CS 107 Lecture 1: Welcome

Monday, January 9, 2023

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   %rbp
   0x000000000040052e <+1>:     mov    %rsp,%rbp
   0x0000000000400531 <+4>:     mov    $0x4005d4,%edi
   0x0000000000400536 <+9>:     callq  0x400410 <puts@plt>
   0x000000000040053b <+14>:    mov    $0x0,%eax
   0x0000000000400540 <+19>:    pop    %rbp
   0x0000000000400541 <+20>:    retq
End of assembler dump.
(gdb)
```


Computer Systems
Winter 2023
Stanford University
Computer Science Department

Lecturer: Chris Gregg

Reading: Course reader: Introduction, Number Formats used in CS 107, Bits and Bytes

Today's Topics

- What is CS107?
- Who We Are
- Course Components and Overview
- The C Language
- Logistics
  - Exams
  - Labs
  - Assignment 0
  - Lab Signup
- Bits and Bytes
- 0b10100 Questions

What is CS107?

- The CS106 series teaches you how to solve problems as a programmer
- Many times CS106 instructors had to say “just don’t worry about that” or “it probably doesn’t make sense why that happens, but ignore it for now” or “just type this to fix it”
- **CS107 *finally* takes you behind the scenes**
- **How do things really work in there?**
  - It’s not quite down to hardware or physics/ electromagnetism (those will have to stay even further behind the scenes for now!)
  - It’s how things work **inside Python/C++** **(we will explore from C)**, and how your programs map onto the components of computer systems

CS107 Learning Goals

- The goals for CS107 are for students to gain **mastery** of
  - writing C programs with complex use of memory and pointers
