# Control Structures - Selection

*Using Control Structures:*

- Algorithm: A procedure for solving a problem in terms of
  - the actions to execute
  - the order in which the actions will execute
- Pseudocode: "fake" code
  - describes the action statments in English
  - helps a programmer "think out" the problem and solution but does not execute
- Flow of Control/Execution: implemented with three basic types of structures
  - Sequential: default mode; Executed line by line, one right after the other
  - Selection: decisions, branching; When there are 2 or more alternatives. Three types:
    - if
    - if...else
    - switch
  - Repetition: used for looping, or repeating a section of code multiple times. Three types:
    - while
    - do...while
    - for
- True and False
  - Statements that are executed are dependent on certain conditions that are evaluated either true or false.
  - Condition represented by logical (Boolean) expression - can be true or false
  - Condition met if evaluates to true
  - **Key:** Any C++ expression taht evaluates to a value can be interpreted as a true/false condition.
    - An expression that evaluates to 0 is **false**.
    - An expression that evaluates to a non-zero value is **true**.

---

# Relational Operators

*Using Relational Operators:*

- Relational operators:
  - Allow comparisons
  - Require two operands (binary)
  - Return 1 (or nonzero integer) if expression true, otherwise 0 false
- Comparing values of different data types may produce unpredictable results
  - For example: 6 < '5' (compares integer 6 with character '5')

*Relational Operator Examples:*

- Each of the following returns a **true** or **false**.
- Commonly used as test expressions in selection and repetition statements.

```
==  equal to
x == y

!=  not equal to
x != y

<  less than
x < y

<=  less than or equal to
x <= y

>  greater than
x > y

>=  greater than or equal to
x >= y

***BE CAREFUL!
'a' < 'A' evaluates to false!
```

*****Under the ASCII Collating Sequence: numbers < ucase letters < lcase letters**
ASCII Chart

*Comparing string Types:*

- Relational operators can be applied to strings
- Strings: compared character by character, beginning with first character
- Comparison continues until either mismatch found, or all characters found to be equal
- If two strings of different lengths compared, and comparison equal to last character of shorter string, shorter string is evaluated as "less" than larger string

*string Comparison Examples:*

```
Declarations:
string str1 = "Hello";
string str2 = "Hi";
string str3 = "Ann";
string str4 = "Bill";

str1 < str2 //true: second character 'e' less than 'i'
str4 < str3 //false: first character 'B' greater than 'A'
str3 >= "An" //true: characters are equal, but str3 is longer than "An"
```

*****Caution!*****
Sometimes, programs are required to compare numbers and/or variable values. For example:

```
int x = 5; // integers have no precision issues
if (x==5)
    cout << "x equals 5" << endl;
else
    cout << "x does not equal 5" << endl;

Program prints: x equals 5.
```

However, when using floating point numbers, unexpected results may occur, if the two numbers have very close values. For example:

```
float myFloat1 = 1.345f;
float myFloat2 = 1.123f;
float fTotal = myFloat1 + myFloat2; // should be 2.468

if (fTotal == 2.468)
    cout << "fTotal is 2.468";
else
    cout << "fTotal is not 2.468";

Prints: fTotal is not 2.468
```

This result is due to rounding errors. The variable fTotal is stored as 2.4679999, which is not 2.468!

**For this reason, comparison operators >, >=, <, and <= may produce incorrect results when comparing floating point numbers. Solution: Code to allow for some tolerance.**

```
if (x > 5.01) && (x < 5.03) cout << "Equal";
```

---

# Logical (Boolean) Operators

*Using Logical (Boolean) Operators:*

- Logical (Boolean) operators take logical values as operands, and yield logical values as results
- Three logical (Boolean) operators:
  - ! (not)
  - && (and)
  - || (or)
- ! is unary; && and || are binary operators
- ! in front of logical expression reverses value
- true and false are keywords
- true (nonzero)
- false (0)

*Logical (Boolean) Operator Examples:*

```
!true //evaluates to false
!(5 >= 8)  //evaluates to true: 5 >= 8 (false), !(false) is true

( numStudents >= 30 && classAverage >= 70 )
//there are at least 30 students and the classAverage is at least 70
```

```
( numStudents >= 30 && !(classAverage < 70) )
//evaluates to same as above
```

---

# Logical (Boolean) Expressions

To clarify the differences between using the logical (boolean) operators AND &&, OR ||, and bitwise operators AND &, OR |:

First of all, let me preface any comments by saying when using && or &, and || or |, ultimately the results are the SAME. When using && or & operators, both operands must be true, otherwise, false. When using || or | operators, both operands must be false, otherwise, true. The difference lies in the evaluation PROCESS.

