12

Les patrons de classes

Le précédent chapitre a montré comment C++ permettait, 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.

Comme nous l'avons fait pour les patrons de fonctions, nous commencerons par vous présenter cette notion de patron de classes à partir d'un exemple simple ne faisant intervenir qu'un paramètre de type. Nous verrons ensuite qu'elle se généralise à un nombre quelconque de paramètres de type et de paramètres expressions. Puis nous examinerons la possibilité de spécialiser un patron de classes, soit en spécialisant certaines de ses fonctions membres, soit en spécialisant toute une classe. Nous ferons alors le point sur l'instanciation de classes patrons, notamment en ce qui concerne l'identité de deux classes. Nous verrons ensuite comment se généralisent les déclarations d'amitiés dans le cas de patrons de classes. Nous terminerons par un exemple d'utilisation de classes patrons imbriquées en vue de manipuler des tableaux (d'objets) à deux indices.

Signalons dès maintenant que malgré leurs ressemblances, les notions de patron de fonctions et de patron de classes présentent des différences assez importantes. Comme vous le constaterez, ce chapitre n'est nullement l'extrapolation aux classes du précédent chapitre consacré aux fonctions.

1 Exemple de création et d'utilisation d'un patron de classes

1.1 Création d'un patron de classes

Nous avons souvent été amené à créer une classe *point* de ce genre (nous ne fournissons pas ici la définition des fonctions membres) :

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

Lorsque nous procédons ainsi, nous imposons que les coordonnées d'un point soient des valeurs de type *int*. Si nous souhaitons disposer de points à coordonnées d'un autre type (*float*, *double*, *long*, *unsigned int*...), 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 ; T 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* ; rappelons que C++ a décidé d'employer le mot clé *class* pour préciser que *T* est un argument de type (pas forcément classe...).

Bien entendu, 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.

Pour une fonction en ligne, les choses restent naturelles ; il suffit simplement d'utiliser le paramètre *T* à bon escient. Voici par exemple comment pourrait être défini notre constructeur :

```
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 (de même qu'avec une classe "ordinaire", il fallait préfixer le nom de la fonction du nom de la classe...) ; 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[1] 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 *point* :

```
#include <iostream>
using namespace std ;
  // création d'un patron de classe
template <class T> class point
{  T x ; T y ;
  public :
   point (T abs=0, T ord=0)
   {  x = abs ; y = ord ;
   }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{  cout << "Coordonnées : " << x << " " << y << "\n" ;
}
```

Création d'un patron de classes

Remarque

Comme on l'a déjà fait remarquer à propos de la définition de patrons de fonctions, depuis la norme, le mot clé *class* peut être remplacé par *typename*[2].

1. Stroustrup, le concepteur du langage C++, se contente de mentionner cette redondance, sans la justifier !

2. En toute rigueur, ce mot clé peut également servir à lever une ambiguïté pour le compilateur, en l'ajoutant en préfixe à un identificateur afin qu'il soit effectivement interprété comme un nom de type. Par exemple, avec cette déclaration :

typename A::truc a ; // équivalent à A::truc a ; si aucune ambiguité n'existe

on précise que *A::truc* est bien un nom de type ; on déclare donc *a* comme étant de type *A::truc*.

Il est rare que l'on ait besoin de recourir à cette possibilité.

1.2 Utilisation d'un patron de classes

Après avoir créé ce patron, une déclaration telle que :

```
point <int> ai ;
```

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 <double> ad ;
```

le compilateur instancie la définition d'une classe *point* dans laquelle le paramètre *T* prend la valeur *double*, 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> ai (3, 5) ;
point <double> ad (3.5, 2.3) ;
```

1.3 Contraintes d'utilisation d'un patron de classes

Comme on peut s'y attendre, les instructions définissant un patron de classes sont des déclarations au même titre que les instructions définissant une classe (y compris les instructions de définition de fonctions en ligne).

Mais il en va de même pour les fonctions membres qui ne sont pas en ligne : leurs instructions sont nécessaires au compilateur pour instancier chaque fois que nécessaire les instructions requises. On retrouve ici la même remarque que celle que nous avons formulée pour les patrons de fonctions (voir paragraphe 1.4 du chapitre 11).

Aussi n'est-il pas possible de livrer à un utilisateur une classe patron toute compilée : il faut lui fournir les instructions source de toutes les fonctions membres (alors que pour une classe "ordinaire", il suffit de lui fournir la déclaration de la classe et un module objet correspondant aux fonctions membres).

Tout se passe encore ici comme s'il existait deux niveaux de déclarations. Par la suite, nous continuerons cependant à parler de "définition d'un patron".

En pratique, on placera les définitions de patron dans un fichier approprié d'extension *h*.

Remarque

Ici encore, les considérations précédentes doivent en fait être pondérées par le fait que la norme a introduit le mot clé *export*. Appliqué à la définition d'un patron de classes, il précise que celle-ci sera accessible depuis un autre fichier source. Par exemple, on pourra définir un patron de classes *point* de cette façon :

