ECE/CS 250
Computer Architecture

Fall 2023

From C to Binary

John Board

Duke University

Slides are derived from work by
Daniel J. Sorin (Duke), Andrew Hilton (Duke), Alvy Lebeck (Duke),
Benjamin Lee (Duke), John Board (Duke), Amir Roth (Penn)

Also contains material adapted from CSC230: C and Software Tools developed by the NC State Computer Science Faculty

Outline

- Previously:
  - Computer is machine that does what we tell it to do
- Next:
  - How do we tell computers what to do?
  - How do we represent data objects in binary?
  - How do we represent data locations in binary?

If I get out my bit microscope...

- ...and look at 32 consecutive bits of memory, I might find

00000000 00000000 00000000 00100000

If I get out my bit microscope...

- ...and look at 32 consecutive bits of memory, I might find

00000000 00000000 00000000 00100000

What does this mean?

- The unsigned integer 32
- The signed integer +32
- A single precision floating point number +4.484155e-44
- The MIPS assembly language instruction ADD $r0 $r0 $r0
- 3 ASCII NULL characters and <space>

Only the context set by your program "tells" the computer which is correct this time!!

Representing High Level Things in Binary

- Computers represent everything in binary
- Instructions are specified in binary
- Instructions must be able to describe
  - Operation types (add, subtract, shift, etc.)
  - Data objects (integers, decimals, characters, etc.)
  - Memory locations
- Example:

```
int x, y;              // Where are x and y?  How to represent an int?
bool decision;         // How do we represent a bool?  Where is it?
y = x + 7;             // How do we specify "add"?  How to represent 7?
decision=(y>18);  //  Etc.
```

Representing Operation Types

- How do we tell computer to add? Shift? Read from memory? Etc.
- Arbitrarily! ☺
- Each Instruction Set Architecture (ISA) has its own binary encodings for each operation type
- E.g., in MIPS:
  - Integer add is: 00000 010000
  - Read from memory (load) is: 010011
  - Etc.

Representing Data Types

- How do we specify an integer? A character? A floating point number? A bool? Etc.
- Same as before: binary!

- **Data and interpretation are separate:**
  - The same 32 bits might mean one thing if interpreted as an integer, but another thing if interpreted as a floating point number

Basic Data Types

Bit (bool): 0, 1

Bit String: sequence of bits of a particular length

4 bits is a nibble
char 8 bits is a byte
short 16 bits is a half-word (for MIPS32)
int/long 32 bits is a word (for MIPS32)
long long 64 bits is a double-word (for MIPS32)
128 bits is a quad-word (for MIPS32)

What is a **word**?
The standard unit of manipulation for a particular system. E.g.:

- **MIPS32: 32 bits**
- **Intel x86_64 (modern): 64 bit**
- Original Nintendo: 8 bit
- Super Nintendo: 16 bit
- Intel x86 (classic): 32 bit
- Nintendo 64: 64 bit

Integers (char, short, int, long):
"2's Complement" (32-bit or 64-bit representation)

Floating Point (float, double):
float Single Precision (32-bit representation)
double Double Precision (64-bit representation)
Extended (Quad) Precision (128-bit representation)

Character (char):
char ASCII 7-bit code

All pink arrows are true for a MIPS32 and Intel x86

Basic Binary

- Advice: memorize the following
  - $2^0 = 1$
  - $2^1 = 2$
  - $2^2 = 4$
  - $2^3 = 8$
  - $2^4 = 16$
  - $2^5 = 32$
  - $2^6 = 64$
  - $2^7 = 128$
  - $2^8 = 256$
  - $2^9 = 512$
  - $2^{10} = 1024$

Bits vs things

- **If you have N bits, you can represent 2^N things.**

⇕

- **If you have T things, need CEIL($\log_2 T$) bits to pick one.**

You will have to answer questions of this form roughly a thousand times in this course – note it now!

- Exercises:
  - I have 8 bits, how many integers can I represent?
    - 2^8 = **256**
  - I need to represent 32 cache sets. How many bits do I need?
    - $\log_2 32$ = **5**
  - I have 4GB of RAM. How many bits do I need to pick one byte of it?
    - $\log_2 4G$ = **……?**

Binary metric system

- **The binary metric system:**
  - $2^{10} = 1024$.
  - This is *basically* 1000, so we can have an alternative form of metric units based on base 2.
  - 2^{10} bytes = 1024 bytes = 1kB.
    - Sometimes written as 1kiB
      (pronounced "ki<u>bi</u>byte" where the 'bi' means 'binary')
      (but nobody says "kibibyte" out loud because it sounds stupid)
  - 2^{20} bytes = 1MB, 2^{30} bytes = 1GB, 2^{40} bytes = 1TB, etc.
  - <u>Easy rule to convert between exponent and binary metric number:</u>

$$\mathbf{2^{XY} \text{ bytes} = 2^{Y} \cdot 2^{X0} \text{ bytes} = 2^{Y} \text{<X\_prefix>B}}$$

$$2^{13} \text{ bytes} = 2^{3} \text{ kB} = 8 \text{ kB}$$

$$2^{39} \text{ bytes} = 2^{9} \text{ GB} = 512 \text{ GB}$$

$$2^{05} \text{ bytes} = 2^{5} \text{ B} = 32 \text{ B}$$

From last slide:
$\log_2 4G = 32$

What does it mean to say base 10 or base 2?

- Integers in regular base 10:
  - 6253 $= 6000 + 200 + 50 + 3$
    $= 6*10^3 + 2*10^2 + 5*10^1 + 3*10^0$
    Digit Base Place

- Integers in base 2:
  - 1101 $= 1000 + 100 + 00 + 1$
    $= 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$
    $= 8 + 4 + 1$
    $= 13$
    Digit Base Place

  - 1 1 0 1

Bit 3
8's place

Bit 2
4's place

Bit 1
2's place

Bit 0
1's place

Decimal to binary using remainders

?	Quotient	Remainder
457 ÷ 2 =	228	1
228 ÷ 2 =	114	0
114 ÷ 2 =	57	0
57 ÷ 2 =	28	1
28 ÷ 2 =	14	0
14 ÷ 2 =	7	0
7 ÷ 2 =	3	1
3 ÷ 2 =	1	1
1 ÷ 2 =	0	1

111001001

Decimal to binary using comparison

111001001

Num	Compare 2^n	≥ ?
457	256	1
201	128	1
73	64	1
9	32	0
9	16	0
9	8	1
1	4	0
1	2	0
1	1	1

Hexadecimal

Hex digit	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Indicates a hex number

0xDEADBEEF

1101 1110 1010 1101 1011 1110 1110 1111

0x02468ACE

0000 0010 0100 0110 1000 1010 1100 1110

0x13579BDF

0001 0011 0101 0111 1001 1011 1101 1111

One hex digit represents 4 bits.
Two hex digits represent a byte (8 bits).

Binary to/from hexadecimal

- 0101101100100011_2 -->
- 0101 1011 0010 0011_2 -->
- 5 B 2 3_{16}

1 F 4 B_{16} -->

0001 1111 0100 1011_2 -->

0001111101001011_2

Hex digit	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

BitOps: Unary

- Bit-wise complement (~)
  - Flips every bit.

```
  ~0x0d    // (binary 00001101)
== 0xf2    // (binary 11110010)
```

Not the same as Logical NOT (!) or sign change (–)

```
char i, j1, j2, j3;
i = 0x0d;      // binary 00001101
j1 = ~i;       // binary 11110010
j2 = -i;       // binary 11110011
j3 = !i;       // binary 00000000
```

BitOps: Two Operands

- Operate bit-by-bit on operands to produce a result operand of the same length
- And (`&`): result 1 if both inputs 1, 0 otherwise
- Or (`|`): result 1 if either input 1, 0 otherwise
- Xor (`^`): result 1 if one input 1, but not both, 0 otherwise

- Useful identities (applied per-bit):
  - `x & 1 = x` *ANDing with 1 does nothing*
  - `x & 0 = 0` *ANDing with 0 gives zero*

  - `x | 0 = x` *ORing with 0 does nothing*
  - `x | 1 = 1` *ORing with 1 gives one*

  - `x ^ 0 = x` *XORing with 0 does nothing*
  - `x ^ 1 = ~x` *XORing with 1 flips the bit*

Two Operands... (cont'd)

