CS 103 Lecture 2 Slides

C/C++ Basics

Mark Redekopp
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

• HW 1

• 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
A quick high-level view before we dive into the details...

PROGRAM STRUCTURE AND COMPILEDATION PROCESS
C/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!

• main() function
  – **Starting point of execution** for the program
  – All code/statements in C must be inside a function
  – Statements execute one after the next and end with a semicolon (;)
  – Ends with a 'return 0;' statement

• Other functions
  – printName() is a function that can be "called"/"invoked" from main or any other function

/* Anything between slash-star and star-slash is ignored even across multiple lines of text or code */

// Anything after "//" is ignored on a line

// #includes allow access to library functions
#include <iostream>
#include <cmath>
using namespace std;

void printName()
{
  cout << "Tommy Trojan" << endl;
}

// Execution always starts at the main() function
int main()
{
  cout << "Hello: " << endl;
  printName();
  printName();
  return 0;
}

Hello:
Tommy Trojan
Tommy Trojan
# Software Process

1. **Edit & write code**
   
   ```
   #include <iostream>
   using namespace std;
   int main()
   {
       int x = 5;
       cout << "Hello"
            << endl;
       cout << "x=" << x;
       return 0;
   }
   ```

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

3. **Load & run the executable program**
   
   ```
   $ ./test
   ```
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)

$ gedit test.cpp &

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

Fix compile-time errors w/ a debugger

Load & run the executable program

1 Edit & write code
2 Compile & fix compiler errors
3 Fix run-time errors w/ a debugger

Compiler

Clang++

Executable Binary Image (test)

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

Std C++ & Other Libraries

Load & Execute

- 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
DATA REPRESENTATION
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
Starting With Numbers

• A single bit can only represent 1 and 0

• To represent more than just 2 values we need to use combinations/sequences of many bits
  – A byte is defined as a group 8-bits
  – A word varies in size but is usually 32-bits

• So how do we interpret those sequences of bits?
  – Let's learn about number systems
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 values, computers use the binary number system
  – Based on number 2
  – 2 binary digits (a.k.a bits): [0,1]
Number System Theory

• Let's understand how number systems work by examining decimal and then moving to binary
• 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

$$(852.7)_{10} =$$
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

\[(852.7)_{10} = 8 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 + 7 \times 10^{-1}\]
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)

\[
(110.1)_2 = \]

\[
(11010)_2 =
\]
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)

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

\[
(110.1)_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 2^2 + 2^1 + 2^{-1} = 4 + 2 + 0.5 = 6.5_{10}
\]

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

\[
(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^1 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26_{10}
\]
Unique Combinations

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of Digits</th>
<th>Base</th>
<th>Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-digit, decimal</td>
<td>2</td>
<td>0-9</td>
<td>___</td>
</tr>
<tr>
<td>3-digit, decimal</td>
<td>3</td>
<td>0-9</td>
<td>___</td>
</tr>
<tr>
<td>4-bit, binary</td>
<td>4</td>
<td>0-1</td>
<td>___</td>
</tr>
<tr>
<td>6-bit, binary</td>
<td>6</td>
<td>0-1</td>
<td>___</td>
</tr>
</tbody>
</table>

Main Point: Given \( n \) digits of base \( r \), \_\_\_ unique numbers can be made with the range [\_\_\_\_\_\_]
Sign

• Is there any limitation if we only use the powers of some base as our weights?
  – Can't make negative numbers

• What if we change things
  – How do humans represent negative numbers?
  – Can we do something similar?
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 optional 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.bbbb * 2^{±exp}

<table>
<thead>
<tr>
<th>S</th>
<th>Exp.</th>
<th>fraction</th>
</tr>
</thead>
</table>

Overall Sign of #
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)
- 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
- In C/C++ a single printing/text character must appear between single-quotes ('')
  - Example: 'a', '!', 'Z'

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
  - Won't be used in our course

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

01000001 = ?

- 65₁₀
- 41₃₂
- ‘A’₁₆

Unsigned Binary system
BCD System
ASCII system
C CONSTANTS & DATA TYPES
What's Your Type

What am I storing?

