CS107, Lecture 2
Unix, C, Bits and Bytes Intro

Reading: Bryant & O'Hallaron, Ch. 2.2-2.3 (skim)

The C Language

C was created by **Dennis Ritchie** at Bell Labs **between 1969 and 1972**, with its first stable form emerging around **1972**, and was invented to solve practical systems engineering problems, including the implementation of **Unix**.

Early **Unix** was written in:

- assembly language (difficult to maintain and impossible to port)
- the **B programming language** (lacked strong types, arrays, and records)

C is an extension of **B** and introduced:

- **`char`**s and **`long`**s, pointers, arrays, and records
- pointer arithmetic and the ability to reason about computer memory

C vs C++ and Java

All three share:

- syntax
- primitive data types
- arithmetic, relational, and logical operators
- common control idioms, e.g., **`for`** loops, **`switch`** statements, **`if`**/**`else`** clauses, functions

C limitations:

- no advanced features like operator overloading, default arguments, true pass by reference, object orientation
- few native libraries (no graphics, networking, etc.)
  - small language footprint, though 😇
- minimalist runtime model, near zero runtime error checking by default

Programming Language Philosophies

C is procedural: you write functions, rather than define new variable types with classes and invoke methods. **C is small, fast and efficient**.

Python is multi-paradigm but dynamically typed: you still write functions and call methods on objects but traditionally omit data types when coding. The development process is very different.

C++ is procedural, but with objects: you still write functions, and define new variable types with classes, and call methods on objects.

Java is primarily object oriented: virtually everything is an object and everything you write must conform to the object-oriented paradigm.

Why C? Nostalgia?

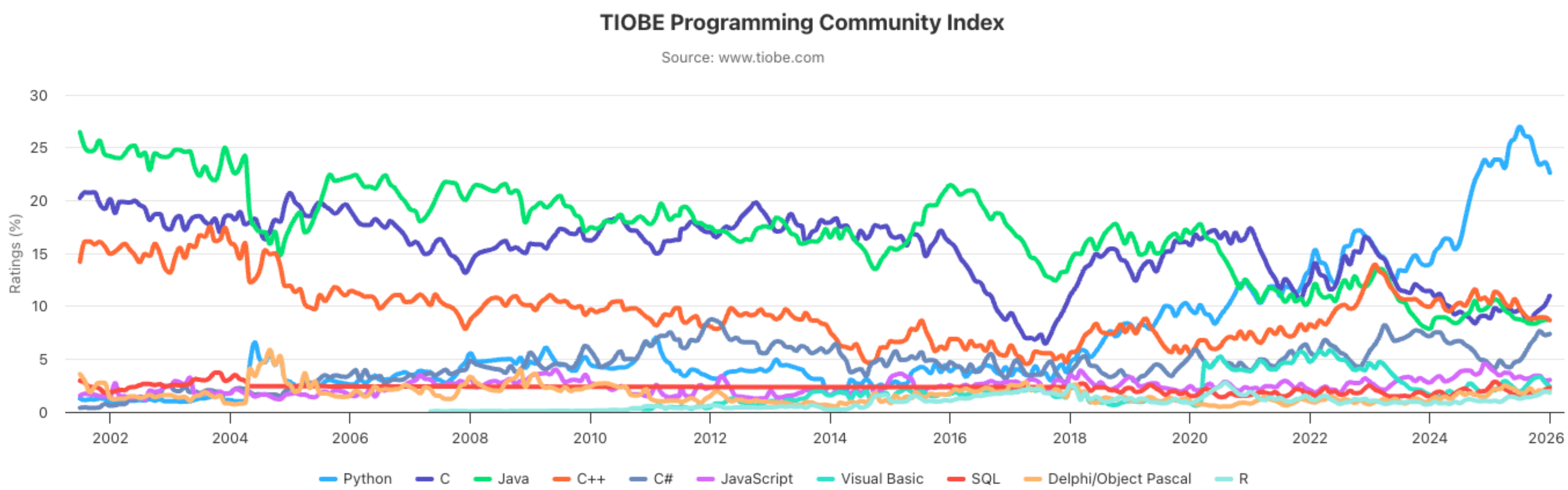
C lets you see **what the computer is really doing**. Memory, pointers, and data layout are explicit and not hidden behind abstractions.

C is the **language many systems are implemented in**. Operating systems, compilers, databases, drivers, and embedded firmware are largely written in C.

C is the **foundation underlying many higher-level programming languages**. Many languages and their runtimes are implemented in C, so understanding C explains language performance and limitations.

