TPC#3 : Vers l'infini et au delà !

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Brian Kernighan

1 Présentation du TP

1.1 Objectifs

Le but de ce TP est d'apprivoiser le fonctionnement de deux des structures les plus importante de tout langage de programmation, afin de vous ~~faire oublier la récursivité~~ introduire les bases de la programmation impérative.

Vous apprendrez de plus à utiliser un des outils les plus performants de Visual Studio, son débogueur graphique. Le débogueur vous sera indispensable pour détecter et trouver les nombreux bugs de vos projets. Il n'est donc pas à négliger.

1.2 Structure du rendu

La structure du rendu pour ce TP ne change pas de celles que vous avez l'habitude de voir. Encore une fois, `login_x` est à remplacer par votre propre login.

Pour ce TP, on vous demande de rendre uniquement les .cs de vos solutions, pas la solutions entières. Les fichiers contiennent les exercices dont ils portent le nom.

```
rendu-tp-login_x.zip
    - login_x/
        - AUTHORS
        - exercices/
            - Maths.cs          <== Toutes les fonctions de la partie 4.1
            - PrettyPrintNumbers.cs
            - Average.cs
            - PrintRectangle.cs
            - Towers/
                - Towers.sln
                - Tower/        <== Le dossier de la solution, sans le fichier .suo
                                    et les dossiers "bin" et "obj"
```

2 Le débogueur

L'une des puissances de Visual Studio est son débogueur. Nous allons voir dans cette partie à quoi sert un débogueur et comment fonctionne celui de Visual Studio.

2.1 Rôle d'un débogueur

Lorsque vous faites un programme ou des fonctions, il se peut que cela fonctionne dans le sens où votre programme compile, mais le résultat n'est pas celui attendu. Pour résoudre ce genre de problème on fait ce qu'on appelle du « debugging » (en français *déboguage*). Cela consiste à exécuter un programme instruction par instruction en ayant des informations sur le contexte actuel (les valeurs des variables, la pile des fonctions appelées/appelantes, etc...). Prenons un exemple concret :

Voici une fonction qui est censée calculer récursivement la factorielle d'un nombre :

```
static int fact(int n)
{
    if (n <= 1)
        return 0;
    else
        return (n * fact(n - 1));
}

static void Main(string[] args)
{
    Console.WriteLine(fact(5));
    Console.ReadLine();
}
```

Aux premiers abords, ce code à l'air d'être correct. Bien entendu, ici la fonction est simple et vous serez nombreux à trouver la faute (pour ceux qui ne voient pas, cela peut être un exercice). Imaginez simplement qu'il ne s'agisse pas de la fonction factorielle mais d'un algorithme beaucoup plus complexe.

Tentons donc d'exécuter ce code avec `fact(5)`, le résultat attendu étant 120. Mais ici, nous obtenons 0. Faisons semblant d'ignorer la solution pour la suite de cet exemple.

Pour pouvoir chercher où se trouve le problème, il faut donc faire du déboguage. Pour le C# sous Visual Studio, il existe le débogueur (programme permettant de faire du déboguage) intégré à Visual Studio. Vous n'avez rien à lancer, il est sous vos yeux. Commençons par voir les fonctionnalités qu'il nous propose.

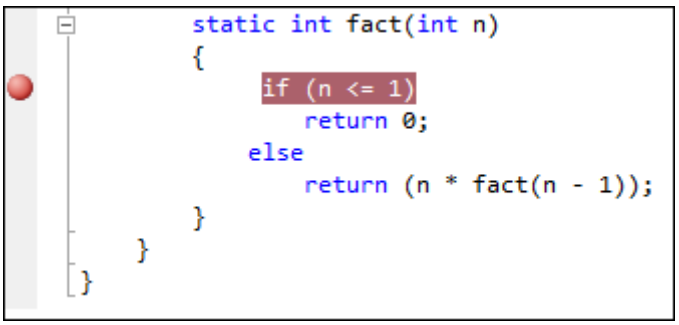
2.2 Les breakpoints

Dans un premier temps, il va falloir interrompre notre programme pendant son exécution pour pouvoir analyser le contexte à un instant donné. Pour ce faire, vous pouvez placer ce qu'on appelle des « breakpoints » (en français *points d'arrêt*).

