Intro, C
CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:
Danny Allen  Dennis Shao  Eddie Huang
Kevin Bi  Jack Xu  Matthew Neldam
Michael Poulain  Renshu Gu  Robby Marver
Waylon Huang  Wei Lin
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 1st time!

- **TAs:**
  - Available in section, office hours, and on Piazza
  - An invaluable source of information and help

- Get to know us
  - We are here to help you succeed!
Introductions: Students

- ~175 students registered, split across two lectures
  - Largest offering of this class EVER!!!
  - There are no longer overload forms for CSE courses
    - Majors must add using the UW system as space becomes available
    - Non-majors must have submitted petition form (closed now)

- Expected background
  - **Prereq:** CSE351 – C, pointers, memory model, linker, system calls
Course Map: 100,000 foot view

- C application
- C++ application
- Java application
- C standard library (glibc)
- C++ STL/boost/standard library
- JRE

OS / app interface (system calls)

HW/SW interface (x86 + devices)

operating system

- CPU
- memory
- storage
- network

hardware

- 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 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/cse333/18sp/syllabus/](https://courses.cs.washington.edu/courses/cse333/18sp/syllabus/)
- C Intro
Communication

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

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

- **Office Hours:** spread throughout the week
  - Can also e-mail 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 (~20)
  - Roughly one per lecture, due the morning of the next lecture
  - Coarse-grained grading (0, 1, 2, or 3)

- Programming Projects (4.5)
  - Warm-up, then 4 “homework” that build on each other

- Exams (2)
  - **Midterm**: Friday, May 4, time TBD (joint)
  - **Final**: Wednesday, June 6, 12:30-2:20 pm (joint)
Grading

- **Exercises:** 20% total
  - Submitted via Canvas
  - 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%)
  - Some old exams on course website

- **EPA:** Effort, Participation, and Altruism (5%)
Deadlines and Student Conduct

- **Late policies**
  - **Exercises**: no late submissions accepted
  - **Projects**: 4 late day “tokens” for quarter, max 2 per project
  - Need to get things done on time – difficult to catch up!

- **Academic Integrity**
  - I will 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 – can still learn from the course staff and peers
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
  - 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/cse333/18sp/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”
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). Needed because C doesn't track array lengths!
  - `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

LINK

Shared libraries

libc.so

LINK

“COMPILE” (compile + assemble)

foo.h

foo.c

bar.c

foo.o

bar.o

bar

(exeutable)

(process)

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
sumstore:
    addl %edi, %esi
    movl %esi, (%rdx)
    ret
```

Assembly file
(sumstore.s)

**Assembler (gcc -c or as)**

Machine code
(sumstore.o)

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

<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>S</td>
<td></td>
</tr>
<tr>
<td>Primitive datatypes</td>
<td>S/D</td>
<td>yes pointers, no String, yes unsigned different data widths (e.g. char)</td>
</tr>
<tr>
<td>Operators</td>
<td>S</td>
<td>Java has <code>&gt;&gt;&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C has <code>-&gt;</code></td>
</tr>
<tr>
<td>Casting</td>
<td>D</td>
<td>C has no casting restrictions</td>
</tr>
<tr>
<td>Arrays</td>
<td>D</td>
<td>C has no length or bounds checking</td>
</tr>
<tr>
<td>Memory management</td>
<td>D</td>
<td>no garbage collection explicit requests: <code>malloc</code> <code>/free</code></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: C Data Type Sizes

<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
    _int8_t w;  // exactly 8 bits, unsigned
    ...
}
```

```
#include <stdint.h>

void sumstore(int x, int y, int* 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
    ```c
    char* x = "hello\n";  \xrightarrow{\text{char}} \ 
    h \
    e \
    l \
    l \
    o \
    \n \
    \0
    ```

- **Structs** are the most object-like feature, but are just collections of fields
Function Definitions

- Generic format:

```c
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 += 1;
    }

    return sum;
}
Function Ordering

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

```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 += 1;
    }
    return sum;
}
```

```
C compiler goes line-by-line:

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 += 1;
    }
    return sum;
}
```
Solution 1: Reverse Ordering

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

```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 += 1;
    }
    return sum;
}

int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}
```

`sum_betterorder.c`
Solution 2: Function Declaration

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

```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
  - 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:

```
$ 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

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

ld or gcc libraries (e.g. libc)
sumnum
```
Peer Instruction 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, and Poll Everywhere

- 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 Wednesday before class (11 am)
  - Find exercise spec on website, submit via Canvas
  - Sample solution will be posted Wednesday at 12 pm