Intro, C Refresher
CSE 333 Winter 2020

Instructor: Justin Hsia

Teaching Assistants:
Andrew Hu          Austin Chan          Brennan Stein
Cheng Ni           Cosmo Wang           Diya Joy
Guramrit Singh     Mengqi Chen          Pat Kosakanchit
Rehaan Bhimar      Renshu Gu            Travis McGaha
Zachary Keyes
Lecture Outline

- **Course Introduction**
- **Course Policies**
  - [https://courses.cs.washington.edu/courses/cse333/20wi/syllabus/](https://courses.cs.washington.edu/courses/cse333/20wi/syllabus/)
- **C Reintroduction**
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 3rd 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

- ~190 students registered, split across two lectures
  - Largest offering ever!
  - 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 (in the Gates Center) to handle enrollment details

- 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

OS / app interface (system calls)
HW/SW interface (x86 + devices)

operating system

hardware

C application
C standard library (glibc)

C++ application
C++ STL/boost/standard library

Java application
JRE

CPU memory storage network
GPU clock audio radio peripherals
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
  - Treat assignment submissions in this class as production code
    - Comments must be updated, no commented-out code, no extra (debugging) output
Lecture Outline

- **Course Introduction**
- **Course Policies**
  - [https://courses.cs.washington.edu/courses/cse333/20wi/syllabus/](https://courses.cs.washington.edu/courses/cse333/20wi/syllabus/)
  - Digest here, but you **must** read the full details online
- **C Reintroduction**
Communication

- **Website:** [http://cs.uw.edu/333](http://cs.uw.edu/333)
  - Schedule, policies, materials, assignments, etc.

- **Discussion:** [http://piazza.com/washington/winter2020/cse333](http://piazza.com/washington/winter2020/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 (26) – *fewer than normal*
  ▪ 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
  ▪ We are checking for: *correctness, memory issues, code style/quality*

❖ Programming Projects (0+4)
  ▪ Warm-up, then 4 “homework” that build on each other

❖ Exams (2)
  ▪ **Midterm:** Friday, February 14, TBD [joint]
  ▪ **Final:** Wednesday, March 18, 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 additive
  - Heavy multitaskers have more trouble focusing and shutting out irrelevant information
  - 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/cse333/20wi/syllabus/
- C Reintroduction
  - 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`

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

Editor (emacs, vi) or IDE (eclipse)

Source files (.c, .h)

Object files (.o)

Statically-linked libraries

libZ.a

Shared libraries

libc.so

“COMPILE” (compile + assemble)

EDIT

LINK

LOAD

EXECUTE, DEBUG, ...
C to Machine Code

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

C source file (sumstore.c)

C compiler (gcc –S)

Assembly file (sumstore.s)

Assembler (gcc -c or as)

Machine code (sumstore.o)

400575: 01 fe
089 32
0c3
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
    - Standard codes found in `stdlib.h`: `EXIT_SUCCESS` (usually 0) and `EXIT_FAILURE` (non-zero)
    - Return value from `main` is a status code
  - 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’)

<table>
<thead>
<tr>
<th>Language Feature</th>
<th>S/D</th>
<th>Differences in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primitive datatypes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Casting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arrays</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory management</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Primitive Types in C

- **Integer types**
  - `char, int`

- **Floating point**
  - `float, double`

- **Modifiers**
  - `short [int]`
  - `long [int, double]`
  - `signed [char, int]`
  - `unsigned [char, int]`

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>32-bit</th>
<th>64-bit</th>
<th>printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>%c</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
<td>%hd</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2</td>
<td>2</td>
<td>%hu</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>%d/%i</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>4</td>
<td>%u</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
<td>%ld</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
<td>%lld</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>%lf</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16</td>
<td>%Lf</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>%p</td>
</tr>
</tbody>
</table>

Typical sizes – see `sizeofs.c`
C99 Extended Integer Types

- Solves the conundrum of “how big is an long int?”

```c
#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
    ...
}

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

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

- Generic format:

```
returnType fname(type param1, ..., type paramN) {
    // statements
}
```

```c
// sum of integers from 1 to max
int32_t sumTo(int32_t max) {
    int32_t 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!

```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
int32_t sumTo(int32_t max) {
    int32_t 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 of integers from 1 to max
int32_t sumTo(int32_t max) {
    int32_t 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 arguments and return types; function definitions can then be in a logical order
  - Function comment usually by the *prototype*

```c
// sum of integers from 1 to max
int32_t sumTo(int32_t); // func prototype

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

int32_t sumTo(int32_t max) {
    int32_t i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```

```c
// sum of integers from 1 to max
int32_t sumTo(int32_t); // func prototype

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

int32_t sumTo(int32_t max) {
    int32_t 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
Multi-file C Programs

C source file 1
(sumstore.c)

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

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;
}
```

Compile together:

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

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

```
ld or gcc  libraries (*e.g.* libc)  sumnum
```
Polling Question

Which of the following statements is FALSE?

A. With the standard `main()` syntax, it is always safe to use `argv[0]`.

B. We can’t use `uint64_t` on a 32-bit machine because there isn’t a C integer primitive of that length.

C. Using function declarations is beneficial to both single- and multi-file C programs.

D. When compiling multi-file programs, not all linking is done by the Linker.

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](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” under “Winter 2020”, Assignment “ex0 - Exercise 0”, then drag-n-drop file(s)!
  - Sample solution will be posted Wednesday afternoon
- **Hint:** look at documentation for `stdlib.h`, `string.h`, and `inttypes.h`