

TP C#3 : BOUCLES ET DEBUGGER

1 Consignes de rendu

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

```
rendu-tp-login_x.zip
|-- login_x/
|   |-- AUTHORS*
|   |-- README
|   |-- TPCS3.sln*
|   |-- TPCS3*
|   |-- Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer "login_x" par votre propre login. Les fichiers annotés d'une astérisque sont *OBLIGATOIRES*.

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

- Le fichier **AUTHORS** doit être au format habituel (une *, un espace, votre login et un retour à la ligne).
- Pas de dossiers **bin** ou **obj** dans le projet.
- Respectez scrupuleusement les prototypes demandés ! Si le nom des fonction ou les types ne sont pas corrects, l'exercice sera compté comme faux.
- Retirez *tous les tests* de votre code.
- **Le code doit compiler !**

2 Introduction

2.1 Objectifs

L'objectif de ce TP est vous l'aurez compris de vous faire manipuler le debugger de Visual Studio. C'est un outil très puissant auquel vous ferez très souvent appel lors de la réalisation de votre projet cette année. Un soin particulier à l'explication des principes de base a été apporté lors de la réalisation de ce sujet, notez néanmoins que les liens fournis en bas de page sont un bon complément au cours dispensé aujourd'hui. En sortant de cette séance nous voulons que vous soyez autonome quant à la résolution de bug sur du code qui semble à première vue correct, aussi veuillez ne pas prêter une grande importance aux bouts de code que vous ne comprenez pas dans la tarball qui vous a été fournie. Ces bouts de codes sont là justement pour vous forcer à utiliser le debugger lorsque vous aurez abandonné l'idée de résoudre l'exercice sans.

Pour mettre vos nouvelles connaissances à l'épreuve, ainsi que réviser ce que vous avez appris sur les boucles lors du TP2, deux exercices d'implémentation vous seront proposés, pour lesquels le debugger sera d'une utilité flagrante.

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it ? – Brian Kernighan

3 Cours : Le debugger

À quoi sert l'outil dont nous parlons depuis tout à l'heure ? Si son nom n'est pas déjà suffisamment explicite il permet de « retirer les bugs » de votre programme. Imaginez que vous ayez maintenant accès à toute la machinerie depuis l'intérieur, il vous est alors possible de suivre l'exécution de votre programme *à votre rythme*. Voici une liste non exhaustive de ce que vous saurez faire à la fin de cette séance :

- Mettre en pause votre programme pendant son exécution.
- Mettre en évidence le contenu de toutes vos variables à un moment donné.
- Modifier le contenu de ces variables pendant l'exécution.
- Modifier le comportement de votre programme pendant son exécution.
- Revenir en arrière dans l'exécution de votre programme.
- Mettre en évidence la série de fonctions appelées pour arriver où vous vous trouvez actuellement dans le programme.
- ...

Cette liste est loin d'être exhaustive mais prétendre à présenter efficacement l'ensemble des fonctionnalités d'un outil aussi complexe en une seule séance n'est pas raisonnable. Notez cependant que cette liste couvre 95% — pour ne pas dire la totalité — des cas d'utilisations que vous en ferez cette année.

3.1 Breakpoint

Imaginez votre programme comme un livre que votre processeur lirait de façon non linéaire comme vous le feriez avec un dictionnaire, le *breakpoint* ou « point d'arrêt » (vous ne voulez pas utiliser ce terme) serait alors un marque page dans ce livre indiquant au processeur de s'arrêter dès qu'il tombe dessus. Vous placez donc un *breakpoint* sur une déclaration de fonction pour mettre en pause l'exécution du programme à chaque fois qu'un appel vers cette fonction est fait, sur le keyword *while* pour vous arrêter à chaque tour d'une boucle... Vous avez compris le principe. Pour placer un *breakpoint* cliquez dans la marge gauche en face de la ligne souhaitée. S'affiche un rond rouge indiquant que le *breakpoint* est en place (Figure ??).

Pour le désactiver sans le supprimer lors de vos différents tests « cliquez droit » dessus et sélectionnez l'action correspondante, le pictogramme devrait passer à un rond cette fois blanc à contour rouge. Après avoir placé correctement vos *breakpoints* vous lancez l'exécution du programme comme d'habitude avec la flèche « play » verte, préférez utiliser le raccourci clavier en pressant simplement la touche `<F5>` ou `Shift+<F5>` si vous étiez déjà en exécution et que vous vouliez relancer depuis le début le programme. Vous comprenez maintenant que *vous utilisez le debugger depuis le début* sans le savoir en ne mettant simplement pas de *breakpoints*.

3.2 Seek'n Destroy

