TP C#2 : Debugging for a while

Assignment

Archive

You must submit a zip file with the following architecture :

```
rendu-tp-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- Debugger/
        |-- Debugger.sln
        |-- Debugger/
            |-- Everything except bin/ and obj/
```

Don't forget to check the following points before submitting :
- You will obviously replace *firstname.lastname* with your login, and don't forget the AUTHORS file.
- The AUTHORS and README files are mandatory.
- Do not leave *(*bin) and *(*obj) files in your project.
- You must follow scrupulously the required function prototypes.
- You must remove every test file from your code.
- *(*Your code must compile !)

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where '$' represents the newline and ' ' a whitespace) :

```
* firstname.lastname$
```

Please note that the filename is AUTHORS with NO extension. To easily create your `AUTHORS` file, you can type the following command in a terminal :

```
echo "* firstname.lastname" > AUTHORS
```

README

In this file, you can write any comments about the practical, your work, or more generally about your strengths / weaknesses. You must list and explain all the boni you have implemented. An empty `README` file will be considered an invalid archive (malus).

1 Introduction

1.1 Objectives

At the end of this TP, you will be able to use loops and the debugger. Loops are a means to repeat a portion of code, either a given number of times or until a given condition becomes false. The debugger is an essential tool in the search and elimination of bugs in your code. But a good tool is useless until you know how to use it.

2 Courses

2.1 Loops

The display of the current line number in C#, without loops, would look like this :

```
1 using System;
2 Console.WriteLine("this is the line number 1");
3 Console.WriteLine("this is the line number 2");
4 Console.WriteLine("this is the line number 3");
5 Console.WriteLine("this is the line number 4");
6 Console.WriteLine("this is the line number 5");
7 [...]
```

Beyond the obvious loss of time, if we have to correct the code because of a bug or a change in our project, we need to change each line independently. Moreover, the duplicate code here is rather small, but could be much bigger.

Loops can be used to solve at least two problems :

- Repeat one or more times an action while a certain condition is true.
  This is a **while** loop.

- Repeat an action a certain number of times. Most of the time, a variable is used as a counter. This kind of repetition is called iteration in computer science ;
  This is a **for** loop.

2.1.1 The while loop

This is the simplest loop. It allows you to execute lines of code while a condition is true. The basic syntax is as follows.

```
1 while (condition)
2 {
3     // do things
4 }
```

Here, the computer understands : *while the 'condition' is true, repeat the instructions between the curly braces.* Obviously, if the 'condition' is false at the beginning, it will not be possible to enter the loop.

Therefore, a basic example of a loop with a counter can be found below :

```
int counter = 0;
while (counter < 42)
{
    Console.WriteLine("If at first you don't succeed, redefine success!");
    counter = counter + 1;
}
```

The condition is **counter < 42**. As (counter) starts at 0, the program will enter the loop, display the text, increment counter and start again, 42 times.

A quick comment about the (counter = counter + 1;) of the example. This line is a way of counting the number of times the program entered the loop. If one is added to the counter, it is called an incrementation, and if one is removed, it is called a decrementation. As you may know, you will have to write lines as counter = counter + 1 really often. Fortunately, there exists a wrapper allowing you to make this writing a little less painful. To increment variables, you can simply write (variable += 1) or (variable++). To decrement, you can simply write (variable -= 1) or (variable−−).

There exists an alternative to (while), the loop (do...while), in which the verification of the condition takes effect after the end of the loop *(*after each run of its body). This implies that this loop is ALWAYS entered at least one time, even if the condition is basically wrong. Careful, do not forget the semi-colon at the end of the loop.

```
var i = 0;
do {
    Console.WriteLine("Palms are sweaty");
    Console.WriteLine("Knees weak, arms are heavy");
    Console.WriteLine("Vomit on my sweater already");
    Console.WriteLine("Mom's spaghetti");
    i++;
} while (i != 42); // Do not forget to close the loop
```

2.1.2 The for loop

Theoretically, the (while) loops could answer to any need that might arise concerning loops. Nevertheless, there exists another type of loop that simplifies the use of loops in some cases : the (for) loop. This is a condensed version of the while loop used to repeat instructions a certain number of times. Its basic syntax is as follow.

```
for (int counter = 0; counter < 42; counter++)
{
    Console.WriteLine("I am nobody");
    Console.WriteLine("Nobody is perfect");
    Console.WriteLine("Therefore I am perfect");
}
```

In a single line, we declare our variable, set the condition and increment the variable. This is really light and simple to understand. But it is no more than syntactic sugar. The first part of the line (before the first ';') is executed before the loop. The part in the middle is the condition, as in a while loop, and the last part is executed after each run of the loop. But be careful : a variable declared between the parentheses of the for loop won't be accessible outside the loop (you can do research on scoping if you want to go further).

