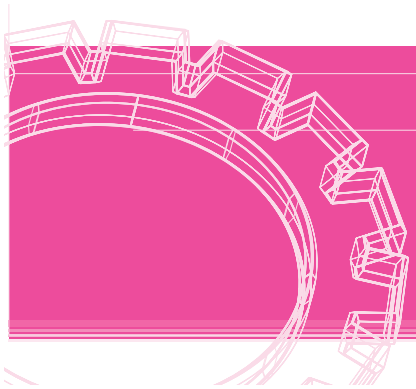
Chapitre 15

Les fonctions virtuelles



Rappels

Typage statique des objets (ou ligature dynamique des fonctions)

Les règles de compatibilité entre une classe de base et une classe dérivée permettent d'affecter à un pointeur sur une classe de base la valeur d'un pointeur sur une classe dérivée. Toutefois, par défaut, le type des objets pointés est défini lors de la compilation. Par exemple, avec :

```
class A
{  .....
   public :
     void fct (...) ;
     .....
} ;
```

```
class B : public A
{  .....
   public :
     void fct (...) ;
     .....
} ;
```

```
A * pta ;
B * ptb ;
```

une affectation telle que `pta = ptb` est autorisée. Néanmoins, quel que soit le contenu de `pta` (autrement dit, quel que soit l'objet pointé par `pta`), `pta->fct(...)` **appelle toujours la fonction** `fct` **de la classe** `A`.

Les fonctions virtuelles

L'emploi des fonctions virtuelles permet d'éviter les problèmes inhérents au typage statique. Lorsqu'une fonction est déclarée virtuelle (mot-clé `virtual`) dans une classe, les appels à une telle fonction ou à n'importe laquelle de ses redéfinitions dans des classes dérivées sont « résolus » au moment de l'exécution, selon le type de l'objet concerné. On parle de typage dynamique des objets (ou de ligature dynamique des fonctions). Par exemple, avec :

```
class A                          class B : public A
{  .....                         {  .....
   public :                         public :
     virtual void fct (...) ;         void fct (...) ;
     .....                            .....
};                                }  ;

A * pta ;
```

l'instruction `pta->fct (...)` appellera la fonction `fct` de la classe correspondant réellement au type de l'objet pointé par `pta`.

N.B. Il peut y avoir ligature dynamique même en dehors de l'utilisation de pointeurs (voyez, par exemple, l'exercice 65).

Règles

- Le mot-clé `virtual` ne s'emploie qu'une fois pour une fonction donnée ; plus précisément, il ne doit pas accompagner les redéfinitions de cette fonction dans les classes dérivées.
- Une méthode déclarée virtuelle dans une classe de base peut ne pas être redéfinie dans ses classes dérivées.
- Une fonction virtuelle peut être surdéfinie (chaque fonction surdéfinie pouvant être ou ne pas être virtuelle).
- Un constructeur ne peut pas être virtuel, un destructeur peut l'être.
- Par sa nature même, le mécanisme de ligature dynamique est limité à une hiérarchie de classes ; souvent, pour qu'il puisse s'appliquer à toute une bibliothèque de classes, on sera amené à faire hériter toutes les classes de la bibliothèque d'une même classe de base.

Les fonctions virtuelles pures

Une « fonction virtuelle pure » se déclare avec une initialisation à zéro, comme dans :

```
virtual void affiche () = 0 ;
```

Lorsqu'une classe comporte au moins une fonction virtuelle pure, elle est considérée comme « abstraite », c'est-à-dire qu'il n'est plus possible de créer des objets de son type.

Une fonction déclarée virtuelle pure dans une classe de base peut ne pas être déclarée dans une classe dérivée et, dans ce cas, elle est à nouveau implicitement fonction virtuelle pure de cette classe dérivée (avant la version 3.0, une fonction virtuelle pure devait obligatoirement être redéfinie dans une classe dérivée ou déclarée à nouveau virtuelle pure).

