TP C#3 : Il était un jeu...

Consignes de rendu

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

```
rendu-tpcs3-prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- Basics/
        |-- Basics.sln
        |-- Basics/
            |-- Everything except bin/ and obj/
    |-- Takuzu/
        |-- Takuzu.sln
        |-- Takuzu/
            |-- Everything except bin/ and obj/
```

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

— Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
— Les fichiers `AUTHORS` et `README` sont obligatoires.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
*␣prenom.nom$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* prenom.nom" > AUTHORS
```

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

1 Introduction

1.1 Objectifs

Maintenant que vous savez tous utiliser les variables et les différentes boucles en C#, nous allons étudier dans ce TP la définition et l'utilisation des paramètres d'une fonction ainsi que l'utilisation de tableaux.
Ces deux notions sont très importantes et vous seront utiles tout au long de l'année. Pour cela, il est très important de comprendre ce TP, n'hésitez pas à relire la partie cours si cela vous est nécessaire

2 Cours

2.1 Passage par référence

Pour l'instant, on vous a appris qu'en C# une fonction utilise (ou non) des paramètres et retourne une valeur (ou pas), mais qu'en est-il quand nous voulons retourner plus d'une valeur, si une seule valeur de retour ne nous suffit pas.
De base, lors du passage en paramètre d'une variable, la fonction qui est appelée, va créer une copie de cette variable pour travailler dessus.
Les changements effectués sur cette variable n'impacteront donc pas la variable qui a été passée en paramètre. Ceci est le passage par valeur, le mode par défaut lors du passage d'une variable en paramètre.

Maintenant, si on souhaite passer en paramètre une variable et que celle-ci soit affectée par les changements effectués dans la fonction, il faut utiliser un autre mode. Pour cela, en C#, il existe une fonctionnalité lors de la déclaration des paramètres d'une fonction : Le passage par référence à l'aide du mot clé "ref" (pour référence).
Cette fonctionnalité permet lors du passage d'une variable en paramètre d'une fonction de dire "Je veux que toutes les modifications faites à cette variable dans cette fonction lui soient également appliquées dans la fonction actuelle". Autrement dit, on souhaite utiliser la même variable dans les deux fonctions et que toutes ses modifications soient effectuées dans les deux fonctions.

Essayons par exemple :

```
1  static void foo(ref string str1, string str2)
2  {
3      str1 = "These violent deadlines"
4      str2 = "have violent ends."
5  }
6
7  static void Main(string[] args)
8  {
9      string str1 = "ACDC 2020"
10     string str2 = "are the best."
11     Console.WriteLine(str1 + " " + str2);
12     foo(ref str1, str2);
13     Console.WriteLine(str1 + " " + str2);
14 }
```

```
>./main
ACDC 2020 are the best.
These violent deadlines are the best.
```

Comme vous pouvez le voir, les deux lignes sont différentes, dans la seconde le contenu de la variable 'str1' a changé mais pas celui de la variable 'str2' alors que dans foo() les deux sont modifiées. C'est en utilisant le mot clé "ref" lors de la déclaration de la fonction et lorsqu'on place notre variable en paramètre, que l'on utilise le passage par référence et que l'on spécifie à notre machine que nous voulons que cette variable subisse les changements effectués dans l'autre fonction.

2.2 Les tableaux

2.2.1 Quelques notions

Un tableau est un ensemble de valeurs rangées dans un certain ordre. Tous les tableaux possèdent les propriétés suivantes :

- Un tableau est toujours passé aux fonctions par référence, pas besoin de préciser "ref" dans le prototype ou l'appel de la fonction.
- La taille d'un tableau est FIXE, elle n'est plus modifiable une fois le tableau créé.
- Toutes les valeurs dans un tableau sont du même type.
- On peut accéder et changer le contenu d'une des cases du tableau à tout moment.
- Un tableau peut très bien contenir un autre tableau.
- Les éléments sont placés dans des cases indexées de 0 à n-1 avec n le nombre d'éléments.

2.2.2 Déclaration d'un tableau

Pour pouvoir utiliser un tableau, il faut commencer par le déclarer, pour cela, on dispose de plusieurs méthodes :