- Examples

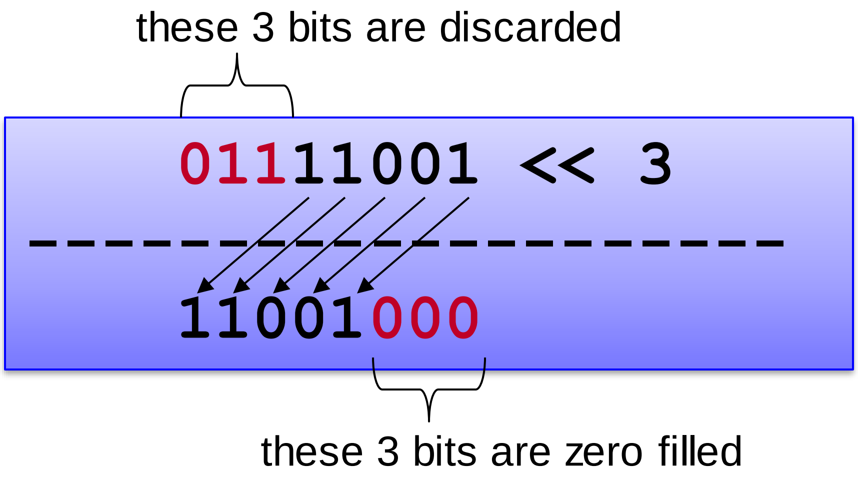
```
  0011 1000
& 1101 1110
  ---------
  0001 1000
```

```
  0011 1000
| 1101 1110
  ---------
  1111 1110
```

```
  0011 1000
^ 1101 1110
  ---------
  1110 0110
```

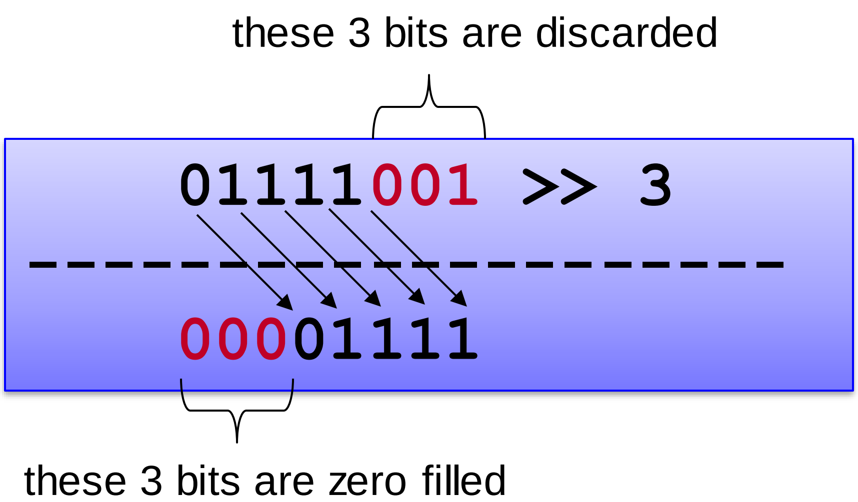
Shift Operations

- `x << y` is left (logical) shift of `x` by `y` positions
  - `x` and `y` must both be integers
  - `x` should be unsigned or positive
  - `y` leftmost bits of `x` are discarded
  - zero fill `y` bits on the right



ShiftOps... (cont'd)

- `x >> y` is right (logical) shift of `x` by `y` positions
  - `y` rightmost bits of `x` are discarded
  - zero fill `y` bits on the left



Bitwise Recipes

- Set a certain bit to 1?
  - Make a MASK with a *one* at every position you want to *set*:

    ```
    m = 0x02;   // 00000010₂
    ```

  - OR the mask with the input:

    ```
    v = 0x41;   // 01000001₂
    v |= m;     // 01000011₂
    ```

- Clear a certain bit to 0?
  - Make a MASK with a *zero* at every position you want to *clear*:

    ```
    m = 0xFD;   // 11111101₂ (could also write ~0x02)
    ```

  - AND the mask with the input:

    ```
    v = 0x27;   // 00100111₂
    v &= m;     // 00100101₂
    ```

- Get a substring of bits (such as bits 2 through 5)?
  Note: bits are numbered right-to-left starting with zero.
  - Shift the bits you want all the way to the right then AND them with an appropriate mask:

    ```
    v = 0x67;   // 01100111₂
    v >>= 2;    // 00011001₂
    v &= 0x0F;  // 00001001₂
    ```

Binary Math : Addition

- Suppose we want to add two numbers:

```
     00011101
+    00101011
-------------
```

- How do we do this?

Binary Math : Addition

- Suppose we want to add two numbers:

```
     00011101          695
+    00101011        + 232
```

- How do we do this?
  - Let’s revisit decimal addition
  - Think about the process as we do it

Binary Math : Addition

- Suppose we want to add two numbers:

```
    00011101          695
+   00101011        + 232
---------------     -----
                        7
```

- First add one's digit 5+2 = 7

Binary Math : Addition

- Suppose we want to add two numbers:

```
                                   1
     00011101                     695
+    00101011                   + 232
                                   27
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)

Binary Math : Addition

- Suppose we want to add two numbers:

```
     00011101          695
+    00101011        + 232
                       927
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)
- Last add hundred's digit 1+6+2 = 9

Binary Math : Addition

- Suppose we want to add two numbers:

```
     00011101
+    00101011
```

- Back to the binary:
- First add 1's digit 1+1 = ...?

Binary Math : Addition

- Suppose we want to add two numbers:

```
           1
      00011101
  +   00101011
  ------------
             0
```

- Back to the binary:
- First add 1's digit 1+1 = 2 (0 carry a 1)

Binary Math : Addition

- Suppose we want to add two numbers:

```
          11
     00011101
 +   00101011
 -------------
           00
```

- Back to the binary:
- First add 1's digit 1+1 = 2 (0 carry a 1)
- Then 2's digit: 1+0+1 =2 (0 carry a 1)
- You all finish it out….

Binary Math : Addition

- Suppose we want to add two numbers:

```
       111111
      00011101    = 29
  +   00101011    = 43
  ------------
      01001000    = 72
```

- Can check our work in decimal

Issues for Binary Representation of Numbers

- **Unsigned integers are easy: in 4-bit world, 0000 is 0, 1111 is 15, that's our unverse**
- **How to represent negative numbers?**
- There are many ways to represent numbers in binary
  - Binary representations are encodings → many encodings possible
  - What are the issues that we must address?
- Issue #1: Complexity of arithmetic operations
- Issue #2: Negative numbers
- Issue #3: Maximum representable number
- Choose representation that makes these issues easy for machine, even if it's not easy for humans (i.e., ECE/CS 250 students)
  - Why? Machine has to do all the work!

Sign Magnitude

- Use leftmost bit for + (0) or – (1):
- 6-bit example (1 sign bit + 5 magnitude bits):
- +17 = 010001
- -17 = 110001
- Pros:
  - Conceptually simple
  - Easy to convert
- Cons:
  - Harder to compute (add, subtract, etc) with
  - Positive and negative 0: 000000 and 100000

NOBODY DOES THIS (well, except the IEEE Floating point standard...!)

1's Complement Representation for Integers

- Use largest positive binary numbers to represent negative numbers
- To negate a number,

  invert ("not") each bit:

  0 → 1

  1 → 0
- Arg! In n bits, -(x) = 2**n – 1 – x
- Same as subtracting from n 1's
- Cons:
  - Still two 0s (yuck)
  - Still hard to compute with

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-7
1001	-6
1010	-5
1011	-4
1100	-3
1101	-2
1110	-1
1111	-0

I (jab) hate the black magic approach of "flip all the bits", we prove how all this works in 350!
NOBODY DOES THIS EITHER

2's Complement Integers

- Use large positives to represent negatives
- $(-x) = 2^n - x$
- This is 1's complement + 1
- So, to negate, **just invert bits and add 1**
  - $(2^n - 1) - x$ [flip all the bits] +1

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

6-bit examples:

$010110_2 = 22_{10}$; $101010_2 = -22_{10}$

$1_{10} = 000001_2$; $-1_{10} = 111111_2$

$0_{10} = 000000_2$; $-0_{10} = 000000_2$ → good!

EVERYBODY DOES THIS

ULTRAVIOLET INTEL! Minimum number of bits

- EVERY positive 2's complement number MUST begin with at least one leading 0, as EVERY number that begins with a 1 is known to be negative in this scheme.
- Thus, if a test question were to read, "Express the quantity +15 in 4-bit 2's complement representation" your answer would be "NOT POSSIBLE!" and even better, "BECAUSE THE LARGEST POSITIVE 4-BIT 2's COMPLEMENT NUMBER is +7"
- It is an ERROR to speak about 2's (or 1's or sign-magnitude numbers) WITHOUT declaring how big the universe is in bits!