Learning **C** first makes **learning other systems languages easier**. Rust, Go, and others make more sense once you understand the problems C exposes directly.

Programming Language Popularity



The **ratings percentage** is the proportion of all programming language related search hits that mention a given language, relative to all languages tracked by TIOBE.

https://www.tiobe.com/tiobe-index/

Baby's First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>    // exposes printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Baby's First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // exposes printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Program comments
You can write block or inline comments.

Baby's First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>    // exposes printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Import statements
C libraries are written with angle brackets.
Local libraries use quotes instead, as with
`#include "wordle-utils.h"`

Baby's First C Program

```
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // exposes printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

main function: entry point for the program, should always return a small integer (0 == success)

Baby's First C Program

```
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>    // exposes printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

main parameters – **main** takes two parameters, both constructed using the command line arguments used to launch the program.

argc is the number of arguments in **argv**
argv is an array of arguments **(char * is C string)**

Baby's First C Program

```
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // exposes printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

printf prints output to the screen

Console Output: `printf`

```
printf(control, arg1, arg2, arg3,...);
```

`printf` makes it easy to print out the values of variables or expressions.

If you include **placeholders** in your printed text, **`printf`** will replace each placeholder with the values of subsequent parameters, passed after the text.

%s (string) %d (integer)

```
// Example
char *department = "CS";
int number = 107;
printf("You are in %s%d\n", department, number); // You are in CS107
```

Familiar Syntax

```
int x = 23
int y = 42 - 5 * x;                    // variables, types
double pi = 3.14159;
char c = 'Q';                          /* two comment styles */

for (int i = 0; i < 100; i++) {        // for loops
    if (i % 2 == 0) {                  // if statements
        x += i;
    }
}

while (x > 0 && c == 'Q' || b) {       // while loops, logic
    x = x / 2;
    if (x == 42) return 0;
}

return binky(x, y, pi, c);             // function call
```

Boolean Variables

To declare Booleans, (e.g., `bool b = __;`), you include `stdbool.h`

```
#include <stdio.h>    // for printf
#include <stdbool.h> // for bool

int main(int argc, char *argv[]) {
    bool x = argc > 2 && argv[argc - 1][0] != 'A';
    if (x) {
       printf("Hello, world!\n");
    } else {
       printf(”Greetings, traveler!\n");
    }
    return 0;
}
```

Command Line Arguments

argv captures array of tokens used to run program, and argc counts how many tokens

```
/* File: args.c */
#include <stdio.h> // for printf
int main(int argc, char *argv[]) {
    printf("This program got %d argument(s).\n", argc);
    for (size_t i = 0; i < argc; i++) {
        printf("Argument %zu: %s\n", i, argv[i]);
    }
    return 0;
}
```

