Lecture #5  
C# Repetition Structures

Introduction

In programming, a repetition structure (or a “loop”) is used to repeatedly process one or more instructions until some condition is met. The number of repetitions is based on a criterion defined in the loop structure, usually a Boolean expression. Repetition structures are constructed using the iteration statements, which cause embedded statements to be executed a number of times, subject to the termination criterion. Every time when a set of iteration is called for execution, the repetition structure is undergoing an “iteration”.

C# has three basic repetition structures: the for statement, the while statement, and the do..while statement. Repetition structures are also called “loops” because of their cyclic nature. In a nutshell, for and while loops are the two most commonly used repetition structures. The do..while loop can be considered a variation of while loop. There is another repetition structure, known as the foreach..in loop, which is used on a “collection” object to returns each element in order. It does not use integer index and is one-of-a-kind loop.

Programmers typically defines three basic components to implement a repetition structure: (1) an initial value, (2) a terminal value, and (3) increment/decrement. These three components are three individual expressions. Programmers may start with declaring a “counter” variable, such as i, and then assigns the initial value to it, as shown below.

```
int i = 5;
```

The same variable is then assigned a terminal value. This expression is typically a Boolean condition that is evaluated to decide whether the loop should repeat or not. In the following example, the terminal value is 15 because it is the largest possible value.

```
i <= 15;
```

The increment/decrement part defines how to reach the terminal value from the initial value. Programmers typically uses either increment or decrement operator to add or subtract a number to/from the counter variable. The following table illustrates the format, assuming i is the “counter” variable and n is a number (e.g. 5, 3.12, or -4).

<table>
<thead>
<tr>
<th>number</th>
<th>increment</th>
<th>decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>n</td>
<td>+=n</td>
<td>-=n</td>
</tr>
</tbody>
</table>

The following demonstrates how to set the “counter” variable to increase by 1.

```
i++;  
```

The following decrements by 1.

```
i--;  
```

According to the following, the value of i increments by 3.

```
i += 3;  
```

The following sets the value of i to decrement by 4.

```
i -= 4;  
```
The for Loop

The for loop is used to handle “counter controlled repetition”. When there is a need to repeatedly execute a block of statements, the for loop is a good choice. In a nutshell, a for statement consists of an iteration structure with three components: (1) initial value, (2) terminal value, and (3) increment or decrement. These three components define when to start, when to stop, and the number of iterations.

During an iteration, the for loop executes a section of statements enclosed by a pair of curly braces: { and }. Then, the for loop increment or decrement the “counter” variable and repeats the iteration until the “counter” variable reaches the terminal value. Finally, it exits the loop and continues with the rest of the code in the program. The following flowchart illustrates the logic.

In C#, for statements share the following syntax:

```csharp
for( initialValue; terminalValue; increment/decrement )
{
 Iteration statements
}
```

The `initialValue` specifies the initial value of the “counter” variable (which is also known as iterator). The for loop uses this variable to count the number of iterations. In the following example, the `i=1` expression specifies the initial value of the variable `i` to be 1. By the way, some programmers prefer declaring the “counter” variable outside the for loop’s condition block.

<table>
<thead>
<tr>
<th>Inside</th>
<th>Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>using System;</td>
<td>using System;</td>
</tr>
<tr>
<td>public class Example</td>
<td>public class Example</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>public static void Main()</td>
<td>public static void Main()</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>String str=&quot;&quot;;</td>
<td>String str=&quot;&quot;;</td>
</tr>
<tr>
<td>for (int i=1; i&lt;=5; i++)</td>
<td>int i;</td>
</tr>
<tr>
<td>{</td>
<td>for (i=1; i&lt;=5; i++)</td>
</tr>
<tr>
<td>str += i + &quot; &quot;;</td>
<td>str += i + &quot; &quot;;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>MessageBox.Show(str);</td>
<td>MessageBox.Show(str);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
Both of the above examples produce the following output. The difference is the scope of the variable \( i \). When \( i \) is declared inside the \textit{for} loop’s condition block, \( i \) can only be accessed by the \textit{for} loop.

In the above example, there is a statement (as shown below) inside the \textit{for} loop that could be confusing to students. The \textit{“+=”} operator is the “appending” operator because it is used to operate on a string literal, \( i + " \ " \), not a numerical value. By the way, the \textit{“+”} sign represents the “concatenation” operator which combine a value with a string literal to produce a new string literal. In C\#, any value “concatenated” with a string becomes a string.

\[
\text{str += i + " "};
\]

Unlike the above statement, \textit{“+=”} is the “increment assignment” operator in the following statement because 5 is a numerical value.

\[
i += 5;
\]

The \textbf{terminal value} defines when the loop should stop. It evaluates the current value of the iterator (the “counter” variable) every time when an iteration (a cycle) of the loop begins. The evaluation of terminal value occurs at the beginning of each loop. If the condition evaluates as true (e.g. \( i=3 \)), then the loop continues; otherwise the entire \textit{for} loop is forced to come to an end. In the above example, the value of the variable \( i \) can only be \{1, 2, 3, 4, 5\}, because 5 is the largest possible value defined by the terminal value expression, namely \( i\leq5 \).