Another way to think about 2's complement

- Regular base 10:
  - 6253 = 6000 + 200 + 50 + 3
    $= 6*10^3 + 2*10^2 + 5*10^1 + 3*10^0$
    Digit Base Place
- Unsigned base 2:
  - 1101 = 1000 + 100 + 00 + 1
    $= 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$
    = 8 + 4 +1
    = 13
    Digit Base Place
- Signed base 2:
  - 1101 = -1000 + 100 + 00 + 1
    $= 1*-2^3 + 1*2^2 + 0*2^1 + 1*2^0$
    = -8 + 4 + 1
    = -3

Alternately,
flip the bits and add 1:
1101
Flip: 0010
+1: 0011

That's 3 in binary,
so the number is indeed -3

Two's complement is like making the highest order bit apply a negative value!

Pros and Cons of 2's Complement

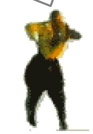
- Advantages:
  - Only one representation for 0 (unlike 1's comp): 0 = 000000
  - Addition algorithm is much easier than with sign and magnitude
    - Independent of sign bits

- Disadvantage:
  - One more negative number than positive
  - Example: 6-bit 2's complement number
    $100000_2 = -32_{10}$; but 32_{10} could not be represented

In 350 we explore the significant consequences of this inconvenience in detail!

All modern computers use 2's complement for integers

Integer ranges



- If I have an n-bit integer:
  - And it's **unsigned**, then I can represent $\{0 \;..\; 2^n - 1\}$
  - And it's **signed**, then I can represent $\{-(2^{n-1}) \;..\; 2^{n-1} - 1\}$

- Result:

Size in bits	Size in bytes	Datatype	Unsigned range	Signed range
8	1	`char`	0 .. 255	-128 .. 127
16	2	`short`	0 .. 65,535	-32,768 .. 32,767
32	4	`int`	0 .. 4,294,967,295	-2,147,483,648 .. 2,147,483,647
64	8	`long long`	0 .. 18,446,744,073,709,600,000	-9,223,372,036,854,780,000 .. 9,223,372,036,854,780,000

How to get unsigned integers in C? Just say `unsigned`:

```
int x;          // defaults to signed
unsigned int y; // explicitly unsigned
```

2's Complement Precision Extension: 0-padding or 1-padding

- Most computers today support 32-bit (int) or 64-bit integers
  - Specify 64-bit using gcc C compiler with long long
- To extend precision, use sign bit extension
  - Integer precision is number of bits used to represent a number

<u>Examples</u>

14_{10} = 001110_2 in 6-bit representation.

14_{10} = 000000001110_2 in 12-bit representation

-14_{10} = 110010_2 in 6-bit representation

-14_{10} = 111111110010_2 in 12-bit representation.

i.e. to go from n bits to n+k bits, prepend k copies of the Most Significant Bit (aka the sign bit) of n to itself!

Binary Math : Signed Addition

- Let’s look at another binary addition, now in 2's complement:

```
      01011101
  +   01101011
  ------------
```

Binary Math : Signed Addition

- What about this one:

```
      1111111
      01011101     =   93
  +   01101011     =  107
  ---------------
      11001000     =  -56
```

- But… that can't be right?
  - What do you expect for the answer?
  - What is it in 8-bit signed 2's complement?

Integer Overflow

- Answer should be 200
  - Not representable in 8-bit signed representation
  - No right answer
- This is called integer Overflow
- Real problem in programs
- How to solve?

- RULE: If you add 2 positive things and the result is negative, OVF
- If you add 2 negative things and the result is positive, OVF
- If you add a positive thing to a negative thing, NO OVF POSSIBLE!!!

Adding works for unsigned and signed

- Addition works the same way for unsigned and signed numbers. WOW!!
  - But watch out for overflow…
    (And overflow for unsigned is different than overflow for signed)

	Meaning if you assume… Signed	Unsigned
1		
0101	5	5
+ 0001	1	1

0110	6	6

	Meaning if you assume… Signed	Unsigned
1111		
1101	-3	13
+ 1111	-1	15

1100	-4	~~28~~ 12???

	Meaning if you assume… Signed	Unsigned
1		
0101	5	5
+ 0100	4	4

1001	~~9~~ -7???	9

	Meaning if you assume… Signed	Unsigned
1111		
0101	5	5
+ 1111	-1	15

0100	4	~~20~~ 4???

Subtraction

- 2's complement makes subtraction easy:
  - Remember: A - B = A + (-B)
  - And: -B = ~B + 1

    ↑ that means flip bits ("not")
  - So we just flip the bits and start with carry-in (CI) = 1
  - Later: No new circuits to subtract (re-use adder hardware!)

```
                                    1
   0110101        ->         0110101
-  1010010                +  0101101
```

What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - Speed of light ~= 3×10^8
  - Pi = 3.1415…
- Fixed number of bits limits range of integers
  - Can't represent some important numbers
  - Even something as humble as 1/3... always an approximation in decimal as well as in binary in a finite number of digits/bits
- Humans use Scientific Notation
  - 1.3×10^4

Option 1: Fixed point

- Use normal integers, but ($X*2^K$) instead of X
  - Example: 32 bit int, but use X*65536
  - 3.1415926 * 65536 = 205887
  - 0.5 * 65536 = 32768 , etc..
- Pros:
  - Addition/subtraction just like integers ("free")
- Cons:
  - Mul/div require renormalizing (divide by 64K) (ECE 458 Embedded Eystems!)
  - Range limited (no good rep for large + small)
- Can be good in specific situations
  - (representing temperatures on a digital thermometer to 0.1 degree of acuracy: range of 0-100 becomes 0.0 - 100.0, so 1001 integers from 0000 to 1000)

Can we do better?

- Think about scientific notation for a second:
- For example:

  $6.02 * 10^{23}$

- Real number, but comprised of ints:
  - 6 generally only 1 digit here
  - 02 any number here
  - 10 always 10 (base we work in)
  - 23 can be positive or negative
- Can we do something like this in binary?

Option 2: Floating Point

- How about:
  $+/- X.YYYYYY * 2^{+/-N}$

- Big numbers: large positive N
- Small numbers (<1): negative N
- Numbers near 0: small N

- This is “floating point” : most common way

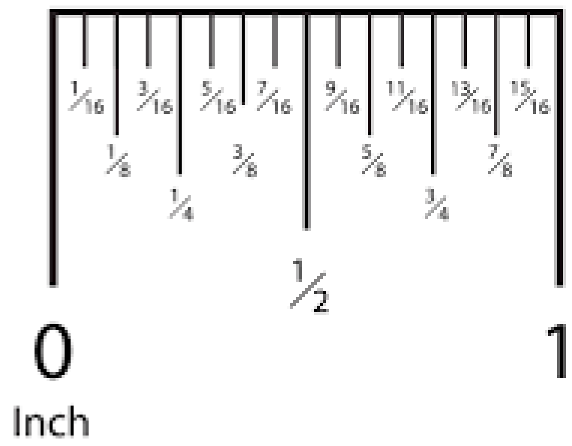
IEEE single precision floating point

- Specific format called IEEE single precision:
  +/- 1.YYYYY * $2^{(N-127)}$
- "float" in Java, C, C++,…

- Assume first bit is always 1 (saves us a bit)
- 1 sign bit (+ = 0, 1 = -)
- 8 bit biased exponent (do N-127) (Seems weird, but good reasons for this – assures smallest legal exponent is 00000000)
- Implicit 1 before *binary point*
- 23-bit *mantissa* (YYYYY)

Binary fractions

- 1.YYYY   has a binary point
  - Like a decimal point but in binary
  - After a decimal point, you have
    - tenths
    - hundredths
    - thousandths
    - …
- So after a binary point you have…
  - Halves
  - Quarters
  - Eighths
  - …



Floating point example

- Binary fraction example:
  101.101 = 4 + 1 + ½ + $^1/_8$ = 5.625
- For floating point, needs normalization:
  $1.01101 * 2^2$
- Sign is +, which = 0
- Exponent = 127 + 2 = 129 = 1000 0001
- Mantissa = 1.011 0100 0000 0000 0000 0000

31	30 … 23	22 … 0
0	1000 0001	011 0100 0000 0000 0000 0000

Can use hex to represent those bits in a less annoying way:

	0100	0000	1011	0100	0000	0000	0000	0000
0x	4	0	b	4	0	0	0	0

Floating Point Representation

Example:

What floating-point number is:

0xC1580000?

Answer

What floating-point number is

0xC1580000?

