CS 103 Lecture 2 Slides

C/C++ Basics

Mark Redekopp
Announcements

• Ensure you can gain access to Vocareum.com

• Lab 1 review answers must be submitted on our website
  – Attend lab to meet your TAs and mentors and get help with lab 1
A quick high-level view before we dive into the details...

PROGRAM STRUCTURE AND COMPILATION 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

```c
/* 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
2. Compile & fix compiler errors
3. Load & run the executable program

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

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

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

Std C++ & Other Libraries

Compiler

g++

Executable Binary Image ("test")

Load & Execute
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 compiler errors**
3. **Load & run the executable program**

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

```
$ g++ -g -Wall -o test test.cpp
or
$ make test
```

```
$ g++ -g -Wall -o test test.cpp
$ ./test
```

Std C++ & Other Libraries

Compiler

Executable Binary Image ("test")

Load & Execute
MODULE 1:
DATA REPRESENTATION AND TYPES
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
- When programming it is necessary to understand how data is stored
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]
Binary Numbers

• To represent numbers, there is an implicit weight or place value for each 1 or 0
• The weights are the powers of 2
  – $2^0, 2^1, 2^2, 2^3, \ldots$
• The value of the number is the sum of the weights in which there is a 1

$$\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
128 & 64 & 32 & 16 & 8 & 4 & 2 & 1
\end{array} = \_\_\_\_$$

$$\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
128 & 64 & 32 & 16 & 8 & 4 & 2 & 1
\end{array} = \_\_\_\_$$
Combinations

• Because we have a finite number of bits, we can only make a finite set of numbers.

• How many numbers (combinations) can we make with \( n \) bits?
  – ________
  – Use the examples on the right to induce the relationship of how many #s can be formed with \( n \)-bits.
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 (Signed)</th>
<th>C Type (Unsigned)</th>
<th>Bytes</th>
<th>Bits</th>
<th>Signed Range</th>
<th>Unsigned Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>unsigned char</td>
<td>1</td>
<td>8</td>
<td>-128 to +127</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>unsigned short</td>
<td>2</td>
<td>16</td>
<td>-32768 to +32767</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>int</td>
<td>unsigned int</td>
<td>4</td>
<td>32</td>
<td>-2 billion to +2 billion</td>
<td>0 to 4 billion</td>
</tr>
<tr>
<td>long long</td>
<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>

- Prefer **double** over **float**
  - Many compilers will upgrade floats to doubles anyhow

- Don't use floating-point if you don't need to
  - It suffers from rounding error
  - Some additional time overhead to perform arithmetic operations
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/
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)
  - Types provide the context (system)

01000001 = ?

65_{10}  'A'_{ASCII}

Unsigned Binary system

ASCII system
MODULE 2: CONSTANTS, VARIABLES, AND EXPRESSIONS
Constants

- **Integer:** 496, 10005, -234
- **Double:** 12.0, -16., 0.23, -2.5E-1, 4e-2
- **Characters (char type):** enclosed in single quotes
  - Printing characters: 'a', '5', 'B', '!
  - Non-printing special characters use "escape" sequence (i.e. preceded by a \): '
' (newline/enter), '	' (tab), '\\' (slash), '\' (apostrophe)
- **C-Strings**
  - 0 or more characters between double quotes
    "hi1\n", "12345", "b", "\tAns. is %d"
  - Ends with a '\0'=NULL character added as the last byte/character to allow code to delimit the end of the string
- **Boolean (C++ only):** true, false
  - Physical representation: 0 = false, (Non-zero) = 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>

Solutions are provided at the end of the slide packet.
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**

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

- **Single**
- **Many**

What range of values might it use?

- **Positive only**
  - Use an...
  - double
    - 3.0, 3.14159, 6.27e23
  - unsigned int
    - 0, 2147682, ...

- **Possibly negative**
  - Use an...
  - int
    - 0, -2147682, 2147682
  - char
    - 'a', '1', '
  - string
    - "Hi", "2020"

- **bool**
  - Use an...
  - true, false
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></td>
</tr>
<tr>
<td></td>
<td>(Integer vs. Double division)</td>
<td>10 / 3 (=3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10.0 / 3 (=3.3333)</td>
</tr>
<tr>
<td>%</td>
<td>Modulus (remainder)</td>
<td>17 % 5</td>
</tr>
<tr>
<td></td>
<td>[for integers only]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(result will be 2)</td>
</tr>
</tbody>
</table>

17 % 10 = __
4 % 7 = __
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*4-3+5/2;

Tips:
  – Use parenthesis to add clarity
  – Add a space between literals

Operators (grouped by precedence)

<table>
<thead>
<tr>
<th>struct member operator</th>
<th>name.member</th>
</tr>
</thead>
<tbody>
<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>indirect via pointer, address of object</td>
<td>*pointer, &amp;name</td>
</tr>
<tr>
<td>cast expression to type</td>
<td>(type) expr</td>
</tr>
<tr>
<td>size of an object</td>
<td>sizeof</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>

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
Division

• Computers perform division differently based on the type of values used as inputs

• **Integer** Division:
  – When dividing two integral values, the result will also be an integer (any remainder/fraction will be dropped)
    – 10 / 4 = 2  52 / 10 = 5  6 / 7 = 0

• **Floating-point** (Double) & Mixed Division
  – 10.0 / 4.0 = 2.5  52.0 / 10 = 5.2  6 / 7.0 = 0.8571
  – Note: If one input is a double, the other will be promoted temporarily to compute the result as a double
Exercise Review

• Evaluate the following:
  – $25 / 3$
  – $17 + 5 \% 2 - 3$
  – $28 - 5 / 2.0$

C/C++ Variables

• Variables allow us to
  – Store a value until it is needed and change its value potentially many times
  – Associate a descriptive name with a value
• Variables are just memory locations that are reserved to store one 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 as the program runs
• We can provide initial values via '=' or leave them uninitialized

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

int main()
{
    // Sample variable declarations
    char c = 'A';
    int x;  // uninitialized variables
            // will have a (random) garbage value until we initialize it
    x = 1;  // Initialize x's value to 1
    c = 'B'; // Change c's value to 'B'
}
```