```
export template <class T> class point
{  T x ; T y ;
  public :
   point (...) ;
   void afiche () ;
    .....
} ;
template <class T> point<T>::point(...) { ..... }  /* définition constructeur */
template <class T> void point<T>::affiche() { ..... }    /* définition affiche */
  .....
```

On peut alors utiliser ce patron depuis un autre fichier source, en se contentant de mentionner sa seule "déclaration" (comme avec les patrons de fonctions, on distingue alors déclaration et définition) :

```
template <class T> point<T>    // déclaration seule de point<T>
{  T x ; T y ;
   point (...) ;
   void afiche () ;
    .....
} ;
```

Ici encore, on aura intérêt à prévoir deux fichiers en-têtes distincts, un pour la déclaration, un pour la définition. Le premier sera inclus dans la définition du patron et dans son utilisation.

Rappelons que ce mécanisme met en jeu une sorte de "précompilation" des définitions de patrons.

1.4 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,
- un exemple d'utilisation (*main*).

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

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

main ()
{
   point <int> ai (3, 5) ;        ai.affiche () ;
   point <char> ac ('d', 'y') ;   ac.affiche () ;
   point <double> ad (3.5, 2.3) ; ad.affiche () ;
}
```

```
coordonnees : 3 5
coordonnees : d y
coordonnees : 3.5 2.3
```

Création et utilisation d'un patron de classes

Remarques

1 Le comportement de *point<char>* est satisfaisant si nous souhaitons effectivement disposer de points repérés par de vrais caractères. En revanche, si nous avons utilisé le type *char* pour disposer de "petits entiers", le résultat est moins satisfaisant. En effet, nous pourrons toujours déclarer un point de cette façon :

```
point <char> pc (4, 9) ;
```

Mais le comportement de la fonction *affiche* ne nous conviendra plus (nous obtiendrons les caractères ayant pour code les coordonnées du point !).

Nous verrons qu'il reste toujours possible de modifier cela en "spécialisant" notre classe *point* pour le type *char* ou encore en spécialisant la fonction *affiche* pour la classe *point<char>*.

2 A priori, on a plutôt envie d'appliquer notre patron *point* à des types *T* standard. Toutefois, rien n'interdit de l'appliquer à un type classe *T* quelconque, même s'il peut alors s'avérer difficile d'attribuer une signification à la classe patron ainsi obtenue. Il faut cependant qu'il existe une conversion de *int* en *T*, utile pour convertir la valeur 0 dans le

type *T* lors de l'initialisation des arguments du constructeur de *point* (sinon, on obtiendra une erreur de compilation). De plus, il est nécessaire que la recopie et l'affectation d'objets de type *T* soient correctement prises en compte (dans le cas contraire, aucun diagnostic ne sera fourni à la compilation ; les conséquences n'en seront perçues qu'à l'exécution).

2 Les paramètres de type d'un patron de classes

Tout comme les patrons de fonctions, les patrons de classes peuvent comporter des paramètres de type et des paramètres expressions. Ce paragraphe étudie les premiers ; les seconds seront étudiés au paragraphe suivant. Une fois de plus, notez bien que, malgré leur ressemblance avec les patrons de fonctions, les contraintes relatives à ces différents types de paramètres ne seront pas les mêmes.

