CMSC202
Computer Science II for Majors

Lecture 02 –
C++ Primer (Continued)

Dr. Katherine Gibson

Last Class We Covered

- Syllabus
- Course Expectations and Objectives

- Differences between Python and C++
  - Interpreted vs compiled
  - Explicitly stating type
  - Semicolons
  - Curly braces
    - C++ is space insensitive!

Any Questions from Last Time?

Announcements

- The course policy agreement is due back in class by Tuesday, February 8th
  - Worth 1% of your grade
  - (Final is now worth 19%)

- The Blackboard site is now available
  - It will be updated with a course schedule; we will not be following Professor Marron's schedule
  - His page still has all of the information on assignments and course policies

Today’s Objectives

- To begin covering the very basics of C++
  - Operators
  - Input and Output
  - Formatting Output
  - Strings
  - If, Else, If-Else
  - Loops
  - Other Control Structures

What We’re Learning

- 202’s goal is not to teach you C++
- Want you to instead
  - Become better problem solvers
  - Learn more advanced techniques
  - Become more confident in your skill
- C++ is merely the tool we use
  - (Which means you do need to learn it as well)

Review: Literal Data

- Literals
  - Examples:

```
2                  // Literal constant int
5.75               // Literal constant double
'Z'                // Literal constant char
"Hello World\n"   // Literal constant string
```

- Cannot change values during execution
- Called "literals" because you "literally typed" them in your program!

Constants

- You should not use literal constants directly in your code
  - It might seem obvious to you, but not so:
    - `limit = 52`
    - Is this weeks per year… or cards in a deck?
- Instead, you should use named constants
  - Represent the constant with a meaningful name
  - Also allows you to change multiple instances in a central place

Constants

- There are two ways to do this:
  - Old way: preprocessor definition:

```
#define WEEKS_PER_YEAR 52
```

(Note: there is no “**=**”)

  - New way: constant variable:
    - Just add the keyword “**const**” to the declaration

```
const float PI = 3.14159;
```

Arithmetic Operators

- Standard Arithmetic Operators
- Precedence rules – standard rules
  - Parentheses
  - Exponents
  - Multiplication and...
  - Division
  - Addition and...
  - Subtraction
- Note: do <u>not</u> use “^” for exponents

Operators, Expressions

- Most programming languages have a variety of *operators*
  - Called unary, binary, and even ternary
  - Depends on the number of operands (things they operate on)
- Usually represented by special symbolic characters: e.g., ‘+’ for addition, ‘*’ for multiplication

Operators, Expressions

- There are also relational operators, and Boolean operators
- Simple units of operands and operators combine into larger units, according to strict rules of *precedence* and *associativity*
- Each computable unit (both simple and larger aggregates) is called an *expression*

Binary Operators

- What is a binary operator?
  - An operator that has two operands

    <operand> <operator> <operand>

  - Arithmetic Operators

    `+        -        *        /        %`

  - Relational Operators

    `<        >        ==        <=        >=`

  - Logical Operators

    `&&        ||`

Relational Operators

- In C++, all relational operators evaluate to a boolean value of either **true** or **false** .

```
x = 5;
y = 6;
```

  x > y will always evaluate to **false**

- C++ has a ternary operator – the general form is:

  (conditional expression) ? true case : false case ;

- Ternary example:

```
cout << (( x > y ) ? "X is greater" : "Y is greater");
```

Unary Operators

- Unary operators only have one operand.

  ! **++** **--**

  ! is logical negation, !true is false, !false is true

  ++ and **--** are the **increment** and **decrement** operators
  x++ **a post-increment** (postfix) operation
  ++x **a pre-increment** (prefix) operation

- **++** and **--** are “shorthand” operators
- More on these later…

Precedence, Associativity

- Order of operations application to operands:
  - Postfix operators: ++ -- (left to right)
  - Prefix operators: ++ -- (right to left)
  - Unary operators: + - ++ -- ! (right to left)
  - * / % (left to right)
  - + - (left to right)
  - < > <= >=
  - == !=
  - &&
  - ||
  - ? :
  - Assignment operator: = (right to left)

Associativity

- What is the value of the expression?