1100 0001 0101 1000 0000 0000 0000 0000

	31	30 … 23	22 … 0
x =	1	1000 0010	101 1000 0000 0000 0000 0000
	s	E	F

Sign = 1 which is negative

Exponent = (128+2)-127 = 3

Mantissa = 1.1011

$-1.1011 \times 2^3 = -1101.1 = -13.5$

Trick question

- How do you represent 0.0?
  - Why is this a trick question?
  - 0.0 = 0.00000
  - But need 1.XXXXX representation?
  - However, all "0"s would be the tiniest non-zero number we could represent: 1.0×2^{-127}
- So we solve with magic: Exponent of 0 is a special case, it is "denormalized" (all of this requires if statements in IEEE FP code!)
  - Implicit 0. instead of 1. in mantissa
  - Allows 0000….0000 to be 0
  - Helps with very small numbers near 0
- Results in +/- 0 in FP (but they are “equal”)

Other Weird FP numbers

- Exponent = 1111 1111 also not standard
  - All 0 mantissa: +/- ∞

    1/0 = +∞

    -1/0 = -∞
  - Non zero mantissa: Not a Number (NaN)

    sqrt(-42) = NaN

    0/0 = NaN

    Again, a bunch of "IF" statements in IEEE FP code!

    The actual text strings "Inf" and "NaN" will print out from C!

fp.c

```
#include <stdio.h>
int main(){
    float a,b,c,d;
    a = 1.0/1.0;
    b = 1.0/0.0;
    c = 0.0/1.0;
    d = 0.0/0.0;
    printf("1/1=%f 1/0=$f 0/1=%f 0/0=%f\n",
a,b,c,d);
}
% gcc fp.c
%./a.out
1/1=1.000000 1/0=Inf 0/1=0.000000 0/0=NaN
```

Floating Point Representation

- Double Precision Floating point:

  64-bit representation:
  - 1-bit sign
  - 11-bit (biased) exponent
  - 52-bit fraction (with implicit 1).
  - So both range and precision dramatically enhanced over float
- “double” in Java, C, C++, …

S	Exp	Mantissa
1	11-bit	52 - bit

What About Strings?

- Many important things stored as strings…
  - E.g., your name
- How should we store strings?

Standardized ASCII (American Standard Code for Information Interchange) (0-127)

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

One Interpretation of 128-255

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	169	_	185	╣	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	█	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	_
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	^{2}
141	ì	158	_	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	░	192	└	208	╨	224	α	240	≡		

Source: www.LookupTables.com

(This allowed totally sweet ASCII art in the 1990s)

Sources:

- http://roy-sac.deviantart.com/art/Cardinal-NFO-File-ASCII-35664604
- http://roy-sac.deviantart.com/art/Siege-ISO-nfo-ASCII-Logo-35940815
- http://roy-sac.deviantart.com/art/deviantART-ANSI-Logo-31556803

About those control codes… from teletype era!!! (think remote controlled typewriter) Some still used

Dec	Hx	Oct	Char	
0	0	000	NUL	(null)
1	1	001	SOH	(start of heading)
2	2	002	STX	(start of text)
3	3	003	ETX	(end of text)
4	4	004	EOT	(end of transmission)
5	5	005	ENQ	(enquiry)
6	6	006	ACK	(acknowledge)
7	7	007	BEL	(bell)
8	8	010	BS	(backspace)
9	9	011	TAB	(horizontal tab)
10	A	012	LF	(NL line feed, new line)
11	B	013	VT	(vertical tab)
12	C	014	FF	(NP form feed, new page)
13	D	015	CR	(carriage return)
14	E	016	SO	(shift out)
15	F	017	SI	(shift in)
16	10	020	DLE	(data link escape)
17	11	021	DC1	(device control 1)
18	12	022	DC2	(device control 2)
19	13	023	DC3	(device control 3)
20	14	024	DC4	(device control 4)
21	15	025	NAK	(negative acknowledge)
22	16	026	SYN	(synchronous idle)
23	17	027	ETB	(end of trans. block)
24	18	030	CAN	(cancel)
25	19	031	EM	(end of medium)
26	1A	032	SUB	(substitute)
27	1B	033	ESC	(escape)
28	1C	034	FS	(file separator)
29	1D	035	GS	(group separator)
30	1E	036	RS	(record separator)
31	1F	037	US	(unit separator)

\0 Null terminator ☺

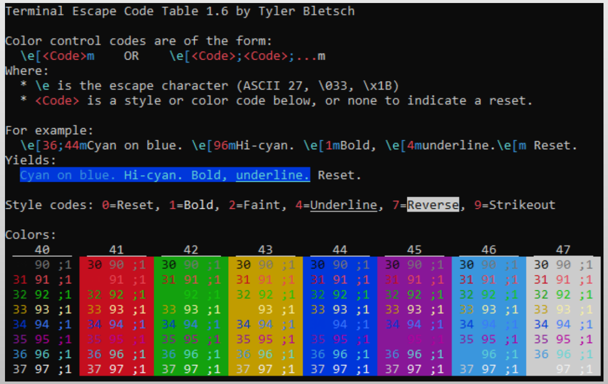
\a Make a beep or flash

Backspace key

\t Tab key

We need to talk about CR and LF…

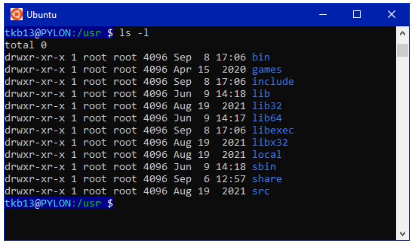
\e Used for "extended" control codes, like terminal color



(Greyed out ones almost never used)

About CR and LF

- History: first computer "displays" were modified typewriters



- CR = "Carriage return" = **`\r`** = 0x0D
  - Move typey part to the left → move cursor to left of screen
- LF = "Line feed" = **`\n`** = 0x0A
  - Move paper one line down → Move cursor one down

- Windows: "Pretend to be a typewriter"
  - Every time you press enter you get CR+LF (bytes 0D,0A)
- Linux/Mac: "You are not a typewriter"
  - Every time you press enter you get LF (byte 0A)
- **This effects ALL TEXT DOCUMENTS!!!**
  - **Not all apps cope automatically! It will bite you one day for sure!**

Outline

- Previously:
  - Computer is machine that does what we tell it to do
- Next:
  - How do we tell computers what to do?
  - How do we represent data objects in binary?
  - How do we represent data locations in binary?

Computer Memory

- Where do we put these numbers?
  - Registers [more on these later]
    - In the processor core
    - Compute directly on them
    - Few of them (~16 or 32 registers, each 32-bit or 64-bit)
  - Memory [Our focus now]
    - External to processor core
    - Load/store values to/from registers
    - Very large (multiple GB)

Memory Organization

- Memory: billions of locations…how to get the right one?
  - Each memory location has an address
  - Processor asks to read or write specific address
    - Memory, please load address 0x123400
    - Memory, please write 0xFE into address 0x8765000
  - Kind of like a giant array
    - Array of what?
      - Bytes?
      - 32-bit ints?
      - 64-bit ints?

Memory Organization

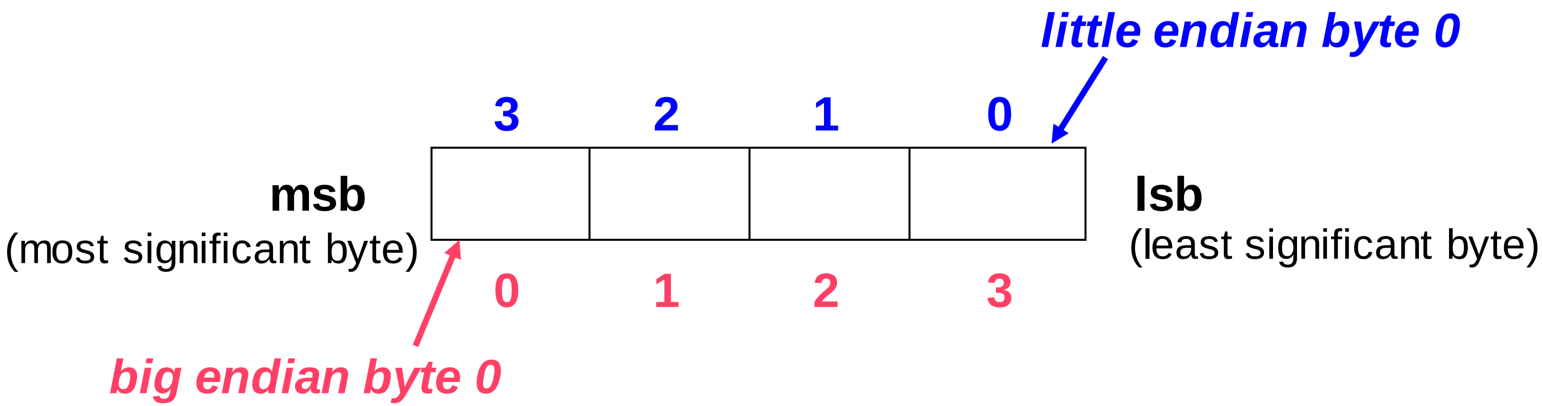
- Most systems: byte (8-bit) addressed
  - Memory is “array of bytes”
    - Each address specifies 1 byte
  - Support to load/store 8, 16, 32, 64 bit quantities
    - Byte ordering varies from system to system

