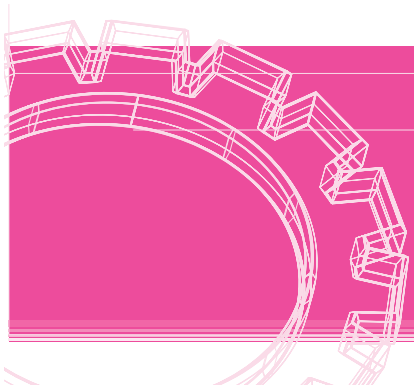
Chapitre 10

Les fonctions amies



Rappels

En C++, l'unité de protection est la classe, et non pas l'objet. Cela signifie qu'une fonction membre d'une classe peut accéder à tous les membres privés de n'importe quel objet de sa classe. En revanche, ces membres privés restent inaccessibles à n'importe quelle fonction membre d'une autre classe ou à n'importe quelle fonction indépendante.

La notion de fonction amie, ou plus exactement de « déclaration d'amitié », permet de déclarer dans une classe les fonctions que l'on autorise à accéder à ses membres privés (données ou fonctions). Il existe plusieurs situations d'amitié.

Fonction indépendante, amie d'une classe A

```
class A
{
   ......
   friend --- fct (-----) ;
   .....
}
```

La fonction `fct` ayant le prototype spécifié est autorisée à accéder aux membres privés de la classe `A`.

Fonction membre d'une classe B, amie d'une autre classe A

```
class A
{
   .....
   friend --- B:fct (-----) ;
   .....
} ;
```

La fonction `fct`, membre de la classe `B`, ayant le prototype spécifié, est autorisée à accéder aux membres privés de la classe `A`.

Pour qu'il puisse compiler convenablement la déclaration de `A`, donc en particulier la déclaration d'amitié relative à `fct`, le compilateur devra connaître la déclaration de `B` (mais pas nécessairement sa définition).

Généralement, la fonction membre `fct` possédera un argument ou une valeur de retour de type `A` (ce qui justifiera sa déclaration d'amitié). Pour compiler sa déclaration (au sein de la déclaration de `A`), il suffira au compilateur de savoir que `A` est une classe ; si sa déclaration n'est pas connue à ce niveau, on pourra se contenter de :

```
class A ;
```

En revanche, pour compiler la définition de `fct`, le compilateur devra posséder les caractéristiques de `A`, donc disposer de sa déclaration.

Toutes les fonctions d'une classe B sont amies d'une autre classe A

Dans ce cas, plutôt que d'utiliser autant de déclarations d'amitié que de fonctions membre, on utilise (dans la déclaration de la classe `A`) la déclaration (globale) suivante :

```
friend class B ;
```

Pour compiler la déclaration de `A`, on précisera simplement que `B` est une classe par :

```
class B ;
```

Quant à la déclaration de la classe `B`, elle nécessitera généralement (dès qu'une de ses fonctions membre possédera un argument ou une valeur de retour de type `A`) la déclaration de la classe `A`.

Exercice 81

Énoncé

Soit la classe `point` suivante :

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

Écrire une fonction indépendante `affiche`, amie de la classe `point`, permettant d'afficher les coordonnées d'un point. On fournira séparément un fichier source contenant la nouvelle déclaration (définition) de `point` et un fichier source contenant la définition de la fonction `affiche`. Écrire un petit programme (`main`) qui crée un point de classe automatique et un point de classe dynamique et qui en affiche les coordonnées.

Solution

Nous devons donc réaliser une fonction indépendante, nommée `affiche`, amie de la classe `point`. Une telle fonction, contrairement à une fonction membre, ne reçoit plus d'argument implicite ; `affiche` devra donc recevoir un argument de type `point`. Son prototype sera donc de la forme :

```
void affiche (point) ;
```

si l'on souhaite transmettre un point par valeur, ou de la forme :

```
void affiche (point &) ;
```

si l'on souhaite transmettre un point par référence. Ici, nous choisirons cette dernière possibilité et, comme `affiche` n'a aucune raison de modifier les valeurs du `point` reçu, nous le protégerons à l'aide du qualificatif `const` :

```
void affiche (const point &) ;
```

Remarque

Le qualificatif `const` permet d'appliquer la fonction `affiche` à un objet constant. Mais elle pourra également être appliquée à une expression de type `point`, voire à une expression d'un type susceptible d'être converti implicitement en un point (voir le chapitre relatif aux conversions définies par l'utilisateur). Ce dernier aspect ne constitue plus nécessairement un avantage !

Manifestement, affiche devra pouvoir accéder aux membres privés x et y de la classe point. Il faut donc prévoir une déclaration d'amitié au sein de cette classe, dont voici la nouvelle déclaration :

```
         /*         fichier POINT1.H        */
         /* déclaration de la classe point */
class point
{   int x, y ;
  public :
    friend void affiche (const point &) ;     // const non obligatoire
    point (int abs=0, int ord=0)
       { x=abs ; y=ord ;
       }
} ;
```

Pour écrire affiche, il nous suffit d'accéder aux membres (privés) x et y de son argument de type point. Si ce dernier se nomme p, les membres en question se notent (classiquement) p.x et p.y. Voici la définition de affiche :

```
#include "point1.h"          // nécessaire pour compiler affiche
#include <iostream>
using namespace std ;
void affiche (const point & p)
{   cout << "Coordonnées : " << p.x << " " << p.y << "\n" ;
}
```

Notez bien que la compilation de affiche nécessite la déclaration de la classe point, et pas seulement une déclaration telle que class point, car le compilateur doit connaître les caractéristiques de la classe point (notamment, ici, la localisation des membres x et y).

Voici le petit programme d'essai demandé :

```
#include "point1.h"
main()
{  point a(1,5) ;
   affiche (a) ;
   point * adp ;
   adp = new point (2, 12) ;
   affiche (*adp) ;              // attention *adp et non adp
}
```

Notez que nous n'avons pas eu à fournir le prototype de la fonction indépendante affiche, car il figure dans la déclaration de la classe point. Le faire constituerait d'ailleurs une erreur.

Exercice 82

Énoncé

Soit la classe `vecteur3d` définie dans l'exercice 70 par :

```
class vecteur3d
{   float x, y, z ;
   public :
    vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
      { x = c1 ; y = c2 ; z = c3 ; }
     .....
} ;
```

Écrire une fonction indépendante `coincide`, amie de la classe `vecteur3d`, permettant de savoir si deux vecteurs ont les mêmes composantes (cette fonction remplacera la fonction membre `coincide` qu'on demandait d'écrire dans l'exercice 70).

Si `v1` et `v2` désignent deux vecteurs de type `vecteur3d`, comment s'écrit maintenant le test de coïncidence de ces deux vecteurs ?

Solution

La fonction `coincide` va donc disposer de deux arguments de type `vecteur3d`. Si l'on prévoit de les transmettre par référence, en interdisant leur éventuelle modification dans la fonction, le prototype de `coincide` sera :

```
int coincide (const vecteur3d &, const vecteur3d &) ;
```

Remarque

Là encore, le qualificatif `const` a un double rôle : il permet d'appliquer la fonction `coincide` à des objets constants ou à des expressions de type `point`. Mais il permettra également de l'appliquer à toute valeur susceptible d'être convertie dans le type `point` (voir le chapitre relatif aux conversions définies par l'utilisateur).

