Chapitre 6. Surcharge des Opérateurs

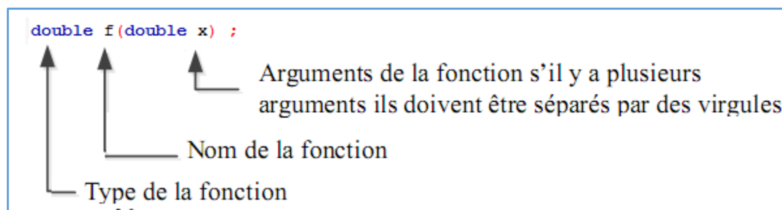
1. Fonctions

La notion de prototype de fonction est un apport du langage C++ qui, comme la notion de constante, a migrée vers le langage C version ANSI. Elle permet de déclarer une fonction à l'intérieur d'une autre fonction (mais pas de la définir, un programme étant toujours une suite de définitions de fonctions).

Une fonction peut :

- Se trouver dans le même module de texte, être incluse automatiquement dans le texte du programme (par une directive #include) ou compilée séparément.
- Etre appelée à partir du programme principal, d'une autre fonction ou de la fonction elle-même (récursivité)
- Admettre ou non des arguments.
- Retourner ou non une valeur.
- Posséder ses propres variables. (variables locales)
- Une fonction à un seul point d'entrée (la première instruction de la fonction)
- Peut avoir plusieurs points de sorties par l'instruction return, ou automatiquement après la dernière instruction de la fonction.
- Ce ne fait aucune distinction entre procédure et fonction mais une procédure n'est rien d'autre qu'une fonction pour laquelle on ne se préoccupe pas de la valeur de retour sans type (void).
- C++ ne permet pas l'imbrication des fonctions, une fonction ne peut être déclarée à l'intérieur d'une autre fonction, mais une fonction peut appeler une autre fonction.
- Les arguments d'une fonction ne peuvent être passés que par valeur.

Exemple :



```
//déclaration d'un local
//==================================================

#include <iostream>
#include <math.h>

using namespace std;

int  main(void)
{
double f(double x) ;
float x, y ;
cout << "x = " ;
cin >> x ;
while (x != 1000)
{
y = f(x) ;
cout << "f(" << x << ") = " << y << "\n" ;
cout << "x = " ;
cin >> x ;
}
return 0;
}
double f(double x)
{
return((sin(x) + log(x))/(exp(x) + 2)) ;
}
```

1.1.Quelques règles sur les fonctions

- Les arguments figurent dans l'entête de la fonction se nomment arguments muets ou formels.
- Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment les arguments effectifs.
- L'instruction return peut mentionner n'importe quelle expression
- L'instruction return peut apparaître à plusieurs reprises dans une fonction. Elle peut apparaître à plusieurs reprises dans une fonction.

```
double  absom(double s)
{
  if (s<0)  return s;
  else return –s;
}
```

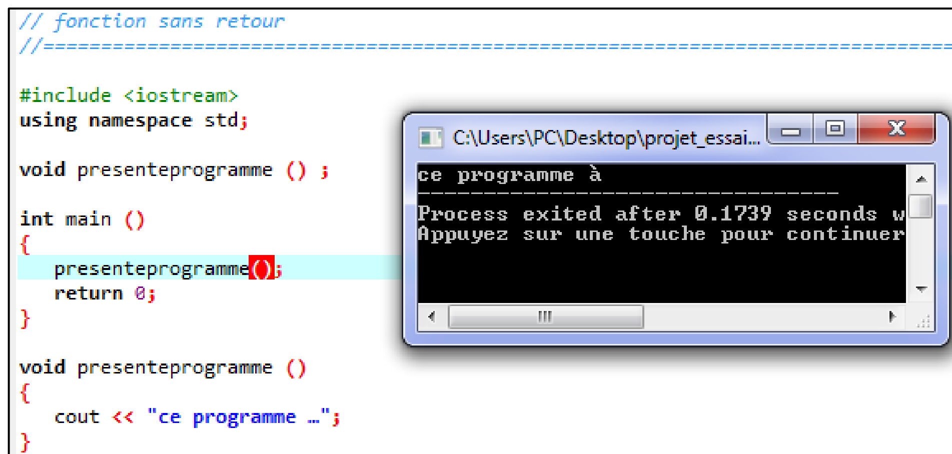
Quand une fonction ne renvoie pas de résultat, on le précise dans l'entête et dans la déclaration, à l'aide du mot clé void.

```
void affiche(int n) ;
```

1.2.Fonctions sans retour

C++ permet de créer des fonctions qui ne renvoient aucun résultat. Mais, quand on la déclare, il faut quand même indiquer un type. On utilise le type void. Cela veut tout dire : il n'y a vraiment rien qui soit renvoyé par la fonction.

