Correction TP de programmation n°4

Cours de programmation impérative

—Licence MPI L1 S2 - Info 121—

▸ **Exercice 1. Mise en place pour Premier Langage**

La semaine prochaine, vous aurez une interrogation sur Premier Langage. Vous devez **impérativement vérifier par vous même** que vous pouvez y accéder. La mise en place des comptes demande un délai, nous **ne pourrons pas vous donner l'accès** le jour même de l'interrogation.

- Pour ceci, vous devez :
  — Aller sur `https://ecampus.paris-saclay.fr/`
  — Vous connecter en cliquant sur «compte établissement» puis «Paris-Sud University».
  — Une fois connecté, vérifier que vous êtes bien inscrit au cours «*Programmation impérative-L1 STS : portail MPI ou double licence Info/Math ou double licence Math/Physique*» (dans ce cas il apparaît sur votre page d'accueil).

- Si vous n'arrivez pas à vous connecter :
  — Cliquez sur : «Activez votre accès à eCampus via votre compte établissement en cliquant ici (création d'un compte dit "mutualisé")»
  — Sélectionnez «Paris-Sud University».
  — Activez le service «Moodle»
  — Et finalement «Valider vos choix».
  L'accès n'est pas immédiat... Vous devez IMPÉRATIVEMENT vérifier le lendemain que cela fonctionne bien et que vous avez bien accès au cours.

- Si vous arrivez à vous connecter, mais que vous n'êtes pas inscrit au cours, cliquez sur le lien suivant : `https://ecampus.paris-saclay.fr/course/view.php?id=17037` Et inscrivez-vous au cours en question.

Les exercices d'entraînement devraient être disponibles à partir de mercredi 27.

ATTENTION ! La procédure d'inscription demande un délai. Si vous êtes en retard, vous prenez le risque de ne pas pouvoir vous connecter ce qui entraînera automatiquement la note de 0, sans rattrapage possible.

Fonctions et procédures: passage de paramètres

Cette séance de travaux pratiques est dédiée au passage de paramètres. On rappelle que en `C++` le mode de passage par défaut est **par valeur**, les arguments sont alors **recopiés**. Le `C++` fournit un autre mode dit **par référence** où l'on fait précéder le paramètre formel du symbole `&`. Voici un exemple : la procédure suivante ajoute 2 à la variable passée en paramètre :

```
void incremente(int &i) {
    i = i + 2;
}
```

▸ **Exercice 2.** La fonction suivante retourne un entier positif donné par l'utilisateur :

```
int litpositif() {
  int resultat;
  do {
    cout << "Donner la valeur d'un  entier positif : ";
    cin >> resultat;
  } while (resultat < 0);
  return resultat;
}
```

1. Modifier la fonction `litpositif` (fournie dans l'archive) pour en faire une procédure qui a comme résultats l'entier positif lu et le nombre d'erreurs (entiers négatifs) faites par l'utilisateur.
2. Compléter le programme principal `main` pour qu'il utilise la procédure précédente et affiche l'entier lu et le nombre d'erreurs. Attention, bien respecter l'énoncé : c'est le `main` qui doit faire les affichages, pas la procédure !

✂ ----------

```
1   #include<iostream>
2   #include<string>
3
4   using namespace std;
5
6   void litpositif(int &i, int &c) {
7       c = -1;
8       do {
9           cout << "Donner la valeur d'un entier positif : ";
10          cin >> i;
11          c++;
12      } while (i < 0);
13  }
14
15  int main() {
16      int i;
17      int c;
18      litpositif(i, c);
19      cout << "L'entier vaut " << i << " après " << c << " essai(s)" << endl;
20      return 0;
21  }
```

▸ **Exercice 3. (Transactions bancaires)**

Nous nous occupons de la gestion de comptes bancaires. Il faut nous assurer qu'aucune somme d'argent n'est créée ou perdue dans les virements et que les comptes restent toujours au-dessus du solde autorisé. Nous disposons des comptes tels que définis dans le fichier `banque.cpp` fourni :

```
1   #include <iostream>
2   using namespace std;
3
4   int compte1 = 1000;
5   int compte2 = 2000;
6   int compte3 = 1500;
7   int compte4 = 3000;
8
9   void etat_comptes();
10  bool virement(int compte_orig, int compte_dest, int somme);
11
12  void etat_comptes() {
13    cout << "Etat des comptes : " << endl;
14    cout << "Compte n 1 : " << compte1 << endl;
15    cout << "Compte n 2 : " << compte2 << endl;
16    cout << "Compte n 3 : " << compte3 << endl;
17    cout << "Compte n 4 : " << compte4 << endl;
18  }
19
20  int main() {
21    bool v;
22    etat_comptes();
23    v = virement(compte1, compte2, 100);
24    etat_comptes();
25    return 0;
26  }
```

