C++ Lecture 2

* Variables, Assignment Statements, Input & Output, and Data Types
* CIS 251 • Shelby-Hoover Campus

Structure of a C++ Program

* Every C++ program begins with the same basic structure
* Memorizing this structure through repetition makes it easier to focus on the elements that vary from one program to the next
* Note that C++ is case sensitive
  + Keywords must be entered exactly as required by the language (don’t use uppercase letters where lowercase letters are required, or vice versa)
  + Programmer-defined identifiers (names of variables, functions, and classes) must have consistent case

Directives

* A program must instruct the compiler as to what optional functionality to include as well as how to interpret names
* An **include directive** specifies a library from which a program uses variables and / or functions:  
    
   #include <iostream>
* A **using directive** specifies the namespace that defines specific identifiers in a program:  
    
   using namespace std;
* Many optional functions require both an include directive and the standard namespace

The Main Function

* C++ applications require the presence of a **main function**:  
    
   int main()
* The keyword intindicates that the function must contain a return statement with an integer (whole number) value; in main, this value is usually 0
* A set of parentheses follows a function’s name (empty for main)
* A set of curly brackets surrounds the contents of a function

The Structure as a Whole

* Programs in the first three chapters follow this pattern:  
    
  #include <iostream>  
  🡨 other include directives may be added as needed on separate lines

using namespace std;

int main()

{

🡨 The program code goes here

return 0;

}

Variables

* A **variable** is a named memory locations that holds various values during a program’s execution
* Each statement with a variable uses the current value of the variable, not necessarily the original
* Each variable must be **declared** before the program uses it
  + The declaration statement indicates the data type and name to be used
  + The data type indicates how much memory should be set aside for the variable

Declaration Statements

* Declaring a single variable
  + Pattern: typeName variableName;
  + Example: int numOfStudents;
* Declaring multiple variables of the same type
  + Pattern: typeName var1, var2, var3;
  + Example: int courseNum, capacity;
* Each declaration statement must end with a semicolon
* Don’t separate declarations for different variable types with a comma!

Rules for Identifiers

* Programmer-defined names must follow specific guidelines
  + The first character of an identifier must be a letter or an underscore
  + Other characters can be letters, underscores, or digits
  + Identifiers cannot include spaces
  + Be consistent with case (home vs. Home vs. HOME)
  + Class names typically start with an uppercase letter, whereas variables start with a lowercase letter
  + Use underscores or camel casing (capitalizing the first letters of the second and later words) in multiword variable names
  + Keywords (reserved words) are off limits as identifiers

Assignment Statements

* The assignment operator (=) assigns the value of the expression on the right side of the operator to the variable on the left side
* A different operator is used for comparison
* Be careful not to reverse the order: the variable receiving the value must be on the left, and the new value for the variable must be on the right!
* Each assignment statement ends with a semicolon

Numeric Literals

* A literal is a value written in the code itself (as opposed to a value entered by the user)
* Numeric literals may begin with a + or – sign to indicate positive or negative (assumed positive)
* Integer (whole number) literals may not contain any other punctuation
* Floating-point (decimal) literals may contain a decimal point
* Floating-point literals may also be written in **scientific / e notation**:
  + 4600 = 4.6e3 (4.6 x 103)
  + 0.00075 = 7.5e-4 (7.5 x 10-4)

Initialization

* Each compiler assigns a default value to variables that can cause errors
* A variable may be **initialized** (assigned an initial value) when it is declared
* To initialize a variable, add an equal sign and the initial value of the variable after the variable’s name in the declaration statement:
  + int courseNum = 251;
  + int sum = 0, count = 0;
* Input variables may not require initialization

Simple Output

* cout is the **output stream** a program uses to output text to the console (c stands for console)
* The **output insertion operator** << connects the items to be displayed in a single output statement
* Enclose a literal string of text to display in quotation marks:  
    
   cout << "Welcome to C++!";
* Variables and strings may be combined in output by separating the items using the output insertion operator (but don’t put quotation marks around a variable name):  
    
   cout << "I am " << age << " years old.";

