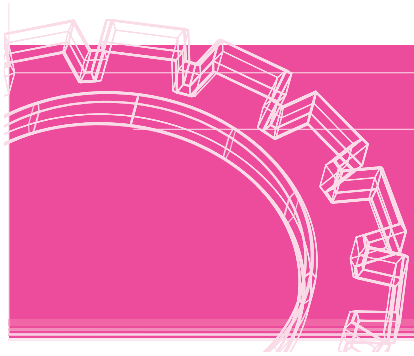
Chapitre 13

La technique de l'héritage



Rappels

Le concept d'héritage constitue l'un des fondements de la Programmation Orientée Objet. Il permet de définir une nouvelle classe *B* dite « dérivée », à partir d'une classe existante *A*, dite « de base » ; pour ce faire, on procède ainsi :

```
class B : public A   // ou :   private A    ou (depuis la version 3)  protected A
{    //  définition des membres supplémentaires (données ou fonctions)
     //  ou redéfinition de membres existants dans A (données ou fonctions)
} ;
```

Avec `public A`, on parle de « dérivation publique » ; avec `private A`, on parle de « dérivation privée » ; avec `protected A`, on parle de « dérivation protégée ».

Modalités d'accès à la classe de base

Les membres privés d'une classe de base ne sont jamais accessibles aux fonctions membre de sa classe dérivée.

Outre les « statuts » public ou privé (présentés au chapitre 3), il existe un statut « protégé ». Un membre protégé se comporte comme un membre privé pour un utilisateur quelconque de la classe ou de la classe dérivée, mais comme un membre public pour la classe dérivée.

D'autre part, il existe trois sortes de dérivation :

- **publique** – les membres de la classe de base conservent leur statut dans la classe dérivée ; c'est la situation la plus usuelle ;
- **privée** – tous les membres de la classe de base deviennent privés dans la classe dérivée ;
- **protégée** (depuis la version 3) – les membres publics de la classe de base deviennent membres protégés de la classe dérivée ; les autres membres conservent leur statut.

Lorsqu'un membre (donnée ou fonction) est redéfini dans une classe dérivée, il reste toujours possible (soit dans les fonctions membre de cette classe, soit pour un client de cette classe) d'accéder aux membres de même nom de la classe de base ; il suffit pour cela d'utiliser l'opérateur de résolution de portée (`::`), sous réserve, bien sûr, qu'un tel accès soit autorisé.

Appel des constructeurs et des destructeurs

Soit `B` une classe dérivée d'une classe de base `A`. Naturellement, dès lors que `B` possède au moins un constructeur, la création d'un objet de type `B` implique obligatoirement l'appel d'un constructeur de `B`. Mais, de plus, ce constructeur de `B` doit prévoir des arguments à destination d'un constructeur de `A` (une exception a lieu soit si `A` n'a pas de constructeur, soit si `A` possède un constructeur sans argument). Ces arguments sont précisés dans la définition du constructeur de `B`, comme dans cet exemple :

```
B (int x, int y, char coul) : A (x, y) ;
```

Les arguments mentionnés pour `A` peuvent éventuellement l'être sous forme d'expressions.

Cas particulier du constructeur par recopie

En plus des règles ci-dessus, il faut ajouter que si la classe dérivée `B` ne possède pas de constructeur par recopie, il y aura appel du constructeur par recopie par défaut de `B`, lequel procédera ainsi :

- appel du constructeur par recopie de `A` (soit celui qui y a été défini, soit le constructeur par recopie par défaut) ;
- initialisation des membres donnée de `B` qui ne sont pas hérités de `A`.

En revanche, un problème se pose lorsque la classe dérivée définit explicitement un constructeur par recopie. En effet, dans ce cas, il faut tenir compte de ce que l'appel de ce constructeur par recopie entraînera l'appel :

- du constructeur de la classe de base mentionné dans son en-tête, comme dans cet exemple (il s'agit ici d'un constructeur par recopie de la classe de base, mais il pourrait s'agir de n'importe quel autre constructeur) :

```
B (B & b) : A(b) ; // appel du constructeur par recopie de A
                   // auquel sera transmise la partie de B héritée de A
                   //  (grâce aux règles de compatibilité entre
                   //   classe dérivée et classe de base)
```

- d'un constructeur sans argument, si aucun constructeur de la classe de base n'est mentionné dans l'en-tête ; dans ce cas, il est nécessaire que la classe de base dispose d'un tel constructeur sans argument, faute de quoi on obtiendrait une erreur de compilation.

