TP C#4

1 Submission rules

You must submit a zip file with the following architecture :

```
rendu-tp-login.zip
    |-- rendu-tpcs4-login/
        |-- AUTHORS
        |-- README
        |-- Parameters/
            |-- Parameters.sln
            |-- Parameters/
                |-- everything except bin/ and obj/
        |-- Arrays/
            |-- Arrays.sln
            |-- Arrays/
                |-- everything except bin/ and obj/
        |-- Connect4/
            |-- Connect4.sln
            |-- Connect4/
                |-- everything except bin/ and obj/
```

You shall obviously replace *login* with your login.

Don't forget to check the following rules before to submit your practical :

- The file AUTHORS must be in the usual format : * *firstname.lastname$* where the character '$' is a line break.
- You must follow the subject **scrupulously**, especially you must **respect the prototype** of each function.
- No folder bin and/or obj in the project.
- **The code must compile !**

2 Introduction

2.1 Objectives

In this practical, you will see the following notions :

- Passing by reference
- Other method parameters (keywords *params* or *out*)
- Arrays manipulation (One or more dimensions)

3 Course

3.1 Passing by reference

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 example :

```
1  /*
2  ** Return Factorial of n
3  */
4  static int fact(int n, int res)
5  {
6      res = 1;
7      for (int i = 2; i <= n; ++i)
8          res *= i;
9      return res;
10 }
11
12 public static void Main (string[] args)
13 {
14     int res = 0;
15     Console.WriteLine (res);
16
17     //Display the value of fact then the value of res
18     Console.WriteLine(fact(4, res) + ", " + res);
19     Console.WriteLine(fact(2, res) + ", " + res);
20 }
```

Now, let it go :

```
0
24, 0
2, 0
```

FIGURE 1 – Console display

You can notice that the value of *res* didn't change even though the value of *res* is changed in the function *fact*. Actually, you do not manipulate the variable *res*. In fact, a local copy is created when a variable is given as a parameter to the function. Therefore, you don't manipulate the value of *res* but the value of its copy. This copy is destroyed at the end of the function.

3.1.2 Using passing by reference

Let's see what happens if we change the function *fact* as follows :

```
/* This time we add the keyword ref so that we can use
** passing by reference
*/
static int fact(int n, ref int res)
{
    res = 1;
    for (int i = 2; i <= n; ++i)
        res *= i;
    return res;
}

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

    /* You MUST write ref before res so that res is passed
    ** by reference because the prototype of fact asks for it
    */
    Console.WriteLine(fact(4, ref res) + ", " + res);
    Console.WriteLine(fact(2, ref res) + ", " + res);
}
```

This time the result becomes :

```
0
24, 24
2, 2
```

FIGURE 2 – Console display

We can see that *res* is modified by the function and we can see this after the call to *fact*. This is because of passing by reference. It's very powerful. To use it in C# you have to use the keyword *ref*. With passing by reference we can modify the variable given in parameters in the function. We ask to the function to not do a local copy so that we can modify it in the called function.

3.1.3 Usefulness

It's not obvious to see its utility at first glance. We can think that only returning the value is efficient. It's true that, for example, with the *fact* function we can only return the value. It's only true with this case but in any other cases the *ref* keyword must be used.
Do you see the utility of passing by reference ? Let's see some examples : Imagine that we want to save more than one value, to manipulate several variables or to save the modifications made on all the variables. For us it's impossible to make a function for each value. So we use passing by reference with the keyword *ref* in order to modify all the values in the same function. With it we save in clarity and we avoid some useless functions calls.

3.2 Arrays

3.2.1 Vectors

A vector is an array with only one dimension. In other words, a vector is a contiguous list. In C# a vector is declared with the syntax "TYPE[ ] name ;".