2.1 Les paramètres de type dans la création d'un patron de classes

Les paramètres de type peuvent être en nombre quelconque et utilisés comme bon vous semble dans la définition du patron de classes. En voici un exemple :

```
template <class T, class U, class V> // liste de trois param. de nom (muet) T, U et V
class essai
{  T x ;              // un membre x de type T
   U t[5]       ;     // un tableau t de 5 éléments de type U
     ...
   V fm1 (int, U) ;   // déclaration d'une fonction membre recevant 2 arguments
                      //  de type int et U et renvoyant un résultat de type V
     ...
} ;
```

2.2 Instanciation d'une classe patron

Rappelons que nous nommons "classe patron" une instance particulière d'un patron de classes.

Une classe patron se déclare simplement en fournissant à la suite du nom de patron un nombre d'arguments effectifs (noms de types) égal au nombre de paramètres figurant dans la liste (*template* < ...>) du patron. Voici des déclarations de classes patron obtenues à partir du patron *essai* précédent (il ne s'agit que de simples exemples d'école auxquels il ne faut pas chercher à attribuer une signification précise) :

```
essai <int, float, int> ce1 ;
essai <int, int *, double > ce2 ;
essai <char *, int, obj>  ce3 ;
```

La dernière suppose bien sûr que le type *obj* a été préalablement défini (il peut s'agir d'un type classe).

Il est même possible d'utiliser comme paramètre de type effectif un type instancié à l'aide d'un patron de classes. Par exemple, si nous disposons du patron de classes nommé *point* tel qu'il a été défini dans le paragraphe précédent, nous pouvons déclarer :

```
essai <float, point<int>, double> ce4 ;
essai <point<int>, point<float>, char *> ce5 ;
```

Remarques

1 Les problèmes de correspondance exacte rencontrés avec les patrons de fonctions n'existent plus pour les patrons de classes (du moins pour les paramètres de types étudiés ici). En effet, dans le cas des patrons de fonctions, l'instanciation se fondait non pas sur la liste des paramètres indiqués à la suite du mot clé *template*, mais sur la liste des paramètres de l'en-tête de la fonction ; un même nom (muet) pouvait apparaître deux fois et il y avait donc risque d'absence de correspondance.

2 Il est tout à fait possible qu'un argument formel (figurant dans l'en-tête) d'une fonction patron soit une classe patron. En voici un exemple, dans lequel nous supposons défini le patron de classes nommé *point* (ce peut être le précédent) :

```
template <class T> void fct (point<T>)
{ ..... }
```

Lorsqu'il devra instancier une fonction *fct* pour un type T donné, le compilateur instanciera également (si cela n'a pas encore été fait) la classe patron *point*<*T*>.

3 Comme dans le cas des patrons de fonctions, on peut rencontrer des difficultés lorsque l'on doit initialiser (au sein de fonctions membres) des variables dont le type figure en paramètre. En effet, il peut s'agir d'un type de base ou, au contraire, d'un type classe. Là encore, la nouvelle syntaxe d'initialisation des types standard (présentée au paragraphe 2.3 du chapitre 11) permet de résoudre le problème.

4 Un patron de classes peut comporter des membres (données ou fonctions) statiques. Dans ce cas, il faut savoir que chaque instance de la classe dispose de son propre jeu de membres statiques : on est en quelque sorte "statique au niveau de l'instance et non au niveau du patron". C'est logique puisque le patron de classes n'est qu'un moule utilisé pour instancier différentes classes ; plus précisément, un patron de classes peut toujours être remplacé par autant de définitions différentes de classes que de classes instanciées.

3 Les paramètres expressions d'un patron de classes

Un patron de classes peut comporter des paramètres expressions. Bien qu'il s'agisse, ici encore, d'une notion voisine de celle présentée pour les patrons de fonctions, certaines différences importantes existent. En particulier, les valeurs effectives d'un paramètre expression devront obligatoirement être constantes dans le cas des classes.

3.1 Exemple

Supposez que nous souhaitions définir une classe *tableau* susceptible de manipuler des tableaux d'objets d'un type quelconque. L'idée vient tout naturellement à l'esprit d'en faire une classe patron possédant un paramètre de type. On peut aussi prévoir un second paramètre permettant de préciser le nombre d'éléments du tableau.

Dans ce cas, la création de la classe se présentera ainsi :

```
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   // .....
} ;
```

La liste de paramètres (*template* <...>) comporte deux paramètres de nature totalement différente :

- un paramètre (désormais classique) de type, introduit par le mot clé *class*,
- un "paramètre expression" de type *int* ; on précisera sa valeur lors de la déclaration d'une instance particulière de la classe *tableau*.

