CSE 333

Lecture 2 - gentle re-introduction to C

Steve Gribble

Department of Computer Science & Engineering

University of Washington

HW0 results

question	T	F
I have a laptop I can bring to class / section	90%	10%
languages I have used: **C** (87%) **C++** (32%) **x86** (56%) **ARM** (0%) **Java** (100%) **Python** (57%) **Perl** (4%) **Ruby** (38%) **JavaScript** (59%) **Go** (0%) **Haskell** (3%) **Klingon** (3%)		
languages I'm awesome at: **C** (4%) **C++** (1%) **x86** (1%) **ARM** (0%) **Java** (100%) **Python** (21%) **Perl** (0%) **Ruby** (3%) **JavaScript** (15%) **Go** (0%) **Haskell** (1%) **Romulan** (3%)		
Most code I have written as part of a product is: 1-100 lines (1%) 100-1000 lines (38%) 1000-10000 lines (55%) 10000+ (8.5%)		

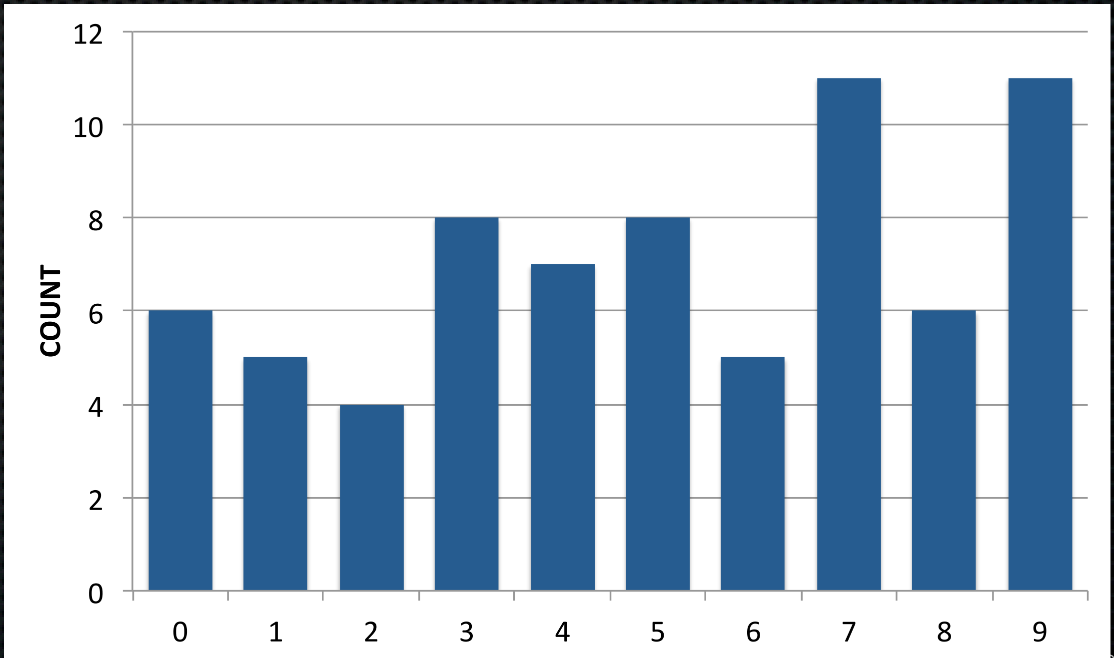
HW0 results

question	T	F
I took 351 from Mark Oskin last quarter	21%	79%
I have debugged pointer errors in my code	76%	24%
I have debugged memory leaks in my code	30%	70%
I have written network code	21%	79%
I have used the file system from my code	52%	48%

HW0 results

question	T	F
I know what a system call is; I've used one	27%	73%
I can write a Makefile	41%	59%
I've used a revision control system	91%	9%

pick a number
between 0 and 9



HW 0 results

Factor a ridiculously large number:

“Do not know, and don't have enough time to find out”

“2, 311, 20048005859189059106449115 27, why would you ask such a question?”

