TP C#3: LOOPS AND THE DEBUGGER

1 Guidelines for handing in

At the end of this tutorial, you have to submit an archive that respects the following architecture:

```
rendu-tp-login_x.zip
|-- login_x/
    |-- AUTHORS*
    |-- README
    |-- TPC3.sln*
    |-- TPC3*
        |-- Everything but bin/ and obj/
```

Of course, you must replace "`login_x`" with your own login. All the files marked with an asterisk are *MANDATORY*.

Don't forget to verify the following before submitting your work:

- The `AUTHORS` file must be in the usual format (a *, a space, your login and a line feed).
- No `bin` or `obj` directory in your project.
- **The code must compile !**

2 Introduction

2.1 Goals

The main goal of this practical as you would imagine is to manipulate Visual Studio's debugger. It's a powerful tool that you'll intensively use this year. A special care was taken to explain basic concepts when making this subject, but please note that the links given given in the foot notes are good supplements to the course given today. At the end of this session, we want you to be autonomous when debugging your programs even though they may seem correct at first. For this session do not focus on the code, on debugging even if you don't understand the code that's done deliberately to force you to use the debugger instead of your brain.

In order to test your you knowledge, and review loops, we gave you two implementation exercises, using the debugger may help...

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?
> – Brian Kernighan

3 Course

3.1 The debugger

What's the purpose of this tool? If the name isn't already clear enough, it allows you to fix bugs in your program. Imagine that you have access to all machinery from the inside. It is now possible to follow the program's execution *at your own pace*. Here is a non-exhaustive list of what you'll be able to do at the end of this session:

- Pause your programm during runtime.
- Show the contents of the variable at a given time.
- Change the content of these variables at runtime.
- Change the behavior of your program at runtime.
- Go back a few steps in the execution.
- Show a list of the unctions called to get to where you are now in the program.
- ...

This list isn't exhaustive at all, but trying to present all those in one session isn't reasonable. Please note, however, that this list covers at least 95% — if not all — of the possible cases that you'll see this year.

3.1.1 Breakpoints

Imagine your program is a book that your processor will read in a non-linear fashion, as you'd do with a dictionnary. A breakpoint is like a bookmark in this book, that tells the processor to pause reading when it reaches it. You'd put a breakpoint onto a function declaration to pause the program every the function is called, onto the **while** keyword in a loop to pause the program at each iteration. To put a breakpoint simply click in the left margin, in front of the choosen line. A red circle is now displayed to let you know that a breakpoint has been placed (Figure **??**).

To deactivate it while keepin git in place, right-click on it and select the corresponding action. The circle should now be white. When your **breakpoint** is set launch your program as always with the green "play" arrow, you can use the shortcuts *<F5>* or *Shift+<F5>* if you where already in execution and wanted to restart from beginning. You now understand that *you were using the debugger right from the beginning*, without knowing it, because no breakpoint was set.

3.1.2 Seek'n Destroy

Alright, you know how to stop the program, but what's the purpose of all of this? The goal behind putting a **breakpoint** in your code is to take the time to analyze the program's context[1] and do what we call *step by step execution*. When you're not in the debugger all the execution is done in a fraction of a second depending on the complexity of the calculatons you're doing[2] and the speed of the computer[3]. Thanks to the **step-by-step** execution no matter what kind of computer you have, you can decide when to execute the next instruction. If you look at the Figure **??** you can see that the breakpoint is set on the 21st line. If we advance once (to the

[1] the function taht is being executed, the values of the local variables
[2] Algorithm complexity
[3] Clock speed, amount of RAM...

next step) the debugger will execute what is on the current line *then* advance to the next one and wait. However, what's going on if the line isn't a simple assignment like we have here but a function call? In fact it's up to you to choose how the debugger will react next. Any good debugger let you choose three types of jump to the next instruction.

- **step over**: Execute the current line and place the cursor to the next one without displaying the execution of the function that was on the previous line.
- **step into**: It's the opposite, this time the debugger bring you into the code of the called function. It takes care of context changes... In short the debugger makes your life easier so that you can focus on the most important: fix bugs.
- **step out**: Continue the exécution until the end of the current function and break when returning to the calling function.

