C++ Lecture 1

* Introduction to Computers and C++ Programming
* CIS 251 • Shelby-Hoover Campus

Computer Basics

* Hardware
  + Input Devices: Keyboard, Mouse
  + Output Devices: Monitor, Printer
  + Processor (CPU)
  + Main Memory (RAM)
  + Secondary Memory (Storage)
* Software
  + Program: a set of instructions for the computer to perform, usually on a set of data
  + Most application software communicates with the computer through the operating system

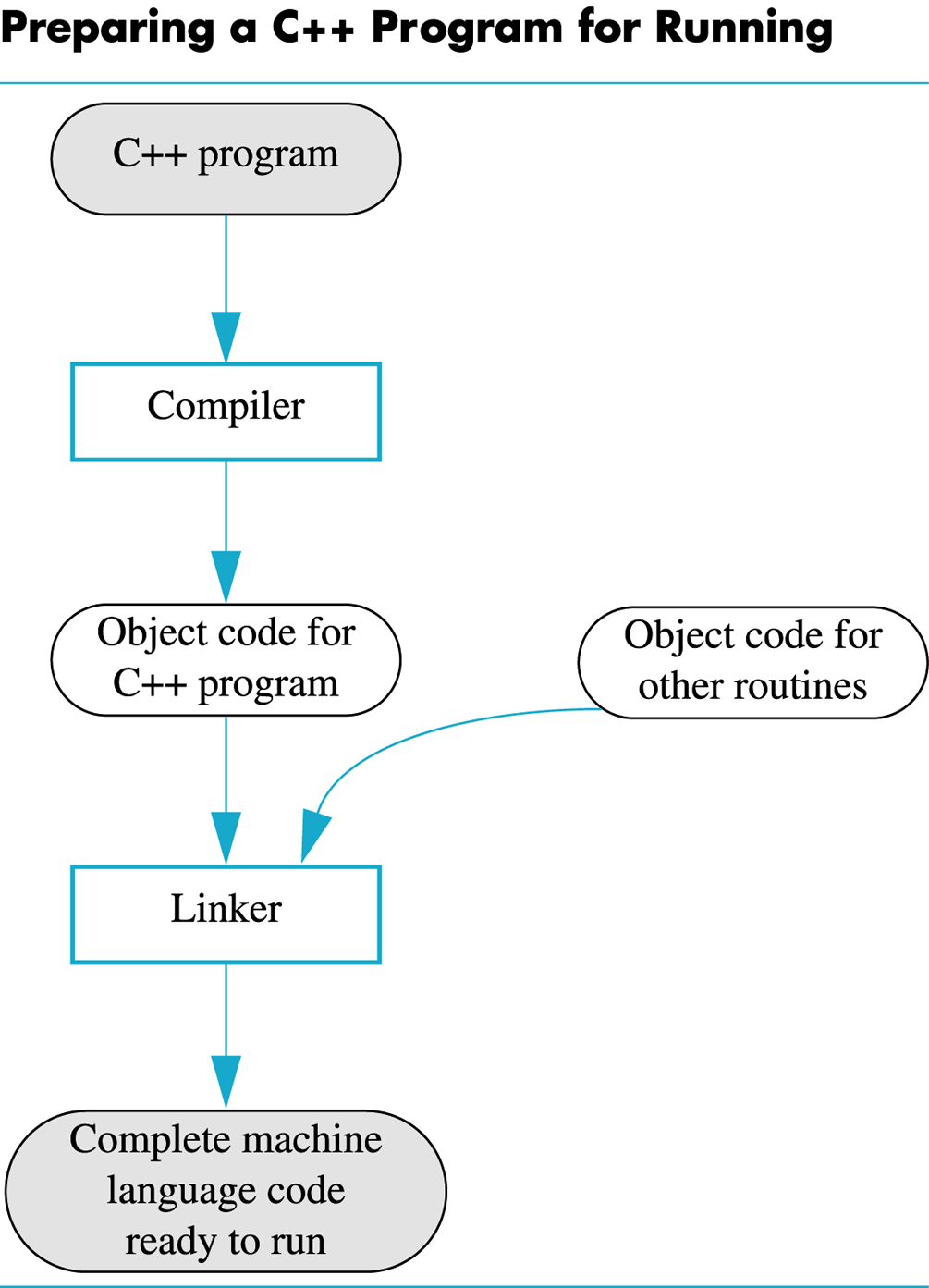
Programming Basics

* Underlying information is stored / manipulated in **binary** (0 for off, 1 for on)
* Processors understand rudimentary memory operations written in a **low-level language**
  + machine language (binary)
  + assembly language (simple conversion to binary)
* Programmers usually write programs in a **high-level language**
  + C++, C#, Java, Basic, Pascal, FORTRAN, COBOL
  + Easier for humans to write, understand

High Level to Low Level

* Code written in a high-level language must be converted to a low-level language form before the instructions can be executed
* A **compiler** converts a set of high-level language instructions into a low-level language all at once
  + The high-level language code is known as **source code**
  + The low-level language code that results is **object code**
* In C++, an additional tool, the **linker**, combines the object code from the compiler with the object code for predefined functions to generate the **executable** form
* Many environments combine the compile and link commands into a single operation (build)

Compiling and Linking



Language Components

* **Keywords / Reserved Words**: words with predefined meaning (cannot be used for other purposes)
* **Operators**: symbols or words that perform operations
* **Punctuation**: characters that mark the beginning or end of a statement or block, separate items in a list, etc.
* **Programmer-Defined Names / Identifiers**: identify memory locations, code units, new data types
* **Syntax**: rules that govern how the various language elements are combined to form statements and programs

The Programming Process

* Remember: the point at which you enter code in a specific language should not be the beginning point of the programming process
* There are several different methods that may be applied to a problem to generate a solution
* Problem-Solving Phase
  + Define the problem
  + Craft an algorithm (flowchart, pseudocode)
  + Test the algorithm and refine if necessary
* Implementation Phase
  + Express the algorithm in a specific language
