• 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
- Original Nintendo: 8 bit
- Super Nintendo: 16 bit
- Intel x86 (classic): 32 bit
- Nintendo 64: 64 bit
- Intel x86_64 (modern): 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?
  • $\log_2 4G = \ldots$?
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:

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

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

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

• 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

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

The binary representation of the decimal number 457 is **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

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

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

1  F  4  16 -->

0001 1111 0100 1011_2 -->

0001111101001011_2

<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>
BitOps: Unary

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

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

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 (|):** 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*
- \(X \mid 0 = X\) *ORing with 0 does nothing*
- \(X \mid 1 = 1\) *ORing with 1 gives one*
- \(X ^ 0 = X\) *XORing with 0 does nothing*
- \(X ^ 1 = \sim X\) *XORing with 1 flips the bit*
Two Operands... (cont’d)

- Examples

| 0011 1000 | 0011 1000 | 0011 1000 |
| & | \^ |

- 1101 1110

---

- 0001 1000

---

- 1111 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

\[
\begin{array}{c}
01111001 \ll 3 \\
\hline
11001000
\end{array}
\]

these 3 bits are discarded

these 3 bits are zero filled
• \( x \gg y \) 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*:
    \[
    m = 0xFD; \quad // \quad 11111101_2 \quad (\text{could also write} \ \sim 0x02)
    \]
  • 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:
    \[
    v = 0x67; \quad // \quad 0110011_2
    v >>= 2; \quad // \quad 00011001_2
    v &= 0x0F; \quad // \quad 00001001_2
    \]
Binary Math : Addition

• Suppose we want to add two numbers:

  00011101
  +  00101011
  ________
  00101011

• How do we do this?
Suppose we want to add two numbers:

- 00011101 + 00101011
- 695 + 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:

\[
\begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
00101011
\end{array}
\quad
\begin{array}{c}
695 \\
+ 232 \\
\hline
7
\end{array}
\]

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

• Suppose we want to add two numbers:

\[
\begin{array}{c}
\phantom{+}00011101 \\
+ 00101011 \\
\hline
1227
\end{array}
\]

\[
\begin{array}{c}
\phantom{+}695 \\
+ 232 \\
\hline
27
\end{array}
\]

• 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:

\[
\begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
00101110
\end{array}
\]

\[
\begin{array}{c}
695 \\
+ 232 \\
\hline
927
\end{array}
\]

• 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:

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

• Back to the binary:

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

```
  1
00011101
+ 00101011
______
00101010
```

• 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:

\[
\begin{array}{c}
11 \\
00011101 \\
+ 00101011 \\
\hline
00
\end{array}
\]

• 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....
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

NOBODY DOES THIS
1’s Complement Representation for Integers

- Use largest positive binary numbers to represent negative numbers
  
- To negate a number, invert ("not") each bit:
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>-7</th>
<th>-6</th>
<th>-5</th>
<th>-4</th>
<th>-3</th>
<th>-2</th>
<th>-1</th>
<th>-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>

- Cons:
  - Still two 0s (yuck)
  - Still hard to compute with

NOBODY DOES THIS EITHER
2’s Complement Integers

- Use large positives to represent negatives
- \((-x) = 2^n - x\)
- This is 1’s complement + 1
- \((-x) = 2^n - 1 - x + 1\)
- So, 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 = 6000 + 200 + 50 + 3
    = 6*10^3 + 2*10^2 + 5*10^1 + 3*10^0

- **Unsigned base 2:**
  - 1101 = 1000 + 100 + 00 + 1
    = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0
    = 8 + 4 + 1
    = 13

- **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:

- 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
    \[ 100000_2 = -32_{10}; \text{ but } 32_{10} \text{ could not be represented} \]

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 \( \{0 \ldots 2^n - 1\} \)
  • And it’s **signed**, then I can represent \( \{-2^{n-1} \ldots 2^{n-1} - 1\} \)

