université PARIS-SACLAY

Correction TP 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 `gdb`

Pour pouvoir déboguer, un programme il faut passer l'option «`-g`» au compilateur. Ainsi si l'on veut compiler le programme `C++ toto.cpp` on écrira :

```
g++ -Wall -std=c++11 -g toto.cpp -o toto
```

Ensuite, on lance le débogueur avec la commande « `gdb -tui toto` » qui affiche dans le terminal une fenêtre coupée en deux. Le haut présente le *code du programme*, le bas est la *zone de commande* où l'on va interagir avec le programme en cours d'exécution.

raccourcis clavier indispensables :

- `<ctrl>+<l>` : ré-affiche la console. C'est souvent utile quand les affichages du programme sont mélangés avec le deboggage ;

Pour éviter d'avoir à retaper une commande on utilise :

- `<entrée>` : répète la dernière commande ;
- `<ctrl>+<p>` : (anglais *previous*) retrouve la commande précédente dans l'historique des commandes ;
- `<ctrl>+<n>` : (anglais *next*) retrouve la commande suivante dans l'historique des commandes.

Contrôler l'exécution du programme avec `gdb` : Voici la liste des commandes du débogueur que nous allons utiliser. Après chaque commande, on a écrit l'abréviation entre parenthèses :

- `run (r)` : lance le programme ;
- `break` *n* `(b)` : ajoute un point d'arrêt (anglais *break point*) à la ligne *n* du programme ;
- `info break` : affiche la liste des points d'arrêt ;
- `del` *n* : supprime le point d'arrêt *n* ;
- `step (s)` : avance d'une ligne dans l'exécution du programme ;
- `finish` : finit l'exécution de la fonction en cours ;
- `cont (c)` : continue l'exécution du programme jusqu'à la fin ou au prochain point d'arrêt ;
- `quit (q)` : quitte de deboggeur.

Afficher l'état du programme :

- `print` *nom* `(p)` : affiche la valeur de la variable *nom* ;
- `display` *nom* : affiche automatiquement la valeur de la variable *nom* à chaque arrêt ;
- `del display` *n* : supprime le *n*-ième affichage automatique ;
- `info stack (info s, bt)` : (anglais *backtrace*) affiche la pile des appels avec les valeur des paramètres ;
- `info stack full` : affiche la pile des appels avec les valeur des paramètres et des variables locales ;

Exemple d'utilisation du débogueur gdb

```
1   #include <iostream>
2   #include <iomanip>
3
4   using namespace std;
5
6   float power(float x, unsigned int n) {
7       unsigned int i;
8       float res = 1.;
9       for (i=0; i<n; i++)
10          res *= x;
11      return res;
12  }
13
14  float factorial(int n) {
15      float res = 1;
16      for (int i = 1; i<= n; i++) res = res*i;
17      return res;
18  }
19
20  float exponentiel(float x) {
21      const float epsilon = 1e-7;
22      float res = 1., term;
23      int i = 1;
24      do {
25          term = power(x, i) / factorial(i);
26          res += term;
27          i++;
28      } while ((term / res) > epsilon);
29      return res;
30  }
31
32
33  void affiche() {
34      cout.precision(3);
35      cout << fixed;
36
37      for (double x=0; x<2; x=x+0.2)
38          cout << setw(5) << x << " ";
39      cout << endl;
40      for (double x=0; x<2; x=x+0.2)
41          cout << setw(5) << exponentiel(x) << " ";
42      cout << endl;
43  }
44
45  int main() {
46      affiche();
47      return 0;
48  }
```

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. Pour pouvoir travailler confortablement, il est utile d'avoir la fenêtre de terminal **la plus haute possible**. Il faut donc agrandir en hauteur la fenêtre au maximum de manière à couvrir un moitié de l'écran (par exemple à droite). Le sujet occupera l'autre moitié de l'écran.
2. Compiler puis lancer le déboggeur sur le programme `exponentielle`.
3. Placer un point d'arrêt (`break`) sur la première ligne de la fonction `power` ;
4. Lancer le programme dans le débogueur (`run`) ;
5. Afficher la pile d'appel (`info stack`) ;
6. Afficher la valeur de la variable `res` (`print`) ;
7. Demander un affichage automatique (`display`) des variables `res` et `i` ;
8. Faire avancer (`step`) le programme plusieurs fois d'une étape et observer l'évolution des variables. En particulier, observer que les variables locales contiennent des valeurs aléatoires en entrée des fonctions.
9. Continuer le programme jusqu'au point d'arrêt (`cont`) ;

▸ **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. On va utiliser le débogueur pour suivre l'exécution du programme ainsi que la valeur des variables afin de découvrir les erreurs et les corriger.

