université PARIS-SACLAY

Correction TP de programmation n°3

Cours de programmation impérative

—Licence MPI L1 S2 - Info 121—

Fonctions et procédures

Cette séance de travaux pratiques est dédiée à l'écriture et l'utilisation de fonctions simples.
Voici quelques exemples de fonctions et procédures de la bibliothèque standard de `C++` :

prototype de la fonction	fichier	description
`int abs(int j)`	`cstdlib`	valeur absolue entière
`float fabs(float x)`	`cmath`	valeur absolue réelle
`float round(float x)`	`cmath`	arrondi à l'entier le plus proche
`float trunc(float x)`	`cmath`	arrondi à l'entier inférieur
`float pow(float x, float y)`	`cmath`	puissance réelle
`float sqrt(float x)`	`cmath`	racine carrée réelle
`float exp(float x)`	`cmath`	exponentielle réelle
`float log(float x)`	`cmath`	logarithme réel
`void exit(int e)`	`cstdlib`	quitte le programme

Pour utiliser une fonction, il faut inclure le fichier de déclaration correspondant (par exemple `#include <cmath>`).
Le `C++` ne fait pas la différence entre une fonction et une procédure : une procédure est juste une fonction qui ne retourne rien (c'est-à-dire `void`).
Voici comment on peut écrire la fonction valeur absolue :

```
float absolue(float x) {
   if (x >= 0.) return x;
   else         return -x;
}
```

▸ Exercice 1. Fonction factorielle et coefficients du binôme de Newton

La fonction pour calculer la factorielle d'un entier est donnée dans le fichier `binome.cpp`.

1. Pour tester la fonction `factoriel`, on utilise la fonction `testFactoriel`. Ajouter dans cette fonction quelques tests en dehors de la convention factoriel$(0) = 1$.
2. On appelle coefficient du binôme de Newton (ou coefficient binomial) $\binom{n}{p}$ le nombre de parties à p éléments dans un ensemble à n éléments. Par exemple :

$$\binom{0}{0} = 1, \quad \binom{3}{2} = 3, \quad \binom{4}{2} = 6$$

Le coefficient binômial $\binom{n}{p}$ peut être calculé par :

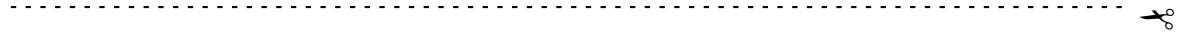
$$\binom{n}{p} = \frac{n!}{p!(n-p)!}$$

En utilisant la fonction factorielle écrite à la question précédente, compléter la fonction `binome` dans le fichier `binome.cpp`, et la tester par la fonction `testBinome`.

✂ ----------

```
1  #include<iostream>
2  #include<cstdlib>
3
4  using namespace std;
5
6  #define ASSERT(test) if (!(test)) cout << "Test failed in file " << __FILE__ \
7                                          << " line " << __LINE__ << ": " #test << endl
8
9  /** Calculer le factoriel d'un entier
10  *  @param n un entier
11  *  @return un entier
12  **/
13 int factoriel(int n) {
14     if (n < 0) {
15         cerr << "Factoriel argument negatif file " << __FILE__ << " line " << __LINE__ << endl;
16         exit(1);
17     }
18     int res = 1, i;
19     for (i = 1 ; i <= n ; i ++) {
20         res *= i;
21     }
22     return res;
23 }
24
25 /** Tester la fonction factorielle
26  **/
27 void testFactoriel() {
28     ASSERT(factoriel(0) == 1);
29     ASSERT(factoriel(1) == 1);
30     ASSERT(factoriel(4) == 24);
31     ASSERT(factoriel(10) == 3628800);
32 }
33
34 /** Calculer le binome de Newton
35  *  @param n entier, p entier, n >= p
36  *  @return un entier
37  **/
38 int binome(int n, int p) {
39     // Il n'y a pas de soucis à utiliser la division entière ici donc il n'y a pas besoin d'un cast
40     // vers un double.
41     return factoriel(n)/(factoriel(p) * factoriel(n - p));
42 }
43
44 /** Tester la fonction binome
45  **/
46 void testBinome() {
47     // valeur de base
48     ASSERT(binome(0, 0) == 1);
49     ASSERT(binome(3, 2) == 3);
50     ASSERT(binome(4, 2) == 6);
51     ASSERT(binome(5, 3) == 10);
52     ASSERT(binome(10, 5) == 252);
53 }
54
```

```
55  /** Fonction principale
56   **/
57  int main() {
58      testFactoriel();
59      testBinome();
60      cout << "OK" << endl;
61  }
```



