Loops – Part II

Additional Control Structures

The switch Statement

- The switch statement
  - is a control structure for multi-way branches
  - similar to nested if statements
  - requires a switch expression – the value of this expression determines which switch label is selected. It cannot be a floating point or string expression

- SyntaxTemplate: SwitchStatement
  switch (IntegralOrEnumExpression)
  |
  |
  |
  |
  { |
  |
  SwitchLabel ... Statement
  |
  |
  |
  |
  ;
  |
  |
  |}

CPE 112
The switch Statement

- IntegralOrEnumExpression is an expression of integral (integer) type (char, short, int, long, bool) or enum type (covered in chapter 10)
- SyntaxTemplate: SwitchLabel
  - case ConstantExpression :
  - default :
- The SwitchLabel is optional
- The SwitchLabel in front of a statement is either a case label or a default label
  - In a case label, ConstantExpression is an integral or enum expression whose operands must be literal or named constants (NO VARIABLES)
- There can be 0 or more statements associated with a SwitchLabel

The switch Statement

- Flow of control through a switch statement is as follows:
  - The switch expression is evaluated
  - If the value* matches one of the values in a case label,
    - control branches to the statement(s) following the case label
    - From that point, control proceeds sequentially until either a break statement or the end of the switch statement is encountered
  - if the value* does not match one of the values in a case label
    - If there is a default label, control branches to the statement(s) following the default label
    - If there is no default label, all statements in the switch are skipped
- Look at examples on pages 305-307
  - *value of the evaluated SwitchExpression
Switch Example

```cpp
switch (month) // month is an int variable previously declared/defined
{
    // JAN, FEB, MAR, OCT, NOV, DEC are previously defined integer constants
    case JAN :
    case FEB :
    case MAR : // following cout prints if month equals JAN, FEB or MAR
        cout << "One of the first three months\n";
        break; // terminates further execution of the switch
    // from a match with JAN, FEB, or MAR
    case OCT :
    case NOV :
    case DEC : // following cout prints if month equals OCT, NOV or DEC
        cout << "One of the last three months\n";
        break; // terminates further execution of the switch
    // from a match with OCT, NOV or DEC
    default : // executed if no case label equals month
        cout << "No condition matched in the switch statement\n";
}
```

Output of the switch statement is one of the three cout statements only.

See program Ch_07_01.cpp for an example of this program

Switch Example (continued)

Equivalent if-then-else-if control structure for the previous example

```cpp
// JAN, FEB, MAR, OCT, NOV, DEC are all defined integer constants
if (month == JAN || month == FEB || month == MAR)
{
    // prints if month is equal to JAN, FEB or MAR
    cout << "One of the first three months\n";
}
else if (month == OCT || month == NOV || month == DEC)
{
    // prints if month is equal to OCT, NOV or DEC
    cout << "One of the last three months\n";
}
else
    cout << "No condition matched in the if statement\n";
```

See program Ch_07_01.cpp for an example of this program
Switch Example - Omission of break

```c
switch (month)
{
    case JAN : // note all statements following a case label are
    case FEB : // executed until a break is encountered or the
    case MAR : // end of the switch statement is reached
        // prints if month is equal to JAN, FEB or MAR
        cout << "One of first three months\n";
    case OCT :
    case NOV :
    case DEC : // prints if month is equal to OCT, NOV or DEC
        cout << "One of the last three months\n";
    default : // executed if no case label equals month
        cout << "No condition matched in the switch statement\n";
}
// The output here is all three cout statements if month is JAN, FEB or MAR; the last 2
// cout statements if month is OCT, NOV or DEC; The default cout if month does not
// match any of the case labels
// JAN, FEB, MAR, OCT, NOV, DEC are all defined as integer constants
See program Ch_07_02.cpp for an example of this program
```

The do-while Statement

- Looping control structure where the loop condition is tested at the end of the loop.
- Loop body executes **at least once** - KEY POINT
- **Syntax Template: DoWhile**
  ```c
do
    Statement (either a single statement or a block)
while (Expression);
```
- Note that the do-while ends in a semicolon
- The do-while statement means execute the statements between do and while as long as Expression still has the value of true at the end of the loop
- Look at examples in book on page 311
  See programs Ch_07_03(a,b,c).cpp for examples of the do-while
Example do-while

- Compare `while` to `do-while` for reading lines from a file

```cpp
getline(InFile,line); // Priming read
while (InFile) // with a priming read, the while loop handles
    // files with 0 or more lines
{
    cout << line << endl;
    getline (InFile,line);
}
```

// do-while loops may or may not use a priming read; however,
// not using one results in an additional test in the loop body
// do-while loops execute at least once, so they
// can correctly handle files with 1 or more lines only
```cpp
getline(InFile, line); // priming read for proper handling
do{
    cout << line << endl;
    getline(InFile, line);
} while (InFile);
```

**Note:** The do-while can be made to handle files of zero lines, but it would be
better to use a while statement if files of zero lines are expected

More on do-while Loops

- do-while loops do not require a priming read, but test the data
twice. They work well for entering information that must belong
to a valid range.

Example from page 311

```cpp
// sample of code prompting for a valid age
do
{
    cout << "Enter your age: ";
    cin >> age;
    if (age <= 0)
        cout << "Your age must be positive.\n";
} while (age <= 0);
```
The **for** Statement

- Recall **SyntaxTemplate** from a previous lecture:
  ```c
  for (InitStatement Expression1 ; Expression2 )
  Statement
  ```
- The **InitStatement** initializes a **loop control variable** and can be 1 of the following:
  - **The null statement** - which is just a semicolon
  - **a declaration statement** - which ends in a semicolon
  - **an expression statement** that ends in a semicolon
- **Expression1 and Expression 2 are optional**
  - Expression1 tests the loop control variable
  - Expression2 increments, decrements or changes in some manner
    the loop control variable
- The statement **for ( ; ; )** is valid, though it is an infinite loop

The **break** and **continue** Statements

- The **break** statement
  - Causes **immediate exit** from the **INNERMOST** switch, while, **do-while** or **for** statement in which it appears
  - A **break** statement in a loop nested in another loop causes control to **exit the inner loop, but not the outer loop**
  - Use **break** statements within loops as a **last resort only**
    - If they make the code more readable, they might be useful
    - If a loop has multiple exit points, they are helpful
    - **Should be clearly commented as to the expected action**
The `continue` Statement

- The `continue` statement is valid in loops only
- It terminates the current loop ITERATION - not the entire loop
- It causes an immediate branch to the bottom of the loop - skipping the rest of the statements in the loop body

Example showing how empty lines in a file can be skipped

```c++
getline(InData, line); // priming read obtains first line in the file
while (InData) // continue until the end of the file is reached
{
    if (line == "") // if the line is empty don’t process the line
    {
        getline(InData, line); // read next line in the file
        continue; // control branches to the bottom of the loop
    }
    cout << line; // or any other processing statements
    getline(InData, line); // read the next line in the file
} // bottom of the loop. When this point is reached the
// while expression is tested
```

See programs Ch_07_04.cpp and Ch_07_05.cpp for examples of using the break and continue statements
Guidelines for Choosing a Looping Statement

- If the loop is a **simple count-controlled loop**, the `for` statement is the best

- If the loop is **event-controlled** and whose body will **execute at least once**, use a `do-while` loop

- If the loop is **event-controlled** and nothing is known about the first execution, use a `while` or `for` statement

- **When in doubt** use a `while` statement

- **Infinite loops** with `break` statements sometimes clarify code, but often reflect undisciplined loop design. Use this type of loop as a last resort only