There are two ways to define the terminal value--with and without the use of “equal to” operator. The following table illustrates the differences.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{increment}</td>
<td>\textbf{for(int i=1; i&lt;6; i++)}</td>
</tr>
<tr>
<td></td>
<td>\textbf{for(int i=1; i&lt;=5; i++)}</td>
</tr>
<tr>
<td>\textbf{decrement}</td>
<td>\textbf{for(int i=5; i&gt;0; i--)}</td>
</tr>
<tr>
<td></td>
<td>\textbf{for(int i=5; i&gt;=1; i--)}</td>
</tr>
</tbody>
</table>

The following example also produces 1 2 3 4 5 as output because the last possible value is 5 while the initial value is 1.

\[
\text{using System;}
\]
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        String str="";
        int i;
        for (i=1; i<6; i++)
        {
            str += i + " ";
        }
        MessageBox.Show(str);
    }
}

The increment/decrement part of expression defines how to increment (or decrement) the value of the iterator. When the increment operator (++ is used, the value increments by 1. When +=2 is used, the value increments by 2. When +=n is used, the increment is n, which is known as “skip counting”. Skip counting is to count forwards or backwards by a number other than 1. Refer to previous lecture for details about increment/decrement operator. The following demonstrates how to increment by 10.

using System;
using System.Windows.Forms;

class Example
{
    static void Main()
    {
        string str = "";
        for (double i = 0; i <= 0.9; i += 0.05)
        {
            str += i + " ";
        }
        MessageBox.Show(str);
    }
}

The output looks:

10 20 30 40 50 60 70 80 90 100

The “counter” variable does not have to be an int, it could be a double or float. Since the increment or decrement, in this case, will always be a value other than 1, the operator must be either “+=” or “-=”.

using System;
using System.Windows.Forms;

class Example
{
    static void Main()
    {
        string str = "";
        for (double i = 0; i <= 0.9; i += 0.05)
Although the increment operator (++ or +=) is frequently used to modify the value of the iterator, it is not the only option. Programmers can use the decrement operator (-=n) to count backwards through a loop. The following reverses the order of one of the previous examples. Its output is 5 4 3 2 1.

```csharp
using System;
using System.Windows.Forms;

class Example
{
    public static void Main()
    {
        String str="";
        for (int i=5; i>0; i--)
        {
            str += i + " ";
        }
        MessageBox.Show(str);
    }
}
```

The output looks:

```
5 4 3 2 1
```

The following demonstrates how to count down by 5 from 50 to 0 using the expression, i=5, to define the decrement.

```csharp
using System;
using System.Windows.Forms;

class Example
{
    public static void Main()
    {
        String str="";
        for (int i=50; i>=0; i-=5)
        {
            str += i + " ";
        }
        MessageBox.Show(str);
    }
}
```

The output looks:

```
50 45 40 35 30 25 20 15 10 5 0
```

The following uses a for loop to calculate the sum of 1 + 2 + 3 + ... + 50. The variable sum of int type is set to be 0 prior to the for loop. The for loop starts with i = 1 and ends at i=50. When i=1, the addition assignment operator (+=) will add 1 to sum, so the value of sum becomes 1. When i=2, 2 is added to sum and the value of sum becomes 3 (1+2=3). When i=3, 3 is added to sum and the value of sum becomes 6 (3+3=6). When i=4, 4 is added to sum and sum becomes 10 (6+4=10).
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        int sum = 0;

        for (int i=1; i<=50; i++)
        {
            sum += i;
        }

        MessageBox.Show(sum + "");
    }
}

The following is one way to find the “factorial” of 7 (or 7! = 1 × 2 × 3 × 4 × 5 × 6). Factorial is the product of an integer and all the integers below it. In mathematics, by definition, 0! = 1 and 1! = 1. Since 1! is defined as 1, the counter variable i can start at 2 because 1 × 1 = 1. The variable name, fac, is short for factorial and is used to keep the result of calculation. When i = 2, fac = 1 * 2, so the value of fac becomes 2. When i = 3, fac = 2 * 3, so the value of fac becomes 6. When i = 4, fac = 6 * 4, so the value of fac becomes 24. When i = 5, fac = 24 * 5, so the value of fac becomes 120. When i = 6, fac = 120 * 6, so the value of fac becomes 720. Since 6 is the terminal value, the output is 720.

using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        int fac = 1;

        for (int i = 2; i<=6; i++)
        {
            fac = fac * i;
        }

        MessageBox.Show(fac + "");
    }
}

The for loop is often used to iterate through an array or to populate elements of an array. By the way, a later lecture will discuss array in detail. The following code contains two for loops. The first for loop creates (populates) 47 elements to the “x” array. The second for loop iterates through the entire “x” array and displays the values of each element on a one-by-one basis.

using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        int[] x = new int[47]; // declare an array of int type

        for (int i=0; i<47; i++)
        {
            // code to populate the array goes here
        }

        // code to display the array values goes here
    }
}
```csharp
{x[i] = i + 1; // populate the element } String str = "";

for (int i=0; i<47; i++)
{ str += x[i] + " " ; // iterate through elements }

MessageBox.Show(str); }
}

The following example demonstrates how to find the minimal value from an array of `int` type. The code starts with assuming that `x[0]` is the minimal. Since the first element is assumed to be the minimal value, `x[0]`, the initial value is set to be 1 instead of 0. With the initial value is 1, the `for` loop will ignore `x[0]` and iterate through `x[1]`, `x[2]`, ..., till the `x[10]` (there are 11 elements in "x"). During each iteration, the `if` statement will check whether the current `x[i]` is less than the value kept by a variable "`min`". When the evaluation is true, the value of `x[i]` will be assigned to "`min`" as the new minimal.