Conséquences de l'héritage

Considérons la situation suivante, dans laquelle la classe `A` possède une fonction membre `f` (dont nous ne précisons pas les arguments) fournissant un résultat de type `t` (quelconque : type de base ou type défini par l'utilisateur, éventuellement type classe) :

```
class A                          class B : public A
{   .....                        {   .....
  public :                       } ;
    t f (...) ;
    .....
} ;

A a ;      // a est du type A
B b ;      // b est du type B, dérivé de A
```

Naturellement, un appel tel que `a.f(...)` a un sens et il fournit un résultat de type `t`. Le fait que `B` hérite publiquement de `A` permet alors de donner un sens à un appel tel que :

```
b.f (...)
```

La fonction `f` agira sur `b`, comme s'il était de type `A`. **Le résultat fourni par `f` sera cependant toujours de type `t`**, même, notamment, lorsque le type `t` est précisément le type `A` (le résultat de `f` pourra toutefois être soumis à d'éventuelles conversions s'il est affecté à une `lvalue`).

Cas particulier de l'opérateur d'affectation

Considérons une classe `B` dérivant d'une classe `A`.

Si la classe dérivée `B` n'a pas surdéfini l'opérateur d'affectation, l'affectation de deux objets de type `B` se déroule membre à membre, en considérant que la « partie héritée de `A` » constitue un membre. Ainsi, les membres propres à `B` sont traités par l'affectation prévue pour leur type (par défaut ou surdéfinie, suivant le cas). La partie héritée de `A` est traitée par l'affectation prévue dans la classe `A`.

Si la classe dérivée `B` a surdéfini l'opérateur `=`, l'affectation de deux objets de type `B` fera nécessairement appel à l'opérateur `=` défini dans `B`. Celui de `A` ne sera pas appelé, même s'il a été surdéfini. **Il faudra donc que l'opérateur `=` de `B` prenne en charge tout ce qui concerne l'affectation d'objets de type `B`**, y compris pour ce qui est des membres hérités de `A` (quitte à faire appel à l'opérateur d'affectation de `A`).

Compatibilité entre objets d'une classe de base et objets d'une classe dérivée

Considérons :

```
class A                          class B : public A
{   .....                        {   .....
};                               } ;

A a ;      // a est du type A
B b ;      // b est du type B, dérivé de A
A * ada ;  // ada est un pointeur sur des objets de type A
B * adb ;  // adb est un pointeur sur des objets de type B
```

Il existe deux conversions implicites :

- d'un objet d'un type dérivé dans un objet d'un type de base. Ainsi l'affectation `a = b` est légale : elle revient à convertir `b` dans le type `A` (c'est-à-dire, en fait, à ne considérer de `b` que ce qui est du type `A`) et à affecter ce résultat à `a` (avec appel, soit de l'opérateur d'affectation de `A` si celui-ci a été surdéfini, soit de l'opérateur d'affectation par défaut de `A`). L'affectation inverse `b = a` est, quant à elle, illégale ;
- d'un pointeur sur une classe dérivée en un pointeur sur une classe de base. Ainsi l'affectation `ada = adb` est légale, tandis que `adb = ada` est illégale (elle peut cependant être forcée par emploi de l'opérateur de `cast` : `adb = (B*) ada`).

Exercice 102

Énoncé

On dispose d'un fichier nommé `point.h` contenant la déclaration suivante de la classe `point` :

```
class point
{   float x, y ;
  public :
    void initialise (float abs=0.0, float ord=0.0)
       { x = abs ; y = ord ;
       }
    void affiche ()
       { cout << "Point de coordonnées : " << x << " " << y << "\n" ;
       }
    float abs ()  { return x ; }
    float ord ()  { return y ; }
} ;
```

a. Créer une classe `pointb`, dérivée de `point` comportant simplement une nouvelle fonction membre nommée `rho`, fournissant la valeur du rayon vecteur (première coordonnée polaire) d'un point.

b. Même question, en supposant que les membres `x` et `y` ont été déclarés protégés (`protected`) dans `point`, et non plus privés.

c. Introduire un constructeur dans la classe `pointb`.

d. Quelles sont les fonctions membre utilisables pour un objet de type `pointb` ?