Voici la nouvelle déclaration de notre classe :

```
        /*         fichier vect3D.H        */
        /* déclaration de la classe vecteur3d */
class vecteur3d
{   float x, y, z ;
  public :
    friend int coincide (const vecteur3d &, const vecteur3d &) ;
    vecteur3d (float c1=0, float c2=0, float c3=0)
       {  x = c1 ; y = c2 ; z = c3 ;
       }
} ;
```

Voici la définition de la fonction `coincide` :

```
#include "vect3d.h"          // nécessaire pour compiler coincide
int coincide (const vecteur3d & v1, const vecteur3d & v2)
{   if ( (v1.x == v2.x) && (v1.y == v2.y) && (v1.z == v2.z) )
            return 1 ;
     else return 0 ;
}
```

Le test de coïncidence de deux vecteurs s'écrit maintenant :

```
coincide (v1, v2)
```

On notera que, tant dans la définition de `coincide` que dans ce test, on voit se manifester la symétrie du problème, ce qui n'était pas le cas lorsque nous avions fait de `coincide` une fonction membre de la classe `vecteur3d`.

Exercice 83

Énoncé

Créer deux classes (dont les membres donnée sont privés) :

- l'une, nommée `vect`, permettant de représenter des vecteurs à 3 composantes de type `double` ; elle comportera un constructeur et une fonction membre d'affichage ;
- l'autre, nommée `matrice`, permettant de représenter des matrices carrées de dimension 3 x 3 ; elle comportera un constructeur avec un argument (adresse d'un tableau de 3 x 3 valeurs) qui initialisera la matrice avec les valeurs correspondantes.

Réaliser une fonction indépendante `prod` permettant de fournir le vecteur correspondant au produit d'une matrice par un vecteur. Écrire un petit programme de test. On fournira séparément les deux déclarations de chacune des classes, leurs deux définitions, la définition de `prod` et le programme de test.

Solution

Ici, pour nous faciliter l'écriture, nous représenterons les composantes d'un vecteur par un tableau à trois éléments et les valeurs d'une matrice par un tableau à 2 indices. La fonction de calcul du produit d'un vecteur par une matrice doit obligatoirement pouvoir accéder aux données des deux classes, ce qui signifie qu'elle devra être déclarée « amie » dans chacune de ces deux classes.

En ce qui concerne ses deux arguments (de type `vect` et `mat`), nous avons choisi la transmission la plus efficace, c'est-à-dire la transmission par référence. Quant au résultat (de type `vect`), il doit obligatoirement être renvoyé par valeur (nous en reparlerons dans la définition de `prod`).

Voici la déclaration de la classe `vect` :

```
                    /* fichier vect1.h */
class matrice ;         // pour pouvoir compiler la déclaration de vect
class vect
{
    double v[3] ;       // vecteur à 3 composantes
  public :
    vect (double v1=0, double v2=0, double v3=0)      // constructeur
      { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ; }
    friend vect prod (const matrice &, const vect &) ;// fonction amie
                                                      // indépendante
    void affiche () ;
} ;
```

et la déclaration de la classe `mat` :

```
                    /* fichier mat1.h */
class vect ;           // pour pouvoir compiler la déclaration de matrice
class matrice
{
    double mat[3] [3] ;       // matrice 3 X 3
  public :
    matrice () ;              // constructeur avec initialisation à zéro
    matrice (double t [3] [3] ) ; // constructeur à partir d'un tableau
                                  // 3 x 3
    friend vect prod (const matrice &, const vect &) ;  // fonction amie
                                                        // indépendante
} ;
```

Remarque

Nous avons déclaré constants les arguments de la fonction `vect`, ce qui nous protège d'une éventuelle faute de programmation dans les instructions de cette fonction. Dans ce cas, la fonction pourra être appelée avec en arguments effectifs non seulement des objets constants, mais aussi des expressions d'un type susceptible d'être converti dans le type voulu (comme on le verra dans le chapitre relatif aux conversions définies par l'utilisateur).

La définition des fonctions membre (en fait `affiche`) de la classe `vect` ne présente pas de difficultés :

```
#include "vect1.h"
#include <iostream>
using namespace std ;
void vect::affiche ()
{ int i ;
  for (i=0 ; i<3 ; i++) cout << v[i] << " " ;
  cout << "\n" ;
}
```

Il en va de même pour les fonctions membre (en fait le constructeur) de la classe `mat` :

```
#include "mat1.h"
#include <iostream>
using namespace std ;
matrice::matrice (double t [3] [3])
{
  int i ; int j ;
  for (i=0 ; i<3 ; i++)
    for (j=0 ; j<3 ; j++)
       mat[i] [j] = t[i] [j] ;
}
```

La définition `prod` nécessite les fichiers contenant les déclarations de `vect` et de `mat` :

```
#include "vect1.h"
#include "mat1.h"
vect prod (const matrice & m, const vect & x)
{
   int i, j ;
   double som ;
   vect res ;       // pour le résultat du produit
   for (i=0 ; i<3 ; i++)
     { for (j=0, som=0 ; j<3 ; j++)
          som += m.mat[i] [j] * x.v[j] ;
       res.v[i] = som ;
     }
   return res ;
}
```

Notez que cette fonction crée un objet automatique `res` de classe `vect`. Il ne peut donc être renvoyé que par valeur. Dans le cas contraire, la fonction appelante récupérerait l'adresse d'un emplacement libéré au moment de la sortie de la fonction.

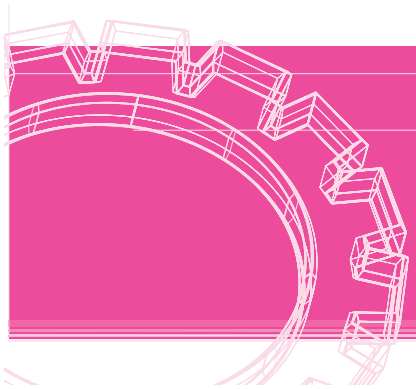
Voici, enfin, un exemple de programme de test, accompagné de son résultat :

```
#include "vect1.h"
#include "mat1.h"
main()
{  vect w (1,2,3) ;
   vect res  ;
   double tb [3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
   matrice a =  tb  ;
   res = prod(a, w) ;
   res.affiche () ;
}
```

```
 14 32 50
```

Chapitre 11

Surdéfinition d'opérateurs



Rappels

C++ vous permet de surdéfinir les opérateurs existants, c'est-à-dire de leur donner une nouvelle signification lorsqu'ils portent (en partie ou en totalité) sur des objets de type classe.

Le mécanisme de la surdéfinition d'opérateurs

Pour surdéfinir un opérateur existant `op`, on définit une fonction nommée `operator op` (on peut placer un ou plusieurs espaces entre le mot `operator` et l'opérateur, mais ce n'est pas une obligation) :

- soit sous forme d'une fonction indépendante (généralement amie d'une ou de plusieurs classes) ;
- soit sous forme d'une fonction membre d'une classe.

