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