Exercice 115

Énoncé

Quels résultats produira ce programme :

```
#include <iostream>
using namespace std ;
class point
{ protected :       // pour que x et y soient accessibles à pointcol
   int x, y ;
  public :
   point (int abs=0, int ord=0) { x=abs ; y=ord ; }
   virtual void affiche ()
     { cout << "Je suis un point \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y << "\n" ;
     }
} ;

class pointcol : public point
{  short couleur ;
  public :
   pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
     { couleur = cl ;
     }
   void affiche ()
     { cout << "Je suis un point coloré \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y ;
       cout << "   et ma couleur est :    " << couleur << "\n" ;
     }
} ;

main()
{
   point p(3,5) ; point * adp = &p ;
   pointcol pc (8,6,2) ; pointcol * adpc = &pc ;
```

```
    adp->affiche () ; adpc->affiche () ;      // instructions 1
    cout << "-----------------\n" ;
    adp = adpc ;
    adp->affiche () ; adpc->affiche () ;      // instructions 2
}
```

Solution

Dans les instructions 1, `adp` (de type `point *`) pointe sur un objet de type `point`, tandis que `adpc` (de type `pointcol *`) pointe sur un objet de type `pointcol`. Il y a appel de la fonction `affiche`, respectivement de `point` et de `pointcol` ; l'existence du « typage dynamique » n'apparaît pas clairement puisque, même en son absence (c'est-à-dire si la fonction `affiche` n'avait pas été déclarée virtuelle), on aurait obtenu le même résultat.

En revanche, dans les instructions 2, `adp` (de type `point *`) pointe maintenant sur un objet de type `pointcol`. Grâce au typage dynamique, `adp->affiche()` appelle bien la fonction `affiche` du type `pointcol`.

Voici les résultats complets fournis par ce programme :

```
Je suis un point
   mes coordonnées sont : 3 5
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
-----------------
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
```

Exercice 116

Énoncé

Quels résultats produira ce programme :

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  virtual void identifie ()
    { cout << "Je suis un point \n" ;
    }
```

```
  void affiche ()
    { identifie () ;
      cout << "Mes coordonnées sont : " << x << " " << y << "\n" ;
    }
} ;
class pointcol : public point
{ short couleur ;
 public :
  pointcol (int abs=0, int ord=0, int cl=1 ) : point (abs, ord)
    { couleur = cl ; }
  void identifie ()
    { cout << "Je suis un point coloré de couleur : " << couleur << "\n" ;
    }
} ;
main()
{ point p(3,4) ;
  pointcol pc(5,9,5) ;
  p.affiche () ;
  pc.affiche () ;
  cout << "---------------\n" ;
  point * adp = &p ;
  pointcol * adpc = &pc ;
  adp->affiche () ; adpc->affiche () ;
  cout << "---------------\n" ;
  adp = adpc ;
  adp->affiche () ; adpc->affiche () ;
}
```

Solution

Dans la fonction `affiche` de `point`, l'appel de `identifie` fait l'objet d'une ligature dynamique (puisque cette dernière fonction a été déclarée virtuelle). Lorsqu'un objet de type `pointcol` appelle une fonction `affiche`, ce sera bien la fonction `affiche` de `point` qui sera appelée (puisque `affiche` n'est pas redéfinie dans `pointcol`). Mais cette dernière fera appel, dans ce cas, à la fonction `identifie` de `pointcol`. Bien entendu, lorsqu'un objet de type `point` appelle `affiche`, cette dernière fera toujours appel à la fonction `identifie` de `point` (le même résultat serait obtenu sans ligature dynamique).

Voici les résultats complets fournis par notre programme :

```
Je suis un point
Mes coordonnées sont : 3 4
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
---------------
```

