Announcements

• Get your VM's installed.
  – Do's and Don'ts with your VM
    • Installing the 'Guest Additions' for the Linux VM
    • Backing up files
    • Not installing any updates to the VM

• Ch. 1-2 HW due by Thursday midnight

• Lab 1 review answers must be submitted on our website
  – Attend lab to meet your TAs and mentors and get help with lab 1 or your VM
Bonus Lecture Available at:

DIGITAL REPRESENTATION
(SKIP TO SLIDE 31)
Why 1’s and 0’s

- Transistors are electronic devices used to build computer hardware
  - Like a switch (2 positions)
  - Conducting / Non-conducting
  - Output voltage of a transistor will either be high or low
- 1’s and 0’s are arbitrary symbols representing high and low voltage outputs.
- 2 states of the transistor lead to only 2 values in computer hardware

![Functional View of a Transistor as a Switch](image)

1. **High Voltage**
   - +5V
   - +12V
   - or
2. **Low Voltage**
   - 0V
   - -12V

- Circuit Diagram of a Switch
  - Circuit is open (off) – no current can flow
  - Circuit is closed (on) – current can flow

The voltage here determines if current can flow between drain and source.
5 Kinds of Information

• Numbers
• Text
• Sound
• Image/Video
• Instructions
Integers and Floating Point

NUMBER REPRESENTATION
Interpreting Binary Strings

• Given a string of 1’s and 0’s, you need to know the *representation system* being used, before you can understand the value of those 1’s and 0’s.

• Information (value) = Bits + Context (System)
Binary Number System

• Humans use the decimal number system
  – Based on number 10
  – 10 digits: [0-9]

• Because computer hardware uses digital signals with 2 states, computers use the binary number system
  – Based on number 2
  – 2 binary digits (a.k.a bits): [0,1]
Number System Theory

- The written digits have implied place values
- Place values are powers of the base (decimal = 10)
- Place value of digit to left of decimal point is $10^0$ and ascend from there, negative powers of 10 to the right of the decimal point
- The value of the number is the sum of each digit times its implied place value

$$(852.7)_{10} = \text{base}$$
Number System Theory

- The written digits have implied place values
- Place values are powers of the base (decimal = 10)
- Place value of digit to left of decimal point is $10^0$ and ascend from there, negative powers of 10 to the right of the decimal point
- The value of the number is the sum of each digit times its implied place value

$$(852.7)_{10} = 8*10^2 + 5*10^1 + 2*10^0 + 7*10^{-1}$$
Binary Number System

- Place values are powers of 2
- The value of the number is the sum of each bit times its implied place value (power of 2)

\[(110.1)_2 = \]
Binary Number System

- Place values are powers of 2
- The value of the number is the sum of each bit times its implied place value (power of 2)

\[
(110.1)_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1}
\]

\[
(110.1)_2 = 1 \times 4 + 1 \times 2 + 1 \times 0.5 = 4 + 2 + 0.5 = 6.5_{10}
\]
Unique Combinations

• Given \( n \) digits of base \( r \), how many unique numbers can be formed? \( r^n \)

2-digit, decimal numbers

3-digit, decimal numbers

4-bit, binary numbers

6-bit, binary numbers

Main Point: Given \( n \) digits of base \( r \), \underline{unique numbers} can be made with the range \[ \underline{[\hspace{10cm}\hspace{10cm}]} \]
Unique Numbers

- Computers represent binary numbers using a fixed number of bits
- Given a fixed number of bits, $n$, what is the range of numbers we can make?

<table>
<thead>
<tr>
<th>If $n=1$ bit:</th>
<th>If $n=2$ bits:</th>
<th>If $n=3$ bits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 = 0</td>
<td>0 0 = 0</td>
<td>0 0 0 = 0</td>
</tr>
<tr>
<td>1 = 1</td>
<td>0 1 = 1</td>
<td>0 0 1 = 1</td>
</tr>
<tr>
<td></td>
<td>1 0 = 2</td>
<td>0 1 0 = 2</td>
</tr>
<tr>
<td></td>
<td>1 1 = 3</td>
<td>0 1 1 = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 1 = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 0 = 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 1 = 7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Given $n$ bits, $2^n$ numbers can be made.
Powers of 2

- Know (memorize) your binary place values

<table>
<thead>
<tr>
<th>n</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>
Now do you see why we told you to memorize your powers of 2

- Need to know them to convert binary to decimal, decimal to binary, and for finding how many numbers can be made with $n$-bits

<table>
<thead>
<tr>
<th>$n$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>
Approximating Large Powers of 2

• Often need to find decimal approximation of a large powers of 2 like \(2^{16}, 2^{32}\), etc.

