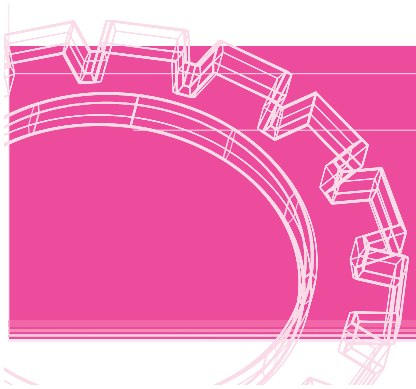
Chapitre 17

Les patrons de fonctions



Rappels

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

La version 3 limitait les paramètres d'un patron de fonctions à des paramètres de type. La norme ANSI a, en outre, introduit les « paramètres expression ».

Définition d'un patron de fonctions

On précise les paramètres (muets) de type, en faisant précéder chacun du mot (relativement arbitraire) `class` sous la forme `template <class ..., class ..., ...>`. La définition de la fonction est classique, hormis le fait que les paramètres muets de type peuvent être employés n'importe où un type effectif est permis.

Par exemple :

```
template <class T, class U> void fct (T a, T * b, U c)
{
   T x ;               // variable locale x de type T
   U *  adr ;          // variable locale adr de type U *
   ...
   adr = new T [10] ;  // allocation tableau de 10 éléments de type T
   ...
   n = sizeof (T) ;    // une instruction utilisant le type T
   ...
}
```

Remarque

Une instruction telle que (T désignant un type quelconque) :

```
T x (3) ;
```

est légale même si T n'est pas un type classe ; dans ce dernier cas, elle est simplement équivalente à :

```
T x = 3 ;
```

Instanciation d'une fonction patron

Chaque fois qu'on utilise une fonction ayant un nom de patron, le compilateur cherche à utiliser ce patron pour créer (instancier) une fonction adéquate. Pour ce faire, il cherche à réaliser une **correspondance absolue** des types : aucune conversion, qu'il s'agisse de promotion numérique ou de conversion standard n'est permise ; qui plus est, les qualifieurs `const` et `volatile` doivent être exactement les mêmes.

Voici des exemples utilisant notre patron précédent :

```
int n, p ; float x ; char c ;
int * adi ; float * adf ;
class point ; point p ; point * adp ;

fct (n, adi, x) ;   // instancie la fonction void fct (int, int *, float)
fct (n, adi, p)     // instancie la fonction void fct (int, int *, int)
fct (x, adf, p) ;   // instancie la fonction void fct (float, float *, int)
fct (c, adi, x) ;   // erreur char et int * ne correspondent pas à T et T*
                    //         ( pas de conversion)
fct (&n, &adi, x) ; // instancie la fonction void fct (int *, int * *, float)
fct (p, adp, n) ;   // instancie la fonction void fct (point, point *, int)
```

D'une manière générale, il est nécessaire que **chaque paramètre de type** apparaisse **au moins une fois dans l'en-tête** du patron.

Remarque

La définition d'un patron de fonctions ne peut pas être compilée seule ; de toute façon, elle doit être connue du compilateur pour qu'il puisse instancier la bonne fonction patron. En général, les définitions de patrons de fonctions figureront dans des fichiers d'extension h, de façon à éviter d'avoir à en fournir systématiquement la liste.

Les paramètres expression d'un patron de fonctions

Ils ont été introduits par la norme. Un paramètre expression d'un patron de fonctions se présente comme un argument usuel de fonction ; il n'apparaît pas dans la liste de paramètres de type (`template`) et il doit apparaître dans l'en-tête du patron. Par exemple :

```
template <class T> int compte (T * tab, int n)
{  // ici, on peut se servir de la valeur de l'entier n
   // comme on le ferait dans n'importe quelle fonction ordinaire
}
```

Un patron de fonctions peut disposer d'un ou de plusieurs paramètres expression. Lors de l'appel, leur type n'a plus besoin de correspondre exactement à celui attendu : il suffit qu'il soit acceptable par affectation, comme dans n'importe quel appel d'une fonction ordinaire.