```
Je suis un point
Mes coordonnées sont : 3 4
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
---------------
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
```

Exercice 117

Énoncé

On souhaite créer une classe nommée `ens_heter` permettant de manipuler des ensembles dont le type des éléments est non seulement inconnu de la classe, mais également susceptible de varier d'un élément à un autre. Pour que la chose soit possible, on imposera simplement la contrainte suivante : tous les types concernés devront dériver d'un même type de base nommé `base`. Le type `base` sera supposé connu au moment où l'on définit la classe `ens_heter`.

La classe `base` disposera au moins d'une fonction virtuelle pure nommée `affiche` ; cette fonction devra être redéfinie dans les classes dérivées pour afficher les caractéristiques de l'objet concerné.

La classe `ens_heter` disposera des fonctions membre suivantes :

- `ajoute` pour ajouter un nouvel élément à l'ensemble (elle devra s'assurer qu'il n'existe pas déjà)þ;
- `appartient` pour tester l'appartenance d'un élément à l'ensembleþ;
- `cardinal` qui fournira le nombre d'éléments de l'ensemble.

De plus, la classe devra être munie d'un « itérateur », c'est-à-dire d'un mécanisme permettant de parcourir les différents éléments de l'ensemble. On prévoira 3 fonctions :

- `init` pour initialiser le mécanisme d'itérationþ;
- `suivant` qui fournira en retour le prochain élément (objet d'un type dérivé de `base`)þ;
- `existe` pour préciser s'il existe encore un élément non examiné.

Enfin, une fonction nommée `liste` permettra d'afficher les caractéristiques de tous les éléments de l'ensemble (elle fera, bien sûr, appel aux fonctions `affiche` des différents objets concernés).

On réalisera ensuite un petit programme d'essai de la classe `ens_heter`, en créant un ensemble comportant des objets de type `point` (deux coordonnées entières) et `complexe` (une partie réelle et une partie imaginaire, toutes deux de type `float`). Naturellement, `point` et `complexe` devront dériver de `base`.

On ne se préoccupera pas des éventuels problèmes posés par l'affectation ou la transmission par valeur d'objets du type `ens_heter`.

N.B. Pour ne pas trop compliquer le programme, on fondera l'égalité de deux éléments d'un ensemble sur l'égalité de leurs adresses et non pas de leurs valeurs. Autrement dit, deux éléments de même type et de même valeur pourront appartenir à un même ensemble, à condition qu'il s'agisse bien de deux objets différents.

Solution

Manifestement, la classe `ens_heter` ne contiendra pas les objets correspondant aux éléments de l'ensemble, mais seulement des pointeurs sur ces différents éléments. À moins de fixer le nombre maximal d'éléments a priori, il est nécessaire de conserver ces pointeurs dans un emplacement alloué dynamiquement par le constructeur ; ce dernier comportera donc un argument permettant de préciser le nombre maximal d'éléments requis pour l'ensemble.

Outre l'adresse (`adel`) de ce tableau de pointeurs, on trouvera en membres donnée :

- le nombre maximal d'éléments (`nmax`) ;
- le nombre courant d'éléments (`nelem`) ;
- un entier (`courant`) qui servira au mécanisme d'itération (il désignera une adresse du tableau de pointeurs).

En ce qui concerne les fonctions `ajoute` et `appartient`, elles devront manifestement recevoir en argument un objet d'un type dérivé de `base`. Puisqu'on ne fait aucune hypothèse a priori sur la nature de tels objets, il est préférable d'éviter une transmission d'argument par valeur. Par souci de simplicité, nous choisirons une transmission par référence.

Les mêmes remarques s'appliquent à la valeur de retour de la fonction `suivant`.

Voici, en définitive, la déclaration de notre classe `ens_heter` :

```
        /*         fichier ensheter.H         */
        /* déclaration de la classe ens_heter */
class base ;
class ens_heter
{  int nmax ;       // nombre maximal d'éléments
   int nelem ;      // nombre courant d'éléments
   base * * adel ;  // adresse zone de pointeurs sur les objets éléments
   int courant ;    // numéro d'élément courant (pour l'itération)
 public :
   ens_heter (int=20) ;       // constructeur
   ~ens_heter () ;            // destructeur
```

```
   void ajoute (base &) ;     // ajout d'un élément
   int appartient (base &) ;  // appartenance d'un élément
   int cardinal () ;          // cardinal de l'ensemble
   void init () ;             // initialisation itération
   base & suivant () ;        // passage élément suivant
   int existe () ;            //
   void liste () ;            // affiche les "valeurs" des différents éléments
} ;
```

La classe `base` (déclaration et définition) découle directement de l'énoncé :

```
class base
{ public :
    virtual void affiche () = 0 ;
} ;
```

Voici la définition des fonctions membre de `ens_heter` (notez que leur compilation nécessite la déclaration (définition) précédente de la classe `base` (et non pas seulement une simple indication de la forme `class base`) :