A picture of computer memory (aka RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01000001</td>
</tr>
<tr>
<td>1</td>
<td>01001011</td>
</tr>
<tr>
<td>2</td>
<td>10010000</td>
</tr>
<tr>
<td>3</td>
<td>11101000</td>
</tr>
<tr>
<td>4</td>
<td>01101000</td>
</tr>
<tr>
<td>5</td>
<td>11010001</td>
</tr>
<tr>
<td>6</td>
<td>01101000</td>
</tr>
<tr>
<td>7</td>
<td>11010001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

char c = 'A';
A single-byte variable

int x;
A four-byte variable

Variables are actually allocated in RAM when the program is run
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 [0-9, A-Z, a-z, _] (but no punctuation other than underscores)
  
  - Use descriptive names (e.g. `numStudents`, `doneFlag`)
  
  - Avoid cryptic names (e.g. `myvar1`, `a_thing`)

- **location** [the address in memory where it is allocated]

- **Value**

Reminder: You must declare a variable before using it

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

```c
double a = (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 = Left Hand-Side, RHS = Right Hand Side

- **Should be read:** Place the value of \textit{expression} into memory location of \textit{variable}

  - \( z = x + y - (2\times z); \)
  - 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

  - \( x + 5; \) // 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 assignment operators for updating a variable based on its current value:** +=, -=, *=, /=, &=, ...

  - \( x += 5; \) \( \text{(x = x+5)} \)
  - \( y *= x; \) \( \text{(y = y*x)} \)
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 is promoted to double)

• Could use C or C++ syntax for **explicit** casting
  – 5 + (double) 3 / (double) 2 (C-Style cast)
  – 5 + static_cast<double>(3) / static_cast<double>(2) (C++-Style)
  – 5 + static_cast<double>(3) / 2 (cast 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; x + y/z;
  – x + static_cast<double>(y) / z
cout and cin

MODULE 3: C++ I/O (INPUT/OUTPUT)
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."
;
}
```

Input stream:

```
7 5 y ...
```

Output stream:

```
It was the
```

Input stream:

```
y ...
```

Output stream:

```
4
```
C++ Output

• Include `<iostream>` (not `iostream.h`)
• Add using namespace std; at top of file
• Use an appropriate `cout` statement
• 'cout' requires appropriate use of separators `<<` between consecutive values or different types of values
• 'cout' does not add spaces between consecutive values; you must do so explicitly
  – Since text strings are a different value we must separate it with the `<<` operator
• Generally good practice to give some descriptive text with your numeric output
  – Note: You may divide up output over multiple 'cout' statements. Unless an 'endl' or '\n' is used the next 'cout' statement will resume where the last one left off

```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;
    cout << "c = " << c << "\n 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

\[
\begin{align*}
\text{c} &= \ 0 \quad \text{y} &= \ 0.0 \\
\text{• If the user types in} & \\
\quad a \ t \ 3.5 \ n \\
\text{• After the first '>>'} & \\
\quad c &= \ 'a' \quad \text{y} &= \ 0.0 \\
\quad \ \ t \ 3.5 \ n \\
\text{• After the second '>>'} & \\
\quad c &= \ 'a' \quad y &= \ 3.5 \\
\quad \ n
\end{align*}
\]

#include<iostream>
using namespace std;

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

\text{\textbf{cin will:}}
\begin{itemize}
\item skip leading whitespace
\item stop at trailing whitespace
\end{itemize}
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

char c = '1';

char c >= 'a' && c <= 'z'; // && means AND
                        // here we are checking if c is
                        // storing a lower case letter
```