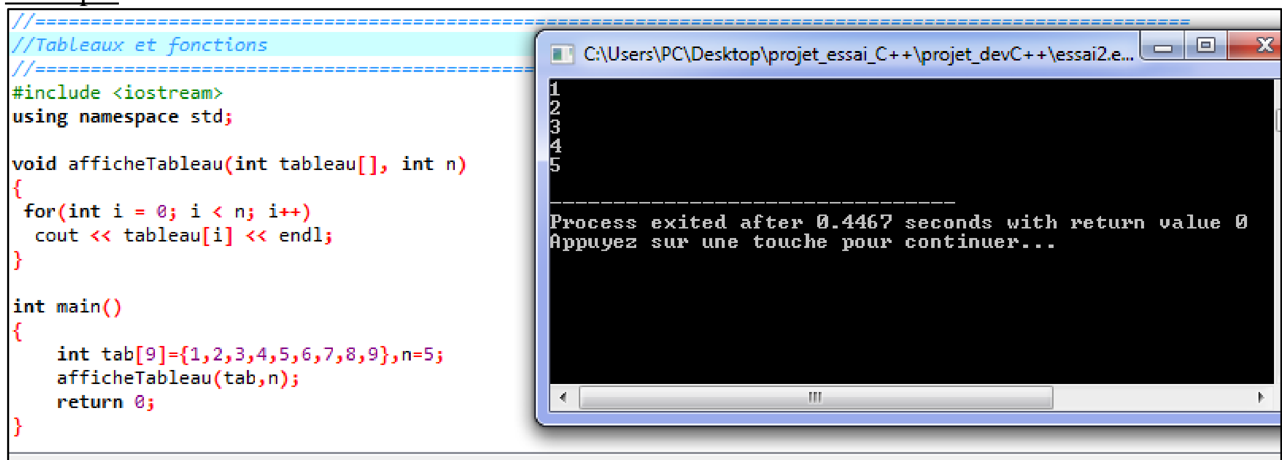
Exemple



1.3.Tableaux et fonctions

On peut passer un tableau comme argument d'une fonction. Voici donc une fonction qui reçoit un tableau en argument :

Exemple



2. Opérateurs et expressions

En combinant des variables, des opérateurs et de fonctions, on obtient des expressions.

2.1.Opérateurs arithmétiques

+ : addition,

- : soustraction,

* : multiplication,

/ : division (19.0 / 5.0 vaut 3.8 et 19 / 5 vaut 3)

% : reste de la division entière de deux entiers (modulo)(19 % 5 vaut 4)

++ : incrémentation. Les expressions i+ + et ++i ont pour valeur :

– pour la première, d’ajouter ensuite 1 à la valeur de i (post-incrémentation),

– pour la seconde, d’ajouter d’abord 1 à la valeur de i (pré-incrémentation).

-- : décrémentation. i-- et --i fonctionnent comme i++ et ++i, mais retranchent 1 à la valeur de i au lieu d’ajouter 1.

Exemple

```
 #include <iostream>
#include <math.h>
using namespace std;
int main()
{
    int a, b=3, c, d=3; a=++b; // équivalent à b++; puis a=b; => a=b=4
 c=d++; // équivalent à c=d; puis d++; => c=3 et d=4
 cout<<"a="<<a<<endl;
 cout<<"c="<<c;
}
```

Dans les expressions arithmétiques, les règles de priorité sont les règles usuelles mathématiques. Par exemple, l'expression 5 + 3 * 2 a pour valeur 11 et non 16. En cas de doute, il faut mettre des parenthèses.

Il arrive très souvent de calculer la nouvelle valeur d'une variable en fonction de son ancienne valeur. C++ fournit pour cela un jeu d'opérateurs combinés, de la forme :
+=, -=, *=, /=, %=
Forme générale : variable opérateur= expression
L'expression est équivalente `a :
variable = variable opérateur expression
Par exemple, l'expression i += 3 équivaut `a i = i + 3.

2.2.Opérateurs logiques

! : non,
&& : et,
|| : ou.

2.3.Opérateurs de comparaison

== : égal,
!= : différent,
< : strictement inférieur,
> : strictement supérieur,
<= : inférieur ou égal,
>= : supérieur ou égal.

