CS 449: Lab 2

adapted from CMU 15-213 recitation 2

Announcements

My email: wel104@pitt.edu

Office hour: Mon/Wed 2:30-4:30

Data lab's deadline is extended

Understanding bit-wise operator

- Output 1 if exactly one 1?
- Preserve 1/0? How to choose the mask?
- Extract MSD? ^

&	0	1
0	0	0
1	0	1

← **AND** (&) outputs a 1 only when both input bits are 1.

OR (|) outputs a 1 when either input bit is 1. →

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

← **XOR** (^) outputs a 1 when either input is *exclusively* 1.

NOT (~) outputs the opposite of its input. →

~	
0	1
1	0

Great Realities

1

Ints are not integers, floats are not reals!

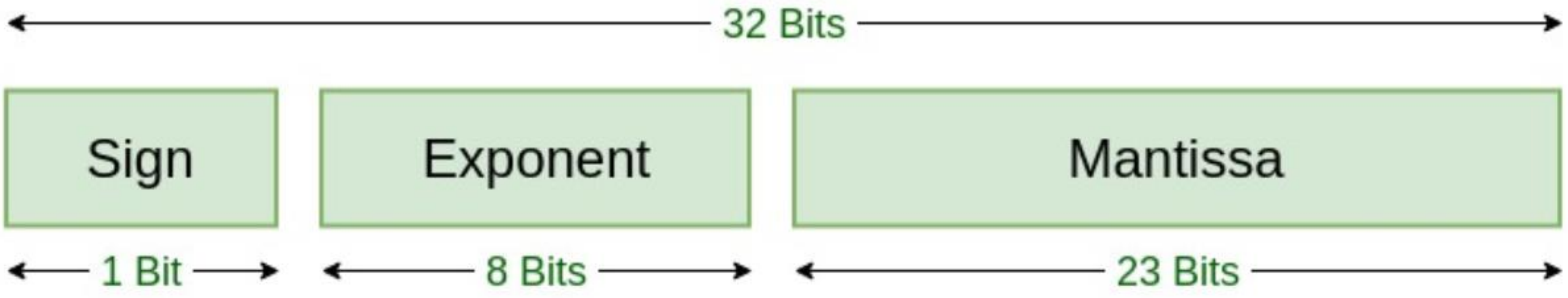
2

Although you're not likely to write assembly (cs 447), it's good for understanding machine-level executions

3

Don't feel bad if you think data lab is hard, this is still a programming class

Single-precision



Single Precision
IEEE 754 Floating-Point Standard

32.2 = ?

32 = 2*2*2*2*2 + 0

.2*2 = 0.4 ...*2 = 0.8 ...*2 = (1).6 ...*2 = (1).2 ...*2 = 0.4

100000.001100110011... 100000 = 1.00000*2^5

5 + 2^8 - 1= 132 = 10000100<- we get exponent

32.2 > 0 -> we get sign = 0

Mantissa = 000000011...

0 10000100 00000001100110011001100

24.0 = ?

- "Both the argument and result are passed as unsigned int's, but they are to be interpreted as the bit-level representations of single-precision floating point values"
- 11000 = 1.1000 * 2 to the ? What operator does this remind you of?
- Recall: use fshow() to help you…or go through the basics

Power-of-2 with SHIFT: an optimization of runtime

Multiplication

- U << k gives u*2^k

Division

- Unsigned (logical shift) U >> k gives floor(u/2^k)
- ~~Signed (arithmetic shift) X >> k gives RoundDown(x/2^k)~~

What could be wrong here?

What about addition/subtraction?

A whole area of modular arithmetic algorithms here...

Modular arithmetic (machine implementation), overflow

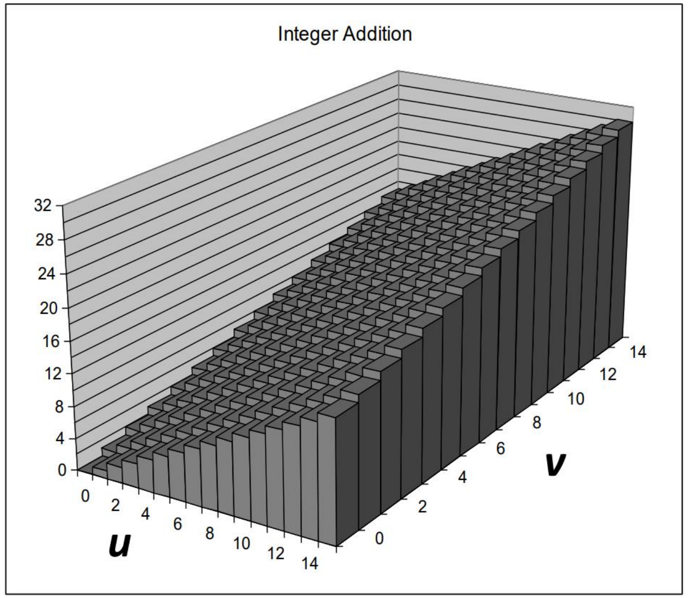
Visualizing Add_w(u,v) = u+((2^w)-v) mod 2^w