Example :

```
1 int[] vectorint; //vectorint is an integer vector
2 char[] vectorchar; //vectorchar is a character vector
3 bool[] vectorbool; //vectorbool is a boolean vector
```

Instantiation of a vector is made by the syntax "VECTOR = new TYPE[X] ;". With this syntax we know that the vector has X elements. (X must be positive 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 :

```
1 int[] vectorint;
2 vectorint = new int[4];
3 //vectorint is an integer vector of length 4
4
5 char[] vectorchar = new char[2];
6 //vectorchar is a character vector of length 2
7
8 bool[] vectorbool = new bool[42];
9 //vectorbool is a Boolean vector of length 42
```

Initialisation of a vector is made with the syntax "VECTOR = new TYPE[X]{ ELT1, ELT2, ..., ELTX } ;" or easier "VECTOR = { ELT1, ELT2, ..., ELTX } ;". We can see that in the second case it's not necessary to instantiate explicitly. In this case, instantiation is implicit. The first syntax is useful when we already know the length of the vector but we only want to initialise the first items. Again it is a common usage and highly recommended to instantiate it at the same time as the declaration is done.

```
int[] vectorint = new int[4]{1, 3, 3, 7};
/* vectorint is an integer vector of length 4
** that contains in this order 1, 3, 3 and 7
*/

char[] vectorchar = {'C', '#'};
/* vectorchar is a character vector of length 2
** that contains in this order 'C' and '#'
*/

bool[] vectorbool = new bool[6]{true, true, true, true, true, true};
/* vectorchar is a Boolean vector of length 42
** where the values are assigned to true
*/
```

Access to an element in a vector is made with the syntax "VECTOR[X];". Thanks to this we can access to the (x+1)th element of the vector. The index in an array is in base 0. We must have $0 <= x < n$ with n the number of elements in the vector.

```
/* We reuse the vectors delared and initialised before */

vectorint[0]; //return 1

vectorchar[1]; //return '#'

vectorbool[3]; //return true

vectorbool[4] = false;
/* We put the value false in vectorbool at the index 4 */
```

To finish we give you a little hint, "VECTOR.Length" is the syntax to know the length of a vector.

3.2.2 multidimensional arrays

A multidimensional array is an array with n dimensions (n > 1). Manipulate a multidimensional array is almost the same as a vector :

- "TYPE[,] name ;" for declaration
- "VECTOR = new TYPE[D1, D2, ..., DX] ;" for instantiation, with x the number of dimensions and DN the length of given dimension in the array.
- "VECTOR = {{ELT1D1, ELT2D1, ..., ELTXD1}, ..., {ELT1DX, ELT2DX, ..., ELTXDX}} ;" for initialisation. We can always use the syntax to do an explicit instanciation when we initialise the array.
- "VECTOR[P1, P2, ..., PX] ;" to access to a value with X the number of dimensions and 0 <= PN < DN

Example :

```
double[,,,] arraydouble = new double[42, 69, 23, 10];
/* arraydouble is an double array with 4 dimensions
** where the dimensions have length:
** 42, 69, 23 and 10
*/

string[,] arrstr = new string[2,2]{{"I love", "my"}, {"ACDC", "!"}};
/* arrstr is a string array with 2 dimensions
** where the dimensions have length 2
*/

double[,] arr = {{0.5, 1.0}, {47.0, 10.0}};
/* arr is a double array with 2 dimensions. Each dimension has length
** 2 and are initialised to 0.5, 1.0, 47.0, 10.0
*/

arrstr[0, 0]; //return "I love"
arrstr[0, 1]; //return "my"
arrstr[1, 0]; //return "ACDC"
arrstr[1, 1] = "!!";
/* We put "!!" in arrstr at the index (1, 1) */
```

Like the vectors "VECTOR.Length" will give the number of elements of the array. Maybe, you only want the elements of one dimension. In that case you can use "VECTOR.GetLength(N)" where N is the asked dimension(in base 0).

3.2.3 Arrays of arrays

An array of arrays is an array where the elements are arrays. To manipulate an array of arrays is the same as an multidimensional array with the syntax of the vectors. You can see it when you transform the array with a known type. It becomes a simple array. Let's see an example :

```
/* To simplify your comprehension, we rewrite
** the old arrays with the arrays of arrays
*/

double [,][][] arrdouble = new double[42, 69][23][10];
/* arrdouble is an array of arrays of arrays of double
** We can see it like an array that contains 10 arrays that contain
** 23 arrays of two dimensions
** Those two dimensions have respectively length of 42 and 69.
*/

string[][] arrstr = new string[2][2]{new string[2]{"I love", "my"},
                                     new string[2]{"ACDC", "!"}};
// tabstr is an array of arrays of string

arrstr[0][0]; //return "I love"
arrstr[0][1]; //return "my"
arrstr[1][0]; //return "ACDC"
arrdouble[0, 12][22][3] = 32.0;
// We put 32 in arrdouble at the index (0, 12, 22, 3)
```

3.3 Method Parameters

3.3.1 out

Now that you are an expert with passing by reference, we can go further !
Let's begin with the keyword *out*. This keyword is almost the same as *ref* but with a little difference. When we put the keyword *out* we just say that the variable will be an exit/return variable. So this variable will be the same as a *return*. The variable need to be modified in all cases of the function because it's like *return*.

