CSE 333
Lecture 1 - Intro, C refresher

Hal Perkins
Department of Computer Science & Engineering
University of Washington
Welcome!

Today’s goals:
- introductions
- course syllabus
- quick C refresher
Introductions

Us (cse333-staff@cs)
- Hal Perkins (Instructor)
- Graham Blair (TA)
- Sunjay Cauligi (TA)
- Bryan Martin (TA)
- Soumya Vasisht (TA)

Most important: You!!
Overloading

The overload signup sheet is up front
- come sign up after lecture
- I’ll hand the sheet in to the ugrad advisors
- by Friday, they’ll let me (and you) know who gets in
Welcome!

Today’s goals:
- *introductions*
- **course syllabus**
- *quick C refresher*
Course map: 100,000 foot view

OS / app interface
(system calls)

HW/SW interface
(x86 + devices)

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
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”
  - quiz: *is data safely on disk after a “write( )” system call returns?*
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
What you will be doing

Attending lectures and sections
- lecture: ~29 of them, MWF in this room
- sections: ~10 of them, Thu (8:30, 9:30, 10:30; rooms on web)

Doing programming projects
- 5 of them, successively building on each other
- includes C, C++; file system, network

Doing programming exercises
- one per lecture, due before the next lecture begins
- coarse-grained grading (0,1,2,3)

Midterm and a final exam (your instructor is a traditionalist)
Deadlines & Conduct

Need to get things done on time (very hard to catch up)
- Programming assignments: 4 late days, 2 max per project
- Exercises: no late days (max benefit that way)

Academic Integrity (details on the web; read them)
- I trust you implicitly; I will follow up if that trust is violated
- The rules boil down to: don’t attempt to gain credit for something you didn’t do, and don’t help others do so
- That does not mean suffer in silence - you have colleagues, instructor, TAs - work with them; learn from each other!
Course calendar

Linked off of the course web page

- master schedule for the class
- will contain links to:
  ‣ lecture slides
  ‣ code discussed in lectures
  ‣ assignments, exercises (including due dates)
  ‣ optional “self-exercise” solutions
Welcome!

Today’s goals:
- introductions
- course syllabus
- quick C refresher
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, smaller standard library than Java
- procedural (not object-oriented)
- typed but unsafe; incorrect programs can fail spectacularly
C workflow

source files (.c, .h)

foo.h

object files (.o)

foo.o

compile

bar.c

link

bar

link

bar.o

link

libZ.a

link

libc.so

execute, debug, profile,

...
From C to machine code

C source file (dosum.c)

```
int dosum(int i, int j) {
    return i+j;
}
```

C compiler (gcc -S)

```
dosum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
popl %ebp
ret
```

assembly source file (dosum.s)

machine code (dosum.o)

```
80483b0:  55
89 e5 8b 45
0c 03 45 08
5d c3
```

assembler (as)
Skipping assembly language

Most C compilers generate .o files (machine code) directly
- i.e., without actually saving the readable .s assembly file

```
gcc -S dosum.c > dosum.s
as dosum.s > dosum.o
gcc -c dosum.o
```
Multi-file C programs

C source file (dosum.c)

```c
#include <stdio.h>

int dosum(int i, int j) {
    return i+j;
}
```

dosum() is implemented in dosum.c

C source file (sumnum.c)

```c
int main(int argc, char **argv) {
    printf("%d\n", dosum(1,2));
    return 0;
}
```

this “prototype” of dosum() tells gcc about the types of dosum’s arguments and its return value
Multi-file C programs

C source file (dosum.c)

```c
#include <stdio.h>

int dosum(int i, int j) {
    return i+j;
}
```

C source file (sumnum.c)

```c
int dosum(int i, int j);

int main(int argc, char **argv) {
    printf("%d\n", dosum(1,2));
    return 0;
}
```

why do we need this `#include`?

where is the implementation of `printf`?
Compiling multi-file programs

Multiple object files are linked to produce an executable

- standard libraries (libc, crt1, ...) are usually also linked in
- a library is just a pre-assembled collection of .o files

```
dosum.c
 gcc -c  dosum.o

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

```
lld (or gcc)

