TP C#4

1 Consignes de rendu

À la fin de ce TP, vous devrez soumettre une archive respectant l'architecture suivante :

```
rendu-tp-login.zip
    |-- rendu-tpcs4-login/
        |-- AUTHORS
        |-- README
        |-- Parameters/
            |-- Parameters.sln
            |-- Parameters/
                |-- Tout sauf bin/ et obj/
        |-- Arrays/
            |-- Arrays.sln
            |-- Arrays/
                |-- Tout sauf bin/ et obj/
        |-- Connect4/
            |-- Connect4.sln
            |-- Connect4/
                |-- Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer *login* par votre propre login.

N'oubliez pas de vérifier les points suivants avant de rendre votre TP :

- Le fichier AUTHORS doit être au format habituel : * *prenom.nom$* où le caractère '$' représente un retour à la ligne.
- Vous devez suivre le sujet **scrupuleusement**, et notamment **respecter les prototypes** des différentes fonctions.
- Pas de dossiers bin et/ou obj dans le projet.
- **Votre code doit compiler !**

2 Introduction

2.1 Objectifs

Au cours de ce TP, nous aborderons les notions suivantes :

- Passage de paramètres par référence
- Utilisation d'autres paramètres de méthodes (avec le mot clé *params* ou *out*)
- Manipulation de tableaux à une ou plusieurs dimensions

3 Cours

3.1 Passage par référence

3.1.1 Pré-requis

Vous ne l'avez peut-être pas remarqué mais lorsque nous passons des variables à une fonction les modifications faites à celle-ci ne sont que locales. En d'autres termes, une fois sorties de la fonction les variables auront leur valeur d'origine.

```
1  /*
2  ** Renvoie Factorielle de 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     //affiche la valeur de fact puis la valeur de res
18     Console.WriteLine(fact(4, res) + ", " + res);
19     Console.WriteLine(fact(2, res) + ", " + res);
20 }
```

Regardons la sortie de ce programme :

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

FIGURE 1 – Affichage dans la console

On peut remarquer ici que la valeur de *res* n'a pas changé alors que dans la fonction *fact* la variable est modifiée. En réalité lors de l'appel à la fonction *fact* une copie locale de *res* est crée et envoyée en tant que paramètre à la fonction. Ainsi, on ne touche pas à la valeur de *res* mais à celle de la copie, qui est détruite à la fin de l'exécution de la fonction.

3.1.2 Utilisation

Voyons ce qu'il se passe si nous changeons le prototype de la fonction *fact* comme suit :

```
1  /* Cette fois-ci nous rajoutons le mot-clé ref afin d'utiliser
2  ** le passage par référence
3  */
4  static int fact(int n, ref 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     /* Nous devons écrire ref devant res afin de le passer
18     ** en référence car demandé par le prototype de la fonction fact
19     */
20     Console.WriteLine(fact(4, ref res) + ", " + res);
21     Console.WriteLine(fact(2, ref res) + ", " + res);
22 }
```

Cette fois-ci le résultat devient :

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

FIGURE 2 – Affichage dans la console

On peut observer ici que *res* est égal à la valeur de retour de *fact*. Cela est dû à l'utilisation du passage par référence. Pour l'utiliser en C# on doit utiliser le mot-clé *ref*. Il permet de modifier la valeur de la variable donnée en paramètre. Nous demandons donc de ne pas faire une copie locale de la variable afin de pouvoir la modifier directement dans la fonction appelée.

3.1.3 Utilité

Ce n'est peut être pas évident de voir d'un coup d'oeil son utilité. On pourrait penser que juste retourner la valeur suffirait comme avec l'exemple de factorielle. C'est vrai pour ce cas mais pour certains le mot-clé devient obligatoire.
Vous ne voyez toujours pas l'utilité du passage par référence ? Voyons quelques exemples courants : Imaginons que l'on ait besoin d'enregistrer plus d'une valeur, de manipuler plusieurs variables ou de conserver les modifications effectuées sur plusieurs d'entre elles. Impossible pour nous de faire plusieurs fonctions afin de pouvoir retourner chaque valeur. Nous utilisons donc le passage par référence avec le mot-clé *ref* afin de pouvoir modifier toutes les valeurs dans la même fonction. Nous gagnons donc en clarté et nous avons évité des appels de fonctions inutiles.

3.2 Les tableaux

3.2.1 Vecteur

Un vecteur est un tableau à une dimension. Autrement dit, il s'agit d'une liste contiguë. En C# un vecteur se déclare avec la syntaxe "TYPE[ ] nom ;".

Exemple :