1. Définir la fonction :
   `bool virement(int compte_orig, int compte_dest, int somme)`
   qui verse si possible (c'est-à-dire si le versement ne rend pas le solde de `compte_orig` négatif) un montant d'argent égal à `somme` depuis le `compte_orig` vers le `compte_dest`. De plus, cette fonction doit retourner `true` si le versement a été effectué et renvoie `false` sinon.
2. Lire le code du programme principal puis exécuter le programme. Vérifier alors les valeurs des comptes qui ont été passées en paramètres de la fonction `virement`. Une fois cela fait, nous constaterons que les valeurs sont inchangées, alors qu'un virement devait normalement les avoir changées. Ceci est dû au fait que les paramètres ont été passés par valeur à la fonction.
3. Pour résoudre le problème constaté à la question précédente, nous allons transformer la fonction en procédure et passer les paramètres par référence. **Modifier** la fonction `virement` en une procédure :

`void virement(int &compte_orig, int &compte_dest, int somme, bool &virement_ok)`

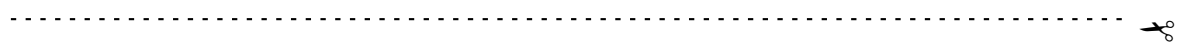
Cette procédure doit avoir le résultat `true` dans le paramètre `virement_ok` si le versement a été effectué et `false` sinon. L'ancienne version de la fonction `virement` ne doit plus figurer dans votre code.

4. Appelez enfin cette procédure dans le programme `main` en affichant «Virement effectué» ou «Virement impossible» selon le cas. Vérifier que cette fois, les valeurs des comptes sont bien changées.

✂ ----------

```
1  #include <iostream>
2  using namespace std;
3
4  int compte1 = 1000;
5  int compte2 = 2000;
6  int compte3 = 1500;
7  int compte4 = 3000;
8
9  void etat_comptes();
10 //bool virement(int compte_orig, int compte_dest, int somme);
11 void virement(int &compte_orig, int &compte_dest, int somme, int &virement_ok);
12
13 void etat_comptes() {
14   cout << "Etat des comptes : " << endl;
15   cout << "Compte n 1 : " << compte1 << endl;
16   cout << "Compte n 2 : " << compte2 << endl;
17   cout << "Compte n 3 : " << compte3 << endl;
18   cout << "Compte n 4 : " << compte4 << endl;
19 }
20
21 /*bool virement(int compte_orig, int compte_dest, int somme) {
22     if (somme <= 0) return false;
23     if (compte_orig - somme < 0) return false;
24     compte_orig -= somme;
25     compte_dest += somme;
26     return true;
27 }*/
28
29 void virement(int &compte_orig, int &compte_dest, int somme, bool &virement_ok) {
30     if (somme <= 0 || compte_orig - somme < 0) virement_ok = false;
31     else {
32         virement_ok = true;
33         compte_orig -= somme;
34         compte_dest += somme;
35     }
36 }
37
38 int main() {
39   bool v;
40   etat_comptes();
41 //  v = virement(compte1, compte2, 100);
42   virement(compte1, compte2, 100, v);
43   etat_comptes();
44   if (v) cout << "Virement effectué !" << endl;
```

```
45      else   cout << "Virement impossible !"<< endl;
46      return 0;
47    }
```



▸ **Exercice 4. Appeler une procédure, appeler une procédure depuis une autre.** Cet exercice reprend une petite partie du TD en la modifiant légèrement.

1. Écrire la procédure `ordonner` qui prend deux variables a et b et échange leur contenu si $a > b$. On impose d'utiliser la procédure suivante :

```
void permuter(int &c, int &d) {
  int temp;
  temp=c; c=d; d=temp;
}
```

✂ ----------

```
void ordonner(int &a, int &b) {
  if(a>b) permuter(a,b);
}
```

---------- ✂

2. Tester cette procédure d'abord avec un appel pour classer deux variables, puis avec trois appels comme en TD, pour classer trois variables.

