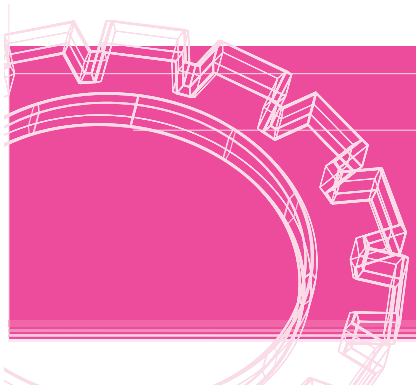
Chapitre 14

L'héritage multiple



Rappels

Depuis la version 2.0, C++ autorise l'héritage multiple : une classe peut hériter de plusieurs autres classes, comme dans cet exemple où la classe `pointcol` hérite simultanément des classes `point` et `coul` :

```
class pointcol : public point, public coul // chaque dérivation, ici publique
                                           // pourrait être privée ou protégée
    {      // définition des membres supplémentaires (données ou fonctions)
           // ou redéfinition de membres existants déjà dans point ou coul
    } ;
```

Chacune des dérivations peut être publique, privée ou protégée. Les modalités d'accès aux membres de chacune des classes de base restent les mêmes que dans le cas d'une dérivation « simple ». L'opérateur de résolution de portée (`::`) peut être utilisé :

- soit lorsque l'on veut accéder à un membre d'une des classes de base, alors qu'il est redéfini dans la classe dérivée,
- soit lorsque deux classes de base possèdent un membre de même nom et qu'il faut alors préciser celui qui nous intéresse.

Appel des constructeurs et des destructeurs

La création d'un objet entraîne l'appel du constructeur de chacune des classes de base, dans l'ordre où ces constructeurs sont mentionnés dans la déclaration de la classe dérivée (ici, `point` puis `coul` puisque nous avons écrit `class pointcol : public point, public coul`). Les destructeurs sont appelés dans l'ordre inverse.

Le constructeur de la classe dérivée peut mentionner, dans son en-tête, des informations à retransmettre à chacun des constructeurs des classes de base (ce sera généralement indispensable, sauf si une classe de base possède un constructeur sans argument ou si elle ne dispose pas du tout de constructeur). En voici un exemple :

```
pointcoul ( .....) : point (.....), coul (.....)
              |              |             |
              |              |             |
          arguments       arguments     arguments
          pointcoul         point          coul
```

Classes virtuelles

Par le biais de dérivations successives, il est tout à fait possible qu'une classe hérite deux fois d'une même classe. En voici un exemple dans lequel `D` hérite deux fois de `A` :

```
class B : public A
   { ..... } ;
class C : public A
   { ..... } ;
class D : public B, public C
   { ..... } ;
```

Dans ce cas, les membres donnée de la classe en question (`A` dans notre exemple) apparaissent **deux fois** dans la classe dérivée de deuxième niveau (ici `D`). Naturellement, il est nécessaire de faire appel à l'opérateur de résolution de portée (`::`) pour lever l'ambiguïté. Si l'on souhaite que de tels membres n'apparaissent qu'une seule fois dans la classe dérivée de deuxième niveau, il faut, dans les déclarations des dérivées de premier niveau (ici `B` et `C`) déclarer avec l'attribut `virtual` la classe dont on veut éviter la duplication (ici `A`).

Voici comment on procéderait dans l'exemple précédent (le mot `virtual` peut être indifféremment placé avant ou après le mot `public` ou le mot `private`) :

```
class B : public virtual A
   { ..... } ;
class C : public virtual A
   { ..... } ;
class D : public B, public C
   { ..... } ;
```

Lorsqu'on a ainsi déclaré une classe virtuelle, il est nécessaire que les constructeurs d'éventuelles classes dérivées puissent préciser les informations à transmettre au constructeur de cette classe virtuelle (dans le cas usuel où l'on autorise la duplication, ce problème ne se pose plus ; en effet, chaque constructeur transmet les informations aux classes ascendantes dont les constructeurs transmettent, à leur tour, les informations aux constructeurs de chacune des occurrences de la classe en question – ces informations pouvant éventuellement être différentes). Dans ce cas, on le précise dans l'en-tête du constructeur de la classe dérivée, en plus des arguments destinés aux constructeurs des classes du niveau immédiatement supérieur, comme dans cet exemple :

```
D ( .....) : B (.....), C ( ..... ), A ( ..... )
      |           |           |           |
      |           |           |           |
  arguments    arguments   arguments    arguments
    de D        pour B      pour C       pour A
```

De plus, dans ce cas, les constructeurs des classes `B` et `C` (qui ont déclaré que `A` était « virtuelle ») n'auront plus à spécifier d'informations pour un constructeur de `A`.

