• 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?
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

• 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
- 8 bits is a byte
- 16 bits is a half-word (for MIPS32)
- 32 bits is a word (for MIPS32)
- 64 bits is a double-word (for MIPS32)
- 128 bits is a quad-word (for MIPS32)

**Integers (char, short, int, long):**
- “2’s Complement” (32-bit or 64-bit representation)

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

**Character (char):**
- ASCII 7-bit code

---

**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

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, you need $\log_2 T$ 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?
    • $\log_2 4G = \ldots \ldots \ldots$
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 “kibibyte” 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.
  - Easy rule to convert between exponent and binary metric number:

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

- $2^{13}$ bytes = $2^3$ kB = 8 kB
- $2^{39}$ bytes = $2^9$ GB = 512 GB
- $2^{05}$ bytes = $2^5$ B = 32 B

This matters a lot later on

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

- **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

<table>
<thead>
<tr>
<th>Digit</th>
<th>Base</th>
<th>Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>?</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>$457 \div 2 =$</td>
<td>228</td>
<td>1</td>
</tr>
<tr>
<td>$228 \div 2 =$</td>
<td>114</td>
<td>0</td>
</tr>
<tr>
<td>$114 \div 2 =$</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>$57 \div 2 =$</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>$28 \div 2 =$</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>$14 \div 2 =$</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>$7 \div 2 =$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$3 \div 2 =$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$1 \div 2 =$</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

111001001
Decimal to binary using comparison

<table>
<thead>
<tr>
<th>Num</th>
<th>Compare $2^n$</th>
<th>$\geq$ ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>457</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>201</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>73</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

111001001
Hexadecimal

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

Indicates a hex number

0xDEADBEEF

0x02468ACE

0x13579BDF

One hex digit represents 4 bits. Two hex digits represent a byte (8 bits).
Binary to/from hexadecimal

- $010110110010011_2 \rightarrow$
- $0101\ 1011\ 0010\ 0011_2 \rightarrow$
- $5\ \ B\ \ 2\ \ 3_{16}$

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

$1\ \ F\ \ 4\ \ B_{16} \rightarrow$

$0001\ 1111\ 0100\ 1011_2 \rightarrow$

$0001111101001011_2$
BitOps: Unary

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

\[
\begin{align*}
\sim 0x0d & \quad // \ (binary \ 00001101) \\
== 0xf2 & \quad // \ (binary \ 11110010)
\end{align*}
\]

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

```c
char i, j1, j2, j3;
i = 0x0d; \quad // binary \ 00001101
j1 = \sim i; \quad // binary \ 11110010
j2 = \sim i; \quad // binary \ 11110011
j3 = \! i; \quad // 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 (\mid): 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):

- \( X \& 1 = X \)  \textit{ANDing with 1 does nothing}
- \( X \& 0 = 0 \)  \textit{ANDing with 0 gives zero}
- \( X \mid 0 = X \)  \textit{ORing with 0 does nothing}
- \( X \mid 1 = 1 \)  \textit{ORing with 1 gives one}
- \( X ^ 0 = X \)  \textit{XORing with 0 does nothing}
- \( X ^ 1 = \sim X \)  \textit{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

• $x \ll y$ 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

01111001 $\ll$ 3

these 3 bits are zero filled
these 3 bits are discarded
ShiftOps... (cont’d)

• $x \ll y$ 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:
    \[ m = 0x02; \quad // \quad 00000010_2 \]
  • OR the mask with the input:
    \[ v = 0x41; \quad // \quad 01000001_2 \]
    \[ v |= m; \quad // \quad 01000011_2 \]

• Clear a certain bit to 0?
  • Make a MASK with a zero at every position you want to clear:
    \[ m = 0xFD; \quad // \quad 11111101_2 \] (could also write \(~0x02\))
  • AND the mask with the input:
    \[ v = 0x27; \quad // \quad 00100111_2 \]
    \[ v &= m; \quad // \quad 00100101_2 \]

• 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:
    \[ v = 0x67; \quad // \quad 01100111_2 \]
    \[ v >>= 2; \quad // \quad 00011001_2 \]
    \[ v &= 0x0F; \quad // \quad 00001001_2 \]
Binary Math : Addition

• Suppose we want to add two numbers:

\[
\begin{align*}
00011101 \\
+ 00101011 \\
\hline
00101011
\end{align*}
\]

• How do we do this?
Binary Math : Addition

• Suppose we want to add two numbers:

\[
\begin{array}{c}
00011101 & 695 \\
+ 00101011 & + 232 \\
\end{array}
\]

• 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:

\[
\begin{align*}
00011101 & \quad 695 \\
+ 00101011 & \quad + 232 \\
\hline
00100101 & \quad 7
\end{align*}
\]

- First add one’s digit 5+2 = 7
Binary Math: Addition

- Suppose we want to add two numbers:
  
  \[
  \begin{array}{c}
  \text{1} \\
  00011101 \quad 695 \\
  + \quad 00101011 \quad + \quad 232 \\
  \hline
  00100111 \quad 27
  \end{array}
  \]

- 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
       + 00101011    + 927
    00101011
```