```
   /* définition de la classe ens_heter */
#include "ensheter.h"
ens_heter::ens_heter (int dim)
{  nmax = dim ;
   adel = new base * [dim] ;
   nelem = 0 ;
   courant = 0 ;       // précaution
}
ens_heter::~ens_heter ()
{  delete adel ;
}
void ens_heter::ajoute (base & obj)
{  if ((nelem < nmax) && (!appartient (obj))) adel [nelem++] = & obj ;
}
int ens_heter::appartient (base & obj)
{  int trouve = 0 ;
   init () ;
   while ( existe () && !trouve)  if ( &suivant() == & obj) trouve=1 ;
   return trouve ;
}
int ens_heter::cardinal ()
{   return nelem ;
}

void ens_heter::init ()
{  courant = 0 ;
}
```

```
base & ens_heter::suivant ()
{  if (courant<nelem) return (* adel [courant++]) ;
      // en pratique, il faudrait renvoyer un objet "bidon" si fin
      // ensemble atteinte
}

int ens_heter::existe ()
{   return (courant<nelem) ;
}

void ens_heter::liste ()
{   init () ;
    while ( existe () )
       suivant () . affiche () ;
}
```

Voici un petit programme qui définit deux classes `point` et `complexe`, dérivées de `base` et qui crée un petit ensemble hétérogène (sa compilation nécessite la déclaration de la classe `base`). À la suite figure le résultat de son exécution :

```
#include "ens_heter.h"
#include "base.h"
#include <iostream>
using namespace std ;
class point : public base
{   int x, y ;
  public :
    point (int abs=0, int ord=0)
      { x = abs ; y = ord ;
      }
    void affiche ()
      { cout << "Point de coordonnées : " << x << " " << y << "\n" ;
      }
} ;

class complexe : public base
{   float re, im ;
  public :
    complexe (float reel=0.0, float imag=0.0)
      { re = reel ; im = imag ;
      }
    void affiche ()
      {  cout << "Complexe - partie réelle : " << re
              << ", partie imaginaire : " << im << "\n" ;
      }
} ;
```

```
   /* utilisation de la classe ens_heter */
main()
{   point p (1,3) ;
    complexe z (0.5, 3) ;
    ens_heter e ;
    cout << "cardinal de e : " << e.cardinal() << "\n" ;
    cout << "contenu de e  \n" ;
    e.liste () ;
    e.ajoute (p) ;
    cout << "cardinal de e : " << e.cardinal() << "\n" ;
    cout << "contenu de e  \n" ;
    e.liste () ;
    e.ajoute (z) ;
    cout << "cardinal de e : " << e.cardinal() << "\n" ;
    cout << "contenu de e  \n" ;
    e.liste () ;
    e.init () ; int n=0 ;
    while (e.existe()) { e.suivant() ;
                         n++ ;
                       }
    cout << "avec l'itérateur, on trouve : " << n << " éléments\n" ;
}
```

```
cardinal de e : 0
contenu de e
cardinal de e : 1
contenu de e
Point de coordonnées : 1 3
cardinal de e : 2
contenu de e
Point de coordonnées : 1 3
Complexe - partie réelle : 0.5, partie imaginaire : 3
avec l'itérateur, on trouve : 2 éléments
```

Remarque

Si l'on cherchait à résoudre les problèmes posés par l'affectation et la transmission par valeur d'objets de type `ens_heter`, on serait amené à effectuer des « copies complètes » (on dit souvent « profondes ») de l'objet, c'est-à-dire tenant compte non seulement de ses membres, mais aussi de ses parties dynamiques.

Cela conduirait effectivement à recopier la partie dynamique de l'ensemble, c'est-à-dire le tableau de pointeurs d'adresse `base *`. Mais que faudrait-il faire pour les éléments eux-mêmes (pointés par les différentes valeurs du tableau) ? Même si l'on admet qu'il faut également les dupliquer, il n'est pas possible de le faire sans connaître la « structure » des objets concernés.

D'une manière générale, vous voyez que cet exemple, par les questions qu'il soulève, plaide largement en faveur de la constitution de bibliothèques d'objets, dans lesquelles sont définies, a priori, un certain nombre de règles communes. Parmi ces règles, on pourrait notamment imposer à chaque objet de posséder une fonction (de nom unique) en assurant la copie profonde ; naturellement, pour pouvoir l'utiliser correctement, il faudrait que la liga-ture dynamique soit possible, c'est-à-dire que tous les objets concernés dérivent d'un même objet de base.