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)
- Sunjay Cauligi (TA)
- Meghan Cowan (TA)
- Renshu Gu (TA)
- Sixto (Joshua) Rios (TA)
- Soumya Vasisht (TA)

Most important: You!!

Welcome!

Today's goals:

introductions

course syllabus

quick C refresher

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)

- - - - - - - - - - - - - - - - - - - -

operating system

HW/SW interface (x86 + devices)

....

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: ~28 of them, MWF here
- sections: ~10 of them, Thur., see course schedule for times, places
- Take notes!!!! Don’t expect everything to be on the web

Doing programming projects

- 4 of them, successively building on each other, plus a warmup
- 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

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; don't help others to do so
- That does ***not*** mean suffer in silence - you have colleagues, instructor, TAs - work with them; learn from each other!

Course web/calendar

Linked off of the course web page

- master schedule for the class (still under construction)
- links to:
  - lecture slides
  - code discussed in lectures
  - assignments, exercises (including due dates)
  - optional “self-exercise” solutions
  - various C/C++/Linux resources

Labs, office hours, &c

Office hours: plan is to have something Mon.-Fri.

- Fill out doodle on web with preferred times to help us pick

Discussion board to stay in touch outside of class

- See main web page for link, post followup to welcome msg

Mailing list for announcements

- You are automatically subscribed when you are registered

Welcome!

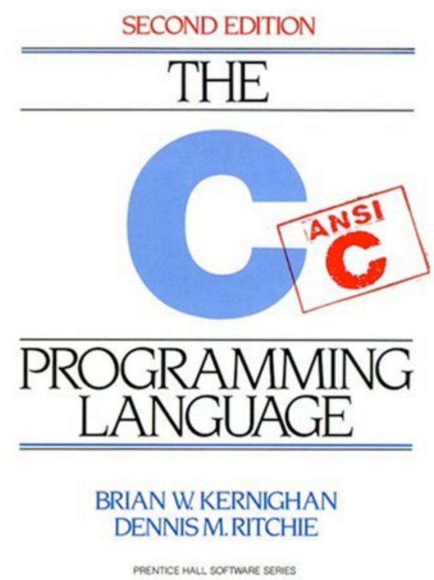
Today's goals:

introductions

course syllabus

quick C refresher

C



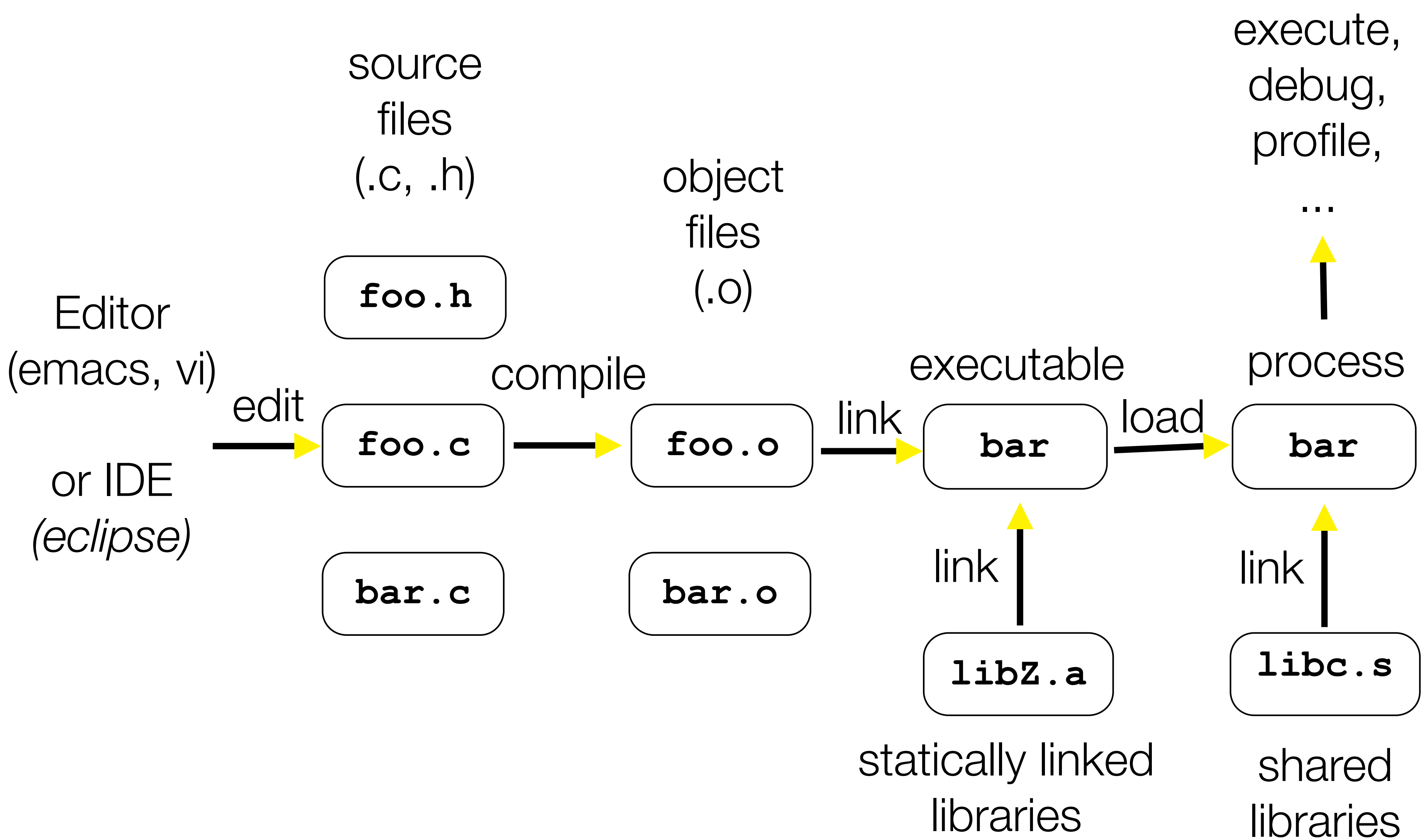
Created in 1972 by Dennis Ritchie

- designed for creating system software
- portable across machine architectures
- most recently updated in 1999 (C99) and 2011 (C11)

Characteristics

- low-level, smaller standard library than Java
- procedural (not object-oriented)
- typed but unsafe; incorrect programs can fail spectacularly

C workflow


source files (.c, .h)
object files (.o)
execute, debug, profile, ...
Editor (emacs, vi)
or IDE (eclipse)
edit
foo.h
foo.c
bar.c
compile
foo.o
bar.o
link
executable
bar
load
process
bar
link
libZ.a
link
libc.s
statically linked libraries
shared libraries

From C to machine code

C source file
(dosum.c)

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

C compiler (gcc -S)

assembly source file
(dosum.s)

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

assembler (as)