• 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
  ________
  00101011

- Back to the binary:
- First add 1’s digit 1+1 = ...?
• Suppose we want to add two numbers:

\[
\begin{array}{c}
1 \\
00011101 \\
+ 00101011 \\
\hline
00100111
\end{array}
\]

• 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:

\[
\begin{array}{cccc}
1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
+ & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
0 & 0
\end{array}
\]

• 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:

\[
\begin{align*}
111111 \\
00011101 & = 29 \\
+ 00101011 & = 43 \\
\hline
01001000 & = 72
\end{align*}
\]

• Can check our work in decimal
Issues for Binary Representation of Numbers

- **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
1’s Complement Representation for Integers

- Use largest positive binary numbers to represent negative numbers
- To negate a number, invert (‘not’) each bit:
  - $0 \rightarrow 1$
  - $1 \rightarrow 0$
- Cons:
  - Still two 0s (yuck)
  - Still hard to compute with

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
</tr>
</tbody>
</table>
2’s Complement Integers

- Use large positives to represent negatives
- \((-x) = 2^n - x\)
- This is 1’s complement + 1
- So, to negate, \textbf{just invert bits and add 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 \rightarrow \text{good!}

EVERYBODY DOES THIS
Another way to think about 2’s complement

- **Regular base 10:**
  - $6253 = 6 \times 10^3 + 2 \times 10^2 + 5 \times 10^1 + 3 \times 10^0$

- **Unsigned base 2:**
  - $1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  - $= 8 + 4 + 1$
  - $= 13$

- **Signed base 2:**
  - $1101 = -1 \times 1000 + 1 \times 100 + 0 \times 00 + 1 \times 1$
  - $= -1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  - $= -8 + 4 + 1$
  - $= -3$

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

Alternately, flip the bits and add 1:
- $1101$
- Flip: 0010
- +1: 0011
- That’s 3 in binary, so the number is indeed -3
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

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:**

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

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

```c
int x;               // defaults to signed
unsigned int y;     // explicitly unsigned
```
2’s Complement Precision Extension

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

Examples

14\textsubscript{10} = 001110\textsubscript{2} in 6-bit representation.
14\textsubscript{10} = 000000001110\textsubscript{2} in 12-bit representation

-14\textsubscript{10} = 110010\textsubscript{2} in 6-bit representation
-14\textsubscript{10} = 111111110010\textsubscript{2} in 12-bit representation.
Binary Math : Addition

- Let’s look at another binary addition:

```
  01011101
+  01101011
______
  01100100
```
Binary Math: Addition

- What about this one:
  \[
  \begin{array}{c}
  \hline
  \text{1111111} \\
  \text{01011101} \quad = \quad 93 \\
  \text{+ \hspace{1em} 01101011} \quad = \quad 107 \\
  \hline
  \text{11001000} \quad = \quad -56
  \end{array}
  \]

- 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?

[Images of two people with text: It hurts when I add two ints and it overflows. Then don't do that.]
Adding works for unsigned and signed numbers.