- Some systems “word addressed”
  - Memory is “array of words”
    - Smaller operations “faked” in processor
  - Not very common

Word of the Day: Endianess

Byte Order

- Big Endian: byte 0 is eight most significant bits
  MIPS, IBM 360/370, Motorola 68k, Sparc, HP PA
- Little Endian: byte 0 is eight least significant bits
  Intel 80x86, DEC Vax, DEC Alpha



Program

```
X = 0x12345678; // X lives at address 0x1000
```

Memory layout on a big endian system

```
0x1000: 12
0x1001: 34
0x1002: 56
0x1003: 78
```

Memory layout on a little endian system

```
0x1000: 78
0x1001: 56
0x1002: 34
0x1003: 12
```

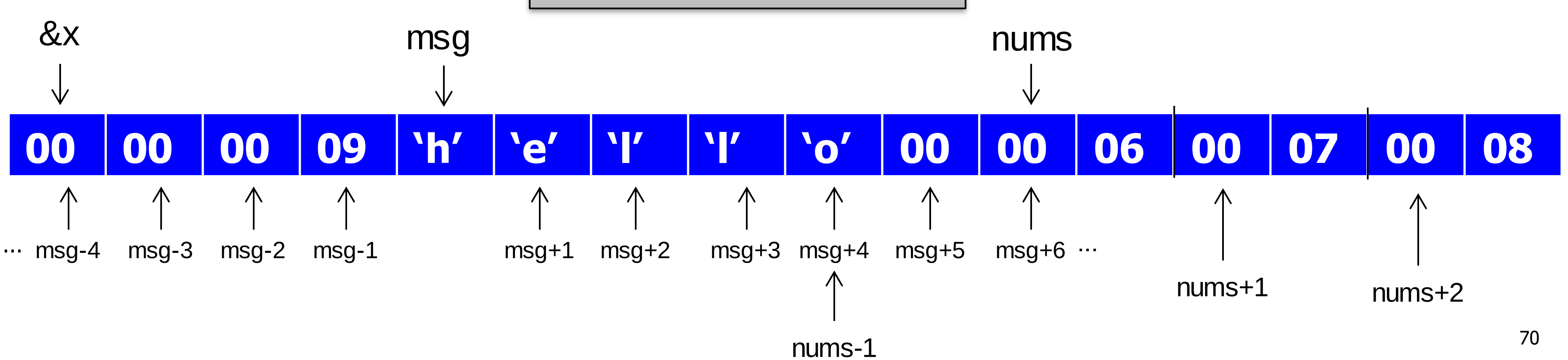
Remember this slide?

What is an array?

Big-endian

- The shocking truth:
  You've been using pointers all along!
- Every array *IS* a pointer to a block of memory
- **Pointer arithmetic:** If you add an integer N to a pointer P, you get the address of N *things* later from pointer P
  - "Thing" depends on the datatype of the P
- Can *dereference* such pointers to get what's there
  - Interpreted according to the datatype of P
  - E.g. *(nums-1) is a number related to how we represent the letter 'o'.

```
int x = 9;
char msg[] = "hello";
short nums[] = {6,7,8};
```



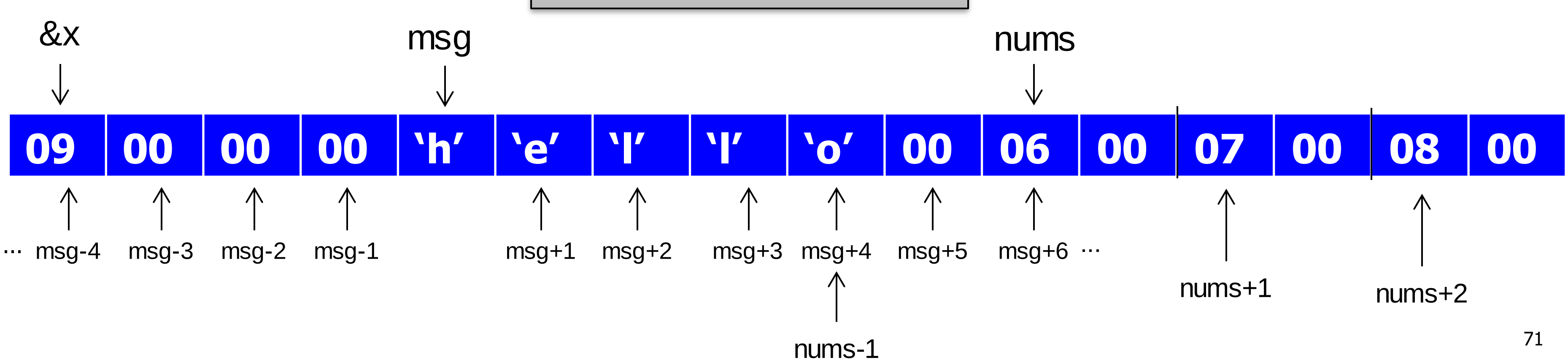
The slide could be like this…

What is an array?

Little-endian

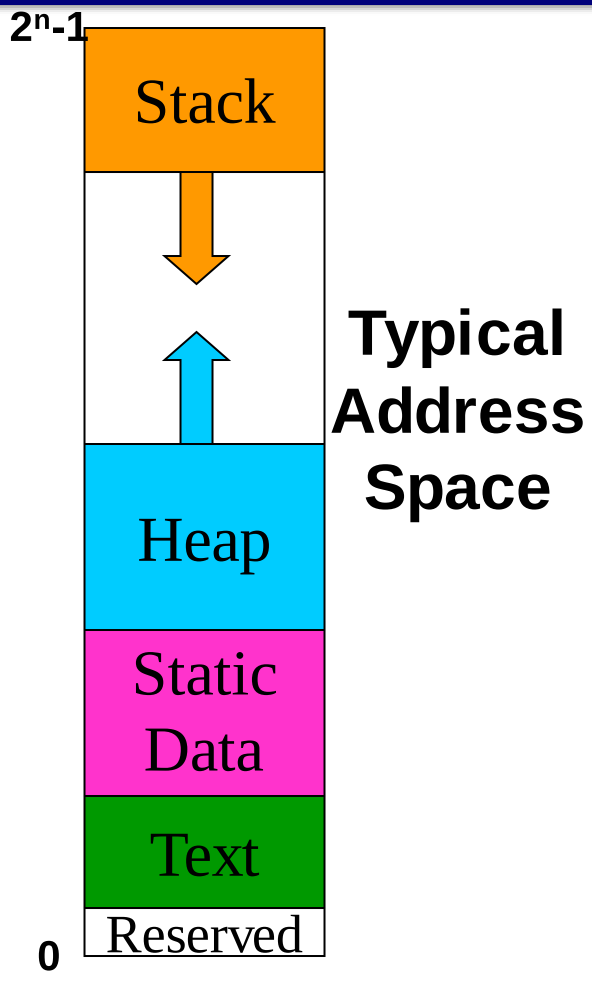
- The shocking truth:
  You've been using pointers all along!
- Every array *IS* a pointer to a block of memory
- **Pointer arithmetic:** If you add an integer N to a pointer P, you get the address of N *things* later from pointer P
  - "Thing" depends on the datatype of the P
- Can *dereference* such pointers to get what's there
  - Interpreted according to the datatype of P
  - E.g. *(nums-1) is a number related to how we represent the letter 'o'.

```
int x = 9;
char msg[] = "hello";
short nums[] = {6,7,8};
```



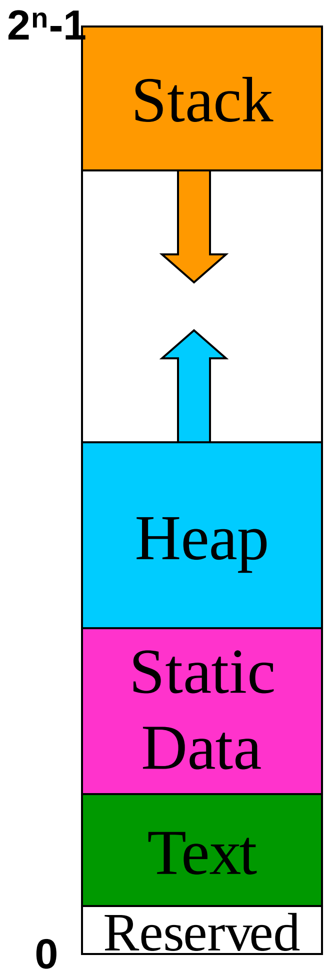
Memory Layout

- Memory is array of bytes, but there are conventions as to what goes where in this array
- Text: instructions (the program to execute)
- Data: global variables
- Stack: local variables and other per-function state; starts at top & grows down
- Heap: dynamically allocated variables; grows up
- What if stack and heap overlap????