```
1 int[] vectorint; //vectorint est un vecteur d'entier
2 char[] vectorchar; //vectorchar est un vecteur de caractère
3 bool[] vectorbool; //vectorbool est un vecteur de booléen
```

L'instanciation d'un vecteur se fait avec la syntaxe "VECTOR = new TYPE[X] ;". On sait ainsi que le vecteur contient X éléments (x doit être positif ou nul). Il est commun et fortement recommandé de faire l'instanciation en même temps que la déclaration.

Exemple :

```
1 int[] vectorint;
2 vecotrint = new int[4];
3 //vectorint est vecteur d'entier de taille 4
4
5 char[] vectorchar = new char[2];
6 //vectorchar est un vecteur de caractère de taille 2
7
8 bool[] vectorbool = new bool[42];
9 //vectorbool est un vecteur de booléen de taille 42
```

L'initialisation d'un vecteur se fait avec la syntaxe "VECTOR = new TYPE[X]{ ELT1, ELT2, ..., ELTX } ;" ou plus simplement "VECTOR = { ELT1, ELT2, ..., ELTX } ;". On peut observer que dans le deuxième cas il n'est pas nécessaire de l'instancier explicitement. En réalité en C#, l'instanciation sera implicite dans ce cas. La première syntaxe est utile lorsque l'on connait la taille du tableau mais que l'on ne veut initialiser que les premiers éléments. A nouveau il est commun et fortement recommandé de faire l'initialisation en même temps que la déclaration.

```
int[] vectorint = new int[4]{1, 3, 3, 7};
/* vectorint est un vecteur d'entiers de taille 4
** contenant dans l'ordre les entiers 1, 3, 3 et 7
*/

char[] vectorchar = {'C', '#'};
/* vectorchar est un vecteur de char de taille 2
** contenant dans l'ordre 'C' et '#'
*/

bool[] vectorbool = new bool[6]{true, true, true, true, true, true};
/* vectorchar est un vecteur de booléen de taille 6
** dont les valeurs sont initialisées à true
*/
```

L'accès à un élément d'un vecteur se fait avec la syntaxe "VECTOR[X];". Cela permet d'accéder au (x + 1)ème élément du vecteur. La position dans un tableau est en base 0. On doit avoir 0 <= x < n avec n le nombre d'éléments du vecteur.

```
/*en réutilisant nos trois vecteurs déjà déclarés et initialisés */

vectorint[0]; //renvoie 1

vectorchar[1]; //renvoie '#'

vectorbool[3]; //renvoie true

vectorbool[4] = false;
/* on enregistre false dans vectorbool à la position 4 */
```

Pour finir on vous donne une petite astuce, vous pouvez récupérer la taille d'un vecteur avec la syntaxe suivante "VECTOR.Length".

3.2.2 Tableaux multidimensionnels

Un tableau multidimensionnel est un tableau à n dimensions avec n > 1.
La manipulation de tableau multidimensionnel est très proche de celle d'un vecteur :

— "TYPE[,] nom ;" pour la déclaration
— "VECTOR = new TYPE[D1, D2, ..., DX] ;" pour l'instanciation, avec x le nombre de dimensions et DN la largeur du tableau dans la dimension donnée.
— "VECTOR = {{ELT1D1, ELT2D1, ..., ELTXD1}, ..., {ELT1DX, ELT2DX, ..., ELTXDX}} ;" pour l'initialisation. On peut toujours utiliser la syntaxe permettant d'instancier de manière explicite pendant que l'on initialise.
— "VECTOR[P1, P2, ..., PX] ;" pour accéder à une valeur avec x le nombre de dimensions et 0 <= PN < DN

Exemple :

