Chapitre 4

Le polymorphisme

Il arrive souvent que nous réécrivions presque exactement les mêmes programmes pour des données de types différents, par exemple des programmes de tri. Le **polymorphisme** consiste à n'écrire un tel programme qu'une seule fois, en laissant le type à définir plus tard. Ceci concerne le polymorphisme des sous-programmes, donc le polymorphisme des fonctions, ce qui conduit à la notion de **fonction générique**.

Le *polymorphisme* est une notion importante qui n'est pas propre aux langages orientés objets. Cependant cette notion n'est pas implémentée dans les premiers langages impératifs.

D'autre part nous avons déjà vu des structures de données abstraites, essentiellement les listes chaînées. Là encore nous avons pratiquement à réécrire la même chose pour manipuler une liste chaînée d'entiers ou une liste chaînée d'un autre type. Les structures de données abstraites correspondent aux classes du C++. On peut faire un pas de plus dans l'abstraction en considérant des **patrons de classes**, dans lesquels les types seront explicités plus tard.

4.1 Les fonctions génériques

Nous venons de voir l'intérêt de la notion de fonction générique. Nous allons voir comment la mettre en place en C++.

4.1.1 Définition des fonctions génériques

Introduction.- Le problème le plus caractéristique pour lequel nous avons certainement besoin de la généricité est celui du tri. Mais nous allons considérer un problème beaucoup plus simple pour voir comment mettre en place la généricité.

Un exemple.- Nous avons souvent besoin d'échanger les valeurs de deux variables, ce qui conduit à la fonction suivante par exemple pour l'échange de deux variables entières :

```
void echange(int &a, int &b)
   {
   int c;
   c = a;
   a = b;
   b = c;
   }
```

On doit réécrire un programme tout à fait analogue pour l'échange de deux réels ou de deux caractères. On ressent donc le besoin d'une fonction générique.

En C++ on peut écrire le **patron de fonction** suivant :

```
template <class T> void echange(T &a,T &b)
   {
   T c;
   c = a;
   a = b;
   b = c;
   }
```

Commentaires.- 1^o) Un *patron de fonction* s'introduit par le mot clé **template**.

- 2^o) On indique ensuite les types indéfinis par des identificateurs placés entre les signes '<' et '>', séparés par des virgules, chacun précédé du mot clé **class**. Ceci indique, pour notre exemple, que `T` est une **variable de type**, ou une **variable de classe** puisque on parle plutôt de classe que de type en C++.

- 3^o) Ces identificateurs sont alors manipulés comme des types de façon habituelle.

4.1.2 Utilisation des fonctions génériques

Introduction.- Le langage C (et donc le langage C++) étant un langage fortement typé, il est inutile d'indiquer *explicitement* le type lors de l'utilisation d'une fonction générique ; on le déduit à partir (du type) des expressions placées comme argument.

Ceci n'est pas le cas de tous les langages dans lesquels on peut utiliser le polymorphisme.

Continuation de l'exemple.- Écrivons un programme complet (sans intérêt) pour montrer le fonctionnement de la fonction générique ci-dessus :

```
// echange.cpp

#include <iostream.h>

template <class T> void echange(T &a,T &b)
   {
   T c;
   c = a;
   a = b;
   b = c;
   }

void main(void)
   {
   int m, n;
   float x, y;
   char c, d;
   cout << "Entrez deux entiers m = ";
   cin >> m;
   cout << "et n = ";
   cin >> n;
   echange(m, n);
   cout << "Ces deux entiers, apres echange, sont "
        << m << " et " << n;
   cout << "\nEntrez deux reels x = ";
   cin >> x;
   cout << "et y = ";
   cin >> y;
   echange(x, y);
   cout << "Ces deux reels, apres echange, sont "
        << x << " et " << y;
   cout << "\nEntrez deux caracteres c = ";
   cin >> c;
   cout << "et d = ";
   cin >> d;
   echange(c, d);
   cout << "Ces deux caracteres, apres echange, sont "
        << c << " et " << d << ". \n";
   // echange(m, y);
   }
```

Exercice.- 1^o) Faites tourner ce programme pour voir ce qu'il donne.

- 2^o) Enlever la mise en commentaire de l'avant-dernière ligne et compiler le programme ainsi obtenu pour voir ce qui arrive.

4.1.3 Fonctions génériques à plusieurs types indéfinis

Introduction.- Lorsqu'il y a plusieurs types indéfinis on commence par en donner la liste, séparés par des virgules.

Exercice.- *Écrire un programme manipulant des tableaux de réels. On utilisera une variable de type* `reel` *(instantiée en* `float`*,* `double` *ou* `long double` *suivant le cas) et une variable de type* `entier` *(instantiée en l'un des types entiers suivant le cas).*

La déclaration des patrons de fonction sera donc de la forme :

```
template <class entier, class reel>
```

4.1.4 Utilisation d'un type d'objet

Introduction.- Les classes étant des types comme les autres, la variable de type peut désigner une classe. Il faut, bien sûr, s'assurer que les opérateurs utilisés dans la définition de la fonction générique sont valables pour cette classe. Cette remarque est d'ailleurs également valable pour les types prédéfinis.