Memory Layout: Example

```
int anumber = 3;

int factorial (int x) {
  if (x == 0) {
    return 1;
  }
  else {
    return x * factorial (x – 1);
  }
}

int main (void) {
  int z = factorial (anumber);
  int* p = malloc(sizeof(int)*64);
  printf("%d\n", z);
  return 0;
}                         // p is a local on stack, *p is in heap
```



Typical Address Space

Summary: From C to Binary

- Everything must be represented in binary!
- Pointer is memory location that contains address of another memory location
- Computer memory is linear array of bytes
  - **Integers**:
    - **unsigned** $\{0..2^n-1\}$ vs **signed** $\{-2^{n-1} .. 2^{n-1}-1\}$ ("2's complement")
    - **char** (8-bit), **short** (16-bit), **int/long** (32-bit), **long long** (64-bit)
  - **Floats**: IEEE representation,
    - **float** (32-bit: 1 sign, 8 exponent, 23 mantissa)
    - **double** (64-bit: 1 sign, 11 exponent, 52 mantissa)
  - **Strings**: char array, ASCII representation
- Memory layout
  - **Stack** for local, **static** for globals, **heap** for malloc'd stuff (must free!)

POINTERS, ARRAYS, AND MEMORY ~AGAIN~

The following slides re-state a lot of what we've covered but in a different way. We'll likely skip it for time, but you can use the slides as an additional reference.

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

- What does this print? Why?

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

Stack

main	
a	42
b	100

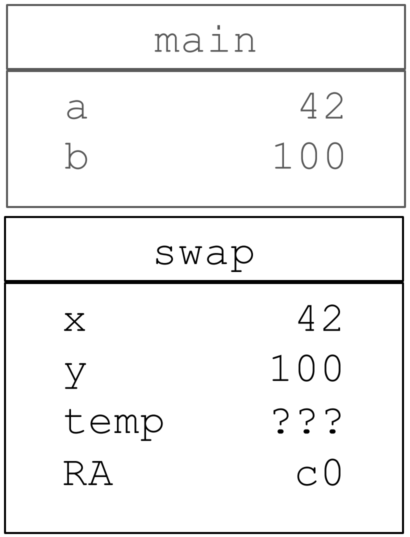
- What does this print? Why?

Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

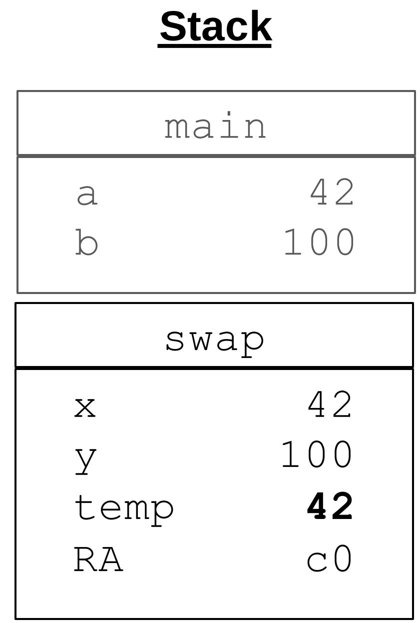
- What does this print? Why?

Stack



Let's do a little Java...

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```



- What does this print? Why?

Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
→   x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0 → swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

Stack

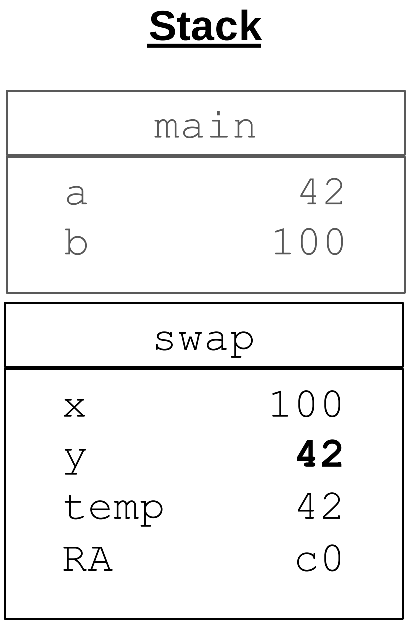
main	
a	42
b	100

swap	
x	**100**
y	100
temp	42
RA	c0

- What does this print? Why?

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```



- What does this print? Why?

Let’s do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
    System.out.println(“a =“ + a + “ b = “ + b);
  }
}
```

Stack

main	
a	42
b	100

- What does this print? Why?

Let’s do some different Java…

```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
    System.out.println(“a =“ + a.data +
                       “ b = “ + b.data);
  }
}
```

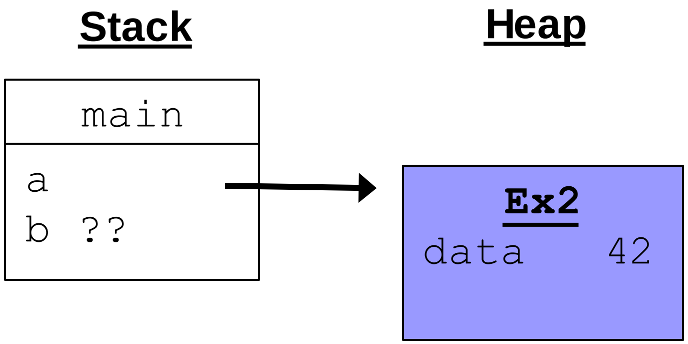
Stack

main
a ??
b ??

- What does this print? Why?

Let's do some different Java…

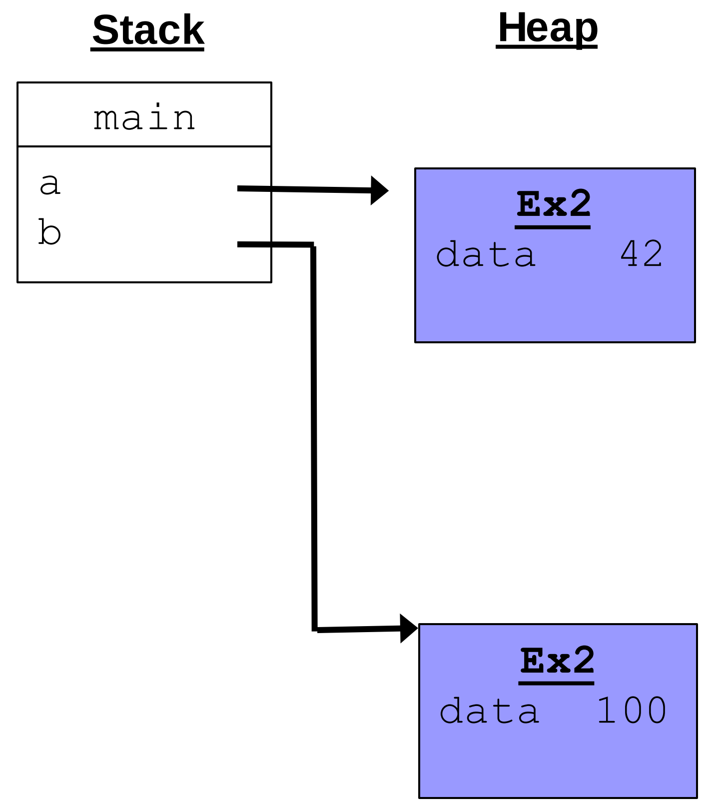
```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