```
1  int[] array;
2  /* Cette variable contiendra un tableau d'entier.*/
3
4  array = new int[5];
5  // On cree un tableau qui contiendra 5 valeurs qui ne sont pas definies.
6
7  int[] bis = {1, 2, 3, 4, 5};
8  // On cree un tableau qui contiendra 5 valeurs qui sont definies.
9
10 int[][] tabtab = new int[5][]; // tableau de tableau.
11 int[][] tabtab2 = { bis, bis };
12 int[,] dim2 = new int[5, 1]; // tableau a 2 dimensions.
13 int[,] dim2bis = { { 1, 2, 3 }, { 4, 5, 6 } };
14
15 public static void Replace(int[] tab, int i, int val)
16 {
17     tab[i] = val;
18 }
19
20 Replace(bis, 2, 12);
21 //bis vaut {1, 2, 12, 4, 5} maintenant.
```

On peut soit déclarer un tableau en initialisant chacun des champs, comme indiqué dans la première méthode. Soit uniquement préciser sa taille sans fixer de valeur pour le moment. Dans les deux cas la taille est définie. Dans la première méthode, la taille est le nombre d'éléments, dans la seconde c'est le nombre spécifié. De plus si on utilise la seconde méthode, tout les champs sont initialisés à 0. Les deux dernières lignes de l'exemple sont des déclarations de tableaux à plusieurs dimension, les deux types de tableaux se comportent de la même façon seule leur utilisation diffère.

2.2.3 Accéder aux valeurs

Pour accéder à une case du tableau, on procède de la façon suivante :

```
1 bis[0]; // retourne 1
2 tabtab2[1][2]; // retourne 3
3 dim2bis[1,2]; // retourne 6
```

On précise dans les crochets à quelle case du tableau on souhaite accéder.
ATTENTION : tenter d'accéder à une case en dehors du tableau fera planter le programme.

Cette méthode permet également de changer le contenu d'une case :

```
1 array[0] = 100;
2 Console.WriteLine("array[0] = {0}", array[0]);
```

2.2.4 Les strings

Les string sont des tableaux de *char*. Elles possèdent certes de plus grandes fonctionnalités, mais tout ce qui marche sur un tableau, marche sur une *string*, y compris l'accès aux cases.

3 Exercices

3.1 Variables

Nous allons voir quelques utilisations basiques des références en C#. Cela vous permettra de vous familiariser avec ce principe qui, vous le verrez vite, est très utile.

3.1.1 Swappons

Écrire une fonction qui swap 2 entiers, on vous avait dit que ce serait basique.

```
1 public static void Swap(ref int a, ref int b);
2 //exemple:
3 int a = 1;
4 int b = 5;
5 Swap(ref a, ref b);
6 // a = 5 et b = 1 maintenant.
```

3.1.2 Let's work on float !

Coder une fonction qui retourne la partie entière d'un *float* et remplace le contenu du *float* par sa partie décimale.

```
1 public static int Trunc(ref float f);
```

3.1.3 RotChar

Vous en rêviez ? Ils l'ont fait, *RotChar* est de retour. Vous devez échanger les majuscules avec les majuscules, les minuscules avec les minuscules et les chiffres avec les chiffres. Vous connaissez la musique, votre fonction doit suivre le prototype suivant :

```
1 public static void RotChar(ref char c, int n);
```

ATTENTION : pensez que 'n' peut être négatif.

3.2 Tableaux

Maintenant que vous avez vu le fonctionnement des tableaux, il est temps de les utiliser.

3.2.1 Cherchons un peu

Coder la fonction *Search*, qui prend en paramètre un tableau d'entiers *arr* et retourne la position de la première occurrence de l'élément *e* dans celui-ci (-1 si *e* n'est pas présent).

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

3.2.2 Le roi de la coline

Coder la fonction *KingOfTheHill* qui prend en paramètre un tableau d'entiers *arr* et retourne le roi de la colline.

```
1 public static int KingOfTheHill(int[] arr)
```

Les collines seront sous cette forme la :

```
int[] hill =
{
  1, 3, 4, 6, 8, 9, 7, 5, 3, 0
};
```

Une suite d'entiers croissante puis décroissante, le roi de la colline est l'entier au sommet, soit ici 9.

Attention, une simple recherche de maximum ne vous garantira pas tous les points.

3.2.3 L'armée des clones

Ecrire la fonction *CloneArray* qui prend en paramètre un tableau *arr* et renvoie une copie de celui-ci.

```
static int[] CloneArray(int[] arr)
```

3.2.4 Des bulles et du tri

Implémenter le tri à bulles :

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

Le BubbleSort est un algorithme basique de tri de tableaux, celui-ci se comporte comme ceci :
- Parcourir tout le tableau
- Comparer l'élément courant et son suivant.
- Si ils ne respectent pas la condition de tri, les échanger
- Recommencer tant qu'au moins un échange a eu lieu

Félicitations, vous avez maintenant un tableau trié.

Pour plus d'informations : `https://en.wikipedia.org/wiki/Bubble_sort`

Vous devez, ici, trier le tableau en ordre croissant.