It's up to you to know when it's better to step over a piece of code or not, depending on what you seek or have already checked. If you want to step over press the key *<F10>*, while the step into is done with the key *<F11>*. There are some graphical buttons corresponding to these two shortcuts, but they're voluntarily omitted here, because it's not a good habit to slow down debugging with useless clicks here and there. We already do it often enough on windows.

Finally, please note that if you think you have gone too far in the function, the debugger save each context's instructions. Thus you can go back a few steps by dragging the arrow on the breakpoint when the program is paused.

3.1.3 Smile you're on CCTV

Now that you know how to go through your code at runtime, it would be good to look at what's hiding beneath our variables and spot potential issues [4]. There are several methods, each with advantages and drawbacks. Please note that it's difficult to access variables that are not in the current scope of execution. So please don't try to accept values stored inside variables of a previous and terminated function, it does not make sense.

- naive : Place your mouse over a variable and wait a little. There is now a popup that contains the variable's content. Do it as often times as you need... Or use the post-it method.
- post-it : When you put your mouse over the previous pop-up you can see a pin, click on it to make pop-up persistent.
- watch[5] : You can see them as variables that you can create on the fly at runtime. You can assign them to a variable but this is a limited use of them (i.e this is the post-it method in another window), or you can create more complex variables that are not directly found in the code. A useless but educational example would be creating a watch that display the sum of two other variables "a" and "b" that are available in the body of a function, it avoid us to mentally computing the sum every time. the sum each time.

[4] this is a problem: 42 / 0
[5] MSDN: Using the Watch Window

3.1.4 Maybe, maybe not

Perhaps you're already wondering how to put breakpoints efficiently in big loops. What if your program become unstable after the 1337th iteration? There are two methods, the first one needs a lot of patience and some solid fingers to press 1336 times the *<F5>* key. The second one is far more interesting and allows you to put a conditional breakpoint. Right-click om the desired breakpoint and select the conditional statement option. In the following example, if we want to skip the first 1336 iterations, the condition to enter in the input box would be i == 1336.

```
int i = 0;
while (i < 4242)
{
  if (i > 1336)
    // BUG HERE
  i++;
}
```

3.1.5 You're the only master on board

The window named "locals" displays an overview of all the local variables of the current context in real time. You can modify any variable by changing the correct line in the window locals (Figure **??**).

It can be very useful to modify the behavior of your program at runtime.

3.2 Figures

4 Exercices

4.1 Exercise 1: The debugger, basics

Open the Visual Studio project given for this session. In the **main** function of your projects you can find commented lines. The first isn't commented so that you can start this exercise. We recommend you to enable only the necessary one so that you should avoid analyzing too much code step by-step-if you set the breakpoint to early or if you forgot to delete unused breakpoint that were here for the previous function. Let's start: put a breakpoint on the first line and run your program.

```
Exercises._1___Basics.ex1.run();
//Exercises._1___Basics.ex2.run();
//Exercises._1___Basics.ex3.run();
```

Your goal is to explain the behavior of functions of each exercise in the corresponding header. Here is the header of the first exercise:

```
/// <summary>
/// COMPLETE ME
/// </summary>
public static unsafe void run()
{
    int my_nbr = 41;
    utils.fex1(&my_nbr);
    Console.WriteLine("Function ex1 terminated.");
}
```

You should understand what "fex1" does, and as you might have already understood neither the code nor the function names are here to help you. Last reminder that will also be valid for the rest of the session unless the subject excplicitly tells the contrary, is that **you're not allowed to modify any bytes of that code**. If you want to modify the content of a variable for testing purposes, do it at runtime, and refer to the section **??**.

4.2 Exercice 2: The debugger, intermediate level

In this series of exercices the goal is not simply understanding the functions behavior, but debugging them in the proper sens of the term. These functions do not run as they should, it's up to you to find where the bug is and correct it. You are *allowed to add only one line* of code inside these functions in order to fix them.

Hint

> You want to be comfortable with string[a] basics to find the bug more easily. Last reminder, don't lose time on code syntax, the goal for you is to solve the exercise with the help of the debugger, not with your brain.

