense, the "action" in these languages has to do with things that happen at compile time
  - For the most part, what can be done at run time hasn't changed much
  - The language helps you write what you mean succinctly
  - It helps you write robust, correct code
  - It helps keep you from writing incorrect code
  - It helps you write performant code
  - [You help it generate performant code]
- C++ example:
  - auto val = CreateValue(y); // Declaration of val. Compiler
    // determines correct type

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`

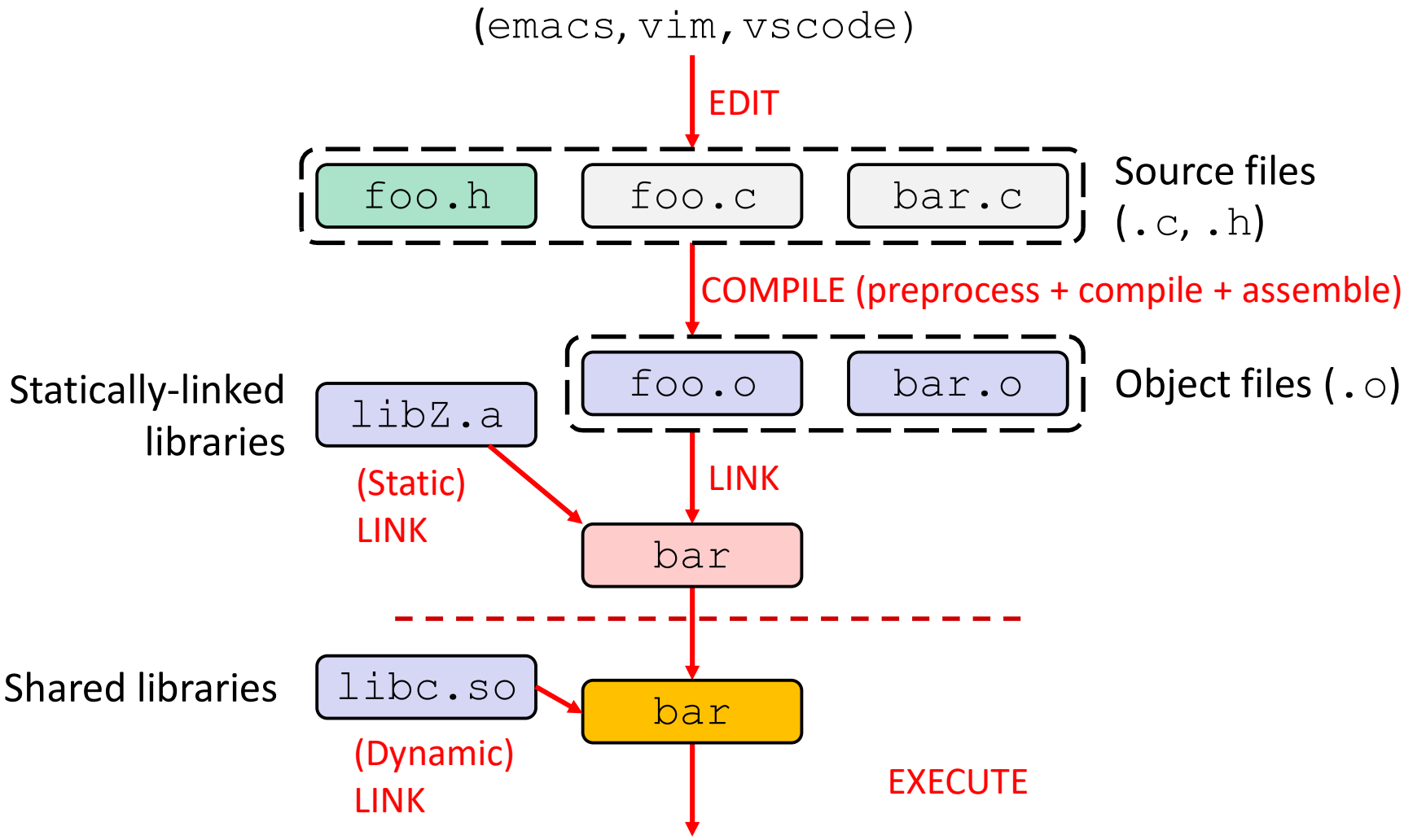
- To get command-line arguments in `main`, use:

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

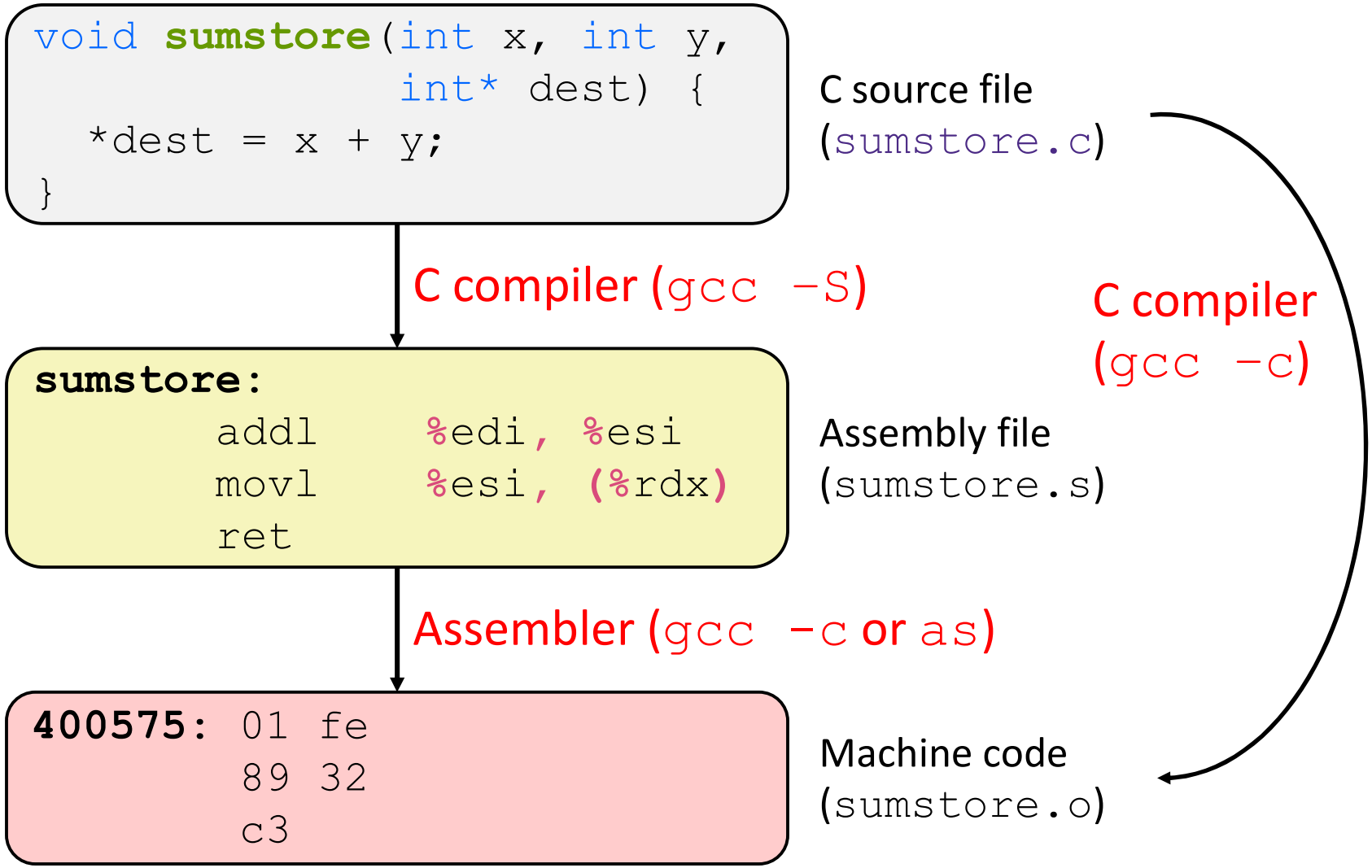
- What does this mean?
  - `argc` contains the number of strings on the command line (the executable name counts as one, plus one for each argument).
  - `argv` is an array containing *pointers* to the arguments as strings (more on pointers later)

- Example: `$ foo hello 87`
  - `argc = 3`
  - `argv[0]="foo", argv[1]="hello", argv[2]="87"`

C Workflow


(emacs, vim, vscode)
EDIT
foo.h
foo.c
bar.c
Source files (.c, .h)
COMPILE (preprocess + compile + assemble)
foo.o
bar.o
Object files (.o)
Statically-linked libraries
libZ.a
(Static) LINK
LINK
bar
Shared libraries
libc.so
(Dynamic) LINK
bar
EXECUTE

C to Machine Code



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')

Language Feature	S/D	Differences in C
Control structures		
Primitive datatypes		
Operators		
Casting		
Arrays		
Memory management		

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, 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
  ...
}
```

Why do we care how big an int is?

Use extended types in cse333 code

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

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

Basic Data Structures

- C does not support objects
  - C programs can follow a somewhat object oriented structure, though
- **Arrays** are contiguous chunks of memory
  - C has a complicated relationship with arrays
  - 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
  - Support assignment

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;
}
```

Why is this a terrible implementation?