  + Correct syntax errors that inhibit compilation
  + Test the resulting executable and refine the code if necessary

Programming Approaches

* **Procedural Programming** separates the data from the instructions performed on the data; the focus is on the sets of instructions (procedures, functions, methods)
* **Object-Oriented Programming** unites individual data elements into units with predefined interaction
  + **Object**: a group of variables representing some entity
  + **Method**: a procedure that can be used to access or modify the data stored in the object
  + **Class**: the code that defines the makeup of a particular type of object, including the data each object contains and what operations can be performed on that object

An Introduction to C++

* Derived from the C language
  + Developed at AT&T Bell Laboratories in the 1970s
  + Used for writing and maintaining UNIX
  + Contains both high-level and low-level attributes
* C++ is a variation on the C language
  + Developed at AT&T Bell Laboratories in the early 1980s
  + “One better” than C (the ++ operation adds one)
  + Supports object-oriented programming
  + Instructions written in C are also compatible with C++ (though the reverse is not always true)

C++ Structure

* Your initial programs will follow this pattern:

#include <iostream>

using namespace std;

int main()

{

🡨 Your code goes here

return 0;

}

Some Basic C++ Syntax

* In C++, a statement (a single instruction) ends with a semicolon
* Declaration
  + Listing the data type and name of a variable to be used
  + Example: int studentCount; (int is used for whole numbers)
* Input and Output
  + The key word cout represents generating output; cin represents obtaining input
  + cout << "Hello!\n"; displays Hello! and starts a new line
  + cin >> studentCount; stores the next item the user enters as input in the variable studentCount (note the difference in the brackets)
* Assignment
  + Storing a predetermined value in a variable
  + The variable must be on the left side of the equal sign; the value goes on the right
  + studentCount = 15; assigns 15 to the variable
  + studentCount = studentCount + 1; adds 1 to the existing value of the variable and assigns the sum as the new value

C++ Compilers

* The code presented in this course conforms to the C++ standard
* Older compilers may use a pre-standard version of C++ syntax
  + #include <iostream.h> instead of <iostream>
  + using namespace std; omitted entirely
  + void main() instead of int main()
  + return 0; omitted entirely