✂ ----------

```
void test() {
 int x=7, y=5, x=3;
 ordonner(x,y); ordonner(y,z); ordonner(x,y);
 cout <<x<<y<<z<<endl;
}
```

---------- ✂

▸ **Exercice 5. Écriture d'une procédure qui met à jour son paramètre.**

En mathématique, on appelle suite de Syracuse une suite d'entiers naturels définie de la manière suivante : On part d'un nombre entier plus grand que zéro ; s'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1. En répétant l'opération, on obtient une suite d'entiers positifs dont chacun ne dépend que de son prédécesseur.

Par exemple, à partir de 14, on construit la suite des nombres : 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2... C'est ce qu'on appelle la suite de Syracuse du nombre 14.

Après que le nombre 1 a été atteint, la suite des valeurs (1,4,2,1,4,2...) se répète indéfiniment en un cycle de longueur 3, appelé cycle trivial.

1. Écrire la procédure `syracuse` qui prend une variable contenant un terme de la suite, calcule le terme suivant et le range dans la même variable.

✂ ----------

```
void syracuse(int &u) {
  if(u%2 == 0) u = u/2;
  else u = 3*u+1;
}
```

---------- ✂

2. La fonction suivante calcule le nombre de fois qu'il faut itérer la suite, pour arriver sur 1, a partir d'un u_n donné.

```
int longueurTransient (int un){
  int i=0;
  while (un != 1) { syracuse(un); i++; }
  return i;
}
```

Cette fonction pourrait ne pas être définie pour tout les entiers. La conjecture de Syracuse dit que elle l'est, mais personne ne l'a jamais démontré ! Calculer cette fonction sur les nombres de 1 à 1000, et afficher le résultat, pour étudier son comportement.

```
void test(){
  for(int i=1; i<1000; i++)
    cout<<longeurTransient (i)<<", ";
}
```

▸ **Exercice 6. (Nombres complexes)**

On rappelle qu'un nombre complexe est un nombre qui s'écrit $a+ib$ où a est appelé partie réelle et b partie imaginaire. Il faudra systématiquement passer deux `float`, pour passer un nombre complexe en paramètre. Nous verrons la semaine prochaine comment utiliser les `struct` pour passer directement un nombre complexe dans un seul paramètre, et ainsi diviser par deux le nombre de paramètres, et rendre le programme plus lisible. Pour cette semaine, ne pas utiliser de structure.

Attention, dans tout cet exercice, faire bien attention à quels sont les paramètres qui sont des résultats et quels sont ceux qui sont des données. On demande de :

1. Écrire une procédure `saisie` qui demande à l'utilisateur d'entrer un nombre complexe, et qui a pour résultat le nombre complexe entré par l'utilisateur.
2. Écrire une procédure `affiche` qui affiche un nombre complexe.
3. Écrire une procédure `somme` qui calcule la somme de deux nombres complexes.
4. Écrire une procédure `produit` qui calcule le produit de deux nombres complexes. On rappelle que $(a + ib) \times (c + id) = (ac - bd) + i(ad + bc)$.
5. Écrire une fonction `norme_carre` qui retourne le carré de la norme d'un nombre complexe. On rappelle que
$$|a + ib|^2 = a^2 + b^2 \,. \tag{1}$$
6. Écrire une procédure `inverse` qui retourne l'inverse d'un nombre complexe. On rappelle que
$$\frac{1}{a + ib} = \frac{a - ib}{|a + ib|^2} \,. \tag{2}$$

7. L'algorithme de calcul de la racine carrée du TP 2 fonctionne en général encore sur les nombres complexes $z = a + ib$: la suite
$$u_0 := z, \qquad u_{n+1} := \frac{u_n + z/u_n}{2} \tag{3}$$

converge (presque toujours, voir la question suivante) vers une racine carrée de z. Écrire une procédure `racine` qui calcule la racine carrée d'un nombre complexe. On considère que l'approximation u est correcte si

$$\frac{|u^2 - z|}{|a|} < 10^{-6}\,, \qquad \text{c'est-à-dire si} \qquad \frac{|u^2 - z|^2}{|z|^2} < 10^{-12}\,. \tag{4}$$