The biggest difference between the keyword *ref* and *out* is that for *ref* the variable needs to be initialised before the function. Whereas with the keyword *out* it's not mandatory because we will modify in all the cases the variable in the function. To help you understand, this notion let's see the example of the function `Int32.TryParse` (You can see the MSDN for more information) which converts a string to integer with the keyword *out*. If the string is not an integer the value of the integer will be 0. The function returns a boolean to know if the string is an integer.

```
public static void Main (string[] args)
{
    String[] strings = { "RTX", "42", "-1984", "10/02", "-322,546",
                         "   +9000   ", "0xFE" };

    for (int i = 0; i < strings.Length; ++i)
    {
        int number;
        if (Int32.TryParse (strings[i], out number))
            Console.WriteLine ("Conversion of {0} to {1}", strings[i],
                               number);
        else
            Console.WriteLine ("Conversion of {0} failed", strings[i]);
    }
}
```

Let's see the result :

```
Conversion of RTX failed
Conversion of 42 to 42
Conversion of -1955 to -1955
Conversion of 10/02 failed
Conversion of -322,546 failed
Conversion of    +9000    to 9000
Conversion of 0xFE failed
```

FIGURE 3 – Console display

3.3.2 params

The keyword "*params*" in C# lets the user give an undefined number of parameters of the same type to the function. This keyword allows to consider all the parameters as an array given to the function. This allows us to manipulate all the parameters easily. The call to the function can be directly made with an array of a known type with several parameters or with no parameters.

Let's see an example :

```
1  static void DisplayArgs(params int[] args)
2  {
3      for (int i = 0; i < args.Length; ++i)
4          Console.Write (args [i] + " ");
5      Console.WriteLine ();
6  }
7
8  public static void Main (string[] args)
9  {
10     int[] array = { 1, 25, 22, 24, 42, 87 };
11     DisplayArgs (array);
12     DisplayArgs (1, 25, 22, 24, 42, 87);
13 }
```

Let's see the result :

```
1 25 22 24 42 87
1 25 22 24 42 87
```

FIGURE 4 – Console display

4 Exercises

4.1 Exercise 1 : Parameters

All the following functions must be done in the project **Parameters**.

4.1.1 Swap

You have to implement the following function :

```
1 static void Swap(ref int a, ref int b);
```

This function takes two integers as parameters and swap their values. Passing by reference is obviously really important. This function will be really useful for the following exercises, so do it well !

4.1.2 Euclidean division

You have to implement the following function :

```
1 static int Div(ref int a, int b);
```

This function takes two integers as parameters and store the result of the division in the first one. The function returns the remainder of the division. If an error occurs (division by zero for example) the function should not modify any of the parameters and return -1.

4.1.3 Modulo

You have to implement the following function :

```
1 static bool Mod(ref int a, int b);
```

This function takes two integers as parameters and store the result of the modulo in the first one. If an error occurs (modulo by zero for example) the function should not modify any of the parameters and return false.

4.1.4 Somme

You have to implement the following function :

```
1 static int Sum(params int[] arr);
```

This function takes an undefined number of integers as a parameter. The function returns the sum of all the parameters. If there is no parameter, the function returns 0.

4.2 Exercice 2 : Arrays

All the following functions must be done in the project **Arrays**.

4.3 Search

You have to implement the following function :

```
1 static int Search(int[] arr, int e);
```

Let's start with an easy research function. You have to implement a function that returns the first occurrence of the element '*e*' in the array '*arr*'. Warning, the array is not sorted. If the element is not in the array, the function should return -1.

4.4 Minimum

You have to implement the following function :

```
1 static int Minimum(int[] arr);
```

Since you know how to parse an array, implement the function *minimum*. This function returns the index of the smallest element in the array. If the array is empty, the function returns -1.