Solutions

a. Il suffit de prévoir, dans la déclaration de `pointb`, une nouvelle fonction membre de prototype :

```
float rho () ;
```

Toutefois, comme les membres `x` et `y` sont privés, ils restent privés pour les fonctions membre de sa classe dérivée `pointb` ; ce qui signifie qu'au sein de la définition de `rho`, il faut faire appel aux « fonctions d'accès » de `point` que sont `abs` et `ord`. Voici ce que pourrait être notre classe `pointb` (ici, nous avons fourni `rho` sous forme d'une fonction en ligne) :

```
#include "point.h"          // pour la déclaration de la classe point
#include <math.h>
class pointb : public point
{ public :
    float rho ()
       { return  sqrt (abs () * abs () + ord () * ord () ) ;
       }
} ;
```

Notez que, telle qu'elle a été définie, la classe `point` n'a pas donné naissance à un fichier objet (puisque toutes ses fonctions membre sont en ligne). Il en va de même ici pour `pointb`. Aussi, pour utiliser `pointb` au sein d'un programme, il suffira d'inclure les fichiers contenant les déclarations de `pointb`. Naturellement, dans la pratique, il en ira rarement ainsi ; en général, on devra fournir non seulement les déclarations de la classe de base et de sa classe dérivée, mais également les fichiers objet correspondant à leurs compilations respectives.

b. La définition précédente reste valable mais, néanmoins, comme les membres `x` et `y` de `point` ont été déclarés protégés, ils sont accessibles aux fonctions membre de sa classe dérivée ; aussi est-il possible, dans la définition de `rho`, de les utiliser « directement ». Voici ce que pourrait devenir notre fonction `rho` (toujours ici « en ligne ») :

```
float rho ()                        // il faut que x et y soient
   { return  sqrt (x * x + y * y) ;  // déclarés "protected" dans point
   }
```

c. Voici ce que pourrait être un constructeur à deux arguments (avec valeurs par défaut) :

```
pointb (float c1=0.0, float c2=0.0)
   { initialise (c1, c2) ;
   }
```

Là encore, si les membres `x` et `y` de *point* ont été déclarés «þprotégésþ», il est possible d'écrire ainsi notre constructeur :

```
pointb (float c1=0.0, float c2=0.0)
   { x = c1 ; y = c2 ;            // il faut que x et y soient
   }                              // déclarés "protected" dans point
```

Notez qu'ici il n'est pas possible au constructeur de `pointb` d'appeler un quelconque constructeur de `point` puisque ce dernier type ne possède pas de constructeur.

d. Un objet de type `pointb` peut utiliser n'importe laquelle des fonctions membre publiques de `point`, c'est-à-dire `initialise`, `affiche`, `abs` et `ord`, ainsi que n'importe laquelle des fonctions membre publiques de `pointb`, c'est-à-dire `rho` ou le constructeur `pointb`. Notez d'ailleurs qu'ici le constructeur et `initialise` font double emploi : cela provient d'une part de ce que `point` ne dispose pas de véritable constructeur, d'autre part de ce que `pointb` n'a pas défini de membres donnée supplémentaires, de sorte qu'il n'y a rien de plus à faire pour initialiser un objet de type `pointb` que pour initialiser un objet de type `point`.

Exercice 103

Énoncé

On dispose d'un fichier `point.h` contenant la déclaration suivante de la classe `point` :

```
#include <iostream>
using namespace std ;
class point
{   float x, y ;
  public :
    point (float abs=0.0, float ord=0.0)
       { x = abs ; y = ord ;
       }
    void affiche ()
       { cout << "Coordonnées : " << x << " " << y << "\n" ;
       }

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

a. Créer une classe `pointcol`, dérivée de `point`, comportant :

- un membre donnée supplémentaire `cl`, de type `int`, contenant la « couleur » d'un point ;
- les fonctions membre suivantes :

