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

Display 1.1 A Sample C++ Program

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

int main( )
{
    int numberOfLanguages;

    cout << "Hello reader.\n"
         << "Welcome to C++.\n";

    cout << "How many programming languages have you used? ";
    cin >> numberOfLanguages;

    if (numberOfLanguages < 1)
        cout << "Read the preface. You may prefer\n"
             << "a more elementary book by the same author.\n";
    else
        cout << "Enjoy the book.\n";

    return 0;
}
```

Display 1.1
A Sample C++ Program (2 of 2)

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to C++.
How many programming languages have you used? 0
Read the preface. You may prefer
a more elementary book by the same author.
```

User types in 0 on the keyboard.

SAMPLE DIALOGUE 2

```
Hello reader.
Welcome to C++.
How many programming languages have you used? 1
Enjoy the book
```

User types in 1 on the keyboard.

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 sum;
float average;
double grade = 98;
```

Semicolon required!

  - 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)

Display 1.2 **Simple Types**

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits

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

`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true`, `false`	Not applicable

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, "Z", "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)

Display 1.3 **Some Escape Sequences**

SEQUENCE	MEANING
\n	New line
\r	Carriage return (Positions the cursor at the start of the current line. You are not likely to use this very much.)
\t	(Horizontal) Tab (Advances the cursor to the next tab stop.)
\a	Alert (Sounds the alert noise, typically a bell.)
\\	Backslash (Allows you to place a backslash in a quoted expression.)

Display 1.3
Some Escape Sequences (2 of 2)

\'	Single quote (Mostly used to place a single quote inside single quotes.)
\"	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\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: **Display 1.4** Named Constant (1 of 2)

- Standard Arithmetic Operators
  - Precedence rules – standard rules

Display 1.4 **Named Constant**

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

int main( )
{
    const double RATE = 6.9;
    double deposit;

    cout << "Enter the amount of your deposit $";
    cin >> deposit;
```


Arithmetic Operators: **Display 1.4** Named Constant (2 of 2)

```
    double newBalance;
    newBalance = deposit + deposit*(RATE/100);
    cout << "In one year, that deposit will grow to\n"
         << "$" << newBalance << " an amount worth waiting for.\n";

    return 0;
}
```

SAMPLE DIALOGUE

Enter the amount of your deposit $**100**
In one year, that deposit will grow to
$106.9 an amount worth waiting for.

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 **<u>true</u>** or **<u>false</u>** .

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

  x > y will always evaluate to **<u>false</u>** .
- 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 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; → 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

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);

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

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

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

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 "as you go" as well.

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 zero
cout << setw(3) << x << endl;
```

Outputs `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 *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"
  - 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

Input/Output (1 of 2)

Display 1.5 Using **cin** and **cout** with a string (part 1 of 2)

```
//Program to demonstrate cin and cout with strings
#include <iostream>
#include <string>

using namespace std;
int main( )
{
  string dogName;
  int actualAge;
  int humanAge;

  cout << "How many years old is your dog?" << endl;
  cin >> actualAge;
  humanAge = actualAge * 7;

  cout << "What is your dog's name?" << endl;
  cin >> dogName;

  cout << dogName << "'s age is approximately " <<
         "equivalent to a " << humanAge << " year old human."
         << endl;

  return 0;
}
```

Needed to access the **string** *class.*

Input/Output (2 of 2)

Display 1.5 Using **cin** and **cout** with a string (part 2 of 2)

Sample Dialogue 1

```
How many years old is your dog?
5
What is your dog's name?
Rex
Rex's age is approximately equivalent to a 35 year old human.
```

Sample Dialogue 2

```
How many years old is your dog?
10
What is your dog's name?
Mr. Bojangles
Mr.'s age is approximately equivalent to a 70 year old human.
```

"Bojangles" is not read into **dogName** *because* **cin** *stops input at the space.*

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