4.4.1 Bubble sort

You have to implement the following function :

```
1 static void BubbleSort(int[] arr);
```

The Bubble sort is a basic sorting algorithm, but nonetheless very effective on arrays with a small amount of elements. The principle is quite simple : we compare an element with the next one and if they are not sorted, we swap them. We repeat the operation on each element of the array : The biggest element is now at the end of the array. After that, we start the operation again from the beginning of the array, but only to the second to the last this time. And we redo this operation again and again, until the array is sorted. For more infos : `https://en.wikipedia.org/wiki/Bubble_sort`

4.5 Exercise 3 : Connect 4

All the following functions must be done in the project **Connect 4**.

4.5.1 Introduction

Now that you know how to manage arrays, let's try to do something more interesting. You will try, in this exercise, to create a "*connect 4*" game. The project is split in several function that will help you obtain a program that works well. However, functions are not independent : the fact that a function works can depend on the result of an other.

4.5.2 CreateGrid

You have to implement the function *CreateGrid* with the following prototype :

```
1 static char[,] CreateGrid(int x, int y);
```

The purpose of this function is to create the grid. This is a multi-dimensional array, with the first dimension for the lines, and the second for the column. So *x* is the number of lines, *y* the number of column.
When created, this array should be filled with the character '-'. The function return this filled array.

4.5.3 Print

You have to implement the function *Print* with the following prototype :

```
1 static void Print(char[,] grid, int cursor);
```

This function will display the *grid* in parameter on the terminal. To be more legible, the *grid* should be printed with a frame (pipe and dash) except at the top. A space between each case should be also printed.

Something else essential to display : the column selection cursor. When it's his time to play, the player should select in which column he wants to put his token. Another function later in this practical will handle that. All you have to do here is to print the cursor (it should be a pipe character '|') at the top of the selected column. The parameter '*cursor*' contains the column number.

4.5.4 CheckInput

You have to implement the function *CheckInput* with the following prototype :

```
1 static int CheckInput(char[,] grid);
```

The purpose of this function is to manage the user input to get the column number chosen by the player.The player should be able to move the cursor at the top of the grid. Don't forget to print the grid for every move. The player should use the left and right arrow of the keyboard to move the cursor. You should use "*ConsoleKeyInfo*" and "*Console.ReadKey*" (For more information : MSDN). Move the cursor position until the enter key is pressed. The function returns an int containing the number of the column chosen by the player.

4.5.5 CheckWin

You have to implement the function *CheckWin* with the following prototype :

```
1 static bool CheckWin(char[,] grid, int h, int w, bool player);
```

The function's goal is to check if the "*player*" won with the last played token at the position *[h,w]*. The insertion will be implemented in the next function, no need to worry about it for now. To check if the player won, the function will call 3 sub-functions, that will check if there is a line, a column or a diagonal of 4 tokens. The function returns true if at least one of the sub-function return true. Otherwise, it returns false.
So you also need 3 more functions :

```
1 static bool CheckLine(char[,] grid, int h, bool player);
2 static bool CheckColumn(char[,] grid, int w, bool player);
3 static bool CheckDiagonal(char[,] grid, int h, int w, bool player);
```

Those functions check that there is 4 adjacent tokens of the current player in the line, column and diagonals including the *[h,w]* token.

4.5.6 AddToken

You have to implement the function *AddToken* with the following prototype :

```
1 static int AddToken(char[,] grid, int i, bool player);
```

This function adds a token of the current player in the column "*i*". The bool "*player*" represent the current player. To add a token, you only have to replace the '-' character in the space that will receive the token by a 'X' or 'O' character.
Be careful :

- The token should be inserted in the lower space of the column that has no token.
- If the column is already full, the token cannot be inserted : this is an error.

Once the token is added, the function should check if the player won, with the "*CheckWin*" function.
Concerning the return value :

- If the token is added and nobody wins, the function returns 0.
- If the token is added and someone wins, the function returns 1.
- If an error happens, the function returns -1.

4.5.7 main

The main of your project will do the initialization of the grid, and the game loop. You have already implemented an *init* function, use it now ! You should ask the player for the size of the grid before using the *init* function. For the main loop, you can do it as you want, you have already done all the functions you should need.
Some ideas for the main loop :

- get the player input.
- Add the token to the grid.
- If an error happens, check if the grid is full. If this is the case, the game ends with no winner. If not, ask another column to the same player.
- If there is no error, change the current player and redo the same.

```
 |
|O - - - -|
|O X - - -|
|O X X X -|
|O O X O X|
-----------
Player O Win !!!
```

FIGURE 5 – Example of display for Connect4

The Code is the Law.