Dans le premier cas, si `op` est un opérateur binaire, la notation `a op b` est équivalente à :

```
operator op (a, b)
```

Dans le second cas, la même notation est équivalente à :

```
a.operator op (b)
```

Les possibilités et les limites de la surdéfinition d'opérateurs

On doit se limiter aux opérateurs existants, en conservant leur « pluralité » (unaire, binaire). Les opérateurs ainsi surdéfinis gardent leur priorité et leur associativité habituelle (voir tableau récapitulatif, un peu plus loin).

Un opérateur surdéfini doit toujours posséder un opérande de type classe (on ne peut donc pas modifier les significations des opérateurs usuels). Il doit donc s'agir :

- soit d'une fonction membre, auquel cas elle dispose obligatoirement d'un argument implicite du type de sa classe (`this`) ;
- soit d'une fonction indépendante (ou plutôt amie) possédant au moins un argument de type classe.

Il ne faut pas faire d'hypothèse sur la signification a priori d'un opérateur ; par exemple, la signification de `+=` pour une classe ne peut en aucun cas être déduite de la significaiton de `+` et de `=` pour cette même classe.

Cas particuliers

Les opérateurs `[]`, `()`, `->`, `new` et `delete` doivent obligatoirement être définis comme fonctions membre.

Les opérateurs `=` (affectation) et `&` (pointeur sur) possèdent une signification prédéfinie pour les objets de n'importe quel type classe. Cela ne les empêche nullement d'être surdéfinis. En ce qui concerne l'opérateur d'affectation, on peut choisir de transmettre son unique argument par valeur ou par référence. Dans le dernier cas, on ne perdra pas de vue que le seul moyen d'autoriser l'affectation d'une expression consiste à déclarer cet argument constant.

La surdéfinition de `new`, pour un type classe donné, se fait par une fonction de prototype :

```
void * new (size_t)
```

Elle reçoit, en unique argument, la taille de l'objet à allouer (cet argument sera généré automatiquement par le compilateur, lors d'un appel de `new`), et elle doit fournir en retour l'adresse de l'objet alloué.

La surdéfinition de `delete`, pour un `type` donné, se fait par une fonction de prototype :

```
void delete (type *)
```

Elle reçoit, en unique argument, l'adresse de l'objet à libérer.

Tableau récapitulatif

Les opérateurs surdéfinissables en C++ (classés par priorité décroissante)

Pluralité	Opérateurs	Associativité
Binaire	`()`(3) `[]`(3) `->`(3)	->
Unaire	`+ - ++`(5) `--`(5) `! ~ * &`(1) `new`(4)(6) `new[]`(4)(6) `delete`(4)(6) `delete[]`(4)(6) `(cast)`	<-
Binaire	`* / %`	->
Binaire	`+ -`	->
Binaire	`<< >>`	->
Binaire	`< <= > >=`	->
Binaire	`== !=`	->
Binaire	`&`	->
Binaire	`^`	->
Binaire	`||`	->
Binaire	`&&`	->
Binaire	`|`	->
Binaire	`=`(1)(3) `+= -= *= /= %= &= ^= |= <<= >>=`	<-
Binaire	`,`	->

(1) S'il n'est pas surdéfini, il possède une signification par défaut.

(3) Doit être défini comme fonction membre.

(4) Soit à un « niveau global » (fonction indépendante), soit pour une classe (fonction membre).

(5) Lorsqu'ils sont définis de façon unaire, ces opérateurs correspondent à la notation « pré » ; mais il en existe une définition binaire (avec deuxième opérande fictif de type `int`) qui correspond à la notation « post ».

(6) On distingue bien `new` de `new[]` et `delete` de `delete[]`

Remarque

Même lorsqu'on a surdéfini les opérateurs `new` et `delete` pour une classe, il reste possible de faire appel aux opérateurs `new` et `delete` usuels, en utilisant l'opérateur de résolution de portée (`::`).

Exercice 84

Énoncé

Soit une classe `vecteur3d` définie comme suit :

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
} ;
```

Définir les opérateurs == et != de manière qu'ils permettent de tester la coïncidence ou la non-coïncidence de deux points :

a. en utilisant des fonctions membreþ;

b. en utilisant des fonctions amies.

Solutions

a. Avec des fonctions membre

Il suffit donc de prévoir, dans la classe `vecteur3d`, deux fonctions membre de nom `operator ==` et `operator !=`, recevant un argument de type `vecteur3d` correspondant au second argument des opérateurs (le premier opérande étant fourni par l'argument implicite `this` des fonctions membre). Voici la déclaration complète de notre classe, accompagnée des définitions des opérateurs :

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
   int operator == (vecteur3d) ;
   int operator != (vecteur3d) ;
} ;

int vecteur3d::operator == (vecteur3d v)
{  if ( (v.x == x) && (v.y == y) && (v.z ==z) ) return 1 ;
            else return 0 ;
}
int vecteur3d::operator != (vecteur3d v)
{  return ! ( (*this) == v ) ;
}
```

Notez que, dans la définition de `!=`, nous nous sommes servi de l'opérateur `==`. En pratique, on sera souvent amené à réécrire entièrement la définition d'un tel opérateur, pour de simples raisons d'efficacité (d'ailleurs, pour les mêmes raisons, on placera « en ligne » les fonctions `operator ==` et `operator !=`).

b. Avec des fonctions amies

Il faut donc prévoir de déclarer comme amies, dans la classe `vecteur3d`, deux fonctions (`operator ==` et `operator !=`) recevant deux arguments de type `vecteur3d` correspondant aux deux opérandes des opérateurs Voici la nouvelle déclaration de notre classe, accompagnée des définitions des opérateurs :

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
   friend int operator == (vecteur3d, vecteur3d) ;
   friend int operator != (vecteur3d, vecteur3d) ;
} ;
int operator == (vecteur3d v, vecteur3d w)
{  if ( (v.x == w.x) && (v.y == w.y) && (v.z == w.z) ) return 1 ;
                                                    else return 0 ;
}
int operator != (vecteur3d v, vecteur3d w)
{  return ! ( v == w ) ;
}
```

Remarque

Voici, à titre indicatif, un exemple de programme, accompagné du résultat fourni par son exécution, utilisant n'importe laquelle des deux classes `vecteur3d` que nous venons de définir :

```
#include "vecteur3d.h"
#include <iostream>
using namespace std ;
main()
{  vecteur3d v1 (3,4,5), v2 (4,5,6), v3 (3,4,5) ;
   cout << "v1==v2 : " << (v1==v2) << " v1!=v2 : " << (v1!=v2) << "\n" ;
   cout << "v1==v3 : " << (v1==v3) << " v1!=v3 : " << (v1!=v3) << "\n" ;
}
```

```
v1==v2 : 0 v1!=v2 : 1
v1==v3 : 1 v1!=v3 : 0
```

Exercice 85

Énoncé

Soit la classe `vecteur3d` ainsi définie :

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
} ;
```

Définir l'opérateur binaire + pour qu'il fournisse la somme de deux vecteurs, et l'opérateur binaire * pour qu'il fournisse le produit scalaire de deux vecteurs. On choisira ici des fonctions amies.