Sous Visual Studio, vous pouvez placer des breakpoints en cliquant sur la ligne voulue dans la colonne grise à gauche de votre code ou bien aller sur la ligne voulue et appuyer sur F9. Une fois le breakpoint placé, lancez votre programme avec F5. Si tout se passe bien, votre programme est censé suspendre son exécution à la ligne où vous avez placer votre breakpoint.

Note : Lorsque vous placez un breakpoint sur une ligne où se trouve une instruction, votre programme s'arrêtera **avant** l'exécution de l'instruction.

Plaçons dans notre code un breakpoint à la première ligne de la fonction `fact` :



Lançons notre programme avec une fonction `main` adéquat. Ici, nous allons appeler la fonction `fact` avec 5. L'exécution s'arrête avant l'évaluation du bloc `if` à cause du breakpoint placé sur cette ligne. Sous Visual Studio, plusieurs fenêtres se remplissent d'informations lors d'un arrêt. Nous allons examiner en particulier les deux fenêtres du bas, *variables locales* (locals) et *Pile d'appel* (Call stack) :

- La fenêtre *locals* affiche l'état des variables locales à l'instant où le programme s'est arrêté. Ici, nous voyons qu'il y a une variable `n` dont sa valeur est 5. Vous pouvez interagir directement avec ces variables pour par exemple changer leur valeur.

- La fenêtre *call stack* affiche l'ensemble des fonctions appelées jusqu'à l'instant où le programme s'est arrêté. La fonction dans la première ligne étant la fonction dans laquel vous vous situez.

Nous allons maintenant voir comment se déplacer dans l'exécution de notre programme.

2.3 Se déplacer dans l'exécution

Pour avancer dans l'exécution du programme, il faut utiliser les raccourcis F10 ou F11. F10 vous permet de vous déplacer d'instruction en instruction sans rentrer à l'intérieur d'une éventuelle fonction. Lorsque vous utilisez F11, si un appel de fonction est présent sur l'instruction sur laquelle vous vous situez, vous rentrez à l'intérieur de cette fonction et vous pouvez continuer le debugging.

Si vous voulez simplement vous arrêter à un breakpoint pour simplement voir l'état des variables ou de la pile d'appels, vous pouvez utiliser F5 lorsque l'exécution est suspendue pour ainsi reprendre l'exécution normalement jusqu'à un éventuel breakpoint.

2.4 Application

Nous allons voir, à l'aide des techniques exposées précédemment, comment déboguer notre fonction `fact`. Reprenons notre exemple lorsque nous avions placé un breakpoint sur la troisième ligne. Si on lance notre programme, l'exécution sera suspendue avant l'évaluation de l'instruction

```
        if (n <= 1)
```

Ici, on peut observer dans la fenêtre des variables locales que `n` vaut 5. Si on avance d'une instruction (F10), on observe que l'on saute l'instruction contenue dans le `if` et l'instruction courante devient la ligne

```
       else
```

La condition `(n <= 1)` était fausse, il est donc normal que l'on ne soit pas rentré dans le `if`. Si on avance encore (F10), on arrive sur la ligne où se trouve l'appel récursif. On continue l'exécution (F10) et on peut s'apercevoir que la pile d'appel a maintenant un élément en plus. Effectivement, l'appel récursif a ajouté dans la pile d'appels un nouvel appel à la fonction `fact`. Maintenant, nous sommes dans la fonction `fact` qui est elle-même dans la fonction `fact`, c'est le principe même de la récursivité. On répète donc ces opérations :

- `n` est supérieur à 1, la condition est fausse
- On avance (F10)
- On va dans l'appel récursif (F10)

Ici on décrémente `n` de 1 à chaque appel récursif. Nous avons un cas d'arrêt, lorsque `n` vaut 1. On va donc continuer à avancer jusqu'à ce que `n` soit égal à 1. La condition du `if` est maintenant vraie, on rentre dans le bloc. On lit l'instruction qui est `return 0;`. Aucun appel récursif n'est fait ici, on termine donc le dernier appel de `fact` et on va remonter toute la pile d'appel. Dès le premier retour, on se rend compte que l'on effectue `1 * 0`. Si on remonte encore, on s'aperçoit que l'on effectue à chaque fois des multiplications par 0, pour au final renvoyer 0. Pour régler ce problème, il fallait simplement remplacer le `return 0;` par un `return 1;` et tout rentre dans l'ordre.