2^(w+1) ---overflow-→ 2^w

=u-v mod 2^w ← Uadd_w(u,v) = (u+v) mod 2^w

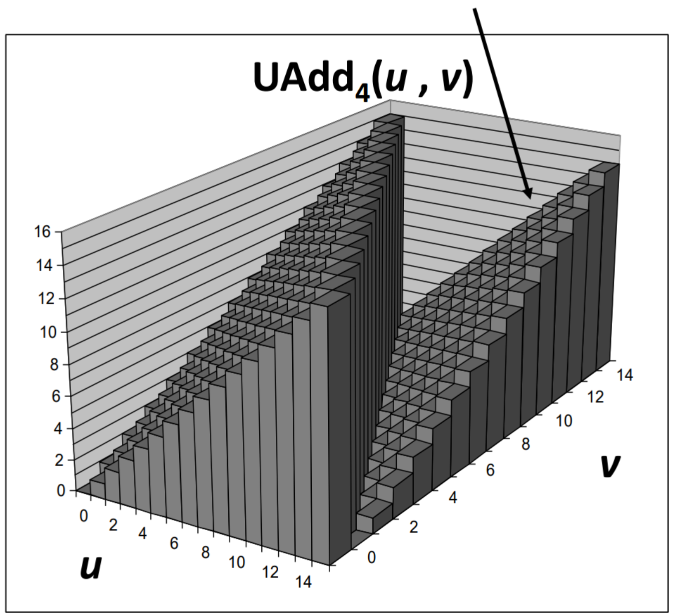
$\mathbf{Add_4}(\boldsymbol{u}\ ,\ \boldsymbol{v})$

2^5 = 32



Overflow

2^4 = 16



Memory

- A 1-dimensional array of BYTES
- Array index = address of the byte
- More than one byte? What does it look like?

Example

Computer Address	Computer Content	Programmers Name	Programmers Type	Programmers Value
90000000	00	sum	int (4 bytes)	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFFF ($4.45015E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

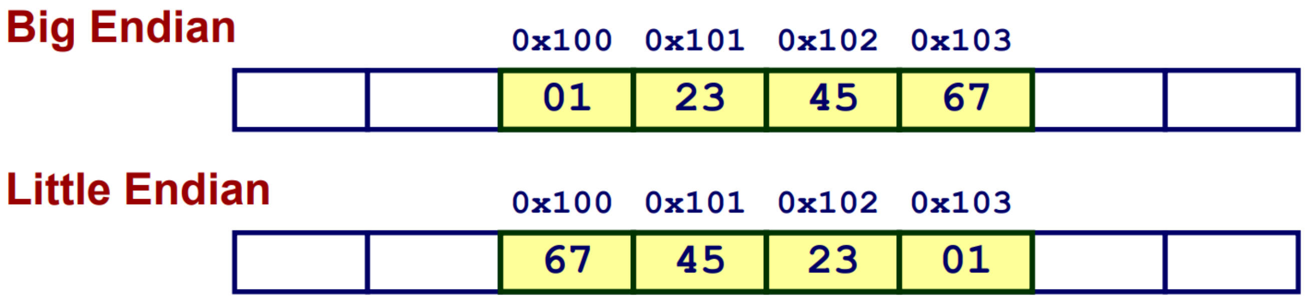
Note: All numbers in hexadecimal

Try it yourself! -- sizeof( )

Char/unsigned char	Int/unsigned int	Short/unsigned short	Long/unsigned long
Double...Is there unsigned double?	Float...Is there unsigned float?	Pointer*	Even struct, Union, enum!