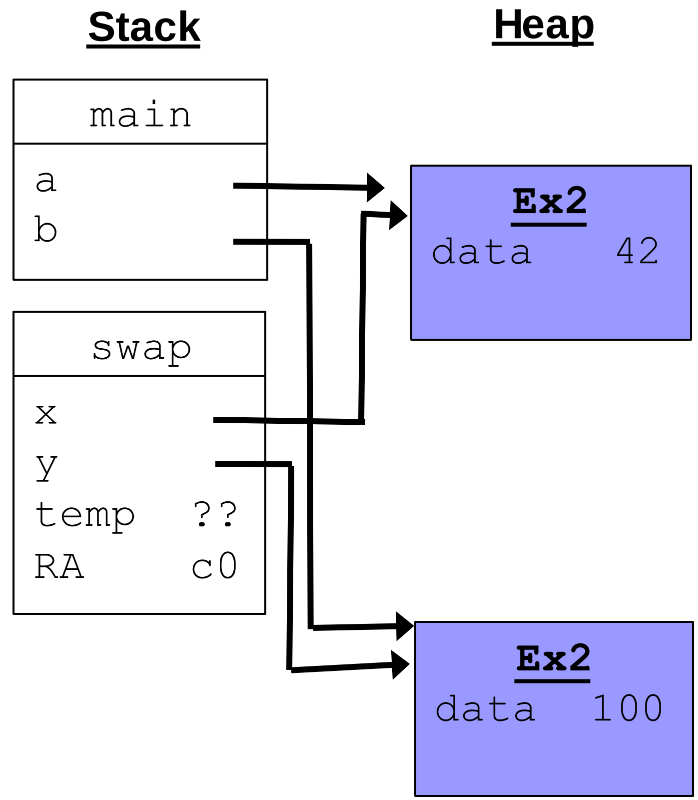
```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let’s do some different Java…

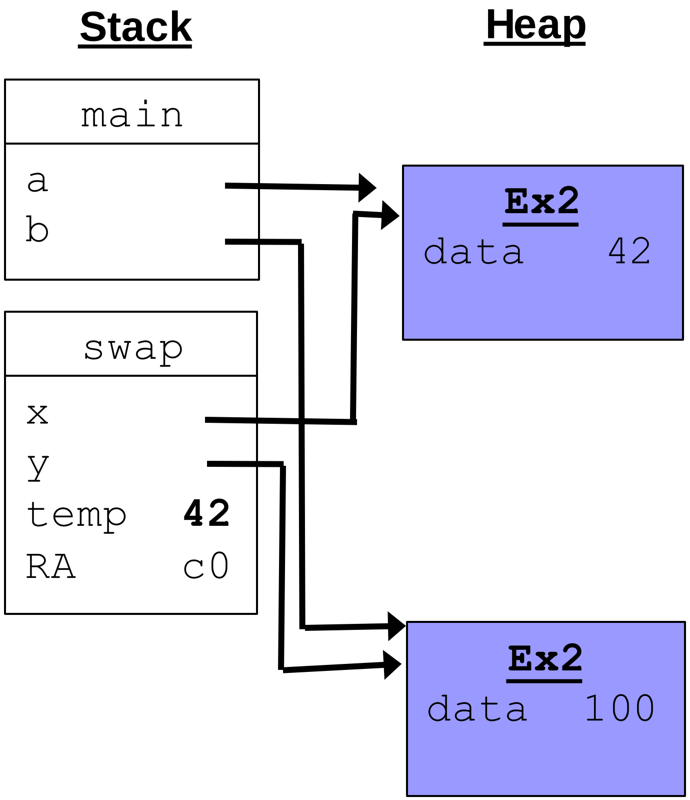
```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
c0  swap (a, b);
    System.out.println(“a =“ + a.data +
                       “ b = “ + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

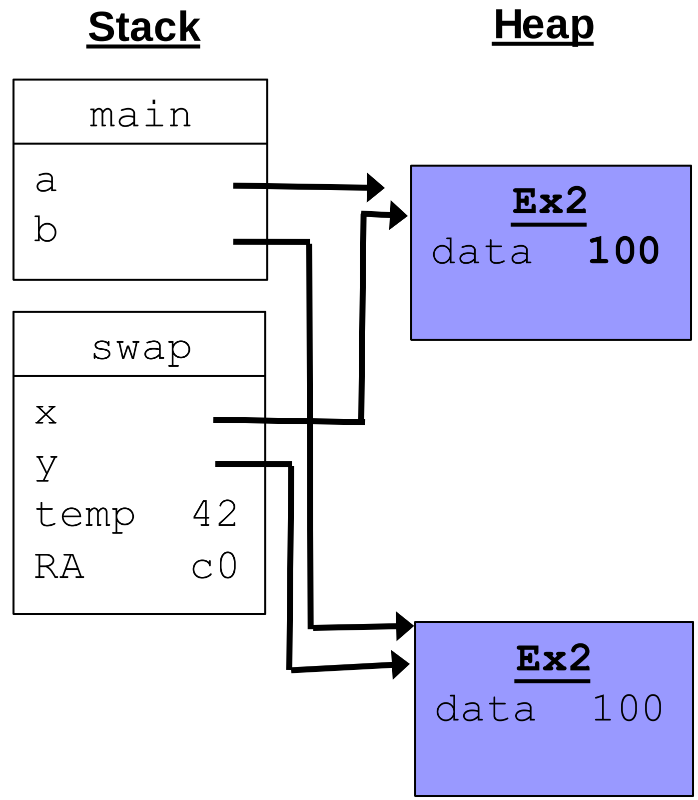
```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
c0  swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

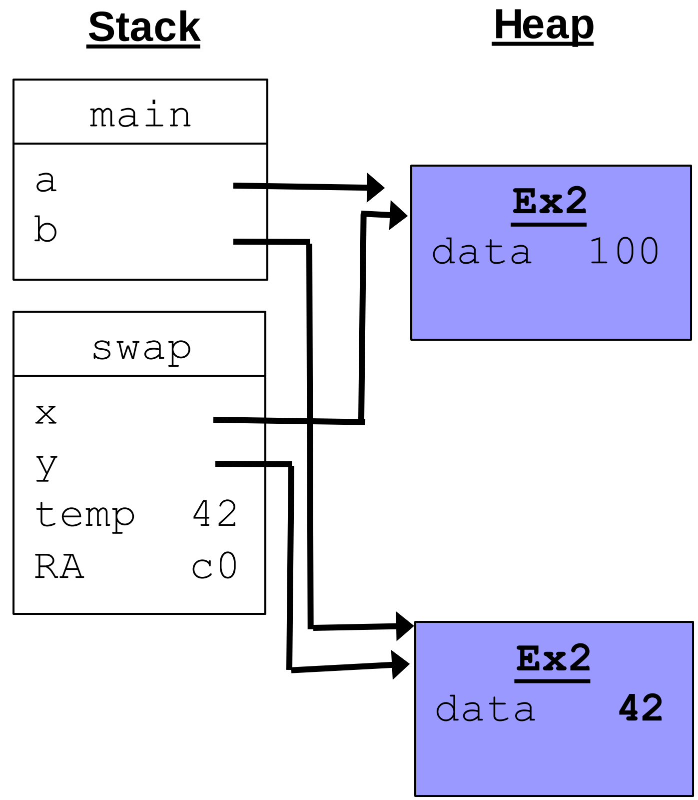
```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
c0  swap (a, b);
    System.out.println("a =" + a.data +
                      " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

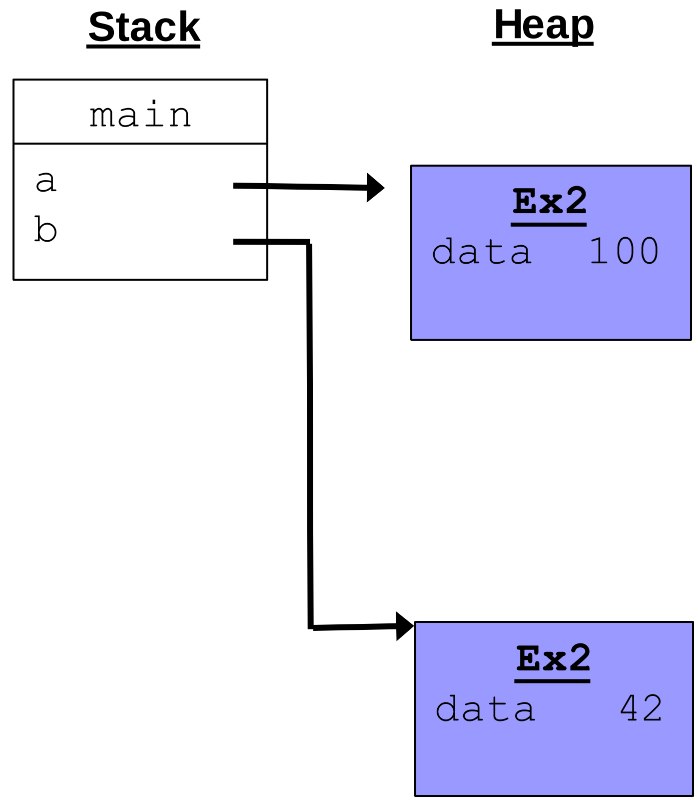
```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
c0  swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let’s do some different Java…