Exemple.- La fonction `echange()` ci-dessus peut s'utiliser pour la classe `point` :

```
x_point.cpp

#include <iostream.h>

// declaration du type point

class point
   {
   // declaration des attributs
   float x, y;
   // declaration des methodes
   public:
   void initialise(float, float);
   void deplace(float, float);
   void affiche();
   };

// definition des fonctions membres

void point::initialise(float abs, float ord)
   {
   x = abs;
   y = ord;
   }

void point::deplace(float dx, float dy)
   {
   x = x + dx;
   y = y + dy;
   }
```

```
void point::affiche()
   {
   cout << "Le point est en (" << x << "," << y <<")\n;
   }

// Definition de la fonction generique

template <class T> void echange(T &a,T &b)
   {
   T c;
   c = a;
   a = b;
   b = c;
   }

// Programme principal

void main(void)
   {
   point A, B;
   A.initialise(2.34, 5.1);
   B.initialise(3.5, -6.2);
   echange(A, B);
   B.affiche();
   }
```

4.2 Les classes génériques

Nous avons vu qu'une classe correspond à une structure de données abstraite. Nous avons étudié quelques structures de données abstraites telles que les chaînes de caractères et les listes. Intéressons-nous, par exemple, aux listes. Nous avons les listes d'entiers, les listes de réels, les listes de caractères... Jusqu'à maintenant nous avons besoin de créer une structure de liste par type d'éléments de la liste alors que l'implémentation est essentiellement la même. Ceci a conduit à la notion de **classe générique** qui permet d'implémenter en une seule fois les structures de données qui sont moralement identiques.

4.2.1 Un exemple

Introduction.- Bien que la notion de classe générique soit intéressante pour des classes génériques telles que les listes, commençons par un exemple beaucoup plus simple.

Le problème.- Nous avons défini la classe `point` avec des coordonnées entières. On peut vouloir des coordonnées réelles ; de plus, même pour les coordonnées entières ou réelles, il existe plusieurs types prédéfinis en C++. On a donc intérêt à définir un **patron de classe** (le nom pour *classe générique* en C++, **class template** en anglais) qui corresponde à tous ces cas.

Un programme.- Le programme suivant définit un tel patron de classe `point` et l'applique à un certain nombre d'exemples :

```
// g_point.cpp

#include <iostream.h>
// declaration du patron de classe point
template <class T> class point
   {
   // declaration des attributs
   T x, y;
   // declaration des methodes
   public:
   void initialise(T, T);
   void deplace(T dx,T dy)
      {
      x = x + dx;
      y = y + dy;
      }
   void affiche(void);
   };
// definition des fonctions membres non en ligne
template <class T> void point<T>::initialise(T abs,T ord)
   {
   x = abs;
   y = ord;
   }
template <class T> void point<T>::affiche(void)
   {
   cout << "Le point est en (" << x << "," << y <<")\n";
```

```
   }
// Programme principal
void main(void)
   {
   point<float> A;
   point<int> B;
   A.initialise(2.34, 5.1);
   A.deplace(1.03, 3.3);
   A.affiche();
   B.initialise(1, 2);
   B.affiche();
   }
```

4.2.2 Commentaires

Déclaration d'un patron de classe.- Comme dans le cas d'un patron de fonctions, la mention `template <class T>` précise que l'on affaire à un patron (*template*) dans lequel apparaît la variable de type `T`. On a une seule variable de types ici mais on peut en avoir plusieurs dans le cas général, séparées par des virgules.

Définition d'une fonction membre.- Les fonctions membres sont des patrons de fonctions, il faut donc le dire. L'en-tête sera donc un peu plus complexe que celui d'une fonction ordinaire. Par exemple, au lieu de :

```
void point::initialise(T abs, T ord)
```

on aura :

```
template <class T> void point<T>::initialise(T abs, T ord)
```

On remarquera que la liste des variables de types apparaît deux fois : une première fois après l'indication qu'on a affaire à un patron de fonctions, une seconde fois après le nom de la fonction. Ceci ne semble pas nécessaire *a priori*. L'inventeur du C++, STROUTRUP, se contente de signaler cette redondance sans l'expliquer.

Définition d'une fonction membre en ligne.- Pour une fonction membre définie en ligne, ce qui est le cas de notre fonction `deplace()`, la déclaration est plus simple.

Utilisation d'un patron de classes.- Pour déclarer une variable d'une classe de ce patron il faut indiquer à la fois le nom du patron et, placées entre les signes '`<`' et '`>`' et séparés par des virgules, les assignations de types. On a, par exemple :

```
point<float> A ;
point<int> B ;
```