Practical 4 : Recursion is a lie

1 Handing in guidelines

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
    |-- my_refs
        |-- my_refs.sln*
        |-- my_refs*
                |-- Everything but bin/ and obj/
    |-- my_arrays
        |-- my_arrays.sln
        |-- my_arrays
                |-- Everything but bin/ and obj/
    |-- my_formulaone
        |-- my_formulaone.sln
        |my_formulaone
                |-- Everything but bin/ and obj/
```

Of course, you must replace "`login_x`" by 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 Objectives

During this tutorial, you will learn to do the following :
— Passing arguments by reference and useful applications.
— Using function parameters in a smart way (using the *params* keyword).
— Using single or multi-dimensional arrays.

3 Lesson

3.1 Reference passing

3.1.1 Prerequisites

Usually, when parameters are given to a function, the modifications made to their values are done locally. To put it simply, once the function stops, the variables that were passed to it will have the same value as before entering the function. For instance :

```
/* The factorial function :
** calculates n!, and puts the result in res
*/
static int fact(int n, int res)
{
    res = 1;
    for (int i = 2; i <= n; i++)
        res *= i;
    return res;
}

static int Main(string[] args)
{
    int res = 0;
    Console.WriteLine(res);

    // Displays the result of res!, then res
    Console.WriteLine(fact(4, res) + ", " + res);
    Console.WriteLine(fact(2, res) + ", " + res);
    Console.Read();
    return 0;
}
```

The result will be :

FIGURE 1 – Console display

What we see here is that the value of *res* doesn't change in the *Main* function, even though we change it in the *fact* function. This is because we do not actually manipulate *res*, but a local copy of it created when calling the *fact* function in *Main*. We modify but a copy of this variable, that will be destroyed once the function ends (after the *return* keyword).

3.1.2 Using references

We modify the *fact* and *Main* functions as follows :

```
/* The factorial function :
** calculates n!, and puts the result in res
*/
static int fact(int n, ref int res)
{
    res = 1;
    for (int i = 2; i <= n; i++)
        res *= i;
    return res;
}

static int Main(string[] args)
{
    int res = 0;
    Console.WriteLine(res);

    // Display the result of res!, then res
    Console.WriteLine(fact(4, ref res) + ", " + res);
    Console.WriteLine(fact(2, ref res) + ", " + res);
    Console.Read();
    return 0;
}
```

The result becomes :

FIGURE 2 – *res*'s value does change this time

What we see here is that the value of *res* was modified by the function. This, folks, is the power of reference passing. It tells our program "Nah, I don't want to make a copy of this value, I want it changed and you better do it well".

3.1.3 Usefulness

It might not be obvious at first sight, for you could perhaps say "Wait mate, why don't we just return the value ?". This is true with the factorial function which returns one only value. But what if we want to return more than one ? For instance, what if we wanted to swap the value of two variables ? Just keep this solution in mind when you want to modify several values in the same function (you will find some useful applications in the exercises).

3.2 Arrays

3.2.1 Vectors

A vector is an array with one dimension. In C#, a vector is declared using the following syntax "<type>[] <name>;".

Example :

```
int[] vectorint; //vectorint is a vector containing ints
char[] vectorchar; //vectorchar is a vector containing chars
bool[] vectorbool; //vectorbool is a vector containing booleans
```

The instantiation of a vector uses the following syntaxe "<vector> = new <type>[<x>];". This way, we know that the vector contains x elements (x must be more than or equal to zero). It is a common usage and highly recommended to instantiate it at the same time as the declaration is done.

Example :

```
int[] vectorint;
vectorint = new int[4];
// vectorint is a vector containing 4 ints

char[] vectorchar = new char[2];
//vectorchar is a vector containing 2 chars

bool[] vectorbool = new bool[42];
//vectorbool is a vector containing 42 booleans
```

The initialisation of a vector is done using the syntaxe "<vector> = new <type>[<x>]<elt1>, <elt2>, <...>, <eltx>;" or simply "<vector> = <elt1>, <elt2>, <...>, <eltx>;". It is observable here that, in the second case, it isn't necessary to explicitly instantiate the vector. In C#, the instantiation will be implicit in this case (thank kind language). The first syntax is more useful when the size of the vector is known and only the firsts elements need to be initialised. Again, it is common and highly recommended to initialise it at the same time as the declaration is done.

Example :

