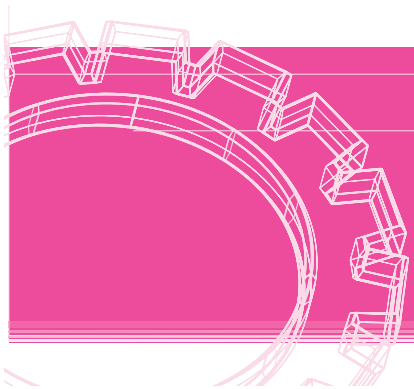
Chapitre 18

Les patrons de classes



Rappels

Introduite par la version 3, la notion de patron de classes permet de définir ce que l'on nomme aussi des « classes génériques ». Plus précisément, à l'aide d'une seule définition comportant des paramètres de type et des paramètres expression, on décrit toute une famille de classes ; le compilateur fabrique (instancie) la ou les classes nécessaires à la demande (on nomme souvent ces instances des « classes patron »).

Remarque

Cette fois, les paramètres expression étaient déjà prévus par la version 3 alors que, dans le cas des patrons de fonctions, ils n'ont été introduits que par la norme ANSI.

Définition d'un patron de classes

On précise les paramètres de type en les faisant précéder du mot clé `class` et les paramètres expression en mentionnant leur type dans une liste de paramètres introduite par le mot `template` (comme pour les patrons de fonctions, avec cette différence qu'ici, tous les paramètres – type ou expression – apparaissent).

Par exemple :

```
template <class T, class U, int n> class gene
{ // ici, T désigne un type quelconque, n une valeur entière quelconque
} ;
```

Si une fonction membre est définie (ce qui est le cas usuel) à l'extérieur de la définition du patron, il faut rappeler au compilateur la liste de paramètres (`template`) et préfixer l'en-tête de la fonction membre du nom du patron accompagné de ses paramètres. (En toute rigueur, il s'agit d'une redondance constatée, mais non justifiée, par le fondateur du langage lui-même, Stroustrup.) Par exemple, pour un constructeur de notre patron de classes précédent :

```
template <class T, class U, int n> gene <T, U, n>::gene (...)
{ ..... }
```

Instanciation d'une classe patron

On déclare une classe patron en fournissant à la suite du nom de patron un nombre de paramètres effectifs (noms de types ou expressions) correspondant aux paramètres figurant dans la liste (`template`). Les paramètres expression doivent obligatoirement être des expressions constantes du même type que celui figurant dans la liste. Par exemple, avec notre précédent patron (on suppose que `pt` est une classe) :

```
class gene <int, float, 5> c1 ;      // T = int, U = float,  n = 5
class gene <int, int, 12>  c2 ;      // T = int, U = int,    n = 12
const int NV=100 ;
class gene <pt, double, NV> c3 ;     // T = pt,  U = double, n=100
int n = 5 ;
class gene <int, double, n> c4 ;     // erreur : n n'est pas constant
const char C = 'e' ;
class gene <int, double, C> c5 ;     // erreur : C de type char et non int
```

Un paramètre de type effectif peut lui-même être une classe patron. Par exemple, si nous avons défini un patron de classes `point` par :

```
template <class T> class point { ..... } ;
```

Voici des instances possibles de `gene` :

```
class gene <point<int>, float, 10> c5 ;       // T=point<int>, U=float, n=10
class gene <point<char>, point<float>, 5> c6 ; // T=point<int>, U=point<float>, n=5
```

Un patron de classes peut comporter des membres (données ou fonctions) statiques ; dans ce cas, chaque instance de la classe dispose de son propre jeu de membres statiques.

Spécialisation d'un patron de classes

Un patron de classes ne peut pas être surdéfini (on ne peut pas définir deux patrons de même nom). En revanche, on peut spécialiser un patron de classes de différentes manières.