Function Ordering

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

Why?

sum_badorder.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

- Frequently used solution; however, imposes ordering restriction on writing functions (who-calls-what?)
  - What if subA calls subB and subB calls subA?

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

sum_declared.c

Hint: code examples from slides are on the course web for you to experiment with

```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;
}
```

(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
  - (Mostly) must be **exactly one** definition of each thing (no duplicates)

- Declaration: description of a thing
  - It's the programmer giving the compiler just the information it needs to compile code, but not enough to create the thing being declared
  - *e.g.* function prototype, external variable declaration
    - **Often in header files and incorporated via `#include`**
      - "Must" also `#include` declaration in the code 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)

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

definition

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

Why is this a terrible way to do this?

Multi-file C Programs Revised

C decl file (sumstore.h)

```
#ifndef sumstore_h
void sumstore(int x, int y, int* dest;
#endif // sumstore.h
```

C source file 1 (sumstore.c)

```
#include "sumstore.h"
void sumstore(int x, int y, int* dest) {
  *dest = x + y;
}
```

C source file 2 (sumnum.c)

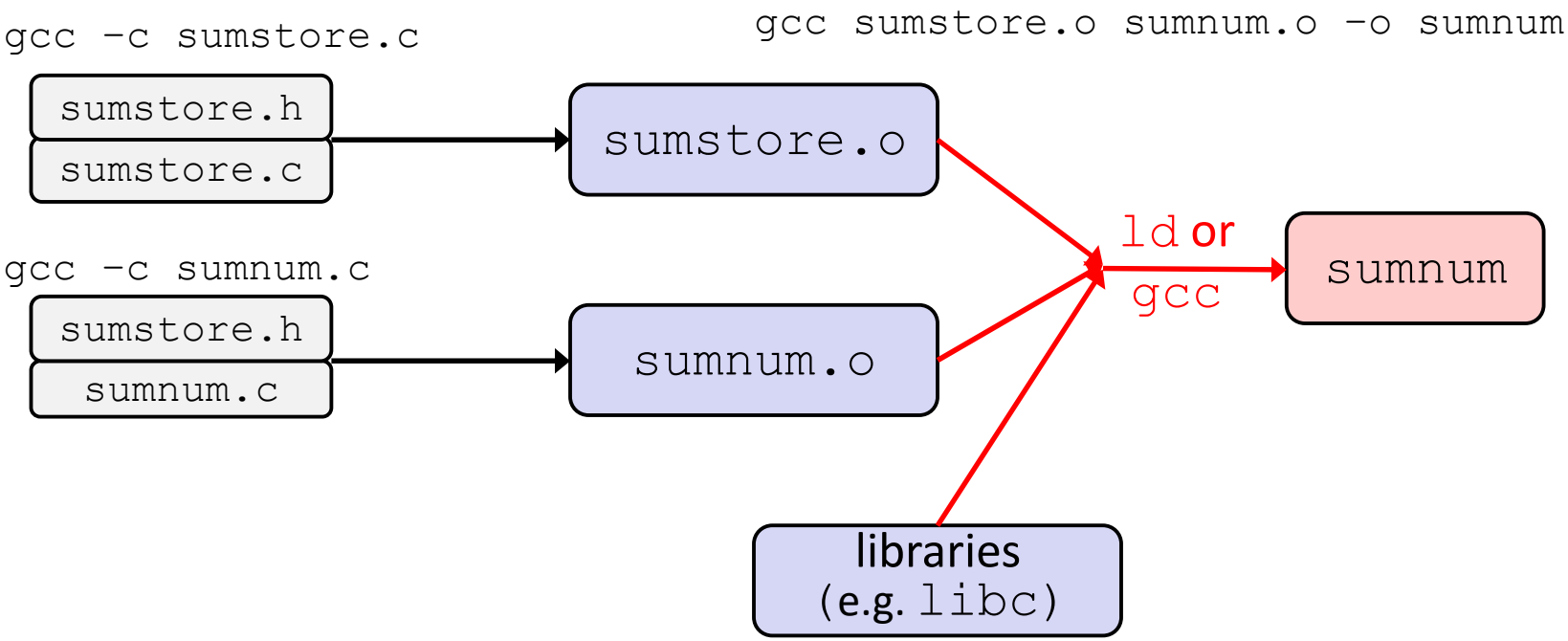
```
#include <stdio.h>
#include "sumstore.h"

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;
}
```

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 remote lab, attu, or CSE Linux VM
- Exercise 0 is due 10 am Friday 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 on time
- Gradescope accounts created just before class
  - Userid is your uw.edu email address
  - Exercise submission: find CSE 333 20au, click on the exercise, drag-n-drop file(s)! That's it!!
- Project repos created and hw0 out by tonight!!
  - All will become clear in sections tomorrow! ☺