```csharp
using System;
using System.Windows.Forms;

public class Example
{
  public static void Main()
  {
    int[] x = {17, 8, 12, 16, 5, 9, 10, 3, 14, 7};
    int min = x[0];
    for (int i=1; i<x.Length; i++)
    { if (x[i] < min) { min = x[i]; } }
    MessageBox.Show("The smallest value is " + min);
  }
}

A nested loop is a loop within a loop. It is created by adding an inner loop within the body of an outer one. The following bordered table illustrates the concept of a nested loop and how it works. The bordered table has three rows and four columns (and is said to be a 3×4 table or 3-by-4 table). Since every row has four columns, it takes one `for` loop to iterate through every column in a row. Since there are three rows, it takes another `for` loop to iterate through every row.

```
|   | 0  | 1  | 2  | 3  |
|---|----------------|
| 0 | A  | B  | C  | D  |
| 1 | E  | F  | G  | H  |
| 2 | I  | J  | K  | L  |
```

The following is a sample that uses two `for` loops. The outer iterates through rows while the inner iterates through columns. Every iteration of the outer `for` loop requires a full execution of the inner `for` loop.
```
for (int r=0; r<3; r++) // iterate rows
{
    for (int c=0; c<4; c++) // iterate columns
    {
    }
}

The following is a complete code. It uses two for loops to iterate through every element of the “x” array. Elements of the “x” array are organized into three sub-arrays. The inner for loop iterates through every element of a sub-array while the outer for loop iterates through the sub-arrays. A later lecture will discuss the concept of two-dimensional array in detail.

```
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        Char[,] x = new char[3,4] { {'A', 'B', 'C', 'D'},
                                {'E', 'F', 'G', 'H'},
                                {'I', 'J', 'K', 'L'}};

        string str = "";
        for (int r=0; r<3; r++) // iterate rows
        {
            for (int c=0; c<4; c++) // iterate columns
            {
                str += x[r,c] + " ";
            }
            str += "\n";
        }
        MessageBox.Show(str);
    }
}
```

The following is a sample code that demonstrates how to use nested for loops to sort an integer array. The inner for loop is responsible for finding the smallest element, and then swapping its value with x[k] where k starts with 0. When k=0, x[0] is the first element. At the end of the inner for loop, x[0] should be the smallest value in the array. Then k can be incremented by 1; therefore, x[k] become x[1]. The outer for loop is responsible to iterate from x[k+1] to x[10] to repeatedly search for the smallest, the second smallest, ..., till the largest element. The reason why the instructor starts with x[k+1] is because x[k] is assumed to be the smallest; therefore, it is not necessary to check if x[k] < x[k]. The comparison should be between the current x[k] and its adjacent value x[k+1]. Both for loops eventually rearranges the order of elements of the x array so that x[0] holds the smallest value, x[1] holds the next smallest, and so forth up to x[n-1], the largest.

```
using System;
class Sample
{
    public static void Main()
    {
        int[] x = { 3, 8, 6, 1, 9, 4, 0, 2, 7, 5 }; // int array
        int k = 0;
        int min = x[k]; // assuming x[0] is the smallest
```
for (int j=k+1; j<10; j++) // iterate through the loop
{
    for (int i=k+1; i<10; i++) // swapping
    {
        if (x[i] < min)
        {
            min = x[i];
            x[i] = x[k];
            x[k] = min;
        }
    }
    k++;
}

for (int i=0; i<10; i++) // display the sorted array
{
    Console.Write(i + " ");
}
}

The while loop

The while loop executes a statement or a block of statements until a specified expression evaluates to false. The following is the barebone structure of a while statement in C#. The expression must be a Boolean expression that can only be evaluated to be either true or false.

while (expression)
{
    statements
}

The following demonstrates how the above syntax works in C#. The Boolean expression to be evaluated is, (new Random()).Next(0, 10) != 5, which will randomly pick an integer from the range 0 to 9 and check if the value is not equal to 5. The while loop will terminate only when the randomly picked value is 5.

using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        while( (new Random()).Next(0, 10) != 5 )
        {
            Console.Write("Hello world!
");
        }
    }
}

In the following example, the initial value of the “result” variable of DialogResult enumeration is set to be “DialogResult.No”. The expression of the while loop is to test if the value of “result” equals to “DialogResult.No”. As long as the expression is true, the while loop must continue to execute the statement enclosed by { and }. The while loop will continue to display the same question unless the user clicks the “Yes” button to change the value of result to “DialogResult.Yes”.

using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        DialogResult result = DialogResult.No;
        while (result == DialogResult.No)
        {
            result = MessageBox.Show(
                "Is Taipei the capital city of Taiwan?", 
                "Quiz", 
                MessageBoxIcon.Question);
        }
    }
}

In the following, when the expression, \( n_1 = n_2 \), is true, \( n_2 \) must be assigned a new random number between 0 and 10.

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        int n1 = (new Random(1)).Next(10);
        int n2 = (new Random(2)).Next(10);
        while (n1 == n2)
        {
            n2 = (new Random(2)).Next(10);
        }
        MessageBox.Show(n1 + " " + n2);
    }
}
```

In the above `while` loops, the expression that defines the terminal value is checked each time at the beginning of the loop; therefore, even if this value changes during the execution of the nested statement(s), execution will not stop until the end of the iteration.

Similar to the `for` loop, programmers can specify the initial value using the following C# syntax of `while` statement.

```csharp
initialValue
while (terminalValue)
{
    statements
    increment/decrement
}
```

In the following example, the `while` statement is defined to start with \( i = 0 \), then checks if \( i \) is less than 4. If true, it displays the value of \( i \) starting at 0 because 0 is the initial value. The loop is forced to stop when \( i \) becomes 4 because 4 is the terminal value.
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        String str="";
        int i=0; // initial
        while (i < 4) // terminal
        {
            str += i + " ";
            i++;
        // increment
        }
        MessageBox.Show(str);
    }
}

The output looks:

0 1 2 3

The following demonstrates one way to calculate \(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{25}\). The variable of sum is declared as double type because the above mathematical series should produce an output of floating-point value, not an integer. It is necessary to note that C# is a type-sensitive language, which means the arithmetic calculation is performed based on the data type. For example, the statement, MessageBox.Show( (1/5) + ""), produce 0 because both 1 and 5 are values of int type. When an int value is performed with another int value, the output must be a value of int type. The statement, MessageBox.Show( (1/5.0) + ""), produce 0.2 because 5.0 is a floating-point value.