```
int[] arr = {2, 4, 6, 1, 4, 9, 0, 5, 2};
BubbleSort(arr);
/* arr = {0, 1, 2, 2, 4, 4, 5, 6, 9} */
```

Bonus : implémenter un autre algorithme de tri, vous devez préciser le nom du tri et en quoi il est plus efficace dans le README.

3.3 Takuzu

Le Takuzu est un jeu de réflexion d'origine belge créé par Frank Coussement et Peter De Schepper en 2009. Il est basé sur la logique un peu comme le sudoku. Il peut s'agir de grilles allant de 6x6 à 14x14 en général, mais pouvant très bien avoir un nombre de colonnes et de lignes différent (voire différents entre eux pourvu qu'ils soient pairs). Chaque grille ne contient que des 0 et des 1, et doit être complétée en respectant trois règles :

- autant de 1 et de 0 sur chaque ligne et sur chaque colonne complète
- pas plus de 2 chiffres identiques côte à côte
- 2 lignes ou 2 colonnes ne peuvent être identiques

Pour plus d'informations : `https://fr.wikipedia.org/wiki/Takuzu`

Nous représenterons une grille par un tableau à deux dimensions, les cases vides valant '-1'
Ainsi le code suivant :

```
1  int[,] grid =
2  {
3    {-1, -1,  0, -1, -1, -1, -1, -1},
4    { 1, -1,  1, -1, -1, -1, -1, -1},
5    {-1, -1, -1, -1,  0,  0, -1, -1},
6    {-1, -1, -1,  1, -1, -1, -1, -1},
7    {-1, -1,  0, -1, -1, -1, -1,  0},
8    {-1, -1, -1, -1,  0, -1, -1,  0},
9    { 0, -1, -1,  0,  0, -1, -1, -1},
10   { 0, -1, -1, -1, -1,  0,  0, -1}
11 };
12 Takuzu.PrintGrid(grid);
```

Affichera ceci :

```
   0 1 2 3 4 5 6 7
0 | | |0| | | | | |
1 |1| |1| | | | | |
2 | | | | |0|0| | |
3 | | | |1| | | | |
4 | | |0| | | | |0|
5 | | | | |0| | |0|
6 |0| | |0|0| | | |
7 |0| | | | |0|0| |
```

3.3.1 Gutenberg

Ecrire la fonction :

```
1  void PrintGrid(int[,] grid);
```

Qui affiche la grille selon le format suivant :

- le numéro des colonnes au dessus de chaque colonne
- le numéro des lignes à gauche de chaque ligne, suivi d'un espace
- chaque ligne commence et finit par un '|'
- chaque valeur est séparée par un '|'
- les cases vides sont représentées par un espace

3.3.2 Est-ce valide ?

Écrire les fonctions suivantes

```
1  bool IsRowValid(int[,] grid, int row);
2  bool IsColumnValid(int[,] grid, int col);
```

Qui vérifient que la ligne *row* (respectivement la colonne *col*) de la grille *grid* peut encore respecter les deux premières conditions du Takuzu.

Puis écrire la fonction :

```
1  bool IsGridValid(int[,] grid);
```

Qui vérifie que chaque ligne et chaque colonne est valide et qu'aucune n'est identique à une autre.

3.3.3 Jouons un peu

Ecrire la fonction :

```
1  bool PutCell(int[,] grid, int x, int y, int val);
```

Qui place la valeur *val* à la position (x,y)
la fonction ne modifie pas la grille et renvoie faux si :

- *val* n'est pas une valeur valide
- les positions x ou y ne sont pas dans les limites du tableau
- La grille n'est plus valide après avoir changé la valeur

La fonction renvoie vrai et modifie la grille dans tous les autres cas.

3.3.4 Jouons beaucoup

Écrire la fonction :

```
1  void Game(int[,] start);
```

Qui commence une partie de Takuzu à partir de la grille start.
Elle se déroule de la manière suivante

- Afficher la grille
- Demander les coordonées
- Demander la valeur à changer

```
   0 1 2 3 4 5 6 7
0 | | |0| | | | | |
1 |1| |1| | | | | |
2 | | | | |0|0| | |
3 | | | |1| | | | |
4 | | |0| | | | |0|
5 | | | | |0| | |0|
6 |0| | |0|0| | | |
7 |0| | | | |0|0| |
x : 1
y : 1
value : 0
   0 1 2 3 4 5 6 7
0 | | |0| | | | | |
1 |1|0|1| | | | | |
2 | | | | |0|0| | |
3 | | | |1| | | | |
4 | | |0| | | | |0|
5 | | | | |0| | |0|
6 |0| | |0|0| | | |
7 |0| | | | |0|0| |
x :
```

La grille n'est pas modifiée si les coordonnées correspondent à un point déjà présent dans la grille d'origine ou si *PutCell* renvoie faux.

Un message est affiché lorsque la grille est finie. Soyez créatif.

3.3.5 Bonus

Ecrire la fonction :

```
1  void AI(int[,] grid);
```

qui finit la grille automatiquement, n'hésitez pas à réfléchir à vos propres algorithmes, dans tous les cas la procédure devra être expliquée dans le README.

These violent deadlines have violent ends.