- 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)

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>+ 0001</td>
<td>1</td>
</tr>
<tr>
<td>------</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1101</td>
</tr>
<tr>
<td>+ 1111</td>
<td>-3</td>
</tr>
<tr>
<td>------</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>+ 0100</td>
<td>4</td>
</tr>
<tr>
<td>------</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>0101</td>
</tr>
<tr>
<td>+ 1111</td>
<td>-1</td>
</tr>
<tr>
<td>------</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td>28 12??</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>20 4??</td>
</tr>
</tbody>
</table>
Subtraction

• 2’s complement makes subtraction easy:
  • Remember: $A - B = A + (-B)$
  • And: $-B = \sim B + 1$
    ↑ 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!)

\[
\begin{array}{c}
1 \\
0110101 \\
- 1010010 \\
\hline
0101101
\end{array}
\]
What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - Speed of light $\sim= 3 \times 10^8$
  - $\pi = 3.1415...$
- Fixed number of bits limits range of integers
  - Can’t represent some important numbers
- Humans use Scientific Notation
  - $1.3 \times 10^4$
Option 1: Fixed point

- Use normal integers, but \((X \times 2^K)\) instead of \(X\)
  - Example: 32 bit int, but use \(X \times 65536\)
    - \(3.1415926 \times 65536 = 205887\)
    - \(0.5 \times 65536 = 32768\), etc..

- Pros:
  - Addition/subtraction just like integers ("free")

- Cons:
  - Mul/div require renormalizing (divide by 64K)
  - Range limited (no good rep for large + small)

- Can be good in specific situations
Can we do better?

- Think about scientific notation for a second:
- For example:
  \[ 6.02 \times 10^{23} \]
- 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: 
  \[ +/- \ X.YYYYYY \times 2^{+/-N} \]

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

- This is “floating point”: most common way
• 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)
• 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
- ...

![Inch measurement diagram]
Floating point example

- Binary fraction example:
  \[101.101 = 4 + 1 + \frac{1}{2} + \frac{1}{8} = 5.625\]

- For floating point, needs normalization:
  \[1.01101 \times 2^2\]

- Sign is +, which = 0

- Exponent = 127 + 2 = 129 = 1000 0001

- Mantissa = 1.011 0100 0000 0000 0000 0000

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

\[0100 0000 1011 0100 0000 0000 0000 0000\]
Example:
What floating-point number is: 0xC1580000?
Answer

What floating-point number is 0xC1580000?

\[
\begin{array}{cccccccccccccccccccc}
1100 & 0001 & 0101 & 1000 & 0000 & 0000 & 0000 & 0000 \\
\end{array}
\]

\[
X = \begin{bmatrix}
1 & 1000 & 0010 & 101 & 1000 & 0000 & 0000 & 0000 & 0000 \\
\end{bmatrix}
\]

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.XXXX representation?

• Exponent of 0 is denormalized
  • 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: +/- \infty
    - 1/0 = +\infty
    - -1/0 = -\infty
  - Non zero mantissa: Not a Number (NaN)
    sqrt(-42) = NaN
Floating Point Representation

• Double Precision Floating point:

64-bit representation:
  • 1-bit sign
  • 11-bit (biased) exponent
  • 52-bit fraction (with implicit 1).

• “double” in Java, C, C++, ...

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11-bit</td>
<td>52 - bit</td>
</tr>
</tbody>
</table>
What About Strings?