```
affiche (redéfinie), qui affiche les coordonnées et la couleur d'un
objet de type pointcol ;
colore (int couleur), qui permet de définir la couleur d'un objet de
type pointcol,
```

un constructeur permettant de définir la couleur et les coordonnées (on ne le définira pas en ligne).

b. Que fera alors précisément cette instruction :

```
pointcol (2.5, 3.25, 5) ;
```

Solutions

a. La fonction `colore` ne pose aucun problème particulier puisqu'elle agit uniquement sur un membre donnée propre à `pointcool`. En ce qui concerne `affiche`, il est nécessaire qu'elle puisse afficher les valeurs des membres `x` et `y`, hérités de `point`. Comme ces membres sont privés (et non protégés), il n'est pas possible que la nouvelle méthode `affiche` de `pointcol` accède à eux directement. Elle doit donc obligatoirement faire appel à la méthode `affiche` du type `point` ; il suffit, pour cela, d'utiliser l'opérateur de résolution de portée. Enfin, le constructeur de `pointcol` doit retransmettre au constructeur de `point` les coordonnées qu'il aura reçues par ses deux premiers arguments.

Voici ce que pourrait être notre classe `pointcol` (ici, toutes les fonctions membre, sauf le constructeur, sont en ligne) :

```
        /****** fichier pointcol.h :déclaration de pointcol ******/
#include "point.h"
#include <iostream>
using namespace std ;
class pointcol : public point
{   int cl ;
  public :
    pointcol (float = 0.0, float = 0.0, int = 0) ;
    void colore (int coul)
      { cl = coul ;
      }
    void affiche ()                    // affiche doit appeler affiche de
      { point::affiche () ;            // point pour les coordonnées
        cout << "  couleur : " << cl ; // mais elle a accès à la couleur
      }
} ;
```

```
        /****** définition du constructeur de pointcol *****/
#include "point.h"
#include "pointcol.h"
pointcol::pointcol (float abs, float ord, int coul) : point (abs, ord)
{  cl = coul ;       // on pourrait aussi écrire colore (coul) ;
}
```

Notez bien que l'on précise le constructeur de `point` devant être appelé par celui de `pointcol`, au niveau du constructeur de `pointcol`, et non de sa déclaration.

b. La déclaration `pointcol (2.5, 3.25, 5)` entraîne la création d'un emplacement pour un objet de type `pointcol`, lequel est initialisé par appel, successivement :

- du constructeur de `point`, qui reçoit en argument les valeurs `2.5` et `3.25` (comme prévu dans l'en-tête du constructeur de `pointcol`)þ;
- du constructeur de `pointcol`.

Exercice 104

Énoncé

On suppose qu'on dispose de la même classe `point` (et donc du fichier `point.h`) que dans l'exercice précédent. Créer une classe `pointcol` possédant les mêmes caractéristiques que ci-dessus, mais sans faire appel à l'héritage. Quelles différences apparaîtront entre cette classe `pointcol` et celle de l'exercice précédent, au niveau des possibilités d'utilisation ?

Solution

La seule démarche possible consiste à créer une classe `pointcol` dans laquelle un des membres donnée est lui-même de type `point`. Sa déclaration et sa définition se présenteraient alors ainsi :

```
        /***** fichier pointcol.h : déclaration de pointcol *****/
#include "point.h"
#include <iostream>
using namespace std ;
class pointcol
{   point p ;
    int cl ;
  public :
    pointcol (float = 0.0, float = 0.0, int = 0) ;
    void colore (int coul)
      { cl = coul ;
      }
```

```
    void affiche ()
      {  p.affiche () ;                  // affiche doit appeler affiche
         cout  << "  couleur : " << cl ; // du point p pour les
                                         // coordonnées
      }
} ;
        /****** définition du constructeur de pointcol *****/
#include "point.h"
#include "pointcol.h"
pointcol::pointcol (float abs, float ord, int coul) : p (abs, ord)
{  cl = coul ;
}
```

Apparemment, il existe une analogie étroite entre cette classe `pointcol` et celle de l'exercice précédent. Néanmoins, l'utilisateur de cette nouvelle classe ne peut plus faire directement appel aux fonctions membre héritées de `point`. Ainsi, pour appliquer la méthode `deplace` à un objet `a` de type `point`, il devrait absolument écrire : `a.p.deplace (...)` ; or, cela n'est pas autorisé ici, puisque `p` est un membre privé de `pointcol`.

Exercice 105

Énoncé

Soit une classe `point` ainsi définie (nous ne fournissons pas la définition de son constructeur) :