    3 * 6 / 9

    (3 * 6) / 9

    18 / 9

    2

- What about this one?

```
int x, y, z;
x = y = z = 0;
```

Arithmetic Precision

- Precision of Calculations
  - VERY important consideration!
    - Expressions in C++ might not evaluate as you’d “expect”!
  - "Highest-order operand" determines type of arithmetic "precision" performed
  - Common pitfall!

Arithmetic Precision Examples

- Examples:
  - 17 / 5 evaluates to 3 in C++!
    - Both operands are integers
    - Integer division is performed!
  - 17.0 / 5 equals 3.4 in C++!
    - Highest-order operand is "double type"
    - Double "precision" division is performed!
  - `int intVar1 = 1, intVar2 = 2;`
    `intVar1 / intVar2;`
    - Performs integer division!
    - Result: 0!

Individual Arithmetic Precision

- Calculations done "one-by-one"

  1 / 2 / 3.0 / 4 performs 3 separate divisions.

  - First→ 1 / 2 equals 0
  - Then→ 0 / 3.0 equals 0.0
  - Then→ 0.0 / 4 equals 0.0!
- So not necessarily sufficient to change just "one operand" in a large expression
  - Must keep in mind all individual calculations that will be performed during evaluation!

Type Casting

- Two types
  - Implicit—also called "Automatic"
    - Done FOR you, automatically
      `17 / 5.5`
    - This expression causes an "implicit type cast" to take place, casting the **`17`** → **`17.0`**
  - Explicit type conversion
    - Programmer specifies conversion with cast operator
      `static_cast<double>17 / 5.5`
    - Same expression as above, using explicit cast
      `static_cast<double>myInt / myDouble`
    - More typical use; cast operator on variable

Shorthand Operators

- Increment & Decrement Operators
  - Just short-hand notation
  - Increment operator, **`++`**

    `intVar++;` is equivalent to

    `intVar = intVar + 1;`
  - Decrement operator, **`--`**

    `intVar--;` is equivalent to

    `intVar = intVar – 1;`

Shorthand Operators: Two Options

- Post-Increment
  `intVar++`
  - Uses current value of variable, THEN increments it
- Pre-Increment
  `++intVar`
  - Increments variable first, THEN uses new value
- "Use" is defined as whatever "context" variable is currently in
  - No difference if "alone" in statement:
    intVar++; and ++intVar; → identical result

Post-Increment in Action

- Post-Increment in Expressions:

```
int n = 2, valueProduced;
valueProduced = 2 * (n++);
cout << valueProduced << endl;
cout << n << endl;
```

  - What output does this code segment produce?

```
4
3
```

  - Since post-increment was used

Pre-Increment in Action

- Now Using Pre-Increment:

```
int n = 2, valueProduced;
valueProduced = 2 * (++n);
cout << valueProduced << endl;
cout << n << endl;
```

