

Programming with C++ as a Second Language

Week 2 – Overview of C++

CSE/ICS 45C

Patricia Lee, PhD

ABSOLUTE C++

SIXTH EDITION



Walter Savitch

Chapter 1

C++ Basics

Copyright © 2016 Pearson, Inc.
All rights reserved.

PEARSON

Learning Objectives

- Introduction to C++
 - Origins, Object-Oriented Programming, Terms
- Variables, Expressions, and Assignment Statements
- Console Input/Output
- Program Style
- Libraries and Namespaces

Introduction to C++

- C++ Origins
 - Low-level languages
 - Machine, assembly
 - High-level languages
 - C, C++, ADA, COBOL, FORTRAN
 - Object-Oriented-Programming in C++
- C++ Terminology
 - *Programs and functions*
 - Basic Input/Output (I/O) with cin and cout

std::C++ Versions

- C++98: C++ 1998/2003 Standard
- C++11: C++ 2011 Standard
- C++14: C++ 2014 Standard (May/may not discuss new features in this version)
- C++17: (TBD in 2017)

Core Language vs. Standard Library

- Core Language is always available to all C++ programs
- Must explicitly ask for the parts of the standard library to be used
 - Via #include directives
 - Usually at beginning of program
 - Standard header: part of the C++ library, enclosed in angle brackets (< and >)

Display 1.1

A Sample C++ Program (1 of 2)

Display 1.1 A Sample C++ Program

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

3  int main( )
4  {
5      int numberOfLanguages;

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

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

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

15     return 0;
16 }
```

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 ← *User types in 0 on the keyboard.*

Read the preface. You may prefer

a more elementary book by the same author.

SAMPLE DIALOGUE 2

Hello reader.

Welcome to C++.

How many programming languages have you used? 1 ← *User types in 1 on the keyboard.*

Enjoy the book

Program Structure

- Free Form: spaces required only when they keep adjacent symbols separated
- 3 entities not free form:
 - string literals: chars in “” may not span lines
 - **#include *name***: must appear on line by itself, excepting comments
 - **// comments**:
 - // followed by text ends at end of current line
 - Comment with **/*** at beginning and ***/** at end is free form and can span multiple lines

Syntax

- Function: a piece of program that has a name that another program can call (cause to run)
 - **int main()**
 - Function Name: **main**
 - Return Type: **int** (core language datatype), type of data returned in **return** statement.
 - Parameters: that our function receives from the implementation, enclosed in parenthesis (in this case, none)
 - Every C++ program must define exactly one function named **main**

Syntax

- `{ }` [curly braces] is used around a sequence of zero or more statements and denotes that they should be treated as a unit (a block)
- `;` [semicolon] is used after an expression to create a statement (called an expression statement) – can have null statements (just semicolon)

Terminology

- Types: data structures/operations defined
 - Two types:
 - Core language (e.g. int)
 - Defined outside core language (e.g. std::ostream)
- Namespaces: mechanism for grouping related names
 - Standard library namespace: **std**
- String Literals (later slide)

Qualified Name/Scope Operator (:: operator)

- Left-associative (type is **std::ostream**)
 - **std::cout** is used with << [output operator]
 - **std::cin** is used with >> [input operator]
- Manipulator: manipulates stream
 - **std::endl** is used to end the current line of output (“\n” is used in the program excerpt)

C++ Variables

- C++ Identifiers
 - Keywords/reserved words vs. Identifiers
 - Case-sensitivity and validity of identifiers
 - *Programmer provides meaningful names
- Variables
 - A memory location to store data for a program
 - Must declare all data before use in program

Data Types:

Display 1.2 Simple Types (1 of 2)

Display 1.2 Simple Types

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
<code>short</code> (also called <code>short int</code>)	2 bytes (2*8 bits = 16 bits)	-32,768 to 32,767 ($2^{16} = 65,536$)	Not applicable
<code>int</code>	4 bytes	-2,147,483,648 to 2,147,483,647	Not applicable
<code>long</code> (also called <code>long int</code>)	4 bytes	-2,147,483,648 to 2,147,483,647	Not applicable
<code>float</code>	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
<code>double</code>	8 bytes	approximately 10^{-308} to 10^{308}	15 digits

Data Types:

Display 1.2 Simple Types (2 of 2)

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

C++11 Fixed Width Integer Types

TYPE NAME	MEMORY USED	SIZE RANGE
int8_t	1 byte	-128 to 127
uint8_t	1 byte	0 to 255
int16_t	2 bytes	-32,768 to 32,767
uint16_t	2 bytes	0 to 65,535
int32_t	4 bytes	-2,147,483,648 to 2,147,483,647
uint32_t	4 bytes	0 to 4,294,967,295
int64_t	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	8 bytes	0 to 18,446,744,073,709,551,615
long long	At least 8 bytes	

Avoids problem of variable integer sizes for different CPUs

New C++11 Types

- **auto**
 - Deduces the type of the variable based on the expression on the right side of the assignment statement