```
class point
{   int x, y ;
  public :
    point (int = 0, int = 0) ;
    friend int operator == (point, point) ;
} ;

int operator == (point a, point b)
{   return  a.x == b.x && a.y == b.y ;
}
```

Soit la classe `pointcol`, dérivée de `point` :

```
class pointcol : public point
{   int cl ;
  public :
    pointcol (int = 0, int = 0, int = 0) ;
    //   éventuelles fonctions membre
}  ;
```

a. Si `a` et `b` sont de type `pointcol` et `p` de type `point`, les instructions suivantes sont-elles correctes et, si oui, que font-elles ?

```
if (a == b) ...      // instruction 1
if (a == p) ...      // instruction 2
if (p == a) ...      // instruction 3
if (a == 5) ...      // instruction 4
if (5 == a) ...      // instruction 5
```

b. Mêmes questions, en supposant, cette fois, que l'opérateur + a été défini au sein de `point`, sous forme d'une fonction membre.

Solutions

a. Les 5 instructions proposées sont correctes. D'une manière générale, `x == y` est interprété comme `operator == (x, y)`. Si `x` et `y` sont de type `point`, aucun problème ne se pose bien sûr. Si l'un des opérandes est de type `pointcol` (ou les deux), il sera converti implicitement dans le type `point`. Si l'un des opérandes est de type `int`, il sera converti implicitement dans le type `point` (par utilisation du constructeur à un argument de `point`).

En ce qui concerne la signification de la comparaison, on voit qu'elle revient à ne considérer d'un objet de type `pointcol` que ses coordonnées. Pour un entier, elle revient à le considérer comme un `point` ayant cet entier pour abscisse et une ordonnée nulle.

N.B. Si les arguments de `operator=` étaient transmis par référence, les deux dernières affectations seraient rejetées, à moins d'avoir en plus prévu l'attribut `const`.

b. Cette fois, `x == y` est interprété comme `x.operator == (y)`. Si `x` est de type `point` et `y` d'un type pouvant se ramener au type `point` (c'est-à-dire soit du type `pointcol` qui sera converti implicitement en un type de base `point`, soit d'un type entier qui sera converti implicitement en un type `point` par l'intermédiaire du constructeur), aucun problème ne se pose (c'est le cas de la troisième instruction).

Si `x` est de type `pointcol` et `y` d'un type pouvant se ramener au type `point`, on retrouve le cas précédent, dans la mesure où la fonction membre `operator ==`, héritée de `point`, peut toujours s'appliquer à un objet de type `point` (c'est le cas des instructions 1, 2 et 4).

En revanche, si `x` est de type `int`, il n'est plus possible de lui appliquer une fonction membre. C'est ce qui se passe dans la dernière instruction, qui sera donc rejetée à la compilation.

N.B. Si l'unique argument de `operator=` était transmis par référence, la quatrième affectation serait rejetée, à moins d'avoir prévu en plus l'attribut `const`.

Exercice 106

Énoncé

Soit une classe `vect` permettant de manipuler des « vecteurs dynamiques » d'entiers (c'est-à-dire dont la dimension peut être fixée au moment de l'exécution) dont la déclaration (fournie dans le fichier `vect.h`) se présente ainsi :

```
class vect
{  int nelem ;        // nombre d'éléments
   int * adr ;        // adresse zone dynamique contenant les éléments
 public :
   vect (int) ;       // constructeur (précise la taille du vecteur)
   ~vect () ;                 // destructeur
   int & operator [] (int) ;  // accès à un élément par son "indice"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire pour le nombre d'entiers reçu en argument et que l'opérateur `[]` peut être utilisé indifféremment dans une expression ou à gauche d'une affectation.