• Use following approximations:
  – \(2^{10} \approx 10^3\) (1 thousand)
  – \(2^{20} \approx 10^6\) (1 million)
  – \(2^{30} \approx 10^9\) (1 billion)

• For other powers of 2, decompose into product of \(2^{10}\) or \(2^{20}\) or \(2^{30}\) and a power of 2 that is less than \(2^{10}\)

• See examples

\[
\begin{align*}
2^{16} &= 2^6 \times 2^{10} \\
&\approx 64 \times 10^3 = 64,000 \\
2^{24} &= 2^4 \times 2^{20} \\
&\approx 16 \times 10^6 = 16,000,000 \\
2^{28} &= 2^8 \times 2^{20} \\
&\approx 256 \times 10^6 = 256,000,000 \\
2^{32} &= 2^2 \times 2^{30} \\
&\approx 4 \times 10^9 = 4,000,000,000
\end{align*}
\]
C Integer Data Types

- In C/C++ variables can be of different types and sizes
  - Integer Types (signed by default...unsigned with leading keyword)

<table>
<thead>
<tr>
<th>C Type</th>
<th>Bytes</th>
<th>Bits</th>
<th>Signed Range</th>
<th>Unsigned Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>[unsigned] char</td>
<td>1</td>
<td>8</td>
<td>-128 to +127</td>
<td>0 to 255</td>
</tr>
<tr>
<td>[unsigned] short</td>
<td>2</td>
<td>16</td>
<td>-32768 to +32767</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>[unsigned] long int</td>
<td>4</td>
<td>32</td>
<td>-2 billion to +2 billion</td>
<td>0 to 4 billion</td>
</tr>
<tr>
<td>[unsigned] long long</td>
<td>8</td>
<td>64</td>
<td>-8<em>10^{18} to +8</em>10^{18}</td>
<td>0 to 16*10^{18}</td>
</tr>
</tbody>
</table>
What About Rational/Real #'s

• Previous binary system assumed binary point was fixed at the far right of the number
  – 10010. *(implied binary point)*

• Consider scientific notation:
  – Avogadro’s Number: +6.0247 * 10^{23}
  – Planck’s Constant: +6.6254 * 10^{-27}

• Can one representation scheme represent such a wide range?
  – Yes! **Floating Point**
    – Represents the sign, significant digits (fraction), exponent as separate bit fields

• Decimal: ±D.DDD * 10^{±exp}

• Binary: ±b.bbbbb * 2^{±exp}
C Floating Point Types

- **float** and **double types:**

<table>
<thead>
<tr>
<th>C Type</th>
<th>Bytes</th>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4</td>
<td>32</td>
<td>±7 significant digits * $10^{+/-38}$</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>64</td>
<td>±16 significant digits * $10^{+/-308}$</td>
</tr>
</tbody>
</table>
HEXADECIMAL REPRESENTATION
Hexadecimal Notation

- For humans working with long binary numbers it is easier to convert them to hexadecimal (a.k.a. hex for short)
- Hexadecimal is another number system based on the number 16 that has the following properties
  - Every 4-bits of binary translates to 1 hex digit and vice versa

<table>
<thead>
<tr>
<th>Hex digit</th>
<th>4-bit Binary Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>
Hex to Binary Translation

**Hex to Binary Translation**

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

**Binary to Hex Translation**

- **0001010011101100₂**: 14EC₁₆
- **11110110101₁₆**: FB5₁₆
- **1101100100111000₂**: D938₁₆

**Hex to Binary Translation**

- **14EC₁₆**: 0001010011101100₂
- **FB5₁₆**: 11110110101₁₆
- **D938₁₆**: 1101100100111000₂
ASCII and UNICODE

TEXT CHARACTERS
Text

- Text characters are usually represented with some kind of binary code (mapping of character to a binary number such as 'a' = 01100001 = 0x61)

- ASCII = Traditionally an 8-bit code
  - How many combinations (i.e. characters)?
  - English only

- UNICODE = 16-bit code
  - How many combinations?
  - Most languages w/ an alphabet
ASCII Code

- Used for representing text characters
- Originally 7-bits but usually stored as 8-bits in a computer
- Example:
  - `cout << "Hello\n";`
  - Each character is converted to ASCII equivalent
    - ‘H’ = 0x48, ‘e’ = 0x65, ...
    - ‘\n’ = newline character
      - CR = carriage return character (moves cursor to start of current line)
      - LF = line feed (moves cursor down a line)
UniCode

