Course map: 100,000 foot view

- **Hardware**
  - CPU
  - Memory
  - Storage
  - Network
  - GPU
  - Clock
  - Audio
  - Radio
  - Peripherals

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

- **OS / App Interface** (system calls)
  - OS / app interface
  - 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 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
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

Editor (emacs, vi)
or IDE (eclipse)

source files (.c, .h)

foo.h

bar.c

edit

compile

foo.c

foo.o

link

bar

link

link

libZ.a

object files (.o)

bar.o

link

load

libc.s

execute, debug, profile, ...

process

statically linked libraries

link

executable

bar

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)

```
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
```
Skipping assembly language

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

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

C source file (dosum.c)

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

C source file (sumnum.c)

```c
#include <stdio.h>

int dosum(int i, int j);  // this "prototype" tells gcc about the types of dosum's arguments and its return value

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)

```c
#include <stdio.h>

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

why do we need this `#include`?

C source file (sumnum.c)

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

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

```
$ gcc -c dosum.c
$ gcc -c sumnum.c
$ ld (or gcc) doSUM.o sumnum.o libraries (e.g., libc)
$ sumnum
```
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

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>[-214748648, 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/C11: 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 uninitialzied!
    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/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`

```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