

LESSON 03 - MORE LOOP EXAMPLES



In this lesson we will examine more examples of using loops to solve problems. We will also look at how to read in files (input files) and how to write to files (output files). We then discuss the definition of what an algorithm is.

Sections:

- I. [TRAVERSING THROUGH STRINGS \(EXAMPLES\) PAGE 1](#)
- II. [INPUT/OUTPUT FILES PAGE 7](#)
- III. [ALGORITHMS PAGE 10](#)

I. TRAVERSING THROUGH STRINGS (EXAMPLES):

Using a **for-loop** we can iterate through each character in a string (**traverse** a string) by making use of a for-loop's counter variable. Let's look at some examples:

Note: We could use while-loops when processing strings, but since strings are of known length it makes more sense to use a for-loop which is a **definite** loop (i.e., we know how many times we need to cycle), whereas while-loops are **indefinite**.

Example 1 – Output each character from a string on a separate line:

Let's start with a simple example which uses a **for-loop** to iterate through each character of a string and output each character on a separate line:

```
string s = "Hello World";
for (int x = 0; x < s.Length; x++)
    Console.WriteLine(s[x]);
```

Sample Output:

H
e
l
l
o

W

Notice the condition in the **for-loop** will only go up to, but not including the length of the string:

```
x < s.Length;
```

Since there are **11** characters in our string 's', our **counter variable 'x'** will start at '**0**' and go up to '**10**'. Inside our **for-loop** we output the character at position '**x**':

```
Console.WriteLine(s[x]);
```

This is straightforward. Here is what **s[x]** equals on every iteration:

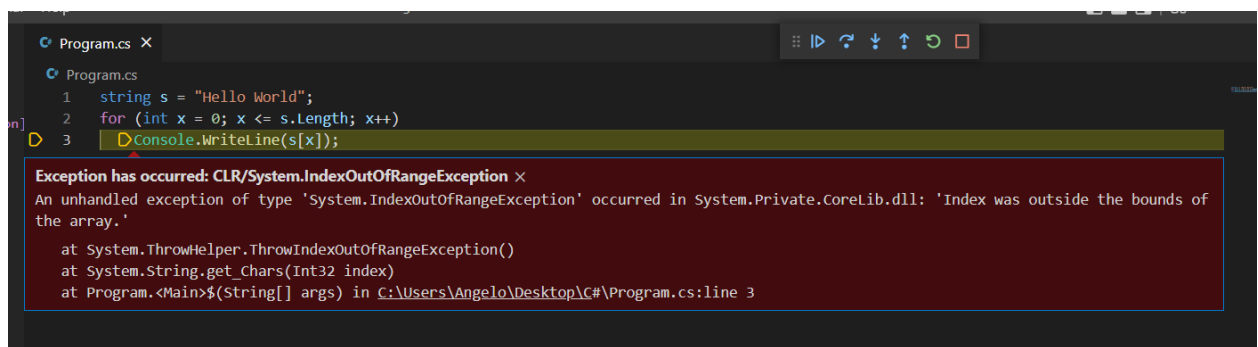
```
s = "Hello World";
```

Iteration #1:	Iteration #2:	Iteration #3:	And so on...
x = 0	x = 1	x = 2	
Therefore:	Therefore:	Therefore:	
s[x] = s[0] = H	s[x] = s[1] = e	s[x] = s[2] = l	

The counter variable '**x**' in our **for-loop** acts as an **index** for our string position. Remember that all strings start at index '0'. Therefore, in our **for-loop**, we first set '**x**' equal to '**0**'. Then on every iteration, '**x**' will increase by '**1**' and go up to, but not including **s.Length** (i.e., up to but not including **11**). If we went up to an including **s.Length** we would get an **out of bounds** error because index '**11**' does not exist in string 's'. Here is a demonstration:

```
string s = "Hello World";
for (int x = 0; x <= s.Length; x++)
    Console.WriteLine(s[x]);
```

Now we are saying to increase 'x' up to and including s.Length. However, when we run our program, we will get the following exception:



This is because `s[11]` does not exist, and we are telling our for-loop to go up to and including 11 in this case. Therefore, we must use our original condition so that we do not go out of bounds:

`x < s.Length;`

~~`x <= s.Length;`~~

Example 2 - Counting vowels in a string:

Let's consider the example where we would like to count all vowels that appear in a string:

```
string s = "Hello ICS. Great progress thus far in the course!";
s = s.ToLower();
int vowels = 0;
for (int x = 0; x < s.Length; x++)
    if (s[x] == 'a' || s[x] == 'e' || s[x] == 'i' || s[x] == 'o' || s[x] == 'u')
        vowels++;
Console.WriteLine(vowels);
```

The first thing to notice in the above code is that we convert the entire string to **lower case** before checking for vowels so that checking each vowel easier to do:

`s = s.ToLower();`

The condition in the **for-loop** will only go up to, but not including the length of the string:

`x < s.Length;`

Since there are 50 characters in our string 's', our counter variable 'x' will start at '0' and go up to '49'.

