Lecture 1: Bits and Binary Operations

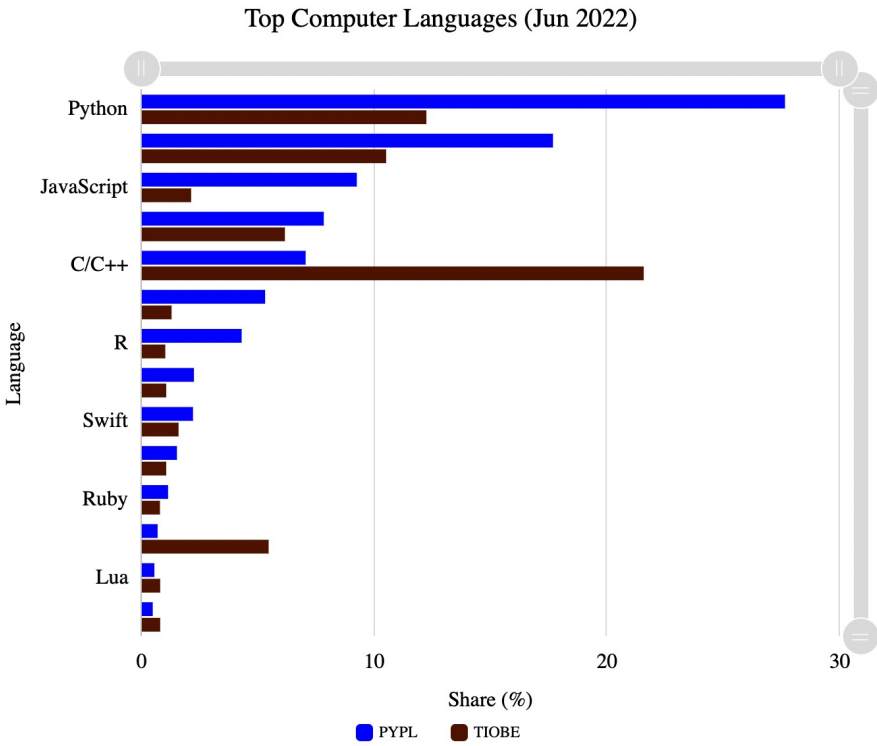
CS 105

Review: Abstraction

Review: C

- compiled, imperative language that provides low-level access to memory
- low overhead, high performance

- developed at Bell labs in the 1970s
- C (and related languages) still today



Review: Pointers

- Pointers are addresses in memory (i.e., indexes into the array of bytes)
- Most pointers declare how to interpret the value at (or starting at) that address
- Examples:

```
int myVariable = 47;
int * ptr = &myVariable;
char * ptr2 = (char *) ptr;
```

- Dereferencing pointers:

```
int var2 = *ptr
char c = *ptr2;
```

Pointer Types	x86-64
void *	8
int *	8
char *	8
⋮	8

& and * are inverses of one another

Pointer Arithmetic

```
int myVariable = 47;
int * ptr = &myVariable;
ptr += 1;

char * ptr2 = (char *) ptr;
ptr2 += 1;
```

- Location of **`ptr+k`** depends on the type of **`ptr`**
- adding 1 to a pointer **`p`** adds **`1*sizeof(*p)`** to the address

- **`array[k]`** is the same as **`*(array+k)`**

Exercise: Pointers

What does x evaluate to in each of the following?

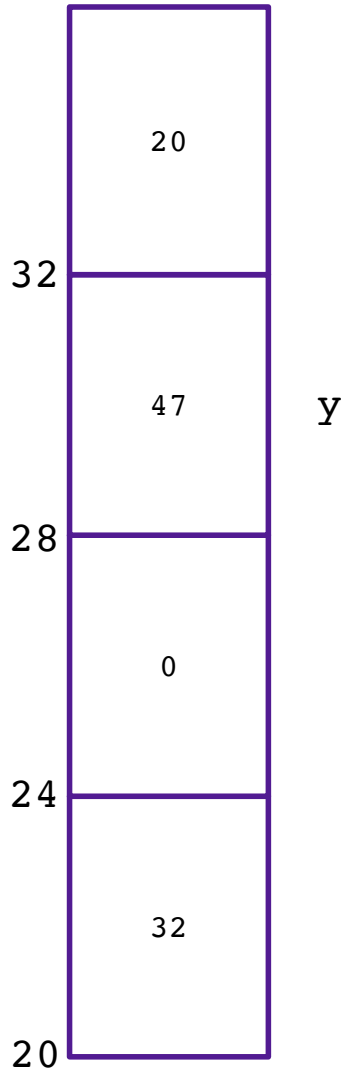
1.
```
int * ptr = 32;
x = *ptr
```

2.
```
int y = 47; // assume at 28
x = &y
```

3.
```
int * ptr = 20;
x = *(*ptr)
```

4.
```
int * ptr = 24;
x = ptr+1
```

5.
```
int * ptr = 24;
x = *(ptr+1)
```



Structs

- Heterogeneous records, like objects
- Typical linked list declaration:

```
typedef struct cell {
  int value;
  struct cell *next;
} cell_t;
```