```
myth$ ./args 1 2 "3 4" five
```

Writing, Debugging and Compiling

We will use:

- the **`emacs`** text editor to write our C programs
- the **`make`** tool to compile our C programs
- the **`gdb`** debugger to debug our programs
- the **`valgrind`** tools to debug memory errors and measure program efficiency

this week (emacs, make)

next week (gdb, valgrind)

Customary Workflow

- **`ssh`** – remotely log in to **`myth`** computers
- **`emacs`** – text editor to write and edit C programs
  - Use the mouse to position cursor, scroll, and highlight text
  - `Ctrl-x Ctrl-s` to save, `Ctrl-x Ctrl-c` to quit
- **`make`** – compile program using provided **`Makefile`**
- **`./myprogram`** – run executable program (perhaps with arguments)
- **`make clean`** – remove executables and other compiler files

Demo: Compiling And Running A C Program

Get up and running with our guide:

http://cs107.stanford.edu/getting-started.html

CS107 Topic 1: Bits and Bytes

How can a computer represent **`int`** values? or **`float`**s?

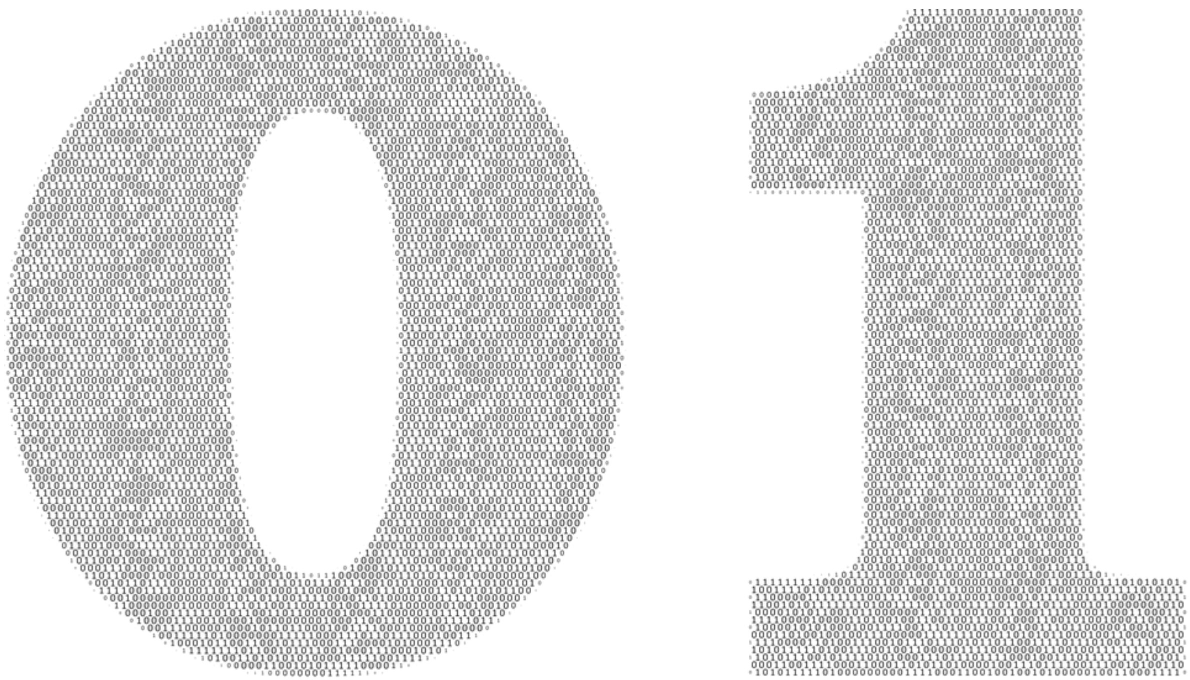
Why is answering this question useful?

- Helps us understand the limitations of computer arithmetic (today and Friday)
- Shows us how to more efficiently perform arithmetic (Friday and Monday)
- Shows us how we can encode data more compactly and efficiently (Monday)

Demo: Unexpected Behavior