Number

Text/Character(s) for display

Logical (true/false) value

What kind of number is it?

Contains a decimal/fractional value

Integer

What range of values might it use?

Positive only

Possible negative

Use a…

double

3.0, 3.14159, 6.27e23

Use an…

unsigned int

0, 2147682...

Use an…

int

0, -2147682, 2147682

Is it a single char or many (i.e. a string of chars)?

Single

Many

Use a…

Use a…

char

's', '1', '

string

"Hi", "2020"

Use a…

bool

ture, false
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
    "hi1\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>
EXPRESSIONS & VARIABLES
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>2 + 5</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>41 - 32</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>4.23 * 3.1e-2</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>10 / 3 (=3)</td>
</tr>
<tr>
<td></td>
<td>(Integer vs. Double division)</td>
<td>10.0 / 3 (=3.3333)</td>
</tr>
<tr>
<td>%</td>
<td>Modulus (remainder) [for integers only]</td>
<td>17 % 5</td>
</tr>
<tr>
<td></td>
<td>(result will be 2)</td>
<td></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 * -4) – 3 + (5 / 2)

Operators (grouped by precedence)

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator</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</td>
</tr>
<tr>
<td>cast expression to type size of an object</td>
<td>(type) expr</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.
Exercise Review

• Evaluate the following:
  – 25 / 3
  – 33 % 7
  – 17 + 5 % 2 – 3

C/C++ Variables

• A computer program needs to operate on and store data values (which are usually inputted from the user)

• 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 reaches the declaration

```c
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
    char c;
    int feet = 50;
    ...
    int inches = 12 * feet;
}
```

Variables must be declared before being used.

Variables are actually allocated in RAM when the program is run.

<table>
<thead>
<tr>
<th>char c; A single-byte variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 11010010</td>
</tr>
<tr>
<td>1 01001011</td>
</tr>
<tr>
<td>2 10010000</td>
</tr>
<tr>
<td>3 11110100</td>
</tr>
<tr>
<td>4 01101000</td>
</tr>
<tr>
<td>5 11010001</td>
</tr>
<tr>
<td>6 01101000</td>
</tr>
<tr>
<td>7 11010001</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>1023 00001011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int x = 1564983; A four-byte variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 11110100</td>
</tr>
<tr>
<td>1 01001011</td>
</tr>
<tr>
<td>2 10010000</td>
</tr>
<tr>
<td>3 11110100</td>
</tr>
<tr>
<td>4 01101000</td>
</tr>
<tr>
<td>5 11010001</td>
</tr>
<tr>
<td>6 01101000</td>
</tr>
<tr>
<td>7 11010001</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>1023 00001011</td>
</tr>
</tbody>
</table>
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`, `num_dozens`, etc.]
    - Identifiers must start with `[A-Z, a-z, or an underscore '_'`] and can then contain any alphanumeric character `[0-9A-Za-z]` (but no punctuation other than underscores)
    - 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

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

### What's in a name?
To give descriptive names we often need to use more than 1 word/term. But we can't use spaces in our identifier names. Thus, most programmers use either camel-case or snake-case to write compound names

**Camel case**: Capitalize the first letter of each word (with the possible exception of the first word)
- `myVariable`, `isHighEnough`

**Snake case**: Separate each word with an underscore `_`
- `my_variable`, `is_high_enough`
When To Introduce a Variable

• When a value will be supplied and/or change at run-time (as the program executes)

• When a value is computed/updated at one time and used (many times) later

• To make the code more readable by another human

```java
double area = (56+34) * (81*6.25);
// readability of above vs. below
double height = 56 + 34;
double width = 81 * 6.25;
double area = height * width;
```
Assignment operator ‘=’

- Syntax:
  
  \[
  \text{variable} = \text{expression};
  \]
  
  (LHS) \quad \leftrightarrow \quad (RHS)
  
  - LHS = Left Hand-Side, RHS = Right Hand Side

- Should be read: Place the value of \text{expression} into memory location of \text{variable}
  
  - \text{z} = \text{x + y - (2*z)};
  - Evaluate RHS first, then place the result into the variable on the LHS
  - If variable is on both sides, we use the old/current value of the variable on the RHS

- **Note**: Without assignment values are computed and then forgotten
  
  - \text{x + 5}; \quad // \text{will take x's value add 5 but NOT update x (just throws the result away)}
  - \text{x = x + 5}; \quad // \text{will actually updated x (i.e. requires an assignment)}

- Shorthand assignment operators for updating a variable based on its current value: +=, -=, *=, /=, &=, ...
  