2.1.3 The foreach loop

The loop **foreach** does not exist in every programming language. It is a variant of (for) that allows to run through a list of objects without a condition. But careful : this loop applies only to arrays or similar constructions that you are going to learn next week. The basic syntax of a foreach loop is :

```
int[] array = {0, 1, 2, 3, 4, 5, 42, 1337};
foreach(int element in array)
{
    Console.WriteLine("Element is:" element);
}
```

This code will simply display each element in textbf(array), one after the other. If you want to know more about it, you can google it to go further.

2.1.4 Break, Continue

There exist sub-keywords that allow the user to bypass loop behavior. As an example, you can ignore a run of your loop, or leave your loop as you wish at any run.
This first keyword is (break). It allows the user to get out of the loop in which the keyword is used.

```
for (int i = 50; i < 1337; i++)
{
    if (i == 53)
        break;

    Console.WriteLine("I only took " + i + " coffees today");
}
// output
// I only took 50 coffees today.
// I only took 51 coffees today.
// I only took 52 coffees today.
```

In this example, the program will stop at the 53rd iteration, when break will be called. As it starts at the 50th iteration, only those three lines will appear.

The (continue) keyword is a way to skip an iteration of the loop.

```
for (int i = 50; i < 1337; i++)
{
    if (i == 52)
        continue;

    Console.WriteLine("I only took " + i + " coffees today");
}
// output
// I only took 50 coffees today.
// I only took 51 coffees today.
// I only took 53 coffees today.
// ...etc...
```

2.2 The Debugger

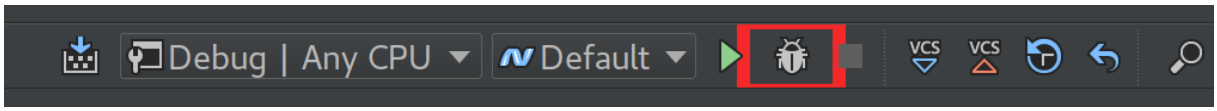
> If debugging is the process of removing bugs, then programming must be the process of putting them in - Edsger Dijkstra

You probably have understood : 'debugging' is the process of looking for and getting rid of bugs. It is almost impossible to create a program without a few errors, even if you are a genius. Knowing how to properly debug is therefore one of your most important skills, and you have to know and use the appropriate tools to be able to do it faster, and more easily.
The debugging phase of a program consists in being able to see what is going on with your program, how it works internally compared to how it should work. You then can follow your program line by line with a debugger to find any unintended behavior in your code. You will be able to :

- Pause your program.
- Display the values of your program at any time.
- Modify those values if you need to.
- Modify directly the program.
- List the called functions to locate your position in the program.
- and more.

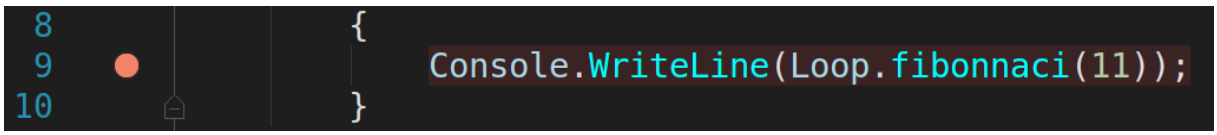
Teaching you how to properly debug in one practical session is almost impossible (we obviously mean impossible for you - nothing is impossible for us ACDC). Just keep in mind that you will need to use everything you will see on this part during this year.

Be careful : executing your code won't be enough for Rider to understand that you want to debug. You have to launch what is called the 'debug mode' by clicking on the small bug next to the 'run' button, in the upper left corner of the screen.



2.2.1 Breakpoints

Breakpoints are simply marked lines of code that will pause the debugger when running over them. They can be anywhere in your code, even in loops. A breakpoint in the body of a loop will make the debugger pause at each loop iteration. Inside a function, the debugger will pause every time it reaches this line, at each of these function calls. To put a breakpoint on a line, just click between the number of the line and the beginning of this line. A red dot will appear and the line will be slightly highlighted in red.