Arrêter le programme oui, mais pour quoi faire ? Le but de placer un *breakpoint* est de pouvoir prendre le temps d'analyser le contexte du programme¹ et de faire ce qu'on appelle de l'exécution pas à pas. Quand vous exécutez un programme hors du debugger cela se fait en une fraction de seconde selon la lourdeur des calculs² et la puissance de votre machine³. Grâce à l'exécution pas à pas, peu importe la puissance de votre machine c'est vous qui décidez quand passer à la prochaine instruction à exécuter. Sur le listing de la Figure ?? le *breakpoint* est placé sur la ligne 21, dire que l'on avance une fois revient à exécuter ce qui se trouve sur la ligne en

1. dans quelle fonction se trouve-t-on, quelles sont les valeurs contenues dans les variables accessibles à ce moment ?

2. Complexité algorithmique

3. La vitesse de l'horloge, la quantité de RAM etc

cours *puis* avancer sur la ligne suivante et attendre. Cependant, que se passe-t-il si la ligne en cours n'est pas une simple assignation de valeur comme ici mais un appel vers une fonction ? Et bien c'est à vous de choisir, en général tout bon debugger propose trois types de saut vers la prochaine instruction :

- Le *step over* : Exécute la ligne en cours et passe à la suivante sans entrer dans le possible appel à une fonction se trouvant sur la ligne.
- Le *step into* : C'est l'inverse, cette fois le debugger vous amène au coeur de la fonction appelée, s'occupe pour vous de changer le contexte du programme... Bref le debugger vous facilite la vie pour que vous puissiez vous concentrer sur le plus important : pourquoi votre programme n'a pas compris ce que vous lui demandiez de faire.
- Le *step out* : Continue l'exécution jusqu'à la fin de la fonction et retourne à la méthode appelante.

C'est à vous de savoir si il est préférable de sauter au dessus d'un bout de code ou pas suivant ce que vous cherchez ou avez déjà vérifié. Le *step over* se fera avec la touche <F10>, alors que le *step into* avec la touche <F11>. Il existe des boutons correspondant à ces deux raccourcis claviers mais ils ne sont volontairement pas indiqués ici car il n'est pas de bonne pratique que de ralentir l'étape de debugging avec des clics inutiles, on en fait déjà suffisamment sous windows.

Finalement, sachez que si vous êtes allés trop loin dans la fonction, le debugger de visual studio sauvegarde les contextes précédents par défaut, il vous est donc possible de remonter de plusieurs étapes dans le programme simplement en glissant vers le haut la flèche se trouvant sur le break point quand le programme est en pause.

3.3 Smile you're on CCTV

Maintenant que l'on sait avancer dans notre programme à la vitesse que l'on veut il serait bon de pouvoir analyser notre environnement et déceler les éventuels problèmes⁴ ; Il existe plusieurs méthodes, toutes ont des avantages et inconvénients et doivent être utilisées au bon moment. Avant toute chose sachez qu'il est difficile d'accéder à des variables qui ne sont pas dans le scope courant. Aussi ne cherchez pas à connaître la valeur contenue dans une variable d'une fonction précédemment appelée et terminée, cela n'a pas de sens.

- La méthode naïve : Survolez la variable avec votre souris, attendez un cours instant et observez une fenêtre vous indiquant la valeur contenue dans la variable. Recommencez ceci autant de fois que nécessaire... Ou utilisez la méthode *post-it*.
- La méthode *post-it* : Quand vous survolez la pop-up de la méthode précédente cliquez sur la punaise se trouvant sur la partie gauche de la fenêtre pour la rendre persistante une fois que votre souris ne la survolera plus.
- Les *watch*⁵ : Ce sont des variables que vous pouvez créer à la volée pendant l'exécution de votre programme. Vous pouvez les assigner à une variable ce qui est un peu réducteur (revient à faire du *post-it*) ou alors vous pouvez créer des variables plus complexes ou qui ne se trouvent pas directement telles quelles dans votre code. Un exemple sans intérêt mais néanmoins didactique serait de surveiller la somme de deux variables « a » et « b » disponibles dans le corps de la fonction, ce qui évite d'avoir à faire le calcul de tête à chaque fois.

4. ceci est un problème : 42 / 0

5. MSDN: Using the *watch* Window

3.4 Maybe, maybe not

Vous vous êtes peut être déjà demandé comment procéder si nous mettons un *breakpoint* dans une boucle et que le programme devenait instable à partir de la 1337ème itération ? Il existe deux méthode, la première est d’avoir beaucoup de patience et de solides doigts pour marteler 1336 fois la touche <F5>. La seconde méthode plus intéressante permet de mettre une condition sur un *breakpoint*. Pour cela « cliquez droit » sur le *breakpoint* et sélectionnez l’item correspondant à l’ajout d’une condition. Dans l’exemple suivant la condition de *breakpoint* pour atteindre la 1336 ème itération, c’est à dire celle qui précède l’itération de la boucle contenant le bug serait `i == 1336` ;