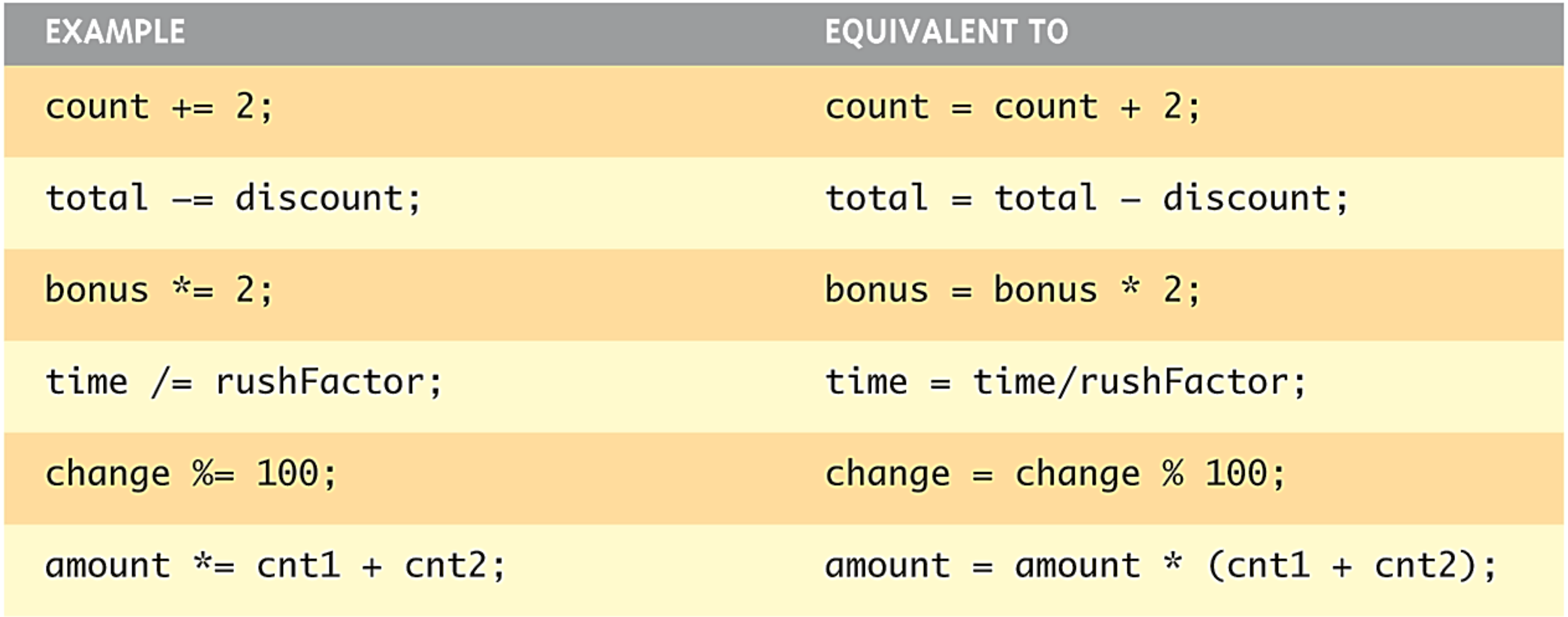
  - What output does this code segment produce?

```
6
3
```

  - Since pre-increment was used

Assigning Data: Shorthand Notations

- You can use shorthand for many operations



EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

Input and Output

Console Input/Output

- Your input and output objects in C++ are called **`cin`**, **`cout`**, **`cerr`**
- Defined in the C++ library called **`<iostream>`**

- Allow us to:
  - Get input from the user
  - Send output to the user
  - Print error messages to the user

Using namespace std

- At top of each file you must have

```
using namespace std;
```

- Otherwise you must use

`std::cin`		`cin`
`std::cout`	instead of	`cout`
`std::endl`		`endl`

- Remember, you also need to have the library

```
#include <iostream>
```

Console Output

- What can be outputted?
  - Any data can be outputted to display screen
    - Variables
    - Constants
    - Literals
    - Expressions (which can include all of above)
  - `cout << numberOfGames << " games played.";`
  - 2 values are outputted:
    - "value" of variable numberOfGames,
    - literal string " games played."

Separating Lines of Output

- New lines in output
  - Recall: "\n" is escape sequence for the char "newline"
- A second option: **`endl`**
- Examples:

```
cout << "Hello World\n";
```

  - Sends string "Hello World" to display, & escape sequence "\n", skipping to next line

```
cout << "Hello World" << endl;
```

  - Same result as above

The << Operator

- Insertion operator; used along with **cout**

- Separates each “type” of thing we print out

```
int x = 3;
cout << "X is: " << x
     << "; squared "
     << x * x << endl;
```

The >> Operator

- Extraction operator; used with **`cin`**
- Skips any leading whitespace, and stops reading at next whitespace

```
cin >> firstName >> lastName >> age;
```

- Separates each “type” of thing we read in

Input Using cin

- No literals allowed for **`cin`**
  - Must input to a variable

- Waits on-screen for keyboard entry
  - **`cin >> num;`**
  - Value entered at keyboard is "assigned" to num

Prompting for Input

- Always "prompt" user for input

```
cout << "Enter number of dragons: ";
cin >> numOfDragons;
```

- Note no "**\n**" in **cout**. Prompt "waits" on same line for keyboard input

- Every cin should have a cout prompt
  - Maximizes user-friendly input/output

Error Output

- Output with cerr
  - cerr works almost the same as cout
  - Provides mechanism for distinguishing between regular output and error output
- Re-direct output streams
  - Most systems allow cout and cerr to be "redirected" to other devices
    - e.g., line printer, output file, error console, etc.