Par exemple, avec la déclaration :

```
tableau <int, 4> ti ;
```

nous déclarerons une classe nommée *ti* correspondant finalement à la déclaration suivante :

```
class ti
{  int tab [4] ;
  public :
   // .....
} ;
```

Voici un exemple complet de programme définissant un peu plus complètement une telle classe patron nommée *tableau* ; nous l'avons simplement dotée de l'opérateur [] et d'un constructeur (sans arguments) qui ne se justifie que par le fait qu'il affiche un message approprié. Nous avons instancié des "tableaux" d'objets de type *point* (ici, *point* est à nouveau une classe "ordinaire" et non une classe patron).

```
#include <iostream>
using namespace std ;
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   tableau () { cout << "construction tableau \n" ; }
   T & operator [] (int i)
    { return tab[i] ;
    }
} ;
class point
{  int x, y ;
  public :
   point (int abs=1, int ord=1 )   // ici init par défaut à 1
     { x=abs ; y=ord ;
       cout << "constr point " << x << " " << y << "\n" ;
     }
   void affiche () { cout << "Coordonnees : " << x << " " << y << "\n" ; }
} ;
main()
{  tableau <int,4> ti ;
   int i ; for (i=0 ; i<4 ; i++) ti[i] = i ;
   cout << "ti : " ;
   for (i=0 ; i<4 ; i++) cout << ti[i] << " " ;
   cout << "\n" ;
   tableau <point, 3> tp ;
   for (i=0 ; i<3 ; i++) tp[i].affiche() ;
}
```

```
construction tableau
ti : 0 1 2 3
const point 1 1
const point 1 1
const point 1 1
construction tableau
coordonnées : 1 1
coordonnées : 1 1
coordonnées : 1 1
```

Exemple de classe patron comportant un paramètre expression

Remarque

La classe *tableau* telle qu'elle est présentée ici n'a pas véritablement d'intérêt pratique. En effet, on obtiendrait le même résultat en déclarant de simples tableaux d'objets, par exemple *int ti[4]* au lieu de *tableau <int,4> ti*. En fait, il ne s'agit que d'un cadre initial qu'on

peut compléter à loisir. Par exemple, on pourrait facilement y ajouter un contrôle d'indice en adaptant la définition de l'opérateur [] ; on pourrait également prévoir d'initialiser les éléments du tableau. C'est d'ailleurs ce que nous aurons l'occasion de faire au paragraphe 9, où nous utiliserons le patron *tableau* pour manipuler des tableaux à plusieurs indices.

3.2 Les propriétés des paramètres expressions

On peut faire apparaître autant de paramètres expressions qu'on le désire dans une liste de paramètres d'un patron de classes. Ces paramètres peuvent intervenir n'importe où dans la définition du patron, au même titre que n'importe quelle expression constante peut apparaître dans la définition d'une classe.

Lors de l'instanciation d'une classe comportant des paramètres expressions, les paramètres effectifs correspondants doivent obligatoirement être des expressions constantes[1] d'un type rigoureusement identique à celui prévu dans la liste d'arguments (aux conversions triviales près) ; autrement dit, aucune conversion n'est possible.

Contrairement à ce qui passait pour les patrons de fonctions, il n'est pas possible de surdéfinir un patron de classes, c'est-à-dire de créer plusieurs patrons de même nom mais comportant une liste de paramètres (de type ou expressions) différents. En conséquence, les problèmes d'ambiguïté évoqués lors de l'instanciation d'une fonction patron ne peuvent plus se poser dans le cas de l'instanciation d'une classe patron.

Sur un plan méthodologique, on pourra souvent hésiter entre l'emploi de paramètres expressions et la transmission d'arguments au constructeur. Ainsi, dans l'exemple de classe tableau, nous aurions pu ne pas prévoir le paramètre expression *n* mais, en revanche, transmettre au constructeur le nombre d'éléments souhaités. Une différence importante serait alors apparue au niveau de la gestion des emplacements mémoire correspondant aux différents éléments du tableau :

- attribution d'emplacement à la compilation (statique ou automatique suivant la classe d'allocation de l'objet de type *tableau*<...,...> correspondant) dans le premier cas,
- allocation dynamique par le constructeur dans le second cas.