Solution

Il suffit de créer deux fonctions amies nommées `operator +` et `operator *`. Elles recevront deux arguments de type `vecteur3d` ; la première fournira en retour un objet de type `vecteur3d`, la seconde fournira en retour un `float`.

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
   friend vecteur3d operator  + (vecteur3d, vecteur3d) ;
   friend float operator * (vecteur3d, vecteur3d) ;
} ;
vecteur3d operator + (vecteur3d v, vecteur3d w)
{  vecteur3d res ;
   res.x = v.x + w.x ;
   res.y = v.y + w.y ;
   return res ;
}
float operator * (vecteur3d v, vecteur3d w)
{  return (v.x * w.x + v.y * w.y + v.z * w.z) ;
}
```

Remarque

Il est possible de transmettre par référence les arguments des deux fonctions amies concernées.

Souvent, dans ce cas, on utilisera le qualificatif `const` puisque la fonction est supposée ne pas modifier les valeurs de ses arguments. Par exemple :

```
vecteur3d operator + (const vecteur3d & v, const vecteur3d & w)
```

Dans ce cas, la fonction peut certes être appelée avec des arguments effectifs constants. Mais il ne faudra pas perdre de vue qu'elle peut aussi être appelée avec arguments fournis sous forme d'expressions de type `vecteur3d`, voire avec des arguments d'un type autre que `vecteur3d`, pour peu qu'il existe une conversion implicite appropriée (voir le chapitre relatif aux conversions définies par l'utilisateur).

En revanche, il n'est pas possible de demander à `operator +` de transmettre son résultat par référence, puisque ce dernier est créé dans la fonction même, sous forme d'un objet de classe automatique. En toute rigueur, `operator *` pourrait transmettre son résultat (`float`) par référence, mais cela n'a guère d'intérêt en pratique.

Exercice 86

Énoncé

Soit la classe `vecteur3d` ainsi définie :

```
class vecteur3d
{
   float v[3] ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     {   // à compléter
     }
} ;
```

Compléter la définition du constructeur (en ligne), puis définir l'opérateur `[]` pour qu'il permette d'accéder à l'une des trois composantes d'un vecteur, et cela aussi bien au sein d'une expression (`... = v1[i]`) qu'à gauche d'un opérateur d'affectation (`v1[i] = ...`) ; de plus, on cherchera à se protéger contre d'éventuels risques de débordement d'indice.

Solution

La définition du constructeur ne pose aucun problème. En ce qui concerne l'opérateur `[]`, C++ ne permet de le surdéfinir que sous la forme d'une fonction membre (cette exception est justifiée par le fait que l'objet concerné ne doit pas risquer d'être soumis à une conversion,

lorsqu'il apparaît à gauche d'une affectation). Elle possédera donc un seul argument de type `int` et elle renverra un résultat de type `vecteur3d`.

Pour que notre opérateur puisse être utilisé à gauche d'une affectation, il faut absolument que le résultat soit renvoyé par référence. Pour nous protéger d'un éventuel débordement d'indice, nous avons simplement prévu que toute tentative d'accès à un élément en dehors des limites conduirait à accéder au premier élément.

Voici la déclaration de notre classe, accompagnée de la définition de la fonction `operator []` :

```
class vecteur3d
{
   float v [3] ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { v[0] = c1 ; v[1] = c2 ; v[2] = c3 ;
     }
   float & operator [] ( int) ;
} ;

float & vecteur3d::operator [] (int i)
{  if ( (i<0) || (i>2) ) i = 0 ;     // pour éviter un "débordement"
   return v[i] ;
}
```

Remarque

À titre indicatif, voici un petit exemple de programme faisant appel à notre classe `vecteur3d`, accompagné du résultat de son exécution :

```
#include "vecteur3d.h"
#include <iostream>
using namespace std ;

main()
{
  vecteur3d v1 (3,4,5) ;
   int i ;
   cout << "v1 = " ;
   for (i=0 ; i<3 ; i++) cout << v1[i] << " " ;
   for (i=0 ; i<3 ; i++) v1[i] = i ;
   cout << "\nv1 = "  ;
   for (i=0 ; i<3 ; i++) cout << v1[i] << " " ;
}


v1 = 3 4 5
v1 = 0 1 2
```

Exercice 87

Énoncé

L'exercice 77 vous avait proposé de créer une classe `set_int` permettant de représenter des ensembles de nombres entiers :

```
class set_int
{
   int * adval ;              // adresse du tableau des valeurs
   int nmax ;                 // nombre maxi d'éléments
   int nelem ;                // nombre courant d'éléments
 public :
   set_int (int = 20) ;       // constructeur
   ~set_int () ;              // destructeur
   ......  // autres fonctions membre
} ;
```

Son implémentation prévoyait de placer les différents éléments dans un tableau alloué dynamiquement ; aussi l'affectation entre objets de type `set_int` posait-elle des problèmes, puisqu'elle aboutissait à des objets différents comportant des pointeurs sur un même emplacement dynamique.

Modifier la classe `set_int` pour qu'elle ne présente plus de telles lacunes. On prévoira que tout objet de type `set_int` comporte son propre emplacement dynamique, comme on l'avait fait pour permettre la transmission par valeur. De plus, on s'arrangera pour que l'affectation multiple soit utilisable.

Solution

Nous sommes en présence d'un problème voisin de celui posé par le constructeur par recopie. Nous l'avions résolu en prévoyant ce que l'on appelle une « copie profonde » de l'objet concerné (c'est-à-dire une copie non seulement de l'objet lui-même, mais de toutes ses parties dynamiques). Quelques différences supplémentaires surgissent néanmoins. En effet, ici :

- on peut se trouver en présence d'une affectation d'un objet à lui-même ;
- avant affectation, il existe deux objets « complets » (avec leur partie dynamique), alors que dans le cas du constructeur par recopie, il n'existait qu'un seul objet, le second étant à créer.

Voici comment traiter l'affectation `b = a`, dans le cas où `b` est différent de `a` :

- libération de l'emplacement pointé par `b` ;
- création dynamique d'un nouvel emplacement dans lequel on recopie les valeurs de l'emplacement désigné par `a` ;
- mise en place des valeurs des membres donnée de `b`.

Si a et b désignent le même objet (on s'en assurera dans l'opérateur d'affectation, en comparant les adresses des objets concernés), on évitera d'appliquer ce mécanisme qui conduirait à un emplacement dynamique pointé par « personne », et qui, donc, ne pourrait jamais être libéré. En fait, on se contentera de... ne rien faire !

Par ailleurs, pour que l'affectation multiple (telle que c = b = a) fonctionne correctement, il est nécessaire que notre opérateur renvoie une valeur de retour (elle sera de type set_int) représentant la valeur de son premier opérande (après affectation, c'est-à-dire la valeur de b après b = a).

Voici ce que pourrait être la définition de notre fonction operator = (en ce qui concerne la déclaration de la classe set_int, il suffirait d'y ajouter set_int & operator = *(*set_int &*)* :