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

C++ Lecture 3

* Simple Branching, Boolean Expressions, Multiway Branching, Scope
* CIS 251 • Shelby-Hoover Campus

Flow of Control

* The material from the previous lecture allows the creation of programs in which the same statements are executed in sequence every time the program executes
* Most programs involve some **flow of control**
  + Allowing the user to select from a set of options and executing statements that correspond to the user’s choice
  + Repeating a set of statements until the user enters a value to exit
  + Processing every record in a file until reaching the end of the file
* Two major types of flow of control
  + Branching (selection, decision): a one-time decision to take one of two (or more) courses of action depending on the value of a variable or expression
  + Looping (repetition, iteration): a repeated decision to execute a series of statements as many times as a certain condition remains in place

Simple Branching

* In the simplest form of branching, a program executes a certain set of statements if a condition evaluates to true and (optionally) an alternate set of statements if the condition evaluates to false
* Syntax: the **if-else statement**  
    
  if (Boolean\_expression)  
   statement\_executed\_if\_true;  
  else  
   statement\_executed\_if\_false;
* Notice that the lines beginning with if and else do **not** end with semicolons
* C++ does not require an “end if” clause

Simple Branching Example

* Determining tuition for full-time versus part-time students:  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   int hours;  
   double tuition;  
   const double FEE\_PER\_HOUR = 138.00;  
   cout << "Enter hours enrolled: ";  
   cin >> hours;  
   if (hours >= 12)  
   tuition = 1500.00;  
   else  
   tuition = hours \* FEE\_PER\_HOUR;  
   cout.setf(ios::fixed);  
   cout.setf(ios::showpoint);  
   cout.precision(2);  
   cout << "Your tuition is $" << tuition << endl;  
   return 0;  
  }

Compound Statements

* More often than not, more than one statement should be executed for a particular outcome of a Boolean expression
* These statements are combined into a **compound statement** by the use of curly braces / brackets:  
    
  if (hours >= 12)  
  {  
   cout << "Full-time tuition applies.\n";  
   tuition = 1500.00;  
  }  
  else  
  {  
   cout << "Part-time tuition applies.\n";  
   tuition = hours \* FEE\_PER\_HOUR;  
  }

Branching and Brackets

* The else clause is optional; if there are no statements to execute if the Boolean expression is false, it may be omitted
* You may place brackets around a single statement to be executed, but you **must** place brackets around a compound statement
* The closing bracket of a compound statement for the if clause should come before the else clause (don’t nest the else clause inside the compound statement)
* You may mix a single statement for one clause and a compound statement for the other

Boolean Expressions

* A **Boolean expression** is an expression that evaluates to true or false
* Most Boolean expressions are comparisons involving one of six comparison operators:
  + equals: ==
  + does not equal: !=
  + is less than: <
  + is less than or equal to: <=
  + is greater than: >
  + is greater than or equal to: >=

Complex Expressions

* Multiple Boolean expressions may be joined together to create a single expression with one result
  + && (and): both subexpressions must be true
  + || (or): at least one subexpression must be true
  + C++ requires parentheses around the entire expression; parentheses around the subexpressions are optional (but recommended)
* To negate the result of a Boolean expression, use the ! (not) operator before the expression in parentheses
  + Example:  
      
    if (!(age >= 21))
  + To avoid using the not operator, reverse the operation:  
      
    if (age < 21)

Expression Pitfalls

* Each Boolean expression in an && or || must be complete by itself:
  + WRONG: if (90 < x < 100)
  + STILL WRONG: if (x > 90 && < 100)
  + RIGHT: if ((x > 90) && (x < 100))
  + ALSO RIGHT: if ((90 < x) && (x < 100))
* You **must** use two equal signs for comparison
  + A single equal sign assigns (it does not compare)
  + The Boolean value of an assignment statement is false if the value assigned is zero; otherwise, it’s true
  + The compiler will **not** flag this as a syntax error