- ASCII can represent only the English alphabet, decimal digits, and punctuation
  - 7-bit code => $2^7 = 128$ characters
  - It would be nice to have one code that represented more alphabets/characters for common languages used around the world
- Unicode
  - 16-bit Code => 65,536 characters
  - Represents many languages alphabets and characters
  - Used by Java as standard character code

Unicode hex value
(i.e. FB52 => 1111101101010010)
INSTRUCTIONS
Instructions

• Tell the computer what operations to perform on the data

• Contains...
  – Operation to perform (OpCode)
  – Source operands (input data)
  – Destination location (where to put output)

\[ Z = X + Y \]

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Dst.</th>
<th>Src1</th>
<th>Src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Z</td>
<td>X</td>
<td>Y</td>
</tr>
</tbody>
</table>
Instruction Format Example

- Excerpt from Freescale’s Coldfire Processor Ref. Manual
  - Reference for an ADD instruction

### ADD

**Add**

First appeared in ISA_A

**Operation:** Source + Destination → Destination

**Assembler Syntax:**
- ADD.L <ea>y,Dx
- ADD.L Dy,<ea>x

**Instruction Format:**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</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>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>Register</td>
<td>Opmode</td>
<td>Effective Address</td>
<td>Mode</td>
<td>Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Fields:**

- Register field—Specifies the data register.
- Opmode field:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Word</th>
<th>Longword</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>—</td>
<td>010</td>
<td>&lt;ea&gt;y + Dx → Dx</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>110</td>
<td>Dy + &lt;ea&gt;x → &lt;ea&gt;x</td>
</tr>
</tbody>
</table>
DATA REPRESENTATION REVIEW
Memory

- Recall all information in a computer is stored in memory
- Memory consists of cells that each store a group of bits (usually, 1 byte = 8 bits)
- Unique address assigned to each cell
  - Used to reference the value in that location
- We first need to understand the various ways our program can represent data and allocate memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11010010</td>
</tr>
<tr>
<td>1</td>
<td>01001011</td>
</tr>
<tr>
<td>2</td>
<td>10010000</td>
</tr>
<tr>
<td>3</td>
<td>11110100</td>
</tr>
<tr>
<td>4</td>
<td>01101000</td>
</tr>
<tr>
<td>5</td>
<td>11010001</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1023</td>
<td>00001011</td>
</tr>
</tbody>
</table>

Memory Device

![Memory diagram with addresses and data values]
C Integer Data Types

• In C/C++ constants & variables can be of different types and sizes
  – A Type indicates how to interpret the bits and how much memory to allocate
  – Integer Types (signed by default...unsigned with leading keyword)

<table>
<thead>
<tr>
<th>C Type</th>
<th>Bytes</th>
<th>Bits</th>
<th>Signed Range</th>
<th>Unsigned Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>[unsigned] char</td>
<td>1</td>
<td>8</td>
<td>-128 to +127</td>
<td>0 to 255</td>
</tr>
<tr>
<td>[unsigned] short</td>
<td>2</td>
<td>16</td>
<td>-32768 to +32767</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>[unsigned] long int</td>
<td>4</td>
<td>32</td>
<td>-2 billion to +2 billion</td>
<td>0 to 4 billion</td>
</tr>
<tr>
<td>[unsigned] long long</td>
<td>8</td>
<td>64</td>
<td>-8<em>10^18 to +8</em>10^18</td>
<td>0 to 16*10^18</td>
</tr>
</tbody>
</table>
C Floating Point Types

- **float and double types:**
  
  Allow decimal representation (e.g. 6.125) as well as very large integers (+6.023E23)

<table>
<thead>
<tr>
<th>C Type</th>
<th>Bytes</th>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4</td>
<td>32</td>
<td>±7 significant digits * 10^+/−38</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>64</td>
<td>±16 significant digits * 10^+/−308</td>
</tr>
</tbody>
</table>
Text

- Text characters are usually represented with some kind of binary code (mapping of character to a binary number such as 'a' = 01100001 bin = 97 dec = 61 hex)

- ASCII = Traditionally an 8-bit code
  - How many combinations (i.e. characters)?
  - English only

- UNICODE = 16-bit code
  - How many combinations?
  - Most languages w/ an alphabet

http://www.theasciicode.com.ar/
UniCode

• ASCII can represent only the English alphabet, decimal digits, and punctuation
  – 7-bit code => $2^7 = 128$ characters
  – It would be nice to have one code that represented more alphabets/characters for common languages used around the world
• Unicode
  – 16-bit Code => 65,536 characters
  – Represents many languages alphabets and characters
  – Used by Java as standard character code