```
set_int & set_int::operator = (set_int & e)   // ou :   const set_int & e
// surdéfinition de l'affectation - les commentaires correspondent à b = a
{
   if (this != &e)                  // on ne fait rien pour a = a
     { delete adval ;               // libération partie dynamique de b
       adval = new int [nmax = e.nmax] ; // allocation nouvel ensemble
                                         // pour a
       nelem = e.nelem ;                 //      dans lequel on recopie
       int i ;                           //      entièrement l'ensemble b
       for (i=0 ; i<nelem ; i++)         //      avec sa partie dynamique
     adval[i] = e.adval[i] ;
     }
   return * this ;
}
```

Remarques

1. On associera obligatoirement const à la transmission par référence, si l'on souhaite pouvoir appliquer l'affectation à une expression de type set_int. Cela n'est pas nécessairement justifié ici.
2. Une telle surdéfinition d'un opérateur d'affectation pour une classe possédant des parties dynamiques ne sera valable que si elle est associée à une surdéfinition comparable du constructeur par recopie (pour la classe set_int, celle proposée dans la solution de l'exercice 26 convient parfaitement).
3. A priori, seule la valeur du premier opérande a vraiment besoin d'être transmise par référence (pour que = puisse le modifier !) ; cette condition est obligatoirement remplie puisque notre opérateur doit être surdéfini comme fonction membre. Toutefois, en pratique, on utilisera également la transmission par référence, à la fois pour le second opérande et pour la valeur de retour, de façon à être plus efficace (en temps). Notez d'ailleurs que si l'opérateur = renvoyait son résultat par valeur, il y aurait alors appel du constructeur de recopie (la remarque précédente s'appliquerait alors à une simple affectation).

Exercice 88

Énoncé

Considérer à nouveau la classe `set_int` créée dans l'exercice 77 (et sur laquelle est également fondé l'exercice précédent) :

```
class set_int
{  int * adval ;              // adresse du tableau des valeurs
   int nmax ;                 // nombre maxi d'éléments
   int nelem ;                // nombre courant d'éléments
 public :
   set_int (int = 20) ;       // constructeur
   ~set_int () ;              // destructeur
   ......
} ;
```

Adapter cette classe, de manière que :

- l'on puisse ajouter un élément à un ensemble de type `set_int` par (`e` désignant un objet de type `set_int` et `n` un entier) : `e < n ;`
- `e[n]` prenne la valeur `1` si `n` appartient à `e` et la valeur `0` dans le cas contraire. On s'arrangera pour qu'une instruction de la forme `e[i] = ...` soit **rejetée à la compilation**.

Solution

Il nous faut donc surdéfinir l'opérateur binaire <, de façon qu'il reçoive comme opérandes un objet de type `set_int` et un entier. Bien que l'énoncé ne prévoie rien, il est probable que l'on souhaitera pouvoir écrire des choses telles que (`e` étant de type `set_int`, `n` et `p` des entiers) :

```
e < n < p ;
```

Cela signifie donc que notre opérateur devra fournir comme valeur de retour l'ensemble concerné, après qu'on lui aura ajouté l'élément voulu.

Nous pouvons ici utiliser indifféremment une fonction membre ou une fonction amie. Nous choisirons la première possibilité. Par ailleurs, nous transmettrons les opérandes et la valeur de retour par référence (c'est possible ici car l'objet correspondant n'est pas créé au sein de l'opérateur même, c'est-à-dire qu'il n'est pas de classe automatique) ; ainsi, notre opérateur fonctionnera même si le constructeur par recopie n'a pas été surdéfini (en pratique toutefois, il faudra le faire dès qu'on souhaitera pouvoir transmettre la valeur d'un objet de type `set_int` en argument).

En ce qui concerne l'opérateur `[]`, il doit être surdéfini comme fonction membre, comme l'impose le C++, et cela bien qu'ici une affectation telle `e[i] =` soit interdite (alors que c'est précisément pour l'autoriser que C++ oblige d'en faire une fonction membre !). Pour interdire

de telles affectations, la démarche la plus simple consiste à faire en sorte que le résultat fourni par l'opérateur ne soit pas une `lvalue`, en la **transmettant par valeur**.

Voici ce que pourraient être la définition et la déclaration de notre nouvelle classe munie de ces deux opérateurs (notez que nous utilisons `[]` pour définir <) :

```
class set_int
{  int * adval ;              // adresse du tableau des valeurs
   int nmax ;                 // nombre maxi d'éléments
   int nelem ;                // nombre courant d'éléments
 public :
   set_int (int = 20) ;       // constructeur
   ~set_int () ;              // destructeur
   set_int & operator < (int) ;
   int operator [] (int) ;    // attention résultat par valeur
} ;

set_int::set_int (int dim)
{ adval = new int [nmax=dim] ;
  nelem = 0 ;
}

set_int::~set_int ()
{ delete adval ;
}

      /* surdéfinition de < */
set_int & set_int::operator < (int nb)
{   // on examine si nb appartient déjà à l'ensemble
    //   en utilisant l'opérateur []
   if ( ! (*this)[nb] && (nelem < nmax) )  adval [nelem++] = nb ;
   return (*this) ;
}
       /* surdéfinition de [] */
int set_int::operator [] (int nb)     // attention résultat par valeur
{  int i=0 ;
  // on examine si nb appartient déjà à l'ensemble
  //  (dans ce cas i vaudra nele en fin de boucle)
   while ( (i<nelem) && (adval[i] != nb) ) i++ ;
   return (i<nelem) ;
}
```

À titre indicatif, voici un exemple d'utilisation accompagné du résultat de son exécution :

```
#include "set_int.h"
#include <iostream>
using namespace std ;
```

```
main()
{ set_int ens(10) ;
  ens < 25 < 2 < 25 < 3 ;
  cout << (ens[25]) << " " << (ens[5]) << "\n" ;
}
```

```
 1 0
```

Exercice 89

Énoncé

Soit une classe `vecteur3d` définie comme suit :

```
class vecteur3d
{  float v [3] ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { v[0] = c1 ; v[1] = c2 ; v[2] = c3 ;
     }
   //   à compléter
} ;
```

Définir l'opérateur `[]` de manière que :

- il permette d'accéder « normalement » à un élément d'un objet non constant de type `vecteur3d`, et cela aussi bien dans une expression qu'en opérande de gauche d'une affectation ;
- il ne permette que la consultation (et non la modification) d'un objet constant de type `vecteur3d` (autrement dit, si v est un tel objet, une instruction de la forme `v[i] = ...` devra être rejetée à la compilation).

Solution

Rappelons que lorsque l'on définit des objets constants (qualificatif `const`), il n'est pas possible de leur appliquer une fonction membre publique, sauf si cette dernière a été déclarée avec le qualificatif `const` (auquel cas, elle peut indifféremment être utilisée avec des objets constants ou non constants). Ici, nous devons donc définir une fonction membre constante de nom `operator []`.

Par ailleurs, pour qu'une affectation de la forme `v[i] = ...` soit interdite, il est nécessaire que notre opérateur renvoie son résultat par valeur (et non par adresse comme on a généralement l'habitude de le faire).

Dans ces conditions, on voit qu'il est nécessaire de prévoir deux fonctions membre différentes, pour traiter chacun des deux cas : objet constant ou objet non constant. Le choix de la « bonne fonction » sera assuré par le compilateur, selon la présence ou l'absence de l'attribut `const` pour l'objet concerné.

Voici la définition complète de notre classe, accompagnée de la définition des deux fonctions `operator []` :