4 Spécialisation d'un patron de classes

Nous avons vu qu'il était possible de "spécialiser" certaines fonctions d'un patron de fonctions. Si la même possibilité existe pour les patrons de classes, elle prend toutefois un aspect légèrement différent, à la fois au niveau de sa syntaxe et de ses possibilités, comme nous le verrons après un exemple d'introduction.

1. Cette contrainte n'existait pas pour les paramètres expressions des patrons de fonctions ; mais leur rôle n'était pas le même.

4.1 Exemple de spécialisation d'une fonction membre

Un patron de classes définit une famille de classes dans laquelle chaque classe comporte à la fois sa définition et la définition de ses fonctions membres. Ainsi, toutes les fonctions membres de nom donné réalisent le même algorithme. Si l'on souhaite adapter une fonction membre à une situation particulière, il est possible d'en fournir une nouvelle.

Voici un exemple qui reprend le patron de classes *point* défini dans le premier paragraphe. Nous y avons spécialisé la fonction *affiche* dans le cas du type *char*, afin qu'elle affiche non plus des caractères mais des nombres entiers.

```
#include <iostream>
using namespace std ;
  // création d'un patron de classe
template <class T> class point
{  T x ; T y ;
  public :
   point (T abs=0, T ord=0)
   {  x = abs ; y = ord ;
   }
   void affiche () ;
} ;
  // définition de la fonction affiche
template <class T> void point<T>::affiche ()
{ cout << "Coordonnees : " << x << " " << y << "\n" ;
}
  // ajout d'une fonction affiche spécialisée pour les caractères
void point<char>::affiche ()
{  cout << "Coordonnees : " << (int)x << " " << (int)y << "\n" ;
}
main ()
{  point <int> ai (3, 5) ;        ai.affiche () ;
   point <char> ac ('d', 'y') ;   ac.affiche () ;
   point <double> ad (3.5, 2.3) ; ad.affiche () ;
}
```

```
coordonnées : 3 5
coordonnées : 100 121
coordonnées : 3.5 2.3
```

Exemple de spécialisation d'une fonction membre d'une classe patron

Notez qu'il nous a suffi d'écrire l'en-tête de *affiche* sous la forme :

```
void point<char>::affiche ()
```

pour préciser au compilateur qu'il devait utiliser cette fonction à la place de la fonction *affiche* du patron *point*, c'est-à-dire à la place de l'instance *point<char>*.

4.2 Les différentes possibilités de spécialisation

4.2.1 On peut spécialiser une fonction membre pour tous les paramètres

Dans notre exemple, la classe patron *point* ne comportait qu'un paramètre de type. Il est possible de spécialiser une fonction membre en se basant sur plusieurs paramètres de type, ainsi que sur des valeurs précises d'un ou plusieurs paramètres expressions (bien que cette dernière possibilité nous paraisse d'un intérêt limité). Par exemple, considérons le patron *tableau* défini au paragraphe 3.1 :

```
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   tableau () { cout << "construction tableau \n" ; }
   //  .....
} ;
```

Nous pouvons écrire une version spécialisée de son constructeur pour les tableaux de 10 éléments de type *point* (il ne s'agit vraiment que d'un exemple d'école !) en procédant ainsi :

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

4.2.2 On peut spécialiser une fonction membre ou une classe

Dans les exemples précédents, nous avons spécialisé une fonction membre d'un patron. En fait, on peut indifféremment :

- spécialiser une ou plusieurs fonctions membres, sans modifier la définition de la classe elle-même (ce sera la situation la plus fréquente),
- spécialiser la classe elle-même, en en fournissant une nouvelle définition ; cette seconde possibilité peut s'accompagner de la spécialisation de certaines fonctions membres.

Par exemple, après avoir défini le patron *template <class T> class point* (comme au paragraphe 4.1), nous pourrions définir une version spécialisée de la classe *point* pour le type *char*; c'est-à-dire une version appropriée de l'instance *point<char>*, en procédant ainsi :

```
class point <char>
{ // nouvelle définition
}
```

Nous pourrions aussi définir des versions spécialisées de certaines des fonctions membre de *point<char>* en procédant comme précédemment ou ne pas en définir, auquel cas on ferait appel aux fonctions membres du patron.