using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        double sum = 0;
        double n = 1.0; // initial value
        while (n <= 25) // termina value
        {
            sum = sum + (1/n);
            n++; // increment
        }
        MessageBox.Show(sum + " ");
    }
}

The output of the above code is 3.81595817775351.
The following is a sample program that can calculate a special series, \( \frac{1}{3} + \frac{3}{5} + \frac{5}{7} + \frac{7}{9} + \frac{9}{11} + \ldots + \frac{95}{97} + \frac{97}{99} \). The output is 45.1244503030502. According to the series, the first value is 1, the last value is 97, and the pattern is \( \frac{n}{n+2} \). When \( n=1 \), the element is \( \frac{1}{3} \). When \( n=97 \), the element is \( \frac{97}{99} \).

```csharp
using System;
using System.Windows.Forms;

double sum = 0;
double n = 1.0;
while (n<=97)
{
    sum += n / (n+2);
    n+=2.0;
}
MessageBox.Show(sum + ";
```

The `while` loop can also iterate through an array, as shown below. The “c” array has five elements with each being an English alphabet. Since the first element has an index 0 and the last element has an index 4. The initial value of the counter variable \( i \) is set to start with 0 and the terminal value is set to end at 4.

```csharp
using System;
using System.Windows.Forms;

c[0] = { 'A', 'P', 'P', 'L', 'E' };
int i=0;
while (i<5)
{
    Console.Write(c[i]);
    i++;
}
```

The following “loan payment formula” is used to calculate the payments on a loan. A loan, by definition, consists of a series of future periodic payments. The PV, or present value, calculates the present-day value of an amount that is received at a future date. The original loan amount is essentially the present value of the future payments on the loan.
\[ P = \frac{r(PV)}{1-(1+r)^{-n}} \]

where,
- \( P \) = monthly payment
- \( PV \) = Present Value
- \( r \) = interest rate period
- \( n \) = number of periods

It is necessary to note that a negative exponent can be expressed as \( a^{-n} = \frac{1}{a^n} \); therefore, the above equation can be rewritten as:

\[ P = \frac{r(PV)}{1-(1+r)^{-n}} \]

The following uses a while loop to calculate only the \( \frac{1}{(1+r)^n} \) part of the above equation. The result is assigned to a variable “exp” of double type.

```csharp
double exp = 1.0;
int i=0;
while (i<n)
{
    exp = exp * (1+r);
    i++;
}
```

The following is the complete code that can return the monthly payment of a $10,000 loan with annual interest rate of 8% and a total of 10 monthly payments. By the way, an annual interest rate of 8% is equal to \((0.08 / 12)\) of monthly interest rate. The `Round(v, d)` method of Math class can round the value \( v \) to the \( d \) number digits next to the decimal point.

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        double r = (0.08) / 12;
        double PV = 10000;
        int n = 10;
        double exp = 1.0;
        int i=0;
        while (i<n)
        {
            exp = exp * (1+r);
            i++;
        }
        double P = (r*PV) / (1 - (1/exp));
    }
}
```
The above program produces the following result.

$1037.03

The **do .. while** loop is a variation of **while** loops, except the expression is checked at the end of iteration instead of in the beginning. The syntax is:

```csharp
do {
    statement
} while (expression);
```

The following example demonstrates the usage of the **do..while** statement. The counter variable, *i*, has an initial value of 0 and a terminal value of 3 (because *i* < 4). The increment is 1. The output of the following code is 0 1 2 3.

```csharp
using System;
using System.Windows.Forms;

class Example {
    public static void Main() {
        string str = "";
        int i = 0;
        do {
            str += i + " ";
            i++;
        } while (i < 4);
        MessageBox.Show(str);
    }
}
```

The main difference from regular **while** loops is that the first iteration of a **do..while** loop is guaranteed to run, the Boolean expression is only checked at the end of the iteration, whereas it may not necessarily run with a regular **while** loop. In a regular **while** loop, as shown below, the Boolean expression is checked at the beginning of iteration, if it evaluates to FALSE right from the beginning, the loop execution would end immediately.

```csharp
int i = 0;
while (i < 4) {
    str += i + " ";
    i++;
}
```
The following is one of the ways to calculate $3^4$ (or $3 \times 3 \times 3 \times 3$). The variable, `exp`, should start at 1 because $3^0 = 1$. The counter variable, `i`, is set to start with 0 and end at 3 because the calculation of $3^4$ requires 4 iterations.

```csharp
using System;
using System.Windows.Forms;

class MyFor
{
    public static void Main()
    {
        int exp = 1;
        int i = 0;
        do
        {
            exp = exp * 3;
            i++;
        } while (i < 4);
        MessageBox.Show(exp + " ");
    }
}
```

The following uses a `do..while` loop and the $C=(F-32)\times\frac{5}{9}$ equation to convert Fahrenheit degrees to Celsius degrees. The initial value is 0, the terminal value is 30, and the increment is 1.

```csharp
using System;
using System.Windows.Forms;

class Example
{
    public static void Main()
    {
        string str = "Fahrenheit\tCelsius\n";
        int f = 0;
        do
        {
            str += f + "\t" + (f - 32)*(5.0/9.0) + "\n";
            f++;
        } while (f <= 30);
        MessageBox.Show(str);
    }
}
```

The following uses a `do..while` loop to count the total number of elements in the “x” array (which is `int` type). The `do..while` loop attempts to re-assign the value back to the element during each of the iteration. The counter variable, `size`, will increment by one. There are 11 elements in the “x” array; therefore, x[0], x[1], x[2], ..., and x[10] are valid elements. When the value of `size` becomes 11, x[11] is to an invalid index, the runtime will raise an exception indicating that an index is out of range. The following code uses a `try..catch` structure to handle the exception and uses a “break” statement to break out the `do..while` loop. By the way, a later lecture will discuss the `try..catch` structure in detail.