2.5 Précisions

Vous n'avez pas d'exercice à rendre sur cette partie, cela ne veut pas dire que vous devez la laisser tomber ! Le déboguage est quelque chose de très important et très utile, vous vous en servirez plus tard pour vos projets. Le débogueur de Visual Studio est un outil très complet et puissant, alors profitez-en !

3 Les boucles

Dans le TP précédent, vous avez vu la boucle `while`. Après un bref rappel de l'utilisation de cette boucle, on va voir d'autres types de boucle qui permettront de simplifier votre code et le rendre plus lisible.

3.1 Petit rappel : boucle `while`

Pour rappel, voici la structure d'une boucle `while` :

```
while ( /* condition */ )
{
    /* instructions */
}
```

Ainsi, si on veut écrire une telle boucle :

```
tant que i > 0 faire
    i <- i + 1
fin tant que
```

Cela donne en C# :

```
while (i > 0)
{
    i = i + 1;
}
```

Cela va exécuter l'instruction `i = i + 1` tant que la variable `i` est strictement positive. C'est-à-dire lorsque la variable `i` devient égale à 0, on ne va plus rentrer dans la boucle.

Petit exercice (pas à rendre)

Faites une fonction qui prend en paramètre un entier `n` et qui affiche dans la console tous les nombres de 1 jusqu'à `n` en utilisant une boucle `while`.

```
static int PrintNumbers(int n) { }
```

Voici un exemple de sortie avec 4 :

```
1
2
3
4
```

3.2 La boucle `for`

Maintenant nous allons voir un nouveau type de boucle : la boucle `for`. Vous avez sûrement déjà vu les boucles `pour ... jusqu'à` en langage algo :

```
pour/* initialisation */ jusqu'à /* fin */ [descendant] faire
    /* instructions */
fin pour
```

Voici sa syntaxe en C# :

```
for ( /* initialisation */; /* condition */; /* instruction de fin de boucle */ )
{
    /* instructions */
}
```

Cela vous paraît compliqué ? Détrompez-vous, le code ci-dessus équivaut **exactement** à celui-ci :

```
/* initialisation */;
while ( /* condition */ )
{
    /* instructions */
    /* instruction de fin de boucle */;
}
```

En vérité, les boucles `for` sont là pour rendre l'écriture d'une boucle qui parcours un intervalle défini à l'avance plus simple et lisible. Si l'on veut par exemple énumérer les `n` premiers entiers, on utilisera une boucle `for` car on connaît le nombre d'entier que l'on veut énumérer.

La déclaration d'une boucle `for` se découpe en trois parties (sans compter les instructions à l'intérieur) :

- **L'initialisation :** cette instruction s'exécute **une** fois, **avant** le démarrage de la boucle, que la condition soit vraie ou non. On s'en sert généralement pour initialiser une ou plusieurs variables utilisées dans la condition.

- **La condition :** comme d'habitude, c'est la condition qui permet d'indiquer quand la boucle doit arrêter de se répéter. Elle est évaluée **avant** que les instructions ne soient exécutées.

- **L'instruction en fin de boucle :** cette instruction s'exécute **après chaque** itération, avant que la condition ne soit évaluée, et sert généralement à modifier la valeur de la variable utilisée dans la condition. Vous noterez que cette instruction est absente de la boucle en langage algo, puisqu'elle est implicite.