  - an accurate model of the address space and compile/runtime behavior of C programs
- to achieve **competence** in
  - translating C to/from assembly
  - writing programs that respect the limitations of computer arithmetic
  - identifying bottlenecks and improving runtime performance
  - writing code that correctly ports to other architectures
  - working effectively in UNIX development environment
- and have **exposure** to
  - a working understanding of the basics of computer architecture

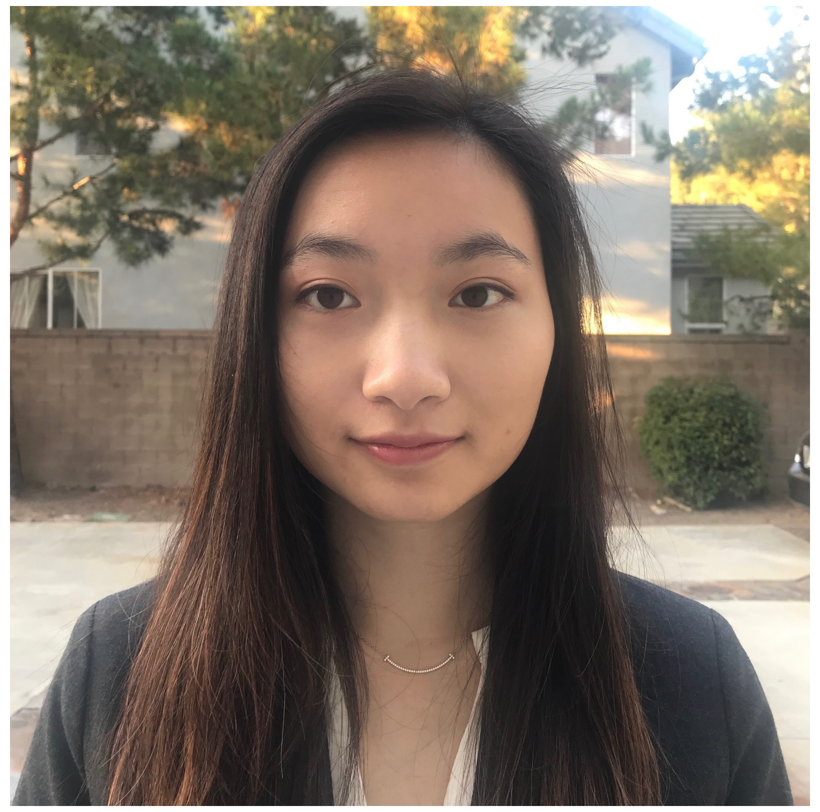
Who We Are: Chris

Chris Gregg
cgregg@stanford.edu

Who We Are: CAs

Daniel Rebelsky

Megan Worrel

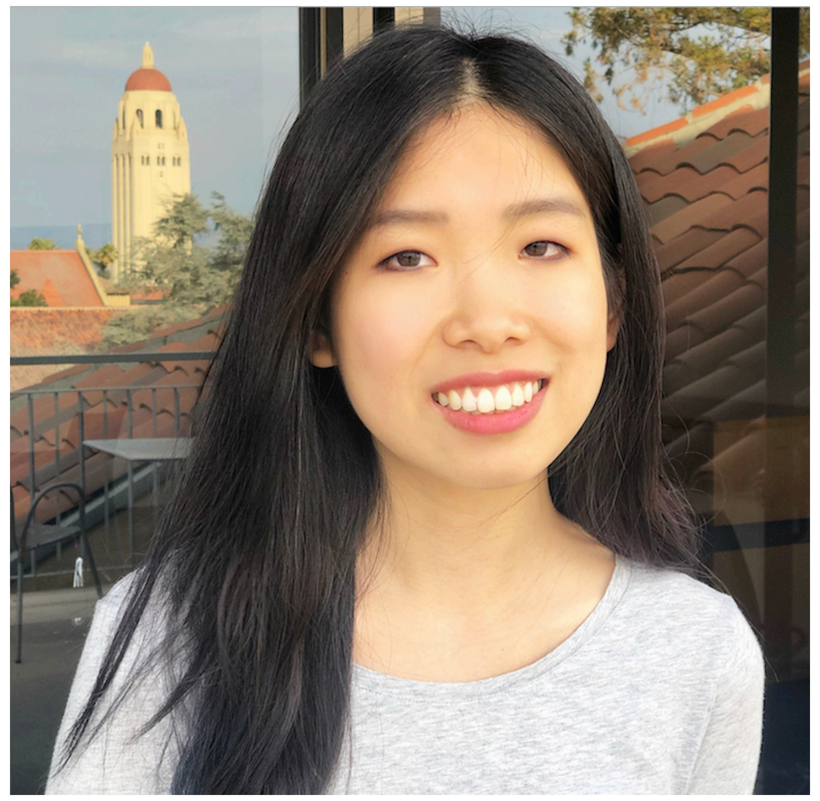


Christine Cheng

Eva Betelaan

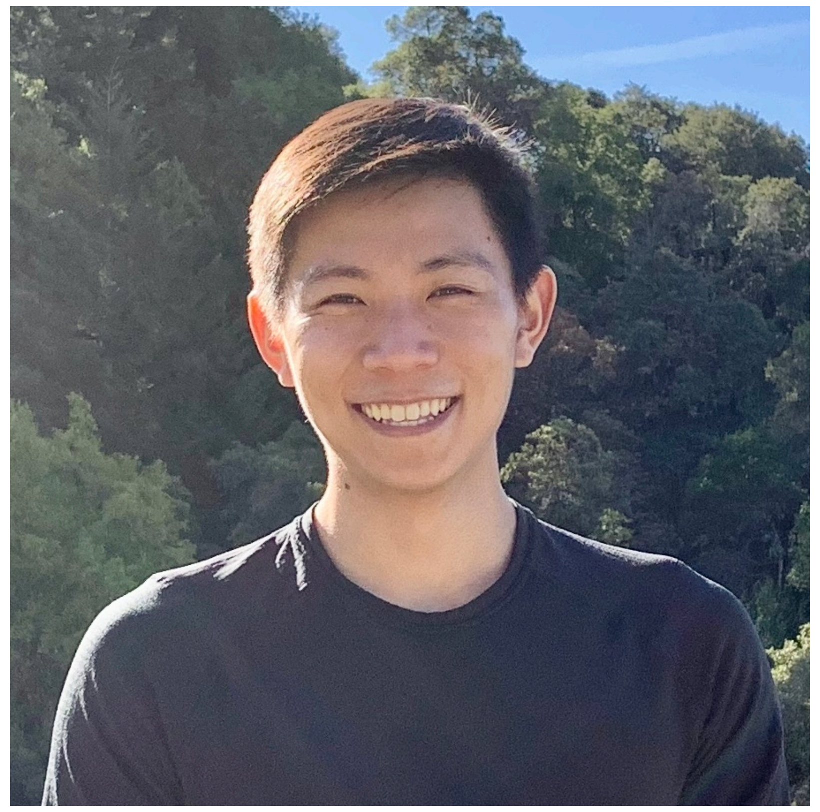
Alex Bradfield

Frankie Cerkvenik
(CS107A)



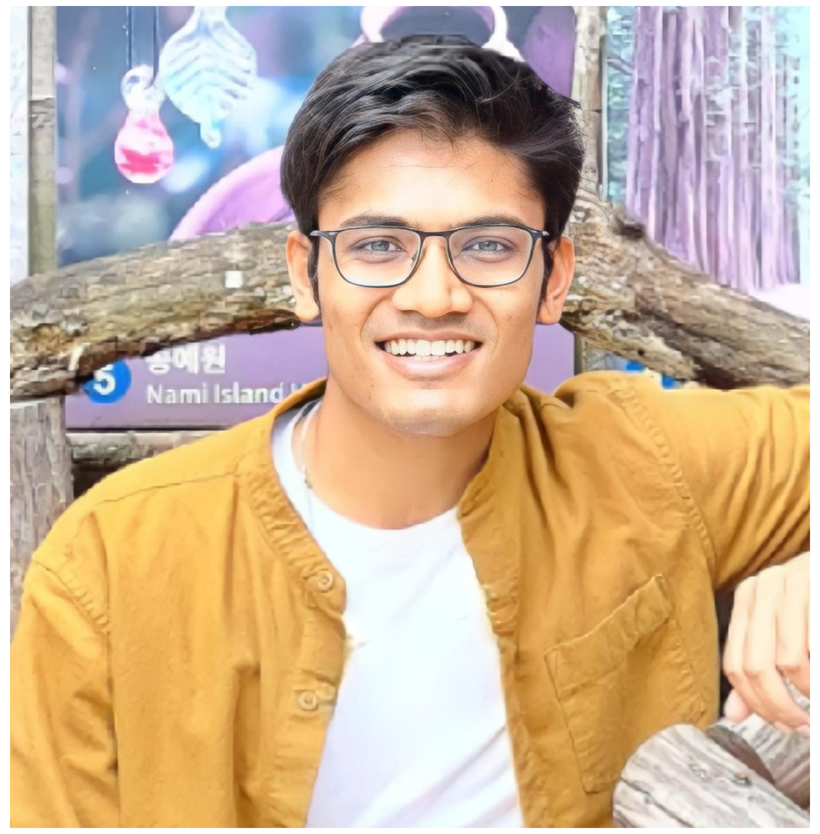
Tori Qiu

Jagriti Dixit



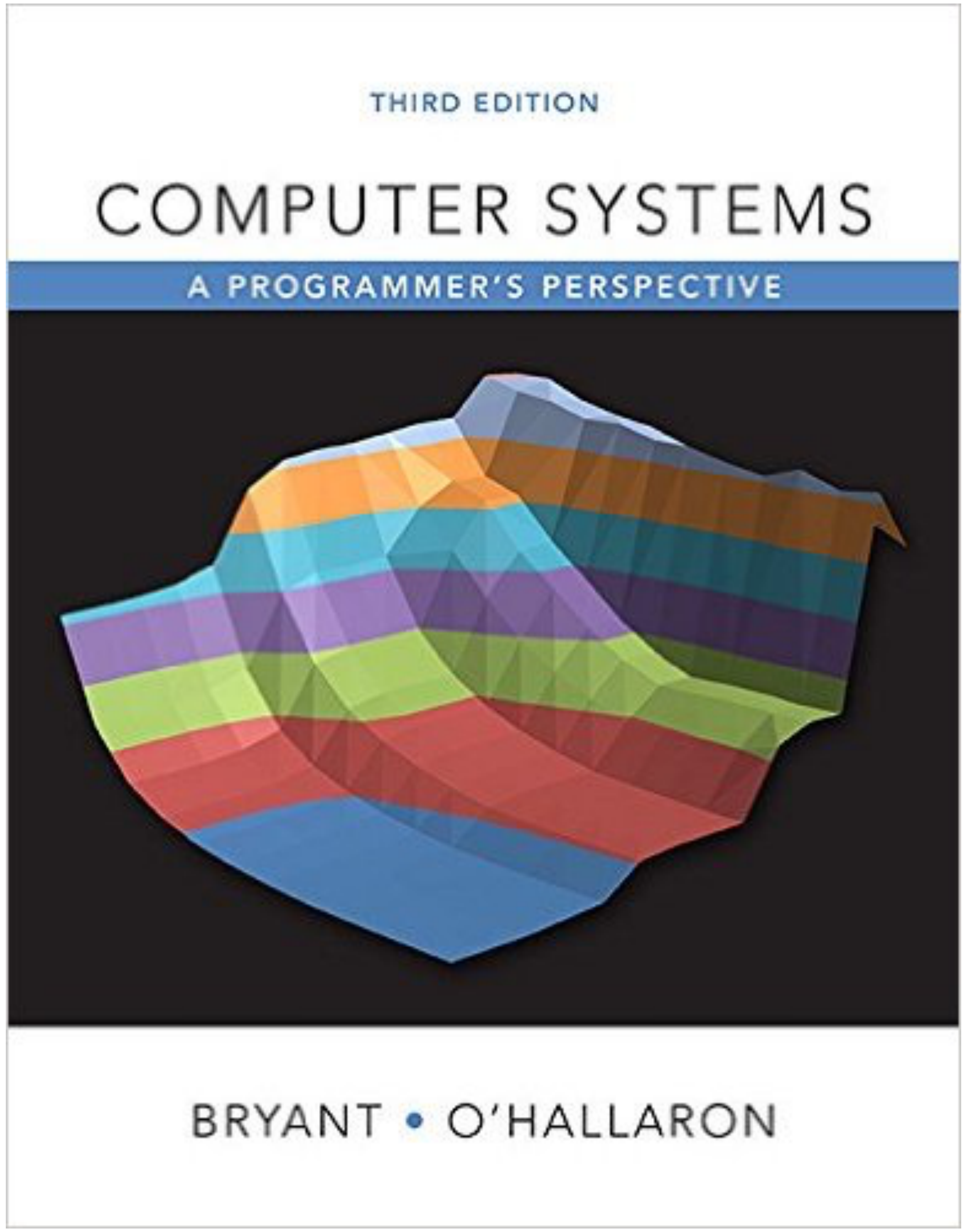
Jerry Chen

Derek Chung



Aman Kansal

Course Components and Overview



Textbook: Bryant and O'Hallaron, 3rd Edition

You must get the 3rd Edition, as things have significantly changed since the previous editions.

- The suggested C reference is just one suggestion
  - You could do just as well with a different C book
  - **You could do just as well with Google or websites like http://www.cplusplus.com/reference/clibrary/**
  - *Just need somewhere to turn when you have a question about C*

Course Components and Overview

CS 107 READER

STANFORD COMPUTER SCIENCE DEPARTMENT

There is a course reader, which condenses much of the material for the course:

https://stanford.edu/~cgregg/cgi-bin/107-reader

- If you find typos, let me know!

Course Components and Overview