```
int i = 0;
while (i < 4242)
{
    if (i > 1336)
        // BUG HERE
    i++;
}
```

3.5 You’re the only master onboard

Via la fenêtre nommée « locals » vous avez un aperçu en temps réel de l’ensemble de votre contexte. Et rien ne vous empêche de le modifier quand bon vous semble en éditant la ligne correspondante à la variable à modifier (Figure ??).

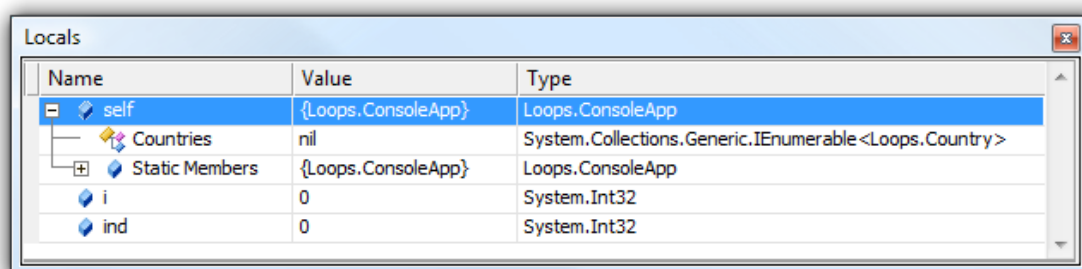
Cela peut s’avérer très utile pour modifier le comportement de votre programme au *runtime*⁶.

6. pendant son exécution

3.6 Figures