```
class vecteur3d
{  float v [3] ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { v[0] = c1 ; v[1] = c2 ; v[2] = c3 ;
     }
   float   operator [] (int) const ; // [] pour un vecteur constant
   float & operator [] (int) ;       // [] pour un vecteur non constant
} ;

         /******** operator [] pour un objet non constant *******/
float & vecteur3d::operator [] (int i)
{  if ( (i<0) || (i>2) ) i = 0 ;     // pour éviter un débordement
   return v[i] ;
}
         /********** operator [] pour un objet constant *********/
float   vecteur3d::operator [] (int i) const
{  if ( (i<0) || (i>2) ) i = 0 ;     // pour éviter un débordement
   return v[i] ;
}
```

À titre indicatif, voici un petit programme utilisant la classe `vecteur3d` ainsi définie, accompagné du résultat produit par son exécution :

```
#include "vecteur3d.h"
#include <iostream>  // voir N.B. du paragraphe Nouvelles possibilités
                     // d'entrées-sorties du chapitre 2
using namespace std ;
main()
{  int i ;
   vecteur3d v1 (3,4,5) ;
   const vecteur3d v2 (1,2,3) ;
   cout << "V1 : " ;
   for (i=0 ; i<3 ; i++) cout << v1[i] << " " ;
   cout << "\nV2 : " ;
   for (i=0 ; i<3 ; i++) cout << v2[i] << " " ;
   for (i=0 ; i<3 ; i++) v1[i] = i ;
   cout << "\nV1 : " ;
   for (i=0 ; i<3 ; i++) cout << v1[i] << " " ;
//      v2[1] = 3 ;  est bien rejeté à la compilation
}
```

```
V1 : 3 4 5
V2 : 1 2 3
V1 : 0 1 2
```

Exercice 90

Énoncé

Définir une classe `vect` permettant de représenter des « vecteurs dynamiques d'entiers », c'est-à-dire dont le nombre d'éléments peut ne pas être connu lors de la compilation. Plus précisément, on prévoira de déclarer de tels vecteurs par une instruction de la forme :

```
vect t(exp) ;
```

dans laquelle `exp` désigne une expression quelconque (de type `entier`).

On définira, de façon appropriée, l'opérateur `[]` de manière qu'il permette d'accéder à des éléments d'un objet d'un type `vect` comme on le ferait avec un tableau classique.

On ne cherchera pas à résoudre les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets de type `vect`. En revanche, on s'arrangera pour qu'aucun risque de « débordement » d'indice n'existe.

NB. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici. Il montrera également comment se protéger des débordements d'indice par une technique de gestion d'exceptions.

Solution

Les éléments d'un objet de type `vect` doivent obligatoirement être rangés en mémoire dynamique. L'emplacement correspondant sera donc alloué par le constructeur qui en recevra la taille en argument. Le destructeur devra donc, naturellement, libérer cet emplacement. En ce qui concerne l'accès à un élément, il se fera en surdéfinissant l'opérateur `[]`, comme nous l'avons déjà fait au cours des précédents exercices ; rappelons qu'il faudra obligatoirement le faire sous forme d'une fonction membre.

Pour nous protéger d'un éventuel débordement d'indice, nous ferons en sorte qu'une tentative d'accès à un élément situé en dehors du vecteur conduise à accéder à l'élément de rang `0`.

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

```
   /********** déclaration de la classe vect ********/
class vect
{  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"
} ;
   /*********** définition de la classe vect ********/
vect::vect (int n)
{  adr = new int [nelem = n] ;
   int i ;
   for (i=0 ; i<nelem ; i++) adr[i] = 0 ;
}
vect::~vect ()
{  delete adr ;
}
int & vect::operator [] (int i)
{  if ( (i<0)  || (i>=nelem) ) i=0 ;    // protection
   return adr [i] ;
}
```

Voici un petit exemple de programme d'utilisation d'une telle classe, accompagné du résultat produit par son exécution :

```
#include "vect.h"
#include <iostream>
using namespace std ;
main()
{  vect v(6) ;
   int i ;
   for (i=0 ; i<6 ; i++) v[i] = i ;
   for (i=0 ; i<6 ; i++) cout << v[i] << " " ;
}
```

```
 0 1 2 3 4 5
```

Exercice 91

Énoncé

En s'inspirant de l'exercice précédent, on souhaite créer une classe `int2d` permettant de représenter des tableaux dynamiques d'entiers à deux indices, c'est-à-dire dont les dimensions peuvent ne pas être connues lors de la compilation. Plus précisément, on prévoira de déclarer de tels tableaux par une déclaration de la forme :

```
int2d t(exp1, exp2) ;
```

dans laquelle `exp1` et `exp2` désignent une expression quelconque (de type entier).

On surdéfinira l'opérateur `()`, de manière qu'il permette d'accéder à des éléments d'un objet d'un type `int2d` comme on le ferait avec un tableau classique.

Là encore, on ne cherchera pas à résoudre les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets de type `int2d`. En revanche, on s'arrangera pour qu'il n'existe aucun risque de débordement d'indice.

N. B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici. Il montrera également comment se protéger contre les débordements d'indice par une technique de gestion d'exceptions.

Solution

Comme dans l'exercice précédent, les éléments d'un objet de type `int2d` doivent obligatoirement être rangés en mémoire dynamique. L'emplacement correspondant sera donc alloué par le constructeur qui en déduira la taille des deux dimensions reçues en arguments Le destructeur libérera cet emplacement.

Nous devons décider de la manière dont seront rangés les différents éléments en mémoire, à savoir « par ligne » ou « par colonne » (en toute rigueur, cette terminologie fait référence à la façon dont on a l'habitude de visualiser des tableaux à deux dimensions, ce que l'on nomme une ligne correspondant en fait à des éléments ayant même valeur du premier indice). Nous choisirons ici la première solution (c'est celle utilisée par C ou C++ pour les tableaux à deux dimensions). Ainsi, un élément repéré par les valeurs `i` et `j` des 2 indices sera situé à l'adresse (`adv` désignant l'adresse de l'emplacement dynamique alloué au tableau) :

```
adv + i * ncol + j
```

En ce qui concerne l'accès à un élément, il se fera en surdéfinissant l'opérateur `()`, d'une manière comparable à ce que nous avions fait pour `[]` ; là encore, C++ impose que `()` soit défini comme fonction membre.

Pour nous protéger d'un éventuel débordement d'indice, nous ferons en sorte qu'une valeur incorrecte d'un indice conduise à « faire comme si » on lui avait attribué la valeur `0`.

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

```
   /******** déclaration de la classe int2d ********/
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"
} ;

   /*********** définition du constructeur **********/
int2d::int2d (int nl, int nc)
{  nlig = nl ;
   ncol = nc ;
   adv = new int [nl*nc] ;
   int i ;
   for (i=0 ; i<nl*nc ; i++) adv[i] = 0 ;   // mise à zéro
}

   /*********** définition du destructeur **********/
int2d::~int2d ()
{  delete adv ;
}