-- En spécialisant une fonction membre

Par exemple, avec ce patron :

```
template <class T, int n> class tableau { ..... } ;
```

On pourra écrire une version spécialisée de constructeur pour le cas où `T` est le type `point` et où `n` vaut `10` en procédant ainsi :

```
tableau <point, 10>:: tableau (...) { ..... }
```

-- En spécialisant une classe

Dans ce cas, on peut éventuellement spécialiser tout ou une partie des fonctions membre, mais ce n'est pas nécessaire. Par exemple, avec ce patron :

```
template <class T> class point { ..... } ;
```

on peut fournir une version spécialisée pour le cas où `T` est le type `char` en procédant ainsi :

```
class point <char>
{ // nouvelle définition de la classe point pour les caractères
} ;
```

Identité de classes patron

On ne peut affecter entre eux que deux objets de même type. Dans le cas d'objets d'un type classe `patron`, on considère qu'il y a identité de type lorsque leurs paramètres de types sont identiques et que les paramètres expression ont les mêmes valeurs.

Classes patron et héritage

On peut « combiner » de plusieurs façons l'héritage avec la notion de patron de classes :

- **Classe « ordinaire » dérivée d'une classe patron** ; par exemple, si `A` est une classe patron définie par `template <class T> A` :

  ```
  class B : public A <int>    // B dérive de la classe patron A<int>
  ```

  On obtient une seule classe nommée `B`.

- **Patron de classes dérivé d'une classe « ordinaire »** ; par exemple, si `A` est une classe ordinaire :

  ```
  template <class T> class B : public A
  ```

  On obtient une famille de classes (de paramètre de type `T`).

- **Patron de classes dérivé d'un patron de classes** ; par exemple, si `A` est une classe patron définie par `template <class T> A`, on peut :

  – définir une nouvelle famille de fonctions dérivées par :

  ```
  template <class T> class B : public A <T>
  ```

  Dans ce cas, il existe autant de classes dérivées possibles que de classes de base possibles.

  – définir une nouvelle famille de fonctions dérivées par :

  ```
  template <class T, class U> class B : public A <T>
  ```

  Dans ce cas, on peut dire que chaque classe de base possible peut engendrer une famille de classes dérivées (de paramètre de type `U`).

Exercice 127

Énoncé

Soit la définition suivante d'un patron de classes :

```
template <class T, int n> class essai
{  T tab [n] ;
  public :
   essai (T) ;    // constructeur
} ;
```

a. Donnez la définition du constructeur `essai`, en supposant :

- qu'elle est fournie « l'extérieur » de la définition précédente ;
- que le constructeur recopie la valeur reçue en argument dans chacun des éléments du tableau `tab` .

b. Disposant ainsi de la définition précédente du patron `essai`, de son constructeur et de ces déclarations :

```
const int n = 3 ;
int p = 5 ;
```

Quelles sont les instructions correctes et les classes instanciées ? On en fournira (dans chaque cas) une définition équivalente sous la forme d'une « classe ordinaire », c'est-à-dire dans laquelle la notion de paramètre a disparu.

```
essai <int, 10> ei (3) ;              // I
essai <float, n> ef (0.0) ;           // II
essai <double, p> ed (2.5) ;          // III
```

Solution

a. La définition du constructeur est analogue à celle que l'on aurait écrite « en ligne » ; il faut simplement « préfixer » son en-tête d'une liste de paramètres introduite par `template`. De plus, il faut préfixer l'en-tête de la fonction membre du nom du patron accompagné de ses paramètres (bien que cela soit redondant) :

```
template <class T, int n> essai<T,n>::essai(T a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

b. Appel I : correct.