• 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
```
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

Examples

\[ 14_{10} = 001110_2 \text{ in 6-bit representation.} \]
\[ 14_{10} = 000000001110_2 \text{ in 12-bit representation} \]

\[-14_{10} = 110010_2 \text{ in 6-bit representation} \]
\[-14_{10} = 111111110010_2 \text{ in 12-bit representation.} \]
Let’s look at another binary addition:

```
01011101
+ 01101011
_______
01100110
```
Binary Math : Addition

- What about this one:
  
  \[
  \begin{array}{c}
  11111111 \\
  01011101 \\
  + 01101011 \\
  \hline
  11001000
  \end{array}
  \]
  
  \[
  \begin{array}{c}
  = 93 \\
  = 107 \\
  = -56
  \end{array}
  \]

- 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?
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!)

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

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - Speed of light $\approx 3 \times 10^8$
  - $\pi = 3.1415...$
- 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. YYYYY 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:
  \[101.101 = 4 + 1 + \frac{1}{2} + \frac{1}{8} = 5.625\]

- 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:

\[0x\ 4\ 0\ b\ 4\ 0\ 0\ 0\ 0\ 0\]
Floating Point Representation

Example:
What floating-point number is: 0xC1580000?
What floating-point number is $0xC1580000$?

$1100\ 0001\ 0101\ 1000\ 0000\ 0000\ 0000\ 0000$

$X = \begin{array}{cccccc}
\text{s} & \text{E} & \text{F} \\
1 & 1000 & 0010 & 101 & 1000 & 0000\ 0000\ 0000\ 0000
\end{array}$

Sign = 1 which is negative

Exponent = $(128+2)-127 = 3$

Mantissa = 1.1011

$-1.1011\times2^3 = -1101.1 = -13.5$
Trick question

• How do you represent 0.0?
  • Why is this a trick question?
  • 0.0 = 0.00000
  • But need 1.XXXXX representation?

• 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: +/- ∞
    - 1/0 = +∞
    - -1/0 = -∞
  - 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)

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

Source: www.LookupTables.com
One Interpretation of 128-255

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>Ç</td>
</tr>
<tr>
<td>129</td>
<td>ü</td>
</tr>
<tr>
<td>130</td>
<td>é</td>
</tr>
<tr>
<td>131</td>
<td>à</td>
</tr>
<tr>
<td>132</td>
<td>ä</td>
</tr>
<tr>
<td>133</td>
<td>à</td>
</tr>
<tr>
<td>134</td>
<td>å</td>
</tr>
<tr>
<td>135</td>
<td>ç</td>
</tr>
<tr>
<td>136</td>
<td>è</td>
</tr>
<tr>
<td>137</td>
<td>ê</td>
</tr>
<tr>
<td>138</td>
<td>è</td>
</tr>
<tr>
<td>139</td>
<td>i</td>
</tr>
<tr>
<td>140</td>
<td>î</td>
</tr>
<tr>
<td>141</td>
<td>ì</td>
</tr>
<tr>
<td>142</td>
<td>Ä</td>
</tr>
<tr>
<td>143</td>
<td>Å</td>
</tr>
</tbody>
</table>

Source: www.LookupTables.com
(This allowed totally sweet ASCII art in the 90s)

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

We need to talk about CR and LF...

(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

Memory layout on a little endian system

Program

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

- **Big endian byte 0**: 12
- **Little endian byte 0**: 78

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
Memory Layout

- Memory is an 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
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!)
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…

```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?
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?

```
Stack

<table>
<thead>
<tr>
<th>main</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>42</td>
</tr>
<tr>
<td>b</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>swap</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>42</td>
</tr>
<tr>
<td>y</td>
<td>100</td>
</tr>
<tr>
<td>temp</td>
<td>???</td>
</tr>
<tr>
<td>RA</td>
<td>c0</td>
</tr>
</tbody>
</table>
```
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…

```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?
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?
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?
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?
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?
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 (nElems * sizeof(int));
  ```
  - `malloc` takes number of bytes
  - `sizeof` tells how many bytes something takes
• *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));

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)
```
• 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])
}
• `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
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?

```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