```
auto x = expression;
```
 - More useful later when we have verbose types
- **decltype**
 - Determines the type of the expression. In the example below, $x*3.5$ is a double so y is declared as a double.

```
decltype(x*3.5) y;
```

Terminology

- Declare/Define (Declaration/Definition): Give a type and name to variable or function
- Initialize (Initialization): First time a variable is assigned a value
- Use: When expression or function is utilized in an executed command

Assigning Data

- Initializing data in declaration statement
 - Results are "undefined" if you don't initialize
 - `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"

Assigning Data: Shorthand Notations

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

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 (not 2.99)`
 - 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

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]

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 1.4

Some Escape Sequences (1 of 2)

Some Escape Sequences

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

Display 1.4

Some Escape Sequences (2 of 2)

<code>\'</code>	Single quote (Mostly used to place a single quote inside single quotes.)
-----------------	--

<code>\"</code>	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:

<code>\v</code>	Vertical tab
-----------------	--------------

<code>\b</code>	Backspace
-----------------	-----------

<code>\f</code>	Form feed
-----------------	-----------

<code>\?</code>	Question mark
-----------------	---------------

Raw String Literals

- Introduced with C++11
- Avoids escape sequences by literally interpreting everything in parens

```
string s = R"(\t\t\n)";
```
- The variable `s` is set to the exact string `"\t\t\n"`
- Useful for filenames with `\` in the filepath

Constants

- Naming your constants
 - Literal constants are "OK", but provide little meaning
 - e.g., seeing 24 in a pgm, tells nothing about what it represents
- Use named constants instead
 - Meaningful name to represent data
const int NUMBER_OF_STUDENTS = 24;
 - Called a "declared constant" or "named constant"
 - Now use it's name wherever needed in program
 - Added benefit: changes to value result in one fix

Arithmetic Operators:

Display 1.5 Named Constant (1 of 2)

- Standard Arithmetic Operators
 - Precedence rules – standard rules

Named Constant

```
1  #include <iostream>
2  using namespace std;
3
4  int main( )
5  {
6      const double RATE = 6.9;
7      double deposit;
8
9      cout << "Enter the amount of your deposit $";
      cin >> deposit;
```

Arithmetic Operators:

Display 1.5 Named Constant (2 of 2)

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

14     return 0;
15 }
```

SAMPLE DIALOGUE

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

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 $\rightarrow 1 / 2$ equals 0
 - Then $\rightarrow 0 / 3.0$ equals 0.0
 - Then $\rightarrow 0.0 / 4$ 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
(double)17 / 5.5
Same expression as above, using explicit 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

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

String type

- C++ has 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

Input/Output (1 of 2)

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

```
1  //Program to demonstrate cin and cout with strings
2  #include <iostream>
3  #include <string> ← Needed to access the
                       string class.
4  using namespace std;
5  int main( )
6  {
7      string dogName;
8      int actualAge;
9      int humanAge;

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

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

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

18     return 0;
19 }
```

Input/Output (2 of 2)

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

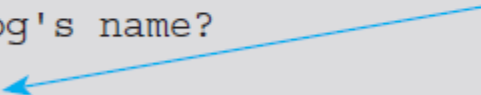
Sample Dialogue 1

```
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.



Formatting Output

- Formatting numeric values for output
 - Values may not display as you'd expect!
cout << "The price is \$" << price << endl;
 - If price (declared double) has value 78.5, you might get:
 - The price is \$78.500000 or:
 - The price is \$78.5
- We must explicitly tell C++ how to output numbers in our programs!

Formatting Numbers

- "Magic Formula" to force decimal sizes:
`cout.setf(ios::fixed);`
`cout.setf(ios::showpoint);`
`cout.precision(2);`
- These stmts 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!

Error Output

- Output with cerr
 - cerr works 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.

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

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