```
class essai
{  int tab [10] ;
  public :
   essai (int) ;    // constructeur
} ;

essai::essai (int a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

b. Appel II : correct.

```
class essai
{  float tab [n] ;
  public :
   essai (float) ;    // constructeur
} ;

essai::essai (float a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

b. Appel III : incorrect car `p` n'est pas une expression constante.

Exercice 128

Énoncé

a. Créer un patron de classes nommé `pointcol`, tel que chaque classe instanciée permette de manipuler des points colorés (deux coordonnées et une couleur) pour lesquels on puisse « choisir » à la fois le type des coordonnées et celui de la couleur. On se limitera à deux fonctions membre : un constructeur possédant trois arguments (sans valeur par défaut) et une fonction `affiche` affichant les coordonnées et la couleur d'un « point coloré ».

b. Dans quelles conditions peut-on instancier une classe patron `pointcol` pour des paramètres de type classe ?

Solution

a. Voici ce que pourrait être la définition du patron demandé, en prévoyant les fonctions membre « en ligne » :

```
template <class T, class U> class pointcol
{
   T x, y ;    // coordonnees
   U coul ;    // couleur
  public :
   pointcol (T abs, T ord, U cl)
     { x = abs ; y = ord ; coul = cl ;
     }
   void affiche ()
     { cout << "point colore - coordonnees " << x << " " << y
            << " couleur " << coul << "\n" ;
     }
} ;
```

À titre indicatif, voici un exemple d'utilisation (on suppose que la définition précédente figure dans `pointcol.h`) :

```
#include "pointcol.h"
#include <iostream>
using namespace std ;

main()
{ pointcol <int, short int > p1 (5, 5, 2) ;  p1.affiche () ;
  pointcol <float, int> p2 (4, 6, 2) ; p2.affiche () ;
  pointcol <double, unsigned short> p3 (1, 5, 2) ; p3.affiche () ;
}
```

b. Il suffit que le type `classe` en question ait convenablement surdéfini l'opérateur <<, afin d'assurer convenablement l'affichage sur `cout` des informations correspondantes.

Exercice 129

Énoncé

On a défini le patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
     { cout << "Coordonnees : " << x << " " << y << "\n" ;
     }
```

a. Que se passe-t-il avec ces instructions :

```
point <char> p (60, 65) ;
p.affiche () ;
```

b. Comment faut-il modifier la définition de notre patron pour que les instructions précédentes affichent bien :

```
Coordonnees : 60 65
```

Solutions

a. On obtient l'affichage des caractères de code `60` et `65` (c'est-à-dire dans une implémentation utilisant le code ASCII : < et A) et non les nombres `60` et `65`.

b. Il faut spécialiser notre patron `point` pour le cas où le type `T` est le type `char`. Pour ce faire, on peut :

- soit fournir une définition complète de `point<char>`, avec ses fonctions membre ;
- soit, puisqu'ici seule la fonction `affiche` est concernée, se contenter de surdéfinir la fonction `point<char>::affiche`, ce qui conduit à cette nouvelle définition de notre patron :

```
    // définition generale du patron point
template <class T> class point
{
   T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{ cout << "Coordonnees : " << x << " " << y << "\n" ;
}
```

```
    // version specialisee de la fonction affiche pour le type char
void point<char>::affiche ()
{ cout << "Coordonnees : " << (int)x << " " << (int)y << "\n" ;
}
```

Exercice 130

Énoncé

Créer un patron de classes permettant de représenter des « vecteurs dynamiques » c'est-à-dire des vecteurs dont la dimension peut ne pas être connue lors de la compilation (ce n'est donc pas obligatoirement une expression constante comme dans le cas de tableaux usuels). On prévoira que les éléments de ces vecteurs puissent être de type quelconque.

On surdéfinira convenablement l'opérateur `[]` pour qu'il permette l'accès aux éléments du vecteur (aussi bien en consultation qu'en modification) et on s'arrangera pour qu'il n'existe aucun risque de « débordement d'indice ». En revanche, on ne cherchera pas à régler les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets du type concerné.

N.B. Il ne faut pas chercher à utiliser les composants standard introduits par la norme. En effet, le patron `vector` répondrait intégralement à la question.