Formatting Output

- Formatting numeric values for output
  - Values may not display as expected

    ```
    cout << "The price is $" << price << endl;
    ```

    - If price (declared a double) has the value 78.5, you might get
      - `The price is $78.5000000`
      - `The price is $78.5`
    - Neither is what you want
  - Have to tell C++ how to output numbers.

Formatting Numbers

- "Magic Formula" to force decimal sizes:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

- These statements force all future **cout**'ed values to have exactly two digits after the decimal place:
  - Example:
    cout << "The price is $" << price << endl;
    - Now results in the following:
      The price is $78.50
- Can modify precision whenever you want in the code

Formatting Integers

- Field width and fill characters
  - Must **`#include <iomanip>`**
  - **`setw(n)`** sets field width to n
  - **`cout.fill(c)`** sets “fill” character to c
- Example:

  -
    ```
    int x = 7;
    cout.fill('0');  //set fill character to 0
    cout << setw(3) << x << endl;
    ```
  - Outputs `007` (left pads with zeros)

C-Strings and the String class

C-strings

- C++ has two kinds of “strings of characters”:
  - the original C-string: array of characters
  - The object-oriented *string* class

- C-strings are terminated with a null character (‘\0’)

  ```
  char myString[80];
  ```

  declares a variable with enough space for a string with 79 usable characters, plus the null char

C-strings

- You can initialize a C-string variable:

```
char myString[80] = “Hello world”;
```

This will set the first 11 characters as given, make the 12th character ‘\0’, and the rest unused for now.

- What would these look like?

```
char str1 [5] = “dog”;
char str2 [5] = “cat”;
char str3 [5];
```

Arrays of Characters

`char str1 [5] = “dog”;`					
element	0	1	2	3	
char	‘d’	‘o’	‘g’	‘\0’	‘x’
`char str2 [5] = “cat”;`					
element	0	1	2	3	
char	‘c’	‘a’	‘t’	‘\0’	‘f’
`char str3 [5];`					
element	0	1	2	3	4
char	‘.’	‘N’	‘=’	‘¿’	‘8’

- str3 was only declared, not initialized, so it’s filled with garbage and has no null terminator

Terrible C-style string Joke

Two strings walk into a bar.

The bartender says, "What'll it be?"

The first string says, "I'll have a gin and tonic#MV*()>SDk+!^&@P&]JEA&#65535".

The second string says, "You'll have to excuse my friend, he's not null-terminated."

String type

- C++ added a data type of “string”
  - Not a primitive data type; distinction will be made later
  - Need to `#include <string>` at the top of the program
  - The “**+**” operator on strings *concatenates* two strings together
  - `cin >> str` where `str` is a string only reads up to the first *whitespace* character

String Equality

- In Python, you can use the simple “==“ operator to compare two strings:

```
 if name == “Fred”:
```

- In C++, you can use “==” to compare two ***string*** class items, ***but not C-strings!***
- To compare two C-strings, you have to use the function **`strcmp();`**
  - It is not syntactically incorrect to compare two C-strings with “==“, but it doesn’t do what you expect

Programming Style

Programming Style

- Bottom-line: Make programs easy to read and modify
- Comments, two methods:
  - // Two slashes indicate entire line is to be ignored
  - /*Delimiters indicates everything between is ignored*/
  - Both methods commonly used
- Identifier naming
  - ALL_CAPS for constants
  - lowerToUpper for variables
  - Most important: MEANINGFUL NAMES!

Libraries

- C++ Standard Libraries
- **`#include <library_name>`**
  - Directive to "add" contents of library file to your program
  - Called "preprocessor directive"
    - Executes before compiler, and simply "copies" library file into your program file
- C++ has many libraries
  - Input/output, math, strings, etc.

Summary Part 1

- C++ is case-sensitive
- Use meaningful names
  - For variables and constants
- Variables must be declared before use
  - Should also be initialized
- Use care in numeric manipulation
  - Precision, parentheses, order of operations
- **`#include`** C++ libraries as needed

Summary Part 2

- Object **`cout`**
  - Used for console output
- Object **`cin`**
  - Used for console input
- Object **`cerr`**
  - Used for error messages
- Use comments to aid understanding of your program
  - Do not over-comment

Compilation

Using the C Compiler at UMBC

- Invoking the compiler is system dependent.
  - At UMBC, we have two C compilers available, **cc** and **gcc**.
  - For this class, we will use the gcc compiler as it is the compiler available on the Linux system.

Invoking the Compiler

- At the prompt, type

```
g++ -Wall program.cpp -o program.out
```

- where **program.cpp** is the C++ program source file

- **-Wall** is an option to turn on all compiler **warnings** (really good idea!)

The Executable File

- If there are no errors in program.cpp, this command produces an **executable file**, which is one that can be executed (run).
  - If you do not use the “-o” option, the compiler names the executable file **a.out**
- To execute the program, at the prompt, type

```
./program.out
```

- Although we call this process “compiling a program,” what actually happens is more complicated.

The “make” System

- We will be using the “make” system to automate what was shown in the previous few slides
- This will be discussed in more detail in lab

Expressions, Statements, and If

Expressions

- An ***expression*** is a construct made up of variables, operators, and method invocations, that evaluates to a single value.
- For example:

```
int cadence = 0;

anArray[0] = 100;

cout << "Element 1 at index 0: " << anArray[0]);

int result = 1 + 2;

cout << (x == y ? "equal" :"not equal");
```

Statements

- ***Statements*** are roughly equivalent to sentences in a language. A ***statement*** forms a complete unit of execution.
- Two types of statements:
  - Expression statements – end with a semicolon ';'
    - Assignment expressions
    - Any use of ++ or --
    - Method invocations
    - Object creation expressions
  - Control Flow statements
    - Selection & repetition structures

If-Then Statement

- The *if-then* statement is the most basic of all the control flow statements.

Python

```
if x == 2:
    print "x is 2"
print "Finished"
```

C++

```
if (x == 2)
   cout << "x is 2";
cout << "Finished";
```

A brief digression…

Notes about C++’s *if-then*:

- Conditional expression must be in parentheses
- Conditional expression has various interpretations of “truthiness” depending on type of expression

- If-then raises questions about
  - Multi-statement blocks
  - Scope
  - Truth in C++

Multiple Statements

- What if our *then* case contains multiple statements?

Python

```
if x == 2:
    print "even"
    print "prime"
print "Done!"
```

C++ *(but incorrect!!)*

```
if(x == 2)
    cout << "even";
    cout << "prime";
cout << "Done!";
```

Unlike Python, spacing plays no role in C++'s selection/repetition structures

- The C++ code is ***syntactically*** fine – no compiler errors
- However, it is ***logically*** incorrect

Blocks

- A ***block*** is a group of zero or more statements that are grouped together by delimiters.
- In C++, blocks are denoted by opening and closing curly braces ‘{’ and ‘}’

```
if(x == 2) {
   cout << "even";
   cout << "prime";
}
cout << "Done!";
```

Note:

- It is generally considered a good practice to include the curly braces even for single line statements. Why?

“Truthiness”**

- What is “true” in C++?
- Like some other languages, C++ has a true Boolean primitive type (*bool*), which can hold the constant values *true* and *false*
- Assigning a Boolean value to an *int* variable will assign 0 for *false,* 1 for *true*

** kudos to Stephen Colbert

“Truthiness”

- For compatibility with C, C++ is very liberal about what it allows in places where Boolean values are called for:
  - *bool* constants, variables, and expressions have the obvious interpretation
  - Any integer-valued type is also allowed
    - 0 is interpreted as “false”, all other values as “true”
    - So, even -1 is considered true!

Gotcha! = versus ==