Week	Topics
1	Admin, UNIX environment, Integer representation
2	Bits/bitwise ops, computer arithmetic, C pointers/arrays
3	C-strings, C stdlib, dynamic allocation
4	C generics, void *, function pointers
5	Floating point representation, intro to assembly
6	x86-64: addressing, ALU ops **Midterm**: **Wed Feb 15th, Evening**
7	x86-64: control, function calls, runtime stack
8	Address space, dynamic memory management
9	Performance / Optimization
10	Advanced topics, wrap/review
11	**Final**: **Friday Mar 24th 8:30AM-11:30AM**

The C Language

The C Language: History and Background

- Birthdate around 1970
- Created to make writing Unix (the OS itself) and tools for Unix easier
- Part of the C/C++/Java family of languages
  - (with C++ and Java coming later)
- Design principles:
  - Small,simple abstractions of hardware
  - Minimalist aesthetic
  - C is much more concerned with efficiency and minimalism than safety (Java/Python) or convenient high-level services and abstractions (Java, Python, C++)

The C Language: Comparison of C and C++

- **Some things will be very familiar:**
  - Syntax
  - Basic data types
  - Arithmetic, relational, and logical operators
- **You may be sad about what's missing:**
  - No power features of C++ (overloading operators, default arguments, pass by reference, classes/objects, fancy ADTs)
  - Thin standard libraries (no graphics, networking, etc)
  - Weak compiler checks, almost no runtime checks
- **Benefits:**
  - Small language footprint (not much to learn)
- **Philosophical difference:**
  - Procedural (C)
  - Procedural + Objects (C++)

The C Language: Hello, World! Compiling, gdb

