Chapitre VII: Patrons et amies " Fonctions et Classes "

VII.1 Les patrons « Template »

Parmi les techniques pour améliorer la réutilisabilité des morceaux de code, nous trouvons la notion de généricité. Cette notion permet d'écrire du code générique en paramétrant des fonctions et des classes par un type de données. Un module générique n'est alors pas directement utilisable : c'est plutôt un modèle, patron (***template***) de module qui sera «instancié » par les types de paramètres qu'il accepte. Dans la suite nous allons montrer comment C++ permet, grâce à la notion de ***patron de fonctions***, de définir une ***famille de fonctions paramétrées*** par un ou plusieurs types, et éventuellement des expressions. D'une manière comparable, C++ permet de définir des "***patrons de classes***". Là encore, il suffira d'écrire une seule fois la définition de la classe pour que le compilateur puisse automatiquement l'adapter à différents types.

VII.1.1 Patrons de fonctions

Pour illustrer les ***patrons de fonctions***, prenons un exemple concret : une fonction min qui accepte deux paramètres et qui renvoie la plus petite des deux valeurs qui lui est fournie. On désire bénéficier de cette fonction pour certains types simples disponibles en C++ (int, char, float). Les notions que nous avons vu en C++ jusqu'à maintenant ne nous permettent de résoudre ce problème qu'avec une seule solution. Cette solution est d'utiliser la surcharge et de définir trois fonctions min, une pour chacun des types considérés.

Exemple :

```
int min (int a, int b)
{ if ( a < b)
   return a ; // return ((a < b)? a : b);
  else
   return b ;
}

float min (float a, float b)
{ if ( a < b)
    return a ;
  else
     return b ;
}

char min (char a, char b)
{  if ( a < b)
    return a ;
  else
    return b ;
}
```

✍ Définition des fonctions min grâce à la surcharge: lors d'un appel à la fonction min, le type des paramètres est alors considéré et l'implantation correspondante est finalement appelée. Ceci présente cependant quelques inconvénients :

- La définition des 3 fonctions (perte de temps, source d'erreur) mène à des instructions identiques, qui ne sont différenciées que par le type des variables qu'elles manipulent.
- Si on souhaite étendre la définition de cette fonction à de nouveaux types, il faut définir une nouvelle implantation de la fonction min par type considéré.

✍ Une autre solution est de définir une fonction ***template***, c'est-à-dire générique. Cette définition définit en fait un patron de fonction, qui est instancié par un type de données (ici le type T) pour produire une fonction par type manipulé.

Exemple :

```
template <class T> // T est le paramètre de modèle
T min (T a, T b)
{  if ( a < b)
     return a
   else
     return b;
}
void main(){
int a = min(1, 7);  // int min(int, int)
float b = min(10.0, 25.0);    // float min(float, float)
char c = min('z', 'c');    // char min(char, char)
}
```

✍ Définition de la fonction min générique : il n'est donc plus nécessaire de définir une implantation par type de données. On définit donc bien plus qu'une fonction, on définit une méthode permettant d'obtenir une certaine abstraction en s'affranchissant des problèmes de type.

Remarques :

- Il est possible de définir des ***fonctions template*** acceptant plusieurs types de données en paramètre. Chaque paramètre désignant une classe est alors précédé du mot-clé ***class***, comme dans l'exemple : ***template <class** T, **class** U**> ....***
- Chaque type de données paramètre d'une ***fonction template*** doit être utilisé dans la définition de cette fonction.
- Pour que cette fonctionnalité soit disponible, les fonctions génériques doivent être définies au début du programme ou dans des fichiers d'interface (fichiers .h).

VII.1.2 Classe template : patron de classes

Il est possible, comme pour les fonctions, de définir des ***classes template***, c'est-à-dire paramétrées par un type de données. Cette technique évite ainsi de définir plusieurs classes similaires pour décrire un même concept appliqué à plusieurs types de données différents. Elle est largement utilisée pour définir tous les types de containers (comme les listes, les tables, les piles, etc.), mais aussi des algorithmes génériques par exemple.

La syntaxe permettant de définir une ***classe template*** est similaire à celle qui permet de définir des ***fonctions template***.

Exemple :

```
int x, y ;
public :
point (int abs=0, int ord=0) ;
void affiche () ;
// .....
} ;
```

✍ Lorsque nous procédons ainsi, nous imposons que les coordonnées d'un point soient de valeurs de type int. Si nous souhaitons disposer de points à coordonnées d'un autre type (float, double, long ...), nous devons définir une autre classe en remplaçant simplement, dans la classe précédente, le mot clé int par le nom de type voulu.