Today’s goals:

- overview of the C material you learned from cse351

Next two weeks’ goals:

- dive in deep into more advanced C topics
- start writing some C code
- introduce you to interacting with the OS

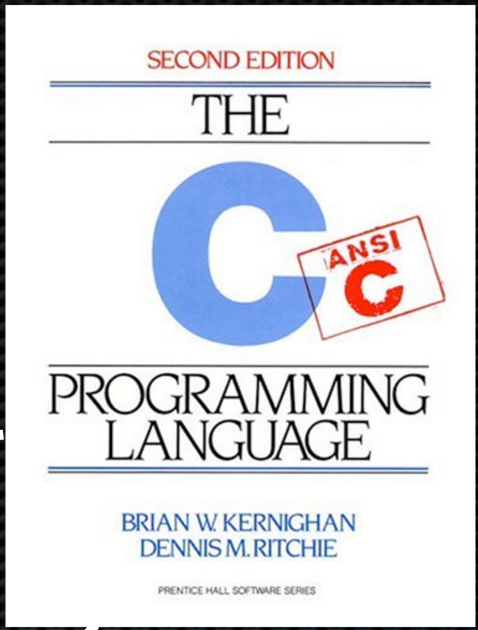
Attribution

The slides I'll be using are a mixture of:

- my own material
- slides from other UW CSE courses (CSE303, CSE351; thanks Magda Balazinska, Marty Stepp, John Zahorjan, Hal Perkins, Gaetano Borriello and others!!)
- material from other universities' courses (particularly CMU's 15-213 and some Harvard courses; thanks Randy Bryant, Dave O'Hallaron, Matt Welsh, and others!!)

All mistakes are mine. (No, really.)

C



Created in 1972 by Dennis Ritchie

- designed for creating system software

This book was typeset **`(pic|tbl|eqn|troff -ms)`** using an Autologic APS-5 phototypesetter and a DEC VAX 8550 running the 9th Edition of the UNIX operating system.

- procedural (not object-oriented)
- typed but unsafe; incorrect programs can fail spectacularly