Also: command line arguments and boolean values

Logistics

See the Course Handout for details (link)

Web site: https://cs107.stanford.edu

Class time: 11:30AM-1PM, M/F, Online (first two weeks), Bishop Auditorium (weeks 3-10, hopefully)

Labs: Various Times Tu/We/Th

Exams: Midterm, Wednesday, February 15th, Time TBD (evening)
Final Exam: Friday, March 24th, 8:30am-11:30am
(Note: there are **no** alternate final exam times)

Assignment 0: Unix!

Assignment page: https://web.stanford.edu/class/cs107/assign0/

Assignment already released, due Monday, 1/16

Six parts:

1. Read / View Unix Overview Documents / Videos
2. "Clone" Assignment 0 starter code
3. Answer Questions in `readme.txt`
4. Honor Code Quiz
5. Run `make` to compile a program, and make minor modifications
6. Submit the assignment

Lab Signup

Online:

https://cs107.stanford.edu/labs

The signup will be available Tuesday, January 10, 10:00am.

Labs will be weekly, starting during **week 2**.

Lab signup will not be first-come, first-served (you'll put in preferences), and the labs are held on Wednesdays, Thursdays

Bits and Bytes Introduction

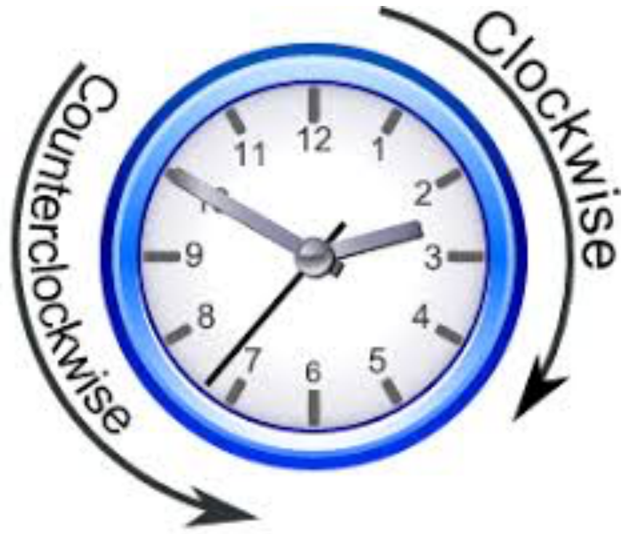
0

Bits and Bytes Introduction

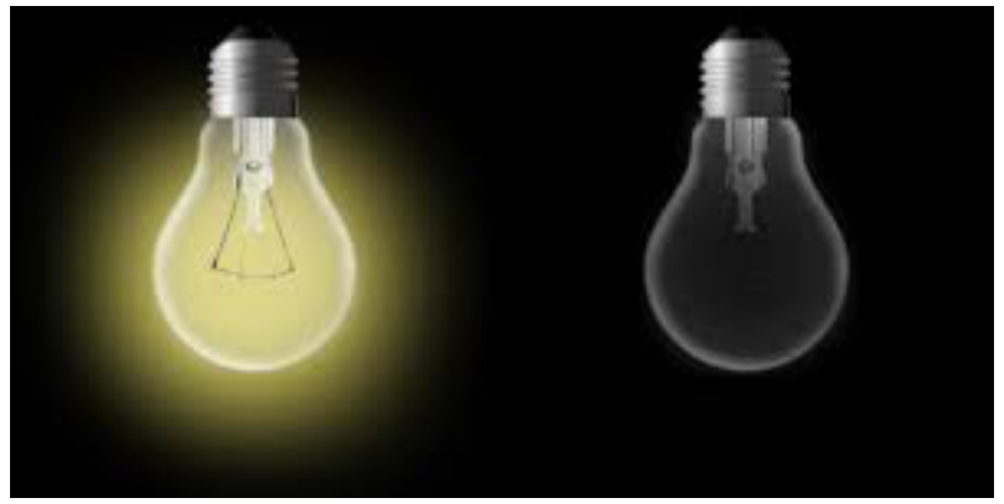
1

Computers are good at detecting "off" or "on"

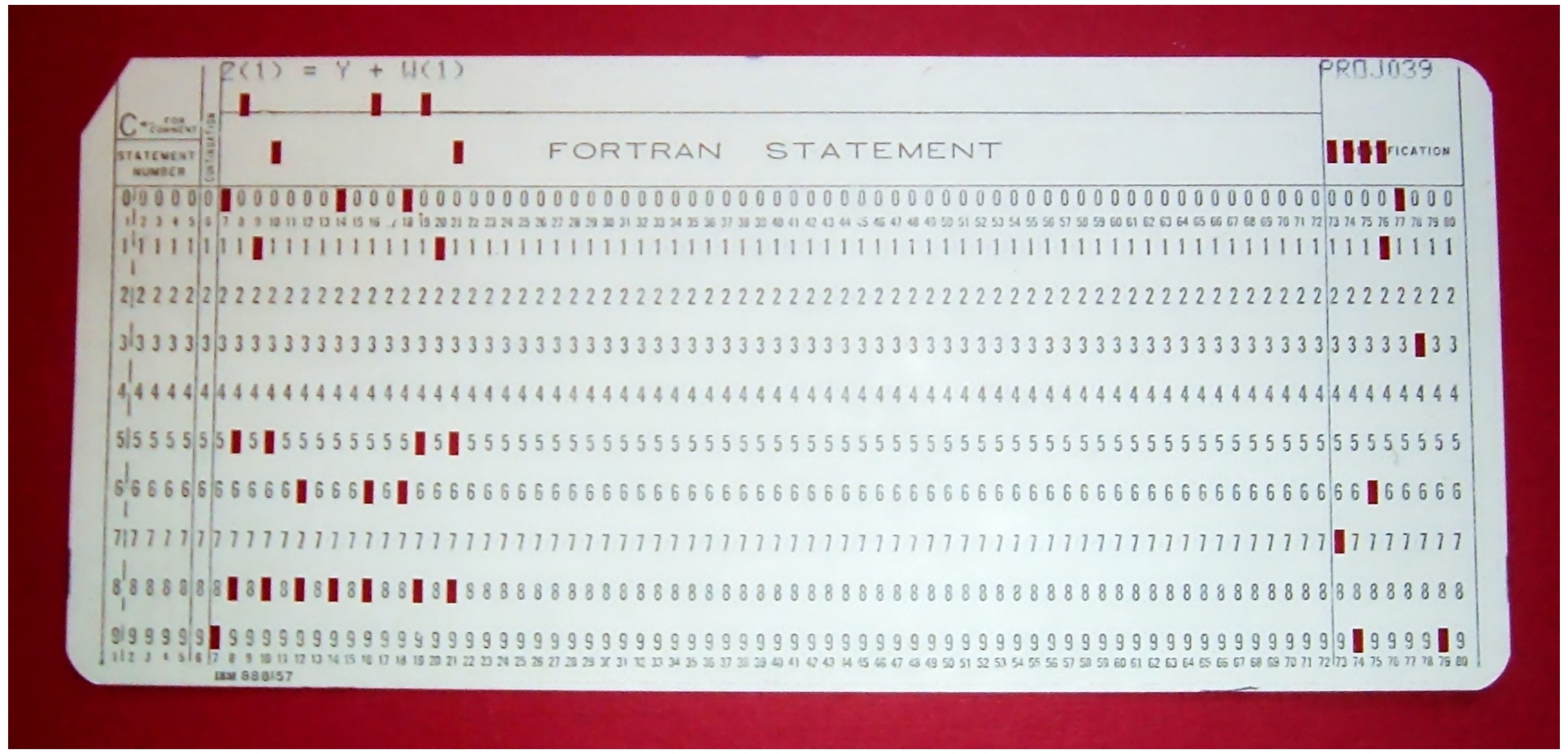
We have lots of ways to tell the difference between two different states:



Clockwise / Counterclockwise



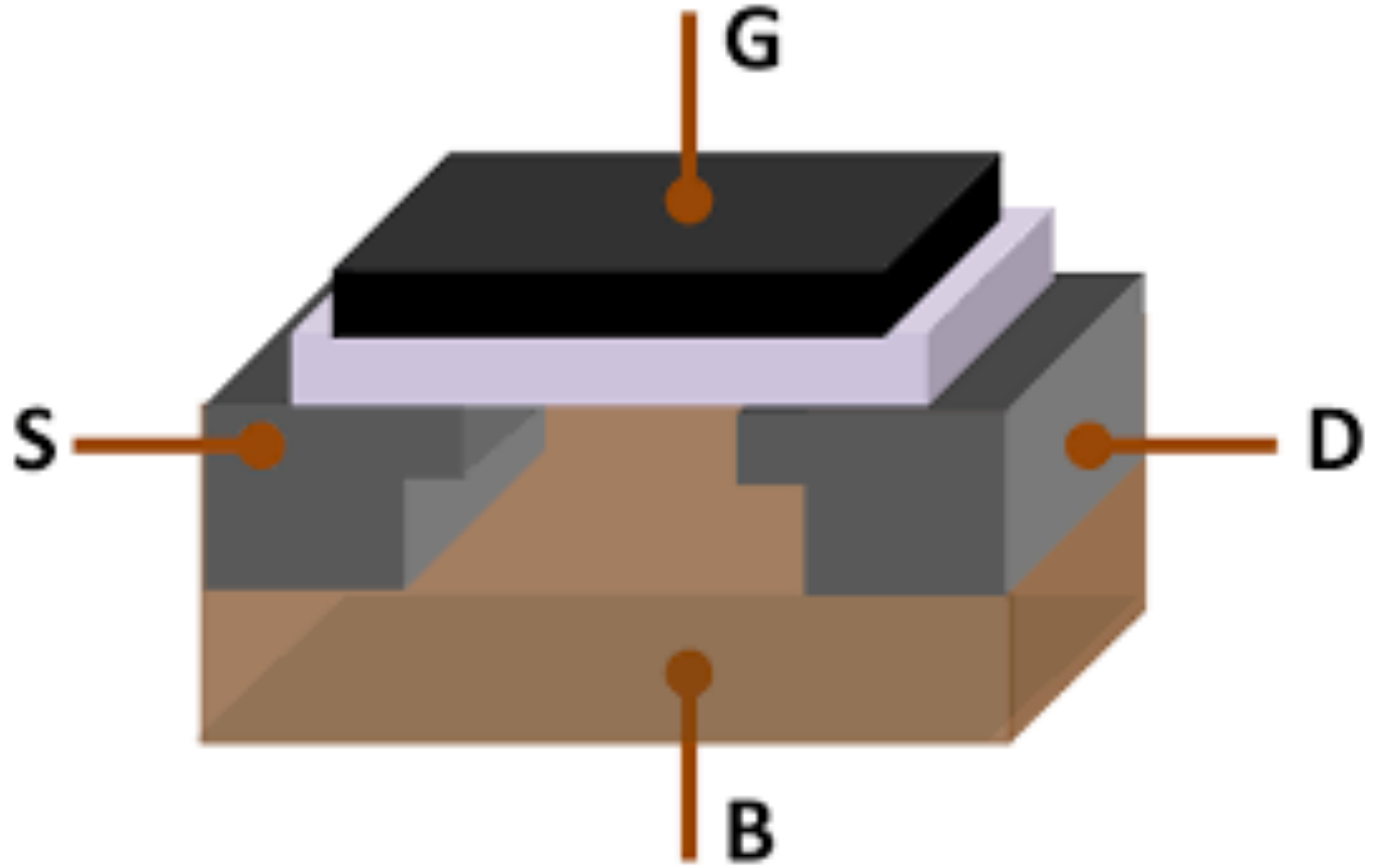
Lightbulb off / on



Punchcard hole / no hole

Computers are good at detecting "off" or "on"

Electronic computers are built using transistors