4.2.3 On peut prévoir des spécialisations partielles de patrons de classes

Nous avons déjà parlé de spécialisation partielle dans le cas de patrons de fonctions (voir au paragraphe 5 du chapitre 11). La norme ANSI autorise également la spécialisation partielle d'un patron de classes[1]. En voici un exemple :

1. Cette possibilité n'existait pas dans la version 3. Elle a été introduite par la norme ANSI et elle n'est pas encore reconnue de tous les compilateurs.

```
template <class T, class U> class A       { ..... } ;  // patron I
template <class T>         class A <T, T*> { ..... } ;  // patron II
```

Une déclaration telle que *A <int, float> a1* utilisera le patron I, tandis qu'une déclaration telle que *A<int, int *> a2* utilisera le patron II plus spécialisé.

5 Paramètres par défaut

Dans la définition d'un patron de classes, il est possible de spécifier des valeurs par défaut pour certains paramètres[1], suivant un mécanisme semblable à celui utilisé pour les paramètres de fonctions usuelles. Voici quelques exemples :

```
template <class T, class U=float> class A { ..... } ;
template <class T, int n=3>       class B { ..... } ;
   .....
A<int,long> a1 ;      /* instanciation usuelle           */
A<int> a2 ;           /* équivaut à A<int, float> a2 ;   */
B<int, 3> b1 ;        /* instanciation usuelle           */
B<int> b2 ;           /* équivaut à B<int, 3> b2 ;       */
```

Remarque

La notion de paramètres par défaut n'a pas de signification pour les patrons de fonctions.

6 Patrons de fonctions membres

Le mécanisme de définition de patrons de fonctions peut s'appliquer à une fonction membre d'une classe ordinaire, comme dans cet exemple :

```
class A
{ .....
  template <class T> void fct (T a) { ..... }
  .....
} ;
```

Cette possibilité peut s'appliquer à une fonction membre d'une classe patron, comme dans cet exemple[2] :

```
template <class T> class A
{ .....
  template <class U> void fct (U x, T y)   /* ici le type T est utilisé, mais */
     { ..... }                                /* il pourrait ne pas l'être       */
  .....
} ;
```

Dans ce dernier cas, l'instanciation de la bonne fonction *fct* se fondera à la fois sur la classe à laquelle elle appartient et sur la nature de son premier argument.

1. Cette possibilité a été introduite par la norme ANSI.
2. Cette possibilité a été introduite par la norme ANSI

7 Identité de classes patrons

Nous avons déjà vu que l'opérateur d'affectation pouvait s'appliquer à deux objets d'un même type. L'expression "même type" est parfaitement définie, tant que l'on n'utilise pas d'instances de patron de classes : deux objets sont de même type s'ils sont déclarés avec le même nom de classe. Mais que devient cette définition dans le cas d'objets dont le type est une instance particulière d'un patron de classes ?

En fait, deux classes patrons correspondront à un même type si leurs paramètres de type correspondent exactement au même type et si leurs paramètres expressions ont la même valeur.

Ainsi, en supposant que nous disposions du patron *tableau* défini au paragraphe 3.1, avec ces déclarations :

```
tableau <int, 12> t1 ;
tableau <float, 12> t2 ;
```

vous n'aurez pas le droit d'écrire :

```
t2 = t1 ;    // incorrect car valeurs différentes du premier paramètre (float et int)
```

De même, avec ces déclarations :

```
tableau <int, 15> ta ;
tableau <int, 20> tb ;
```

vous n'aurez pas le droit d'écrire :

```
ta = tb ;     // incorrect car valeurs différentes du second paramètre (15 et 20)
```

Ces règles, apparemment restrictives, ne servent en fait qu'à assurer un bon fonctionnement de l'affectation, qu'il s'agisse de l'affectation par défaut (membre à membre : il faut donc bien disposer exactement des mêmes membres dans les deux objets) ou de l'affectation surdéfinie (pour que cela fonctionne toujours, il faudrait que le concepteur du patron de classe prévoie toutes les combinaisons possibles et, de plus, être sûr qu'une éventuelle spécialisation ne risque pas de perturber les choses...).

Certes, dans le premier cas (*t2=t1*), une conversion *int->float* nous aurait peut-être convenu. Mais pour que le compilateur puisse la mettre en œuvre, il faudrait qu'il "sache" qu'une classe *tableau<int, 10>* ne comporte que des membres de type *int*, qu'une classe *tableau<float, 10>* ne comporte que des membres de type *float*, que les deux classes ont le même nombre de membres données...