```
int[] vectorint = new int[4]{1, 3, 3, 7};
/* vectorint is a vector containing 4 integers
** in the following order: 1, 3, 3 and 7
*/

char[] vectorchar = {'C', '#'};
/* vectorchar is a vector containing 2 characters
** in the following order: 'C' and '#'
*/

bool[] vectorbool = new bool[42]{true, true, true, true, true, true};
/* vectorbool is a vector containing 42 booleans
** for wich the 6 first values are initialised with true
*/
```

Accessing an element in a vector is done with the syntax "<vector>[<x>];". This will get the (x - 1)th element in the vector. The first index in an array is 0. An index x must be 0 <= x < n with n the number of elements in the vector.

Example :

```
/* We use the three vectors declared and initialised before */

vectorint[0]; // returns 1

vectorchar[1]; // returns '#'

vectorbool[3]; //returns true
vectorbool[4] = false; /* saves false in
                        ** vectorbool at index 4
                        */
```

3.2.2 Multidimensional array

A multidimensional array is an array of n dimensions with n > 1. The manipulation of a multidimensional array is really close to the one of a vector :
- "<type>[,] <name>;" to declare it.
- "<vector> = new <type>[<d1>, <d2>, <...>, <dx>];" to instantiate it, with x the number of dimensions and dn the size of the array in the given dimension.
- "<vector> = <elt11>, <elt12>, <...>, <elt1d1>, <elt21>, <elt22>, <...>, <elt2d2>, <...>, <eltx1>, <eltx2>, <...>, <eltxdx>;" to initialize it. It can still be used with syntax that allows the explicit instantiation with the initialization. - "<vector>[<p1>, <p2>, <...>, <px>];" to access a value with x the number of dimensions and 0 <= pn < dn.

Example :

```
double[,,,] arrdouble = new double[42, 69, 23, 10];
/* arrdouble is a 4-dimensional array of doubles
** and each dimension is respectively of size 42, 69, 23 and 10
*/

string[,] arrstr = new string[4, 2]{{"You", "shall not"}, {"pass"}};
/* arrstr is a  2-dimensional array of strings
```

```
** and each dimension is respectively of size 4 and 2
*/

float[,] arrfloat = {{0.5, 1.0}, {47., 10.}};
/* arrfloat is a  2-dimensional array of floats
** each of size 2 and initialized with 0.5, 1.0, 47. and 10.
*/

arrstr[0, 0]; // returns "You"
arrstr[0, 1]; // returns "shall not"
arrstr[1, 0]; // returns "pass"
arrstr[1, 1] = "!!"; /* saves "!!" in tabstr
                     ** at position (1, 1);
```

3.2.3 Array of arrays (jagged arrays)

An array of arrays is an array containing arrays. Its manipulation is similar to the one seen before. A simple way to understand and envision its behaviour is to consider the object array as any known type/object.

Example :

```
/* In order to simplify the concept, we re-write
** the multidimensional arrays of the previous example
** as jagged arrays
*/

double[,][][] arrdouble = new double[42, 69][23] [10];
/* arrdouble is an array of arrays of arrays
** of doubles each containg in order:
** 10 arrays of arrays
** 23 2-dimensional arrays of doubles
** 2 dimensions respectively of size 42 and 69
*/

string[][] arrstr = new string[4][2]{new string[2]{"You", "shall not"},
                                     new string[2]{"pass"}};
/* arrstr is an array of arrays of strings
*/


arrstr[0][0]; // returns "You"
arrstr[0][1]; // returns "shall not"
arrstr[1][0]; // returns "pass"
arrdouble[0, 12][22][3] = 32.0;
/* saves 32 in arrdouble at
** index (0, 12, 22, 3)
*/
```

3.2.4 Going further

The keyword "params" in C# allows passing an undefined number of parameters of a known type to a function. This tool allows the passed parameters that'll be given to the function to be grouped in an array and to be manipulated in an easy way. The call to the function can be made with an array of the expected type or with several parameters matching the expected types contained in the array or even no parameters
Here is an example of how to use "params" :