```csharp
using System;
using System.Windows.Forms;

class Example
{
    public static void Main()
    {
        int[] x = new int[11];
        int size = 0;
        do
        {
            x[size] = size;
            size++;
        } while (size < 11);

        try
        {
            int[] y = new int[size];
        }
        catch
        {
            MessageBox.Show("Index is out of range.");
            break;
        }
        MessageBox.Show("Number of elements = "+ size);
    }
}
```
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        int[] x = {17, 8, 12, 16, 5, 9, 10, 3, 14, 7};

        int size = 0;
        do
        {
            try
            {
                x[size] = x[size];
            }
            catch
            {
                break;
            }
            size++;
        } while (true);

        size++; // index starts at 0
        MessageBox.Show((size) + "");
    }
}

It is necessary to note that indexes of an array must start at 0 and end at size-1 (the term “size” refers to the total number of elements of the array). Therefore, the total number of an array is always one number larger than the last index.

Advanced Flow Control

Flow control means exactly what it sounds like. It is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated. Programmer can regulate the order in which the code should be executed, how many times it is executed, and if it is executed at all.

C# provides several “constructs” for advanced flow control. The term “language construct” is often used as a synonym for control structure and should not be confused with a function. The three commonly used constructs are:

- **break**: The *break* statement is used to stop (or terminate) the execution of a loop.
- **continue**: In C#, the *continue* statement forces a loop to skip to the next evaluation of the loop.
- **goto**: The *goto* statement transfers the program control directly to a labeled statement.

The *break* statement ends execution of the current *for*, *while*, *do..while* or *switch..case* structure. In the following example, the *while* loop will stop when *i* equals to 7. “Stopping a loop” means to completely jump of the loop and hand over the control to the next program object.

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
```
{  
    String str="";
    int i = 0;
    while (i <= 9)  
    {  
        str += i + " ";
        if (i == 7)  
        {  
            break;  //force the while loop to stop here  
        }  
        i++;
    }  
    MessageBox.Show(str);
}  
}  

Since the loop is forced to stop at the value 7. 8 and 9 will not be displayed. So, the output is  
0 1 2 3 4 5 6 7  

The following is another example that can determine whether or not a given number is a prime number. It is based on a very simple mathematic definition: Any number that has only two factors, 1 and itself, is a prime number. The for loop does the “integer division” from 1 to the given number, which means divide the given number from 1 to itself to find how many factors the given number has. The variable “factor” is a counter that counts the number of factors of the given number. When the value of “factor” is greater than 2 (meaning there are more than two factors found), then the number is not a prime number, the break statement will force the for loop to terminate.

    using System;
    using System.Windows.Forms;

    public class FindPrimeNumber  
    {  
        public static void Main()  
        {  
            int x = Convert.ToInt32(InputBox.Show("Enter a number:"));  
            String str="It is a prime number";
            int factor = 0;
            for (int i=1; i<=x; i++)  
            {  
                if (x%i==0) { factor++; }  
                if (factor > 2)  
                {  
                    str = "It is NOT a prime number";
                    break;  //terminate the loop  
                }  
            }  
            MessageBox.Show(str);
        }  
    }
The following is a sample output.

The continue statement is used within a loop structures to skip the current iteration and continue the next iteration. In other words, the continue statement passes control to the next iteration. In the following example, the expression, \((i \% 3 == 0)\), will check whether or not the current value of \(i\) is a multiple of 3 and skip all the multiple of 3.

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        String str = "";
        for (int i = 0; i < 10; i++)
        {
            if (i % 3 == 0)
            {
                continue;
            }
            str += i + " ";
        }
        MessageBox.Show(str);
    }
}
```

The following is a sample output. 3, 6 and 9 are skipped because they are multiple of 3. It is necessary to note that the continue statement does not break out of the loop, it just skips the current iteration.

```
1 2 4 5 7 8
```

The goto statement transfers the program control directly to a labeled statement. A labeled statement is a named code block that can be accessed from other part of the program. In the following example, the “square” label statement appends the word “three” to the \(str\) variable. It is necessary to note that the goto statement unconditionally transfers control to the statement labeled by the specified identifier, and it is a one-way transfer which means the goto statement does not return the controls to its calling party.

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        String str = "";
        for (int i = 0; i < 10; i++)
        {
            if (i == 7)
            {
```
goto square;
}
else
{
    str += i + " 
"
}
}
square:
str += "three";
MessageBox.Show(str);
}
}

The output looks:

0 1 2 3 4 5 6 three

The foreach loop

C# also provides the `foreach` statement to execute a block of statements for each element in an instance of the “collection” type. In C#, an object of collection type are groups of related objects with a common identifier. Array is an example of collection type. The following illustrates the syntax, where `delegate` is a variable of the same data type as the “collection” object (such as an array). During each iteration, delegate represent an element of the “collection” object.

```csharp
foreach (dataType delegate in collectionObject) { }
```

The following example uses a `foreach..in` statement to provide a simple, clean way to iterate through the elements of a `string` array. Since the “color” array is declared as `string` type, the “delegate” variable, `c`, is also declared as `string` type. When the first iteration starts, `c` represents “blue”. When the second iteration starts, `c` represents “red”, and so on.

```csharp
using System;
using System.Windows.Forms;

class Example
{
    public static void Main()
    {
        string str = "";

        string[] colors = { "blue", "red", "orange", "yellow", "green" };

        foreach (string c in colors)
        {
            str += c + "\n";
        }

        MessageBox.Show(str);
    }
}
```

The following code creates an array called `n` of `int` type and iterates through it with the `foreach` statement. A later lecture will discuss the concept of array in details.

```csharp
using System;
using System.Windows.Forms;

class Example
{
```csharp
{ public static void Main()
{
    String str="";
    int[] n = { 4, 5, 6, 1, 2, 3, -2, -1, 0 }; // array
    foreach (int i in n)
    {
        str += i + " ";
    }
    MessageBox.Show(str);
}
}
```

The output looks:

```
4 5 6 1 2 3 -2 -1 0
```

With multidimensional arrays, developers can use the same method to iterate through the elements. In the following example, the "n" array has three elements: {9, 99}, {3, 33}, and {5, 55}. Although each of the element is an individual array, the `foreach` loop can display all elements of the "n" array.

