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()
{
    cout << "Hello, world!\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
    - bankRate
    - timeOfArrival
- Note: variable names are case sensitive!

Data Types:

**Display 1.2 Simple Types (1 of 2)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>0 to 255</td>
<td>Not applicable</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2^31 to 2^31-1</td>
<td>Not applicable</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>-2^63 to 2^63-1</td>
<td>Not applicable</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>approximately (10^{-38} \text{ to } 10^38)</td>
<td>7 digits</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>approximately (10^{-308} \text{ to } 10^{308})</td>
<td>15 digits</td>
</tr>
</tbody>
</table>
Data Types:

**Display 1.2 Simple Types (2 of 2)**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Significant Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>2 bytes</td>
<td>10</td>
</tr>
<tr>
<td>char</td>
<td>1 byte</td>
<td>All ASCII characters</td>
</tr>
</tbody>
</table>

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, 3.575, "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

### Display 1.3
Some Escape Sequences (1 of 2)

<table>
<thead>
<tr>
<th>SEQUENCE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\n</code></td>
<td>New line</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage return (Positions the cursor at the start of the current line. You are not likely to use this very much.)</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Tab (Advances the cursor to the next tab stop)</td>
</tr>
<tr>
<td><code>\a</code></td>
<td>Alarm (Sounds the alert noise, typically a bell)</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Backslash (Allows you to place a backslash in a quoted expression)</td>
</tr>
</tbody>
</table>

### Display 1.3
Some Escape Sequences (2 of 2)

- 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:

- Vertical tab
- Backspace
- 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 rate; // define rate variable
  double deposit; // define deposit variable
  cout << "Enter the amount of your deposit ":
  cin >> deposit;
  rate = 0.05; // set rate to 5%
  deposit = deposit * (1 + rate); // calculate new deposit
  cout << "Your deposit will grow to ":
  cout << deposit;
  return 0;
}
```

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

Arithmetic Operators:

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

```
double calculate:
  double deposit, years, balance = deposit;
  for (years = 1; years <= 10; years++)
  {
    balance = balance * (1 + 0.05);
    cout << "Balance after year ": years << " is ": balance << endl;
  }
```

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

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

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.
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:
  C.out << (( x > y ) ? "X is greater" : "Y is greater");
Unary Operators

• Unary operators only have one operand.
  
  - !
  
  - ++

  1. Logical negation; true is false, false is true
  
  2. ++ and -- are the increment and decrement operators
  
  3. ++ is a post-increment (postfix) operation
  
  4. -- is 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)
  
  • < > <= >=
  
  • == !=
  
  • &&
  
  • ||
  
  • ? :
  
  • 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; \(\rightarrow\) 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 /= rushFactor;</td>
<td>time = time/rushFactor;</td>
</tr>
<tr>
<td>change %= 100;</td>
<td>change = change % 100;</td>
</tr>
<tr>
<td>amount += cnt1 + cnt2;</td>
<td>amount = amount + (cnt1 + cnt2);</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 pre-processor 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:
  ```cpp```
cout << "Hello World\n";
  ```cpp```
- Sends string "Hello World" to display, & escape sequence "\n", skipping to next line
  ```cpp```
cout << "Hello World" << endl;
  ```cpp```
- Same result as above

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

Formatting Output

- Formatting numeric values for output
  - Values may not display as expected
    ```cpp```
cout << "The price is $" << price << endl;
    ```cpp```
  - 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.

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

Formatting Numbers

- "Magic Formula" to force decimal sizes:
  ```cpp```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
  ```cpp```
- These statements force all future cout'ed values to have exactly two digits after the decimal place:
  - Example:
    ```cpp```
cout << "The price is $" << price << endl;
    ```cpp```
  - Now results in the following:
    The price is $78.50
- Can modify precision "as you go" as well.
Formatting Integers

• Field width and fill characters
  – Must \ include <iomanip>
  – \texttt{setw(n)} sets field width to \texttt{n}
  – \texttt{cout.fill(c)} sets “fill” character to \texttt{c}
• Example:
  \begin{verbatim}
  int \texttt{x} = 7;
  cout.fill('0'); // set fill character to zero
  cout << \texttt{setw(3)} << \texttt{x} << \texttt{endl};
  \end{verbatim}
  Outputs \texttt{007} (left-pads with zeros)

C-strings

• C++ has two different kinds of “string of characters”:
  – the original C-string: array of characters
  – The object-oriented \texttt{string} class
• C-strings are terminated with a null character (‘\0’)
  \begin{verbatim}
  char \texttt{myString}[80];
  \end{verbatim}
  declares a variable with enough space for a string
  with 79 usable characters, plus null.

C-strings

• You can initialize a C-string variable:
  \begin{verbatim}
  char \texttt{myString}[80] = "Hello world";
  \end{verbatim}
  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”
  – Not a primitive data type; distinction will be made later
  – May need #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
**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
  - lowercase/upper 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