```
18 private void button1_Click(object sender, EventArgs e)
19 {
20     int LetterCount = 0;
21     string strText = "Debugging";
22     string letter;
23
24     for (int i = 0; i < strText.Length; i++)
25     {
26         letter = strText.Substring(i, 1);
27
28         if (letter == "g")
29         {
30             LetterCount++;
31         }
32     }
33
34     textBox1.Text = "g appears " + LetterCount + " times";
35 }
```

FIGURE 1 – *breakpoint* sur la ligne 21, programme arrêté



| Name | Value | Type |
|----------------|--------------------|---|
| self | {Loops.ConsoleApp} | Loops.ConsoleApp |
| Countries | nil | System.Collections.Generic.IEnumerable<Loops.Country> |
| Static Members | {Loops.ConsoleApp} | Loops.ConsoleApp |
| i | 0 | System.Int32 |
| ind | 0 | System.Int32 |

FIGURE 2 – La fenêtre « locals »

4 Exercices

4.1 Exercice 1 : Le debugger, les bases

Ouvrez le projet visual studio fourni avec ce TP. Dans la fonction main de votre projet se trouvent plusieurs lignes de code pour la plupart commentées. La première est déjà dé-commentée pour que vous puissiez faire cet exercice. Pour chacun des exercices nous vous conseillons de n'activer que la ligne correspondante au numéro afin de vous éviter de parcourir trop de code au *step-by-step* si votre *breakpoint* initial s'avérait mal placé ou maintenant obsolète. Placez donc votre premier *breakpoint* sur la ligne non commentée et à vous de jouer.

```
Exercices._1___Basics.ex1.run();  
//Exercices._1___Basics.ex2.run();  
//Exercices._1___Basics.ex3.run();
```

Votre but est de dire ce que font les fonctions de chaque exercice dans le header correspondant, voici celui de l'exercice 1 :

```
/// <summary>  
/// COMPLETE ME  
/// </summary>  
public static unsafe void run()  
{  
    int my_nbr = 41;  
    utils.fex1(&my_nbr);  
    Console.WriteLine("Function ex1 terminated.");  
}
```

Pour cela vous devez comprendre le comportement des fonctions appelées, il serait donc judicieux de comprendre ici ce que fait « fex1 ». Vous l'aurez compris ni le code, ni le nom des fonctions ne sont là pour vous aider. Dernier rappel qui sera valable pour le reste de la séance sauf mention explicite : **vous n'êtes pas autorisés à modifier le code**, si vous voulez modifier la variable « my_nbr » afin de faire différents tests par exemple alors référez vous à la section ??.

4.2 Exercice 2 : Le debugger, niveau intermédiaire

Dans cette série d'exercices votre but n'est plus de comprendre simplement la fonction, vous devrez la debugger au sens propre du terme. Ces fonctions ne réagissent pas comme elles devraient et c'est à vous de trouver où cela ne va pas. Vous êtes *exceptionnellement autorisés à ajouter une seule ligne de code* au sein de la méthode afin de la corriger.

Hint

On va principalement parler de string ^a, veuillez donc à avoir quelques connaissances dessus. Encore une fois ne tenez pas compte de la syntaxe des fonctions, le but est que vous résolviez l'exercice avec le debugger, pas grâce au code.

a. String datatypes

4.3 Exercice 3 : Le debugger, niveau avancé

Puisque tout le monde est prêt pour le plat de résistance voici quelques exercices qui demandent un peu plus de réflexion !

4.3.1 Crypto-Box

Un message chiffré se trouve en mémoire, retrouvez l'original sans toucher au code. Vous spécifierez la phrase déchiffrée dans le fichier README.