Ici encore, nous pouvons simplifier considérablement les choses en définissant un seul patron de classe de cette façon:

```
template <class T> class point {
T x , y ;
public :
point (T abs=0, T ord=0) ;
void affiche () ;
} ;
```

✍ Comme dans le cas des patrons de fonctions, la mention ***template <class T>*** précise que l'on a affaire à un patron (template) dans lequel apparaît un paramètre de type nommé T ;
La définition de notre patron de classes n'est pas encore complète puisqu'il y manque la définition des fonctions membres, à savoir le constructeur point et la fonction affiche().

Pour ce faire, la démarche va légèrement différer selon que la fonction concernée est en ligne ou non. Voici par exemple comment pourrait être défini notre constructeur en ligne:

```
point (T abs=0, T ord=0)
{ x = abs ; y = ord ; }
```

En revanche, lorsque la fonction est définie en dehors de la définition de la classe, il est nécessaire de rappeler au compilateur :

- que, dans la définition de cette fonction, vont apparaître des paramètres de type ; pour ce faire, on fournira à nouveau la liste de paramètre sous la forme: ***template <class T>***
- le nom du patron concerné. Par exemple, si nous définissons ainsi la fonction affiche, son nom sera: ***point<T>::affiche ()***

En définitive, voici comment se présenterait l'en-tête de la fonction affiche si nous le définissions ainsi en dehors de la classe :

```
template <class T>
void point<T>::affiche ()
```

En toute rigueur, le rappel du paramètre T à la suite du nom de patron (point) est redondant puisqu'il a déjà été spécifié dans la liste de paramètres suivant le mot clé template.

Voici ce que pourrait être finalement la définition de notre patron de classe point :

```
template <class T> class point    // Création d'un patron de classe
{  T x , y ;
  public :
point (T abs=0, T ord=0)
{ x = abs ; y = ord ; }
void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{ cout << "Paire : " << x << " " << y << "\n" ; }
```

VII.1.3 Utilisation d'un patron de classes

Comme pour les patrons de fonctions, l'instanciation de tels patrons est effectuée automatiquement par le compilateur selon les déclarations rencontrées. Après avoir créé ce patron, une déclaration telle que:

point <int> *a;*

conduit le compilateur à instancier la définition d'une classe point dans laquelle le paramètre T prend la valeur ***int***. Autrement dit, tout se passe comme si nous avions fourni une définition complète de cette classe. Si nous déclarons :

point <float> ad ;

le compilateur instancie la définition d'une classe point dans laquelle le paramètre T prend la valeur **float**, exactement comme si nous avions fourni une autre définition complète de cette classe.
Si nous avons besoin de fournir des arguments au constructeur, nous procéderons de façon classique comme dans :

point <int> a (3, 5) ;
point <float> ad (1.5, 9.6) ;

Si nous faisons abstraction de la signification des paramètres (coordonnées d'un point) et nous les déclarons comme des caractères ou chaînes de caractères, le compilateur ne signalera pas d'incohérence et la fonction *affiche()* remplacera *x* et *y* par les arguments, comme le montre l'exemple récapitulatif.

Exemple récapitulatif :

Voici un programme complet comportant :

- la création d'un patron de classes point dotée d'un constructeur en ligne et d'une fonction membre (affiche) non en ligne,
- la création d'un patron de fonctions min (en ligne dans la patron de classes) qui retourne le minimum des arguments,
- un exemple d'utilisation (main).

```
#include <iostream>
#include <string>
using namespace std ;

template <class T> class point   // Création d'un patron de classe
{  T x ;
   T y ;
public :
point (T abs=0, T ord=0)
{ x = abs ; y = ord ; }
void affiche () ;

T min ()
{ if ( x < y)
    return x;
  else
    return y;
}
};

template <class T> void point<T>::affiche ()
{ cout << "Paire : " << x << " " << y << "\n" ;  }

void main ()
{  point <int> ai (3, 5) ; // T prend la valeur int pour la classe point
    ai.affiche() ;
    cout << " Min : ... " << ai.min() << endl;
    point <char> ac ('z', 't') ; ac.affiche() ;
    cout << " Min : ... " << ac.min() << endl;
    point <double> ad (1.5, 9.6) ; ad.affiche() ;
    cout << " Min : ... " << ad.min() << endl;
    point <string> as ("Salut", " A vous") ; as.affiche() ;
    cout << " Min : ... " << as.min() << endl;
}
```