```csharp
using System;
using System.Windows.Forms;

public class ArrayClass
{
    public static void Main()
    {
        String str="";
        int[,] n = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };  
        foreach (int i in n)
        {
            str += i + " ";
        }
        MessageBox.Show(str);
    }
}
```

The output of this example is:

```
9 99 3 33 5 55
```

By the way, some programmers believe that a nested `for` loop can give programmers a more flexible control over the array elements of a multidimensional arrays. A later lecture will demonstrate how to use a nested `for` loop to iterate through a multidimensional array.

**Problem of infinite loop**

As stated previously, a repetition structure typically requires three basic components: an initial value, a terminal value, and an increment/decrement. One or more of these three components may be omitted, but care must be taken not to create an infinite loop or an endless loop.

A `for` loop must be written to eventually reach the terminal value. If not, the `for` loop will never terminate, and this is called an **infinite loop**. In the following example, the instructor purposely eliminates the increment; therefore, the value of `i` will stay as 0 and never advance to 1, 2, ...., and 10.
using System;

public class MyFor
{
    public static void Main()
    {
        for (int i=0; i<10; ) // missing increment/decrement
        {
            Console.Write(i + " ");
        }
    }
}

The output looks:

000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000

The following example does not define a terminal value; therefore, it is an **endless loop**. Executing an infinite loop or an endless loop can cause a situation called **memory overflow**. The consequence is possibly the over-sized utilization of memory space and a noticeable increase in CPU processing which will significantly slow down the computer.

using System;

public class MyFor
{
    public static void Main()
    {
        for (int i=0; ; i) // missing terminal value
        {
            Console.Write(i + " ");
        }
    }
}

**Practical Examples**

Julian calendar is a calendar introduced by the authority of Julius Caesar in 46 BC, in which the year consisted of 365 days, every fourth year having 366 days. It was superseded by the Gregorian calendar though it is still used by some Orthodox Churches to calculate the date of Easter. The following figure demonstrates what a Julian Calendar is.

<table>
<thead>
<tr>
<th>Year 2025</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>..........................</td>
</tr>
<tr>
<td>..........................</td>
</tr>
<tr>
<td>348</td>
</tr>
<tr>
<td>355</td>
</tr>
<tr>
<td>362</td>
</tr>
</tbody>
</table>

According to the above figure, January 1, 2025 is a Wednesday. In the row of week 1, no value should be displayed on “Sun”, “Mon”, and “Tue” columns while 1 is placed in “Wed”, 2
in “Thu”, 3 in “Fri”, and 4 in “Sat” columns. In the previous lecture, the instructor discussed about how to write codes to determine the weekday of January of a given year. The following are the codes with minor modifications. Be sure to refer to the previous lecture for details of the programming logic.

```csharp
int y = 2025;
//determine the weekday of Jan 1
int firstDay = 0; //0-Sun, 1-Mon, 6-Sat
if (y%28 == 12 || y%28 == 17 || y%28 == 23 || y%28 == 6)
{ firstDay = 6; }
if (y%28 == 22 || y%28 == 0 || y%28 == 5 || y%28 == 11)
{ firstDay = 5; }
if (y%28 == 10 || y%28 == 16 || y%28 == 21 || y%28 == 27)
{ firstDay = 4; }
if (y%28 == 15 || y%28 == 26 || y%28 == 4 || y%28 == 9)
{ firstDay = 3; }
if (y%28 == 14 || y%28 == 20 || y%28 == 25 || y%28 == 3)
{ firstDay = 2; }
if (y%28 == 13 || y%28 == 19 || y%28 == 2 || y%28 == 8)
{ firstDay = 1; }

Once the weekday of January 1 is determined. The following for loop, can insert a “tab” (blank spaces) to the column(s) that is/are supposed to be blank. For example, when the value of firstDay is 3 (which means Wednesday), k can only be 0, 1, and 3; therefore, three “tab” (specified by “\t”) will be added to the week 1 row.

```csharp
// insert tab before Jan 1
for (int k = 0; k < firstDay; k++)
{
    str += "\t";
}
```

The following is a program that asks user to enter the year. It, then, displays the Julian calendar for the year.

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        string input = InputBox.Show("Enter a year (e.g. 2029):");
        int y = Convert.ToInt32(input);
```
// determine the weekday of Jan 1
int firstDay = 0; // 0-Sun, 1-Mon, 6-Sat
if (y%28 == 12 || y%28 == 17 || y%28 == 23 || y%28 == 6) 
    { firstDay = 6; }
if (y%28 == 22 || y%28 == 0 || y%28 == 5 || y%28 == 11) 
    { firstDay = 5; }
if (y%28 == 10 || y%28 == 16 || y%28 == 21 || y%28 == 27) 
    { firstDay = 4; }
if (y%28 == 15 || y%28 == 26 || y%28 == 4 || y%28 == 9) 
    { firstDay = 3; }
if (y%28 == 14 || y%28 == 20 || y%28 == 25 || y%28 == 3) 
    { firstDay = 2; }
if (y%28 == 13 || y%28 == 19 || y%28 == 2 || y%28 == 8) 
    { firstDay = 1; }

int lastDayOfYear = 365; // assume 365 days in a year

// check if leap year
if (y%4 == 0 && (y%100 != 0 || y%400 == 0)) 
    { lastDayOfYear = 366; // 366 days in a leap year }

// set the display format
String str = "Sun	Mon	Tue	Wed	Thu	Fri	Sat
";

// insert tab before Jan 1
for (int k = 0; k < firstDay; k++)
    { str += "\t"; }

// add the days
int i=1;
while ( i <= lastDayOfYear )
    { 
        if ( (i+firstDay)%7 == 0 ) { str += i + "\n"; } 
        else { str += i + "\t"; }
        i++;
    }

Console.Write("Year " + y + "\n\n" + str + "");

The while loop has an initial value of 1 which is January 1. Its terminal value is either 365 or 366 (leap year). A previously lecture discussed how to use if statements to determine whether a given year is a leap year. Inside the while loop, there is an if..else statement. The expression, (i+firstDay)%7 == 0, evaluates whether the value of i is supposed to be placed on the “Sun” column. If so, add a new line “\n” instead of “\t” to break a line.
1. What is the result of the following code segment?