```
int a = 0;

if (a = 1) {
    cout << "a is one\n" ;
}
```

What happens here?

How do we fix it?

If-Then-Else Statement

- The *if-then-else* statement looks much like it does in Python (aside from the parentheses and curly braces)

Python

```
if x % 2 == 1:
    print "odd"
else:
    print "even"
```

C++

```
if(x % 2 == 1) {
    cout << "odd";
} else {
    cout << "even";
}
```

If-Else If-Else Statement

- Again, very similar…

Python

```
if x < y:
    print "x < y"
elif x > y:
    print "x > y"
else:
    print "x == y"
```

C++

```
if (x < y) {
    cout << "x < y";
} else if (x > y) {
   cout << "x > y";
} else {
   cout << "x == y";
}
```

Other Control Structures

Switch Statement

- Unlike *if-then* and *if-then-else*, the *switch* statement allows for any number of possible execution paths.
- Works with any <u>integer</u>-based (e.g., *char, int, long*) or enumerated type (covered later)

Switch Statement

```
int cardValue = /* get value from somewhere */;
switch(cardValue) {
    case 1:
        cout << "Ace";
        break;
    case 11:
        cout << "Jack";
        break;
    case 12:
        cout << "Queen";
        break;
    case 13:
        cout << "King";
        break;
    default:
        cout << cardValue;
        break;
}
```

Notes:

- *break* statements are typically used to terminate each *case.*
- It is usually a good practice to include a *default* case.

Switch Statement

```
switch (month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        cout << "31 days";
        break;
    case 4: case 6: case 9: case 11:
        cout << "30 days";
        break;
    case 2:
        cout << "28 or 29 days";
        break;
    default:
        cout << "Invalid month!";
        break;
}
```

Note:

- Without a break statement, cases “fall through” to the next statement.

Switch Statement

- The switching value <u>must</u> evaluate to an integer or enumerated type
- The *case* values must be constant or literal, or enum value
- The case values must be of the same type as the switch expression

While Loops

- The *while* loop executes a block of statements while a particular condition is *true.*

- Pretty much the same as Python…

Python

```
count = 0;
while(count < 10):
    print count
    count += 1
print "Done!"
```

C++

```
int count = 0;
while(count < 10) {
    cout << count;
    count++;
}
cout << "Done!";
```

For Loop

- The **`for`** statement provides a compact way to iterate over a range of values.

```
for (initialization; termination; increment)
{
    /* ... statement(s) ... */
}
```

- The ***initialization expression*** initializes the loop – it is executed once, as the loop begins.
- When the ***termination expression*** evaluates to false, the loop terminates.
- The ***increment expression*** is invoked after each iteration through the loop.

For Loop

- The equivalent loop written as a *for* loop
  - – Counting from start value (zero) up to (excluding) some number (10)

Python

```
for count in range(0, 10):
    print count
print "Done!"
```

C++

```
for (int count = 0; count < 10; count++) {
   cout << count;
}
cout << "Done!";
```

For Loop

- Counting from 25 up to (excluding) 50 by 5s

Python

```
for count in range(25, 50, 5):
    print count
print "Done!"
```

C++

```
for (int count = 25; count < 50; count += 5){
   cout << count;
}
cout << "Done!";
```

Variable Scope

Variable Scope

- You can define new variables in many places in your code, so where is it in effect?
- A variable’s *scope* is the set of code statements in which the variable is known to the compiler.
- Where a variable can be referenced from in your program
- Limited to the code block in which the variable is defined
- For example:

```
if(age >= 18) {
   bool adult = true;
}
/* can't access adult here */
```

Scope Example

What will this code do?

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

int main() {
  int x = 3, y = 4;

  {
    int x = 7;
    cout << "x in block is " << x << endl;
    cout << "y in block is " << y << endl;
  }

  cout << "x in main is " << x <<  endl;

return 0;
}
```

Announcements

- The course policy agreement is due back in class by Tuesday, February 8th
- The add/drop date has been extended to February 10th
- Next Time: Functions and Arrays