1. Lecture 2 – C++ Primer - (continued)
   1. Operators and Expressions
      1. Standard Arithmetic Operators
         1. Left-to-Right Associativity, Standard rules of arithmetic Precedence ⮚ Parentheses
            1. ⮚  Multiplication ( \* ), Division ( / ), Modulo ( % ) - Precedence Group 5
            2. ⮚  Addition ( + ), Subtraction ( - ) - Precedence Group 6
            3. ⮚  Exponents ... (Note: Do not use ( ^ ) for exponents.)
      2. Standard Relational Operators
         1. Testing for:
            1. ⮚  Equality ( **==** ) , Inequality ( **!=** )
            2. ⮚  Less-Than ( **<** ) , Higher-Than ( **>** )
            3. ⮚  Less/Equal-To ( **<=** ) , Higher/Equal-To ( **>=** )
            4. ⮚  Evaluate to (**true**) or (**false**)
      3. Standard Logical Operators
         1. Evaluating:
            1. ⮚  Logical AND (**&&**) , OR (**||**) , NOT(**!**)
            2. ⮚  Evaluate to (**true**) or (**false**)
      4. Standard Bitwise Operators
         1. Useful to conduct Bitwise operations: (Boolean , bit-by-bit operations on Registers)
            1. ⮚  AND ( **&** ) , OR ( **|** ) , XOR ( **^** ) , NOT( **~**
            2. ⮚  Bitwise Shifting Left ( **<<** ) , Right ( **>>** )
      5. Operators (General)
         1. A variety of operators in programming languages: ⮚ Unary (1), Binary (2), Ternary (3) (depends on number of operands, i.e. things they operate on) Represented by special symbolic characters
            1. ⮚ (**+**) means **add ( • , • )**, hence it is a Binary operator.
      6. Unary Operators
         1. ⮚ Logical Negation ( ! ) ( **! true** ) is **false** ( **! false** ) is **true**
         2. ⮚ Post-Increment ( • **++** ) and Post-Decrement ( • **--** ) (**x ++**) evaluates to (**x**) ,**x** is increased by **1**(**x --**) evaluates to (**x**),**x** is decreased by **1**
         3. ⮚ Pre-Increment ( **++** • ) and Pre-Decrement ( **--** • ) (**++ x**) evaluates to (**x + 1**) ,**x** is increased by**1** (**-- x**) evaluates to (**x – 1**) ,**x.** is decreased by**1**
      7. Expressions
         1. When simple units of *operands and operators are combined* into larger units, (always following the strict rules of precedence and associativity).  
            1. ⮚ Expression is each **aggregate computable unit** (simpler or larger).
      8. Conditional Ternary Operator
         1. Composed of Expressions:
            1. **(Test\_Expression) ? (Evaluated\_Expression\_If\_TRUE) : (Evaluated\_Expression\_If\_FALSE)   
               5==7 ? printf("5 equals 7") : printf("5 does not equal 7");   
               int a = 10;  
               int b = (5==7) ? 1\*a : -1\*a ;**
      9. Operator Associativity
         1. Kicks in when operators of the same precedence appear in an Expression.
         2. Postfix operators: **++ --** (left to right)
         3. Prefix operators: **++ --** (right to left)
         4. Unary operators: **+ - !** (right to left)
         5. **\* / %** (left to right)   
            **+ -** (left to right)   
            **< > <= >=   
            == !=  
            &&**  
            | | **? :**Assignment operator: **=** (right to left)
      10. Type Casting with ( ) ~ or ( ~ )
          1. Can add “**.0**” to literals to force precision:
          2. **convertedVar = (new\_type)originalVar;   
             convertedVar = new\_type(originalVar);**
          3. **double x = (double) intVar1 / intVar2;   
             double x = double( intVar1 / intVar2 );** 
             1. Casting to force double-precision division among two integer variables! DOES IT?
      11. Alternative C++ expression:
          1. **double x = static\_cast<double>( X );**
      12. Type Conversion
          1. ⮚ Implicit type conversion  
             Done by the compiler:  
             **17 / 5.5;**“Implicit type cast” **17** → **17.0**
          2. ⮚ Explicit type conversion   
             Programmer-enforced:   
             **(double)17 / 5.5;  
             double(17) / 5.5; static\_cast<double>( 17 ) / 5.5;**
      13. Shorthand Operators
          1. Count += 2; --- count = count +2
          2. Total -= discount; --- total = total- discount;
          3. Bonus \*= 2; --- bonus = bonus \*2;
          4. Time /= rushfactor; --- time = time/rushfactor
          5. Post increment/decrement i++
          6. Pre-increment/decrement ++i
   2. Statements
      1. A complete unit of execution (equivalent to a sentence in a language).
         1. ⮚  Expression statements
            1. Assignment expressions Use of (**++**) or (**--**) Method invocations Object creation
         2. ⮚  Flow Control statements
            1. Selection structures Repetition/Iteration structures
      2. Follow scope rules and end with semicolon ( ; )
      3. Flow Control Statements
         1. If, then, else
         2. Switch
            1. ⮚ The switching value must evaluate to an integer or enumerated type  
               ⮚ The case values must be either:

a) a constant or literal, or

b) an **enum** value

* + - * 1. ⮚ The case values must be of the same type as the switch expression
        2. Notes:

**break** statements are typically used to terminate each **case***.*

**I**t is usually a good practice to include a **default** case.

* + - 1. While
         1. Executes a block of statements while a particular condition/expression is true
      2. Do While
         1. Performs at least one block execution
      3. For
         1. Iterate over a range of values.

⮚ The *initialization* expression initializes the loop it is executed once, as the loop begins. ⮚ Loop ends when the *termination* expression evaluates to **false**.

⮚ The *increment* expression is invoked after each iteration.

* 1. Input/output
     1. Console input/output
        1. ⮚  Console Input, Output, and Error stream objects in C++ are called: **cin, cout, cerr**
        2. ⮚  They are Global Objects of the classes  
           **ostream** (outputstream) and **istream** (inputstream)
        3. ⮚  Defined in the C++ library header called **<iostream>** (we’ll leave it at that for now)
        4. Useful for:
           1. ⮚ User input
           2. ⮚ User output
           3. ⮚ Error messages (exclusive stream, redirection if required)
        5. ⮚ Console Input, Output, and Error stream objects in C++ are called: **cin, cout, cerr**
        6. ⮚ They are Global Objects of the classes  
           **ostream** (outputstream) and **istream** (inputstream)
        7. ⮚ Defined in the C++ library called **<iostream>**
        8. *Note*:
           1. **std::cout** and **std::cin** are Global Objects of the classes **std::ostream** and **std::istream**
           2. ⮚ **#include <iostream>** is responsible for including their corresponding declarations in your programs.
     2. Console Output (std: :cout)
        1. Any standard C++ data can be output:
           1. ⮚ Variables
           2. ⮚ Constants
           3. ⮚ Literals
           4. ⮚ Expressions (which can include all of above)
        2. **cout << numberOfGames << " games played.";** 
           1. 2 values are output:

Value of variable **numberOfGames**

Literal string **" games played."**

* + 1. Output (std: :cout)
       1. New lines in output
          1. ⮚ Escape sequences are valid: **"\n"** is “newline”
       2. A second method:
          1. ⮚ Object **std::end1**
          2. ⮚ Flushes output buffer ( **std : : flush** )
       3. Examples
          1. Cout << “Hello World\n”;
          2. Cout << “Hello World” << end1;
    2. Output Format
       1. Numeric values may not display as you’d expect:
          1. **cout << "The price is $" << price << endl;**
       2. If **double price = 78.5;** we might get:
       3. ⮚ Force Decimals:
          1. **cout.setf(ios::fixed);**