Operation Precedence

* In addition to the order of operations for arithmetic, the assignment and Boolean operations have specific **precedence rules**
  + Highest: unary operations + - ++ -- !
  + Some binary arithmetic operations \* / %
  + Other binary arithmetic operators + -
  + Some comparison operators < > <= >=
  + Other comparison operators == !=
  + &&
  + Lowest: ||
* Remember: a program can override the normal order of operations using parentheses (e.g., if you want an || operation to be performed before an &&)

Short-Circuit Evaluation

* To optimize efficiency, C++ performs **short-circuit evaluation**
  + &&: if the subexpression on the left is false, the overall result will be false, so it skips the subexpression on the right
  + ||: if the subexpression on the left is true, the overall result will be true, so it skips the subexpression on the right
* Other languages that evaluate both subexpressions regardless of the outcome are said to perform **complete evaluation**
* Practical use: avoiding division by zero  
    
  if ((creditHours > 0) && (totalPoints / creditHours > 2.0))  
   cout << "Eligible for membership." << endl;

Enumeration Types

* Sometimes a variable needs to be restricted to hold only one of a preset list of values
* A simple way to accomplish this in C++ is by the use of an **enumeration type**
* The declaration of an enumeration type begins with the key word enum and a programmer-defined type name
* Then, in a set of curly braces, the programmer provides names for several constants (usually in uppercase letters) separated by commas
  + These constants can be initialized to any int value
  + If none are initialized, C++ defaults to consecutive values starting at 0
  + If some are initialized, the others are assigned consecutive values continuing from the previously assigned constant
* After the closing curly brace, the programmer enters a semicolon (which is rare – most blocks end with a curly brace but no semicolon)

Defining Enumeration Types

* In this definition, all of the constants are initialized:  
  enum MonthLength { JAN\_LENGTH = 31, FEB\_LENGTH = 28, MAR\_LENGTH = 31,   
   APR\_LENGTH = 30, MAY\_LENGTH = 31, JUN\_LENGTH = 30, JUL\_LENGTH = 31,   
   AUG\_LENGTH = 31, SEP\_LENGTH = 30, OCT\_LENGTH = 31, NOV\_LENGTH = 30,   
   DEC\_LENGTH = 31 };
* In this definition, none of the constants are initialized:  
  enum Division { BUS, CIS, OAD }; // BUS == 0, CIS == 1, OAD == 2
* In this definition, some of the constants are initialized:  
  enum RandomConstants { ALPHA = 31, BETA, GAMMA, DELTA = -15, PHI };  
  // BETA == 32, GAMMA == 33, PHI == -14

Using Enumeration Types

* Any constant in an enumeration type can be assigned to a variable of type int:  
  int lengthOfQuarter = JAN\_LENGTH + FEB\_LENGTH + MAR\_LENGTH;
* A program may contain a variable of an enumeration type that can only be assigned the constants for that type:  
  Division myDivision = CIS;
* Be careful not to declare a separate constant with the same name as one of the enumeration type constants

Nesting Statements

* Any block can contain another similar block nested within it
* When nesting an if statement inside of another if statement, be careful with the pairing of else statements with if statements
  + If there are no brackets, the else statement is assumed to correspond to the nearest (most recent) if (the dangling else problem)
  + If the else should be matched with the outer if, not the nested if, place the nested if in brackets:  
      
    if (creditScore < 750)  
    {  
     if (creditScore < 500)  
     cout << "You do not qualify." << endl;  
    }  
    else  
     cout << "You get our best rate!" << endl;

Multiway Branching

* Evaluating a series of Boolean expressions, stopping with the first true result, is known as **multiway branching**
* Multiway branching can be accomplished by nesting an if statement inside an else block
* It can also be accomplished using the shorthand “else-if” notation, which requires less indentation:  
    
  if (guess > number)  
   cout << "Too high." << endl;  
  else if (guess < number)  
   cout << "Too low." << endl;  
  else if (guess == number)  
   cout << "Correct!" << endl;