8. En fait, la suite précédente ne converge pas dans le cas particulier où a est un réel négatif. On pourra dans ce cas choisir $u_0 = a + i\epsilon$. Modifier le programme en conséquence.

```
1   #include<stdlib.h>
2   #include<iostream>
3   #include<iomanip>
4   #include<cmath>
5
6   using namespace std;
7
8   #define ASSERT(test) if (!(test)) cout << "Test failed in file " << __FILE__ \
9                                          << " line " << __LINE__ << ": " #test << endl
10
11  const double epsilon = 1e-6;
12
13  /** teste si deux nombres sont egaux à epsilon près
14  * @param[in] x et y deux nombres réels
15  * @return le booléan correspondant au test
16  **/
17  bool presqueEgal(double x, double y) {
18      if ((fabs(x-y) <= epsilon*fabs(x)) and (fabs(x-y) <= epsilon*fabs(y))) {
19          return true;
20      }
21      return false;
22  }
23
24
25
26  /** saisie d'un nombre complexe
27  * @param[out] r : la partie réelle
28  * @param[out] i : la partie imaginaire
29  **/
30  void saisie(double &r, double &i) {
31      cout << "Partie réelle : " << endl;
32      cin >> r;
33      cout << "Partie imaginaire : " << endl;
34      cin >> i;
35  }
36
37  /** affichage d'un nombre complexe
38  * @param[in] r : la partie réelle
39  * @param[in] i : la partie imaginaire
40  **/
41  void affiche(double r, double i) {
42      if (i >= 0)
```

```
43          cout << r << "+" << i << "i" << endl;
44      else
45          cout << r << "-" << -i << "i" << endl;
46  }
47
48  /** somme de deux nombres complexes
49  * @param[in] r1 : la partie réelle de z1
50  * @param[in] i1 : la partie imaginaire de z1
51  * @param[in] r2 : la partie réelle de z2
52  * @param[in] i2 : la partie imaginaire de z2
53  * @param[out] sommer : la partie réelle de z1 + z2
54  * @param[out] sommei : la partie imaginaire de z1 + z2
55  **/
56  void somme(double r1, double i1,
57             double r2, double i2,
58             double &sommer, double &sommei) {
59      sommer = r1 + r2;
60      sommei = i1 + i2;
61  }
62
63  /** test de la fonction somme
64      affiche un message en cas de test raté
65  **/
66  void testSomme() {
67      double sommer;
68      double sommei;
69      somme(1.2367, 0, -1.2, 0, sommer, sommei);
70      ASSERT(presqueEgal(sommer, 0.0367));
71      ASSERT(presqueEgal(sommei, 0));
72      somme(0, 4.57, 0, 1.29, sommer, sommei);
73      ASSERT(presqueEgal(sommer, 0));
74      ASSERT(presqueEgal(sommei, 5.86));
75      somme(1.2367, -4.5, 2.35, -5.2, sommer, sommei);
76      ASSERT(presqueEgal(sommer, 3.5867));
77      ASSERT(presqueEgal(sommei, -9.7));
78  }
79
80  /** produit de deux nombres complexes
81  * @param[in] r1 : la partie réelle de z1
82  * @param[in] i1 : la partie imaginaire de z1
83  * @param[in] r2 : la partie réelle de z2
84  * @param[in] i2 : la partie imaginaire de z2
85  * @param[out] prodr : la partie réelle de z1 * z2
86  * @param[out] prodi : la partie imaginaire de z1 * z2
87  **/
88  void produit(double r1, double i1,
89               double r2, double i2,
90               double &prodr, double &prodi) {
91      prodr = r1*r2 - i1*i2;
92      prodi = i1*r2 + i2*r1;
93  }
94
95  /** test de la fonction produit
96      affiche un message en cas de test raté
97  **/
```

```
98  void testProduit() {
99      double prodr;
100     double prodi;
101     produit(1.2367, 0, -1.2, 0, prodr, prodi);
102     ASSERT(presqueEgal(prodr, -1.48404));
103     ASSERT(presqueEgal(prodi, 0));
104     produit(0, 4.57, 0, 1.29, prodr, prodi);
105     ASSERT(presqueEgal(prodr, -5.8953));
106     ASSERT(presqueEgal(prodi, 0));
107     produit(1.2367, -4.5, 2.35, -5.2, prodr, prodi);
108     ASSERT(presqueEgal(prodr, -20.493755));
109     ASSERT(presqueEgal(prodi, -17.00584));
110 }
111
112 double norme_carree(double r, double i) {
113     return r*r + i*i;
114 }
115
116 /** test de la fonction norme_carree