- Many important things stored as strings...
  - E.g., your name
- How should we store strings?
# Standardized ASCII (0-127)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>HTML</th>
<th>CHr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>NUL</td>
<td>(null)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>001</td>
<td>SOH</td>
<td>(start of heading)</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>002</td>
<td>STX</td>
<td>(start of text)</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>003</td>
<td>ETX</td>
<td>(end of text)</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>004</td>
<td>EOT</td>
<td>(end of transmission)</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>005</td>
<td>ENQ</td>
<td>(enquiry)</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>006</td>
<td>ACK</td>
<td>(acknowledge)</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
<td>007</td>
<td>BEL</td>
<td>(bell)</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td>010</td>
<td>BS</td>
<td>(backspace)</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
<td>011</td>
<td>TAB</td>
<td>(horizontal tab)</td>
</tr>
<tr>
<td>10</td>
<td>A0</td>
<td>012</td>
<td>LF</td>
<td>(NL line feed, new line)</td>
</tr>
<tr>
<td>11</td>
<td>B0</td>
<td>013</td>
<td>VT</td>
<td>(vertical tab)</td>
</tr>
<tr>
<td>12</td>
<td>C0</td>
<td>014</td>
<td>FF</td>
<td>(NP form feed, new page)</td>
</tr>
<tr>
<td>13</td>
<td>D0</td>
<td>015</td>
<td>CR</td>
<td>(carriage return)</td>
</tr>
<tr>
<td>14</td>
<td>E0</td>
<td>016</td>
<td>SO</td>
<td>(shift out)</td>
</tr>
<tr>
<td>15</td>
<td>F0</td>
<td>017</td>
<td>SI</td>
<td>(shift in)</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>020</td>
<td>DLE</td>
<td>(data link escape)</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>021</td>
<td>DC1</td>
<td>(device control 1)</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>022</td>
<td>DC2</td>
<td>(device control 2)</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>023</td>
<td>DC3</td>
<td>(device control 3)</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>024</td>
<td>DC4</td>
<td>(device control 4)</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>025</td>
<td>NAK</td>
<td>(negative acknowledge)</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>026</td>
<td>SYN</td>
<td>(synchronous idle)</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>027</td>
<td>ETB</td>
<td>(end of trans. block)</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>030</td>
<td>CAN</td>
<td>(cancel)</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>031</td>
<td>EM</td>
<td>(end of medium)</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>032</td>
<td>SUB</td>
<td>(substitute)</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>033</td>
<td>ESC</td>
<td>(escape)</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>034</td>
<td>FS</td>
<td>(file separator)</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>035</td>
<td>GS</td>
<td>(group separator)</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>036</td>
<td>RS</td>
<td>(record separator)</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>037</td>
<td>US</td>
<td>(unit separator)</td>
</tr>
</tbody>
</table>

Source: [www.LookupTables.com](http://www.LookupTables.com)
One Interpretation of 128-255

| 128 | ç | 144 | é | 161 | i | 177 | | | | 193 | | | | 209 | ÷ | | | 225 | b | 241 | ± |
| 129 | ü | 145 | ë | 162 | ó | 178 | | | | 194 | | | | 210 | ÷ | | | 226 | g | 242 | ⨽ |
| 130 | é | 146 | AE | 163 | ú | 179 | | | | 195 | | | | 211 | ÷ | | | 227 | π | 243 |
| 131 | ã | 147 | ò | 164 | ŋ | 180 | | | | 196 | | | | 212 | ÷ | | | 228 | Σ | 244 |
| 132 | ã | 148 | ò | 165 | ŋ | 181 | | | | 197 | | | | 213 | ÷ | | | 229 | σ | 245 |
| 133 | ã | 149 | ò | 166 | å | 182 | | | | 198 | | | | 214 | ÷ | | | 230 | μ | 246 |
| 134 | å | 150 | ū | 167 | å | 183 | | | | 199 | | | | 215 | ÷ | | | 231 | τ | 247 |
| 135 | ç | 151 | ū | 168 | ã | 184 | | | | 200 | | | | 216 | ÷ | | | 232 | Φ | 248 |
| 136 | ã | 152 | _ | 169 | _ | 185 | | | | 201 | | | | 217 | ÷ | | | 233 | φ | 249 |
| 137 | è | 153 | Ö | 170 | _ | 186 | | | | 202 | | | | 218 | ÷ | | | 234 | Ω | 250 |
| 138 | è | 154 | Ü | 171 | _ | 187 | | | | 203 | | | | 219 | ÷ | | | 235 | δ | 251 |
| 139 | i | 156 | £ | 172 | _ | 188 | | | | 204 | | | | 220 | ÷ | | | 236 | ∞ | 252 |
| 140 | î | 157 | ¥ | 173 | i | 189 | | | | 205 | | | | 221 | ÷ | | | 237 | Ψ | 253 |
| 141 | î | 158 | _ | 174 | « | 190 | | | | 206 | | | | 222 | ÷ | | | 238 | Ε | 254 |
| 142 | A | 159 | ý | 175 | » | 191 | | | | 207 | | | | 223 | ÷ | | | 239 | Π | 255 |
| 143 | Å | 160 | á | 176 | Å | 192 | | | | 208 | | | | 224 | ÷ | | | 240 | α | 255 |

Source: www.LookupTables.com
(This allowed totally sweet ASCII art in the 1990s)