The **if-statement** inside our **for-loop** checks if the current character in the string is a vowel or not, and we increase a **separate counter variable** named '**vowels**' if the current character is indeed a vowel:

```
if (s[x] == 'a' || s[x] == 'e' || s[x] == 'i' || s[x] == 'o' || s[x] == 'u')
    vowels++;
```

Each iteration of the **for-loop** checks each letter of the string 's' against all vowels (a, e, i, o, u). When the for-loop completes, the result of '**vowels**' will be the number of vowels that appeared in the sentence regardless of uppercase/lowercase since we initially converted the entire string to lower case.

Note: Our **for-loop** does not contain any braces `{ }`. This is because there is only **one** statement after. Even though the **if-statement** is on two lines; it is still considered a single statement since there is only one line of code for that if-statement. However, to avoid confusion, you may want to make use of the braces for this for-loop.

Example 3 – Simple string encryption:

The definition of **encryption** is:

The process of converting information or data into a code, especially to prevent unauthorized access. (Google)

What does this mean? Let us take the following sentence and encrypt it:

hello ics. how are you?

Here is our encryption:

hell*@ics.@h*w@ar*@yo*?

What did we do?

- We replaced every second vowel with an asterisk (*)
- We replaced all spaces with the 'at' symbol (@)

Let us first look at the code for replacing every vowel with an asterisk:

```
string s = "Hello ICS. How are you?";
s = s.ToLower();
string sEncrypt = "";
bool asterisk = false;
for (int x = 0; x < s.Length; x++)
{
    if (s[x] == 'a' || s[x] == 'e' || s[x] == 'i' || s[x] == 'o' || s[x] == 'u')
    {
        if (asterisk)
        {
            sEncrypt = sEncrypt + '*';
            asterisk = false;
        }
        else
        {
            sEncrypt = sEncrypt + s[x];
            asterisk = true;
        }
    }
    else
    {
        sEncrypt = sEncrypt + s[x];
    }
}
Console.WriteLine("Original string: " + s);
Console.WriteLine("Encrypted string: " + sEncrypt);
```

Sample Output:

Original string: hello ics. how are you?
 Encrypted string: hell* ics. h*w ar* yo*?

In the above code, we create a string variable named **'sEncrypt'** which will hold our encrypted string. In the **for-loop**, we check for a vowel on every iteration (just like the previous example). We then use a **Boolean flag** named **'asterisk'** which will determine if we should save the letter as an asterisk or as just the letter (remember we are only replacing every second vowel with an asterisk).

Every time a vowel is found, the Boolean variable **'asterisk'** will flip (i.e., if **'asterisk'** currently equals **'false'** then it will be set to **'true'**, and vice versa). If **'asterisk'** equals **'false'** then **'sEncrypt'** equals itself plus an asterisk **'*'**. If **'asterisk'** equals **'true'** then **'sEncrypt'** equals itself plus an the current letter **'s[x]'**.

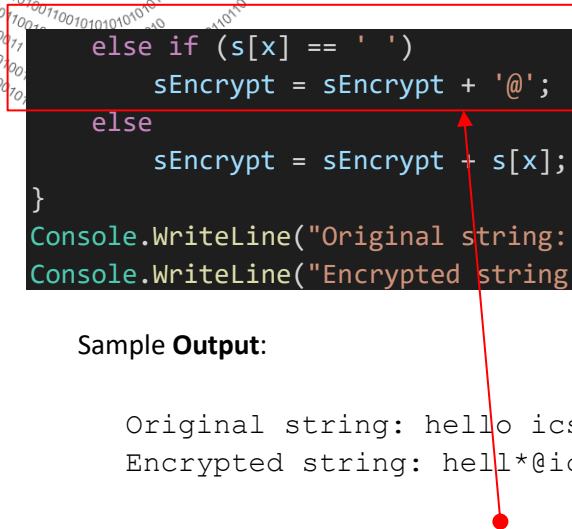
If the current letter **'s[x]'** is not a vowel, then **'sEncrypt'** will equal itself plus an the current letter **'s[x]'**.

