Lecture 03: Control Statements

In this lecture, we shall examine how control flow is managed in C++. In particular, we will discuss:

- Boolean Expressions
- The If-Statement
- The Switch-Statement
- The Break-Statement and Continue-Statement
- The While-Statement
- The For-Statement
- The Do-while-Statement
- The Goto-Statement
- A Programming Example
- Summary and Concluding Remarks

3.1 Boolean Expressions

In traditional C, there is no distinction between an arithmetic expression (see lecture 1) and a Boolean expression, except for the context of usage (for Boolean expressions 0 means ***false*** and a positive integer value means ***true)***. However, in the interest of clarity, we shall assume this distinction. A Boolean expression is an expression (condition) that evaluates to true or false. C++ now has the data type **bool**, but it still supports the traditional convention. Boolean expressions are formed by using Boolean operators. The required syntax is shown in figure 3-1:

Figure 3-1: Syntax for Boolean Expression

```
BooleanExp ::= [!] <BooleanExp> |
[!] <BooleanVariable> |
<Variable> <BoolOperator> <Variable> |
<Variable> <BoolOperator> <BooleanExp> |
<BooleanExp> <BoolOperator> <Variable> |
<BooleanExp> <BoolOperator> < BooleanExp > |
<BooleanExp> <BoolOperator> < Literal >  |
<Variable> <BoolOperator> < Literal >

BoolOperator ::=  < | <= | == | != | > | >= | || | && | ! | ?:
```

Meanings of Boolean operators are as follows:

Operator	Meaning
<	Less then
<=	Less than or equal to
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
||	Logical OR
&&	Logical AND
!	Logical NOT
?:	Conditional operator; equivalent to a simple if-then-else scenario

When constructing Boolean expressions, be sure to observe the operator precedence schedule that was provided in lecture 1 (figure 1-18). A simplified extraction from figure 1-18 is shown in figure 3-2.

Figure 3-2: C++ Operator Precedence Schedule for Boolean Operators

Precedence	Operator	Description
1	(. . .)	Parenthesized expression
2	!	Logical NOT
3	< <= > >=	Less-than, less-than-or-equal-to, greater-than, and greater-than-or-equal-to
4	== !=	Equal and Not-equal operators
5	&&	Logical AND
6	||	Logical OR
7	?:	Ternary conditional

3.1 Boolean Expressions (continued)

The conditional operator is a ternary operator which, though optional, is quite convenient. The syntax for usage is as follows:

```
<Condition> ?: <Expression1> : <Expression2>;
```

This is equivalent to the following:

```
if <Condition>
   <Expression1>;
else <Expression2>;
```

The condition specified is a Boolean expression, and expression specified is any valid statement. In the interest of clarity, this course recommends that you use the if-then-else structure (explained in the upcoming section) in favor over the conditional operator. However, observe that use of the conditional operator allows for more compact code.

3.2 The If-Statement

The **if-statement** is a selection structure that facilitates choices based on existing circumstances. The statement has the following syntax (shown in figure 3-3):

Figure 3-3: Syntax for If-Statement

```
If_Statement ::=
if (<BooleanExp>)
   <Statement>; | <Compound_Statement>
[else <Statement2>; | <Compound_Statement>]

Compound_Statement ::=
{ <Statement>; [* <Statement>; *] }
```

Following are some relevant guidelines:

1. The Boolean expression immediately following the **if**-keyword is called the **if-condition**, and must conform to the syntax clarified in the previous section. **Statement1** and **statement2** may be any valid statement, including another **if-statement** respectively. Thus, we could have nested **if-statements** to several levels. Although C++ allows for 256 levels of nesting, it is prudent to restrict this to eight or less.

2. The expression specified (after the keyword **if**) may be any valid expression (Boolean or arithmetic) that evaluates to zero or an integer greater than zero, according to the earlier established convention (section 3.1). However, in the interest of clarity, try to keep them as Boolean expressions.

3.2 The If-Statement (continued)

3. If you desire to specify more than one statement under a given **if-condition**, you must include a *compound statement* (also called a *block*). The compound statement commences with a left curly brace ({) and ends with a right curly brace (}). The block typically contains multiple statements, but could also contain zero or one statement (in which case the block would be redundant). When these braces are used, statement termination of the block via the semicolon (;) is **not** required (the right brace takes the place of the semicolon).