```
public class Ex2 {
  int data;
  public Ex2 (int d) { data = d; }
  public static void swap (Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Example a = new Example (42);
    Example b = new Example (100);
    swap (a, b);
    System.out.println(“a =“ + a.data +
                         “ b = “ + b.data);
  }
}
```



- What does this print? Why?

References and Pointers (review)

- Java has references:
  - Any variable of object type is a reference
  - Point at objects (which are all in the heap)
    - Under the hood: is the memory address of the object
  - Cannot explicitly manipulate them (*e.g.,* add 4)

- Some languages (C,C++,assembly) have explicit pointers:
  - Hold the memory address of something
  - Can explicitly compute on them
  - Can de-reference the pointer (*ptr) to get thing-pointed-to
  - Can take the address-of (&x) to get something’s address
  - Can do very unsafe things, shoot yourself in the foot

Pointers

- “address of” operator &
  - don’t confuse with bitwise AND operator (&&)

Given

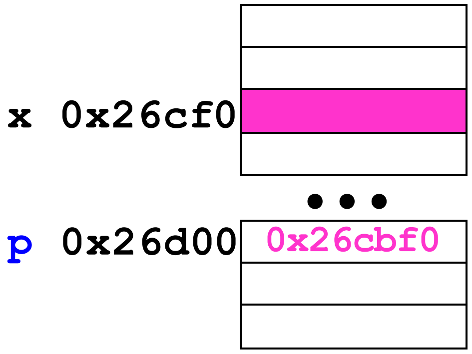
```
int x; int* p;  // p points to an int
p = &x;
```

Then

```
*p = 2;  and x = 2; produce the same result
 Note: p is a pointer, *p is an int
```

- What happens for `p = 2?;`

On 32-bit machine, p is 32-bits



Back to Arrays

- Java:

```
int [] x = new int [nElems];
```

- C:

```
int data[42]; //if size is known constant
int* data = (int*)malloc (nElem * sizeof(int));
```

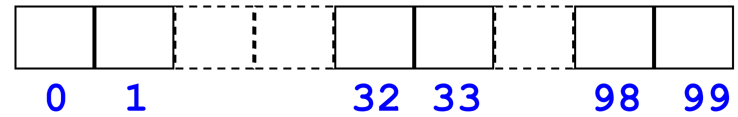
  - `malloc` takes number of bytes
  - `sizeof` tells how many bytes something takes

Arrays, Pointers, and Address Calculation

- x is a pointer, what is x+33?
- A pointer, but where?
  - what does calculation depend on?
- Result of adding an int to a pointer depends on size of object pointed to

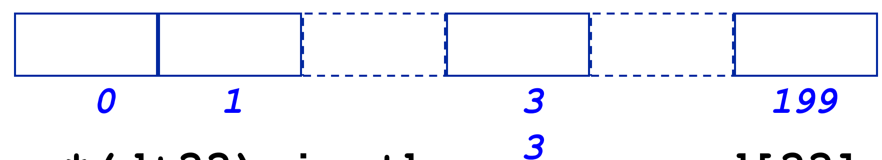
  - One reason why we tell compiler what type of pointer we have, even though all pointers are really the same thing (and same size)

```
int* a=malloc(100*sizeof(int));
```




```
a[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is
0x00a4, a+2 is 0x00a8
(decimal 160, 164, 168)
```

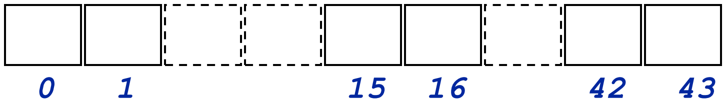
```
double* d=malloc(200*sizeof(double));
```




```
*(d+33) is the same as d[33]
if d is 0x00b0, then d+1 is
0x00b8, d+2 is 0x00c0
(decimal 176, 184, 192)
```

More Pointer Arithmetic

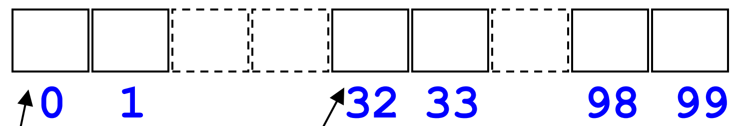
- address one past the end of an array is ok for pointer comparison only
- what's at `*(begin+44)`?
- what does `begin++` mean?
- how are pointers compared using < and using == ?
- what is value of `end - begin`?




```
char* a = new char[44];
char* begin = a;
char* end = a + 44;

while (begin < end)
{
    *begin = 'z';
    begin++;
}
```

More Pointers & Arrays

```
int* a = new int[100];
```



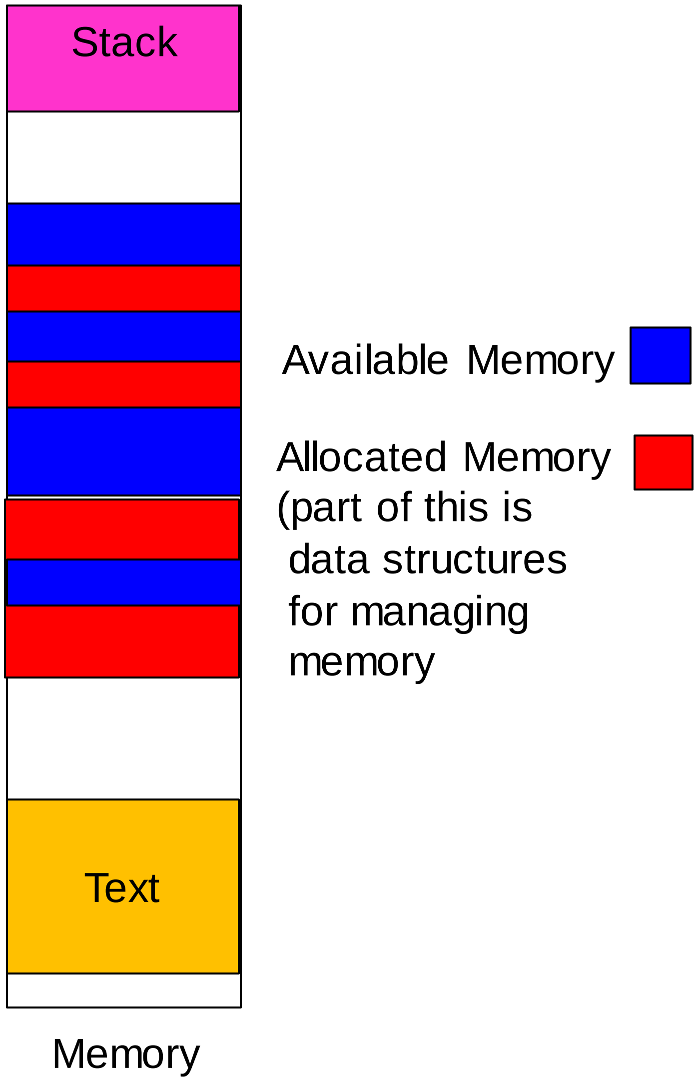

```
a is a pointer
*a is an int
a[0] is an int (same as *a)
a[1] is an int
a+1 is a pointer
a+32 is a pointer
*(a+1) is an int (same as a[1])
*(a+99) is an int
*(a+100) is trouble
```

Array Example

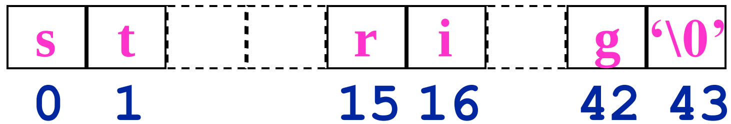
```
#include <stdio.h>

main()
{
  int* a = (int*)malloc (100 * sizeof(int));
  int* p = a;
  int k;

  for (k = 0; k < 100; k++)
    {
      *p = k;
      p++;
    }
  printf("entry 3 = %d\n", a[3])
}
```

Memory Manager (Heap Manager)

- malloc() and free()
- Library routines that handle memory management for heap (allocation / deallocation)
- Java has garbage collection (reclaim memory of unreferenced objects)
- C must use free, else memory leak



Strings as Arrays (review)



- A string is an array of characters with '\0' at the end
- Each element is one byte, ASCII code
- '\0' is null (ASCII code 0)

strlen() again

- `strlen()` returns the number of characters in a string
  - same as number elements in char array?

```
int strlen(char * s)
 // pre: '\0' terminated
 // post: returns # chars
 {
      int count=0;
      while (*s++)
              count++;
      return count;
 }
```

Vector Class vs. Arrays

- Vector Class
  - insulates programmers
  - array bounds checking
  - automagically growing/shrinking when more items are added/deleted
- How are Vectors implemented?
  - Arrays, re-allocated as needed
- Arrays can be more efficient