1. Lire la documentation de chaque fonction pour comprendre ce qu'elles sont sensées faire.
2. Lancer le debugger et observer le comportement de chaque fonction. Indication : mettre un point d'arrêt au début de la fonction puis utiliser `cont` jusqu'à être dans un cas d'appel non évident. Par exemple, on ne verra rien d'intéressant dans la fonction puissance si l'on calcule les puissances de 0 ou de 1 ou un exposant 0 ou 1.

✂ ----------

Voici la version corrigée

```
1   #include <iostream>
2   #include <vector>
3
4   using namespace std;
5
6
7   /** Puissance d'un nombre réel
8    *  @param n un entier positif
9    *  @param a un nombre réel
10   *  @return a^n
11   **/
```

```
12  double power(double a, unsigned int n) {
13     double res = 1.;
14     for (unsigned int i = 0; i<n; i++)
15       res = res*a;
16     return res;
17  }
18
19  /** Calcule la valeur d'un polynôme en un nombre réel
20   *  @param pol un vecteur qui contient les coefficients du polynôme
21   *  @param x un nombre réel
22   *  @return pol(x)
23   **/
24  double evalpoly(vector<double> pol, double x) {
25     double res = 0;
26     for (unsigned int i = 0; i < pol.size(); i++)
27       res = res + pol[i]*power(x, i);
28     return res;
29  }
30
31  int main() {
32     vector<double> pol = {1.5, 2.5, 2.5, 4.0};
33     vector<double> vals = {0, 1, 1.5, 2};
34     for (unsigned int i = 0; i < vals.size(); i++)
35       cout << "P(" << vals[i] << ") = " << evalpoly(pol, vals[i]) << endl;
36     return 0;
37  }
```

avec la différence

```
--- polybug.cpp 2015-02-03 14:33:58.078403025 +0100
+++ polybug_correction.cpp 2015-02-03 14:34:15.882531078 +0100
@@ -10,7 +10,7 @@
  *  @return a^n
  **/
 double power(double a, unsigned int n) {
-   double res = 0.;
+   double res = 1.;
    for (unsigned int i = 0; i<n; i++)
      res = res*a;
    return res;
@@ -23,7 +23,7 @@
  **/
 double evalpoly(vector<double> pol, double x) {
    double res = 0;
-   for (unsigned int i = 1; i <= pol.size(); i++)
+   for (unsigned int i = 0; i < pol.size(); i++)
      res = res + pol[i]*power(x, i);
    return res;
 }
```

▸ **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 (`suite_sqrt.cpp` dans l'archive) 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()
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. Observer l'affichage pour vérifier que u_n converge bien vers $\sqrt{2}$, et u_n^2 vers 2.
2. Modifier le programme précédent pour que le calcul se fasse à partir d'un nombre a donné par l'utilisateur. Si le nombre donné par l'utilisateur est négatif on demandera un nouveau nombre.
3. Augmenter la précision de l'affichage en ajoutant `<< setprecision(10)` dans le `cout` qui affiche u_n (avant l'affichage de u_n), et lancer le calcul. Essayez de calculer les racines carrées de 2, 1.1 et 2000000. Que remarquez-vous ?

À cause des erreurs d'arrondi, dans le cas du calcul de $\sqrt{2}$, la valeur de u_n^2 ne tombe jamais sur 2 mais sur un nombre très proche. De plus, en fonction de a, les 10 itérations de la boucle sont pas suffisantes pour avoir une bonne valeur approchée de $\sqrt{a}$. 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

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

4. Modifier le programme précédent pour calculer à ϵ près la racine carrée du nombre a donné par l'utilisateur.

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

const float epsilon = 1e-6;

int main()
{
  float a, un, erreur;
  do {
    cout << "Donnez un nombre positif : ";
    cin >> a;
  } while (a <= 0.);

  un = a;
  erreur = (un*un/a) - 1;
  if (erreur < 0) erreur = -erreur;
  while (erreur >= epsilon)
    {
      un = (un + a/un)/2.;
      erreur = (un*un/a) - 1;
      if (erreur < 0) erreur = -erreur;
    }
  cout << "sqrt(" << a << ") = " << un << ", "<< un << "^2=" << un*un << endl;
  return 0;
}
```

▸ **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. 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.