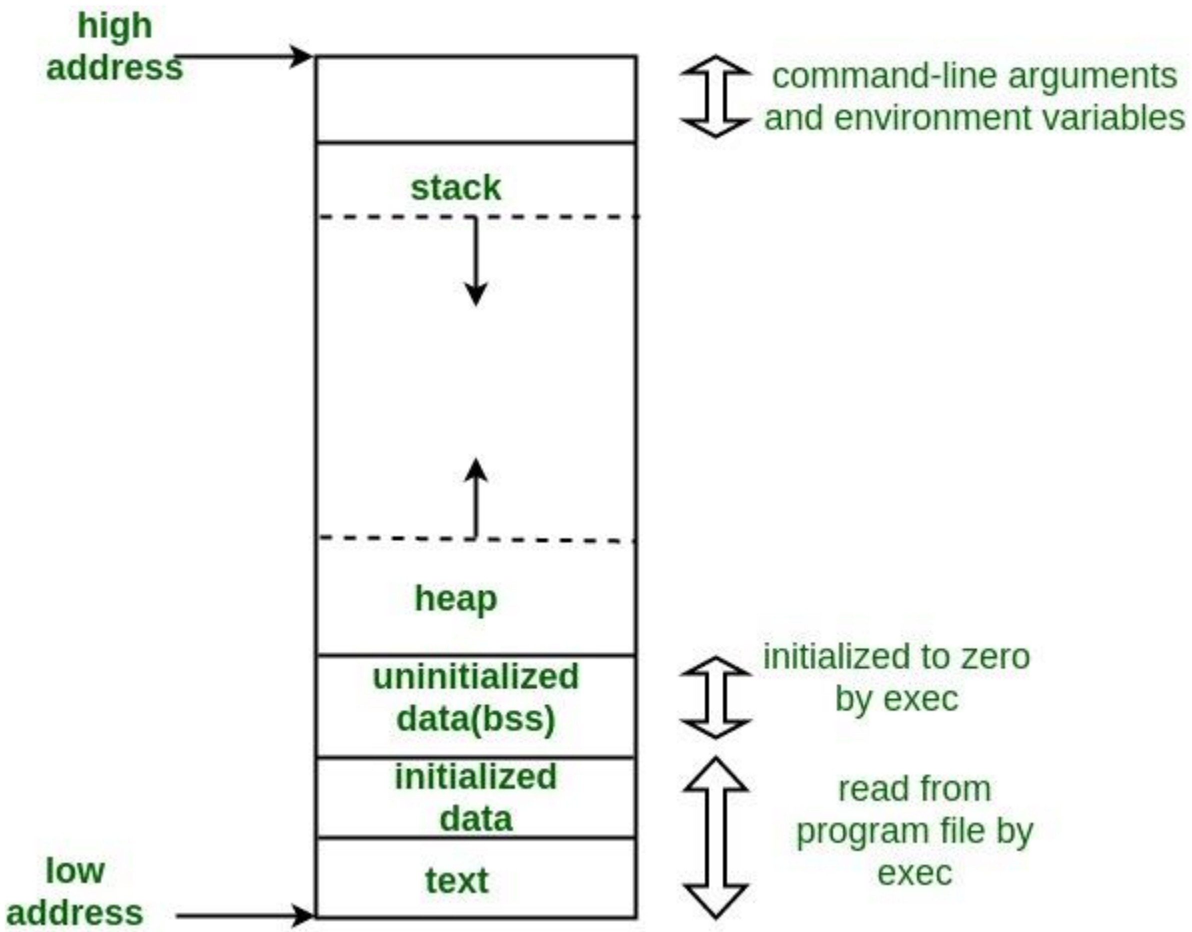
Memory (cont)

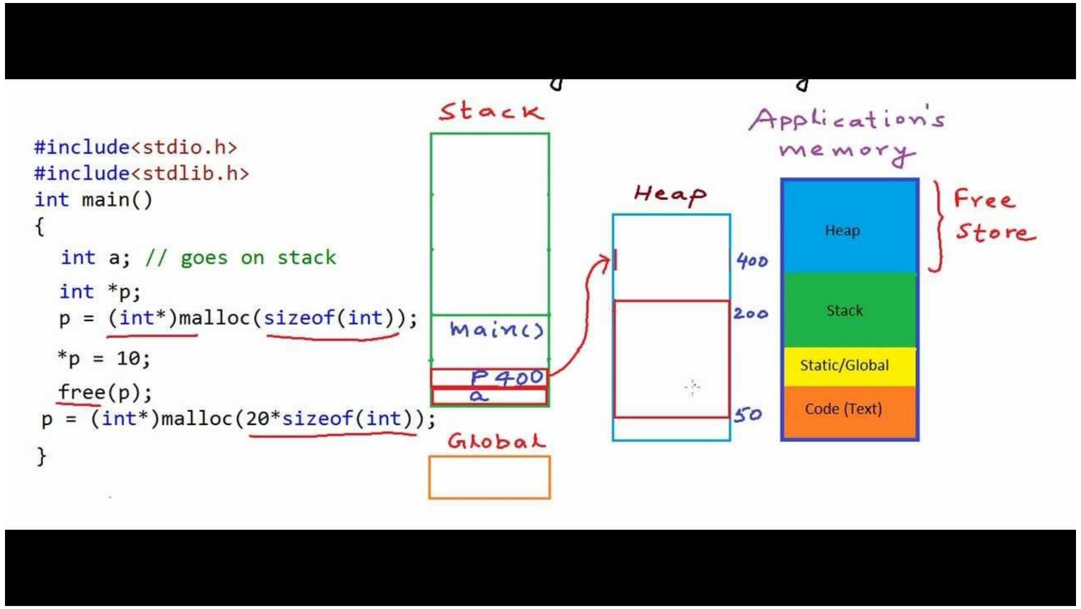
- Remember 447? (Big Endian, little endian)