The Trailing else

* If a program should execute a statement (simple or compound) when none of the expressions in a multiway branch are true, use a trailing else with no Boolean expression  
    
  if (grade >= 90)  
   cout << "You made an A!" << endl;  
  else if (grade >= 80)  
   cout << "You made a B!" << endl;  
  else if (grade >= 70)  
   cout << "You made a C." << endl;  
  else if (grade >= 60)  
   cout << "You made a D." << endl;  
  else  
   cout << "You failed." << endl;
* A Boolean expression never follows a plain else

The switch statement

* An alternative for multiway branching when a single expression is compared to several unique values is the **switch statement**
* The controlling expression (to be compared to each of the values) must evaluate to a value of type bool, one of the integer types, a char, or an enum constant
* Each value has its own case:  
    
  switch (controlling\_expression)  
  {  
   case value1:  
   statement\_for\_value1;  
   break;  
   case value2:  
   statement\_for\_value2;  
   break;  
   // continue with as many case values as needed  
   default:  
   statement\_for\_no\_match;  
  }

switch Details

* Once C++ finds a match between the expression and a particular case value, execution begins with the first statement beneath the case and continues until it reaches a break statement or the switch structure’s closing bracket
  + The statements to be executed for a case value do not require a set of brackets (the only brackets required are those for the entire structure)
  + Omitting the break statements may cause the statements for multiple cases to be executed
* Multiple cases that involve the same resulting action may be stacked together:  
    
   case 'F':  
   case 'f':  
   cout << "You are female.\n";  
   break;
* The optional default section operates in a manner similar to the trailing else, executing if none of the case values match

switch Example

* Displaying the user’s gender based on a letter:  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   char gender;  
   cout << "Please enter the letter of your gender (M or F): ";  
   cin >> gender;  
    
   switch (gender)  
   {  
   case 'F':  
   case 'f':  
   cout << "You are female." << endl;  
   break;  
   case 'M':  
   case 'm':  
   cout << "You are male." << endl;  
   break;  
   default:  
   cout << "Ask your physician." << endl;  
   }  
    
   return 0;  
  }

switch Applied

* A switch structure can be used to organize the code executed for several menu options
  + Display the menu
  + Prompt the user for a selection
  + Create a switch structure with the variable holding the user’s selection as the controlling expression
  + List the statements executed for each menu option under a case statement for that value, with a break statement at the end of each option’s statement set
* switch can only be used with individual values (a case statement may not contain a comparison operator)

Blocks and Scope

* A **block** is a segment of code between curly brackets
* A variable declared inside a block is only available within that block
  + A variable is **local** to the block in which it is declared
  + Where a variable can be used is its **scope**
* Each block can have its own set of variables, so a nested block can contain a declaration for a variable with the same name as a variable in the outer block
  + Access to the inner block variable is limited to the block
  + Access to the outer block variable is prohibited inside the block
  + Naming a variable in an inner block with the same name as a variable in its outer block is not a good idea
  + Use different names instead

C++ Lecture 4

* Simple Looping, Loop Control, Loop Options, Applications
* CIS 251 • Shelby-Hoover Campus

Looping

* In any of the branching structures, the Boolean (or controlling) expression is only evaluated one time; execution of the program continues with the statements after the branching structure
* Each loop structure also involves a Boolean expression
  + The Boolean expression may be evaluated before the first time the loop’s statements are executed
  + After each repetition of the loop’s statements, the program returns to the Boolean expression to see if the condition remains true
  + Repetition continues until the Boolean expression is false, at which point the program proceeds to the statements after the loop structure

Simple Looping

* The simplest loop form involves evaluating a condition before executing a statement or group of statements; it reevaluates the condition after each execution to determine whether to repeat
  + The statements executed each time the condition is true are known as the **loop body**
  + Each repetition of the loop body is known as an **iteration**
* Syntax: the **while statement**  
    
  while (Boolean\_expression)  
  {  
   statements\_executed\_as\_long\_as\_true;  
  }