   Thus:

```
if (<Condition >)
{
   <Statement1>;
     ….
   <StatementN> }
[else <StatementB>]
```

4. Ambiguity (i.e. potentially misleading statement) can be avoided by using compound statements.

5. If nested loops are used, indent to improve the readability of your code.

Example 3-1:

```
/* The following is an example of ambiguity: which condition is the else related to? Typically, the
compiler will associate it with the latest if-condition. But based on the text of the message, this would
not coincide with the intent of the programmer. */

double studentGPA;
// . . .
if (StudentGPA <= 4.0)
   if (StudentGPA < 2.5)
      cout << "Your performance is below the required level; you must withdraw.\n";
else cout << "Invalid GPA value.\n";

// This problem can be corrected by tightening the logic with the use of a block.
```

Example 3-2:

```
// Correction of the ambiguity in example1.
double studentGPA;
// . . .
if (studentGPA <= 4.0)
{
  if (StudentGPA < 2.5)
     cout << "Your performance is below the required level; you must withdraw.\n";
  else cout << "You are doing fine.\n";
}
else cout << "Invalid GPA value.\n";

// Note: The compiler treats the block as one statement.
```

3.2 The If-Statement (continued)

Example 3-3:

```
int year, month, day;
bool leapYear;
…
//…
if (year % 400) == 0) || ((year % 4) == 0) && (year % 100) != 0))
   leapYear = true;
else leapYear = false;
//…
if (leapYear && month ==2)
{
   cout << “This is a leap year and it is February.\n”;
   cout << “There will be 29 days in the month. My best friend will have a birthday – “;
   cout << “ a rare event for her.\n”;
}
…
```

Example 3-4:

```
double studentGPA;
// . . .
if (studentGPA <= 2.5)
   cout << “Your performance is low; you must withdraw.\n”;
else if (studentGPA <= 3.5)
          cout << “You are doing fairly well, but there is room for improvement.\n”;
       else if (studentGPA <= 4.0)
                 cout << “Excellent!\n”;
               else cout << “Invalid GPA value.\n”;
```

3.3 The Switch Statement

The **switch-statement** is the selection structure that is implemented in many languages as the case-statement; its syntax is shown in figure 3-4:

Figure 3-4: Syntax for the Switch-Statement

```
Switch_Statement ::=
switch (<Expression>)
{
case <ConstantExpression1>: <Statement>; | <CompoundStatement>
…
case <ConstantExpressionN>: <Statement>; | <CompoundStatement>
[default: <Statement>; | <CompoundStatement>
}

Compound_Statement ::=
{ <Statement>; [* <Statement>; *] }
```

In using the **switch-statement**, be mindful of the following guidelines:

1. The **switch-statement** is used when an expression could have any of a finite set of values and the action required varies with each value.

2. The order in which the constant expressions are listed is unimportant.

3. Each constant expression may lead to a simple statement or a compound statement.

4. To avoid a fall-through case, it is good practice to specify a **break-statement** after each case.

5. The switch expression must evaluate to an integer or a character.

6. Nesting is facilitated — the statements inside a case option may be any valid statement, including another **switch-statement**.

Example 3-5: The **switch-statement** is often used for constructing user menus as in the following illustration:

```
switch (option)
{
case 1:      {lineDraw (); break;}
case 2:      {rectangleDraw ();break;}
case 3:      {squareDraw (); break;}
case 4:      {triangleDraw (); break;}
default :    cout << "Invalid Option";
}
```

3.4 Break-Statement and Continue-Statement

Two of the simplest statements in C++ are the **break-statement** and the **continue-statement**. The required syntax in either case is simply the keyword, followed by a semicolon.

Figure 3-5: Syntax for the break-statement and the continue-statement

```
Break_Statement ::=   break;
Continue_Statement ::= continue;
```

The **break-statement** causes an immediate exit from a loop. The **continue-statement** (not applicable to the **switch-statement**) causes the next iteration of the loop to begin. It is used within iterative loops (sections 3.5 – 3.7).