sumnum

libraries (e.g., libc)
```
Object files

sumnum.o, dosum.o are **object files**

- each contains machine code produced by the compiler
- each might contain references to external symbols
  - variables and functions not defined in the associated .c file
  - e.g., sumnum.o contains code that relies on printf( ) and dosum( ), but these are defined in libc.a and dosum.o, respectively
- linking resolves these external symbols while smooshing together object files and libraries
Let’s dive into C itself

Things that are the same as Java
- syntax for statements, control structures, function calls
- types: int, double, char, long, float
- type-casting syntax: float x = (float) 5 / 3;
- expressions, operators, precedence
  + - * / % ++ -- = += -= *= /= %= < <= == != > >= && || !
- scope (local scope is within a set of { } braces)
- comments: /* comment */ // comment
Primitive types in C

see sizeofs.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>type</th>
<th>bytes (32 bit)</th>
<th>bytes (64 bit)</th>
<th>32 bit range</th>
<th>printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>[0, 255]</td>
<td>%c</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
<td>[-32768, 32767]</td>
<td>%hd</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2</td>
<td>2</td>
<td>[0, 65535]</td>
<td>%hu</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>[-2147483648, 2147483647]</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>4</td>
<td>[0, 4294967295]</td>
<td>%u</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
<td>[-2147483648, 2147483647]</td>
<td>%ld</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
<td>[-9223372036854775808, 9223372036854775807]</td>
<td>%lld</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>approx [10^{-38}, 10^{38}]</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>approx [10^{-308}, 10^{308}]</td>
<td>%lf</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16</td>
<td>approx [10^{-4932}, 10^{4932}]</td>
<td>%Lf</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>[0, 4294967295]</td>
<td>%p</td>
</tr>
</tbody>
</table>
C99 extended integer types

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

```c
#include <stdint.h>

void foo(void) {
    int8_t  w;    // exactly 8 bits, signed
    int16_t x;   // exactly 16 bits, signed
    int32_t y;   // exactly 32 bits, signed
    int64_t z;   // exactly 64 bits, signed

    uint8_t a;    // exactly 8 bits, unsigned
    ...etc.
}
```
Similar to Java...

- variables
  - C99: don’t have to declare at start of a function or block
  - need not be initialized before use  \((gcc -Wall will warn)\)

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int x, y = 5;    // note x is uninitializated!
    long z = x+y;

    printf("z is '%ld'\n", z); // what's printed?
    {  
        int y = 10;  
        printf("y is '%d'\n", y);
    }
    int w = 20;    // ok in c99
    printf("y is '%d', w is '%d'\n", y, w);  
    return 0;
}
```
Similar to Java...

**const**

- a qualifier that indicates the variable’s value cannot change
- compiler will issue an **error** if you try to violate this
- why is this qualifier useful?

```c
#include <stdio.h>

int main(int argc, char **argv) {
    const double MAX_GPA = 4.0;

    printf("MAX_GPA: %g\n", MAX_GPA);
    MAX_GPA = 5.0; // illegal!
    return 0;
}
```

`consty.c`
Similar to Java...

for loops
- C99: can declare variables in the loop header
if/else, while, and do/while loops
- C99: bool type supported, with #include <stdbool.h>
- any type can be used; 0 means false, everything else true

```c
int i;
for (i = 0; i < 100; i++) {
    if (i % 10 == 0) {
        printf("i: %d\n", i);
    }
}
loopy.c
```
Similar to Java...

parameters / return value

- C always passes arguments by value

- "pointers"
  - lets you pass by reference
  - more on these soon
  - least intuitive part of C
  - very dangerous part of C
Very different than Java

arrays
- just a bare, contiguous block of memory of the correct size
- an array of 10 ints requires $10 \times 4$ bytes = 40 bytes of memory

arrays have no methods, do not know their own length
- C doesn’t stop you from overstepping the end of an array!!
- many, many security bugs come from this
Very different than Java

strings
- array of char
- terminated by the NULL character ‘\0’
- are not objects, have no methods; string.h has helpful utilities

```
char *x = "hello\n";
```
Very different than Java

errors and exceptions

- C has no exceptions (no try / catch)
- errors are returned as integer error codes from functions
- makes error handling ugly and inelegant

crashes

- if you do something bad, you’ll end up spraying bytes around memory, hopefully causing a “segmentation fault” and crash

objects

- there aren’t any; struct is closest feature (set of fields)
Very different than Java

memory management

- **you** must to worry about this; there is no garbage collector
- local variables are allocated off of the stack
  - freed when you return from the function
- global and static variables are allocated in a data segment
  - are freed when your program exits
- you can allocate memory in the heap segment using malloc()
  - you must free malloc’ed memory with free()
  - failing to free is a leak, double-freeing is an error (hopefully crash)
Very different than Java

Libraries you can count on

- C has very few compared to most other languages
- no built-in trees, hash tables, linked lists, sort, etc.
- you have to write many things on your own
  - particularly data structures
  - error prone, tedious, hard to build efficiently and portably
- this is one of the main reasons C is a much less productive language than Java, C++, python, or others
For Wednesday

Exercise 0 is due:

- http://www.cs.washington.edu/education/courses/cse333/13sp/exercises/ex0.html
- (Easier: look on the calendar or homework page for the link)

Homework 0 out before class Wednesday

- Mostly logistics (get files, fiddle with files, turn in files)
- Watch for email to course mailing list (and you are already subscribed if you are enrolled)
See you on Wednesday!

Sign up now if you aren’t in the class yet