**Fixed Percision**

* + - * 1. **cout.setf(ios::showpoint);**

**Show Decimal Point**

* + - * 1. **cout.precision(2);**

**Set Percision Decimals**

* + 1. Console Input (std : : cin)
       1. No literals allowed for **cin** 
          1. ⮚ Must input to a variable
       2. Waits on-screen for keyboard entry
          1. ⮚ **cin >> num;**
       3. Value entered at keyboard is ‘assigned’ to **num**.
       4. ⮚ Consumes any leading whitespaces and stops reading at next whitespace.
       5. ⮚ Can also be cascaded, **>>** operators separate each “type” of thing we read in.
    2. User input/output
       1. Prompt user for input
          1. Cout << “Enter number of objects: “;  
             cin >> numOfObjects
       2. User-friendly input/output design:
          1. Every cin should have a corresponding prior cout prompt.
    3. Error Output (std : : cerr)
       1. **cerr** works same as **cout** 
          1. ⮚ Mechanism for distinguishing between regular output and error output
          2. ⮚ Most systems allow **cout** and **cerr** to be “redirected” to other devices e.g., line printer, output file, error console, etc.
    4. File Input / Output
       1. Similarly to cin, a combination of:
          1. Cin >> num;
       2. At the top:
          1. #include <fstream>  
             using namespace std;
       3. An input stream object (creation just as with any other variable):
          1. Ifstream inputStream;
       4. “Connect” the inputStream variable to a text file (via pathname):
          1. inputStream.open(“filename.txt”);
       5. Read-in by using thee Extraction Operator (>>):
          1. inputStream >> var;
       6. The result is the same as using cin >> var except the input is coming from the text file and not the keyboard
       7. Check that EEOF hasn’t been reached:
          1. If (!inputStream.eof())
       8. Close with:
          1. inputStream.close();
  1. Namespaces – Resolution
     1. Namespaces
        1. A collection of name definitions under a **top-level identifier**. Most common is **namespace std** 
           1. ⮚ Contains *all* standard library definitions !
        2. The **using** keyword: Instruct the compiler to attempt to resolve names therein
  2. Scope
     1. You can define new variables in many places in your code. So where is it in effect / What is its Variable Scope?
        1. ⮚ The set of statements in which the variable is known to the compiler.
     2. Where a variable can be referenced from in your program
        1. ⮚ Limited by the code **Block** in which the variable is defined
     3. ⮚ Behaves “as-if” it’s placed together with **#include** statements, even though it’s trying to import names into the Local Scope only.

1. Arrays
   1. A collection of related data items.
      1. ⮚  Can be of any data type.
      2. ⮚  They are static   
         Their size does not change.
   2. They are declared contiguously in memory. In other words, an array’s data is stored in one big block, together.
      1. Recall simple variables:
         1. ⮚ Allocated memory in an "address"
      2. Array declarations allocate memory for entire array
         1. ⮚ Sequential allocation
            1. Addresses allocated "back-to-back“.
            2. Allows indexing calculations.
            3. Simple "addition" from array beginning (index 0)
   3. Array Declaration
   4. Array Limitations
      1. ⮚ Does not know how large it is – there is no C++ **size()** function for arrays.
      2. ⮚ No bounds checking is performed.
      3. Arrays are static
         1. ⮚ Size must be known at compile time (cannot change once set).
         2. *Normally* can’t do user input for array size: “How many numbers would you like to store?”
      4. C / C++ Benefits:
         1. ⮚ Efficiency.
         2. ⮚ Backwards Compatibility.
   5. Array Declaration / Initialization
      1. ⮚A *declaration* alone generally will not initialize the data stored in the memory locations.
      2. ⮚ They will contain “garbage” leftover data.
      3. ⮚ *Initialization* ensures specific values for the contained data.
      4. Auto – initialization (fewer values than the given size) : ⮚ Fills values starting at the beginning.
         1. ⮚ Remainder is filled with that data type’s “zero”.
      5. If no array size is given array is created only as big as is needed:
   6. C-strings (as char arrays)
      1. ⮚They are **char** type arrays.
      2. ⮚ Initialization (normal way):  
         **char name[5] = {'J', 'o', 'h', 'n', 0 };**
      3. ⮚ Initialization (string constant literal): **char name[5] = {‘J’, ‘O’, ‘H’, ‘N’, ‘0’};**
      4. Note: Different quotes have different purposes !!! ⮚ Double quotes are for strings
         1. ⮚ Single quotes are for chars (characters)
   7. Array Element Access
   8. Arrays (as Arguments in Functions)
   9. Multi-Dimensional Arrays