A transistor can be set up to either be "off" or "on" -- this gives us our 0 and 1!

One bit doesn't do much for us!

- We call a single on/off representation a 'bit'.
- But having one bit isn't particularly helpful!
- We only have two states we can represent with one bit!

- If we want more states, we simply combine bits together, much like we do with base 10 representation.
- If we want to combine more than ten states with base 10, we add another digit.

- Base 10 has ten digits: 0 1 2 3 4 5 6 7 8 9
- We can represent up to ten numbers with one digit in base 10
- If we want to represent more numbers, we add more digits: 10 11 12 13 14 ...

- Base 2 is the same. We can represent two numbers with one digit: 0 or 1
- To represent more numbers, we add more digits! 10 11 100 101 110 ...

Combinations of bits can represent everything

We can encode anything we want with bits. E.g., the ASCII character set.

ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(	0010 1000
N	0100 1110	l	0110 1100	)	0010 1001
				space	0010 0000

CS107: Three Number Representations

Unsigned Integers: positive integers and zero only
Ex. 0, 1, 2, ..., 74629, 99999999

Signed Integers: negative, positive, and zero integers only
Ex. 0, 1, 2, ..., 74629, 99999999
(represented in "two's complement")

Floating Point Numbers: a base-2 representation of scientific notation, for real numbers
Ex. 0.0, 0.1, -12.2, 4.87563×10^{3}, -1.00005×10^{-12}

Computers use a limited number of bits for numbers

Let's write a little program...

Computers use a limited number of bits for numbers

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

