ECE 250 / CPS 250
Computer Architecture

From C to Binary

Benjamin Lee
Slides based on those from
Andrew Hilton (Duke), Alvy Lebeck (Duke)
Benjamin Lee (Duke), and Amir Roth (Penn)
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?
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!

• Key Idea: 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**
- 32 bits is a **word**
- 64 bits is a **double-word**
- 128 bits is a **quad-word**

**Integers (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
Issues for Binary Representation of 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:
- Cons:
  - Still two 0s (yuck)
  - Still hard to compute with
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!}
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
2’s Complement Precision Extension

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

\textbf{Examples}

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

\[\text{-}14_{10} = 110010_2 \text{ in 6-bit representation}\]
\[-14_{10} = 111111110010_2 \text{ in 12-bit representation.}\]
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:

\[
\begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
00110111
\end{array}
\]

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:

\[
\begin{align*}
00011101 & \quad 695 \\
+ 00101011 & \quad + 232 \\
\hline
00101100 & \quad 7
\end{align*}
\]

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

• Suppose we want to add two numbers:

\[
\begin{array}{c}
1 \\
00011101 \\
+ 00101011 \\
\hline
695 + 232 = 27
\end{array}
\]

• 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:

\[ \begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
695 \\
+ 232 \\
\hline
927
\end{array} \]

• 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:

\[
\begin{array}{c}
00011101 \\
+ \ 
00101011 \\
\hline
00110100
\end{array}
\]

• 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:

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

• 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....
Binary Math : Addition

• Suppose we want to add two numbers:

\[
\begin{array}{c}
  00011101 \\
+ 00101011 \\
= 01001000
\end{array}
\]

= 29

\[
\begin{array}{c}
  + 00101011 \\
= 01001000
\end{array}
\]

= 43

= 72

• Can check our work in decimal
Binary Math : Addition

• What about this one:

\[
\begin{array}{c}
01011101 \\
+ 01101011 \\
\hline
01101011
\end{array}
\]
Binary Math : Addition

• What about this one:

\[
\begin{array}{c}
1111111 \\
01011101 &= 93 \\
01101011 &+ 01101011 = 107 \\
11001000 &= -56 \\
\end{array}
\]

• 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

• Call Integer Overflow

• Real problem in programs
Subtraction

- 2’s complement makes subtraction easy:
  - Remember: \( A - B = A + (-B) \)
  - And: \(-B = \sim B + 1\)
    \[ \uparrow \] 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!)

\[
\begin{array}{c}
1 \\
0110101 \\
-1010010 \\
+0101101
\end{array}
\]

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

• There are infinitely many real numbers between two integers

• Many important numbers are real
  • Speed of light $\sim 3 \times 10^8$
  • 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

• Represent non-integer in two parts
  • Integer and fraction parts separated by binary point
  • Example: 8 bit fixed-point number with 3 fractional bits
    • $(00010.110)_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-2} = (2.75)_{10}$

• Pros:
  • Similar to integer representation, except for binary point
  • Addition/subtraction just like integers

• Cons:
  • Loss of range and precision
  • Example: 1 fractional bit gives precision to within 0.5
Can we do better?

- Think about scientific notation for a second:
  - For example:
    - $6.82 \times 10^{23}$
  - Real number, but comprised of ints:
    - 6 generally only 1 digit here
    - 82 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 X 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...
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 + \frac{1}{2} + \frac{1}{8} = 5.625

• For floating point, needs normalization:
  1.01101 * 2^2

• Sign is +, which = 0

• Exponent = 127 + 2 = 129 = \textcolor{red}{1000} 0001

• Mantissa = \textcolor{blue}{1.011 0100 0000 0000 0000 0000}

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>0001</td>
<td>011</td>
<td>0100</td>
</tr>
</tbody>
</table>
Floating Point Representation

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

What floating-point number is \(0xC1580000\)?

\[
\begin{array}{c}
\text{Sign} = 1 \text{ which is negative} \\
\text{Exponent} = (128+2)-127 = 3 \\
\text{Mantissa} = 1.1011 \\
-1.1011 \times 2^3 = -1101.1 = -13.5
\end{array}
\]
Trick question

• How do you represent 0.0?
  • Why is this a trick question?
Trick question

• How do you represent 0.0?
  • Why is this a trick question?
  • 0.0 = 000000000
  • But need 1.XXXXX representation?
Trick question

• How do you represent 0.0?
  • Why is this a trick question?
  • 0.0 = 000000000
  • But need 1.XXXXXX representation?

• Exponent = 0000 0000 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)
    \[ \sqrt{-42} = \text{NaN} \]
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?
## ASCII Character Representation