   /********** définition de l'opérateur () *********/
int & int2d::operator () (int i, int j)
{  if ( (i<0) || (i>=nlig) ) i=0 ;  // protections sur premier indice
   if ( (j<0) || (j>=ncol) ) j=0 ;  // protections sur second indice
   return * (adv + i * ncol + j) ;
}
```

Voici un petit exemple d'utilisation de notre classe `int2d`, accompagné du résultat fourni par son exécution :

```
#include "int2d.h"
#include <iostream>
using namespace std ;
```

```
main()
{   int2d t1 (4,3) ;
    int i, j ;
    for (i=0 ; i<4 ; i++)
        for (j=0 ; j<3 ; j++)
            t1(i, j) = i+j ;
    for (i=0 ; i<4 ; i++)
      { for (j=0 ; j<3 ; j++)
            cout << t1 (i, j) << " " ;
        cout << "\n" ;
      }
}
```

```
0 1 2
1 2 3
2 3 4
3 4 5
```

Remarques

1. Dans la pratique, on sera amené à définir deux opérateurs `( )` comme nous l'avions fait dans l'exercice précédent pour l'opérateur `[]` : l'un pour des objets non constants (celui défini ici), l'autre pour des objets constants (il sera prévu de manière à ne pas pouvoir modifier l'objet sur lequel il porte).
2. Généralement, par souci d'efficacité, les opérateurs tels que `( )` seront définis « en ligne ».
3. On aurait pu songer à employer l'opérateur `[]` dont l'utilisation s'avère plus naturelle que celle de l'opérateur `( )`. Cependant, `[]` ne peut être surdéfini que sous forme binaire. Il n'est donc pas possible de l'employer sous la forme `t[i, j]` pour accéder à un élément d'un tableau dynamique. On pourrait alors penser à l'utiliser sous la forme habituelle `t[i][j]`. Or, cette écriture doit être interprétée comme `(t[i]) [j])`, ce qui signifie qu'on n'y applique plus le second opérateur à un objet de type `int2d`.

   Comme, de surcroît, `[]` doit être défini comme fonction membre, l'écriture en question demanderait de définir `[]` pour une classe `int2d`, en s'arrangeant pour qu'il fournisse un résultat (« vecteur ligne ») lui-même de type classe. Dans ce dernier cas, il faudrait également surdéfinir l'opérateur `[]` pour qu'il fournisse un résultat de type `int`. Pour obtenir le résultat souhaité, il serait alors nécessaire de définir d'autres classes que `int2d`.

Exercice 92

Énoncé

Créer une classe nommée `histo` permettant de manipuler des « histogrammes ». On rappelle que l'on obtient un histogramme à partir d'un ensemble de valeurs `x(i)`, en définissant `n` tranches (intervalles) contiguës (souvent de même amplitude) et en comptabilisant le nombre de valeurs `x(i)` appartenant à chacune de ces tranches.

On prévoira :

- un constructeur de la forme `histo (float min, float max, int ninter)`, dont les arguments précisent les bornes (`min` et `max`) des valeurs à prendre en compte et le nombre de tranches (`ninter`) supposées de même amplitude ;
- un opérateur << défini tel que `h<<x` ajoute la valeur `x` à l'histogramme `h`, c'est-à-dire qu'elle incrémente de `1` le compteur relatif à la tranche à laquelle appartient `x`. Les valeurs sortant des limites (`min - max`) ne seront pas comptabilisées ;
- un opérateur `[]` défini tel que `h[i]` représente le nombre de valeurs répertoriées dans la tranche de rang `i` (la première tranche portant le numéro `1` ; un numéro incorrect de tranche conduira à considérer celle de rang `1`). On s'arrangera pour qu'une instruction de la forme `h[i] = ...` soit rejetée en compilation.

On ne cherchera pas ici à régler les problèmes posés par l'affectation ou la transmission par valeur d'objets du type `histo`.

Solution

Nous n'avons pas besoin de conserver les différentes valeurs `x(i)`, mais seulement les compteurs relatifs à chaque tranche. En revanche, il faut prévoir d'allouer dynamiquement (dans le constructeur) l'emplacement nécessaire à ces compteurs, puisque le nombre de tranches n'est pas connu lors de la compilation de la classe `histo`. Il faudra naturellement prévoir de libérer cet emplacement dans le destructeur de la classe. Les membres donnée de `histo` seront donc les valeurs extrêmes (minimale et maximale), le nombre de tranches et un pointeur sur l'emplacement contenant les compteurs.

L'opérateur << peut être surdéfini, soit comme fonction membre, soit comme fonction amie ; nous choisirons ici la première solution. En revanche, l'opérateur `[]` doit absolument être surdéfini comme fonction membre. Pour qu'il ne soit pas possible d'écrire des affectations de la forme `h[i] = ...`, on fera en sorte que cet opérateur fournisse son résultat par valeur.

Voici ce que pourrait être la déclaration de notre classe `histo` :

```
    /*** fichier histo.h : déclaration de la classe histo ***/
class histo
{   float min ;  // borne inférieure
    float max ;  // borne supérieure
    int nint ;   // nombre de tranches utiles
```

```
    int * adc ;  // pointeur sur les compteurs associés à chaque intervalle
                 // adc [i-1] = compteur valeurs de la tranche de rang i
    float ecart ;// larg d'une tranche (pour éviter un recalcul systématique)
  public :
    histo (float=0.0, float=1.0, int=10) ; // constructeur
    ~histo () ;                    // destructeur
    histo & operator << (float) ;  // ajoute une valeur
    int operator [] (int) ;        // nombre de valeurs dans chaque tranche
} ;
```

Notez que nous avons prévu des valeurs par défaut pour les arguments du constructeur (celles-ci n'étaient pas imposées par l'énoncé).

En ce qui concerne la définition des différents membres, il faut noter qu'il est indispensable qu'une telle classe soit protégée contre toute utilisation incorrecte. Certes, cela passe par un contrôle de la valeur du numéro de tranche fourni à l'opérateur `[]`, ou par le refus de prendre en compte une valeur hors limites (qui fournirait un numéro de tranche conduisant à un emplacement situé en dehors de celui alloué par le constructeur).

Mais nous devons de surcroît nous assurer que les valeurs des arguments fournis au constructeur ne risquent pas de mettre en cause le fonctionnement ultérieur des différentes fonctions membre. En particulier, il est bon de vérifier que le nombre de tranches n'est pas négatif (notamment, une valeur nulle conduirait dans `<<` à une division par zéro) et que les valeurs du minimum et du maximum sont convenablement ordonnées et différentes l'une de l'autre (dans ce dernier cas, on aboutirait encore à une division par zéro). Ici, nous avons prévu de régler ce problème en attribuant le cas échéant des valeurs par défaut arbitraires (`max = min + 1, nint = 1`).

Voici ce que pourrait être la définition des différentes fonctions membre de notre classe `histo` :

```
      /************** définition de la classe histo **********/
#include "histo.h"
#include <iostream>
using namespace std ;