To remove a breakpoint, you just have to click on the red dot. To deactivate it without removing it, you need to right-click on it and deactivate the option *(*Enabled). We also advise you to launch the debug mode with the keyboard shortcut **<Shift>** + **<F9>**.

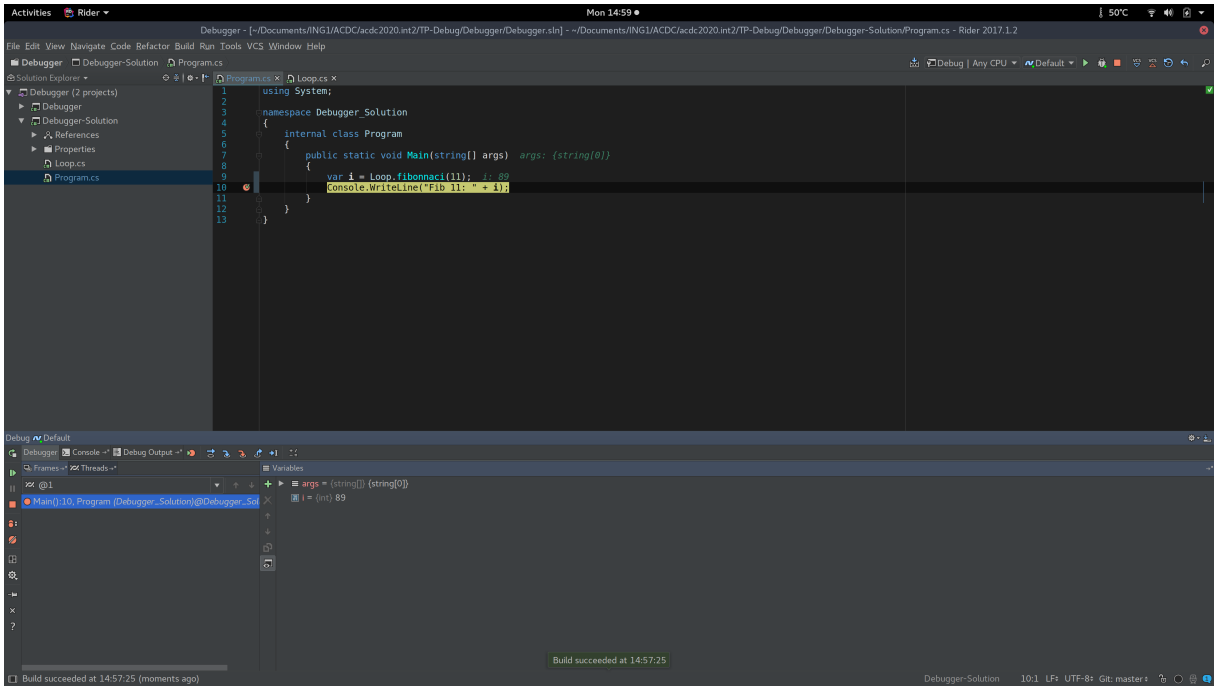
2.2.2 Steps

Pause the program, and then what ? Once the program has paused, it's your turn to play. There exist a lot of different ways to debug step by step after a breakpoint. For example, let's say that the breakpoint is on the line of the initialization of a new variable by the return value of a function call. We know that the bug is somewhere on this line, or following it. To move forward inside the body of the function, we do what is called a *(*step-in). To move forward, without entering the function, we make a *(*step-over). If we are already inside a function, and if we want to jump after the end of this function, we make a *(*step-out). With these three steps, it becomes easy to move in the code. But there are more ! Here is a list of possible operations in Rider :

— **Step Over** : Execute the current line and go to the next one (without stepping into the function if there is a function call in the current line) : <F8>
— **Step Into** : Execute the current line (and step into the function if there is a function call, otherwise go to the next line) : <F7>
— **Force Step Into** : Force the step into a function you don't have the right to step into (there are some) : alt+shift+<F7>
— **Step Out** : Continue the execution of lines until the end of the current function and go back to the calling method : shift+<F8>

2.2.3 Illuminati are watching you

Now that you know how to move in your code, you may need to know the value of your variables during debugging. Rider does that for you already. It will automatically add the value of the current variables in comments at the end of lines. If you want more details about a variable or an object, you just have to click on the variable in the list that is displayed at the bottom of your windows in debug mode.
You can also 'watch' a variable.