Créer une classe `vectb`, dérivée de `vect`, permettant de manipuler des vecteurs dynamiques, dans lesquels on peut fixer les « bornes » des indices, lesquelles seront fournies au constructeur de `vectb`. La classe `vect` apparaîtra ainsi comme un cas particulier de `vectb` (un objet de type `vect` étant un objet de type `vectb` dans lequel la limite inférieure de l'indice est `0`).

On ne se préoccupera pas, ici, des problèmes éventuellement posés par la recopie ou l'affectation d'objets de type `vectb`.

Solution

Nous prévoirons, dans `vectb`, deux membres donnée supplémentaires (`debut` et `fin`) pour conserver les bornes de l'indice (en toute rigueur, on pourrait se contenter d'un membre supplémentaire contenant la limite inférieure, sachant que la valeur supérieure pourrait s'en déduire à partir de la connaissance de la taille du vecteur ; toutefois, cette dernière information n'étant pas publique, nous rencontrerions des problèmes d'accès !).

Manifestement, `vectb` nécessite un constructeur à deux arguments entiers correspondant aux bornes de l'indice ; son en-tête pourrait commencer ainsi :

```
vectb (int d, int f)
```

Comme l'appel de ce constructeur entraînera automatiquement celui du constructeur de `vect`, il n'est pas question de faire l'allocation dynamique de notre vecteur dans `vectb`. Au contraire, nous réutilisons le travail effectué par `vect`, auquel nous transmettrons simplement

le nombre d'éléments souhaités, c'est-à-dire ici `f - d + 1`. Voici l'en-tête complet du constructeur de `vectb` :

```
vectb (int d, int f) : vect (f-d+1)
```

La tâche spécifique de `vectb` se limitera à renseigner les valeurs des membres donnée `debut` et `fin`.

Aucun destructeur n'est nécessaire pour `vectb`, dans la mesure où son constructeur n'alloue aucun autre emplacement dynamique que celui alloué par `vect`.

En ce qui concerne l'opérateur `[]`, on peut penser que `vectb` l'hérite de `vect` et que, par conséquent, il n'est pas nécessaire de le surdéfinir. Toutefois, la notation `t[i]` ne désigne plus forcément l'élément de rang `i` d'un objet de type `vectb`. Or, manifestement, on souhaitera qu'il en aille toujours ainsi. Il faut donc redéfinir `[]` pour `vectb`, quitte d'ailleurs à réutiliser l'opérateur défini dans `vect`.

Voici ce que pourrait être notre classe `vectb` (ici, on ne trouve qu'une définition, puisque les deux fonctions membre ont été définies en ligne) :

```
#include "vect.h"
class vectb : public vect
{   int debut, fin ;
  public :
    vectb ( int d, int f) : vect (f-d+1)
      { debut = d ; fin = f ;
      }
    int & operator [] (int i)
      { return vect::operator [] (i-debut) ;
      }
} ;
```

Remarques

1. Si le membre donnée `adr` avait été déclaré protégé (`protected`) dans la classe `vect`, nous aurions pu redéfinir l'opérateur `[]` de `vectb`, sans faire appel à celui de `vect` :

   ```
   int & operator [] (int i)
     { return adr[i-debut] ;
     }
   ```

2. Aucune protection d'indices n'est à prévoir dans `vectb`, dès lors qu'elle a déjà été prévue dans `vect`.

Exercice 107

Énoncé

Soit une classe `int2d` (telle que celle créée dans l'exercice 91) permettant de manipuler des « tableaux dynamiques » d'entiers à deux dimensions dont la déclaration (fournie dans le fichier `int2d.h`) se présente ainsi :

```
class int2d
{  int nlig ;     // nombre de "lignes"
   int ncol ;     // nombre de "colonnes"
   int * adv ;    // adresse emplacement dynamique contenant les valeurs
  public :
   int2d (int nl, int nc) ;       // constructeur
   ~int2d () ;                    // destructeur
   int & operator () (int, int) ; // accès à un élément, par ses 2 "indices"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire et que l'opérateur `[]` peut être utilisé indifféremment dans une expression ou à gauche d'une affectation.

Créer une classe `int2db`, dérivée de `int2d`, permettant de manipuler des tableaux dynamiques, dans lesquels on peut fixer les « bornes » (valeur minimale et valeur maximale) des deux indices ; les quatre valeurs correspondantes seront fournies en arguments du constructeur de `int2db`.

On ne se préoccupera pas, ici, des problèmes éventuellement posés par la recopie ou l'affectation d'objets de type `int2db`.

Solution

Il suffit, en fait, de généraliser à la classe `int2d` le travail réalisé dans l'exercice précédent pour la classe `vect`. Voici ce que pourrait être notre classe `int2db` (ici, on ne trouve qu'une définition, dans la mesure où les fonctions membre de `int2db` ont été fournies en ligne) :