```
static int[] createarr(params int[] tab)
{
    return tab;
}

static void printarr(params int[] tab)
{
    //tab.Length returns the number of elements in tab
    for (int i = 0; i < tab.Length; i++)
        Console.Write(tab[i] + " ");
    Console.Write("\n");
}

static void Main()
{
    int[] my_arr = createarr(1, 3, 3, 7, 42);
    printarr(my_arr);
    printarr(1, 3, 3, 7, 42);
    Console.Read();
}
```

Output is :

FIGURE 3 – Printing in the console

4 Exercises

4.1 Exercise 1 : I luv maths

4.1.1 Addition

In this part, you have to implement the following function :

```
static void add(ref float result, float value);
```

This function takes as parameters two floats, adds them, and puts the result in the first parameter.

4.1.2 Substraction

In this part, you have to implement the following function :

```
static void sub(ref float result, float value);
```

This function takes as parameters two floats, substracts them, and puts the result in the first parameter.

4.1.3 Multiplication

In this part, you have to implement the following function :

```
static void mul(ref float result, float value);
```

This function takes as parameters two floats, multiplies them, and puts the result in the first parameter.

4.1.4 Division

In this part, you have to implement the following function :

```
static bool div(ref float result, float value);
```

This function takes as parameters two floats, divides them, and puts the result in the first parameter. The result shall be "*true*" if everything went OK or "*false*" in case of an error (division by 0).

4.1.5 Pawa !

In this part, you have to implement the following function :

```
static bool pow(ref float result, int p);
```

This function takes as parameters a float and an int, calculates "*result*" of power "*p* and puts the result in...result. This function returns *"true"* if everything went OK or *"false"* in case of a negative power or of zero of power zero, which is undefined.

4.1.6 Arithmetic

In this part, you have to implement the following function :

```
static void arit(ref float Un, float r, int n);
```

This function takes as parameters two floats and an integer and calculates the (n - 1)th term of the arithmetic sequence defined by the first term *Un* and the common difference *r*. It then puts the result in *Un*.

4.1.7 Geometric

In this part, you have to implement the following function :

```
static void geom(ref float Un, float q, int n);
```

This function takes as parameters two floats and an integer and calculates the (n - 1)th term of the geometric sequence defined by the first term *Un* and the common ratio *q*. It then puts the result in *Un*.

4.1.8 Refs are useful I swear !

In this part, you have to implement the following function :

```
static void swap(ref int a, ref int b);
```

This function takes as parameters two integers and swaps their values. Notice that, since you have to return two values, passing a reference is quite useful.

4.2 Enter the matrix

4.2.1 Simple way

You have to implement :

```
static int mintab(int[] tab, ref int min);
```

This function takes as parameters an array of ints and an integer. It will save in this int the minimum value found in the array and return its position in this array.

You have to implement :

```
static int maxtab(int[] tab, ref int max);
```

This function takes as parameters an array of integers and an integer. It will save in this integer the maximum value found in the array and return its position in this array.

4.3 Bubble sort

You have to implement :

```
static void bubblesort(int[] tab);
```

This function takes as parameters an array of ints and sorts it. You will have to use a bubble sort for this exercise. The function won't return anything(surprised ? ask your ACDCs to explain).

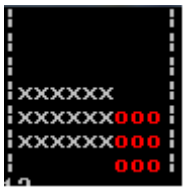
4.4 Exercise 2 : My tiny formulaone

The goal of this exercice is to apply your newly acquired knowledge on a real project. Your goal is the following :

— Implement a "string" to "matrix" convertor.
— Display said matrix on the console.
— Register the keys pressed by the user and react accordingly.
— Profit !

The final result will be a basic racing game displayed on your console. This will allow you to simply have fun with it, or to discover tougher subjects through the boni.

FIGURE 4 – Your glorious F1 dodging an obstacle single-handedly



4.4.1 Caml with you, always

You have to implement the following function :