8 Classes patrons et déclarations d'amitié

L'existence des patrons de classes introduit de nouvelles possibilités de déclaration d'amitié.

8.1 Déclaration de classes ou fonctions "ordinaires" amies

La démarche reste celle que nous avons rencontrée dans le cas des classes ordinaires. Par exemple, si A est une classe ordinaire et *fct* une fonction ordinaire :

```
template <class T>
class essai
{  int x ;
  public :
   friend class A ;          // A est amie de toute instance du patron essai
   friend int fct (float) ;  // fct est amie de toute instance du patron essai
   ...
} ;
```

8.2 Déclaration d'instances particulières de classes patrons ou de fonctions patrons

En fait, cette possibilité peut prendre deux aspects différents selon que les paramètres utilisés pour définir l'instance concernée sont effectifs ou muets (définis dans la liste de paramètres du patron de classe).

Supposons que *point* est une classe patron ainsi définie :

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

et *fct* une fonction patron ainsi définie :

```
template <class T> int fct (T x) { ... }
```

Voici un exemple illustrant le premier aspect :

```
template <class T, class U>
class essai1
{  int x ;
  public :
   friend class point<int> ;     // la classe patron point<int> est amie
                                 // de toutes les instances de essai1
   friend int fct (double) ;     // la fonction patron int fct (double
                                 // de toutes les instances de essai1
   ...
} ;
```

Voici un exemple illustrant le second aspect :

```
template <class T, class U>
class essai2
{  int x ;
  public :
   friend class point<T> ;
   friend int fct (U) ;
}
```

Notez bien, que dans le second cas, on établit un "couplage" entre la classe patron générée par le patron *essai2* et les déclarations d'amitié correspondantes. Par exemple, pour l'intance *essai2 <int, double>*, les déclarations d'amitié porteront sur *point<int>* et *int fct (double)*.

8.3 Déclaration d'un autre patron de fonctions ou de classes

Voici un exemple faisant appel aux mêmes patrons *point* et *fct* que ci-dessus :

```
template <class T, class U>
class essai2
{ int x ;
 public :
  template <class X> friend class point <X> ;
  template <class X> friend class int fct (point <X>) ;
} ;
```

Cette fois, toutes les instances du patron *point* sont amies de n'importe quelle instance du patron *essai2*. De même, toutes les instances du patron de fonctions *fct* sont amies de n'importe quelle instance du patron *essai2*.

9 Exemple de classe tableau à deux indices

Nous avons vu à plusieurs reprises comment surdéfinir l'opérateur [] au sein d'une classe tableau. Néanmoins, nous nous sommes toujours limité à des tableaux à un indice.

Ici, nous allons voir qu'il est très facile, une fois qu'on a défini un patron de tableau à un indice, de l'appliquer à un tableau à deux indices (ou plus) par le simple jeu de la composition des patrons.

Si nous considérons pour l'instant la classe *tableau* définie de cette façon simplifiée :

```
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   T & operator [] (int i)       // opérateur []
     { return tab[i] ;
     }
} ;
```

nous pouvons tout à fait déclarer :

```
tableau <tableau<int,2>,3> t2d ;
```

En effet, *t2d* est un tableau de 3 éléments ayant chacun le type *tableau <int,2>* ; autrement dit, chacun de ces 3 éléments est lui-même un tableau de 2 entiers.

Une notation telle que *t2d [1] [2]* a un sens ; elle représente la référence au troisième élément de *t2d [1]*, c'est-à-dire au troisième élément du deuxième tableau de deux entiers de *t2d*.

Voici un exemple complet (mais toujours simplifié) illustrant cela. Nous avons simplement ajouté artificiellement un constructeur afin d'obtenir une trace des différentes constructions.

```
// implémentation d'un tableau à deux dimensions
#include <iostream>
using namespace std ;
```

```
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   tableau ()                    // constructeur
     {cout << "construction tableau a " << n << " elements\n" ;
     }
   T & operator [] (int i)       // opérateur []
     { return tab[i] ;
     }
} ;
main()
{   tableau <tableau<int,2>,3> t2d ;
    t2d [1] [2] = 15 ;
    cout << "t2d [1] [2] = " << t2d [1] [2] << "\n" ;
    cout << "t2d [0] [1] = " << t2d [0] [1] << "\n" ;
}
```

```
construction tableau a 2 elements
construction tableau a 2 elements
construction tableau a 2 elements
construction tableau a 3 elements
t2d [1] [2] = 15
t2d [0] [1] = -858993460
```

Utilisation du patron tableau pour manipuler des tableaux à deux indices (1)

On notera bien que notre patron *tableau* est a priori un tableau à un indice. Seule la manière dont on l'utilise permet de l'appliquer à des tableaux à un nombre quelconque d'indices.