```
#include "int2d.h"
class int2db : public int2d
{   int ligdeb, ligfin ;      // bornes (mini, maxi) premier indice
    int coldeb, colfin ;      // bornes (mini, maxi) second indice
  public :                    // constructeur
    int2db (int ld, int lf, int cd, int cf) : int2d (lf-ld+1, cf-cd+1)
      {  ligdeb = ld ; ligfin = lf ;
         coldeb = cd ; colfin = cf ;
      }
    int & int2db::operator () (int i, int j   // redéfinition de operator ()
      {  return int2d::operator () (i-ligdeb, j-coldeb) ;
      }
} ;
```

Notez que, là non plus, aucune protection d'indice supplémentaire n'est à prévoir dans `int2db`, dès lors qu'elle a déjà été prévue dans `int2d`.

Exercice 108

Énoncé

Soit une classe `vect` permettant de manipuler des « vecteurs dynamiques » d'entiers, dont la déclaration (fournie dans un fichier `vect.h`) se présente ainsi (notez la présence de membres protégés) :

```
class vect
{ protected :     // en prévision d'une éventuelle classe dérivée
    int nelem ;        // nombre d'éléments
    int * adr ;        // adresse zone dynamique contenant les éléments
  public :
    vect (int) ;               // constructeur
    ~vect () ;                 // destructeur
    int & operator [] (int) ;  // accès à un élément par son "indice"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire et que l'opérateur `[]` peut être utilisé indifféremment dans une expression ou à gauche d'une affectation. En revanche, comme on peut le voir, cette classe n'a pas prévu de constructeur par recopie et elle n'a pas surdéfini l'opérateur d'affectation. L'affectation et la transmission par valeur d'objets de type `vect` posent donc les « problèmes habituels ».

Créer une classe `vectok`, dérivée de `vect`, telle que l'affectation et la transmission par valeur d'objets de type `vectok` s'y déroulent convenablement. Pour faciliter l'utilisation de cette nouvelle classe, introduire une fonction membre `taille` fournissant la dimension d'un vecteur.

Écrire un petit programme d'essai.

Solution

Manifestement, la classe `vectok` n'a pas besoin de définir de nouveaux membres donnée. Pour déclarer des objets de type `vectok`, il faudra pouvoir en préciser la dimension, ce qui signifie que `vectok` devra absolument disposer d'un constructeur approprié. Ce dernier se contentera toutefois de retransmettre la valeur reçue en argument au constructeur de `vect` ; il aura donc un corps vide. Notez qu'il n'est pas nécessaire de prévoir un destructeur (car celui de `vect` sera appelé en cas de destruction d'un objet de type `vectok` et il n'y a rien de plus à faire).

Notez que pour gérer convenablement la recopie ou l'affectation d'objets, nous nous contenterons de la méthode déjà rencontrée qui consiste à dupliquer les objets concernés (en en faisant une « copie profonde »).

Pour satisfaire aux contraintes de l'énoncé, il nous faut donc prévoir de définir, dans `vectok`, un constructeur par recopie. Il faut alors tenir compte de ce que la recopie d'un objet de type `vectok` (qui fera donc appel à ce constructeur) entraînera alors l'appel du constructeur de `vect` qui sera indiqué dans l'en-tête du constructeur par recopie de `vectok`, du moins si une telle information est précisée (dans le cas contraire, il y aurait appel d'un constructeur sans argument de `vect`, ce qui n'est pas possible ici). Ici, nous disposons donc de deux possibilités :

- demander l'appel du constructeur par recopie de `vect`, ce qui conduit à cet en-tête :

  ```
  vectok::vectok (vectok & v) : vect (v)
  ```

  (rappelons que l'argument `v`, de type `vectok`, sera implicitement converti en type `vect`, pour pouvoir être transmis au constructeur `vect`).

