Intro, C refresher
CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:
Dao Yi            Farrell Fileas        Lukas Joswiak
Nathan Lipiarski  Renshu Gu           Travis McGaha
Yibo Cao          Yifan Bai            Yifan Xu
Lecture Outline

❖ Course Introduction

❖ Course Policies
  ▪ https://courses.cs.washington.edu/courses/cse333/19au/syllabus.html

❖ C Intro
Introductions: Course Staff

❖ Hannah C. Tang
  ▪ UW CSE alumna with 17 years of bugs in industry

❖ TAs:
  ▪ Dao Yi, Farrell Fileas, Lukas Joswiak, Nathan Lipiarski, Renshu Gu, Travis McGaha, Yibo Cao, Yifan Bai, Yifan Xu
  ▪ Available in section, office hours, and discussion group
  ▪ An invaluable source of information and help

❖ Get to know us
  ▪ We are excited to help you succeed!
Introductions: Students

❖ ~128 students registered
  ▪ There are no add codes 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

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

---

operating system

---

hardware

CPU memory storage network
GPU clock audio radio peripherals

OS / app interface (system calls)

HW/SW interface (x86 + devices)
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/cse333/19au/syllabus.html
  ▪ 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/fall2019/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
  ▪ Introduce the concepts; take notes!!!

❖ Sections
  ▪ Applied concepts, important tools and skills for assignments, clarification of lectures, exam review and preparation

❖ Programming Exercises
  ▪ One for most lectures, due the morning before the next lecture
  ▪ 4-point scale

❖ Programming Homeworks
  ▪ Warm-up, then 4 projects that build on each other

❖ Exams
  ▪ **Midterm:** Fri, Nov 1 @ 11:30-12:20
  ▪ **Final:** Wed, Dec 11 @ 2:30-4:20
Grading

❖ **Exercises:** 25% total
  ▪ Submitted via GradeScope (account info mailed later today)
  ▪ Graded on correctness and style by TAs

❖ **Homeworks:** 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

❖ **Participation:** Not strictly required, but it will only help!

❖ 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 10 am
▪ Projects: 4 late day “tokens” for quarter, max 2 per homework
▪ Need to get things done on time – difficult to catch up!

❖ Academic Conduct (read the full policy on the web)

▪ 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
  ▪ Seriously, you will learn more if you use *paper* instead!!!

❖ Non-disruptive use is 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/cse333/19au/syllabus.html](https://courses.cs.washington.edu/courses/cse333/19au/syllabus.html)

❖ 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

```c
#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)

“COMPILE” (compile + assemble)

EDIT

EDIT

foo.h

foo.c

bar.c

Statically-linked libraries

libZ.a

foo.o

bar.o

Shared libraries

libc.so

bar

bar

LINK

LINK

LOAD

LOAD

EXECUTE, DEBUG, ...

18
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`)

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

Machine code (sumstore.o)
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 or in global variables (!!!)
  ▪ 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]`

### C Data Type

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>32-bit</th>
<th>64-bit</th>
<th>printf</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char</code></td>
<td>1</td>
<td>1</td>
<td><code>%c</code></td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
<td><code>%hd</code></td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2</td>
<td>2</td>
<td><code>%hu</code></td>
</tr>
</tbody>
</table>
| `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 a 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
    ...
}
```

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

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

❖ C does not support objects!!!
  ▪ **Structs** are the most object-like feature, but are just collections of fields – no “methods” or functions

❖ **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!!!**

```c
char *x = "hello\n";    x → hello
```

❖ **Strings** are null-terminated char arrays
  ▪ Strings have no methods, but `string.h` has helpful utilities
Function Definitions

- Generic format:

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

Example:

```
// 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

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

```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;
}
```

sum_badorder.c
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 += 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

```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;
}
```

sum_declared.c
Function Declaration vs. Definition

❖ C/C++ make a very 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

```
gcc -c sumstore.c  ➔  sumstore.o

 gcc -c sumnum.c ➔  sumnum.o

 ld or gcc  ➔  sumnum

 libraries (e.g. libc)
```
Discuss with your neighbor

- Next lecture: we will vote at http://PollEv.com/cse333
- This lecture: just practice!

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 primitive of that length.

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

D. I’m not sure...
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 10 am on Friday
  ▪ Find exercise spec on website, submit via Gradescope
    • Course “CSE 333 Fall 19”, Assignment “Exercise 0”, then drag-n-drop file(s)! Ignore any messages about autograding.
  ▪ Sample solution will be posted Friday afternoon