Enfin, le constructeur d'une classe virtuelle est toujours appelé avant les autres.

Exercice 109

Énoncé

Quels seront les résultats fournis par ce programme :

```
#include <iostream>
using namespace std ;
class A
{   int n ;
    float x ;
  public :
    A (int p = 2)
    { n = p ; x = 1 ;
      cout << "** construction objet A : " << n << " " << x << "\n" ;
    }
} ;
class B
{   int n ;
    float y ;
  public :
    B (float v = 0.0)
    { n = 1 ;  y = v ;
      cout << "** construction objet B : " << n << " " << y << "\n" ;
    }
} ;
```

```
class C : public B, public A
{   int n ;
    int p ;
  public :
    C (int n1=1, int n2=2, int n3=3, float v=0.0) : A (n1), B(v)
    { n = n3 ; p = n1+n2 ;
     cout << "** construction objet C : " << n << " " << p <<"\n" ;
    }
} ;

main()
{  C c1 ;
   C c2 (10, 11, 12, 5.0) ;
}
```

Solution

L'objet `c1` est créé par appels successifs des constructeurs de `B`, puis de `A` (ordre imposé par la déclaration de la classe `C`, et non par l'en-tête du constructeur de `C` !). Le jeu de la transmission des arguments et des arguments par défaut conduit au résultat suivant :

```
** construction objet B : 1 0
** construction objet A : 1 1
** construction objet C : 3 3
** construction objet B : 1 5
** construction objet A : 10 1
** construction objet C : 12 21
```

Exercice 110

Énoncé

Même question que précédemment, en remplaçant simplement l'en-tête du constructeur de `C` par :

```
C (int n1=1, int n2=2, int n3=3, float v=0.0) :  B(v)
```

Solution

Ici, comme le constructeur de `C` n'a prévu aucun argument pour un éventuel constructeur de `A`, il y aura appel d'un constructeur sans argument, c'est-à-dire, en fait, appel du constructeur de `A`, avec toutes les valeurs prévues par défaut. Voici le résultat obtenu :

```
** construction objet B : 1 0
** construction objet A : 2 1
** construction objet C : 3 3
```

```
** construction objet B : 1 5
** construction objet A : 2 1
** construction objet C : 12 21
```

Exercice 111

Enoncé

Même question que dans l'exercice 58, en supposant que l'en-tête du constructeur de C est la suivante :

```
C (int n1=1, int n2=2, int n3=3, float v=0.0)
```

Solution

Cette fois, la construction d'un objet de type C entraînera l'appel d'un constructeur sans argument, à la fois pour B et pour A. Voici les résultats obtenus :

```
** construction objet B : 1 0
** construction objet A : 2 1
** construction objet C : 3 3
** construction objet B : 1 0
** construction objet A : 2 1
** construction objet C : 12 21
```

Exercice 112

Énoncé

Quels seront les résultats fournis par ce programme :

```
#include <iostream>
using namespace std ;
class A
{   int na ;
  public :
    A (int nn=1)
      { na = nn ; cout << "$$construction objet A " << na << "\n" ;
      }
} ;
```

```
class B : public A
{   float xb ;
  public :
    B (float xx=0.0)
      { xb = xx ; cout << "$$construction objet B " << xb << "\n" ;
      }
} ;
class C : public A
{   int nc ;
  public :
    C (int nn= 2) : A (2*nn+1)
      { nc = nn ;
        cout << "$$construction objet C " << nc << "\n" ;
      }
} ;
class D : public B, public C
{   int nd ;
  public :
    D (int n1, int n2, float x) : C (n1), B (x)
     { nd = n2 ;
       cout << "$$construction objet D " << nd << "\n" ;
     }
} ;

main()
{  D d (10, 20, 5.0) ;
}
```

Solution

La construction d'un objet de type `D` entraînera l'appel des constructeurs de `B` et de `C`, lesquels, avant leur exécution, appelleront chacun un constructeur de `A` : dans le cas de `B`, il y aura appel d'un constructeur sans argument (puisque l'en-tête de `B` ne mentionne pas de liste d'arguments pour `A`) ; en revanche, dans le cas de `C`, il s'agira (plus classiquement) d'un constructeur à un argument, comme mentionné dans l'en-tête de `C`.