  - \text{x += 5}; \quad (x = x+5)
  - \text{y *= x}; \quad (y = y*x)
Evaluate 5 + 3/2

• The answer is 6.5 ??
Casting

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

```cpp
#include<iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    return 0;
}
```

```cpp
#include<iostream>
using namespace std;
int main()
{
    cout << "It was the" << endl;
    cout << "best of times."
}
```

```
#include<iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{
    cout << "It was the" << endl;
    cout << "best of times."
}
```

```
#include<iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{
    cout << "It was the" << endl;
    cout << "best of times."
}
```
C++ Output

- 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++ Input

- '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 '>>' 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
```
cin

\[ \text{myc} = \begin{array}{|c|} \hline 0 \\ \hline \end{array} \quad \text{y} = \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} \]

- If the user types in

\[ \begin{array}{|c|} \hline a \t 3.5 \n \hline \end{array} \]

- After the first `\text{\textgreater\textgreater}'

\[ \begin{array}{|c|} \hline 'a' \t 3.5 \n \hline \end{array} \]

- After the second `\text{\textgreater\textgreater}'

\[ \begin{array}{|c|} \hline 'a' \t 3.5 \n \hline \end{array} \]

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

int main()
{
    char myc = 0;
    double y = 0.0;
    cin >> myc >> y;
    // use the variables somehow...
    return 0;
}
```

Cin... skips leading whitespace; stops at trailing whitespace.
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); // res = 1.0

    // can call functions in an
    // expression
    res = sqrt(2) / 2; // res = 1.414/2

    cout << max(34, 56) << endl; // outputs 56
    return 0;
}
```
Statements

• C/C++ programs are composed of statements
• Most common kinds of 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 (immediately ends a function)
  – `return x+y;`
Understanding ASCII and chars

- Characters can still be treated as numbers