3.5 The While-Statement

The **while-statement** is used for constructing a while loop. It is iterative. The syntax for usage is as shown in figure 3-6:

Figure 3-6: Syntax for the while-statement

```
while-statement ::=
while   (<BooleanExp>)
        <Statement>; | <CompoundStatement

Compound_Statement ::=
{ <Statement>; [* <Statement>; *] }
```

The following guidelines apply to the use of the **while-statement**:

1. The expression specified may be an arithmetic expression or a Boolean expression (in the interest of clarity, the latter is recommended).

2. The statement specified could be any valid statement, including another **while statement**, thus leading to nested **while-statements**.

3. Whenever it is required to have more than one statement within the while-loop, a compound statement is applicable. If a simple statement is specified, it may be prudent to place the entire **while-statement** on a single line. If a compound statement is required in the while-loop, then you should use indentation to improve readability.

4. If nested loops are used, indent to improve the readability of the program.

5. Your loops must have an exit strategy; otherwise it is an infinite loop. The exit strategy is a statement that will eventually cause the while-condition to become false.

3.5 The While-Statement (continued)

Example 3-6:

```
bool exitFlag = false;
while (!exitFlag)
{
        // Several statements one of which must change exitFlag to true
}
// End-while; this loop will iterate until exitFlag becomes true
```

Example 3-7:

```
int count = 1, limit = 12; // You would set limit to any desired value
while (count <= limit)
{
        // Several statements
        count++; // the exit strategy
}       // End-while; this loop will iterate limit times
```

Example 3-8:

```
int count = 1, limit = 12;
while (count <= limit) count ++;
// a time delay loop
```

is equivalent to:

```
int count = 1, limit = 12;
while (count++ <= limit);
// a time delay loop
```

3.6 The For-Statement

The **for-statement** is the most flexible (and widely used) iterative statement in C^{++}. It has the following syntax figure 3-7):

Figure 3-7: Syntax for the for-statement

```
for-statement ::=
for     (<Expression1>); <BooleanExp2>; <Expression3>)
        <Statement>; | <CompoundStatement

Compound_Statement ::=
{ <Statement>; [* <Statement>; *] }
```

Following are some related guidelines for this statement:

1. **Expression1** is the initialization expression (typically an assignment); **BooleanExp2** is the (typically Boolean) condition which determines whether the loop iterates; **Expression3** is the increment expression (typically an assignment).

2. The **for-statement** can be replaced by a while-statement and vice versa. The equivalent while-structure for the for-structure of figure 3-7 is shown in figure 3-8:

Figure 3-8: Equivalent while-structure for a for-structure

```
<Expression1>;
while <BooleanExp2>
{
  <Statement>; // followed by possibly other statements
  <Expression3>;
}
```

3. Three (sets of) arguments are specified within the parentheses: initialization expression(s), exit expression(s) and increment expression(s). For each category, more than one expression may be specified (separated via use of the comma). Expressions at each category are parallel, are evaluated left to right and should be of the same type.

4. Whenever it is required to have more than one statement within the for-loop, a compound statement is applicable. If a simple statement is specified, it may be prudent to place the entire **for-statement** on a single line. If a compound statement is required in the for-loop, then you should use indentation to improve readability.

3.6 The For-Statement (continued)

Example 3-9: The code of example 3-7 may be reconstructed using a **for-statement:**

```
int count, limit = 12;
for (count = 1;  count <= limit; count++)
{
        // several statements
} // End-for
```

Equivalent to:

```
int count = 1, limit = 12;
while (count <= limit)
{
        // several statements
        count++; // the exit strategy
}       // End-while;
```

Example 3-10: The code of example 3-8 may be reconstructed using a **for-statement:**

```
int count, limit = 12;
for (count = 1; count  <=limit; count++);
```

Is equivalent to the following:

```
int count = 1, limit = 12;
while (count++ <= limit);
// a time delay loop
```

```
int count = 1, limit = 12;
while (count <= limit) count ++;
// a time delay loop
```

Example 3-11: Figure 3-9 shows two versions of a string reversal function that reverses the incoming string parameter and returns it to the calling statement.