```c#
string str = "";
int i;
for (i=0; i<5; i++) {
    str += i + "";
}
MessageBox.Show(str + " ");
```

A. 01234
B. 012345
C. 1234
D. 12345

2. What is the result of the following code segment?

```c#
string str = "";
int i;
for (i=5; i>=0; i-=2) {
    str += i + "";
}
MessageBox.Show(str + " ");
```

A. 5310
B. 531
C. 543210
D. 420

3. What is the result of the following code segment?

```c#
string str = "";
int i=5;
while (i++ <= 9) {
    str +=i++ + " ";
}
MessageBox.Show(str + " ");
```

A. 5 6 7 8 9
B. 6 8
C. 6 8 10
D. 5 7 9

4. What is the result of the following code segment?

```c#
string str = "";
int n = 10;
while (n>3) {
    str += n + " ";
    n-=2;
}
```
5. What is the result of the following code segment?

```csharp
string str = "";
int n = 2;

do {
    str += n + " ";
    n++;
} while (n<=5);

MessageBox.Show(str + " ");
```

A. 10 8 6 4 2  
B. 8 6 4 2  
C. 8 6 4 2  
D. 10 8 6 4  

6. What is the result of the following code segment?

```csharp
string str = "";
int n = 10;

while (n>3) {
    if (n<6) { break; }
    str += n + " ";
    n-=2;
}

MessageBox.Show(str + " ");
```

A. 2 3 4 5  
B. 2 3 4  
C. 3 4 5  
D. 3 4  

7. What is the result of the following code segment?

```csharp
string str = "";
int n = 15;

do {
    str += n + " ";
    n-=5;
} while (n>=5);

MessageBox.Show(str + " ");
```

A. 10 5  
B. 15 10
8. What is the result of the following code segment?

```csharp
string str = "";
int[] n = {1, 2, 3, 4, 5};
foreach (int i in n){
    str += i*i + " ";
}
MessageBox.Show(str + " ");
```

A. 1, 2, 3, 4, 5  
B. 1 2 3 4 5  
C. 1, 4, 9, 16, 25  
D. 1 4 9 16 25

9. What is the result of the following code segment?

```csharp
string str = "";
for (int i=1; i<=5; i++){  
    if (i >= 3) { break; }  
    else { str += i + " ";}
}
MessageBox.Show(str + " ");
```

A. 1 2  
B. 1 2 3  
C. 1 2 3 4  
D. 1 2 3 4 5

10. What is the result of the following code segment?

```csharp
string str = "";
for (int i=1; i<=5; i++){  
    if (i >= 3) { goto end; }  
    else { str += i + " ";}
}
end:
    str += "3 4";
MessageBox.Show(str + " ");
```

A. 1 2  
B. 1 2 3  
C. 1 2 3 4  
D. 1 2 3 4 5
Learning Activity #1:
1. Create a new directory called C:\CIS218 if it does not exist.
2. Make sure the InputBox.cs file is the C:\cis218 directory.
3. Launch the Development Command Prompt (not the Windows Command Prompt). (See Lab #1 for details)
4. In the prompt, type cd c:\cis218 and press [Enter] to change to the C:\CIS218 directory.
5. In the prompt, type notepad lab5_1.cs and press [Enter] to use Notepad to create a new source file called lab5_1.cs with the following contents.

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        string str = "for version:
\n";
        int fac = 1;
        int i;
        for (i = 2; i<=6; i++)
        {
            fac = fac * i;
        }
        str += "6! = " + fac + "\n";

        str += "\nwhile version:
\n";
        fac = 1;
        i = 2;
        while (i<=6)
        {
            fac = fac * i;
            i++;
        }
        str += "6! = " + fac + "\n";

        str += "\ndo..while version:
\n";
        fac = 1;
        i = 2;
        do
        {
            fac = fac * i;
            i++;
        }while (i<=6);
    }
```
6. In the prompt, type `csc /t:winexe lab5_1.cs` and press [Enter] to compile the source code. The compiler creates a new file called `lab5_1.exe`.

7. Type `lab5_1.exe` and press [Enter] to test the program. A sample output looks:

8. Download the “assignment template” and rename it to `lab5.doc` if necessary. Capture a screen shot similar to the above and paste it to the Word document named lab5.doc (or .docx).