- Programs refer to data by address
- System provide "private" address space to processes*

Memory diagram (more on this later in the semester)





Pointers: & and *

- Remember...memory has addresses
- Variable to hold address
- &: reference operator
- *: dereference operator
- Show me some code
  - https://colab.research.google.com/drive/1Bi0yKj4ueKj7frQ9ewCKhGehhhhQihv2
- Wait, what about strings?

Exercise:

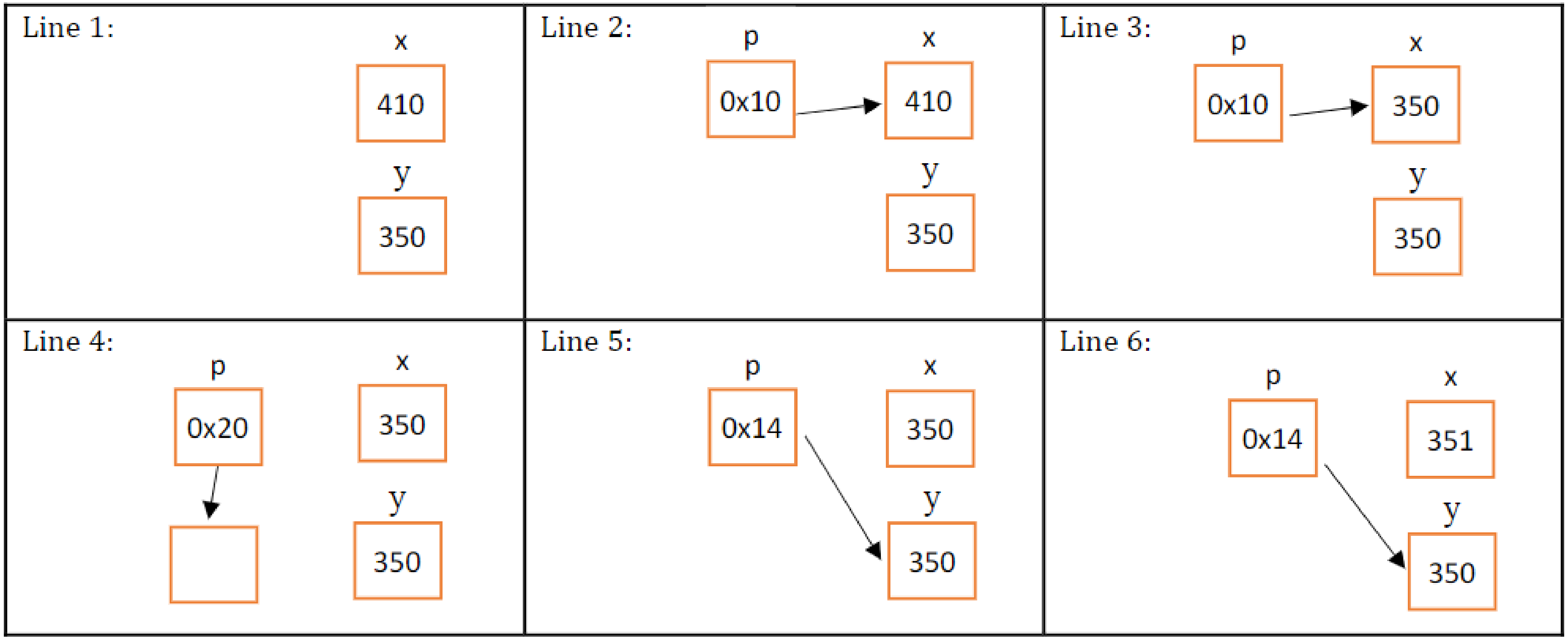
Draw out the memory diagram after sequential execution of each of the lines below:

```
int main(int argc, char **argv) {
  int x = 410, y = 350;   // assume &x = 0x10, &y = 0x14
  int *p = &x;            // p is a pointer to an integer
  *p = y;
  p = p + 4;
  p = &y;
  x = *p + 1;
}
```