Résultat de l'exécution:

```
Paire : 3 5
 Min : ... 3
Paire : z t
 Min : ... t
Paire : 1.5 9.6
 Min : ... 1.5
Paire : Salut  A vous
 Min : ...  A vous

Process returned 0 (0x0)   execution time : 3.488 s
Press any key to continue.
```

Remarque :

Il faut noter aussi que le nombre de paramètres n'est pas limité à 1, on peut en avoir plusieurs :

```
template < class A, class B, class C, ... >
class MaClasse
{
// ...
public:
// ...
};
```

L'exemple suivant montre comment implémenter les fonctions de la classe template.

```
template <class T, class T1, class T2> class Etudiant
{  T Nom, Prenom;
   T1 Id;
   T2 Age;
public:
//...
T getNom()
 {return nom;}
T1 getId()
 {return Id;}
T2 getAge();
void Saisie();
void affiche();
//...
};

template <class T, class T1, class T2> T2 Etudiant<T,T1,T2>::getAge()
{ return Age; }

template <class T, class T1, class T2> void Etudiant<T,T1,T2>::affiche ()
{  cout << "Data : " << Nom << " " << Id << " " << Age <<"\n" ; }
```

VII.2 Fonctions amies

VII.2.1 Problème

Comment faire pour qu'une ***fonction f()*** et/ou une ***classe B*** puisse accéder aux données membres privées d'une ***classe A*** ?

Exemple :

```
class ratio
{  private:
     int num, den ;
   public:
    ratio(int n, int d);
    void affiche();
    float val_reel();
}
ratio somme (ratio r1, ratio r2);
```

Solution:[2]

- **Rendre publiques les données membres des classes.**
  Inconvénient: on perd leurs protections.
- **Ajout de fonctions d'accès aux membres privés.**
  Inconvénient : temps d'exécution pénalisant.
- **Fonctions amies :** Il est possible de déclarer qu'une ou plusieurs fonctions (extérieurs à la classe), sont des « amies » ; une telle déclaration d'amitié les autorise alors à accéder aux données privées au même titre que n'importe quelle fonction membre. L'avantage de cette méthode est de permettre le contrôle des accès au niveau de la classe concernée.

Exemple de déclaration d'une fonction amie:

```
class A
{   private :
     int i ;
     friend class B ;
     friend void f();
}
class B
{
   //tout ce qui appartient à B,
  //peut se servir des données membres privés de A
 // comme si elle était de A.
...
}
void f(A& a)
{  a.i = 10 ;  }
```

VII.2.2 Situation d'amitiés

- Fonction indépendante amie d'une classe.
- Fonction membre d'une classe amie d'une autre classe.
- Fonction amie de plusieurs classes
- Toutes les fonctions membres d'une classe, amies d'une autre classe.

VII.2.2.1 Exemple de fonction indépendante amie d'une classe [2]

```
class point
{  private :
     int x,y ;
   public :
     point (int abs= 0, int ord = 0)
     { x= abs ;
       y= ord ;
     } ;
     friend int coincide (point p, point q) ;  // déclaration d'une fonction amie indépendante
} ;
int coincide (point p, point q)
{  if ((p.x == q.x) && (p.y == q.y) )
```

```
    return (1);
  else
    return (0);
} ;
void main()
{  point a(1,0), b (1), c ;
  if ( coincide (a,b) )
    cout<< "A coincide avec B"<<endl;
  else
     cout<<"A et B sont différents"<<endl;
} ;
```

Résultat de l'exécution :

```
A coincide avec B

Process returned 0 (0x0)   execution time : 2.785 s
Press any key to continue.
```

VII.2.2.2 Fonction membre d'une classe amie d'une autre classe

On considère deux classes A et B

int f(char, A) fonction membre de B

f doit pouvoir accéder aux membres privés de A, elle sera déclarée amie 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
} ;
```

VII.2.2.3 Fonction amie de plusieurs classes

Rien n'empêche qu'une même fonction (indépendante ou membre) fasse l'objet de déclaration d'amitié dans différentes classes.

✍ Toutes les fonctions d'une classe sont amies d'une autre classe

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

VII.2.4 Toutes les fonctions membres d'une classe, amie d'une autre classe

C'est une généralisation du cas précédent. On pourrait d'ailleurs effectuer autant de déclarations d'amitié qu'il n'y a de fonctions concernées. Mais il est plus simple d'effectuer une déclaration globale. Ainsi pour dire que toutes les fonctions membres de la classe B sont amies de la classe A, on placera, dans la classe A, la déclaration [2] :

```
friend class B ;
```

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
    ......
};
```