3. Surcharge des fonctions

Plusieurs fonctions peuvent porter le même nom si leurs signatures diffèrent. La signature d'une fonction correspond aux caractéristiques de ses paramètres

- Leur nombre
- Le type respectif de chacun d'eux

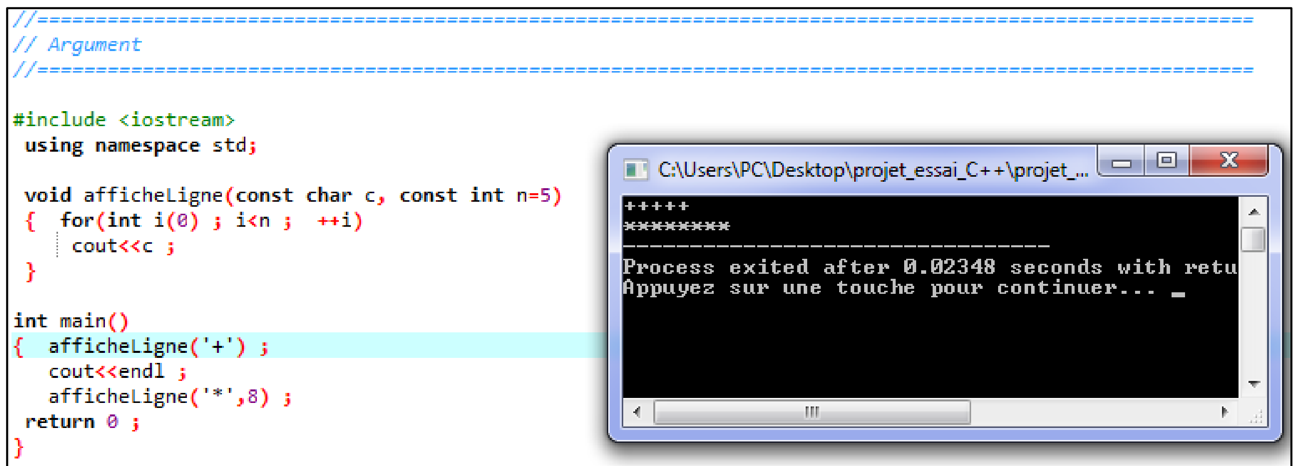
Le compilateur choisira la fonction à utiliser selon les paramètres effectifs par rapport aux paramètres formels des fonctions candidates.

```
//==================================================================
// surcharge des fonctions
//==================================================================
#include <iostream>
 using namespace std;

 // définition de la fonction somme qui calcul la somme de deux réels

 double somme(double a, double b)
 {  double r;
     r = a + b;
    return r;
 }

 // définition de la fonction somme qui calcul la somme de deux entiers

double somme(int a, int b)
 {  double r;
     r = a + b;
     return r;
 }

 // définition de la fonction somme qui calcul la somme de trois réels

 double somme(double a, double b, double c)
 {  double r;
```

3.1.Arguments par défaut

On peut, lors de la déclaration d'une fonction, donner des valeurs par défaut à certains paramètres des fonctions. Ainsi, lorsqu'on appelle une fonction, on ne sera pas obligé d'indiquer à chaque fois tous les paramètres.

Exemple



Remarques

- Seul le prototype doit contenir les valeurs par défaut.
- Les valeurs par défaut doivent se trouver à la fin de la liste des paramètres.
- Vous pouvez rendre tous les paramètres de votre fonction facultatifs.

4. Fonctions amies

Pour accéder aux données privées d'une classe A, il est nécessaire de publier les données membres des classes et cela entraîne la perte de sa protection. La deuxième solution est d'ajouter des fonctions d'accès aux membres privés ce qui augmente le temps d'exécution.

Une autre solution d'introduire les fonctions amies, la notion de fonction amie propose une solution intéressante, sous la forme d'un compromis entre encapsulation formelle des données privées et des données publiques. L'avantage de cette méthode est de permettre le contrôle des accès au niveau de la classe concernée, on ne peut pas s'imposer comme fonction amie d'une classe si cela n'a pas été prévu dans la classe.