Escape Sequences

* An **escape sequence** is a special character that otherwise can’t be typed
* Each escape sequence starts with the backslash (\)
  + \n 🡨 new line within a string
  + \t 🡨 horizontal tab to the next tab stop
  + \\ 🡨 printable backslash
  + \' 🡨 printable single quote
  + \" 🡨 printable double quote
  + \a 🡨 system beep
* Be careful not to use the forward slash (/) in an escape sequence, as it means something else in C++

Formatting Output

* To force the console to move to the next line, use the \n escape sequence (in quotes) or endl:  
   cout << "Have a nice day!\n";  
   cout << "Have a nice day!" << endl;
* Environments have different default settings for floating-point numbers, so a program may require all of the following statements to format these numbers properly:
  + cout.setf(ios::fixed); 🡨 force fixed (non-scientific) notation
  + cout.setf(ios::showpoint); 🡨 require the decimal point
  + cout.precision(2); 🡨 require 2 digits after the decimal point

Simple Input

* cin is the **input stream** a program uses to accept input from the keyboard via the console
* The **input insertion operator** >> connects the variables to hold the input in a single input statement (note that this operator is not the same as the operator used for output)
* Type a variable name without quotation marks:  
   cin >> myVariable;
* A program may capture input for several variables in a single statement:  
   cin >> var1 >> var2;
* A cin statement never ends with endl or "\n"!
* The user must separate input items with spacing

Interactive Programs

* When requiring input, make sure the user knows how to proceed
* A short message known as a **prompt** should let the user know what data to enter, how it should be formatted, etc.
* The user’s response may be on the same line as the prompt or on a separate line
* A program may also **echo** input back to the user to verify that the values have been entered correctly

Data Types (1)

* The specifics of each type will vary by compiler (the values below are examples)
* Integer (whole number) Types:
  + short (short int): consumes 2 bytes, ranges from -32,768 to +32,767
  + int: consumes 4 bytes, ranges from -2,147,483,648 to +2,147,483,647
  + long (long int): at least the same size as int
  + Adding the key word unsigned to the beginning of the declaration (before the data type) causes a variable of one of these three types to store only non-negative values (0 or larger)

Data Types (2)

* Floating-Point (decimal) Types:
  + float: consumes 4 bytes, ranges from roughly ±10-38 to ±1038 with 7 digits of accuracy
  + double: consumes 8 bytes, ranges from roughly ±10-308 to ±10308 with 15 digits of accuracy
  + long double: consumes 10 bytes, ranges from roughly ±10-4932 to ±104932 with 19 digits of accuracy
* char: stores a single ASCII (one-byte) character in single quotes (Appendix 3)

Data Types (3)

* bool: stores a value of true or false
  + Don’t put quotation marks around these values
  + Results of comparisons can be stored in these variables
* string: stores a group of characters
  + A class type (size may vary)
  + Not included by default: requires #include <string> and the standard namespace
  + Values assigned to a string variable must be within quotation marks
  + The **concatenation** operator (+) combines two strings to make one long string

Literal Values

* The declared data type of a variable determines what values can be assigned to it
* Numeric literal values should not have any quotes:  
    
   age = 25;
* Character literal values should be placed in single quotes:  
    
   gender = 'M';
* String literal values should be placed in double quotes:  
    
   name = "Priscilla";

Data Type Compatibility

* Compilers may allow the mixture of data types in an assignment statement, but the result may be harmful
* Assigning a floating-point number to an integer variable results in **truncation** (the decimal portion is cut off with no rounding)
* Characters are really small integers, so a program may manipulate them as integer variables, but this may not be the best idea
* C++ uses 1 and 0 as the values for true and false
  + Any nonzero value is interpreted to mean true
  + This becomes dangerous when confusing the assignment operator and the equality comparison operator

Arithmetic Operators

* Unary operators require only one operand (e.g., negation)
* Binary operators require two operands
  + Addition: +
  + Subtraction: -
  + Multiplication: \*
  + Division: /
  + Modulus: % (percent symbol)