### ASCII printable characters

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>51</td>
<td>3</td>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>58</td>
<td>:</td>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
<td>95</td>
<td>_</td>
</tr>
</tbody>
</table>
In-Class Exercises

• Checkpoint 1
MODULE 4: ODDS & ENDS
(LIBRARY FUNCTIONS, ASSIGNMENT, AND CASTING)
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;
}
```
Problem Solving Idioms

- An idiom is a colloquial or common mode of expression
  - Example: "raining cats and dogs"
- Programming has common modes of expression that are used quite often to solve problems algorithmically
- We have developed a repository of these common programming idioms. We STRONGLY suggest you
  - Reference them when attempting to solve programming problems
  - Familiarize yourself with them and their structure until you feel comfortable identifying them

---

**Rule / Exception Idiom**

- **Name**: Rule/Exception
- **Description**: Perform a default action and then use an `if` to correct for an exceptional case
- **Structure**: Code for some default action (i.e. the rule) is followed by exceptional case

```c
// Default action
if( /* Exceptional Case */ )
{
    // Code for exceptional case
}
```

- **Example(s)**:
  - Base pay plus bonus for certain exceptional employees

```c
bool earnedBonus = /* set somehow */;
int bonus = /* set somehow */;

int basePay = 100;
if( earnedBonus == true )
{
    basePay += bonus;
}
```

- **Notes**: This can be implemented with an `if/else` where an `else` implements the other.
Assignment Idioms: Shifting and Rotation

- The **shifting idiom** shifts data among variables usually replacing/dropping some elements to make room for new ones
  - The key pattern is some elements get dropped/overwritten and other elements are reassigned/moved
  - It is important to start by assigning the variable to be replaced/dropped and then move in order to variables receiving newer data
  - Examples: Top k items (high score list)

- The **rotation idiom** reorders or rearranges data among variables without replacing/dropping elements
  - The key pattern is all elements are kept but just reordered
  - It is usually necessary to declare and maintain some temporary variable to avoid elements getting dropped/overwritten
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

```cpp
/*----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;
}
```
Math & Other Library Functions

• C++ predefines a variety of functions for you. Here are a few of them:
  – \( \text{sqrt}(x) \): returns the square root of \( x \) (in <cmath>)
  – \( \text{pow}(x, y) \): returns \( x^y \), or \( x \) to the power \( y \) (in <cmath>)
  – \( \sin(x)/\cos(x)/\tan(s) \): returns the trig. Function's value for \( x \) if \( x \) is in radians (in <cmath>)
  – \( \text{abs}(x) \): returns the absolute value of \( x \) (in <cstdlib>)
  – \( \text{max}(x, y) \) and \( \text{min}(x,y) \): returns the maximum/minimum of \( x \) and \( y \) (in <algorithm>)

• You call these by writing them similarly to how you would use a function in mathematics [using parentheses for the inputs (aka) arguments]

• Result is replaced into bigger expression

• Must #include the correct library
  – #includes tell the compiler about the various pre-defined functions that your program may choose to call

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

int main()
{
    // 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;
}
```

http://www.cplusplus.com/reference/cmath/
Statements

• C/C++ programs are composed of statements
• Most common kinds of statements end with a semicolon
• Declarations (e.g. `int x=3;`)
• Assignment + Expression (suppose `int x=3; int y;`)
  – `x = x * 5 / 9;` // compute the expression & place result in x
  // `x = (3*5)/9 = 15/9 = 1`
• Assignment + Function Call (+ Expression)
  – `x = cos(0.0) + 1.5;`
  – `sin(3.14);` // Must save or print out the result (x = sin(3.14), etc.)
• `cin, cout` statements
  – `cout << cos(0.0) + 1.5 << " is the answer." << endl;`
• Return statement (immediately ends a function)
  – `return value;`
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]
In-Class Exercises

• Checkpoint 2
SOLUTIONS
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>C-string</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 (char = some number that fits in 8-bits/1-byte</td>
</tr>
<tr>
<td>&quot;5.0&quot;</td>
<td>double</td>
<td>C-string</td>
</tr>
<tr>
<td>'5'</td>
<td>int</td>
<td>char</td>
</tr>
</tbody>
</table>