Upon completion of the **for-loop**, the encrypted string **'sEncrypt'** will look like this:

hell* ics. h*w ar* yo*?

Now we need to add the code to replace all spaces with **'@'**. Let's add to our code:

```
string s = "Hello ICS. How are you?";
s = s.ToLower();
string sEncrypt = "";
bool asterisk = false;
for (int x = 0; x < s.Length; x++)
{
    if (s[x] == 'a' || s[x] == 'e' || s[x] == 'i' || s[x] == 'o' || s[x] == 'u')
    {
        if (asterisk)
        {
            sEncrypt = sEncrypt + '*';
            asterisk = false;
        }
        else
        {
            sEncrypt = sEncrypt + s[x];
            asterisk = true;
        }
    }
}
```



```
else if (s[x] == ' ')
    sEncrypt = sEncrypt + '@';
else
    sEncrypt = sEncrypt + s[x];
}
Console.WriteLine("Original string: " + s);
Console.WriteLine("Encrypted string: " + sEncrypt);
```

Sample Output:

```
Original string: hello ics. how are you?
Encrypted string: hell*@ics.@h*w@ar*@yo*?
```

All we had to do is add an extra **else-if** to handle the spaces (i.e., if a space is encountered then **'sEncrypt'** equals itself plus '@').

Our result for **'sEncrypt'** will now be:

```
hell*@ics.@h*w@ar*@yo*?
```

II. INPUT/OUTPUT FILES:

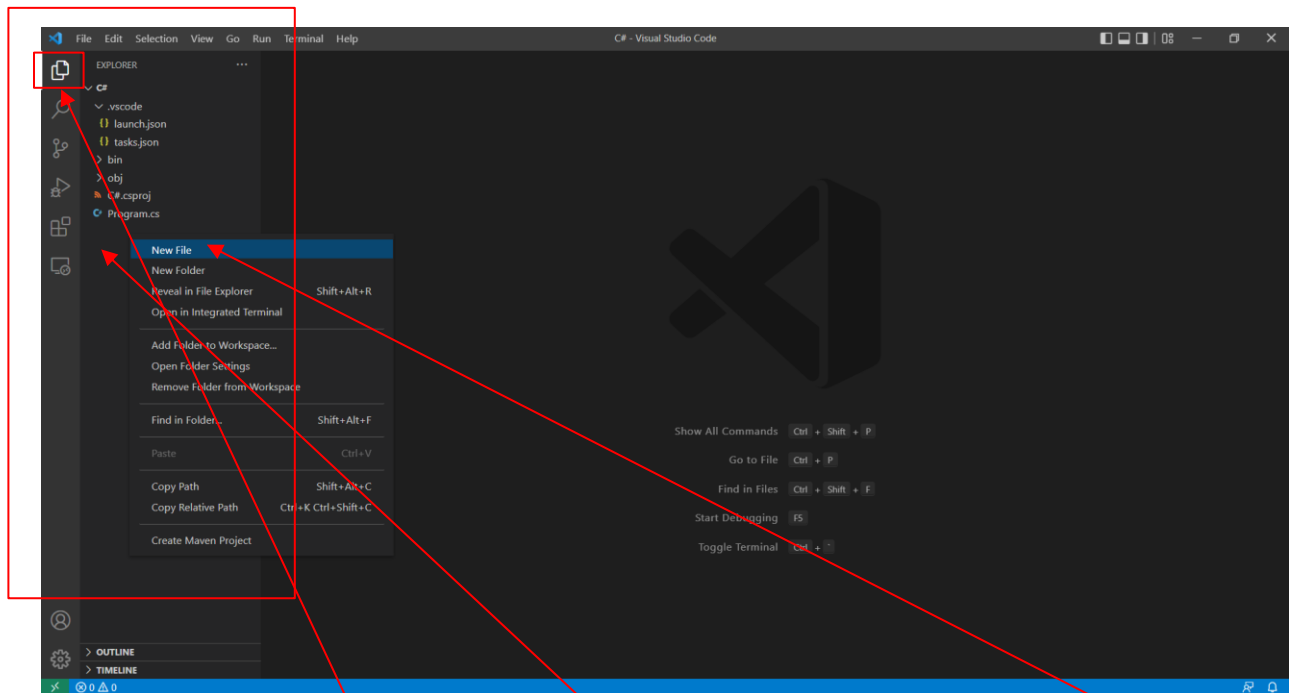
For this example, we will use a **while-loop** to iterate through all the records of an **input file**. Since we do not know how many records our input file may have, it makes more sense to use a **while-loop** which is an **indefinite** loop.