Comparing if and while

* Both can have a single or compound statement as the body executing if / while the Boolean expression is true; curly braces are required for compound statements
* Neither has a semicolon at the end of its header
* A while loop never has an else statement (once the while loop terminates, the program proceeds to the code that comes after the loop)
* After a while loop body executes, control of the program returns to the loop’s Boolean expression to determine if another iteration is needed; the Boolean expression of an if statement won’t be evaluated again

Infinite Loops

* The loop body should contain a statement that modifies a variable involved in the Boolean expression
* Failure to modify one of these variables may result in an infinite loop if the Boolean expression never evaluates to false:  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   int greetings = 0;  
   while (greetings < 10)  
   cout << "Hello!" << endl;  
    
   return 0;  
  }

Count-Controlled Loops

* In a count-controlled loop, a loop control variable is assigned some starting value before the loop, compared to a stopping value, and modified inside the loop body:  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   int count = 1;  
    
   while (count < 6)  
   {  
   cout << "Perform step " << count << endl;  
   count++;  
   }  
    
   cout << "Program complete." << endl;  
    
   return 0;  
  }
* The loop control variable may have any name (count is frequently used but not required)
* The statement to modify the loop control variable may be anything (addition, subtraction, etc.)

Sentinel-Controlled Loops

* In a sentinel-controlled loop, the loop control variable is modified by getting more input from the user (or a file):  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   int quantity;  
   double cost;  
    
   cout << "Enter the first order quantity: ";  
   cin >> quantity;  
    
   while (quantity > 0)  
   {  
   cost = quantity \* 1.99;  
   cout << "Order cost: $" << cost << endl;  
    
   cout << "Enter another order (enter 0 or less to exit): ";  
   cin >> quantity;  
   }  
    
   cout << "Thanks for using the order cost calculator." << endl;  
    
   return 0;  
  }

The do-while Loop

* The while loop is known as a **pretest** loop
  + The controlling question is asked before the first iteration
  + If the question is false the first time it is evaluated, there will be zero iterations of the loop body
* A program may require a **posttest** loop that executes one iteration of the loop body before evaluating the question
* In C++, the posttest loop is **do-while**:  
    
  do  
  {  
   statements\_to\_repeat;  
  } while (Boolean\_expression);

do-while Characteristics

* In do-while, the Boolean expression is placed after the loop body and followed with a semicolon (this is the only selection or loop structure that has a semicolon after the closing parenthesis of the Boolean expression)
* A do-while loop can be used when the repetition depends on the user’s response:  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   char ans;  
    
   do  
   {  
   cout << "Hello" << endl;  
   cout << "Want another greeting?" << endl;  
   cout << "(Y = Yes, N = No): ";  
   cin >> ans;  
   } while ((ans == 'Y') || (ans == 'y'));  
    
   cout << "Goodbye" << endl;  
    
   return 0;  
  }

The for Loop

* There are three actions required in every loop
  + Provide a starting value for the loop control variable before evaluation
  + Test / evaluate the loop control variable
  + Update the loop control variable
* A third type of loop, the **for loop**, places all three actions in the header, with semicolons between the actions:  
    
  for (start\_action; test; update)  
  {  
   statements\_to\_repeat;  
  }

for Loops in Execution

* The first portion of the for loop header executes only at the very beginning of the loop execution
* The Boolean expression is evaluated before proceeding to the loop body
* The update step is performed after the loop body executes; then, the Boolean expression is reevaluated
* for loops are especially useful as count-controlled loops (e.g., labeling each line of output with a number):  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   int line;  
    
   for (line = 1; line <= 10; line++)  
   cout << line << ". " << endl;  
    
   return 0;  
  }

for Loop Options

* The first and third sections of a for loop header may contain more than one statement, separated by commas instead of semicolons (the semicolons separate the sections):  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   int a, b;  
    
   for (a = 1, b = 10; a <= b; a++, b--)  
   cout << a << " times " << b << " equals " << (a \* b) << endl;  
    
   return 0;  
  }

A program may also omit the statements for these sections from the for loop header if their actions are accomplished elsewhere (but the semicolons that separate the sections must be included):  
  