Un exemple simple, qui calcule la somme des nombres de 1 à n :

```
int sum = 0;

for (int i = 1; i <= n; i++) {
    sum += i;
}
```

Note : Ici, on utilise un sucre syntaxique pour incrémenter les variables `i` et `sum`. Ces trois instructions sont équivalentes :

```
sum = sum + 1;
sum += 1;
sum++; /* Ici, le ++ signifie « incrémenter de 1 » la variable sum */
```

Un autre exemple, cette fois-ci pour parcourir tous les éléments d'un tableau :

```
string[] names = new string[] { "Nathalie", "Krisboul", "Xavier" };

for (int i = 0; i < names.Length; i++) {
    Console.WriteLine(names[i]);
}
```

Vous verrez les tableaux plus en détails dans le prochain TP. Pour l'instant, retenez juste les points suivants :

- les tableaux sont des variables contenant plusieurs éléments.
- vous pouvez accéder à un élément en particulier en indiquant son numéro entre crochets après le nom du tableau.
- l'indexation des tableaux commence à 0. Par exemple, le troisième élément d'un tableau se situe à l'index 2.
- le champ `Length` contient le nombre d'éléments du tableau. Dans l'exemple ci-dessus, `names.Length` vaut donc 3.

Petit exercice II, le retour (pas à rendre)

Réécrivez la fonction précédente en utilisant une boucle `for`.

```
static int PrintNumbers(int n) { }
```

C'est plus simple, non ?

4 Exercices à rendre

4.1 Un peu de mathématiques

4.1.1 Multiplication

Écrivez une fonction qui calcule $a \times b$ en utilisant uniquement des additions.

```
static int Multiplication(int a, int b)
```

4.1.2 Division

Écrivez une fonction qui calcule a/b en utilisant uniquement des soustractions. On supposera que a est toujours un multiple de b.

```
static int Division(int a, int b)
```

4.1.3 Factorielle

Réécrivez la fonction factorielle que vous aviez vu en Caml mais cette fois-ci en itératif, c'est-à-dire en utilisant uniquement des boucles et sans appel récursif.

```
static int Factorielle(int n)
```

4.1.4 Fibonacci (récursif)

Écrivez la fonction qui calcul le terme `n` de la suite de Fibonacci en récursif. Pour rappel, la suite est définie telle que :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_n = U_{n-1} + U_{n-2} \end{cases}$$

```
static int Fibonacci(int n)
```

4.1.5 Bonus - Fibonacci (itératif)

Réécrivez la fonction précédente en itératif, sans aucun appel récursif.

```
static int FibonacciIter(int n)
```

4.2 PrettyPrintNumbers

Vous aller devoir améliorer les deux petits exercices vus précédemment afin de rendre l'affichage plus sympathique. Écrivez un programme console qui demande à l'utilisateur de rentrer un nombre et affiche tout les nombres de 1 jusqu'à ce nombre. Voici un exemple d'utilisation et de sortie :

```
Entrez un nombre :
10
1 2 3 4 5 6 7 8 9 10
```

Vous pouvez utiliser `Console.Write()` qui est identique à `Console.WriteLine()` sans saut de ligne automatique. Si vous voulez ajouter un saut de ligne avec `Console.Write()`, utilisez le caractère « \n ».

Améliorez votre programme pour que l'on puisse entrer le premier, le dernier nombre ainsi qu'un pas entre chaque nombre. Améliorez également l'affichage des nombres, par exemple une séparation avec des virgules.

```
Entrez le premier nombre :
6
Entrez le dernier nombre :
16
Entrez le pas :
2
6, 8, 10, 12, 14 et 16
```

4.3 Average

Écrivez un programme console qui demande à l'utilisateur de rentrer un nombre `n` suivit de `n` nombres entiers et calcule la moyenne de ces nombres. Si le nombre entré est négatif, comptez-le comme si c'était un 0.

```
Combien de nombres ?
5
Nombre 1 :
12
Nombre 2 :
8
Nombre 3 :
16
Nombre 4 :
-6
Nombre 5 :
10
Moyenne : 9
```

```
Combien de nombres ?
0
Moyenne : 0
```

4.4 PrintRectangle

Écrivez une fonction qui prend en paramètre un nombre `n` et qui affiche des caractères de façon à former un carré de taille `n`. Vous ne devez pas gérer les nombres négatifs. Exemple :

```
static void PrintSquare(int n)
```

PrintSquare(4)

```
****
****
****
****
```

Modifiez cette fonction de façon à faire un programme qui demande à l'utilisateur de rentrer deux nombres `n` et `m` et qui dessine un rectangle de largeur `n` et de hauteur `m`.

```
Entrez la largeur :
6
Entrez la hauteur :
3
******
******
******
```

Modifiez l'affichage afin d'afficher uniquement les bordures du rectangles.

```
Entrez la largeur :
6
Entrez la hauteur :
3
******
*    *
******
```

4.5 Towers

Cet exercice comprends plusieurs sous-exercices découpés en niveaux. Faites les niveaux dans l'ordre afin d'atteindre le plus haut niveau et éventuellement faire le bonus.

