C++ Primer
Part 1
CMSC 202

Topics Covered
• Our first “Hello world” program
• Basic program structure
• main()
• Variables, identifiers, types
• Expressions, statements
• Operators, precedence, associativity

A Sample C++ Program

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

int main() {
    int userInt;
    cin >> userInt;
    cout << "Hello, reader.\n" << "Welcome to C++!\n" << "What is the most programming language you know?\n";
    cin >> userInt;
    if (userInt == 1)
        cout << "Java\n" << "This is a great language to work with\n";
    else
        cout << "Enjoy the book, too!\n";
    return 0;
}
```
C++ Variables

• C++ Identifiers
  – Keywords/reserved words vs. Identifiers
  – Case-sensitivity and validity of identifiers
  – Meaningful names!

• Variables
  – A memory location to store data for a program
  – Must declare all data before use in program

Variable Declaration

• Syntax: `<type> <legal identifier>`;
• Examples:
  ```
  int sun;
  float average;
  double grade = 98;
  ```
  – Must be declared before being used
  – May appear in various places and contexts (described later)
  – Must be declared of a given type (e.g. int, float, char, etc.)
Variable Declarations (con’t)

When we declare a variable, we tell the compiler:

• When and where to set aside memory space for the variable
• How much memory to set aside
• How to interpret the contents of that memory: the specified data type
• What name we will be referring to that location by: its identifier

Naming Conventions

• Naming conventions are rules for names of variables to improve readability
• CMSC 202 has its own standards, described in detail on the course website
  - Start with a lowercase letter
  - Indicate "word" boundaries with an uppercase letter
  - Restrict the remaining characters to digits and lowercase letters
    topSpeed bankRate1 timeOfArrival

• Note: variable names are case sensitive!

Data Types:
Display 1.2 Simple Types (1 of 2)

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>WIDTH (bits)</th>
<th>SIGN</th>
<th>DECIMAL</th>
<th>PRECISION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>16</td>
<td>1</td>
<td>31–30</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Long</td>
<td>32</td>
<td>1</td>
<td>63–62</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Float</td>
<td>32</td>
<td>1</td>
<td>Approximately 7 digits</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>1</td>
<td>Approximately 15 digits</td>
<td></td>
</tr>
</tbody>
</table>
Data Types:
Display 1.2 Simple Types (2 of 2)

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bytes)</th>
<th>Approximate Size (number of decimal places)</th>
<th>Number of Digits</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>255</td>
<td>255</td>
<td>Not applicable</td>
</tr>
<tr>
<td>bool</td>
<td>1</td>
<td>True, False</td>
<td></td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. Precision refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types float, double, and long double are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

Assigning Data

• Initializing data in declaration statement
  – Results "undefined" if you don’t!
    • int myValue = 0;

• Assigning data during execution
  – Lvalues (left-side) & Rvalues (right-side)
    • Lvalues must be variables
    • Rvalues can be any expression
    • Example:
      distance = rate * time;
      Lvalue: "distance"
      Rvalue: "rate * time"

Data Assignment Rules

• Compatibility of Data Assignments
  – Type mismatches
    • General Rule: Cannot place value of one type into variable of another type
    • intVar = 2.99; //2 is assigned to intVar!
      • Only integer part "fits", so that’s all that goes
      • Called "implicit" or "automatic type conversion"
  – Literals
    • 2, 5.75, "2", "Hello World"
    • Considered "constants": can’t change in program
Escape Sequences

• “Extend” character set
• Backslash, \ preceding a character
  – Instructs compiler: a special “escape character” is coming
  – Following character treated as “escape sequence char”
  – Display 1.3 next slide

<table>
<thead>
<tr>
<th>SEQUENCE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>New line</td>
</tr>
<tr>
<td>\v</td>
<td>Carriage return (Positions the cursor at the start of the current line. May not work as you expect (it doesn’t always work).)</td>
</tr>
<tr>
<td>\t</td>
<td>(Horizontal Tab) (Advances the cursor to the right by a fixed amount of width.)</td>
</tr>
<tr>
<td>\a</td>
<td>Alarm (Sends the alert noise, typically a bell.)</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace (Advances you replace a backslash in a quoted expression.)</td>
</tr>
</tbody>
</table>

\' Single quote (Mostly used to place a single quote inside a single quote.)
\" Double quote (Mostly used to place a double quote inside a quoted string.)

The following are not as commonly used, but we include them for completeness:
\v Vertical tab
\b Backspace
\f Form feed
\? Question mark
Literal Data

• Literals
  – Examples:
    • 2    // Literal constant int
    • 5.75 // Literal constant double
    • 'Z'  // Literal constant char
    • "Hello World!" // 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:

**Display 1.4** Named Constant (1 of 2)

- Standard Arithmetic Operators
  - Precedence rules – standard rules

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

int main()
{
    double interest = 0.05;
    double deposit = 1000.0;
    double amount = deposit * interest;
    cout << "one year, that deposit will grow to:
" << amount << " an amount worth waiting for.
";
    return 0;
}
```