Notez bien qu'il y a création de deux objets de type `A`. Voici les résultats obtenus :

```
$$construction objet A 1
$$construction objet B 5
$$construction objet A 21
$$construction objet C 10
$$construction objet D 20
```

Exercice 113

Énoncé

Transformer le programme précédent, de manière qu'un objet de type `D` ne contienne qu'une seule fois les membres de `A` (qui se réduisent en fait à l'entier na). On s'arrangera pour que le constructeur de `A` soit appelé avec la valeur `2*nn+1`, dans laquelle `nn` désigne l'argument du constructeur de `C`.

Solution

Dans la déclaration des classes `B` et `C`, il faut indiquer que la classe `A` est « virtuelle », de façon qu'elle ne soit incluse qu'une fois dans d'éventuelles descendantes de ces classes. D'autre part, le constructeur de `D` doit prévoir, outre les arguments pour les constructeurs de `B` et de `C`, ceux destinés à un constructeur de `A`.

En résumé, la déclaration de `A` reste inchangée, celle de `B` est transformée en :

```
class B : public virtual A
 {  // le reste est inchangé
 }
```

Celle de `C` est transformée de façon analogue :

```
class C : public virtual A
 {  // le reste est inchangé

 }
```

Enfin, dans `D`, l'en-tête du constructeur devient :

```
D (int n1, int n2, float x) : C (n1), B (x), A (2*n1+1)
```

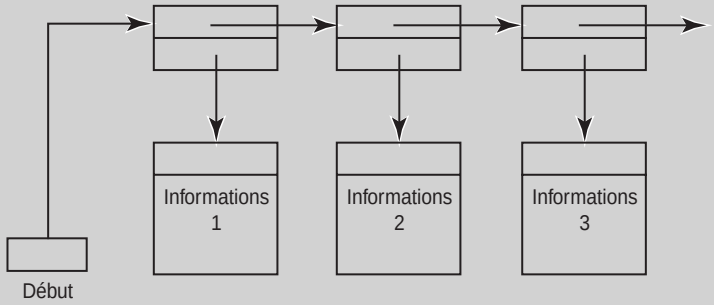
À titre indicatif, voici les résultats que fournirait le programme précédent ainsi transformé :

```
$$construction objet A 21
$$construction objet B 5
$$construction objet C 10
$$construction objet D 20
```

Exercice 114

Énoncé

On souhaite créer une classe `liste` permettant de manipuler des « listes chaînées » dans lesquelles la nature de l'information associée à chaque « nœud » de la liste n'est pas connue (par la classe). Une telle liste correspondra au schéma suivant :



La déclaration de la classe `liste` se présentera ainsi :

```
struct element                    // structure d'un élément de liste
{ element * suivant ;             // pointeur sur l'élément suivant
  void * contenu ;                // pointeur sur un objet quelconque
} ;

class liste
{ element * debut ;               // pointeur sur premier élément
         // autres membres données éventuels
 public :
  liste () ;                      // constructeur
  ~liste () ;                     // destructeur
  void ajoute (void *) ;          // ajoute un élément en début de
liste
  void * premier () ;             // positionne sur premier élément
  void * prochain () ;            // positionne sur prochain élément
  int fini () ;
} ;
```

La fonction `ajoute` devra ajouter, en début de liste, un élément pointant sur l'information dont l'adresse est fournie en argument (`void *`). Pour « explorer » la liste, on a prévu trois fonctions :

- `premier`, qui fournira l'adresse de l'information associée au premier nœud de la liste et qui, en même temps, préparera le processus de parcours de la liste ;

- `prochain`, qui fournira l'adresse de l'information associée au « prochain nœud » ; des appels successifs de `prochain` devront permettre de parcourir la liste (sans qu'il soit nécessaire d'appeler une autre fonction) ;
- `fini`, qui permettra de savoir si la fin de liste est atteinte ou non.

1. Compléter la déclaration précédente de la classe `liste` et en fournir la définition de manière qu'elle fonctionne comme demandé.

2. Soit la classe `point` suivante :

```
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  void affiche () { cout << "Coordonnées : " << x << " " << y << "\n" ; }
} ;
```

Créer une classe `liste_points`, dérivée à la fois de `liste` et de `point`, pour qu'elle puisse permettre de manipuler des listes chaînées de points, c'est-à-dire des listes comparables à celles présentées ci-dessus, et dans lesquelles l'information associée est de type `point`. On devra pouvoir, notamment :

- ajouter un `point` en début d'une telle liste ;
- disposer d'une fonction membre `affiche` affichant les informations associées à chacun des points de la liste de points.