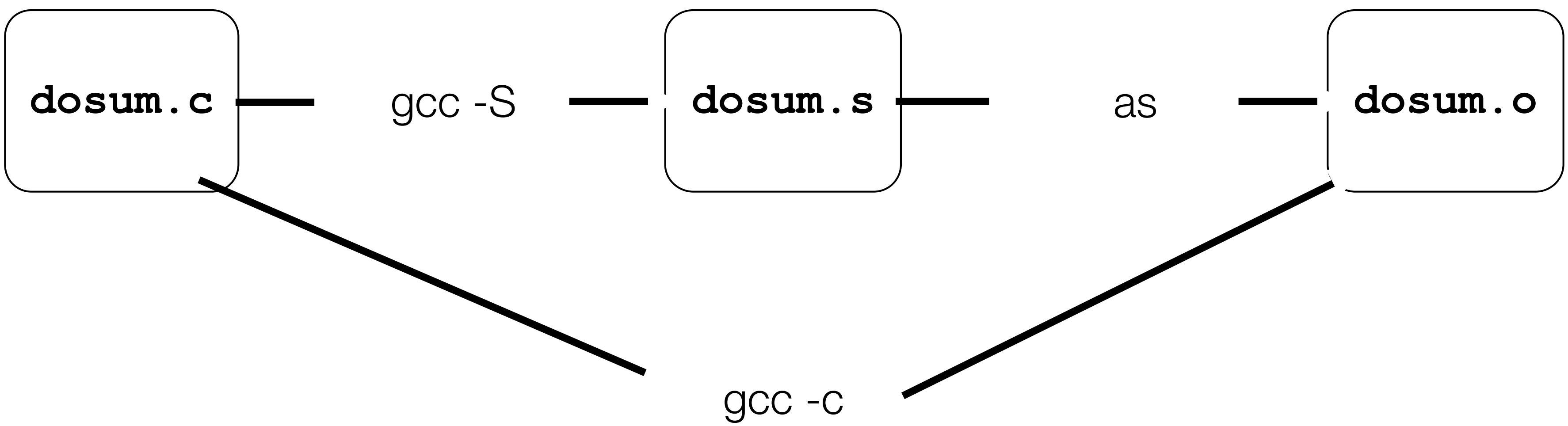
machine code
(dosum.o)

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

Skipping assembly language

Most C compilers generate .o files (machine code) directly

- i.e., without actually saving the readable .s assembly file



Multi-file C programs

C source file
(dosum.c)

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

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

C source file
(sumnum.c)

```
#include <stdio.h>

int dosum(int i, int j);

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

dosum( ) is implemented in dosum.c

Multi-file C programs

C source file
(dosum.c)

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

C source file
(sumnum.c)

```
#include <stdio.h>

int dosum(int i, int j);

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

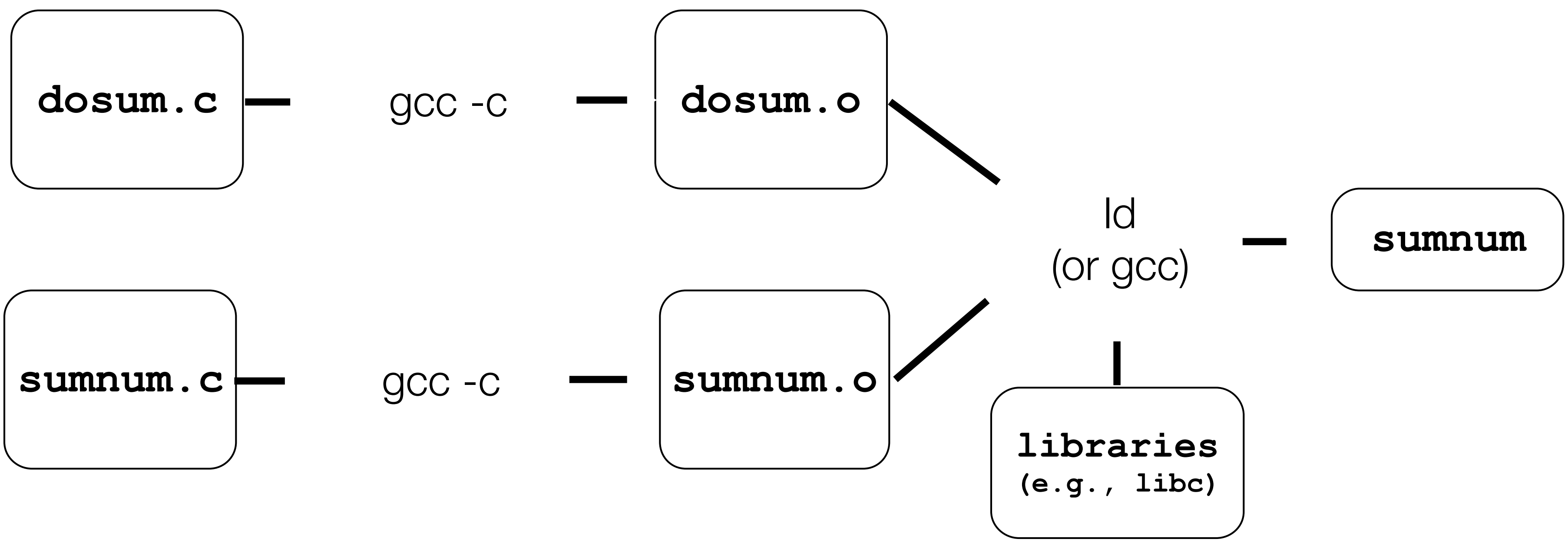
why do we need this #include?