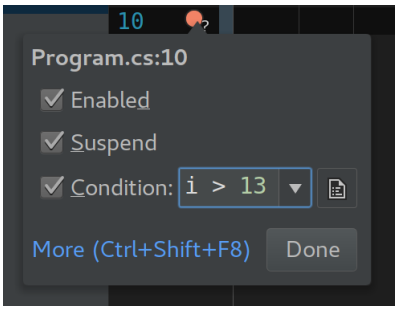


2.2.4 Conditional Breakpoints

It is really nice to be able to do all of that but if my bug is located on the 1000000th iteration, am I going to click a million times on continue ? If you are trying to find an answer to that, you are looking for the word 'no'.
There exists a way to do that easily, without getting a cramp in your finger, or even in your brain the same time. It is called conditional breakpoints. As the name suggests, you can put conditions on the activation of a breakpoint.

```
1  for (var i = 0; i < 42 000 000; i++)
2  {
3      ...
4      //bug here if i > 13 370 000
5      ...
6  }
```

If you look at the previous code, you may want to write an if statement inside the loop with a condition as 'i > 13 370 000' and then put a breakpoint inside it, or simply put a conditional breakpoint that pauses the program if 'i > 13 370 000'. Guess which way will be easier.



2.3 That's all folks

Voila, your turn now.

3 Exercises

3.1 Loops

3.1.1 You are a natural

Let's begin with simple things, and neatly done. You have to display every natural integer from 1 to n, with spaces between numbers but no space after the last number.

```
1 public static void Print_Naturals(int n);
```

3.1.2 Optimus Prime

You have to print the list of prime integers between 1 and n. Once again, with spaces between numbers but no space after the last number.

```
1 public static void Print_Primes(int n);
```

3.1.3 Always Fibonacci

Here, you need to reproduce the Fibonacci sequence. It is a sequence in which every term is the sum of the two previous terms. Of course, you absolutely need to implement the iterative version, as it is a lot faster than the recursive version that you already made. We will test your function on pretty big numbers. Little reminder : first terms are 0, 1, 1, 2, 3, 5, 8, and 13.

$$F(n) = F(n-1) + F(n-2)$$

```
1 public static long Fibonacci(long n);
```

3.1.4 Again Factorial

You need to rewrite the factorial function in its iterative version. We remind you that a 0 ! is 1.

$$n! = \prod_{1 \leqslant i \leqslant n} i = 1 * 2 * 3 * ... * (n-1) * n$$

```
1 public static long Factorial(long n);
```

3.1.5 I'm Stronk

You have to code a function that prints every 'strong' integer, between 1 and n. Those are integers that verify that the sum of the factorials of each of its numbers is equal to itself. An example would be 145 because $1! + 4! + 5! = 145$.

```
1 public static void Print_Strong(int n);
```

3.1.6 Square Root

First, you need to rewrite the function that gives the absolute value of a float given as an argument.

```
1 public static float Abs(float n);
```

By using the previous function, you have to write a function that gives the square root of the number given as an argument. You will need the Newton [1]'s method.

```
1  public static float Sqrt(float n);
```

3.1.7 Powerrrrrrr ! ! ! !

You have to rewrite the power function in its iterative version, where a is to the power of b.

```
1  public static long Power(long a, long b);
```

3.1.8 Vic-tree is ours

You have to code a function that prints a tree in the console. The n corresponds to the height of the tree. If n > 3, it needs 2 lines of trunk, or else it only needs one.

```
1  public static void Print_Tree(int n);
```

```
1  // n = 3
2     *
3    ***
4   *****
5     *
6
7  // n = 5
8        *
9      ***
10    *****
11   *******
12  *********
13       *
14       *
```

3.2 Debugging

For these exercises, an archive containing the code to debug is provided. Each exercise contains one or more bugs. We ask you to use the debugger to find them, but if you don't, you will need a lot more time to find them. Once you have found the bugs, correct them and add the project in your submission. To verify if you have correctly used the debugger, you have to explain in your README what and where the bugs are, how you found them, and how you corrected them. Warning : if this work has not been done, you will be graded 0 on this part.

These violent deadlines have violent ends !

1. https ://math.mit.edu/ stevenj/18.335/newton-sqrt.pdf