The Modulus Operator

* When two integers are used in a division operation, the remainder is lost
* The modulus operator can be used to obtain the remainder in integer division  
  + leftOver = 9 % 5;  
    // leftOver is 4 (5 goes into 9 once with 4 left over)
  + remainder = 12 % 3;  
    // remainder is 0 (3 goes into 12 four times with 0 left over)
  + discard = 4 % 10;  
    // discard is 4 (10 doesn’t go into 4, so 4 is left over)
* This operator can be used to determine when a number is a multiple of another number
  + if a % 2 is zero, then a is even
  + if a % 2 is one, then a is odd

Mixing Types in Arithmetic

* Operands are considered in pairs
* If both operands are of the same type, the result will be of that type
* If an expression involves operands of differing types, the operand of the lowest-ranking type will be converted to the highest-ranking type
  + If both are in the same category (integer or floating-point), the largest capacity type will be chosen
  + When there’s a mixture of an integer type and a floating-point type, the floating-point type will be used

Arithmetic Guidelines

* Integer division always **truncates** the result
  + The decimal component is lost; no rounding up
  + Use at least one floating-point operand to prevent this
* Operators have varying levels of precedence
  + Unary operators are highest (evaluated first)
  + Multiplication, Division, Modulus
  + Addition, Subtraction are lowest (evaluated last)
* Operations of the same level are evaluated left to right
* Parentheses can be used to override precedence

Combined Assignment

* Frequently a programmer chooses to modify the value of an existing variable:  
    
   count = count + 1; // adds 1 to count
* C++ contains several combined assignment operators to shorten this action
  + Operators: +=, -=, \*=, /=, %=
  + Example: count += 1;
  + Example: price -= discount;

Increment and Decrement

* A frequent operation in programming is to add 1 to, or subtract 1 from, the existing value of a variable
* There are already two ways to write this:  
    
  count = count + 1; count = count – 1;  
  count += 1; count -= 1;
* The increment and decrement operators perform these operations in a compact statement:  
    
  count++; count--;
* These operators can only be applied to variables (not complete expressions or constants)

Prefix vs. Postfix (Ch. 3)

* The increment and decrement operators can be written before or after the variable being changed
* This difference only has an effect when the operator is embedded within another statement:  
  + x = y \* (count++);
    - Use the existing value of count in calculating x
    - Then, add 1 to count
  + x = y \* (++count);
    - Add 1 to count
    - Then, use the new value of count in calculating x
* The result may not be what is intended, so a programmer may prefer to write the increment and decrement operations in separate statements to control the order

Documentation and Style

* The compiler does not require specific spacing in most circumstances
* Nevertheless, indentation and spacing makes C++ code easier to read and modify
* A program is considered to be **self-documenting** if the identifiers and style used in the code make the program’s details obvious
* Additional explanation may be added via **comments**
  + Line comments begin with two forward slashes:   
      
    // the compiler ignores everything to the end of the line
  + Block comments have special markers at the beginning and end and may span multiple lines:   
      
    /\* the compiler ignores everything between the markers,  
     no matter how many lines there are \*/

Style Guidelines

* Use enough comments, but don’t overdo them
  + Explanatory / introductory comments at top
  + Non-obvious code details: units, implementation of logic, relationship with an algorithm
* Indent lines of code to group them
  + Block levels (function, selection / loop)
  + Multi-line statements
  + Consistent spacing

Named Constants

* The value of a variable can change as often as needed during a program
* Adding the key word const to the beginning of a declaration (before the data type) creates a **named constant** instead of a variable
  + const typeName CONST\_NAME = value;
  + The name of a named constant usually contains all uppercase letters
  + The program must initialize the named constant in the declaration; it cannot assign a new value to the constant later
* Example:  
    
  const double PRICE\_PER\_GAL = 3.199;
* Named constants make the value used in multiple formulas consistent; each statement that needs the constant contains the name rather than using the value directly