```
static int[][] string_of_int_array(string input, int width);
```

Where "**input**" is the string you have to turn into a matrix and "**width**" is the width of said matrix. These two parameters are enough to create our matrix. Since we know how to get the length of a string (obviously...), and we have the width of the matrix. Given the fact that the length of the given string will always be a multiple of the width, we can easily find the matrix's height.

Warning : Note that we use *jagged arrays* (**type**[][]. They are different from the "traditional" two dimensionnal arrays (**type**[,]). Refer to the previous lesson and exercises.

4.4.2 Let's build a road

Now that we have an elegant way of building our tracks (you may adapt the previous function to create a *bool* matrix), we will now see how we can display them. However, our tracks might be bigger than the size of the console. displaying the whole track and then moving the car on it would then obligate us to scroll at the cost of precious APMs (or Actions Per Minute). The solution is not to make the car move but the whole track by displaying it bit by bit (by chunks of 3 lines for instance). If we want to do that, we have to keep track of our position on the track.
The last thing to do is to scale the matrix's size to the size of the display. Indeed, if we show each index on a single character, the display will look quite small. You will therefore have to represent each index by a block of 3x3 characters (just like in Figure 1).

You have to implement the following function :

```
static bool draw_road(bool[][] race, ref int where)
```

"**race**" is the matrix representing our track (the index is *true* if it's a road and *false* if it's a wall), and "**where**" is our position in the track. Note the use of a reference passing that allows

us to increment "**where**" in the *draw_road* function and not in the game's loop which we'll see later. The return value is "*true*" when you can display the "*where*" line and the two after that and "*false*" otherwise.

Warning : don't forget that you do not have to display one line but 3 (hence the return value).

4.4.3 Let's make it move, smoothly

In order to move the track, there is nothing easier ! You only have to create a loop in the "*Main*" function that goes on while "**draw_road**" is true.
If you launch your program now, you will notice a problem. The display is way too fast ! In order to fix this problem, you are allowed to use the "*Thread.Sleep*" function that pauses your program. In order to use it, just add "*using System.Threading*" at the beginning of your code.
If you launch your program one more time, you will notice that the display is not smooth. The blocks move by chunks of 3 lines at each step. We'd like them to move one at a time.
In order to do that, you have to divide "*where*" by 3 in your function "**draw_road**" (don't forget that we passed "*where*" by reference, we don't want it divided by 3 in the "*Main*" function). If you do that, your loop will do 3 times the steps. Now, you can use the result of "*where mod 3*" in order to display the blocks properly (fuzzy drawings are a programmer's best friend).

4.4.4 Where's the car ?

Now that you have a pretty display, one thing is missing : the car.
Since it's the track that moves, the car is only represented by a horizontal position in said track.
Modify your "*draw road*" as such :

```
static bool draw_road(bool[][] race, ref int where, int car)
```

You will make sure that the car is displayed (refer to Figure 1) and that the game stops ("*draw_road*" returns "*false*") once the car hits a wall.

4.4.5 Move the car

Now that we have our car, we want to make it move. In order to do that, we must retrieve the keys pressed by the user. We can not use the "*Console.Readkey*" function alone, for this function will block our program's execution until the user presses a key.
The solution is to use the boolean "*Console.KeyAvailable*" that turns true when a key is pressed. Once this boolean turns true, you may use "*Console.ReadKey*".

4.5 Bonii

Here are the bonii you can do :
— A more elegant/optimized sort than the bubble sort (Google is your friend).
— Pimp my output (with colors for instance).
— An AI for your formulaone.
— Random playable (there must exist at least one way to finish it) track generation.

Warning : if you decide to implement these boni, you will be asked to explain your code. So don't just go and copy/paste a quick sort found on Google.

The Bullshit is strong in this one...