Solution

En généralisant ce qui a été fait dans l'exercice 90 (sans toutefois initialiser les éléments du vecteur lors de sa construction), nous aboutissons au patron de classes suivant :

```
template <class T> class vect
{  int nelem ;    // nombre d'elements
   T * adr ;      // adresse zone dynamique contenant les elements
  public :
   vect (int) ;              // constructeur
   ~vect () ;                // destructeur
   T & operator [] (int) ;   // operateur d'acces a un element
} ;
template <class T> vect<T>::vect (int n)
{ adr = new T [nelem = n] ;
}
template <class T> vect<T>::~vect ()
{ delete adr ;
}
template <class T> T & vect<T>::operator [] (int i)
{ if ( (i<0) || (i>nelem) ) i = 0 ;     // protection indice hors limites
  return adr [i] ;
}
```

Remarque La définition du patron de classes serait plus simple si les fonctions membre étaient « en ligne ».

Notez que, ici encore, nous avons fait en sorte qu'une tentative d'accès à un élément situé en dehors du vecteur conduise à accéder à l'élément de rang `0`. Dans la pratique, on aura intérêt à utiliser des protections plus élaborées.

À titre indicatif, voici un petit programme, accompagné du résultat fourni par son exécution, utilisant ce patron (dont on suppose que la définition figure dans `vectgen.h`) :

```
#include "vectgen.h"
#include <iostream>
using namespace std ;
main()
{ vect<int> vi (10) ;
  vi[5] = 5 ; vi[2] = 2 ;
  cout << vi[2] << " " << vi[5] << "\n" ;
  vect<double> vd (3) ;
  vd[0] = 0.0 ; vd[1] = 0.1 ; vd[2] = 0.2 ;
  cout << vd[0] << " " << vd[1] << " " << vd[2] << "\n" ;
  cout << vd[12] << "\n" ;
  vd[12] = 1.2 ; cout << vd[12] << " " << vd[0] ;
}
```

```
 2 5
 0 0.1 0.2
 0
 1.2 1.2
```

Remarque Notre patron `vect` permet d'instancier des vecteurs dynamiques dans lesquels les éléments sont de type absolument quelconque, en particulier de type `classe` (pourvu que ladite classe dispose d'un constructeur sans argument). Il n'en serait pas allé ainsi si nous avions initialisé les éléments du tableau lors de leur construction par :

```
int i ;
for (i=0 ; i<nelem ; i++) adr[i] = 0 ;
```

En effet, dans ce cas, ces instructions auraient convenablement fonctionné pour n'importe quel type de base (par conversion de l'entier `0` dans le type voulu). En revanche, pour être applicable à des éléments de type `classe`, il aurait fallu, en outre, que la classe concernée dispose d'une conversion d'un `int` dans ce type `classe`, c'est-à-dire d'un constructeur à un argument de type numérique.

Exercice 131

Énoncé

Comme dans l'exercice précédent, réaliser un patron de classes permettant de manipuler des vecteurs dont les éléments sont de type quelconque mais pour lesquels la dimension, supposée être cette fois une expression constante, apparaîtra comme un paramètre (expression) du patron. Hormis cette différence, les « fonctionnalités » du patron resteront les mêmes.

Solution

Il n'est plus nécessaire d'allouer un emplacement dynamique pour notre vecteur qui peut donc figurer directement dans les membres donnée de notre patron. Le constructeur n'est plus nécessaire (voir toutefois la remarque ci-dessous), pas plus que le destructeur. Voici ce que pourrait être la définition de notre patron :

```
template <class T, int n> class vect
{  T v [n] ;      // vecteur de n elements de type T
  public :
   T & operator [] (int) ;   // operateur d'acces a un element
} ;
template <class T, int n> T & vect<T,n>::operator [] (int i)
{ if ( (i<0) || (i>n) ) i = 0 ;     // protection indice hors limites
  return v [i] ;
}
```

Voici, toujours à titre indicatif, ce que deviendrait le petit programme d'essai :

```
#include "vectgen1.h"
#include <iostream>
using namespace std ;
main()
{ vect<int, 10> vi  ;
  vi[5] = 5 ; vi[2] = 2 ;
  cout << vi[2] << " " << vi[5] << "\n" ;
  vect<double, 3> vd ;
  vd[0] = 0.0 ; vd[1] = 0.1 ; vd[2] = 0.2 ;
  cout << vd[0] << " " << vd[1] << " " << vd[2] << "\n" ;
  cout << vd[12] << "\n" ;
  vd[12] = 1.2 ; cout << vd[12] << " " << vd[0] ;
}
```