Manifestement, cet exemple est trop simpliste ; d'ailleurs, tel quel, il n'apporte rien de plus qu'un banal tableau. Pour le rendre plus réaliste, nous allons prévoir :

- de gérer les débordements d'indices : ici, nous nous contenterons d'afficher un message et de "faire comme si" l'utilisateur avait fourni un indice nul[1] ;
- d'initialiser tous les éléments du tableau lors de sa construction : nous utiliserons pour ce faire la valeur 0. Mais encore faut-il que la chose soit possible, c'est-à-dire que, quel que soit le type T des éléments du tableau, on puisse leur affecter la valeur 0. Cela signifie qu'il doit exister une conversion de T en *int*. Il est facile de la réaliser avec un constructeur à un élément de type *int*. Du même coup, cela permettra de prévoir une valeur initiale lors de la déclaration d'un tableau (par sécurité, nous prévoirons la valeur 0 par défaut).

Voici la classe ainsi modifiée et un exemple d'utilisation :

1. Il pourrait également être judicieux de déclencher une "exception", comme nous apprendrons à le faire, sur ce même exemple, au paragraphe 1 du chapitre 17.

```
  // implémentation d'un tableau 2d avec test débordement d'indices
#include <iostream>
using namespace std ;
template <class T, int n> class tableau
{  T tab [n] ;
   int limite ;         // nombre d'éléments du tableau
  public :
   tableau (int init=0)
     {  int i ;
        for (i=0 ; i<n ; i++) tab[i] = init ;
                     // il doit exister un constructeur à un argument
                     // pour le cas où tab[i] est un objet
        limite = n-1 ;
        cout << "appel constructeur tableau de taille " << n
             << " init = " << init << "\n" ;
     }
   T & operator [] (int i)
   { if (i<0 || i>limite) { cout << "--debordement " << i << "\n" ;
                            i=0 ;  // choix arbitraire
                          }
     return tab[i] ;
   }
} ;
main()
{   tableau <tableau<int,3>,2> ti ;         // pas d'initialisation
    tableau <tableau<float,4>,2> td (10) ;  // initialisation à 10
    ti [1] [6] = 15 ;
    ti [8] [-1] = 20 ;
    cout << ti [1] [2] << "\n" ;   // élément initialisé à valeur par défaut (0)
    cout << td [1] [0] << "\n" ;   // élément initialisé explicitement
}
```

```
appel constructeur tableau de taille 3 init = 0
appel constructeur tableau de taille 3 init = 0
appel constructeur tableau de taille 3 init = 0
appel constructeur tableau de taille 3 init = 0
appel constructeur tableau de taille 2 init = 0
appel constructeur tableau de taille 4 init = 0
appel constructeur tableau de taille 4 init = 0
appel constructeur tableau de taille 4 init = 10
appel constructeur tableau de taille 4 init = 10
appel constructeur tableau de taille 2 init = 10
--debordement 6
--debordement 8
--debordement -1
0
10
```

Utilisation du patron tableau pour manipuler des tableaux à deux indices (2)

Remarque

Si vous examinez bien les messages de construction des différents tableaux, vous observerez que l'on obtient deux fois plus de messages que prévu pour les tableaux à un indice. L'explication réside dans l'instruction *tab[i] = init* du constructeur *tableau*. En effet, lorsque *tab[i]* désigne un élément de type de base, il y a simplement conversion de la valeur entière *init* dans ce type de base. En revanche, lorsque l'on a affaire à un objet de type T (ici T est de la forme *tableau*<...>), cette instruction provoque l'appel du constructeur *tableau(int)* pour créer un objet temporaire de ce type. Cela se voit très clairement dans le cas du tableau *td*, pour lequel on trouve une construction d'un tableau temporaire initialisé avec la valeur 0 et une construction d'un tableau initialisé avec la valeur 10.