117     affiche un message en cas de test raté
118 **/
119 void testNormeCarree() {
120     ASSERT(presqueEgal(norme_carree(1.2367, 0), 1.52942689));
121     ASSERT(presqueEgal(norme_carree(0, -4.57), 20.8849));
122     ASSERT(presqueEgal(norme_carree(1.2367, -4.5), 21.77942689));
123 }
124
125
126 /** inverse d'un nombre complexe
127 * @param[in] r : la partie réelle de z
128 * @param[in] i : la partie imaginaire de z
129 * @param[out] invr : la partie réelle de 1/z
130 * @param[out] invi : la partie imaginaire de 1/z
131 **/
132 void inverse(double r, double i, double &invr, double &invi) {
133     double norme;
134     norme = norme_carree(r, i);
135     if (norme == 0) {
136         cout << "Inverse de 0";
137         exit(1);
138     }
139     invr = r/norme;
140     invi = -i/norme;
141 }
142
143 /** test de la fonction inverse
144     affiche un message en cas de teste raté
145 **/
146 void testInverse() {
147     double invr, invi;
148     inverse(1.2367, 0, invr, invi);
149     ASSERT(presqueEgal(invr, 0.8086035));
150     ASSERT(presqueEgal(invi, 0));
151     inverse(0, -4.57, invr, invi);
152     ASSERT(presqueEgal(invr, 0));
```

```
153     ASSERT(presqueEgal(invi, 0.2188183));
154     inverse(1.2367, -4.5, invr, invi);
155     ASSERT(presqueEgal(invr, 0.0567829));
156     ASSERT(presqueEgal(invi, 0.2066170));
157 }
158 
159 /** l'erreur relative de la fonction racine
160 * @param[in] ur : la partie réelle de u
161 * @param[in] ui : la partie imaginaire de u
162 * @param[in] ar : la partie réelle de a
163 * @param[in] ai : la partie imaginaire de a
164 * @return |u^2 - a|^2 / |a|^2
165 **/
166 double erreur(double ur, double ui, double ar, double ai) {
167     double u2r, u2i;
168     double sommer, sommei;
169     produit(ur, ui, ur, ui, u2r, u2i);
170     somme(u2r, u2i, -ar, -ai, sommer, sommei);
171     return norme_carree(sommer, sommei) / norme_carree(ar, ai);
172 }
173 
174 /** racine carré d'un nombre complexe
175 * @param[in] ar : la partie réelle de a
176 * @param[in] ai : la partie imaginaire de a
177 * @param[out] raciner : la partie réelle de racine(a)
178 * @param[out] racinei : la partie imaginaire de racine(a)
179 **/
180 void racine_carree(double ar, double ai, double &raciner, double &racinei) {
181     double unr, uni;
182     double invr, invi;
183     double prodr, prodi;
184     double sommer, sommei;
185     unr = ar;
186     uni = ai;
187     if (ai == 0 && ar < 0) uni += epsilon;
188     while (erreur(unr, uni, ar, ai) >= epsilon*epsilon) {
189         inverse(unr, uni, invr, invi);
190         produit(ar, ai, invr, invi, prodr, prodi);
191         somme(unr, uni, prodr, prodi, sommer, sommei);
192         unr = 1.0/2.0*sommer;
193         uni = 1.0/2.0*sommei;
194     }
195     raciner = unr;
196     racinei = uni;
197 }
198 
199 /** test de la fonction racineCarre
200     affiche un message en cas de teste raté
201 **/
202 void testRacineCarree() {
203     double raciner;
204     double racinei;
205     racine_carree(1.2367, 0, raciner, racinei);
206     ASSERT(presqueEgal(raciner, 1.1120701));
207     ASSERT(presqueEgal(racinei, 0));
```

```
208       racine_carree(0, -4.57, raciner, racinei);
209       ASSERT(presqueEgal(raciner, 1.5116216));
210       ASSERT(presqueEgal(racinei, -1.5116216));
211       racine_carree(1.2367, -4.5, raciner, racinei);
212       ASSERT(presqueEgal(raciner, 1.7180720));
213       ASSERT(presqueEgal(racinei, -1.3096074));
214       racine_carree(-2., 0, raciner, racinei);
215       ASSERT(presqueEgal(racinei, 1.4142135));
216   }
217
218
219   int main() {
220       double r;
221       double i;
222       saisie(r, i);
223       affiche(r, i);
224       testSomme();
225       testProduit();
226       testNormeCarree();
227       testInverse();
228       testRacineCarree();
229       return 0;
230   }
```