[a] String datatypes

```
18  private void button1_Click(object sender, EventArgs e)
19  {
20      int LetterCount = 0;
21      string strText = "Debugging";
22      string letter;
23
24      for (int i = 0; i < strText.Length; i++)
25      {
26          letter = strText.Substring(1, 1);
27
28          if (letter == "g")
29          {
30              LetterCount++;
31          }
32      }
33
34      textBox1.Text = "g appears " + LetterCount + " times";
35  }
```

Figure 1: Breakpoint on line 21, the program is stopped

Locals

Name	Value	Type
self	{Loops.ConsoleApp}	Loops.ConsoleApp
Countries	nil	System.Collections.Generic.IEnumerable<Loops.Country>
Static Members	{Loops.ConsoleApp}	Loops.ConsoleApp
i	0	System.Int32
ind	0	System.Int32

Figure 2: The “locals” window

4.3 Exercice 3 : The debugger, advanced level

Since everyone survived, here are some exercises that require deeper anderstanding of what's going on inside your program.

4.3.1 Crypto-Box

A ciphered string is hidden into the memory, retrieve the original one without adding code. You must specify the deciphered string into your README file.

```
ciphered = "bw wvu urr'k demd hfz xf vfbl, ak oirlk iuon ksn mf blblj";
key = "tryharder";
```

4.3.2 The Bomb: Bonus

By playing with your code you have triggered a deadly bomb that will force you to use all your knowledge in order to defuse it and save the city.

Your goal is to input defuse codes in a correct order to disarm the bomb. You're not allowed to modify the code. You must entirely specify your approach to defuse the bomb in your README file.

Hint

> At each level the bomb gives you a hint about the type of input it wait for. This depend on the level of the bomb.
>
> - Digit [0-9] : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
> - string : **String Class**

Please do not hesitate to give feedback, either good or bad criticism on the part dealing with the debugger by email:
To: Naam <elatma_n@epita.fr>
Subject: [SUP][TPC3] feedback debugger

4.4 Exercice 4: Happy

> In mathematics, a natural number is happy if, when we compute the sum of its digits squared in its base 10 representation, the sum of the squared digits of the result and so on, we get to the number 1. – Wikipedia

For instance, 19 is a happy number:

- $1^2 + 9^2 = 1 + 81 = 82$
- $8^2 + 2^2 = 64 + 4 = 68$
- $6^2 + 8^2 = 36 + 64 = 100$
- $1^2 + 0^2 + 0^2 = 1 + 0 + 0 = 1$

The final goal of this exercise is to implement the function `public static void checkHappy();` that will ask the user to input an integer, and then will display a message saying if the number is happy or not. The function must keep asking for numbers to check until the user enters 0.

Here is an example of the output:

```
Enter an integer (0 to stop): 69
69 is not happy...
Enter an integer (0 to stop): 1337
1337 is happy!
Enter an integer (0 to stop): 42
42 is not happy...
Enter an integer (0 to stop): hello!
Enter an integer (0 to stop): 0
```

The `checkHappy` function is based on 3 other functions that you have to implement first. The first of those 3 functions will simply ask the user to enter an integer. But what happens if the user is an idiot and enters "hello !"... ? In this case, you must ask them again, until they enter a correct valid[6].

Your function must take exactly one parameter of type `string`, that will be printed every time before asking for a number. The prototype of the function must be `public static int readInt(string prompt);`.

```
int num = readInt("Enter an integer: ");
Console.WriteLine("I love " + num + " too!");
```

```
Enter an integer: Hello!
Enter an integer: 42.0001
Enter an integer: 1337
I love 1337 too!
```

Then, write the function `public static int sumOfSquaredDigits(int n);` that computes the sum of the squared digits. For instance, with $n = 24$, we have $2^2 + 4^2 = 20$.

When you're done, you can move on to the function `public static bool isHappy(int n);`. It returns **true** if n is a happy number, and **false** otherwise.

Hint

| You should explore the class List<T>...

You are now able to implement the function `checkHappy` described in the beginning of the exercise. Its prototype must be `public static void checkHappy();`.

[6] You might be interested in the function `Int32.TryParse()`

4.5 Exercice 5: Pascal's triangle

As you already know, Pascal's triangle is a triangular array that is used to retrive binomial coefficients. The coefficient on the ith line, jth column is the sum of the coefficients of the j - 1th and the jth columns of the i - 1th line.

Here is what the beginning of Pascal's triangle looks like:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

You will implement a function that will take an integer **n** as then only parameter and must print the **n** first lines of Pascal's triangle. Its prototype must be `public static void pascal(int n);`.

Bullshit is strong with this one...