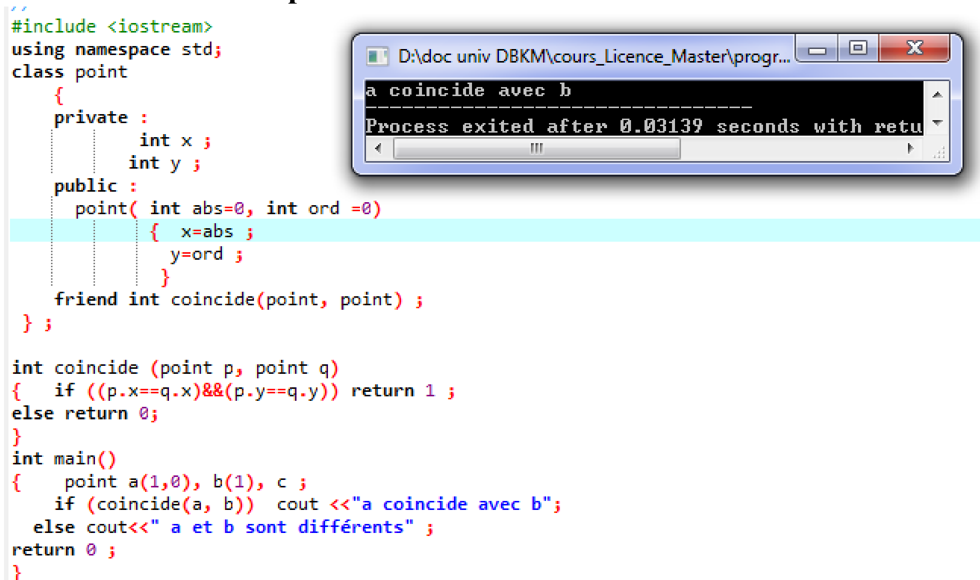
Il existe plusieurs situations d'amies :

• Fonctions indépendante amie d'une classe

• Fonction membre d'une classe, amie d'une autre classe

• Fonctions amie de plusieurs classes.

4.1. Fonction indépendante amie d'une classe



4.2. Fonction membre d'une classe amie d'une autre classe

On a deux classes A et B, et **int f(char, A)** fonction membre de B. pour que f puisse accéder aux membres privés de A, elle doit déclarer ami au sein de la classe A :

```
friend int B :: f(char, A) ;
```

Exemple :

```
class A
{  private :
    // partie privée
   public :
    // partie publique
   friend int B :: f (char, A) ;
   ...
   ...
} ;
class B
{  private :
    // partie privée
   public :
    // partie publique
    int f (char, A) ;
    ...
    ...
} ;
int B :: f (char, A) ;
{    // on a ici accès aux membres privés de tout objet de type A
} ;
```

4.3.Fonction amie de plusieurs classes

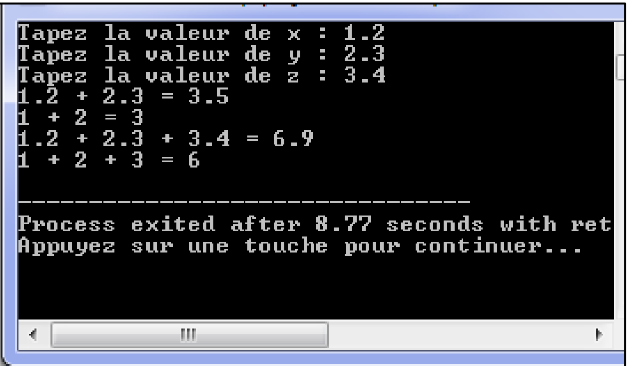
Toutes les fonctions d'une classe sont amies d'une autre classe. Toutes les fonctions membres de la classe B sont amies de la classe A, on placera, dans la classe A, la déclaration :

```
friend class B ; // dans la classe A
```

Exemple :

```
class A
{  // partie privée
    ......
   // partie publique
   friend class B;
   ......
};
class B
{     ......
     //on a accès totale aux membres privés de tout
    //objet de type A
      ......
};
```

```
    r = a + b + c;
    return r;
 }


 int main()
{  double x, y,z, resultat;
   cout << "Tapez la valeur de x : "; cin >> x;
   cout << "Tapez la valeur de y : "; cin >> y;
   cout << "Tapez la valeur de z : "; cin >> z;

//appel de notre fonction somme (double,double)
  resultat = somme(x, y);
  cout <<  x << " + " << y << " = " << resultat << endl;

//appel de notre fonction somme (int,int)
  resultat = somme(static_cast<int>(x), static_cast<int> ( y));
  cout <<   static_cast<int>(x) << " + " << static_cast<int>( y) << " = " << resultat << endl;

//appel de notre fonction somme (double, double, double)
  resultat = somme(x, y,z);
 cout<<  x << " + " << y << " + " << z<< " = " << resultat << endl;

//appel de notre fonction somme (double, double, double)
  resultat = somme(static_cast<int>(x), static_cast<int>( y), static_cast<int>(z));
 cout<< static_cast<int> ( x) << " + " << static_cast<int>(y) << " + " << static_cast<int>( z)<< " = " <<
resultat << endl;
 return 0;
}
```



```
Tapez la valeur de x : 1.2
Tapez la valeur de y : 2.3
Tapez la valeur de z : 3.4
1.2 + 2.3 = 3.5
1 + 2 = 3
1.2 + 2.3 + 3.4 = 6.9
1 + 2 + 3 = 6

--------------------------------
Process exited after 8.77 seconds with ret
Appuyez sur une touche pour continuer...
```