```cpp
#include <iostream>

using namespace std;
```

```
4
5   int main() {
6     int a, b, r;
7     cout << "Donnez deux entiers ";
8     cin >> a >> b;
9     r = a % b;
10    while (r != 0)
11      {
12        a = b;
13        b = r;
14        r = a % b;
15      }
16    cout << "Le PGCD est " << b << endl;
17  }
```

2. En déduire un programme qui simplifie les fractions.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main() {
6     int num, den, a, b, r;
7     cout << "Donnez une fraction : ";
8     cin >> num >> den;
9     a = num;
10    b = den;
11    r = a % b;
12    while (r != 0)
13      {
14        a = b;
15        b = r;
16        r = a % b;
17      }
18    cout << num << "/" << den << " = " << num/b << "/" << den/b << endl;
19  }
```

▸ **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 :

```
1   #include <iostream>
2   #include <iomanip>
3   #include <vector>
```

```
4
5   using namespace std;
6
7   vector<int> notes = {3, 1, 3, 1, 2, 2, 5, 5, 5, 2, 3, 2, 2, 1, 1, 1, 2,
8                        0, 5, 2, 4, 3, 5, 5, 1, 5, 2, 5, 1, 0, 2, 2, 1, 5};
9
10  const int max_note = 5;
11
12  int main(void)
13  {
14    unsigned int i;
15    vector<int> res;
16    res = vector<int>(max_note+1);
17
18    for (i=0; i<=max_note; i++) res[i] = 0;
19    for (i=0; i<notes.size(); i++) res[notes[i]]++;
20    for (i=0; i<=max_note; i++) cout << setw(3) << i;
21    cout << endl;
22    for (i=0; i<=max_note; i++) cout << setw(3) << res[i];
23    cout << endl;
24  }
```

```
  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 : *********
```

```
1   #include <iostream>
2   #include <iomanip>
3   #include <vector>
4
5   using namespace std;
6
7   vector<int> notes = {3, 1, 3, 1, 2, 2, 5, 5, 5, 2, 3, 2, 2, 1, 1, 1, 2,
8                        0, 5, 2, 4, 3, 5, 5, 1, 5, 2, 5, 1, 0, 2, 2, 1, 5};
9
10  const int max_note = 5;
11
12  int main(void)
13  {
14    unsigned int i;
15    vector<int> res;
16    res = vector<int>(max_note+1);
17
```

```
18      for (i=0; i<=max_note; i++) res[i] = 0;
19      for (i=0; i<notes.size(); i++) res[notes[i]]++;
20      for (i=0; i<max_note; i++) {
21        cout << setw(3) << i << " : " ;
22        for (int j=0; j<res[i]; j++) cout << "*";
23        cout << endl;
24      }
25    }
26
```

▸ **Exercice 5.** (**permutation**)

Soit N une constante fixée. Une suite de N nombres est une permutation si chaque nombre entre 0 et $N-1$ apparaît une fois et une seule dans la suite. Par exemple pour $N = 5$ la suite $[1, 4, 0, 3, 2]$ est une permutation alors que les suites $[1, 0, 6, 2, 4]$ et $[2, 1, 0, 4, 2]$ n'en sont pas (car 3 n'apparaît pas). On stockera la permutation dans un vecteur.

On demande d'écrire un programme qui

1. déclare un vecteur de N entiers permettant de stocker la suite ;
2. demande à l'utilisateur de saisir la suite ;
3. affiche à l'écran si la suite saisie est une permutation ou non. Indication : on pourra utiliser un deuxième tableau où l'on place dans la case i le nombre de fois que i apparaît dans la suite donnée.

```
1   #include<stdio.h>
2
3   #define N 5
4   typedef int permutation[N];
5
6   int main(void)
7   {
8     permutation p, ptest;
9     int i, erreur;
10
11    for (i=0; i<N; i++)
12      scanf("%d", &(p[i]));
13
14    erreur = 0;
15    for (i=0; i<N; i++) ptest[i]=0;
16    for (i=0; i<N && !erreur; i++)
17      {
18        if (0<=p[i] && p[i]<N)
19          ptest[p[i]]++;
20        else
21          erreur = 1;
22      }
23    for (i=0; i<N && !erreur; i++)
24      if (ptest[i] != 1)
25        erreur = 1;
26    if (erreur) printf("Pas une permutation\n");
```

```
27      else        printf("Une permutation\n");
28      return 0;
29    }
```