Unicode hex value
(i.e. FB52 => 111101101010010)
C CONSTANTS & DATA TYPES
Constants

- **Integer**: 496, 10005, -234
- **Double**: 12.0, -16., 0.23, -2.5E-1, 4e-2
- **Float**: 12.0F  // F = float vs. double
- **Characters appear in single quotes**
  - 'a', '5', 'B', '!', '\n', '\t', '\\', '\'
  - Non-printing special characters use "escape" sequence (i.e. preceded by a `\`)
  - '\n' = newline/enter, '\t' = tab

- **C-Strings**
  - Multiple characters between double quotes
    "hi\n", "12345\n", "b", "\tAns. is %d"
  - Ends with a '\0'=NULL character added as the last byte/character

- **Boolean (C++ only)**: true, false
  - Physical representation: 0 = false, (!= 0) = true
You're Just My Type

- Indicate which constants are matched with the correct type.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Right / Wrong</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>'a'</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>&quot;5.0&quot;</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>'5'</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>
## You're Just My Type

- Indicate which constants are matched with the correct type.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Right / Wrong</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>int</td>
<td>double (.0)</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>'a'</td>
<td>string</td>
<td>char</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>string</td>
<td>string (char * or char [])</td>
</tr>
<tr>
<td>5.</td>
<td>double</td>
<td>float/double (. = non-integer)</td>
</tr>
<tr>
<td>5</td>
<td>char</td>
<td>Int...but if you store 5 in a char variable it'd be okay</td>
</tr>
<tr>
<td>&quot;5.0&quot;</td>
<td>double</td>
<td>string (char * or char [])</td>
</tr>
<tr>
<td>'5'</td>
<td>int</td>
<td>char</td>
</tr>
</tbody>
</table>
Understanding ASCII and chars

- Characters can still be treated as numbers

```cpp
char c = 'a'; // same as char c = 97;
char c = 'a' + 1; // c now contains 'b' = 98;
cout << a << endl; // I will see 'b' on the screen

char c = '1'; // c contains decimal 49, not 1
// i.e. '1' not equal to 1

`c >= 'a' && c <= 'z'; // && means AND
// here we are checking if c
// contains a lower case letter
```
VARIABLES AND EXPRESSIONS
C/C++ Variables

• A computer program needs to operate on and produce data values which are stored in memory/RAM

• ‘Variables’ are just memory locations that are reserved to store a piece of data of specific size and type

• Programmer indicates what variables they want when they write their code
  – Difference: C requires declaring all variables at the beginning of a function before any operations. C++ relaxes this requirement.

• The computer will allocate memory for those variables when the program starts to run

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    char c;
    int x = 1564983;
    ...
}
```

Variables are declared in the C code
C/C++ Variables

- **Variables have a:**
  - **type** [int, char, unsigned int, float, double, etc.]
  - **name/identifier** that the programmer will use to reference the value in that memory location [e.g. x, myVariable, cs101_variable_name, etc.]
    - Lookup what legal characters can start and/or be part of a variable identifier/name (e.g. must start with A/a – Z/z or an underscore ‘_’, etc.)
    - Use descriptive names (e.g. numStudents, doneFlag)
    - Avoid cryptic names (myvar1, a_thing)
  - **location** [the address in memory where it is allocated]
  - **Value**

- **Reminder:** You must declare a variable before using it

---

**Code**