```
$ gcc -g -O0 mult-test.c -o mult-test
$ ./mult-test
-884901888
$
```

Computers use a limited number of bits for numbers

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

Recall that in base 10, you can represent: 10 numbers with one digit (0 - 9),
100 numbers with two digits (00 - 99),
1000 numbers with three digits (000 - 999)

I.e., with *n* digits, you can represent up to 10^n numbers.

In base 2, you can represent:
2 numbers with one digit (0 - 1)
4 numbers with two digits (00 - 11)
8 numbers with three digits (000 - 111)

I.e., with *n* digits, you can represent up to 2^n numbers

The C `int` type is a "32-bit" number, meaning it uses 32 digits. That means we can represent up to 2^{32} numbers.

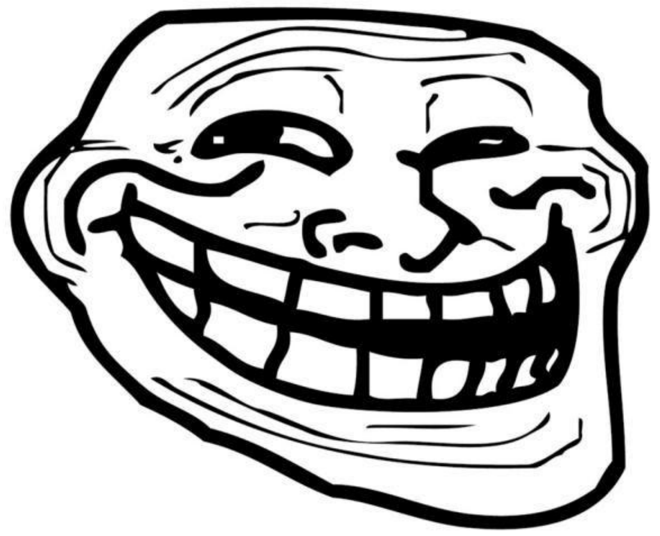
Computers use a limited number of bits for numbers

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