Figure 3-9: String Reversal Function

```
//A function to reverse a string
char* reverseS (char thisString [ ])
{
        int y, z;
        char x;
        for (y = 0, z = strlen(ThisString) –1; y < z; y++, z--)
        {
                x = thisString [y];
                thisString [y] = thisString [z];
                thisString [z] = x;
        }
      return thisString;
}
```

```
//Alternate function to reverse a string
char* reverseS(char thisString [ ])
{
   int y, z, sLength = strlen(thisString);
   char x;
   char revString[sLength] = " ";
   for (z = sLength -1; z >= 0; z--)
   {strcat (revString, thisString[z]);} // append to revString
   return revString;
} // End of Reverse Method

// We will discuss string manipulation functions in lecture 4
```

Example 3-12:

```
// An infinite for-loop
for (; ;);
```

3.7 The Do-Statement

In contrast to the **while-statement** and the **for-statement**, the **do-statement** sets up a do loop where the condition is tested at the end of the loop. The statement is the C^{++} implementation of the repeat-until iterative structure. The syntax for the usage is shown in figure 3-10:

Figure 3-10: Syntax for the do-statement

```
Do_Statement ::=
do
  <Statement>; | <CompoundStatement
while (<BooleanExpr);

Compound_Statement ::=
{ <Statement>; [* <Statement>; *] }
```

Following are some guidelines related to using the **do-statement**:

1. The expression specified may be an arithmetic expression or preferably, a Boolean expression.
2. The statement specified may be any valid statement, including another **do-statement**, thus leading to nested **do-statements**.
3. Where it is required to have more than one statement, within the loop a compound statement is used.
4. If nested loops are used, remember to indent to improve readability.
5. As for all iterative loops, there must be an exit strategy, to avoid having an indefinite loop.

Example 3-13: The code of example 3-7 may be reconstructed using a **do-statement:**

```
int count, limit = 12;
do {
        // several statements
        count++;
    } while (count <= limit);
```

Is equivalent to:

```
int count, limit = 12;
while (count <= limit)
{
  // several statements
  count++;
}
```

Example 3-14: The code of example 3-8 may be reconstructed using a **do-statement:**

```
int count, limit = 12;
do count++;
while (count <=limit);
// a time delay loop
```

Is equivalent to:

```
int count, limit = 12;
while (count <= limit) count ++;
// a time delay loop
```

```
int count, limit = 12;
while (count++ <= limit);
// a time delay loop
```

3.8 The Goto–Statement

Goto statements were very prevalent in the early days of programming. In many scenarios, they were unwisely used and therefore rendered programs virtually unreadable and difficult to maintain. However, with discipline, they could be useful in providing clarity rather than causing confusion.

A **goto-statement** can always be replaced by an iterative loop. A **goto-statement** by necessity, works with a label. To set up a label in C++ simply specify the label-name, followed by a colon. The goto-statement simply directs the compiler to branch to a particular label. The syntax for the statement is shown in figure 3-11.

Figure 3-11: Syntax for the goto-statement

```
Label_Statement ::=
<LabelName>:

Goto_Statement ::=
goto <LabelName>;
```

Here are three suggested rules for goto-statements:
1. Do not branch from one function to another.
2. Do not branch across program blocks.
3. Use the **goto-statement** to control an iterative loop.

Example 3-15: The code of example 3-7 may be reconstructed using a **goto-statement:**

```
int count, limit = 12;
Again:
   //several statements
  // count ++;
  if (count++ <= limit) goto Again;
```

Is equivalent to:

```
int count, limit = 12;
while (count <= limit)
{
  // Several statements
   count++;
}
```

Example 3-16: The code of example 3-8 may be reconstructed using a **goto-statement:**

```
int count, limit = 12;
Again:
  if (count++ <= limit) goto Again;
// a time delay loop
```

Is equivalent to:

```
int count, limit = 12;
while (count <= limit)
count ++;
// a time delay loop
```

```
int count, limit = 12;
while (count++ <= limit);
// a time delay loop
```

3.9 A Programming Example

Let us write a program that will accept a student's test score, determine the equivalent grade and quality point according to the following table:

Score	Grade	Quality Points
90-100	A	4.0
85-89	A-	3.67
80-84	B+	3.33
75-79	B	3.0
70-74	B-	2.67
65-69	C+	2.33
60-64	C	2.0
55-59	C-	1.67
50-54	D	1.0
0-49	F	0

The program will output on screen the equivalent letter grade and quality point for the score entered. The program will perform this operation until the user indicates to stop. The solution is shown in figure 3-12.