Remarque

Ici, nous n'avons pas eu besoin de faire du nombre d'éléments un membre donnée de nos classes patron : en effet, lorsqu'on en a besoin, on l'obtient comme étant la valeur du second paramètre fourni lors de l'instanciation. Si nous avions voulu conserver ce nombre d'éléments sous forme d'un membre donnée, il aurait été nécessaire de prévoir un constructeur, par exemple :

```
template <class T, int n> class vect
{  int nelem ;    // nombre d'elements
   T v [n] ;      // vecteur de n elements de type T
  public :
   vect () ;
   T & operator [] (int) ;   // operateur d'acces a un element
} ;
template <class T, int n> vect<T,n>::vect ()
{ nelem = n ;
}
```

Exercice 132

Énoncé

On dispose du patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche ()
     { cout << "Coordonnees : " << x << " " << y << "\n" ;
     }
} ;
```

a. Créer, par dérivation, un patron de classes `pointcol` permettant de manipuler des « points colorés » dans lesquels les coordonnées et la couleur sont de même type. On redéfinira convenablement les fonctions membre en réutilisant les fonctions membre de la classe de base.

b. Même question, mais en prévoyant que les coordonnées et la couleur puissent être de deux types différents.

c. Toujours par dérivation, créer cette fois une « classe ordinaire » (c'est-à-dire une classe qui ne soit plus un patron de classes, autrement dit qui ne dépende plus de paramètres...) dans laquelle les coordonnées sont de type `int`, tandis que la couleur est de type `short`.

Solutions

a. Aucun problème particulier ne se pose ; il suffit de faire dériver `pointcol<T>` de `point<T>`. Voici ce que peut être la définition de notre patron (ici, nous avons laissé le constructeur « en ligne » mais nous avons défini `affiche` en dehors de la classe) :

```
template <class T> class pointcol : public point<T>
{  T cl ;
  public :
   pointcol (T abs, T ord, T coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche () ;
} ;

template <class T> void pointcol<T>::affiche ()
{  point<T>::affiche () ;
   cout << "    couleur : " << cl << "\n" ;
}
```

Voici un petit exemple d'utilisation (il nécessite les déclarations appropriées ou l'incorporation des fichiers en-tête correspondants) :

```
main()
{ pointcol <int> p1 (2, 5, 1) ; p1.affiche () ;
  pointcol <float> p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

b. Cette fois, la classe dérivée dépend de deux paramètres (nommés ici `T` et `U`). Voici ce que pourrait être la définition de notre patron (avec, toujours, un constructeur en ligne et une fonction `affiche` définie à l'extérieur de la classe) :

```
template <class T, class U> class pointcol : public point<T>
{  U cl ;
  public :
   pointcol (T abs, T ord, U coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche () ;
} ;

template <class T, class U> void pointcol<T, U>::affiche ()
{  point<T>::affiche () ;
   cout << "    couleur : " << cl << "\n" ;
}
```

Voici un exemple d'utilisation (on suppose qu'il est muni des déclarations appropriées) :

```
main()
{
  pointcol <int, short> p1 (2, 5, 1) ; p1.affiche () ;
  pointcol <float, int> p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

c. Cette fois, `pointcol` est une simple classe, ne dépendant plus d'aucun paramètre. Voici ce que pourrait être sa définition :

```
class pointcol : public point<int>
{
   short cl ;
  public :
   pointcol (int abs, int ord, short coul) : point<int> (abs, ord)
     { cl = coul ;
     }
   void affiche ()
     { point<int>::affiche () ;
       cout << "    couleur : " << cl << "\n" ;
     }
} ;
```

Et un petit exemple d'utilisation :