```
cp -r /afs/ir/class/cs107/lecture-code/lect02 .
```

The Binary Digit aka Bit



One Bit At A Time

We can combine bits, as with base-10 numbers, to represent a larger collection of values

8 bits = 1 byte.

- Computer memory is just a large array of bytes. It is **byte addressable**, meaning you can’t address a bit in isolation, only a full byte.
- Computers still fundamentally operate on bits. It's just that we’ve gotten more creative about how to encode information.
  - images
  - audio
  - video
  - text

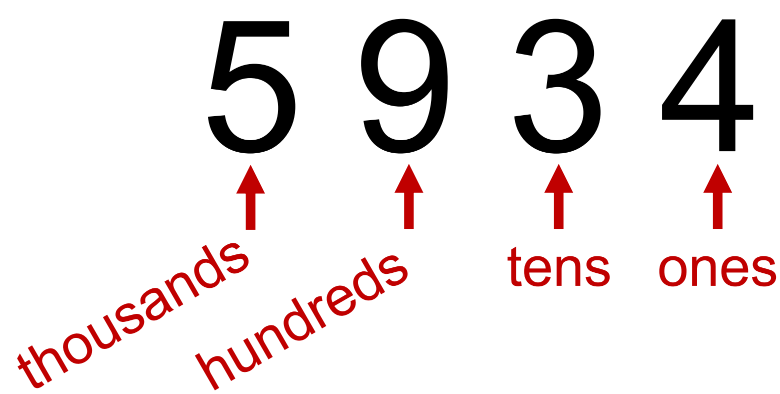
Base 10

5 9 3 4

digits 0 – 9

(or rather, 0 through base – 1)

Base 10



= **5** * 1000 + **9** * 100 + **3** * 10 + **4** * 1

Base 10

5 9 3 4

10^3 10^2 10^1 10^0

Base 10

	5	9	3	4
10^x:	3	2	1	0

Base 2

1 0 1 1

2^n: 3 2 1 0

digits 0 – 1

(or rather, 0 through base – 1)

Base 2

1 0 1 1

2^3 2^2 2^1 2^0

Base 2

most significant bit (MSB)

least significant bit (LSB)

1 0 1 1

eights fours twos ones

$$= \mathbf{1} * 8 + \mathbf{0} * 4 + \mathbf{1} * 2 + \mathbf{1} * 1 = 11_{10}$$

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **2^2=4**
  - Now, what is the largest power of 2 ≤ 6 – 2^2? **2^1=2**
  - 6 – 2^2 – 2^1 = 0

0	1	1	0
2^3	2^2	2^1	2^0

= **0***8 + **1***4 + **1***2 + **0***1 = 6

Practice: Base 2 to Base 10

What is the base-2 value of 1010 in base-10?

a) **20**
b) **101**
c) **10**
d) **5**
e) **Other**

1010 isn't 1010 so much as it is 8 + 2

Practice: Base 10 to Base 2

What is the base-10 value of 14 in base 2?

a) **1111**
b) **1110**
c) **1010**
d) **Other**

14 can be written as a sum of powers

14 isn't 14 so much as it is 8 + 4 + 2

that can be encoded as 1110

Byte Values

What are the **minimum** and **maximum** base-10 values that a single byte can represent?

minimum = 0 **maximum = 255**

11111111

2^x: 7 6 5 4 3 2 1 0

- **Strategy 1:** $1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 255$
- **Strategy 2:** $2^8 - 1 = 255$

Multiplying by Base

$$7453 \times 10 = 7453\mathbf{0}$$

$$1100_2 \times 10_2 = 1100\mathbf{0}$$

Key Idea: appending a 0 to the end effectively multiplies by the base.

Integer Dividing by Base

$$1458 / 10 = 145$$

$$1101_2 / 10_2 = 110$$

Key Idea: chomping off the last digit at the end integer divides by the base.

Hexadecimal

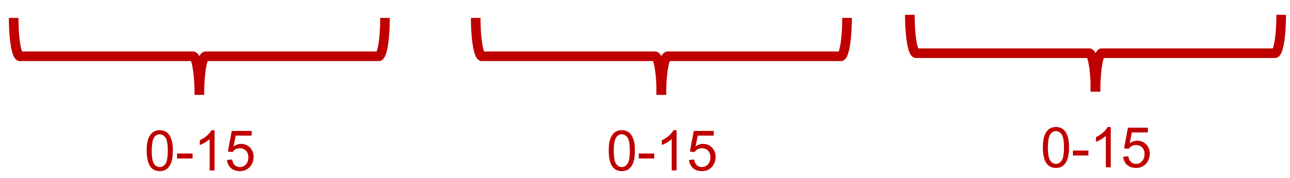
When working with 32- or 64-bit figures, binary representations get long. Instead, we'll often encode numbers in **base 16**, or **hexadecimal**.

0110 1010 0011

0-15 0-15 0-15

Hexadecimal

When working with 32- or 64-bit figures, binary representations get long.
Instead, we'll often encode numbers in **base 16**, or **hexadecimal**.



Each quartet of bits can be rewritten as a single digit in **base 16**!

Hexadecimal

Hexadecimal is **base 16**, so we need digits for 0 through 15, inclusive.
But how?

0 1 2 3 4 5 6 7 8 9

Hexadecimal

If it's not clear from context, we can explicitly identify numbers as hexadecimal by prefixing them with **`0x`** and identify numbers as binary using **`0b`**.

`0xf5` (or **`0xF5`**) is binary number **`0b11110101`** is decimal number **`245`**

0x f 5

1111 0101

Practice: Hexadecimal to Binary

What is **`0x173A`** in binary?

Hexadecimal	**1**	**7**	**3**	**A**
Binary	**0001**	**0111**	**0011**	**1010**

`0x173A = 0b1011100111010`

Practice: Binary to Hexadecimal

What is `0b1111001010` in hexadecimal? (Hint: start from the right)

Binary	**11**	**1100**	**1010**
Hexadecimal	**3**	**C**	**A**

`0b1111001010` = `0x3CA`

Hexadecimal: Quirky but concise

Let's look at a single byte, encoded three ways:

`165`	base 10: **human-readable**, but cannot easily interpret on/off bits
`0b10100101`	base 2: computers **love** this, but most humans lack that love.
`0xA5`	base 16: **easy to convert to base 2**, more easily digested format for humans

Number Representations

- **Unsigned Integers**: positive integers and 0 (e.g., 0, 1, 2, … 99999…)
- **Signed Integers**: negative, positive and 0 (e.g., …-2, -1, 0, 1, … 99999, …)
- **Floating Point Numbers**: real numbers (e,g. 0.1, -12.2, 1.18743 x 10^{12})