```
ciphered = "bw wvu urr'k demd hfz xf vfbl, ak oirlk iuon ksn mf blblj";  
key = "tryharder";
```

4.3.2 The Bomb : Bonus

À force de vouloir jouer dans votre code vous avez déclenché la mise sous tension d'une bombe hautement dangereuse pour laquelle vous devrez faire appel à tout ce que vous savez afin de la désamorcer et ainsi sauver la ville.

Votre but est de fournir à la bombe les codes de désamorçage dans le bon ordre afin de la désactiver. Vous n'avez pas le droit de modifier le code directement. Vous spécifierez entièrement votre démarche pour désamorcer la bombe dans le fichier README.

Hint

Il vous est spécifié (en fonction du niveau au sein de la bombe) le type d'input attendu :

- Digit [0-9] : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- string : String Class

N'hésitez pas à faire remonter vos impressions bonnes comme mauvaises sur la partie debugger par email :

To : Naam <elasma_n@epita.fr>

Subject : [SUP]/[TPC3] feedback debugger

4.4 Exercice 4 : Happy

En mathématiques, un entier naturel est un nombre heureux si, lorsqu'on calcule la somme des carrés de ses chiffres dans son écriture en base dix puis la somme des carrés des chiffres du nombre obtenu et ainsi de suite, on aboutit au nombre 1. – Wikipedia

Par exemple, 19 est un nombre heureux :

— $1^2 + 9^2 = 1 + 81 = 82$

— $8^2 + 2^2 = 64 + 4 = 68$

— $6^2 + 8^2 = 36 + 64 = 100$

— $1^2 + 0^2 + 0^2 = 1 + 0 + 0 = 1$

Vous l'aurez compris, cet exercice va manipuler des nombres heureux. Le but final de cet exercice est d'implémenter la fonction `public static void checkHappy()` ; qui va demander à l'utilisateur d'entrer un nombre entier, puis va afficher un message disant si le nombre entré est heureux. La fonction doit continuer à demander un nouveau nombre jusqu'à ce que l'utilisateur entre la valeur 0.

Voici un exemple d'utilisation :

```
Enter an integer (0 to stop): 69
69 is not happy...
Enter an integer (0 to stop): 1337
1337 is happy!
Enter an integer (0 to stop): 42
42 is not happy...
Enter an integer (0 to stop): coucou
Enter an integer (0 to stop): 0
```

La fonction `checkHappy` repose sur 3 autres fonctions que vous devez implémenter. La première de ces 3 fonctions va simplement demander à l'utilisateur d'entrer un nombre entier. Mais que ce passe-t-il si'il veut faire le malin et entre "salut!"...? Vous allez ici alors demander à nouveau, jusqu'à ce qu'il se soit enfin décidé à entrer quelque chose de correct.

Hint

Pour ce faire, regardez du côté de `Int32.TryParse()` :

```
string str = Console.ReadLine();
int n = 0;

if (Int32.TryParse(str, out n))
    // str is a number, and n is the integer represented by str
```

Votre fonction prend en paramètre une string qui devra être affichée à chaque fois avant de demander d'entrer un nombre. Le prototype de votre fonction doit être : `public static int readInt(string prompt);`.

Voici un exemple d'utilisation :

```
int num = readInt("Enter an integer: ");
Console.WriteLine("I love " + num + " too!");
```

```
Enter an integer: Salut !
Enter an integer: ça va ?
Enter an integer: 42.0001
Enter an integer: 1337
I love 1337 too!
```

Ensuite, écrivez la fonction `public static int sumOfSquaredDigits(int n);` qui calcule la somme des carrés des chiffres qui composent l'entier `n` passé en paramètre. Par exemple, pour $n = 24$, on aura $2^2 + 4^2 = 20$.

Continuez avec la fonction `public static bool isHappy(int n);` qui retourne vrai si `n` est un nombre heureux, faux sinon.

Hint

Vous aurez sûrement besoin d'aller fouiller du côté de la classe `List<T>`... (la doc sera toujours votre meilleur amie!)

Vous pouvez enfin implémenter la fonction `checkHappy` décrite au début de cet exercice.

4.5 Exercice 5 : Triangle de pascal

Comme vous le savez tous, le triangle de Pascal est un tableau triangulaire, qui permet de retrouver les coefficients binomiaux. Le coefficient de la $i^{\text{ème}}$ ligne, $j^{\text{ème}}$ colonne est la somme des $j - 1^{\text{ème}}$ et $j^{\text{ème}}$ coefficients de la $i - 1^{\text{ème}}$ ligne.

Voici à quoi ressemble le début du triangle de Pascal :

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

La fonction que vous devez implémenter doit afficher un certain nombre de lignes du triangle de Pascal. Elle doit suivre le prototype : `public static void pascal(int n);` où n représente le nombre de que vous devez afficher.

Bullshit is strong with this one...