Dans la suite, on demande d'écrire soi-même les fonctions dont on a besoin. De plus, on commentera et testera, dans la mesure du possible, toutes les fonctions à l'aide de la macro `ASSERT` ci-dessous :

```
#define ASSERT(test) if (!(test)) cout << "Test failed in file " << __FILE__ \
                                       << " line " << __LINE__ << ": " #test << endl
```

▸ **Exercice 2. Ecriture et test d'une fonction simple.** Vous allez maintenant faire la démarche complète d'ecrire une fonction vous-même puis de l'appeler. Le but est de maîtriser la syntaxe de base de définition et d'appel d'une fonction. Vous pouvez consulter votre cours ou bien repartir de l'exemple de la factorielle.

1. Rajoutez dans le même fichier le code d'une fonction `square` qui prend en paramètre un nombre entier et calcule son carré.
2. Rajoutez aussi le code d'une fonction `testSquare` qui teste la fonction `square` en l'appellant avec différentes valeurs (vous pouvez vous inspirer des fonctions `testFactoriel` et `testBinome`).
3. Rajoutez un test avec deux appels imbriqués, pour vérifier que $(3^2)^2 = 81$.
4. Modifiez votre programme `main` pour lancer les tests de cette nouvelle fonction.

✂ ----------

```
1  #include<iostream>
2  #include<stdlib.h>
3
4  using namespace std;
5
6  #define ASSERT(test) if (!(test)) cout << "Test failed in file " << __FILE__ \
7                                        << " line " << __LINE__ << ": " #test << endl
8
9  /** Calculer le carré d'un nombre
10  *  @param n réel
11  *  @return un réel
12  **/
13 int square(int n) {
14     return n * n;
15 }
16
17 /** Tester la fonction binome
18  **/
19 void testSquare() {
20     // valeur de base
21     ASSERT(square(0) == 0);
22     ASSERT(square(1) == 1);
23     ASSERT(square(2) == 4);
24     ASSERT(square(3) == 9);
25     ASSERT(square(10) == 100);
26     ASSERT(square(square(3))== 81);
27 }
28
29 /** Fonction principale
30  **/
```

```
31  int main() {
32      testSquare();
33      cout << "OK" << endl;
34  }
```

▸ **Exercice 3. (Racine carrée et n-ième)** Cet exercice est obligatoire, ceux qui ne l'ont pas fini en TP devront envoyer le corrigé par email au professeur du TP. On reprend l'exercice 2 du TP précédent. Sur les nombres à virgule (`float`), l'opérateur `==` n'est pas très utile à cause des erreurs d'arrondis. Pour résoudre ce problème, quand on veut comparer deux nombres à virgule, on teste si la valeur absolue de la différence est négligeable devant les deux nombres :

$$|x - y| \leq \epsilon|x| \qquad \text{et} \qquad |x - y| \leq \epsilon|y| \tag{1}$$

où ϵ est un très petit nombre.

1. Définir une constante `epsilon` égale à 10^{-6} (`1e-6` en `C++`) ;
2. Écrire une fonction `presqueEgal` qui prend deux nombres x et y et qui teste s'ils vérifient la condition ci-dessus, c'est à dire s'ils sont égaux avec une précision de ϵ. Remarque importante : quand on vous demande d'écrire une fonction qui teste quelque chose, cela signifie qu'on veut une fonction qui renvoie un booléen.
3. Écrire une fonction `testpresqueEgal` que vérifie par des `ASSERT` que `presqueEgal(1, 1+epsilon/2)`, `presqueEgal(1, 1)`, `presqueEgal(1+1, 2)`, `presqueEgal(0, 0)` retournent bien vrai et que `presqueEgal(1, 1+2*epsilon)`, `presqueEgal(0, 1)` retournent bien faux.

On montre en mathématique que étant donné un réel positif a la suite

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

converge vers $\sqrt{a}$.

4. Écrire une fonction qui prend en argument un réel a et calcule sa racine carrée en utilisant la suite définie ci-dessus. Par définition, la racine carrée est la solution positive x de l'équation $x^2 = a$. On utilisera ce test et la fonction `presqueEgal` définie plus haut pour arrêter le calcul au bon moment. Si vous en avez besoin, vous pouvez adapter la fonction `square` précédente pour qu'elle travaille aussi avec les réels (type `float`).
5. Tester cette fonction en vérifiant entre autre que $\sqrt{0} = 0$, $\sqrt{1} = 1$, $\sqrt{4} = 2$ et $\sqrt{2} \approx 1.4142135$
6. Écrire un programme qui demande un nombre positif à l'utilisateur et qui affiche sa racine carrée.