```
$ gcc -g -O0 mult-test.c -o mult-test
$ ./mult-test
-884901888
$
```

2^{32} = 4,294,967,296

200 * 300 * 400 * 500 = 12,000,000,000



problem?

Turns out it is worse -- `int`s are signed, meaning that the largest positive number is

$(2^{32} / 2) - 1 =$

$2^{31} - 1 = 2,147,483,647$

Computers use a limited number of bits for numbers

```c
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

```
$ gcc -g -O0 mult-test.c -o mult-
test
$ ./mult-test
-884901888
$
```

The good news: all of the following produce the same (wrong) answer:

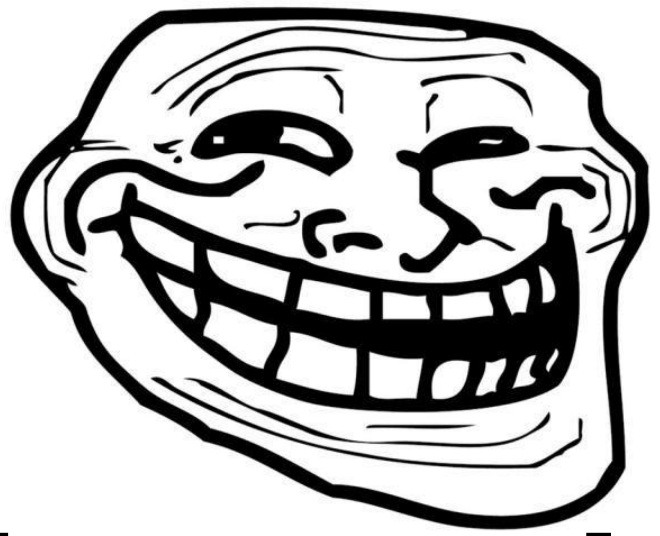
```
(500 * 400) * (300 * 200)
((500 * 400) * 300) * 200
((200 * 500) * 300) * 400
400 * (200 * (300 * 500))
```

Let's look at a different program

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    float a = 3.14;
    float b = 1e20;

    printf("(3.14 + 1e20) - 1e20 = %f\n", (a + b) - b);
    printf("3.14 + (1e20 - 1e20) = %f\n", a + (b - b));

    return 0;
}
```

```
$ gcc -g -Og -std=gnu99 float-mult-
test.c -o float-mult-test
```

```
$ ./float-mult-test.c
(3.14 + 1e20) - 1e20 = 0.000000
3.14 + (1e20 - 1e20) = 3.140000
$
```



bigger problem!

Let's look at a different program