```
1  double[,,,] arraydouble = new double[42, 69, 23, 10];
2  /* arraydouble est un tableau de double à 4 dimensions
3  ** dont chacune est respectivement de largeur
4  ** 42, 69, 23 et 10
5  */
6
7  string[,] arrstr = new string[2, 2]{{"J'adore", "mes"}, {"ACDC", "!"}};
8  /* arrstr est un tableau de string à 2 dimensions
9  ** dont chacune est respectivement de largeur 2
10 */
11
12 double[,] arr = {{0.5, 1.0}, {47.0, 10.0}};
13 /* arr est un tableau de double à 2 dimensions chacune de
14 ** largeur 2 et initialisé avec 0.5, 1.0, 47.0, 10.0
15 */
16
17 arrstr[0, 0]; //renvoie "J'adore"
18 arrstr[0, 1]; //renvoie "mes"
19 arrstr[1, 0]; //renvoie "ACDC"
20 arrstr[1, 1] = "!!";
21 /* on enregistre "!!" dans arrstr à la position (1, 1) */
```

Comme pour les vecteurs "VECTOR.Length" va renvoyer le nombre d'éléments du tableau. Mais si vous ne voulez récupérer que le nombre d'éléments d'une des dimensions vous pouvez utiliser "VECTOR.GetLength(N)" où N est la dimension voulue (en base 0).

3.2.3 Tableaux de tableaux (en escalier)

Un tableau de tableaux, comme son nom l'indique, est un tableau contenant des tableaux. La manipulation d'un tableau en escalier est identique à celle d'un tableau multidimensionnel associé à celle d'un vecteur. Une manière simple de voir et comprendre son fonctionnement est de considérer l'objet tableau comme n'importe quel type/objet déjà connu.

Assez t-il tout étant, regardons un exemple :

```
/* Pour simplifier la compréhension, on réecrit les tableaux
** multidimensionnels de l'exemple précédent en tableaux en escalier
*/

double [,][][] arrdouble = new double[42, 69][23][10];
/* arrdouble est un tableau de tableaux de tableaux de double
** Il peut être vu comme un tableau contenant 10 tableaux contenant chacun
** 23 tableaux à deux dimensions
** Ces deux dimensions sont de largeur 42 et 69.
*/

string[][] arrstr = new string[2][2]{new string[2]{"J'adore", "mes"},
                                     new string[2]{"ACDC", "!"}};
// tabstr est un vecteur de vecteur de string

arrstr[0][0]; //renvoie "J'adore"
arrstr[0][1]; //renvoie "mes"
arrstr[1][0]; //renvoie "ACDC"
arrdouble[0, 12][22][3] = 32.0;
// On enregistre 32 dans arrdouble à la position (0, 12, 22, 3)
```

3.3 Paramètres de méthodes

3.3.1 out

Maintenant que vous êtes expert en passage par référence nous allons pouvoir voir des choses un peu plus avancées.
Commençons doucement avec le mot-clé *out*. Ce mot-clé ressemble énormément au mot-clé *ref* mais avec une petite subtilité.
En mettant le mot-clé out nous disons juste que nous voulons que la variable soit une variable de sortie/retour. Vous l'avez donc compris, cette variable se comporte comme un *return*, il faudra donc que dans tous les cas la variable soit modifiée.

La plus grande différence entre les deux mots-clés est le fait que pour *ref* la variable doit être initialisée auparavant car elle peut être modifiée dans la fonction. Alors qu'avec le mot-clé *out* la variable sera obligatoirement modifiée.

Pour mieux comprendre regardons un exemple avec l'utilisation de la fonction `Int32.TryParse` (Voir le MSDN pour plus d'informations) qui va convertir une string en entier si possible en utilisant le mot-clé *out*. Si la string ne peut pas être modifiée l'entier aura la valeur 0. La fonction retourne un booléen afin de savoir si la convertion s'est bien passée.

```
1  public static void Main (string[] args)
2  {
3      String[] strings = { "RTX", "42", "-1984", "10/02", "-322,546",
4                           "   +9000   ", "0xFE" };
5
6      for (int i = 0; i < strings.Length; ++i)
7      {
8          int number;
9          if (Int32.TryParse (strings[i], out number))
10             Console.WriteLine ("Conversion of {0} to {1}", strings[i],
11                                number);
12         else
13             Console.WriteLine ("Conversion of {0} failed", strings[i]);
14     }
15 }
```

Regardons le résultat :

```
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 – Affichage dans la console

3.3.2 params

Il existe le mot-clé *params* en C# qui permet de passer un nombre indéfini de paramètres du même type. Cet outil permet de considérer que les paramètres que l'on donnera à la fonction seront regroupés dans un tableau, et de manipuler ainsi cet ensemble de manière aisée. L'appel de la fonction pourra se faire avec un tableau du type attendu ou directement avec plusieurs paramètres du type attendu par la fonction mais aussi avec aucun argument.