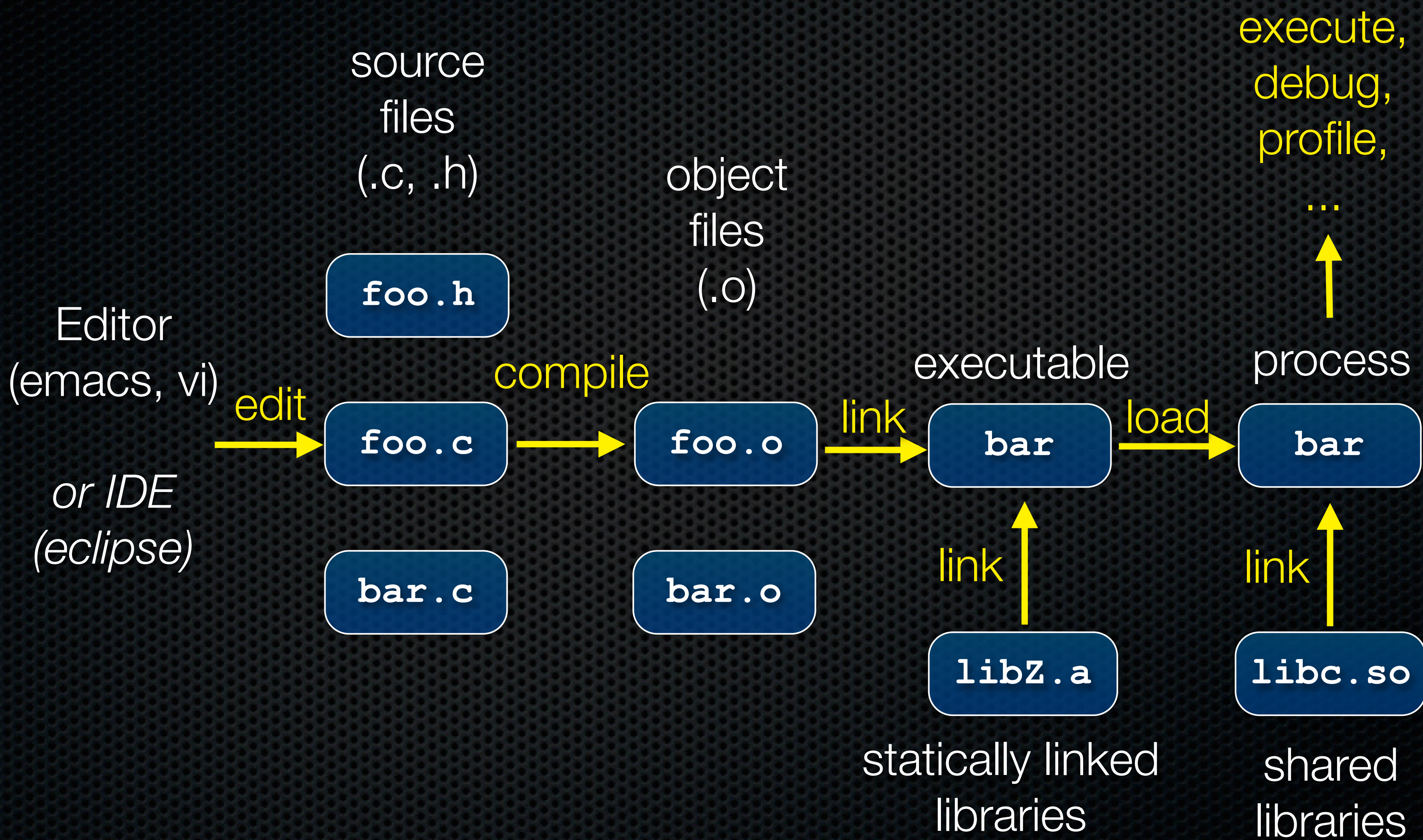
Mindset of C

"The PDP-11/45 on which our UNIX installation is implemented is a:

- 16-bit word (8-bit byte) computer with
  - 144K bytes of core memory; UNIX occupies 42K bytes
  - a 1M byte fixed-head disk
  - a moving-head disk with 40M byte disk packs
- The greater part of UNIX software is written in C."

Dennis M. Ritchie and Ken Thompson
Bell Laboratories
1974

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
link
libZ.a
libc.so
statically linked libraries
shared libraries

From C to machine code

C source file
(dosum.c)

```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)

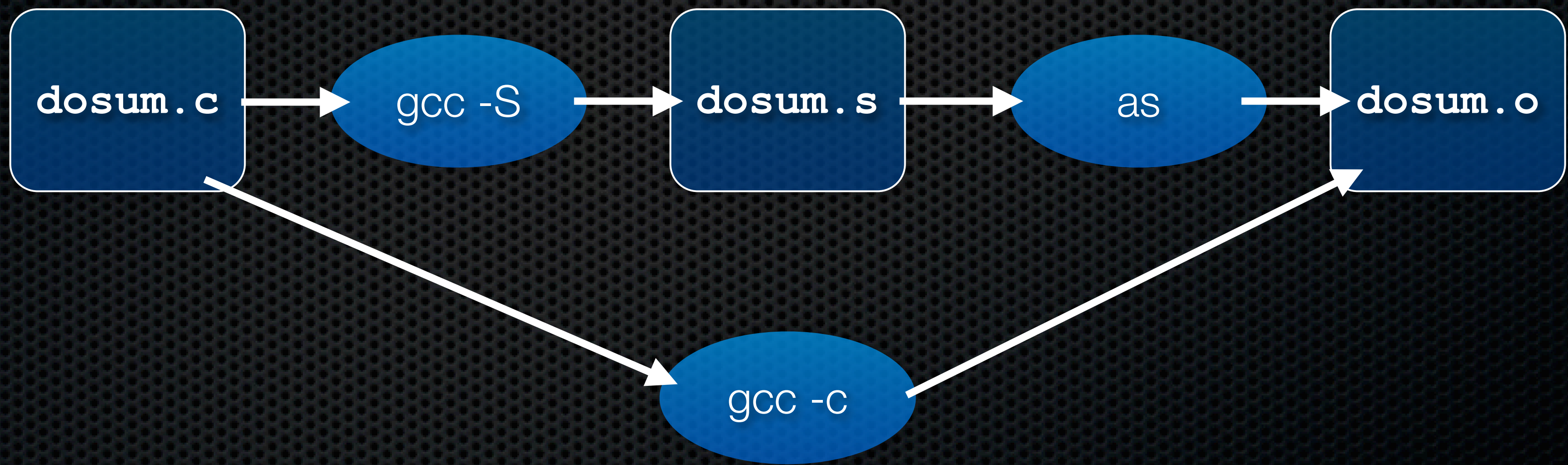
machine code
(dosum.o)