- Usage:

```
cell_t c;
c.value = 42;
c.next = NULL;
```

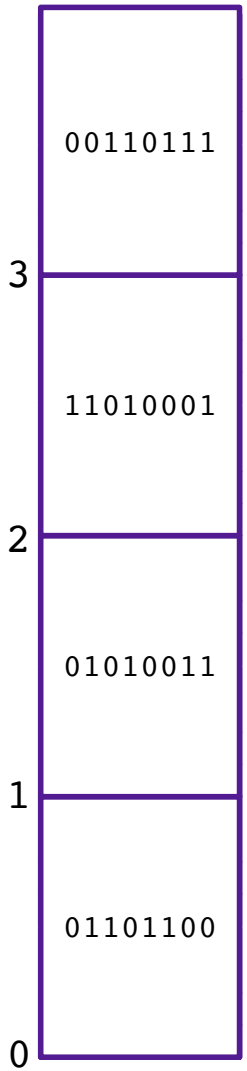
- Usage with pointers:

```
cell_t *p;
p->value = 42;
p->next = NULL;
```

`p->next` is an abbreviation for `(*p).next`

Review: Bytes and Memory

- **Memory** is an array of ~~bits~~ bytes
- A **byte** is a unit of eight bits
- An index into the array is an **address**, **location**, or **pointer**
  - Often expressed in hexadecimal
- We speak of the *value* in memory at an address
  - The value may be a single byte …
  - … or a multi-byte quantity starting at that address



Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic---encode “True” as 1 and “False” as 0

And

&	0	1
0	0	0
1	0	1

Or

|	0	1
0	0	1
1	1	1

Not

~	
0	1
1	0

Exclusive-Or (Xor)

^	0	1
0	0	1
1	1	0

- How does this map to set operations?

Exercise: Boolean Operations

- Evaluate each of the following expressions
  1. `1 | (~1)`
  2. `~( 1 | 1)`
  3. `(~1) & 1`
  4. `~( 1 ^ 1)`

General Boolean algebras

- Bitwise operations on bytes

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

Exercise: Bitwise Operations

- Assume: `a = 01101100, b = 10101010`

- What are the results of evaluating the following Boolean operations?

  - `~a`
  - `~b`
  - `a & b`
  - `a | b`
  - `a ^ b`

Bitwise vs Logical Operations in C

- Bitwise Operators &, |, ~, ^
  - View arguments as bit vectors
  - operations applied bit-wise in parallel

- Logical Operators &&, ||, !
  - View 0 as “False”
  - View anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

Exercise: Bitwise vs Logical Operations

- `~01000001`
- `~00000000`
- `~~01000001`

- `!01000001`
- `!00000000`
- `!!01000001`

- `01101001 & 01010101`
- `01101001 | 01010101`

- `01101001 && 01010101`
- `01101001 || 01010101`

Bit Shifting

- Left Shift: `x << y`
  - Shift bit-vector `x` left `y` positions
  - Throw away extra bits on left
  - Fill with 0's on right

Undefined Behavior if you shift amount < 0 or ≥ word size

- Right Shift: `x >> y`
  - Shift bit-vector `x` right `y` positions
  - Throw away extra bits on right
  - Logical shift: Fill with 0's on left
  - Arithmetic shift: Replicate most significant bit on left

Choice between logical and arithmetic depends on the type of data

Example: Bit Shifting

- `01101001 << 4` `10010000`
- `01101001` $>>_l$ `2` `00011010`
- `01101001` $>>_a$ `4` `00000110`

Exercise : Bit Shifting

- `10101010 << 4`
- $10101010 >>_l 4$
- $10101010 >>_a 4$

Bits and Bytes Require Interpretation

10001100 00001100 10101100 00000000

might be interpreted as

- The integer 3,485,745
- A floating point number close to 4.884569×10^{-39}
- The string “105”
- A portion of an image or video
- An address in memory

Information is Bits + Context

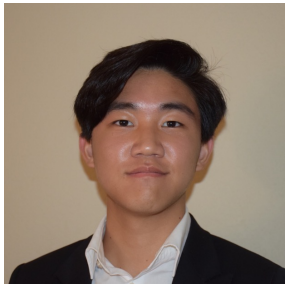
LOGISTICS

Course staff

Prof. Eleanor Birrell
Edmunds 221

Research in security and privacy
OH: M 7-9pm, T 2-4pm

Claire LeBlanc



Josh Yum

Pei Qin

Tonya Chivandire

Ziang Xue

The Course in a Nutshell

- Textbooks (Optional)
  - Bryant and O'Halloran, *Computer Systems: A Programmer's Perspective*, **third edition,** Pearson, 2016 (Optional)
  - Arpaci-Dusseau and Arpaci-Dusseau. Operating Systems: Three Easy Pieces (Optional, free online)
- Classes
  - Monday and Wednesday, 11am – 12:15pm in Edmunds 101
- Labs
  - Wednesdays 7-8:15 in Edmunds 229/219

Mentor Session Schedule (Edmunds 227)						
Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
4-6pm 7-9pm*	2-4pm* 7-9pm	LAB	7-9pm	1-3pm	2-4pm	3-5pm

Grading

- Assignments
  - Introduced during labs, Due Tuesdays at 11:59pm
  - Tremendous fun, work in pairs
  - must complete them all
  - Thirteen late days
- Check-ins
  - one-question exams at the start of lab next week
  - graded "Got it" / "Not yet"
  - Can improve from "Not yet" to "Got it" via one-on-one meeting
  - no limit on number of attempts to improve grade
  - Extra chance checkpoints
- Grades
  - Must successfully complete all the assignments
  - Beyond that, grade determined by the number of "Got it" topics

Course website

https://www.cs.pomona.edu/classes/cs105

- All information is on the course website
- All course materials get posted on the course website
- Links from the course page:
  - Slack (#cs105-2023sp), for questions and discussion
  - Gradescope, for submitting assignments and seeing grades
  - Additional resources