Figure 3-12a: The Grade Program Pseudo-code

```
Variables Required:

testScore:    N3,1
outGrade:     String A2
outQPoints:   N3,2
exitTime:     Boolean
more:         Character A1

Mainline:
START

        exitTime := False;
        While NOT exitTime do the following
                Prompt for and accept testScore;
                Case testScore is
                        90-100:       outGrade := A; outQPoints := 4;
                        85 - 89:      outGrade := A-; outQPoints := 3.67;
                           :
                        0-49:         outGrade := F; outQPoints := 0;
                End-Case;

                Display outGrade and outQpoints;
                Determine whether user wishes to enter more;
                If not, exitTime := True;
        End-While;

STOP
```

Figure 3-12b: The Grade Program C++ Code

```
// Program Name: MyGrade
// Description: The Grade Program

#include <iostream.h>
#include <stdio.h>
using namespace std;

// Variables
char    outGrade [3] = " ";    bool    exitTime;
float   testScore, outQpoints;        char    more;

// Mainline
void    main ()
{
        exitTime  = false;
        while (!exitTime)
        {
                cout<< "\nGrade Evaluation Exercise \n";
                cout<< endl;
                cout << "Enter Student Score:   ";
                cin>> testScore; gets();

                /* convert test score */
                if (testScore < 49.5)
                {outGrade[0] = 'F';     outGrade[1] = ' ';       outQpoints = 0; }
                else if ( testScore < 54.5)
                {outGrade[0] = 'D';     outGrade[1] = ' ';       outQpoints = 1.0; }
                else if (testScore < 59.5)
                {outGrade[0]= 'C';      outGrade[1] = '-';       outQpoints = 1.67; }

                // … Similar code continues to grade A
                else
                {outGrade[0] = 'A';   outGrade[1] = ' ';         outQpoints = 4.0; }

                // Print the grade
                cout<< endl; cout<< "The grade is: " << outGrade[0] << outGrade[1];
                cout << " The quality points: " << outQpoints << endl;
                // printf ("The quality points:      ", "%4.2f \n", outOpoints);

                // Prompt for more
                cout << endl;
                cout<< "Press X to exit or any other letter to continue \n";
                more = getchar();
                if ((more == 'X') || (more == 'x'))        //User wishes to exit
                  exitTime = true;

        } // end while

} // End main
```

3.10 Summary and Concluding Remarks

It is time to summarize what was covered in this lecture:

- In C++, selection statements include the **if-statement** and the **switch-statement**. The former is applicable to any Boolean condition; the latter is applicable when a variable could have any of several specific values and for each value the action to be taken varies.
- Iterative statements include the **while-statement**, the **for-statement**, and the **do-statement**. The **while-statement** ensures execution of its enclosed statement(s) zero or more times. The **do-statement** ensures execution of its enclosed statement(s) one or more times. The **for-statement** is the most flexible, since it can be constructed to behave as either a **while-statement** or a **do-statement.** Moreover, it can also be used for controlling periodic behavior controlled by multiple increments or decrements with much less effort than it would take using either of the other two statements.
- C++ also supports the **goto-statement**. However, its use is not recommended.

Except for the **goto-statement**, you will find the C++ control statements virtually identical to those in Java. The next lecture discusses C++ functions.

3.11 Recommended Readings

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6th Edition. Boston: Addison-Wesley. See chapters 3 – 5.

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapters 4 & 5.

[Kernighan & Richie 1988] Kernighan, Brian W. & Dennis M. Richie. 1988. *The C Programming Language*. Boston: Prentice Hall. See chapter 3.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapter 2.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapter 3.