```
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;
}
```

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;
}
```

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

dosum( ) is implemented in sumnum.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;
}
```

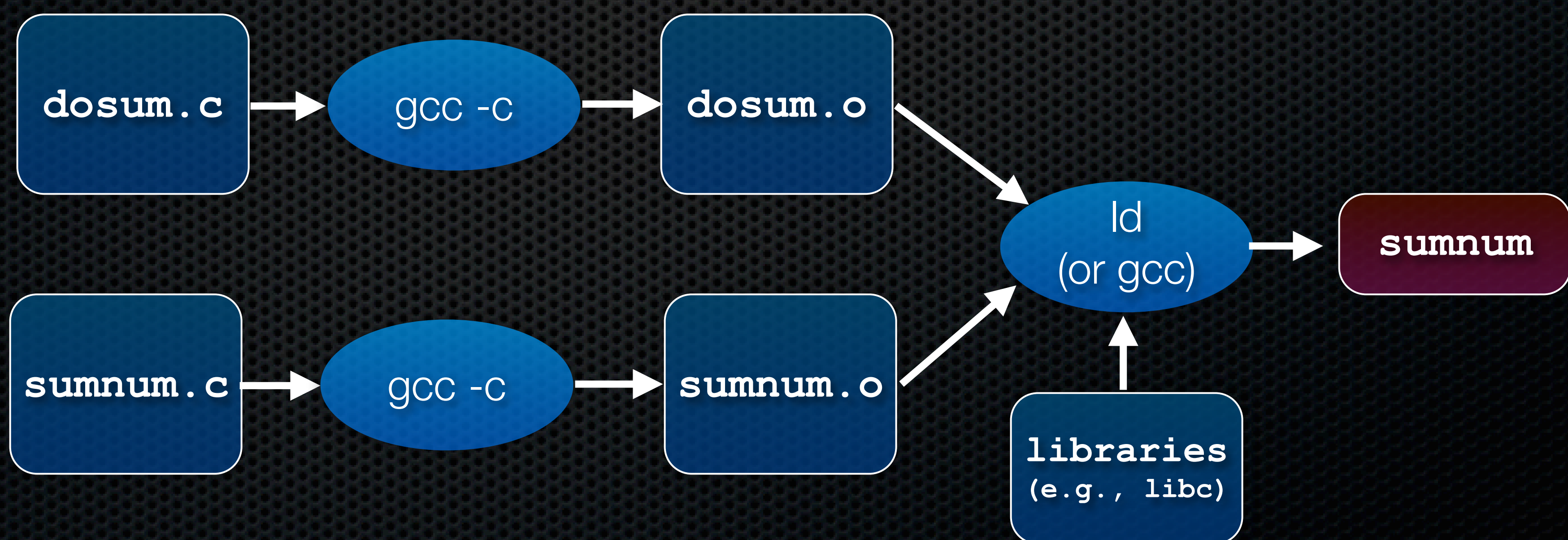
why do we need this #include?

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

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

varscope.c

```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

```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: 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**

loopy.c

```
int i;

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

Similar to Java...

pointy.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;
}
```

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 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 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

console I/O

- C standard library has portable routines for reading/writing
  - scanf, printf

file I/O

- C standard library has portable routines for reading/writing
  - fopen, fread, fwrite, fclose, etc.
  - does **buffering** by default, is **blocking** by default
- OS provides (less portable) routines
  - we'll be using these: more control over buffering, blocking

Very different than Java

network I/O

- C standard library has no notion of network I/O
- OS provides (somewhat portable) routines
- lots of complexity lies here
  - errors: network can fail
  - performance: network can be slow
  - concurrency: servers speak to thousands of clients simultaneously

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

See you on Friday!