Voyons un exemple :

```
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 }
```

Voyons le résultat :

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

FIGURE 4 – Affichage dans la console

4 Exercices

4.1 Exercice 1 : Parameters

Toutes les fonctions suivantes seront à faire dans le projet **Parameters**.

4.1.1 Swap

Vous allez devoir coder une fonction avec le prototype suivant :

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

Cette fonction prend en paramètres deux entiers et va échanger leur valeur. L'utilisation ici du passage par référence est donc très importante. Cette fonction sera très intéressante pour la suite du TP donc ne la négligez pas !

4.1.2 Division euclidienne

Vous allez devoir coder une fonction avec le prototype suivant :

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

Cette fonction prend en paramètres deux entiers et enregistre le résultat de la division dans le premier des deux. La fonction retournera le reste de la division euclidienne. En cas d'erreur (division par zéro par exemple) la fonction ne modifiera pas la valeur de *a* et renverra -1.

4.1.3 Modulo

Vous allez devoir coder une fonction avec le prototype suivant :

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

Cette fonction prend en paramètres deux entiers et enregistre le résultat du modulo dans le premier des deux. La fonction retourne true si tout s'est bien passé et false en cas d'erreur(modulo par 0 par exemple) ;

4.1.4 Somme

Vous allez devoir coder une fonction avec le prototype suivant :

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

Cette fonction prend en paramètres un nombre indéfini d'entiers. La fonction retournera la somme de tous les paramètres. Si aucun paramètre n'est donné la fonction devra retourner 0.

4.2 Exercice 2 : Arrays

Toutes les fonctions suivantes seront à faire dans le projet **Arrays**.

4.3 Recherche

Vous allez devoir coder une fonction avec le prototype suivant :

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

Commençons doucement avec une fonction de recherche basique. Il vous faut implémenter une fonction qui va renvoyer l'emplacement de la première apparition d'un élément *e* dans le tableau *arr*. Attention, le tableau n'est pas forcement trié. Si l'élément n'est pas dans le tableau, la fonction renvoie -1.

4.4 Minimum

Vous allez devoir coder une fonction avec le prototype suivant :

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

Maintenant que vous savez comment parcourir un tableau, implémentez la fonction *minimum*. Cette fonction renvoie l'index du plus petit élément du tableau. Si le tableau est vide, la fonction renvoie -1.

4.4.1 Bubble sort

Vous allez devoir coder une fonction avec le prototype suivant :

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

Le Bubble sort est un algorithme de tri assez basique mais efficace sur les tableaux contenant un nombre restreint d'éléments. Le principe est assez simple : on compare les éléments deux à deux, et si le premier élément est plus grand que le second, on inverse leur place dans le tableau. On répète l'opération jusqu'à la fin du tableau : le plus grand élément est maintenant à sa place, à la dernière case. On recommence alors le tri du début, jusqu'à la dernière case - 1 cette fois ci, et ainsi de suite jusqu'à ce que le tableau soit entièrement trié.
Pour plus d'informations : `https://en.wikipedia.org/wiki/Bubble_sort`

4.5 Exercice 3 : Puissance 4

Toutes les fonctions suivantes seront à implémenter dans le projet **Connect 4**.

4.5.1 Introduction

Maintenant que vous savez comment fonctionnent les tableaux, utilisons-les pour faire quelque chose d'un peu plus intéressant. Nous vous proposons dans cet exercice de créer un "*puissance 4*", qui fonctionne de la même manière que le jeu de société. L'exercice est découpé en plusieurs fonctions pour vous aider à avoir un résultat fonctionnel. Cependant, les fonctions ne sont pas indépendantes : le résultat d'une fonction peut dépendre du bon fonctionnement d'une autre.

4.5.2 CreateGrid

Vous allez devoir coder la fonction *CreateGrid* avec le prototype suivant :

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

Le but de cette fonction est de créer la grille de jeu. Elle est représentée par un tableau de caractères à double dimension, avec la première dimension représentant les lignes, et la seconde les colonnes. Le paramètre *x* est donc le nombre de lignes, le paramètre *y* le nombre de colonnes.
Une fois créé, ce tableau doit être rempli du caractère '-'. La fonction retourne ce tableau rempli.

4.5.3 Print

Vous allez devoir coder la fonction *Print* avec le prototype suivant :

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

Cette fonction va afficher la grille de jeu passé en paramètre dans la console. Pour améliorer la lisibilité, un cadre doit être affiché autour de la grille, excepté en haut. Un espace entre chaque case est également nécessaire.