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



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]

type	bytes (32 bit)	bytes (64 bit)	32 bit range	printf
char	1	1	[0, 255]	%c
short int	2	2	[-32768,32767]	%hd
unsigned short int	2	2	[0, 65535]	%hu
int	4	4	[-214748648, 2147483647]	%d
unsigned int	4	4	[0, 4294967295]	%u
long int	4	8	[-2147483648, 2147483647]	%ld
long long int	8	8	[-9223372036854775808, 9223372036854775807]	%lld
float	4	4	approx [10^{-38}, 10^{38}]	%f
double	8	8	approx [10^{-308}, 10^{308}]	%lf
long double	12	16	approx [10^{-4932}, 10^{4932}]	%Lf
pointer	4	8	[0, 4294967295]	%p

C99 extended integer types

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

```
#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/C11: don’t have to declare at start of a function or block
  - need not be initialized before use *(gcc -Wall will warn)*

varscope.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int x, y = 5;   // note x is uninitialized!
  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?

consty.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;
}
```

Similar to Java...

for loops

- C99/C11: can declare variables in the loop header

if/else, while, and do/while loops

- C99/C11: **bool** type supported, with #include <stdbool.h>
- any type can be used; 0 means **false**, everything else **true**

loopy.c

```
int i;

for (i = 0; i < 100; i++) {
  if (i % 10 == 0) {
    printf("i: %d\n", i);
  }
}
```

Similar to Java...

pointy.c

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

```
void add_pbv(int c) {
  c += 10;
  printf("pbv c: %d\n", c);
}

void add_pbr(int *c) {
  *c += 10;
  printf("pbr *c: %d\n", *c);
}

int main(int argc, char **argv) {
  int x = 1;

  printf("x: %d\n", x);

  add_pbv(x);
  printf("x: %d\n", x);

  add_pbr(&x);
  printf("x: %d\n", x);

  return 0;
}
```

Very different than Java

arrays

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

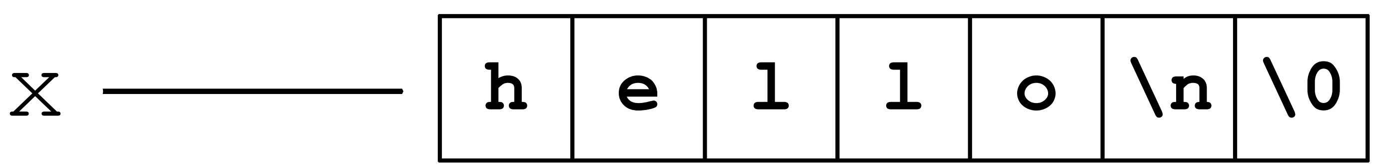
arrays have no methods, do not know their own length

- 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 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 *before* class:

http://www.cs.washington.edu/education/courses/cse333/15sp/exercises/ex00.html

(Easier: look on the calendar or homework page for the link)

Post a message on the discussion board

Get it to keep track of new messages for you!

Put office hour time preferences on doodle

HW0 out later this week - will announce when ready

Mostly logistics (get files, fiddle with files, turn in files)

See you on Wednesday!