```c++
int quantity = 4;
double cost = 5.75;
cout << quantity*cost << endl;
```

**Values:**
- quantity: 1008412
- cost: 287144
- quantity*cost: 4
- cost: 5.75
C/C++ Variables

• Programmer will decide what variables are needed based on:
  – What values will be entered at run-time (don't know at compile-time)
  – What values will change over the course of execution
    • Variable to store the URL of your web-browser
    • A value that counts how many students have a grade > 70%
  – What values can we 'save' rather than computing over and over (i.e. compute it once & store it in a variable)
    • Need the value of $3x^2 - 4x + 5$ at two points in time during execution of the code (assume $x$ doesn’t change)...Compute & store the 1st time, access that variable the second time
  – Desire to make code more readable by decomposing computations into understandable chunks
    • Compute area of a rectangle with height = $(a^2 + 4a + 5)$ and width = $(5i^3 - 3i + 8)$...
    • Could write $(a^2 + 4a + 5) \times (5i^3 - 3i + 8)$
    • Might be more readable to put terms into separate variables ‘h’ and ‘w’ and then multiply result and place into a variable ‘area’ (double area = h * w;)

What Variables Might Be Needed

• Calculator
  – Current number input, current result

• TV
  – Current channel, volume level

• Tic-Tac-Toe
  – 9 square values, Whose turn?
Arithmetic Operators

- Addition, subtraction, multiplication work as expected for both integer and floating point types
- Division works ‘differently’ for integer vs. doubles/floats
- Modulus is only defined for integers

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>b + 5</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>c - x</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>a * 3.1e-2</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>10 / 3</td>
</tr>
<tr>
<td></td>
<td>(Integer vs. Double division)</td>
<td>(result will be 3)</td>
</tr>
<tr>
<td>%</td>
<td>Modulus (remainder)</td>
<td>17 % 5</td>
</tr>
<tr>
<td></td>
<td>[for integers only]</td>
<td>(result will be 2)</td>
</tr>
<tr>
<td>++ or --</td>
<td>Increment (add 1) or Decrement (subtract 1)</td>
<td>e++ (e = e+1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i-- (i = i-1)</td>
</tr>
</tbody>
</table>
Precedence

- Order of operations/evaluation of an expression
- Top Priority = highest (done first)
- Notice operations with the same level or precedence usually are evaluated left to right (explained at bottom)

Evaluate:
- \(2 \times -4 - 3 + 5 / 2\);

Tips:
- Use parenthesis to add clarity
- Add a space between literals \((2 \times -4) - 3 + (5 / 2)\)

Operators (grouped by precedence)

<table>
<thead>
<tr>
<th>Structure/Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct member operator</td>
<td>name.member</td>
</tr>
<tr>
<td>struct member through pointer</td>
<td>pointer-&gt;member</td>
</tr>
<tr>
<td>increment, decrement</td>
<td>++, --</td>
</tr>
<tr>
<td>plus, minus, logical not, bitwise not</td>
<td>+, -, !, ~</td>
</tr>
<tr>
<td>indirection via pointer, address of object</td>
<td>*pointer, &amp;name (type) expr sizeof</td>
</tr>
<tr>
<td>cast expression to type size of an object</td>
<td></td>
</tr>
<tr>
<td>multiply, divide, modulus (remainder)</td>
<td>*, /, %</td>
</tr>
<tr>
<td>add, subtract</td>
<td>+, -</td>
</tr>
<tr>
<td>left, right shift [bit ops]</td>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
<tr>
<td>relational comparisons</td>
<td>&gt;, &gt;=, &lt;, &lt;=</td>
</tr>
<tr>
<td>equality comparisons</td>
<td>==, !=</td>
</tr>
<tr>
<td>and [bit op]</td>
<td>&amp;</td>
</tr>
<tr>
<td>exclusive or [bit op]</td>
<td>^</td>
</tr>
<tr>
<td>or (inclusive) [bit op]</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>conditional expression</td>
<td>expr1 ? expr2 : expr3</td>
</tr>
<tr>
<td>assignment operators</td>
<td>+=, -=, *=, ...</td>
</tr>
<tr>
<td>expression evaluation separator</td>
<td>,</td>
</tr>
</tbody>
</table>

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

January 2007 v2.2. Copyright © 2007 Joseph H. Silverman
Permission is granted to make and distribute copies of this card provided the copyright notice and this permission notice are preserved on all copies.
Send comments and corrections to J.H. Silverman, Math. Dept., Brown Univ., Providence, RI 02912 USA. hjhs@math.brown.edu
Evaluate 5 + 3/2

• The answer is 6.5 ??
Casting

• To achieve the correct answer for \( 5 + 3 / 2 \)
• Could make everything a double
  – Write \( 5.0 + 3.0 / 2.0 \) [explicitly use doubles]
• Could use **implicit** casting (mixed expression)
  – Could just write \( 5 + 3.0 / 2 \)
    • If operator is applied to mixed type inputs, less expressive type is automatically promoted to more expressive (int => double)
• Could use C or C++ syntax for **explicit** casting
  – \( 5 + \text{(double)} 3 / \text{(double)} 2 \) (C-Style cast)
  – \( 5 + \text{static	extunderscore cast\textless double\textgreater}(3) / \text{static	extunderscore cast\textless double\textgreater}(2) \) (C++-Style cast)
  – \( 5 + \text{static	extunderscore cast\textless double\textgreater}(3) / 2 \) (can cast only one, rely on implicit cast of the other)
  – This looks like a lot of typing compared to just writing \( 5 + 3.0 / 2 \)...but what if instead of constants we have variables
  – int \( x=5, y=3, z=2; \quad x + y/z; \)
  – \( x + \text{static	extunderscore cast\textless double\textgreater}(y) / z \)
Exercise Review

• D.S. Malik, C++ Programming, 5th Ed., Ch. 2
  – Q6:
    • 25/3
    • 20-12/4*2
    • 33 % 7
    • 3 – 5 % 7
    • 18.0 / 4
    • 28 - 5 / 2.0
    • 17 + 5 % 2 - 3
Assignment operator ‘=’

- Syntax:
  
  \[
  \text{variable} = \text{expression};
  \]

  (LHS) (RHS)

- LHS = Left Hand-Side, RHS = Right Hand Side
- Should be read: Place the value of expression into memory location of variable
  - \( z = x + y - (2 \times z) \);
- When variables appearing on RHS indicate the use of their associated value. Variables on LHS indicate location to place a value.
- **Note:** Without assignment values are computed and then forgotten
  - \( x + 5 \) will take \( x \)'s value add 5 but NOT update \( x \) (just throws the result away)
  - \( x = x + 5 \) will actually updated \( x \) (i.e. requires an assignment)
- Shorthand operators for updating a variable based on its current value:
  - \( +=, \ -=, \ *=, \ /=, \ &=, \ ... \)
  - \( x += 5; \quad (x = x+5) \)
  - \( y *= x; \quad (y = y\times x) \)
More Assignments

- Assigning a variable makes a copy
- Challenge: Make a copy