#include <iostream>  
using namespace std;  
  
int main()  
{  
 int a = 1, b = 10;  
  
 for ( ; a <= b; )  
 {  
 cout << a << " times " << b << " equals " << (a \* b) << endl;  
 a++;  
 b--;  
 }  
  
 return 0;  
}

More for Loop Details

* The Boolean expression may be composed of several individual Boolean expressions joined by &&, ||
* The loop control variable(s) may be declared in the first section of a for loop header, though these variables only exist as long as the loop executes (they are local to the loop):  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   for (int val = 10; val > 0; val--)  
   cout << val << endl;  
    
   // val no longer exists here  
    
   return 0;  
  }
* Be careful not to write the update action both in the header and in the loop body, or it will execute twice

Loop Commonalities

* In every loop, the program must provide a starting value for the loop control variable before reaching a Boolean expression, test that variable in the Boolean expression, and update it such that the test eventually evaluates to false
* Only the do-while loop has a semicolon after the closing parenthesis surrounding the Boolean expression (while and for do not)
* Only the for loop has semicolons inside the parentheses (two)
* Two statements can be used to alter loop execution (these are generally avoided because they make the loop more difficult to follow):
  + break; // causes the loop to terminate immediately
  + continue; /\* ends the current iteration, proceeds to the next step after the iteration (evaluating the Boolean expression or executing the update statement) \*/

Loop Usage: Statistics

* A program uses a loop to compute a sum or product of a set of input values
  + A variable storing a sum should be initialized to 0
  + A variable storing a product should be initialized to 1
  + Each time the user enters a value, the loop adds the input to the sum or multiplies the product by it
* The average of a set of values is the sum divided by the count

Sum Loop Example

* Calculating the sum (and average) of ten values:  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   int sum = 0, value;  
   double average;  
    
   cout << "Enter ten values:" << endl;  
    
   for (int a = 1; a <= 10; a++)  
   {  
   cin >> value;  
   sum += value; // sum = sum + value  
   }  
    
   average = sum / 10.0; // real number division  
    
   cout.setf(ios::fixed); // number formatting  
   cout.setf(ios::showpoint);  
   cout.precision(1);  
    
   cout << "Sum of the values: " << sum << endl;  
   cout << "Average value: " << average << endl;  
    
   return 0;  
  }

Loop Usage: Termination

* A count-controlled loop terminates when a counter variable reaches its limit (a program may ask the user for this limit prior to the loop, in which it is a **list headed by size**)
* If the user does not know how many values are to be entered, another approach is to write an **ask before iterating** loop, in which the program asks at the end of each iteration whether or not another iteration is required
* A variable that indicates that an event has taken place is called a **flag**; the Boolean expression may test the value of the flag to see if it indicates that the loop must terminate (the loop **exits on a flag condition**)

Loop Usage: Nesting

* When a program nests one loop inside another, the inner loop must go through all of its iterations before a single iteration of the outer loop terminates
* Example: a program to calculate a total for seven days in each of four weeks:  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   int grandTotal = 0, weekTotal, dailyAmount, week, day;  
    
   for (week = 1; week <= 4; week++)  
   {  
   weekTotal = 0;  
    
   cout << "Enter amounts for week " << week << endl;  
    
   for (day = 1; day <= 7; day++)  
   {  
   cout << "Day #" << day << ": ";  
   cin >> dailyAmount;  
   weekTotal += dailyAmount;  
   }  
    
   cout << "Total for Week #" << week << ": " << weekTotal  
   << endl << endl; // a blank line between weeks  
    
   grandTotal = grandTotal + weekTotal;  
   }  
    
   cout << "Grand Total: " << grandTotal << endl;  
    
   return 0;  
  }

Loop Errors

* A common mistake that occurs when writing loops is an **off-by-one error**, in which the loop has one too few or one too many iterations
  + Counter starting at 0 vs. 1
  + Comparison being < vs. <=
* If a loop is not functioning properly, it may be helpful to add extra temporary output statements to see the current values of crucial variables during different iterations (remove these from the final version of the program)
* Tracking the changes in the value of a variable is called **tracing** the variable