Sources:
### About those control codes…

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td><strong>NULL</strong> (null)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td>SOH      (start of heading)</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>2</td>
<td>STX      (start of text)</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>3</td>
<td>ETX      (end of text)</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>4</td>
<td>EOT      (end of transmission)</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>5</td>
<td>ENQ      (enquiry)</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>6</td>
<td>ACK      (acknowledge)</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>7</td>
<td><strong>BEL</strong> (bell)</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>10</td>
<td>BS       (backspace)</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>9</td>
<td><strong>TAB</strong> (horizontal tab)</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>12</td>
<td>LF       (NL line feed, new line)</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>13</td>
<td>VT       (vertical tab)</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>14</td>
<td>FF       (NP form feed, new page)</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>15</td>
<td><strong>CR</strong>   (carriage return)</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>16</td>
<td>SO       (shift out)</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>17</td>
<td>SI       (shift in)</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>2</td>
<td>DLE      (data link escape)</td>
</tr>
<tr>
<td>17</td>
<td>21</td>
<td>1</td>
<td>DC1      (device control 1)</td>
</tr>
<tr>
<td>18</td>
<td>22</td>
<td>2</td>
<td>DC2      (device control 2)</td>
</tr>
<tr>
<td>19</td>
<td>23</td>
<td>3</td>
<td>DC3      (device control 3)</td>
</tr>
<tr>
<td>20</td>
<td>24</td>
<td>4</td>
<td>DC4      (device control 4)</td>
</tr>
<tr>
<td>21</td>
<td>25</td>
<td>5</td>
<td>NAK      (negative acknowledge)</td>
</tr>
<tr>
<td>22</td>
<td>26</td>
<td>6</td>
<td>SYN      (synchronous idle)</td>
</tr>
<tr>
<td>23</td>
<td>27</td>
<td>7</td>
<td>ETB      (end of trans. block)</td>
</tr>
<tr>
<td>24</td>
<td>30</td>
<td>0</td>
<td>CAN      (cancel)</td>
</tr>
<tr>
<td>25</td>
<td>31</td>
<td>1</td>
<td>EM       (end of medium)</td>
</tr>
<tr>
<td>26</td>
<td>32</td>
<td>2</td>
<td>SUB      (substitute)</td>
</tr>
<tr>
<td>27</td>
<td>33</td>
<td>3</td>
<td>ESC      (escape)</td>
</tr>
<tr>
<td>28</td>
<td>34</td>
<td>4</td>
<td>FS       (file separator)</td>
</tr>
<tr>
<td>29</td>
<td>35</td>
<td>5</td>
<td>GS       (group separator)</td>
</tr>
<tr>
<td>30</td>
<td>36</td>
<td>6</td>
<td>RS       (record separator)</td>
</tr>
<tr>
<td>31</td>
<td>37</td>
<td>7</td>
<td>US       (unit separator)</td>
</tr>
</tbody>
</table>

(Greyed out ones almost never used)

We need to talk about CR and LF…
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!
• 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

```c
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
What is an array?

- 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’.

```c
int x = 9;
char msg[] = “hello”;
short nums[] = {6, 7, 8};
```
What is an array?

- **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’.

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

```
&x  ↓
  ↓
09 00 00 00  ‘h’  ‘e’  ‘l’  ‘l’  ‘o’  00 06 00 07 00 08 00

msg
  ↓
msg+1  ↑  msg+2  ↑  msg+3  ↑  msg+4  ↑  msg+5  ↑  msg+6  ↑
  ↓
nums
  ↓
nums+1  ↑  nums+2
```
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???
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
• 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!)
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.
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?
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?
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…

```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?
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?
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?
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?
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?
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?
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...

```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?
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?
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?
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?
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
```
x  0x26cf0
...  
p  0x26d00  0x26cbf0
```
Back to Arrays

- **Java:**
  ```java
  int [] x = new int [nElems];
  ```

- **C:**
  ```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)

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

int a[100];
```

```text
A[32] is the same as *(a+33)
if a is 0x00a0, then a+1 is 0x00a4, a+2 is 0x00a8
(decimal 160, 164, 168)
```

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

```text
*(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)
```
• 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?

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

while (begin < end)
{
    *begin = ‘z’;
    begin++;
}
```
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
#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()` returns the number of characters in a string**
  
  • same as number elements in char array?

```c
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**