```c
int main()
{
    int x = 5, y = 3;
    x = y;    // copy y into x

    // now consider swapping
    // the value of 2 variables
    int a = 7, b = 9;
    a = b;
    b = a;

    return 0;
}
```
More Assignments

• Assigning a variable makes a copy
• Challenge: Make a copy
  – Easiest method: Use a 3rd temporary variable to save one value and then replace that variable

```c
int main()
{
    int x = 5, y = 3;
    x = y;   // copy y into x

    // let's try again
    int a = 7, b = 9, temp;
    temp = a;
    a = b;
    b = temp;

    return 0;
}
```
Summary Examples 1

• [http://bits.usc.edu/cs103/in-class-exercises](http://bits.usc.edu/cs103/in-class-exercises)
  - maxplus
  - 4swap

• Full Links
Function call statements

• C++ predefines a variety of functions for you. Here are a few of them:
  – `sqrt(x)`: returns the square root of x (in `<cmath>`)
  – `pow(x, y)`: returns $x^y$, or x to the power y (in `<cmath>`)
  – `sin(x)`: returns the sine of x if x is in radians (in `<cmath>`)
  – `abs(x)`: returns the absolute value of x (in `<cstdlib>`)
  – `max(x, y)`: returns the maximum of x and y (in `<algorithm>`)
  – `min(x, y)`: returns the maximum of x and y (in `<algorithm>`)

• You call these by writing them similarly to how you would use a function in mathematics:

```cpp
#include <iostream>
#include <cmath>
#include <algorithm>
using namespace std;

int main(int argc, char *argv[]) {
    // can call functions
    // in an assignment
    double res = cos(0);

    // can call functions in an expression
    res = sqrt(2) + 2.3 << endl;

    // can call them as part of a output statement
    cout << max(34, 56) << endl;
    return 0;
}
```
Statements

- End with a semicolon ‘;’
- **Assignment** (use initial conditions of `int x=3; int y;`)
  - `x = x * 5 / 9; // compute the expression & place result in x`
  - `x = (3*5)/9 = 15/9 = 1`
- **Function Call**
  - `sin(3.14); // Beware of just calling a function w/o assignment`
  - `x = cos(0.0);`
- Mixture of assignments, expressions and/or function calls
  - `x = x * y – 5 + max(5,9);`
- **Return statement** (immediate ends a function)
  - `return x+y;`
Pre- and Post-Increment

-++ and -- operator and be used in an expression and cause the associated variable to "increment-by-1" or "decrement-by-1"

-Timing of the increment or decrement depends on whether pre- or post-increment (or decrement) is used
  - y = x++ + 5; // Post-increment since ++ is used after the variable x
  - y = ++x + 5; // Pre-decrement since ++ is used before the variable x

-Meaning:
  - Pre: Update the variable before using it in the expression
  - Post: Use the old value of the variable in the expression then update it

-Examples [suppose int y; int x = 3; ]
  - y = x++ + 5; // Use old value of x and subtract 5, but add 1 to x after computing result
    // Result: y = 8, x = 4
  - y = ++x + 5; // Increment x and use its new value when you subtract 5
    // [Result: y = 9, x = 4]
  - y = x-- + 5; // Use old value of x and add 5, but subtract 1 from x after evaluating the expression  [Result: y = 8, x = 2]
Exercise

• Consider the code below
  – `int x=5, y=7, z;`  
  – `z = x++ + 3*--y + 2*x++;`

• What is the value of x, y, and z after this code executes
I/O Streams

- I/O is placed in temporary buffers/streams by the OS/C++ libraries
- `cin` goes and gets data from the input stream (skipping over preceding whitespace then stopping at following whitespace)
- `cout` puts data into the output stream for display by the OS (a flush forces the OS to display the contents immediately)

**input stream:**
```
7 5 y ...
```