Surdéfinition de patrons de fonctions et spécialisation de fonctions de patrons

On peut définir plusieurs patrons de même nom, possédant des paramètres (de type ou expression) différents. La seule règle à respecter dans ce cas est que l'appel d'une fonction de ce nom ne doit pas conduire à une ambiguïté : un seul patron de fonctions doit pouvoir être utilisé à chaque fois.

Par ailleurs, il est possible de fournir la définition d'une ou plusieurs fonctions particulières qui seront utilisées en lieu et place de celle instanciée par un patron. Par exemple, avec :

```
template <class T> T min (T a, T b)   // patron de fonctions
{ ... }
char * min (char * cha, char * chb)   // version spécialisée pour le type char *
{ ... }
int n, p;
char * adr1, * adr2 ;

min (n, p)        // appelle la fonction instanciée par le patron général
                  //  soit ici  :    int min (int, int)
min (adr1, adr2)  // appelle la fonction spécialisée
                  //                 char * min (char *, char *)
```

Algorithme d'instanciation ou d'appel d'une fonction

Précisons comment doivent être aménagées les règles de recherche d'une fonction surdéfinie, dans le cas où il existe un ou plusieurs patrons de fonctions.

Lors d'un appel de fonction, le compilateur recherche tout d'abord une correspondance exacte avec les fonctions « ordinaires ». S'il y a ambiguïté, la recherche échoue (comme à l'accoutumée). Si aucune fonction « ordinaire » ne convient, on examine alors tous les patrons ayant le nom voulu (en ne considérant que les paramètres de type). Si une seule correspondance exacte est trouvée, la fonction correspondante est instanciée (du moins, si elle ne l'a pas déjà été) et le problème est résolu. S'il y en a plusieurs, la recherche échoue.

Enfin, si aucun patron de fonction ne convient, on examine à nouveau toutes les fonctions « ordinaires » en les traitant cette fois comme de simples fonctions surdéfinies (promotions numériques, conversions standard…).

Exercice 123

Énoncé

Créer un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque (le résultat possédera le même type). Écrire un petit programme utilisant ce patron.

Solution

Ici, notre patron ne comportera qu'un seul paramètre de type (correspondant à la fois à l'unique argument et à la valeur de retour de la fonction). Sa définition ne pose pas de problème particulier.

```
#include <iostream>
using namespace std ;
template <class T> T carre (T a)
  { return a * a ;
  }
main()
{ int n = 5 ;
  float x = 1.5 ;
  cout << "carre de " << n << " = " << carre (n) << "\n" ;
  cout << "carre de " << x << " = " << carre (x) << "\n" ;
}
```

Exercice 124

Énoncé

Soit cette définition de patron de fonctions :

```
template <class T, class U> T fct (T a, U b, T c)
{ .....
}

Avec les déclarations suivantes :
int n, p, q ;
float x ;
char t[20] ;
char c ;
```

Quels sont les appels corrects et, dans ce cas, quels sont les prototypes des fonctions instanciées ?

```
fct (n, p, q) ;          // appel I
fct (n, x, q) ;          // appel II
fct (x, n, q) ;          // appel III
fct (t, n, &c) ;         // appel IV
```

Solution

a. L'appel I est correct ; il instancie la fonction :

```
int fct (int, int, int)
```

b. L'appel II est correct ; il instancie la fonction :

```
int fct (int, float, int)
```

c. L'appel III est incorrect.

d. L'appel IV est correct selon la norme. Il instancie la fonction :

```
char * fct (char *, int, char *)
```

Remarque

L'appelþIV n'était pas accepté dans la version 3 qui ne considérait pas `char *` comme une correspondance absolue pour `char[20]`.

Exercice 125

Énoncé

Créer un patron de fonctions permettant de calculer la somme d'un tableau d'éléments de type quelconque, le nombre d'éléments du tableau étant fourni en paramètre (on supposera que l'environnement utilisé accepte les « paramètres expression »). Écrire un petit programme utilisant ce patron.