**Short-circuit evaluation:** evaluation of a logical expression stops as soon as the value of the expression is known. When the && operator is used to compare a logical expression, the compiler will stop the evaluation upon the first false condition, unlike the & operator which will continue to evaluate the ensuing expression(s) being compared. The same holds true for the operators || and |, when evaluating true conditions.

With respect to logical vs. bitwise operators, there is a difference between using the logical "and" (&&), rather than the bitwise "and" (&). And, yes, there is a very important difference--and, no "one size fits all" solution:

When the logical (&&) operator is used to compare a logical expression, the compiler will stop the evaluation upon the first false condition, unlike the & operator which will continue to evaluate the ensuing expression(s) being compared.

```
if(++myVal % 2 == 0 && ++count < limit)
   // Do something
```

If the left hand operand is evaluated to be false **(++myVal % 2 == 0)**, the compiler will stop and not evaluate the right-hand operand **(++count < limit)**. More importantly, if myVal is not an even number, count will NOT increment.

This may be fine, IF count doesn't need to increment every time. Though, if it does, use the bitwise "and" (&) operator to evaluate both expressions. As the following example:

```
if(++myVal % 2 == 0 & ++count < limit)
   // Do something
```

Again, fundamentally, the logical result is the same (i.e., a truth table) for both & and &&: "Do something" only will be executed if BOTH expressions are true. The benefit is that the bitwise "and" (&) operator will allow count to increment.

On the other hand, you may not want the right hand expression to be evaluated (that is, use && not &). Example:

```
If(count > 0 && total/count > 5)
```

Here, it's easy to see that one would NOT want a compile or run-time error if the value of count were 0!

Lastly, the same "truth table" logic holds true for logical "or" (||) vs. the bitwise "or" (|). That is, the || operator is like && in that the right hand operand is NOT evaluated if the left hand operand is TRUE. You would need to use | if you wanted BOTH expressions to be evaluated.

*Using bool Data Type and Logical (Boolean) Expressions:*

- bool data type uses logical (Boolean) values true and false
- bool, true, and false are reserved words
- Identifier true contains value 1
- Identifier false contains value 0
- **Be careful** when using logical expressions:
  - 0 <= x <= 10 (**Incorrect**)
    Always evaluates to true because 0 <= x evaluates to either 0 or 1, and 0 <= 10 is true and 1 <= 10 is true
  - 0 <= x && x <= 10 (**Correct**)

---

# Operator Precedence and Associativity

Two operator characteristics determine how operands group with operators: precedence and associativity. Precedence is the priority for grouping different types of operators with their operands. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses.

Also, operators follow a strict precedence, which defines the evaluation order of expressions containing these operators.

Operators associate with either the expression on their left or the expression on their right; this is called "associativity." The following table shows the precedence and associativity of C++ operators (from highest to lowest precedence). Operators in the same segment of the table have equal precedence and are evaluated left to right in an expression unless explicitly forced by parentheses.
Operator Precedence and Associativity 1
And...
Operator Precedence and Associativity 2

*Mnemonic: "Math Relates Logically"*

- Generally, the following 3 types of operators are evaluated in the following order: 1. Mathematical, 2. Relational, 3. Logical
- Operators with same level of precedence evaluated from left to right
- **Unary operators have higher order of precedence than binary operators: e.g., logical not (!), unary minus (-), unary plus (+)**
- Associativity is left to right for most mathematical, relational, and logical operators
- **Unary operators are right associative: e.g., logical not (!), unary minus (-), unary plus (+)--not including some postfix operators**
- **Parentheses can be used to override precedence**

*Order of Precedence Examples (for complete listing, see links above):*

| Operators | Precedence |
|---|---|
| !, +, – (unary operators) | first |
| *, /, % | second |
| +, – | third |
| <, <=, >=, > | fourth |
| ==, != | fifth |
| && | sixth |
| \|\| | seventh |
| = (assignment operator) | last |

---

# One-Way (if) Selection

*Using One-Way (if) Selection:*

- Syntax of one-way (if) selection:

```
if (expression)
  statement1;
  statement2;
```