```cpp
#include<iostream>
using namespace std;
int main(int argc, char * argv[])
{
  int x; cin << x;
  return 0;
}
```

**output stream:**
```
It was the
```

**output stream:**
```
4
```

**input stream:**
```
y ...
```

```cpp
#include<iostream>
using namespace std;
int main(int argc, char * argv[])
{
  cout << "It was the" << endl;
  cout << "4";
}
```
C++ I/O

- Include `<iostream>` (not `iostream.h`)
- Add `using namespace std;` at top of file
- `cout` (character output) object used to print to the monitor
  - Use the `<<` operator to separate any number of variables or constants you want printed
  - Compiler uses the implied type of the variable to determine how to print it out
  - `endl` constant can be used for the newline character (`\n`) though you can still use `\n` as well.
    - `endl` also ‘flushes’ the buffer/stream (forces the OS to show the text on the screen) which can be important in many contexts.

```cpp
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int x = 5;
    char c = 'Y';
    double y = 4.5;
    cout << "Hello world" << endl;
    cout << "x = " << x << " c = ";
    cout << c << "\ny is " << y << endl;
    return 0;
}
```

Output from program:
Hello world
x = 5 c = Y
y is 4.5
C++ I/O

- ‘cin’ (character input) object used to accept input from the user and write the value into a variable
  - Use the ‘>>’ operator to separate any number of variables or constants you want to read in
  - Every '>>' means will skip over any leading whitespace looking for text it can convert to the variable form, then stop at the trailing whitespace

```cpp
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char *argv[]) {
    int x;
    char c;
    string mystr;
    double y;

    cout << "Enter an integer, character, string, and double separated by spaces:" << endl;
    cin >> x >> c >> mystr >> y;
    cout << "x = " << x << " c = ";
    cout << c << " mystr is " << mystr;
    cout << "y is " << y << endl;
    return 0;
}
```

Output from program:

```
Enter an integer, character, string, and double separated by spaces:
5 Y hi 4.5
x = 5 c = Y mystr is hi y is 4.5
```
#include<iostream>
using namespace std;

int main(int argc, char *argv[])
{
    char myc = 0;
    double y = 0.0;
    cin >> myc >> y;
}

• If the user types in

```
   a t 3.5 
```

myc = 'a'    y = 0.0

• After the first '>>'

```
   a t 3.5 
```

myc = 'a'    y = 0.0

• After the second '>>'

```
   \n```

myc = 'a'    y = 3.5
A Few Odds and Ends

• Comments
  – Anywhere in the code
  – C-Style => “/*” and “*/”
  – C++ Style => “//”

• Variable
  – When declared they will have "garbage" (random or unknown) values unless you initialize them
  – Each variable must be initialized separately

• Scope
  – Global variables are visible to all the code/functions in the program and are declared outside of any function
  – Local variables are declared inside of a function and are only visible in that function and die when the function ends

/* Anything between slash-star and star-slash is ignored even across multiple lines of text or code */
/*----Section 1: Compiler Directives ----*/
#include <iostream>
#include <cmath>
using namespace std;

// Global Variables
int x; // Anything after "//" is ignored

int add_1 (int input)
{
  return (input + 1);
}

int main(int argc, char *argv[])
{
  // y and z are "local" variables
  int y, z=5; // y is garbage, z is five

  z = add_1(z);
  y = z+1; // an assignment stmt
  cout << y << endl;
  return 0;
}
Summary Examples 2

- [http://bits.usc.edu/cs103/in-class-exercises](http://bits.usc.edu/cs103/in-class-exercises)
  - funccall
  - Tacos
  - Hello
  - Quadratic
  - Math

- **Full Links**
C PROGRAM STRUCTURE AND COMPILATION
C Program Format/Structure

• Comments
  – Anywhere in the code
  – C-Style => "/*" and "*/"
  – C++ Style => "//"

• Compiler Directives
  – #includes tell compiler what other library functions you plan on using
  – 'using namespace std;' -- Just do it for now!