```
$ gcc -g -O0 mult-test.c -o mult-
test
$ ./mult-test
-884901888
$
```

```
$ gcc -g -Og -std=gnu99 float-mult-
test.c -o float-mult-test

$ ./float-mult-test.c
(3.14 + 1e20) - 1e20 = 0.000000
3.14 + (1e20 - 1e20) = 3.140000
$
```

Both C and C++ have specific representations of numbers that allow for these kinds of bugs.

Information Storage

Information Storage

In C, everything can be thought of as a block of 8 bits

Information Storage

In C, everything can be thought of as a block of 8 bits called a "byte"

Information Storage

We will discuss manipulating bytes on a bit-by-bit level, but we won't be able to consider an individual bit on its own.

In a computer, the memory system is simply a large array of bytes (sound familiar, from CS106B?)

values (ints):	7	2	8	3	14	99	-6	3	45	11
address (decimal):	200d	204d	208d	212d	216d	220d	224d	228d	232d	236d
address (hex):	0xc8	0xcc	0xd0	0xd4	0xd8	0xdc	0xe0	0xe4	0xe8	0xec

Each address (a pointer!) represents the next byte in memory.

E.g., address 0 is a byte, then address 1 is the next full byte, etc.

Again: you can't address a bit. You must address at the byte level.

Byte Range

Because a byte is made up of 8 bits, we can represent the range of a byte as follows:

`00000000` to `11111111`

This range is 0 to 255 in decimal.

But, neither binary nor decimal is particularly convenient to write out bytes (binary is too long, and decimal isn't numerically friendly for byte representation)

So, we use "hexadecimal," (base 16).

Hexadecimal

Hexadecimal has 16 digits, so we augment our normal 0-9 digits with six more digits: A, B, C, D, E, and F.

Figure 2.2 in the textbook shows the hex digits and their binary and decimal values:

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

- In C, we write a hexadecimal with a starting `0x`. So, you will see numbers such as `0xfa1d37b`, which means that it is a hex number.
- You should memorize the binary representations for each hex digit. One trick is to memorize A (`1010`), C (`1100`), and F (`1111`), and the others are easy to figure out.
- Let's practice some hex to binary and binary to hex conversions:

Convert: `0x173A4C` to binary.

Hexadecimal	1	7	3	A	4	C
Binary	0001	0111	0011	1010	0100	1100

`0x173A4C` is binary
`0b000101110011101001001100`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011` is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011`

is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011`
is hexadecimal **`3CADB3`**

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011`
is hexadecimal **`3CADB3`**

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Decimal to Hexadecimal

To convert from decimal to hexadecimal, you need to repeatedly divide the number in question by 16, and the remainders make up the digits of the hex number:

314156 decimal:

```
314,156 / 16 = 19,634 with 12 remainder:  C
19,634 / 16  = 1,227 with 2 remainder:    2
1,227 / 16   = 76 with 11 remainder:      B
76 / 16 = 4 with 12 remainder:            C
4 / 16 = 0 with 4 remainder:              4

Reading from bottom up: 0x4CB2C
```

Hexidecimal to Decimal

To convert from hexadecimal to decimal, multiply each of the hexadecimal digits by the appropriate power of 16:

```
0x7AF:

7 * 16^2 + 10 * 16 + 15
= 7 * 256 + 160 + 15
= 1792 + 160 + 15 = 1967
```

Let the computer do it!

Honestly, hex to decimal and vice versa are easy to let the computer handle. You can either use a search engine (Google does this automatically), or you can use a python one-liner:

4. cgregg@myth10: ~ (ssh)

```
cgregg@myth10:~$ python -c "print(hex(314156))"
0x4cb2c
cgregg@myth10:~$ python -c "print(0x7af)"
1967
cgregg@myth10:~$
```

Let the computer do it!

You can also use Python to convert to and from binary:

4. cgregg@myth10: ~ (ssh)

```
cgregg@myth10:~$ python -c "print(bin(0x173A4C))"
0b101110011101001001100
cgregg@myth10:~$ python -c "print(hex(0b1111001010110110110011))"
0x3cadb3
cgregg@myth10:~$ 
```

(but you should memorize this as it is easy and you will use it frequently)

20 Questions for Chris

In the last few minutes of class, you get to ask me 20 questions

- The questions can be about the class, about me, about computing, about philosophy, etc.
- I do reserve the right to not answer something too personal. :)

Code:

References and Advanced Reading

- **References:**
  - Tiobe Programming Index: https://www.tiobe.com/tiobe-index/
  - The C Language: https://en.wikipedia.org/wiki/C_(programming_language)
  - Kernighan and Ritchie (K&R) C: https://www.youtube.com/watch?v=de2Hsvxaf8M
  - C Standard Library: http://www.cplusplus.com/reference/clibrary/

- **Advanced Reading:**
  - After All These Years, the World is Still Powered by C Programming
  - Is C Still Relevant in the 21st Century?
  - Why Every Programmer Should Learn C