```c
char c = 'a'; // same as char c = 97;
char d = 'a' + 1; // c now contains 'b' = 98;
cout << d << 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
```

### ASCII printable characters

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code> </code></td>
<td>32</td>
</tr>
<tr>
<td><code>!</code></td>
<td>33</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>34</td>
</tr>
<tr>
<td><code>#</code></td>
<td>35</td>
</tr>
<tr>
<td><code>$</code></td>
<td>36</td>
</tr>
<tr>
<td><code>%</code></td>
<td>37</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>38</td>
</tr>
<tr>
<td><code>'</code></td>
<td>39</td>
</tr>
<tr>
<td><code>(</code></td>
<td>40</td>
</tr>
<tr>
<td><code>)</code></td>
<td>41</td>
</tr>
<tr>
<td><code>*</code></td>
<td>42</td>
</tr>
<tr>
<td><code>+</code></td>
<td>43</td>
</tr>
<tr>
<td><code>,</code></td>
<td>44</td>
</tr>
<tr>
<td><code>.</code></td>
<td>45</td>
</tr>
<tr>
<td><code>/</code></td>
<td>46</td>
</tr>
<tr>
<td><code>0</code></td>
<td>48</td>
</tr>
<tr>
<td><code>1</code></td>
<td>49</td>
</tr>
<tr>
<td><code>2</code></td>
<td>50</td>
</tr>
<tr>
<td><code>3</code></td>
<td>51</td>
</tr>
<tr>
<td><code>4</code></td>
<td>52</td>
</tr>
<tr>
<td><code>5</code></td>
<td>53</td>
</tr>
<tr>
<td><code>6</code></td>
<td>54</td>
</tr>
<tr>
<td><code>7</code></td>
<td>55</td>
</tr>
<tr>
<td><code>8</code></td>
<td>56</td>
</tr>
<tr>
<td><code>9</code></td>
<td>57</td>
</tr>
<tr>
<td><code>;</code></td>
<td>58</td>
</tr>
<tr>
<td><code>:</code></td>
<td>59</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>60</td>
</tr>
<tr>
<td><code>=</code></td>
<td>61</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>62</td>
</tr>
<tr>
<td><code>?</code></td>
<td>63</td>
</tr>
<tr>
<td><code>@</code></td>
<td>64</td>
</tr>
<tr>
<td><code>A</code></td>
<td>65</td>
</tr>
<tr>
<td><code>B</code></td>
<td>66</td>
</tr>
<tr>
<td><code>C</code></td>
<td>67</td>
</tr>
<tr>
<td><code>D</code></td>
<td>68</td>
</tr>
<tr>
<td><code>E</code></td>
<td>69</td>
</tr>
<tr>
<td><code>F</code></td>
<td>70</td>
</tr>
<tr>
<td><code>G</code></td>
<td>71</td>
</tr>
<tr>
<td><code>H</code></td>
<td>72</td>
</tr>
<tr>
<td><code>I</code></td>
<td>73</td>
</tr>
<tr>
<td><code>J</code></td>
<td>74</td>
</tr>
<tr>
<td><code>K</code></td>
<td>75</td>
</tr>
<tr>
<td><code>L</code></td>
<td>76</td>
</tr>
<tr>
<td><code>M</code></td>
<td>77</td>
</tr>
<tr>
<td><code>N</code></td>
<td>78</td>
</tr>
<tr>
<td><code>O</code></td>
<td>79</td>
</tr>
<tr>
<td><code>P</code></td>
<td>80</td>
</tr>
<tr>
<td><code>Q</code></td>
<td>81</td>
</tr>
<tr>
<td><code>R</code></td>
<td>82</td>
</tr>
<tr>
<td><code>S</code></td>
<td>83</td>
</tr>
<tr>
<td><code>T</code></td>
<td>84</td>
</tr>
<tr>
<td><code>U</code></td>
<td>85</td>
</tr>
<tr>
<td><code>V</code></td>
<td>86</td>
</tr>
<tr>
<td><code>W</code></td>
<td>87</td>
</tr>
<tr>
<td><code>X</code></td>
<td>88</td>
</tr>
<tr>
<td><code>Y</code></td>
<td>89</td>
</tr>
<tr>
<td><code>Z</code></td>
<td>90</td>
</tr>
<tr>
<td><code>[</code></td>
<td>91</td>
</tr>
<tr>
<td><code>{</code></td>
<td>92</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>}</code></td>
<td>125</td>
</tr>
<tr>
<td><code>^</code></td>
<td>94</td>
</tr>
<tr>
<td><code>_</code></td>
<td>95</td>
</tr>
<tr>
<td><code> </code></td>
<td>32</td>
</tr>
<tr>
<td><code>a</code></td>
<td>97</td>
</tr>
<tr>
<td><code>b</code></td>
<td>98</td>
</tr>
<tr>
<td><code>c</code></td>
<td>99</td>
</tr>
<tr>
<td><code>d</code></td>
<td>100</td>
</tr>
<tr>
<td><code>e</code></td>
<td>101</td>
</tr>
<tr>
<td><code>f</code></td>
<td>102</td>
</tr>
<tr>
<td><code>g</code></td>
<td>103</td>
</tr>
<tr>
<td><code>h</code></td>
<td>104</td>
</tr>
<tr>
<td><code>i</code></td>
<td>105</td>
</tr>
<tr>
<td><code>j</code></td>
<td>106</td>
</tr>
<tr>
<td><code>k</code></td>
<td>107</td>
</tr>
<tr>
<td><code>l</code></td>
<td>108</td>
</tr>
<tr>
<td><code>m</code></td>
<td>109</td>
</tr>
<tr>
<td><code>n</code></td>
<td>110</td>
</tr>
<tr>
<td><code>o</code></td>
<td>111</td>
</tr>
<tr>
<td><code>p</code></td>
<td>112</td>
</tr>
<tr>
<td><code>q</code></td>
<td>113</td>
</tr>
<tr>
<td><code>r</code></td>
<td>114</td>
</tr>
<tr>
<td><code>s</code></td>
<td>115</td>
</tr>
<tr>
<td><code>t</code></td>
<td>116</td>
</tr>
<tr>
<td><code>u</code></td>
<td>117</td>
</tr>
<tr>
<td><code>v</code></td>
<td>118</td>
</tr>
<tr>
<td><code>w</code></td>
<td>119</td>
</tr>
<tr>
<td><code>x</code></td>
<td>120</td>
</tr>
<tr>
<td><code>y</code></td>
<td>121</td>
</tr>
<tr>
<td><code>z</code></td>
<td>122</td>
</tr>
<tr>
<td><code> </code></td>
<td>32</td>
</tr>
<tr>
<td><code> </code></td>
<td>32</td>
</tr>
<tr>
<td><code> </code></td>
<td>32</td>
</tr>
</tbody>
</table>
In-Class Exercises

• Checkpoint 1
LECTURE 2 / LECTURE 3 END POINT
Assignment Means Copy

- Assigning a variable makes a copy
- Challenge: Swap the value of 2 variables