• Global variables (more on this later)

• main() function
  – Starting point of execution for the program
  – Variable declarations often appear at the start of a function
  – All code/statements in C must be inside a function
  – Statements execute one after the next
  – Ends with a 'return' statement

• Other functions

/* Anything between slash-star and star-slash is ignored even across multiple lines of text or code */
/*-----Section 1: Compiler Directives-----*/
#include <iostream>
#include <cmath>
using namespace std;

/*------------ Section 2 ----------------*/
/*Global variables & Function Prototypes */
int x; // Anything after "//" is ignored
void other_unused_function();

/*-----Section 3: Function Definitions----*/
void other_unused_function()
{
  cout << "No one uses me!" << endl;
}

int main(int argc, char *argv[])
{
  int y; // a variable declaration stmt
  y = 5+1; // an assignment stmt
  cout << y << endl;
  return 0;
}
Software Process

#include <iostream>
using namespace std;

int main()
{
  int x = 5;
  cout << "Hello"
       << endl;
  cout << "x=" << x;
  return 0;
}

C++ file(s)
(test.cpp)

Compile & fix compiler errors
- g = Enable Debugging
- Wall = Show all warnings
- o test = Specify Output executable name

1. Edit & write code
$ gedit test.cpp &

2. Compile & fix compiler errors
$ gedit test.cpp &
$ g++ -g -Wall -o test test.cpp
or
$ make test

3. Load & run the executable program
$ gedit test.cpp &
$ g++ -g -Wall -o test test.cpp
$ ./test
Software Process

#include <iostream>
using namespace std;

int main()
{
    int x = 5;
    cout << "Hello"
         << endl;
    cout << "x=" << x;
    return 0;
}

1. Edit & write code
2. Compile & fix errors
3. Load & run the executable program

- g = Enable Debugging
- Wall = Show all warnings
- o test = Specify Output executable name

Fix compile-time errors w/ a debugger
Fix run-time errors w/ a debugger

C++ file(s)
(test.cpp)

Compiler

Executable Binary Image
(test)

Load & Execute

Std C++ & Other Libraries

Binary Image (test)

1110 0010 0101 1001
0110 1011 0000 1100
0100 1101 0111 1111
1010 1100 0010 1011
0001 0110 0011 1000

$ gedit test.cpp &
$ g++ -g -Wall -o test test.cpp
or
$ make test

$ gedit test.cpp &
$ g++ -g -Wall -o test test.cpp
$ ./test

$ gedit test.cpp &
$ g++ -g -Wall -o test test.cpp
$ make test

$ g++ -g -Wall -o test test.cpp
$ ./test
gdb / ddd / kdbg

- To debug your program you must have compiled with the ‘–g’ tag in g++ (i.e. g++ –g –Wall –o test test.cpp).
- gdb is the main workhorse of Unix/Linux debuggers (but it is text-based while 'ddd' and 'kdbg' are graphical based debuggers)
  - Run using: $ gdb ./test
- Allows you to...
  - Set breakpoints (a point in the code where your program will be stopped so you can inspect something of interest)
    - 'break 7' will cause the program to halt on line 7
  - Run: Will start the program running until it hits a breakpoint of completes
  - Step: Execute next line of code
  - Next: Like ‘Step’ but if you are at a function step will go into that function while ‘Next’ will run the function stopping at the next line of code
  - Print variable values ('print x')
Memory Operations

- Memories perform 2 operations
  - Read: retrieves data value in a particular location (specified using the address)
  - Write: changes data in a location to a new value

- To perform these operations a set of address, data, and control inputs/outputs are used
  - Note: A group of wires/signals is referred to as a ‘bus’
  - Thus, we say that memories have an address, data, and control bus.
Activity 1

• Consider the code below & memory layout
  – int x=5, y=7, z=1;
  – z = x + y - z;

• Order the memory activities & choose Read or Write
  1. R / W value @ addr. 0x01008
  2. Allocate & init. memory for x, y, & z
  3. Read value @ addr. 0x01000
  4. Write value @ addr. 0x01000
  5. R / W value @ addr. 0x01004

• Answer: 2, 1(R), 5(R), 3, 4