- if is reserved word
- expression must evaluate to true (nonzero) or false (0)
- statement1 executed if value of expression true
- statement1 bypassed if value false; program execution moves to statement2
- statement2 is executed regardless

```
if (grade >= 90)
  cout << "Student has an A";
```

---

# Two-Way (if...else) Selection

*Using Two-Way (if...else) Selection:*

- Syntax of two-way (if...else) selection:

```
if (expression)
  statement1;
else
  statement2;
```

- If expression true, statement1 executed, otherwise statement2 executed
- else is reserved word

---

# Compound (Block) Statement

*Using Compound (Block) Statement:*

- Syntax of compound statement (or block of statements):

```
{
 statement1;
 statement2;
 .
 .
 .
 statementn;
}
```

- Compound statement treated as single statement

*Compound Statement Example:*

```
if (age > 18)
{
  cout << "Eligible to vote." << endl;
  cout << "No longer a minor." << endl;
}
else
{
  cout << "Not eligible to vote." << endl;
  cout << "Still a minor." << endl;
}
```

---

# Nested if and if...else Statements

*Using Nested if and if...else Statements:*

- Nesting: one control statement in another
- Use consistent layout for readability
- else always belongs to closest if without an else
- Syntax of nested if...else statements:

```
{
if (condition1)
  statement1;
else if (condition2)
  statement2;
. . .
else if (condition-n)
  statement-n;
else
  statement-z;
}
```

**Example 1:**

```
{
if (x < 0)
  y = -1;
else if (x == 0)
  y = 0;
else       //(x > 0)
  y = 1;
}
```

**Example 2:**

```
{
if (x < 0)
```

```
  y = -1;
if (x == 0)
  y = 0;
if (x > 0)
   y = 1;
}
```

- Example 2 not recommended: a) not clear that only one statement will be executed for given value of x; b) inefficient since all conditions are always tested!

---

# Preventing Input Failure Using if Statement

*Using if Statement to Prevent Input Failure:*

- When input stream enters fail state:
  - All subsequent input statements associated with stream ignored
  - Program continues to execute--though, may produce erroneous results
- Can use if statements to check status of input stream
- If stream enters fail state, include instructions to halt program execution

---

# Conditional Operator (?:)

*Using if Statement to Prevent Input Failure:*

- "Short-hand" version of if...else statement
- Conditional operator (?:) takes three arguments (ternary operator)
- Syntax for using conditional operator:

  ```
  test_expression ? true_expression : false_expression
  ```

- If test_expression true, true_expression is invoked; otherwise, false_expression is invoked

  ```
  cout << (x > y ? "x is greater than y" : "x is less than or equal to y");
  ```

  //Equivalent to

  ```
  if (x > y)
    cout << "x is greater than y";
  else
    cout << "x is less than or equal to y");
  ```

---

# switch Structures

*Using switch Structures:*

- Convenient when there are multiple cases to choose from.
- switch structure: alternate to if...else
- switch expression evaluated first (must be integral value) then compared to values in case labels. If a match is found, execution jumps to that case label.
- Value of expression determines corresponding action
- One or more statements may follow case label
- Braces **not** required to execute multiple statements (different to other structures)
- Required statements: switch, case
- Optional statements: break, default
- break statement may or may not appear after each statement
  - To execute only the code that matched the case, end with a break statement
  - Otherwise execution will continue to the next case
- switch, case, break, and default are reserved words
- When value of expression matched against case value,
  statements execute until break statement found, or end of switch structure reached
- If value of expression does not match any case values, statements following default label execute
- If no default label, and no match, entire switch statement skipped
- break statement invokes immediate exit from switch structure
- switch statement can evaluate expression for multiple values
- Syntax for using switch statement:

  ```
  Note: series of case labels and optional default case
  Example 1:
  ```

```
switch(x)
   {
      case x1:
          statements1;
          break;

      case x2:
          statements2;
          break;

      case x3:
          statements3;
          break;

      default:
          statements4;
          break;
   }
```

**Example 2:**

```
switch (integralTestExpression)
{
case value1:  // taken if integralTestExpression == value1
statements...;
break;          // necessary to exit switch

case value2:
case value3:  // taken if integralTestExpression == value2 or == value3
statements...;
break;

default:     // taken if integralTestExpression matches no other cases
statements...;
break;
}
```

---