Arithmetic Operators:

**Display 1.4** Named Constant (2 of 2)

```
double calculations;
    calculations = deposit * interest; if (i) {
    cout << "one year, that deposit will grow to
" << calculations << " an amount worth waiting for.
";
    return 0;
```

Operators, Expressions

- Recall: most programming languages have a variety of **operators**; called unary, binary, and even ternary, depending 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) are called expressions

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.
  • `!`: logical negation, `true` is `false`, `false` is `true`
  • `++` and `--` are the increment and decrement operators
  • `++` a post-increment (postfix) operation
  • `--` a pre-increment (prefix) operation
• `++` and `--` are “shorthand” operators. More on these later...

Precedence, Associativity

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

- Casting for Variables
  - Can add ".0" to literals to force precision arithmetic, but what about variables?
  - We can’t use "myInt.0"!
  - static_cast<double>intVar
  - Explicitly "casts" or "converts" intVar to double type
  - Result of conversion is then used
  - Example expression:
    doubleVar = static_cast<double>intVar1 / intVar2;
    - Casting forces double-precision division to take place among two integer variables!

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;
  - This code segment produces the output:
    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;
  - This code segment produces the output:
    6
    3
  - Because pre-increment was used
Assigning Data: Shorthand Notations

- Display, page 14

<table>
<thead>
<tr>
<th>EXAMPLE</th>
<th>EQUIVALENT TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>count += 2;</td>
<td>count = count + 2;</td>
</tr>
<tr>
<td>total -= discount;</td>
<td>total = total - discount;</td>
</tr>
<tr>
<td>bonus *= 2;</td>
<td>bonus = bonus * 2;</td>
</tr>
<tr>
<td>time /= runFactor;</td>
<td>time = time/runFactor;</td>
</tr>
<tr>
<td>change %= 100;</td>
<td>change = change % 100;</td>
</tr>
<tr>
<td>amount += cent1 + cent2;</td>
<td>amount = amount + (cent1 + cent2);</td>
</tr>
</tbody>
</table>

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

Console Input/Output

- I/O objects cin, cout, cerr
- Defined in the C++ library called <iostream>
- Must have these lines (called preprocessor directives) near start of file:
  - #include <iostream>
  - using namespace std;
  - Tells C++ to use appropriate library so we can use the I/O objects cin, cout, cerr

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.

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.”
  - Cascading: multiple values in one cout

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.
Separating Lines of Output

- New lines in output
  - Recall: "\n" is escape sequence for the char "newline"
- A second method: object 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

C-strings

- C++ has two different kinds of "string 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 null.

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.
String type

• C++ added a data type of “string” to store sequences of characters
  – Not a primitive data type; distinction will be made later
  – Must add `#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 does not do what you expect…

Input Using cin

• `cin` for input, `cout` for output
• Differences:
  – “>>” (extraction operator) points opposite
    • Think of it as “pointing toward where the data goes”
    • Object name “cin” used instead of “cout”
  – No literals allowed for cin
    • Must input “to a variable”
• `cin >> num;
  – Waits on-screen for keyboard entry
  – Value entered at keyboard is “assigned” to `num`
Prompting for Input: cin and cout

- 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 as follows:
  Enter number of dragons:
- Underscore above denotes where keyboard entry is made
- Every cin should have cout prompt
  - Maximizes user-friendly input/output

---

Input/Output (1 of 2)

```
1-46
```

Input/Output (2 of 2)

```
1-47
```

---

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.
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.

Program 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.
Namespaces

- Namespaces defined:
  - Collection of name definitions
- For now: interested in namespace "std"
  - Has all standard library definitions we need
- Examples:
  - #include <iostream>
    using namespace std;
    - Includes entire standard library of name definitions
  - #include <iostream>
    using std::cin;
    using std::cout;
    - Can specify just the objects we want

Summary 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 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 overcomment
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 gcc Compiler

At the prompt, type

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

where `program.cpp` is the C++ program source file (the compiler also accepts “.cc” as a file extension for C++ source)

- `-Wall` is an option to turn on all compiler **warnings** (best for new programmers).

The Result: **a.out**

- 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.
UNIX Programming Tools

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