Pour calculer la racine n-ième d'un nombre, on procède de la même manière que pour la racine carrée en utilisant la suite

$$u_0 := a, \qquad u_{k+1} = \frac{1}{n}\left((n-1)u_k + \frac{a}{u_k^{n-1}}\right)$$

qui converge vers $\sqrt[n]{a}$ si $a > 0$.

7. Écrire une fonction qui calcule la racine n-ième d'un réel a. Par définition, la racine n-ième est la solution positive x de l'équation $x^n = a$. On utilisera ce test et la fonction `presqueEgal` définie plus haut pour arrêter le calcul au bon moment. On reprendra la fonction puissance du TP précédent.
8. Tester la fonction sachant que $\sqrt[5]{2} \approx 1.1486983$.

▸ Exercice 4. Exponentielle

La fonction exponentielle est définie par $\exp(a) := \sum_{i=0}^{\infty} a^i/i!$.

1. En réutilisant la fonction `presqueEgal` écrire une fonction `exponentielle` qui calcule l'exponentielle d'un nombre a. On utilisera une boucle et un accumulateur pour calculer les sommes $\sum_{i=0}^{N} a^i/i!$. On stoppe la boucle dès que deux sommes calculées consécutivement sont «presque égales».

Cette méthode n'est pas très efficace car, en utilisant les fonctions `factorielle` et `puissance`, on recalcule plusieurs fois les mêmes produits. Pour aller plus vite, on peut, dans la même boucle, accumuler la factorielle, la puissance et la somme.

2. Écrire une fonction `exponentielle2` qui fait le calcul plus rapidement en utilisant les trois accumulateurs dans la même boucle. On gardera la même condition d'arrêt de la boucle.


```
1   #include<iostream>
2   #include<stdlib.h>
3
4   using namespace std;
5
6   #define ASSERT(test) if (!(test)) cout << "Test failed in file " << __FILE__ \
7                                          << " line " << __LINE__ << ": " #test << endl
8
9   const float epsilon = 1e-6;
10
11  /** Calculer le factoriel d'un entier
12   *  @param n un entier
13   *  @return un entier
14   **/
15  int factoriel(int n) {
16      if (n < 0) {
17          cout << "Factoriel argument negatif file " << __FILE__ << " line " << __LINE__ << endl;
18          exit(1);
19      }
20      int res = 1, i;
21      for (i = 1 ; i <= n ; i ++) {
22          res *= i;
23      }
24      return res;
25  }
26
27  /** Puissance d'un nombre réel
28   *  @param n un entier positif
29   *  @param a un nombre réel
```

```
30   *  @return a^n
31   **/
32  float puissance(float a, unsigned int n) {
33     float res = 1.;
34     for (unsigned int i = 0; i<n; i++)
35       res = res*a;
36     return res;
37  }
38
39  /** Valeur absolue d'un float
40   *  @param x un float
41   *  @return |x|
42   **/
43  float absolue(float x) {
44    if (x < 0) return -x;
45    else       return x;
46  }
47
48  /** Égalité des float
49   *  @param x, y deux float
50   *  @return si x == y à epsilon près
51   **/
52  bool presqueEgal(float x, float y) {
53    return absolue(x-y) <= epsilon*absolue(x) and absolue(x-y) <= epsilon*absolue(y);
54  }
55
56  /** Exponentielle d'un float
57   *  @param a, un float
58   *  @return exp(a)
59   **/
60  float exponentielle(float a) {
61      int i = 0;
62      float res = 0.;
63      bool stop = false;
64      while (not stop) {
65          float tmp = res + (float)puissance(a, i) / (float)factoriel(i);
66          if (presqueEgal(res, tmp))
67              stop = true;
68          else
69              res = tmp;
70          i = i + 1;
71      }
72      return res;
73  }
74
75  void testExponentielle() {
76      ASSERT(presqueEgal(exponentielle(0.), 1.));
77      ASSERT(presqueEgal(exponentielle(1.), 2.7182818));
78      ASSERT(presqueEgal(exponentielle(2.), 7.3890561));
79  }
80
81  /** Exponentielle d'un float avec la méthode rapide
82   *  @param a, un float
83   *  @return exp(a)
84   **/
```

```
85  float exponentielle2(float a) {
86      int i = 0;
87      float acc_puiss = 1;
88      float acc_fact  = 1;
89      float res = 0.;
90      bool stop = false;
91      while (not stop) {
92          float tmp = res + (float)acc_puiss / (float)acc_fact;
93          if (presqueEgal(res, tmp))
94              stop = true;
95          else
96              res = tmp;
97          i = i + 1;
98          acc_puiss = acc_puiss * a;
99          acc_fact  = acc_fact  * i;
100     }
101     return res;
102 }
103
104 void testExponentielle2() {
105     ASSERT(presqueEgal(exponentielle2(0.), 1.));
106     ASSERT(presqueEgal(exponentielle2(1.), 2.7182818));
107     ASSERT(presqueEgal(exponentielle2(2.), 7.3890561));
108 }
109
110 /** Fonction principale
111  **/
112 int main() {
113     testExponentielle();
114     cout << "OK" << endl;
115 }
```