  Cette façon de faire conduit à la création d'un objet de type `vect`, obtenu par recopie (par défaut) de `v`. Il faudra alors compléter le travail en créant un nouvel emplacement dynamique pour le vecteur et en adaptant correctement la valeur de `adr`.

- demander l'appel du constructeur à un argument de `vect`, ce qui conduit à cet en-tête :

  ```
  vectok::vectok (vectok & v) : vect (v.nelem)
  ```

Cette fois, il y aura création d'un nouvel objet de type `vect`, avec son propre emplacement dynamique, dans lequel il faudra néanmoins recopier les valeurs de `v`. C'est cette dernière solution que nous choisirons ici.

Toujours pour satisfaire aux contraintes de l'énoncé, nous devons surdéfinir l'affectation dans la classe `vectok`. Ici, aucun choix ne se présente. Nous utiliserons l'algorithme présenté dans l'exercice 87.

Voici ce que pourraient être la déclaration et la définition de notre classe `vectok` :

```
          /**** déclaration de la classe vectok ****/
#include "vect.h"
class vectok : public vect
{        // pas de nouveaux membres donnée
  public :
    vectok (int dim) : vect (dim)   // constructeur de vectok : se contente
     {}                             // de passer dim au constructeur de vect
    vectok (vectok &) ;             // constructeur par recopie de vectok
    vectok & operator = (vectok &); // surdéfinition de l'affectation de vectok
    int taille ()
       { return nelem ;
       }
} ;
   /***** définition du constructeur par recopie de vectok *****/
   // il doit obligatoirement prévoir des arguments pour un constructeur
   //    (quelconque) de vect (ici le constructeur à un argument)
vectok::vectok (vectok & v) : vect (v.nelem) // ou const vectok & v
{  int i ;
   for (i=0 ; i<nelem ; i++)  adr[i] = v.adr[i] ;
}
```

```
  /***** définition de l'affectation entre vectok *****/
vectok & vectok::operator = (vectok & v)      // ou const vectok & v
{  if (this != &v)
     { delete adr ;
       adr = new int [nelem = v.nelem] ;
       int i ;
       for (i=0 ; i<nelem ; i++) adr[i] = v.adr[i] ;
     }
   return (*this) ;
}
```

Voici un exemple de programme utilisant la classe `vectok`, accompagné du résultat de son exécution :

```
#include <iostream>  // voir N.B. du paragraphe Nouvelles possibilités
                     // d'entrées-sorties du chapitre 2
using namespace std ;
#include "vectok.h"
main()
{  void fct (vectok) ;
   vectok v(6) ;
   int i ;
   for (i=0 ; i<v.taille() ; i++) v[i] = i ;
   cout << "vecteur v : " ;
   for (i=0 ; i<v.taille() ; i++) cout << v[i] << " " ;
   cout << "\n" ;
   vectok w(3) ;
   w = v ;
   cout << "vecteur w : " ;
   for (i=0 ; i<w.taille() ; i++) cout << w[i] << " " ;
   cout << "\n" ;
   fct (v) ;

}
void fct (vectok v)
{  cout << "vecteur reçu par fct : " << "\n" ;
   int i ;
   for (i=0 ; i<v.taille() ; i++) cout << v[i] << " " ;
}
```

```
vecteur v : 0 1 2 3 4 5
vecteur w : 0 1 2 3 4 5
vecteur reçu par fct :
0 1 2 3 4 5
```

Remarque

Voici, à titre indicatif, ce que serait la définition du constructeur par recopie de `vectok`, dans le cas où l'on ferait appel au constructeur par recopie (par défaut) de `vect` :

```
vectok::vectok (vectok & v) : vect (v)   // ou const vectok & v
{  nelem = v.nelem ;
   adr = new int [nelem] ;
   int i ;
   for (i=0 ; i<nelem ; i++)
      adr[i] = v.adr[i] ;
}
```

Ici, il a fallu allouer un nouvel emplacement pour un vecteur, ce qui n'était pas le cas lorsque l'on faisait appel au constructeur à un argument de `vect` (puisque ce dernier faisait déjà une telle allocation).