Un second élément important à afficher est le curseur de sélection de la colonne. Lors de son tour, le joueur va devoir sélectionner la colonne dans laquelle il veut insérer son jeton. Une fonction plus tard dans le TP implémentera cette fonctionnalité. Tout ce qu'il reste à faire ici dans la fonction *Print* est d'afficher un curseur (représenté par un pipe '|') au dessus de la colonne sélectionnée. Le paramètre "*cursor*" contient le numéro de la colonne.

4.5.4 CheckInput

Vous allez devoir coder la fonction *CheckInput* avec le prototype suivant :

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

Le but de cette fonction est de gérer les entrées clavier afin de récupérer le numéro de la colonne choisie par le joueur. L'utilisateur doit pouvoir bouger le curseur en haut du tableau (pensez à *Print* la grille à chaque fois que le curseur bouge) grâce aux flèche gauche et droite du clavier. Pour cela, utilisez

"*ConsoleKeyInfo*" et "*Console.ReadKey*" (Pour plus d'infos : MSDN). Continuez à modifier la position du curseur jusqu'à ce que la touche "entrer" soit pressée. La fonction retourne un entier désignant la colonne choisie par le joueur.

4.5.5 CheckWin

Vous allez devoir coder la fonction *CheckWin* avec le prototype suivant :

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

Cette fonction va avoir pour but de vérifier si le joueur "*player*" a gagné en insérant son nouveau jeton à la place *[h,w]*. L'insertion sera implémentée dans la prochaine fonction du TP, pas besoin de s'en préoccuper ici. Pour vérifier cela, cette fonction va simplement appeler 3 "sous-fonctions" qui vont vérifier si il a fait une ligne, une colonne ou une diagonale de 4 jetons. La fonction va retourner vrai si au moins une des 3 sous-fonctions retourne vrai. Autrement, la fonction retourne faux.
Nous avons donc 3 fonctions :

```
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);
```

Ces 3 fonctions vont donc vérifier respectivement si il y a 4 jetons du joueur "*player*" qui se suivent sur la ligne *h*, la colonne *w*, ou une diagonale passant par le points *[h,w]*. Elles retournent vrai si le joueur gagne, sinon faux.

4.5.6 AddToken

Vous allez devoir coder la fonction *AddToken* avec le prototype suivant :

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

Cette fonction va devoir ajouter un jeton dans la colonne de "*grid*" désignée par l'entier *i* en paramètre. Le booléen "*player*" passé en paramètre permet de savoir quel jeton ajouter. Pour ajouter un jeton, il suffit de remplacer le caractère '-' déjà présent dans la case qui va recevoir le jeton par un *'X'* ou un *'O'*.
Attention à plusieurs choses :

- Le jeton doit être placé à la case la plus basse possible dans la colonne, mais qui n'est pas déjà occupée par un jeton.
- Si la colonne est pleine, le jeton ne peut pas être inséré, c'est un cas d'erreur.

Après avoir ajouté le jeton, il faut vérifier si le joueur a gagné. Pour cela, utilisez la fonction "*CheckWin*" implémentée précédemment. En ce qui concerne la valeur de retour :

- Si le jeton a été inséré et que personne n'a gagné, la fonction retourne 0.
- Si le jeton a été inséré et que le joueur a gagné, la fonction retourne 1.
- Si un cas d'erreur survient, la fonction retourne -1.

4.5.7 main

Le Main de votre projet va contenir l'initialisation de la grille puis la boucle de jeu. Vous avez déjà codé une fonction pour l'initialisation, c'est le moment de vous en servir. Avant de démarrer la boucle de jeu, la fonction demande au joueur la hauteur et la largeur de la grille désirée. Pour la boucle de jeu, libre à vous de gérer cela comme vous voulez, mais vous avez dans tout les cas déjà implémenté toutes les fonctions qui pourraient vous être utiles.
Idée pour la boucle de jeu :

- Récupérer l'input du joueur.
- Ajouter le jeton à la grille.
- Si une erreur survient, vérifier que la grille n'est pas pleine. Si elle est pleine, le jeu est terminé. Sinon, redemander un choix au même joueur.
- Sinon, passer au joueur suivant et recommencer.

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

FIGURE 5 – Exemple d'affichage pour le puissance 4

The Code is the Law.