Lecture 1: Bits and Binary Operations

CS 105

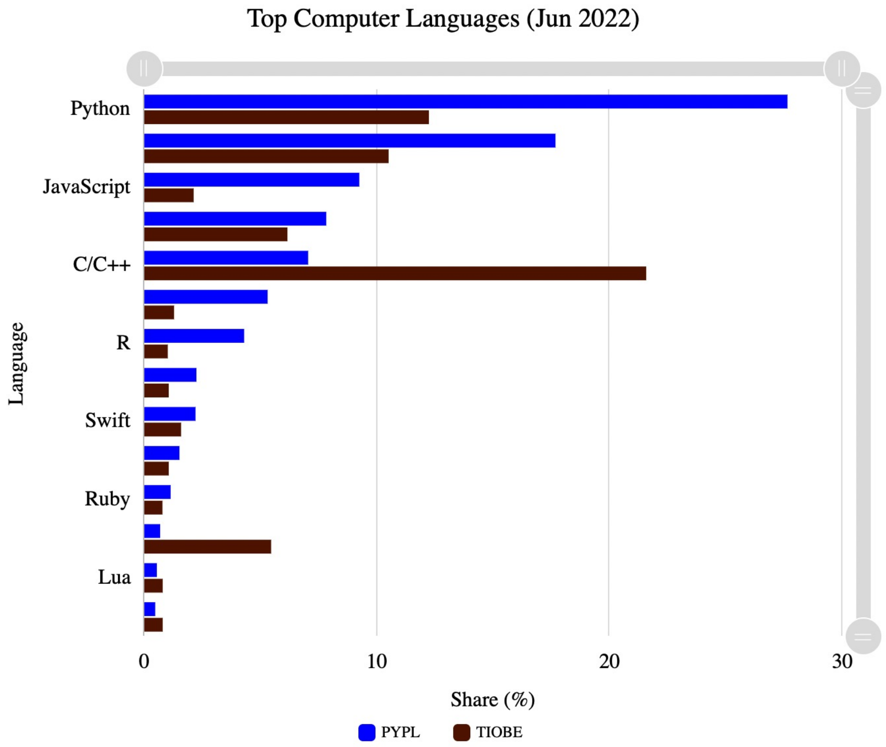
Fall 2023

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

- Example:

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

- Dereferencing pointers:

```
int var2 = *ptr
```

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

& is an "address of" operator
* is a "value at" operator

& and * are inverses of one another

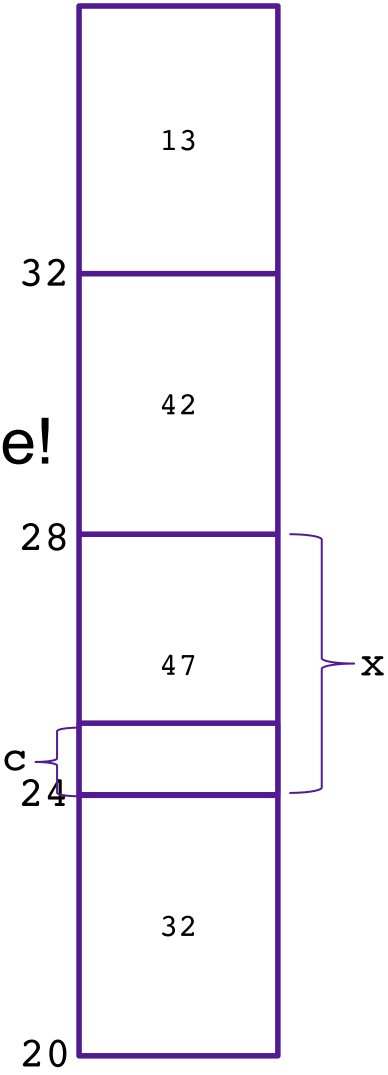
Casting between Pointer Types

- You can cast values between different types
- This includes between different pointer types!

- Doesn't change value of address
- Does change what you get when you dereference!

- Example:

```
int x = 47; // assume allocated at address 24
int* ptr = &x; // ptr == 24
char* ptr2 = (char*) ptr; // ptr2 == 24
int y = *ptr; // y == 47
char c = *ptr2; // c == ??
```



Review: Arrays

- Contiguous block of memory
- Random access by index
  - Indices start at zero
- Declaring an array:

```
int array1[5]; // array of 5 ints named array1
char array2[47]; // array of 47 chars named array2
int array3[7][4]; // two dimensional array named array3
```

- Accessing an array:

```
int x = array1[0];
```

- Arrays are pointers!
  - The array variable stores the address of the first element in the array
  - Strings are arrays of characters -> strings are char*s

Pointer Arithmetic

```
char* ptr = &my_char;   // assume ptr == 32
int* ptr2 = (int*) ptr; // ptr2 == 32

ptr += 1;               // ptr == 33
ptr2 += 1;              // ptr2 == 36
```

- 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 1

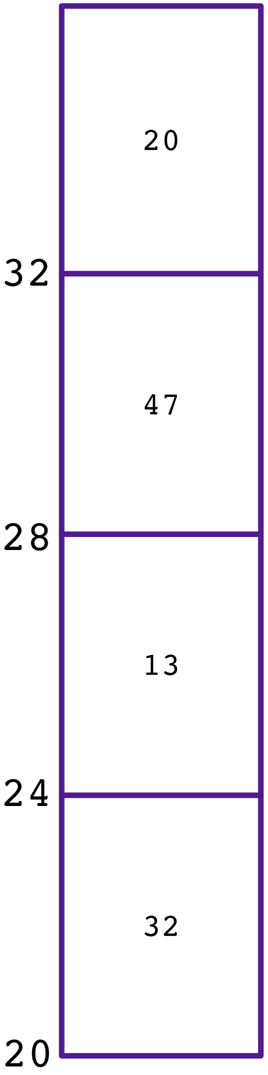
What does x evaluate to in each of the following?

1.
```
int* ptr = 20;
int* x = ptr+2;
```

2.
```
int* ptr = 20;
int x = *(ptr+2)
```

3.
```
char* ptr = 20;
char* x = ptr+2;
```

4.
```
char* ptr = 20;
int x = *((int*)(ptr + 4));
```



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;
```

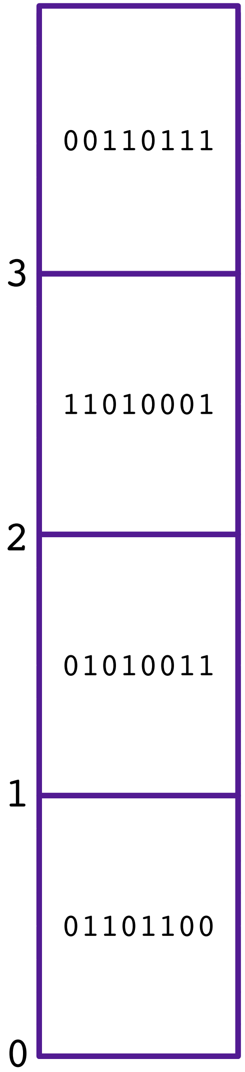
- Usage with pointers:

```
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



00110111

3

11010001

2

01010011

1

01101100

0

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 2: 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 3: 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 4: 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 5: 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 x 10^{-39}
- The string “105”
- A portion of an image or video
- An address in memory

Information is Bits + Context