Solution

```
// définition du patron de fonctions
template <class T> T somme (T * tab, int nelem)
{ T som ;
  int i ;
  som = 0 ;
  for (i=0 ; i<nelem ; i++)  som = som + tab[i] ;
  return som ;
}

// exemple d'utilisation
#include <iostream>
using namespace std ;
main()
{ int ti[] = {3, 5, 2, 1} ;
  float tf [] = {2.5, 3.2, 1.8} ;
  char tc[] = { 'a', 'e', 'i', 'o', 'u' } ;
  cout << somme (ti, 4) << "\n"  ;
  cout << somme (tf, 3) << "\n"  ;
  cout << somme (tc, 5) << "\n"  ;
}
```

Remarques

1. tel qu'il a été conçu, le patron `somme` ne peut être appliqué qu'à un type `T` pour lequel :

– l'opération d'addition a un sens ; cela signifie donc qu'il ne peut pas s'agir d'un type pointeur ; il peut s'agir d'un type `classe`, à condition que cette dernière ait surdéfini l'opérateur d'addition ;

– la déclaration `T som` est correcte ; cela signifie que si `T` est un type classe, il est nécessaire qu'il dispose d'un constructeur sans argument ;

– l'affectation `som = 0` est correcte ; cela signifie que si `T` est un type classe, il est nécessaire qu'il ait surdéfini l'affectation.

À ce propos, notons qu'il est possible d'initialiser `som` lors de sa déclaration, en procédant ainsi :

```
T som (0) ;
```

Cela est équivalent à `T som = 0` si `T` est un type prédéfini. En revanche, si `T` est de type classe, cela provoque l'appel d'un constructeur à 1 argument de `T`, en lui transmettant la valeur 0 ; le problème relatif à l'affectation `som = 0` ne se pose plus alors.

2. L'exécution de l'exemple proposé fourni des résultats peu satisfaisants dans le cas où l'on applique `somme` à un tableau de caractères, compte tenu de la capacité limitée de ce type. On pourrait améliorer la situation en « spécialisant » notre patron pour les tableaux de caractères (en prévoyant, par exemple, une valeur de retour de type `int`).

Exercice 126

Énoncé

Soient les définitions suivantes de patrons de fonctions :

```
template <class T, class U> void fct (T a, U b)   { ... } // patron I
template <class T, class U> void fct (T * a, U b) { ... } // patron II
template <class T>          void fct (T, T, T)    { ... } // patron III
void fct (int a, float b) { .....}                        // fonction IV
```

Avec ces déclarations :

```
int n, p, q ;
float x, y ;
double z ;
```

Quels sont les appels corrects et, dans ce cas, quels sont les patrons utilisés et les prototypes des fonctions instanciées ?

```
fct (n, p) ;     // appel I
fct (x, y ) ;    // appel II
fct (n, x) ;     // appel III
fct (n, z) ;     // appel IV
fct (&n, p) ;    // appel V
fct (&n, x) ;    // appel VI
fct (&n, &p, &q) // appel VII
```

Solution

Ici, on fait appel à la fois à une surdéfinition (patrons I, II et III) et à une spécialisation de patron (fonction IV).

```
I)      patron I       void fct (int, int) ;
II)     patron I       void fct (float, float) ;
III)    fonction IV    void fct (int, float) ;
IV)     patron I       void fct (int, double) ;
V)      erreur : ambigüité entre fct (T, U) et fct (T*, U)
VI)     erreur : ambigüité entre fct (T, U) et fct (T*, U)
VII)    patron III     void fct (int *, int *, int *) ;
```

Remarque

Le patron II ne peut jamais être utilisé. En effet, chaque fois qu'il pourrait l'être, le patron I peut l'être également, de sorte qu'il y a ambiguïté. Le patron II est donc, ici, parfaitement inutile.

Notez que si nous avions défini simultanément les deux patrons :

```
template <class T, class U> void fct (T a, U b) ;
template <class T>          void fct (T a, T b)
```

le même phénomène d'ambiguïté (entre ces deux patrons) serait apparu lors d'appels tels que `fct (n,p)` ou `fct(x,y)`.

Rappelons que l'ambiguïté n'est détectée que lorsque le compilateur doit instancier une fonction et non simplement au vu des définitions de patrons elles-mêmes : ces dernières restent donc acceptées tant que l'ambiguïté n'est pas mise en évidence par un appel la révélant.