```
main()
{
  pointcol  p1 (2, 5, 1) ; p1.affiche () ;
  pointcol  p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

Exercice 133

Énoncé

On dispose du même patron de classes que précédemment :

```
template <class T> class point
{
   T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche ()
     { cout << "Coordonnees : " << x << " " << y << "\n" ;
     }
} ;
```

a. Lui ajouter une version spécialisée de `affiche` pour le cas où T est le type caractère.

b. Comme dans la question **a** de l'exercice précédent, créer un patron de classes `pointcol` permettant de manipuler des « points colorés » dans lesquels les coordonnées et la couleur sont de même type. On redéfinira convenablement les fonctions membre en réutilisant les fonctions membre de la classe de base et l'on prévoira une version spécialisée de `affiche` de `pointcol` dans le cas du type caractère.

Solution

a.

```
void point<char>::affiche ()
{ cout << "Coordonnees : " << (int)x << " " << (int)y << "\n" ;
}
```

b.

```
template <class T> class pointcol : public point<T>
{  T cl ;
  public :
   pointcol (T abs, T ord, T coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche ()
    {  point<T>::affiche () ;
       cout << "    couleur : " << cl << "\n" ;
    }
} ;

void pointcol<char>::affiche ()
{  point<char>::affiche () ;
   cout << "    couleur : " << (int)cl << "\n" ;
}
```

Remarque

Seule la question **a** de l'exercice 132 se prêtait à une spécialisation pour le type caractère. En effet, pour la classe demandée en **c**, n'ayant plus affaire à un patron de classes, la question n'aurait aucun sens. En ce qui concerne la classe demandée en **b**, en revanche, on se trouve en présence d'une classe dérivée dépendant de 2 paramètres `T` et `U`. Il faudrait alors pouvoir spécialiser une classe, non plus pour des valeurs données de tous les (deux) paramètres, mais pour une valeur donnée (`char`) de l'un d'entre eux ; cette notion de famille de spécialisation n'existe pas dans la norme actuelle de C++.

Exercice 134

Énoncé

On dispose du patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche ()
     { cout << "Coordonnees : " << x << " " << y << "\n" ;
     }
} ;
```

On souhaite créer un patron de classes `cercle` permettant de manipuler des cercles, définis par leur centre (de type `point`) et un rayon. On n'y prévoira, comme fonctions membre, qu'un constructeur et une fonction `affiche` se contentant d'afficher les coordonnées du centre et la valeur du rayon.

a. Le faire par héritage (un cercle est un point qui possède un rayon).

b. Le faire par composition d'objets membre (un cercle possède un point et un rayon).

Solutions

a. La démarche est analogue à celle de la question **a** de l'exercice 132. On a affaire à un patron dépendant de deux paramètres (ici `T` et `U`) :

```
template <class T, class U> class cercle : public point<T>
{  U r ;       // rayon
  public :
   cercle (T abs, T ord, U ray) : point<T> (abs, ord)
     { r = ray ;
     }
   void affiche ()
     { point<T>::affiche () ;
       cout << "    rayon : " << r ;
     }
} ;
```

b. Le patron dépend toujours de deux paramètres (`T` et `U`) mais il n'y a plus de notion d'héritage :

```
template <class T, class U> class cercle
{  point<T> c ;  // centre
   U r ;         // rayon
  public :
   cercle (T abs, T ord, U ray) : c(abs, ord)     // pourrait r(ray)
     { r = ray ;
     }
   void affiche ()
     { c.affiche () ;
       cout << "    rayon : " << r ;
     }
} ;
```

Notez que, dans la définition du constructeur `cercle`, nous avons transmis les arguments `abs` et `ord` à un constructeur de `point` pour le membre `c`. Nous aurions pu utiliser la même notation pour `r`, bien que ce membre soit d'un type de base et non d'un type classe ; cela nous aurait conduit au constructeur suivant (de corps vide) :

```
cercle (T abs, T ord, U ray) : c(abs, ord), r(ray)
  { }
```