**Learning Activity #2:**

1. Under the C:\cis218 directory, use Notepad to create a new source file called `lab5_2.cs` with the following contents:

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        string str = "for version:\n";
        str += "Fahrenheit\tCelsius\n";

        int f;
        for (f = 0; f < 10; f++)
        {
            str += f + "\t" + (f-32)*(5.0/9.0) + "\n";
        }

        str += "\n\nwhile version:\n";
        str += "Fahrenheit\tCelsius\n";

        f = 10;
        while (f < 20)
        {
            str += f + "\t" + (f-32)*(5.0/9.0) + "\n";
            f++;
        }

        str += "\nndo..while version:\n";
        str += "Fahrenheit\tCelsius\n";

        f = 20;
    }
}
```
do
{
    str += f + "\t" + (f-32)*(5.0/9.0) + "\n";
    f++;
} while (f < 30);
MessageBox.Show(str);
}

2. Compile and test the program. The output looks:

![Output Image]

3. Capture a screen shot similar to the above and paste it to the Word document named lab5.doc (or .docx).

**Learning Activity #3:**
1. Under the C:\cis218 directory, use Notepad to create a new source file called lab5_3.cs with the following contents:

```csharp
using System;
using System.Windows.Forms;

public class Example
{
    public static void Main()
    {
        string input = InputBox.Show("Enter APR in percent (%):");         
        double r = (Convert.ToDouble(input) / 100) / 12;

        input = InputBox.Show("Enter the capital:");
        double PV = Convert.ToDouble(input);

        input = InputBox.Show("Enter total number of payments:");
        int n = Convert.ToInt32(input);

        double exp = 1.0;
        int i=0;
        while (i<n)
        {
            exp = exp * (1+r);
        }
    }
```
2. In the prompt, type `csc /t:winexe lab5_1.cs InputBox.cs` and press [Enter] to compile the source code.

3. Test the program. A sample output looks:

![Sample output image]

4. Capture a screen shot similar to the above and paste it to the Word document named `lab5.doc` (or `.docx`).

**Learning Activity #4:**

1. Under the C:\cis218 directory, use Notepad to create a new source file called `lab5_4.cs` with the following contents to calculate \( \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \):

```csharp
using System;
using System.Windows.Forms;
public class Example
{
  public static void Main()
  {
    double lastValue = Convert.ToDouble(InputBox.Show("Enter a value:"));
    
    string str = "for loop:\n";
    double sum = 0;
    double n = 1.0;
    for (n = 1.0; n <= lastValue; n++)
    {
      sum = sum + (1/n);
    }
    str += sum + "\n";
    str += "\n\nwhile loop:\n";
    sum = 0;
    n = 1.0;
    while (n <= lastValue) // terminate value
    {
      sum = sum + (1/n);
      n++;// increment
    }
    str += sum + "\n";
  }
}
```
str += "\n\ndo..while loop:
";
sum = 0;
n = 1.0;

do {
    sum = sum + (1/n);
n++; //increment
} while (n <= lastValue); //termination value
str += sum + "\n";
MessageBox.Show(str + " ");
```csharp
if (y%28 == 13 || y%28 == 19 || y%28 == 2 || y%28 == 8) { firstDay = 1; }

int lastDayOfYear = 365; //assume 365 days in a year

// check if leap year
if (y%4 == 0 && (y%100 != 0 || y%400 == 0))
{
    lastDayOfYear = 366; //366 days in a leap year
}

// set the display format
String str ="<!Doctype html><html><body><table>");
str += "<caption>Year " + y + " Julian Calendar</caption>";
str += "<tr><td align='right'>Week</td>|<td>Sun</td>|<td>Mon</td>|<td>Tue</td>|<td>Wed</td>|<td>Thu</td>|<td>Fri</td>|<td>Sat</td> </tr>";

int wk = 1; // number of week
str += "<tr><td align='right'>" + wk + " |</td>");

// insert tab before Jan 1
for (int k = 0; k < firstDay; k++)
{
    str += "<td></td>");
}

// add the days
int i=1;
while ( i <= lastDayOfYear)
{
    if ((i+firstDay)%7 == 0)
    {
        wk++;
        str += "<td>" + i + "</td><tr><td align='right'>" + wk + " |</td>");
    }
    else { str += "<td>" + i + "</td>"); }
    i++; }

str += "</tr></table>";
Console.Write(str);
```

2. Type `csc /t:winexe lab5_5.cs InputBox.cs` and press [Enter] to compile.

3. Type `lab5_5 > lab5_5.htm` and press [Enter] to test the program but save the output in the “lab5_5.htm” file. The following dialog box appears. Enter an integer such as 2025.
4. Use a web browser to open the “lab5_5.htm” file. A sample output looks:

![Sample Output]

5. Capture a screen shot similar to the above and paste it to the Word document named lab5.doc (or .docx).

**Submittal**

1. Complete all the 5 learning activities.

2. Create a .zip file named lab5.zip containing ONLY the following self-executable files.
   - Lab5_1.exe
   - Lab5_2.exe
   - Lab5_3.exe
   - Lab5_4.exe
   - Lab5_5.exe
   - Lab5.doc (or .docx) [You may be given zero point if this Word document is missing]

3. Log in to course site and enter the course site.

4. Upload the zipped file as response to question 11.

**Programming Exercise: while loop**

1. Use Notepad to create a new file named ex05.cs with the following heading lines (be sure to replace YourFullNameHere with the correct one):

   ```c#
   //File Name: ex05.cs
   //Programmer: YourFullNameHere
   ```

2. Under the above two heading lines, write C# codes that declare a variable name “str” of string type. Then, use a while loop with initial value 1, terminal value 10, and increment 1. During every iteration, adds three values to the “str” variable: (1) the current value of the counter variable, (2) the value of square of the counter variable, and (3) the value of cube of the counter variable.

3. The following is a sample output. Be sure to use the while loop for iteration, not you might receive zero point.
4. Download the “programming exercise template”, and rename it to ex05.doc. Capture a screen shot similar to the above figure and then paste it to the Word document named “ex05.doc” (or .docx).

5. Compress the source code (ex05.cs), the executable (ex05.exe), and the Word document (ex05.doc or .docx) to a .zip file named “ex05.zip”. You may be given zero point if any of the required file is missing.

**Grading Criteria:**
- You must be the sole author of the codes.
- You must meet all the requirements in order to earn credits.
- No partial credit is given.