Line 1:	Line 2:	Line 3:
Line 4:	**Line 5:**	**Line 6:**

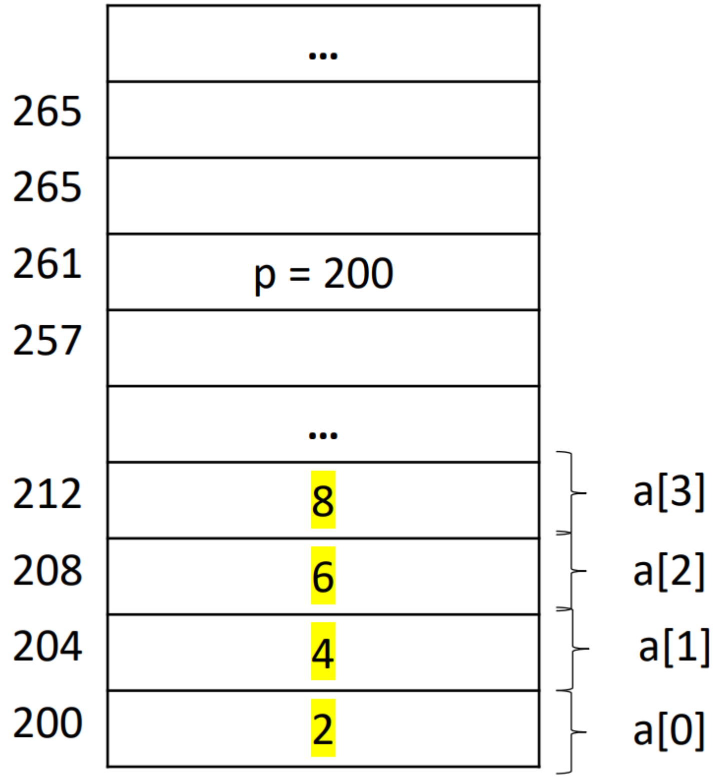
Draw out the memory diagram after sequential execution of each of the lines below:

```c
int main(int argc, char **argv) {
  int x = 410, y = 350;   // assume &x = 0x10, &y = 0x14
  int *p = &x;            // p is a pointer to an integer
  *p = y;
  p = p + 4;
  p = &y;
  x = *p + 1;
}
```



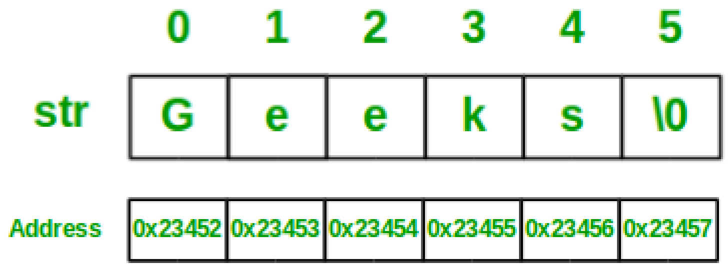
Arrays (More examples on this…maybe next recitation)

```
int a[] = {2, 4, 6, 8};

int *p = a; //Equivalent to *p = &a[0]

printf(“%d , %d”, (p+1), *(p+1)); //204, 4
printf(“%d , %d”, (a+1), *(a+1)); //204, 4
printf(“%d , %d”, &a[1], a[1]); //204, 4
```

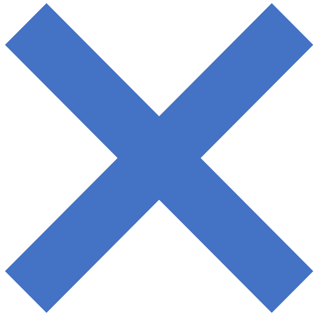


"String" as "Array"

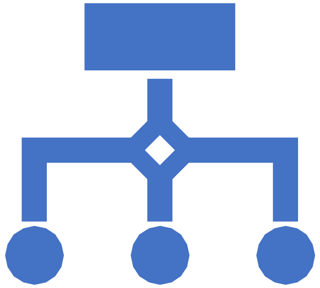
- Represented by "array" of chars
- Each char encoded in ASCII format
  - Need to end with \0: C doesn't know when a string ends
    - If not...warning: you don't know what will happen!
- But don't get too comfortable saying "array" like you are still coding in Java...



Arrays ain't real



DO NOT PASS ARRAY AS PARAMETERS



Use POINTERS (in fact, array gets converted to pointer if you are determined to break the rules...)

Classic examples

Next hw (I don't have it yet): Start early! Ask questions!