▸ Exercice 5. Logarithme

Pour calculer le logarithme d'un nombre positif a, on peut utiliser de la même manière que pour la racine le fait que la suite

$$u_0 := a, \qquad u_{n+1} := u_n + \frac{a}{\exp(u_n)} - 1$$

converge vers $\ln(a)$.

1. Écrire une fonction `logarithme` qui calcule le logarithme d'un nombre positif. Par définition, le logarithme de a est la solution x de l'équation $\exp(x) = a$. On utilisera ce test et la fonction `presqueEgal` définie plus haut pour vérifier que l'on a bien le résultat.
2. Écrire un programme qui vérifie que pour un nombre x, on a bien $\exp(\ln(x)) = x$.
3. Écrire un programme qui calcule $\sqrt{x}$ par la formule

$$\sqrt{x} = x^{\frac{1}{2}} = \exp\left(\frac{\ln(x)}{2}\right).$$

Vérifier que l'on obtient bien le même résultat qu'avec la fonction de l'exercice précédent.

4. Même question pour $\sqrt[n]{x}$, avec la formule

$$\sqrt[n]{x} = x^{\frac{1}{n}} = \exp\left(\frac{\ln(x)}{n}\right) .$$

✂ ----------

```
1   #include <iostream>
2   #include <iomanip>
3   #include <cstdlib>
4
5   #define ASSERT(test) if (!(test)) cout << "Test failed in file " << __FILE__ \
6                                          << " line " << __LINE__ << ": " #test << endl
7
8   using namespace std;
9
10  const float epsilon = 1e-6;
11
12  /** Valeur absolue d'un float
13   *  @param x un float
14   *  @return |x|
15   **/
16  float absolue(float x) {
17      if (x < 0) return -x;
18      else       return x;
19  }
20
21  /** Égalité des float
22   *  @param x, y deux float
23   *  @return si x == y à epsilon près
24   **/
25  bool presqueEgal(float x, float y) {
26      return absolue(x-y) <= epsilon*absolue(x) and absolue(x-y) <= epsilon*absolue(y);
27  }
28
29  /** Exponentielle d'un float avec la méthode rapide
30   *  @param a, un float
31   *  @return exp(a)
32   **/
33  float exponentielle(float a) {
34      int i = 0;
35      float acc_puiss = 1;
36      float acc_fact  = 1;
37      float res = 0., res_old;
38      do {
39        res_old = res;
40        res += acc_puiss / acc_fact;
41        i = i + 1;
42        acc_puiss = acc_puiss * a;
43        acc_fact  = acc_fact  * i;
44      } while (not presqueEgal(res, res_old));
45      return res;
46  }
47
48  /** logarithme d'un float
```

```
49   *  @param a un float
50   *  @return le logarithme de a
51   **/
52  float logarithme(float a) {
53      float un;
54      if (a <= 0) {
55          cerr << "Racine carré d'un nombre negatif ou nul !" << endl;
56          exit(1);
57      };
58      un = 1;
59      while (not presqueEgal(exponentielle(un), a)) {
60          un = un + a / exponentielle(un) - 1.;
61          cout << un << endl;
62      }
63      return un;
64  }
65
66  void testLogarithme() {
67      ASSERT( presqueEgal(exponentielle(logarithme(1)), 1) );
68      ASSERT( presqueEgal(exponentielle(logarithme(2)), 2) );
69      ASSERT( presqueEgal(exponentielle(logarithme(3)), 3) );
70      ASSERT( presqueEgal(exponentielle(logarithme(4)), 4) );
71      ASSERT( presqueEgal(exponentielle(logarithme(100)), 100) );
72  }
73
74  int main(void) {
75      testLogarithme();
76      return 0;
77  }
```