Le principe de cet exercice est de pouvoir générer un affichage d'une tour dessinée à l'aide de caractère entièrement personnalisée par l'utilisateur. Vous allez donc devoir utiliser toutes les connaissances que vous avez acquises sur le C# jusqu'à maintenant.

4.5.1 Niveau 1 : Récupérer les paramètres de l'utilisateurs

Vous allez devoir coder une fonction qui demande à l'utilisateur d'entrer trois types de chaînes de caractères et ainsi les stockers afin de pouvoir les réutiliser plus tard :

- Le haut de la tour (le toit)
- Un étage quelconque de la tour
- Le bas de la tour

Exemple :

```
Entrez le haut de la tour :
.---------.
Entrez un étage quelconque :
|  O   O  |
Entrez le bas de la tour :
|___/"\___|
```

4.5.2 Niveau 2 : Afficher la tour (un seul étage)

Réutilisez les paramètres de l'utilisateurs précédemment stockés afin d'afficher successivement le haut, un étage, et le bas de la tour.

Exemple :

```
Votre tour :
.---------.
|  O   O  |
|___/"\___|
```

4.5.3 Niveau 3 : Afficher la tour (plusieurs étages)

Modifiez le code précédent afin de demander à l'utilisateur de rentrer le nombre d'étage qu'il souhaite afficher et générer l'affichage en conséquence.

Exemples :

```
Entrez le nombre d'étage que vous souhaitez afficher :
3
Votre tour :
.---------.
|  O   O  |
|  O   O  |
|  O   O  |
|___/"\___|
```

```
Entrez le nombre d'étage que vous souhaitez afficher :
0
Votre tour :
.---------.
|___/"\___|
```

4.5.4 Niveau 4 : Ajout d'un étage vide

Modifiez votre code afin de demander à l'utilisateur d'entrer un étage vide et d'afficher la tour finale en ajoutant cette séparation entre chaque partie de la tour. Regardez l'exemple ci-dessous si vous ne voyez pas comment va être le nouvel affichage.

Exemple :

```
Entrez le haut de la tour :
.---------.
Entrez un étage décoré :
|  O   O  |
Entrez un étage vide :
|         |
Entrez le bas de la tour :
|___/"\___|
Entrez le nombre d'étage que vous souhaitez afficher :
3
Votre tour :
.---------.
|         |
|  O   O  |
|         |
|  O   O  |
|         |
|  O   O  |
|         |
|___/"\___|
```

```
Entrez le nombre d'étage que vous souhaitez afficher :
0
Votre tour :
.---------.
|         |
|___/"\___|
```

4.5.5 Bonus : Plusieurs tours

En plus d'afficher plusieurs étages, vous devrez dans cette partie pouvoir afficher plusieurs tours côte à côte. Demandez à l'utilisateur de rentrer le nombre de tours désirées, puis pour chaque tour demandez le nombre d'étage à afficher et affichez-les, séparées par un espace.

Exemple :

```
Entrez le nombre de tour à afficher :
4
Entrez le nombre d'étage de la tour 1 :
3
Entrez le nombre d'étage de la tour 2 :
1
Entrez le nombre d'étage de la tour 3 :
2
Entrez le nombre d'étage de la tour 4 :
4
Vos tours :
                                        .---------.
                                        |         |
.---------.                             |  O   O  |
|         |                             |         |
|  O   O  |              .----------.   |  O   O  |
|         |              |          |   |         |
|  O   O  | .----------. |  O   O   |   |  O   O  |
|         | |          | |          |   |         |
|  O   O  | |  O   O   | |  O   O   |   |  O   O  |
|         | |          | |          |   |         |
|___/"\___| |___/"\____| |___/"\____|   |___/"\___|
```