```c
int main()
{
    int x = 5, y = 3;
    x = y;    // copy y into x
    return 0;
}
```

```c
int main()
{
    int a = 7, b = 9;
    // now consider swapping
    // the value of 2 variables
    a = b;
    b = a;
    return 0;
}
```
More Assignments

• Assigning a variable makes a copy
• Challenge: Swap the value of 2 variables
  – Easiest method: Use a 3rd temporary variable to save one value and then replace that variable

```c
int main()
{
    int a = 7, b = 9, temp;
    // let's try again
    temp = a;
    a = b;
    b = temp;
    return 0;
}
```
A Few Odds and Ends

• Variable Initialization
  – 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

/*----Section 1: Compiler Directives ----*/
#include <iostream>
#include <cmath>
using namespace std;

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

int add_1(int input)
{
  // y and z not visible here, but x is
  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;
}
Pre- and Post-Increment Operators

• ++ and -- operators can be used to "increment-by-1" or "decrement-by-1"
  – If ++ comes before a variable it is call pre-increment; if after, it is called post-increment
  – x++; // If x was 2 it will be updated to 3 (x = x + 1)
  – ++x; // Same as above (no difference when not in a larger expression)
  – x--; // If x was 2 it will be updated to 1 (x = x - 1)
  – --x; // Same as above (no difference when not in a larger expression)

• Difference between pre- and post- is only evident when used in a larger expression

• Meaning:
  – Pre: Update (inc./dec.) the variable before using it in the expression
  – Post: Use the old value of the variable in the expression then update (inc./dec.) it

• Examples [suppose we start each example with: int y; int x = 3;]
  – y = x++ + 5; // Post-inc.; Use x=3 in expr. then inc. [y=8, x=4]
  – y = ++x + 5; // Pre-inc.; Inc. x=4 first, then use in expr. [y=9, x=4]
  – y = x-- + 5; // Post-dec.; Use x=3 in expr. then dec. [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
In-Class Exercises

• Checkpoint 2
Not for lecture presentations

BACKUP
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

```c
/* 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[])
{
    // anything inside these brackets is part of the main function
    int y; // a variable declaration stmt
    y = 5+1; // an assignment stmt
    cout << y << endl;
    return 0;
}
```
Software Process

Std C++ & Other Libraries

Compiler

Executable Binary Image ("test")

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

#include <iostream>
using namespace std;

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

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

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

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

Edit & write code

Compile & fix compiler errors

Load & run the executable program
Software Process

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

2. Compile & fix compiler errors
   - $ g++ -g -Wall -o test test.cpp
     or
   - $ make test
   - Fix compile-time errors w/ a debugger

3. Load & run the executable program
   - $ gedit test.cpp &
   - $ g++ -g -Wall -o test test.cpp
   - $ ./test
   - Fix run-time errors w/ a debugger

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

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

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

Compiler

Executable Binary Image (test)

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

Load & Execute

Std C++ & Other Libraries

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