Here is what our input file looks like:

Input file (input.txt):

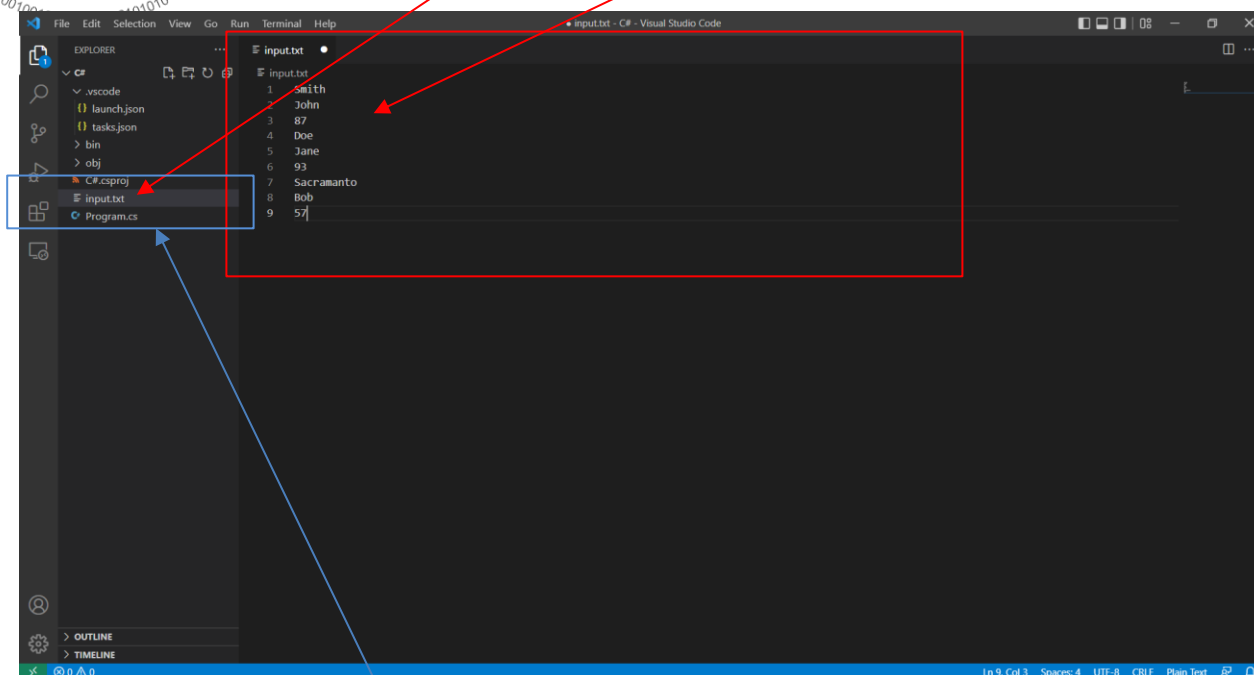
```
Smith
John
87
Doe
Jane
93
Sacramento
Bob
57
```

You can easily create this input file using an editor like Windows notepad, or by using Visual Studio Code:



In the Visual Studio 'File Explorer', you can **right click** in the empty space and then select '**New File**'.

From here you can name the file **'input.txt'** and hit enter. Once you **double click** the file it will open as a tab on the right, and you can type your data:



Note: It is important that the **input.txt** file resides in the same directory as your **.cs** file, so that it is easily accessible from your code.

