Travaux Pratiques de programmation n°2

Cours de programmation impérative

—Licence MPI L1 S2 - Info 121—

Utilisation du Débogueur

Dans cette séance nous allons apprendre à utiliser le débogueur. C'est un outil de développement qui permet d'interrompre temporairement un programme en cours d'exécution, d'afficher les valeurs des variables ainsi que l'état de la pile des appels de fonctions.

Le débogueur de `Code::Block`

S'il est possible de travailler sur un fichier `C/C++` seul sous `Code::Block`, pour pouvoir déboguer, nous allons devoir créer un projet. On conseille de créer un nouveau projet pour chaque exercice. Pour ceci, on clique sur « Create a new project ». Choisir ensuite « Console application » comme type de projet, puis « C++ » comme langage. Et enfin choisir un répertoire pour stocker le projet. On peut ensuite copier le fichier de l'énoncé dans le fichier créé par `Code::Block`.

Le dégogueur s'utilise à l'aide de la barre de menu de débogage dont voici une copie d'écran (selon votre version de `Code::Block`, elle peut différer légèrement) :

Nous allons utiliser les boutons suivants :

- Debug/Continue : lance le programme en mode débogage ou continue un programme interrompu ;
- Run to Cursor : exécute le programme en mode débogage jusqu'au curseur ;
- Next Line : exécute jusqu'à la ligne suivante ;
- Step Into : exécute l'appel de fonction ;
- Step Out : exécute jusqu'au retour de la fonction ;
- Stop Debugger : arrête le programme ;
- Debugging Window : configure les fenêtres visibles.
  Nous allons utiliser les deux fenêtres suivantes :
  — Watches : qui affiche les valeurs des variables locales et paramètres ;
  — Call Stack : qui affiche la pile d'appels.

Exemple d'utilisation du débogueur de `Code::Block`

Le programme `exponentielle.cpp` affiche les valeurs de la fonction exponentielle entre 0 et 2 par pas de 0.2. Il calcule l'exponentielle grâce à la formule :

$$\exp(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots + \frac{x^n}{n!} + \ldots$$

1. En plaçant le curseur sur la première ligne du corps de la fonction `power`, lancer le débogueur avec «Run to Cursor» ;
2. Ouvrir les fenêtres «Watches» et «Call Stack» pour inspecter les valeurs des variables et la pile d'appels des fonctions ;

3. Appuyer plusieurs fois sur «Next Line» et observer les valeurs des variables qui évoluent ;
4. en appuyant sur le bouton droit à l'intérieur du programme, mettre un point d'arrêt («toggle breakpoint») à l'intérieur de la fonction factorielle ;
5. continuer le programme jusqu'au point d'arrêt.

▸ **Exercice 1. (Utilisation du débogueur)**

On considère le polynôme $P = 1.5 + 2.5x + 2.5x^2 + 4.0x^3$. On peut calculer les valeurs suivantes :

x	0	1	1.5	2
$P(x)$	1.5	10.5	24.375	48.5

Le programme `polybug.cpp` de l'archive est censé calculer ces valeurs mais retourne un résultat faux. Utiliser le débogueur pour découvrir les erreurs et les corriger.

▸ **Exercice 2. (Calcul de la racine carrée)**

Soit a un nombre réel positif. La racine carrée $b = \sqrt{a}$ de a est l'unique nombre réel positif qui vérifie $b^2 = a$. On montre en mathématiques que, étant donné un réel positif a, la suite

$$u_0 := a, \qquad u_{n+1} := \frac{u_n + a/u_n}{2}$$

converge vers $\sqrt{a}$. Le programme suivant affiche les 10 premiers termes de la suite u_n ainsi que u_n^2 pour $a = 2$:

```
1   #include <iostream>
2   #include <iomanip>
3
4   using namespace std;
5
6   int main(void)
7   {
8     int n;
9     float a=2., un, un1; // u_n et u_{n+1}
10
11    n = 0;  un = a;
12    while (n<10) {
13      un1 = (un + a/un)/2.;       // calcul de u_{n+1}
14      n = n+1; un = un1;          // passage de n à n+1
15      cout << "n=" << n << ", un=" << un << ", un^2=" << un*un << endl;
16    }
17    return 0;
18  }
```

1. Vérifier que u_n converge bien vers $\sqrt{2}$, et u_n^2 vers 2.
2. Augmenter la précision de l'affichage (en affichant `setprecision(10)` avant `un` et `un*un` et lancer le calcul. Que remarquez-vous ?

À cause des erreurs d'arrondi, u_n^2 ne tombe jamais sur 2 mais sur un nombre très proche. Il faut donc trouver un moyen d'arrêter le calcul au bon moment. Pour cela, on fixe une précision, par exemple $\epsilon = 10^{-6}$, ce qui s'écrit en C++ (au début du programme) :

```
const float EPSILON = 1e-6;
```

et on continue le calcul tant que u_n^2 n'est pas égal à a à ϵ près, c'est-à-dire tant que

$$\left|\frac{u_n^2}{a} - 1\right| \geq \epsilon\,.$$

3. Modifier le programme précédent pour calculer la racine carrée, avec une précision de ϵ, d'un nombre a donné par l'utilisateur. Si le nombre donné par l'utilisateur est négatif on affichera un message d'erreur puis on demandera un nouveau nombre.

▸ **Exercice 3. (Algorithme d'Euclide)**

L'algorithme d'Euclide est probablement l'un des plus vieux algorithmes (300 ans avant J.-C.). Il calcule le Plus Grand Commun Diviseur (PGCD) de deux entiers naturels non nuls a et b. Supposons que $a > b > 0$, et notons r le reste de la division de a par b. Alors

— si $r = 0$, c'est que b divise a et donc que le PGCD de a et b est égal à b ;
— sinon le PGCD de a et b est égal au PGCD de b et r. On recommence donc le calcul en posant $a \leftarrow b$ et $b \leftarrow r$.

Voici un exemple où $a = 96$ et $b = 36$:

a	b	r
96	36	24
36	24	12
24	12	0

le PGCD de 96 et 36 est donc 12.

1. Écrire un programme qui calcule le Plus Grand Commun Diviseur (PGCD) de deux entiers naturels non nuls a et b.
2. En déduire un programme qui simplifie les fractions.

▸ **Exercice 4.** Le tableau suivant contient les notes (entre 0 et 5) d'une classe d'élèves.

```
vector<int> notes = {3, 1, 3, 1, 2, 2, 5, 5, 5, 2, 3, 2, 2, 1, 1, 1, 2,
                     0, 5, 2, 4, 3, 5, 5, 1, 5, 2, 5, 1, 0, 2, 2, 1, 5};
```

1. Écrire un programme qui affiche la répartition des notes comme suit :

```
0  1  2  3  4  5
2  8 10  4  1  9
```

car 2 élèves ont `0`, 8 élèves ont `1`, 10 élèves ont `2`, etc.

2. Reprendre l'exercice précédent et faire un histogramme sous la forme de bandes horizontales comme suit :

```
0 : **
1 : ********
2 : **********
3 : ****
4 : *
5 : *********
```