3. Écrire un petit programme d'essai.

Solutions

1. Manifestement, les fonctions `premier` et `prochain` nécessitent un « pointeur sur un élément courant ». Il sera membre donnée de la classe `liste`. Nous conviendrons (classiquement) que la fin de liste est matérialisée par un nœud comportant un pointeur « nul ». La classe `liste` devra disposer d'un constructeur dont le rôle se limitera à l'initialiser à une « liste vide », ce qui s'obtiendra simplement en plaçant un pointeur nul comme adresse de début de liste (cette façon de procéder simplifie grandement l'algorithme d'ajout d'un élément en début de liste, puisqu'elle évite d'avoir à distinguer des autres le cas de la liste vide).

Comme un objet de type `liste` est amené à créer différents emplacements dynamiques, il est nécessaire de prévoir la libération de ces emplacements lorsque l'objet est détruit. Il faudra donc prévoir un destructeur, chargé de détruire les différents nœuds de la liste. À ce propos, notez qu'il n'est pas possible ici de demander au destructeur de détruire également les informations associées ; en effet, ce n'est pas l'objet de type `liste` qui a alloué ces emplacements : ils sont sous la responsabilité de l'utilisateur de la classe `liste`.

Voici ce que pourrait être notre classe `liste` complète :

```
#include <stdlib.h>               // pour NULL
struct element                    // structure d'un élément de liste
{ element * suivant ;             // pointeur sur l'élément suivant
  void * contenu ;                // pointeur sur un objet quelconque
} ;
class liste
{ element * debut ;               // pointeur sur premier élément
  element * courant ;             // pointeur sur élément courant
 public :
  liste ()                        // constructeur
    { debut = NULL ;
      courant = debut ;           //   par sécurité
    }
  ~liste () ;                     // destructeur
  void ajoute (void *) ;          // ajoute un élément en début de liste
  void * premier ()               // positionne sur premier élément
  { courant = debut ;
    if (courant != NULL) return (courant->contenu) ;
                  else return NULL ;
  }
  void * prochain ()              // positionne sur prochain élément
  { if (courant != NULL)
  { courant = courant->suivant ;
       if (courant != NULL) return (courant->contenu) ;
  }
    return NULL ;
  }
  int fini () { return (courant == NULL) ; }
} ;
liste::~liste ()
{ element * suiv ;
  courant = debut ;
  while (courant != NULL )
   { suiv = courant->suivant ; delete courant ; courant = suiv ; }
}
void liste::ajoute (void * chose)
{ element * adel = new element ;
  adel->suivant = debut ;
  adel->contenu = chose ;
  debut = adel ;
}
```

2. Comme nous le demande l'énoncé, nous allons donc créer une classe `liste_points` par :

```
class liste_points : public liste, public point
```

Notez que cet héritage, apparemment naturel, conduit néanmoins à introduire, dans la classe `liste_points`, deux membres donnée (`x` et `y`) n'ayant aucun intérêt par la suite.

En revanche, la création des fonctions membre demandées devient extrêmement simple. En effet, la fonction d'insertion d'un point en début de liste peut être la fonction `ajoute` de la classe `liste` : nous n'aurons donc même pas besoin de la surdéfinir. En ce qui concerne la fonction d'affichage de tous les points de la liste (que nous nommerons également `affiche`), il lui suffira de faire appel :

– aux fonctions `premier`, `prochain` et `fini` de la classe `liste` pour le parcours de la liste de pointsþ;

– à la fonction `affiche` de la classe `point` pour l'affichage d'un point.

Nous aboutissons à ceci :

```
class liste_points : public liste, public point
{ public :
   liste_points () {}
   void affiche () ;
} ;
void liste_points::affiche ()
{ point * ptr = (point *) premier() ;
  while ( ! fini() ) {  ptr->affiche () ; ptr = (point *) prochain() ; }
}
```

3. Exemple de programme d'essai :

```
#include "listepts.h"
main()
{ liste_points l ;
  point a(2,3), b(5,9), c(0,8) ;
  l.ajoute (&a) ; l.affiche () ; cout << "---------\n" ;
  l.ajoute (&b) ; l.affiche () ; cout << "---------\n" ;
  l.ajoute (&c) ; l.affiche () ; cout << "---------\n" ;
}
```

À titre indicatif, voici les résultats fournis par ce programme :

```
Coordonnées : 2 3
---------
Coordonnées : 5 9
Coordonnées : 2 3
---------
Coordonnées : 0 8
Coordonnées : 5 9
Coordonnées : 2 3
---------
```