Notice that there are **3 records** in this input file, where each record consists of 3 lines: **last name, first name, and mark**. We are going to **read** and output each record. We are also going to calculate the class average of all 3 marks and output that as well. Finally, we are going to **write** the average to an output file named: **output.txt**. Here is our code:

```
1 // constant variables
2 const string INPUT_FILE = "input.txt";
3 const string OUTPUT_FILE = "output.txt";
4
5 // open the input file
6 StreamReader sr = new StreamReader(INPUT_FILE);
7
8 // fields used for input file
9 string? line = "";
10 string firstName = "", lastName = "";
11 double mark = 0;
12
13 // variables for calculating average
14 double total = 0, count = 0, avg = 0;
15
```



```
16 // read the first line of text from the input file
17 line = sr.ReadLine();
18
19 // continue to read until you reach end of file
20 while (line != null)
21 {
22     // get firstName
23     firstName = line;
24
25     // read next line & get last name
26     line = sr.ReadLine();
27     lastName = line;
28
29     // read next line & get mark
30     line = sr.ReadLine();
31     mark = Convert.ToDouble(line);
32
33     // output record
34     Console.WriteLine(firstName + ' ' + lastName + ": " + mark);
35
36     // accumulate 'total' & increment 'count'
37     total = total + mark;
38     count++;
39
40     // read the next line
41     line = sr.ReadLine();
42 }
43
44 //close input file
45 sr.Close();
46
47 // calculate average
48 avg = total / count;
49
50 // output 'avg'
51 Console.WriteLine("\nClass Average: " + avg);
52
53 // open an output file
54 StreamWriter sw = new StreamWriter(OUTPUT_FILE);
55 // write 'avg' to output file
56 sw.WriteLine(avg);
57 // close output file
58 sw.Close();
```

First thing to notice is on **line 2 & 3** where we declare two variables for our input/output filenames with the **'const'** (short for **constant**) keyword in the front. This just tells C# that these variables **cannot** change in our code. If we try to change a variable in our code that is declared as constant, then we will get a build error!

On **line 6** we open our **input file** by using the built-in C# **StreamReader**. We declare a **StreamReader** variable named **'sr'**, and the **StreamReader** is passed the **'INPUT_FILE'** constant as a parameter. On **line 17** we read in our first line from the input file by using our **'sr' StreamReader** variable. The line is stored in a string variable called **'line'**. Therefore, at this point **'line'** will equal **'Smith'** which is the first line in our input file.

Line 20 is where we start our **while-loop** which will keep looping as long as the **'line'** variable is not null. **Null** is a special value which means **nothing**. Let's break down what is happening in this loop:

- **Line 23** – We store the value of **'line'** into **'firstName'**
- **Lines 26 & 27** – We read the next line and store it into **'lastName'**
- **Lines 30 & 31** – We read the next line and store it into **'mark'**
- **Line 34** – We output the entire record (the 3 fields): **first name, last name, and mark**
- **Lines 37 & 38** – We accumulate our **'total'** variable and increment our **'count'** variable used later to calculate the average
- **Line 41** – We read the next line from our file and store it into **'line'**. At this point, if there are no more lines in the file then **'line'** will equal **'null'** and the **while-loop** will **break** (stop).

On **Line 45** we **close** the **StreamReader 'sr'**. This is an important step because it 'closes' the file. If the file is not properly closed, then you may not be able to open/edit the file later.

After the while-loop stops we output the average. Then on **lines 54 to 58** we use a **StreamWriter** to create an **output file** that we can write to. It is used very similarly to **StreamReader**, and you can write to the file the same way you do for the console with **sr.WriteLine()**. Of course, we must close the **StreamWriter 'sr'** when finished. In our example, we simply write the average to the output file. Notice that this will create a new file called **'output.txt'** in your project root directory (where both your input.txt and .cs file reside). Every time your program runs, this output.txt file will be overwritten.



III. ALGORITHMS:

What we have been creating with these examples (and with some examples in previous lessons) are called **algorithms**. In computers, an algorithm is a set of steps that are created to solve a problem. When creating an algorithm, it should be able to solve the problem at hand using any input that was given. For example, when we encrypted our string in example 2, the algorithm that we created works for any string provided (i.e. it can encrypt any string).

Algorithms can become very complex. Take for example the Google search algorithm. We cannot see the source code on how it was designed, but I can guarantee you it is very complex! Their algorithm runs on servers which you connect to when you go to google.com. The code for their algorithm is constantly being improved by their software engineers and is worth billions of dollars!

Algorithms solve problems, and problem solving is at the core of Computer Science.

NOTE: There is no right or wrong for how you design/code your algorithms. The more you analyze pre-made algorithms (like the ones I give you) the better you will be able to design your own algorithms. A good algorithm usually comes down to simplicity and efficiency. This is a skill that only gets better with practice and experience.