| Oct. Char | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | 010 | 011 | 012 | 013 | 014 | 015 | 016 | 017 | 020 | 021 | 022 | 023 | 024 | 025 | 026 | 027 | 030 | 031 | 032 | 033 | 036 | 037 | 040 | 041 | 042 | 043 | 044 | 045 | 046 | 047 |
|-----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| nul       | 000 | nul | 001 | soh | 002 | stx | 003 | etx | 004 | eot | 005 | enq | 006 | ack | 007 | bel | 010 | bs  | 011 | ht  | 012 | nl  | 013 | vt  | 014 | np  | 015 | cr  | 016 | so  | 017 | si  | 020 | dle | 021 | dcl | 022 | dc2 | 023 | dc3 | 024 | dc4 | 025 | nak | 026 | syn | 027 | etb | 030 | can | 031 | em | 032 | sub | 033 | esc | 034 | fs  | 035 | gs  | 036 | rs  | 037 | us  |
| bs        | 010 | bs  | 011 | ht  | 012 | nl  | 013 | vt  | 014 | np  | 015 | cr  | 016 | so  | 017 | si  | 020 | dle | 021 | dcl | 022 | dc2 | 023 | dc3 | 024 | dc4 | 025 | nak | 026 | syn | 027 | etb | 030 | can | 031 | em | 032 | sub | 033 | esc | 034 | fs  | 035 | gs  | 036 | rs  | 037 | us  |
| dle       | 020 | dle | 021 | dcl | 022 | dc2 | 023 | dc3 | 024 | dc4 | 025 | nak | 026 | syn | 027 | etb | 030 | can | 031 | em | 032 | sub | 033 | esc | 034 | fs  | 035 | gs  | 036 | rs  | 037 | us  |
| can       | 030 | can | 031 | em | 032 | sub | 033 | esc | 034 | fs  | 035 | gs  | 036 | rs  | 037 | us  |
| sp        | 040 | sp  | 041 | !   | 042 | "   | 043 | #   | 044 | $   | 045 | %   | 046 | &   | 047 | '   | 050 | (   | 051 | )   | 052 | *   | 053 | +   | 054 | ,   | 055 | -   | 056 | .   | 057 | /   | 060 | 0   | 061 | 1   | 062 | 2   | 063 | 3   | 064 | 4   | 065 | 5   | 066 | 6   | 067 | 7   | 070 | 8   | 071 | 9   | 072 | :   | 073 | ;   | 074 | <   | 075 | =   | 076 | >   | 077 | ?   | 100 | @  | 101 | A   | 102 | B   | 103 | C   | 104 | D   | 105 | E   | 106 | F   | 107 | G   | 110 | H  | 111 | I   | 112 | J   | 113 | K   | 114 | L   | 115 | M   | 116 | N   | 117 | O   | 120 | P  | 121 | Q   | 122 | R   | 123 | S   | 124 | T   | 125 | U   | 126 | V   | 127 | W   | 130 | X  | 131 | Y   | 132 | Z   | 133 | [   | 134 | \  | 135 | ]   | 136 | ^   | 137 | _   | 140 | \  | 141 | a   | 142 | b   | 143 | c   | 144 | d   | 145 | e   | 146 | f   | 147 | g   | 150 | h  | 151 | i   | 152 | j   | 153 | k   | 154 | l   | 155 | m   | 156 | n   | 157 | o   | 160 | p  | 161 | q   | 162 | r   | 163 | s   | 164 | t   | 165 | u   | 166 | v   | 167 | w   | 170 | x  | 171 | y   | 172 | z   | 173 | {   | 174 | |   | 175 | }   |

- Each character represented by 7-bit ASCII code.
- Packed into 8-bits
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 the data (and instructions)?
  • Registers  [more on these later]
    • In the processor core
    • Compute directly on them
    • Relatively few of them (~16-64)

• Memory
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
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 16, 32, 64 bit quantities
  • Byte ordering varies from system to system
Word of the Day: Endianess

Byte Order

- **Big Endian**: byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian**: byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC Alpha
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 downward
  - Heap: dynamically allocated variables; grows upward
- 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);
    printf("%d\n", z);
    return 0;
}
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?
```
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?

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

<table>
<thead>
<tr>
<th>swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>temp</td>
</tr>
<tr>
<td>RA</td>
</tr>
</tbody>
</table>
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?

---

© Daniel J. Sorin  
from Hilton, Lebeck, Lee, Roth  

ECE/CS 250
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>100</td>
</tr>
<tr>
<td>y</td>
<td>42</td>
</tr>
<tr>
<td>temp</td>
<td>42</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 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?
```

© Daniel J. Sorin
from Hilton, Lebeck, Lee, Roth
Let's do some different Java...

```java
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…

```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…

```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…

```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?
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?
```

© Daniel J. Sorin
from Hilton, Lebeck, Lee, Roth
ECE/CS 250
References and Pointers

• 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)
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

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

Then

```c
*p = 2;  and x = 2; produce the same result
```

Note: p is a pointer, *p is an int

- What happens when stating `p = 2`?

On 32-bit machine, p is 32 bits
On 64-bit machine, p is 64 bits

© Daniel J. Sorin
from Hilton, Lebeck, Lee, Roth
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));
  ```

• `sizeof` tells how many bytes something takes
• `malloc` takes number of bytes
• `malloc` returns pointer to first allocated byte
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 = (int*)malloc(100*sizeof(int));

| 0 | 1 | 32 | 33 | 98 | 99 |

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

| 0 | 1 | 33 | 199 |

*(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?

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

while (begin < end)
{
    *begin = 'z';
    begin++;
}
```
More Pointers & Arrays

```c
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]);
}
Strings as Arrays

- 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()

- `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
Memory Manager (Heap Manager)

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

Available Memory

Allocated Memory (part of this is data structures for managing memory)

© Daniel J. Sorin
from Hilton, Lebeck, Lee, Roth

ECE/CS 250
Summary: From C to Binary

- Everything must be represented in binary!
- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- We’ll visit these topics again throughout semester