/******************** constructeur ********************/
histo::histo (float mini, float maxi, int ninter)
{           // protection contre arguments erronés
    if (maxi < mini)
       { float temp = maxi ; maxi = mini ; mini = temp ; }
    if (maxi == mini) maxi = mini + 1.0 ;   // arbitraire
    if (ninter < 1) nint =1 ;
    min = mini ; max = maxi ; nint = ninter ;
    adc = new int [nint] ;                  // alloc emplacements compteurs
    int i ;
    for (i=0 ; i<=nint-1 ; i++) adc[i] = 0 ; // et r.a.z.
    ecart = (max - min) / nint ;             // largeur d'une tranche
}
```

```
      /******************** destructeur ********************/
histo::~histo ()
{   delete adc ;
}

      /******************** opérateur << *******************/
histo & histo::operator << (float v)
{   int nt =  (v-min) / ecart ;
    // on ne comptabilise que les valeurs "convenables"
    if ( (nt>=0) && (nt<=nint-1) )  adc [nt] ++ ;
    return (*this) ;
}

      /******************** opérateur [] *******************/
int histo::operator [] (int n)         // résultat par valeur ici
{   if ( (n<1) || (n>nint) ) n=1 ;     // protection "débordement"
    return adc[n-1] ;
}
```

Voici, à titre indicatif, un petit programme d'essai de la classe `histo`, accompagné du résultat fourni par son exécution :

```
#include "histo.h"
#include <iostream>
using namespace std ;

main()
{   const int nint = 4 ;
    int i ;
    histo h (0., 5., nint) ;     // 4 tranches entre 0 et 5
    h << 1.5 << 2.4 << 3.8 << 3.0 << 2.0 << 3.5 << 2.8 << 4.6 ;
    h << 12.0 << -3.5 ;
    for (i=0 ; i<10 ; i++) h << i/2.0 ;
    cout << "valeurs des tranches \n" ;
    for (i=1 ; i<=nint ; i++)
       cout << "numéro " << i << " : " << h[i] << "\n"  ;
    // une affectation telle que  h[2] = ... serait rejetée en compilation
}
```

```
valeurs des tranches
numéro 1 : 3
numéro 2 : 5
numéro 3 : 6
numéro 4 : 4
```

Exercice 93

Énoncé

Réaliser une classe nommée `stack_int` permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un emplacement alloué dynamiquement ; sa dimension sera déterminée par l'argument fourni à son constructeur (on lui prévoira une valeur par défaut de 20). Cette classe devra comporter les opérateurs suivants (nous supposons que `p` est un objet de type `stack_int` et `n` un entier) :

- <<, tel que `p<<n` ajoute l'entier `n` à la pile `p` (si la pile est pleine, rien ne se passe) ;
- >>, tel que `p>>n` place dans `n` la valeur du haut de la pile `p`, en la supprimant de la pile (si la pile est vide, la valeur de `n` ne sera pas modifiée) ;
- `++`, tel que `p++` vale `1` si la pile `p` est pleine et `0` dans le cas contraire ;
- `--`, tel que `p--` vale `1` si la pile `p` est vide et `0` dans le cas contraire.

On prévoira que les opérateurs << et >> pourront être utilisés sous les formes suivantes (`n1`, `n2` et `n3` étant des entiers) :

```
p << n1 << n2 << n3 ;          p >> n1 >> n2 << n3 ;
```

On fera en sorte qu'il soit possible de transmettre une pile par valeur. En revanche, l'affectation entre piles ne sera pas permise, et on s'arrangera pour que cette situation aboutisse à un arrêt de l'exécution.

N. B. Le chapitre 17 vous montrera comment résoudre le présent exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solution

La classe `stack_int` contiendra comme membres donnée : la taille de l'emplacement réservé pour la pile (`nmax`), le nombre d'éléments placés à un moment donné sur la pile (`nelem`) et un pointeur sur l'emplacement qui sera alloué par le constructeur pour y ranger les éléments de la pile (`adv`). Notez qu'il n'est pas nécessaire de prévoir une donnée supplémentaire pour un éventuel « pointeur » de pile, dans la mesure où c'est le nombre d'éléments `nelem` qui joue ici ce rôle.

Les opérateurs requis peuvent indifféremment être définis comme fonctions membre ou comme fonctions amies. Nous choisirons ici la première solution. Pour que les opérateurs << et >> puissent être utilisés à plusieurs reprises dans une même expression, il est nécessaire que, par exemple :

```
p << n1 << n2 << n3 ;
```

soit équivalent à :

```
( ( (p << n1) << n2 ) << n3 ) ;
```

Pour ce faire, les opérateurs << et >> doivent fournir comme résultat la pile reçue en premier opérande, après qu'elle a subi l'opération voulue (empilage ou dépilage). Il est préférable que ce résultat soit transmis par référence (on évitera la perte de temps due au constructeur par recopie).

En ce qui concerne la transmission par valeur d'un objet de type `stack_int`, nous ne pouvons pas nous contenter du constructeur par recopie par défaut puisque ce dernier ne recopierait pas la partie dynamique de l'objet, ce qui poserait les problèmes habituels. Nous devons donc surdéfinir le constructeur par recopie de façon qu'il réalise une « copie profonde » de l'objet.

Pour satisfaire à la contrainte imposée par l'énoncé sur l'affectation, nous allons surdéfinir l'opérateur d'affectation. Comme ce dernier doit se contenter d'afficher un message d'erreur et d'interrompre l'exécution, il n'est pas nécessaire qu'il renvoie une quelconque valeur (d'où la déclaration `void`).

Voici ce que pourrait être la déclaration de notre classe :

```
#include <stdlib.h>
#include <iostream>
using namespace std ;
class stack_int
{   int nmax ;                // nombre maximal de la valeurs de la pile
    int nelem ;               // nombre courant de valeurs de la pile
    int * adv ;               // pointeur sur les valeurs
  public :
    stack_int (int = 20) ;           // constructeur
    ~stack_int () ;                  // destructeur
    stack_int (stack_int &) ;        // constructeur par recopie
                                     // voir remarque 1 ci-après
    void operator = (stack_int &) ;  // affectation – voir remarque 2
    stack_int & operator << (int) ;  // opérateur d'empilage
    stack_int & operator >> (int &) ; // opérateur de dépilage
                                     // (attention int &)
    int operator ++ () ;             // opérateur de test pile pleine
    int operator -- () ;             // opérateur de test pile vide
} ;
```

Remarques

1. Il est nécessaire que l'argument de `stack_int` soit transmis par référence. On peut utiliser `const` ; dans ce cas, on pourra initialiser un objet avec un objet constant ou une expression d'un type susceptible d'être converti implicitement dans le type `stack_int` (ce qui est ici le cas des types entiers ou flottants ; voir le chapitre relatif aux conversions définies par l'utilisateur).
2. En revanche, il n'est pas nécessaire que l'argument de `operator=` soit transmis par référence. Si l'on souhaitait pouvoir affecter des objets constants, il faudrait ajouter le qualificatif `const` ; dans ce cas, on pourrait alors, du même coup, affecter des expressions d'un type convertible dans le type `stack_int`, c'est-à-dire ici d'un type entier ou flottant (voir le chapitre relatif aux conversions définies par l'utilisateur).
3. L'opérateur >> doit absolument recevoir son deuxième opérande par référence, puisqu'il doit pouvoir en modifier la valeur.

Voici la définition des fonctions membre de `stack_int` :

```
#include "stack-int.h"
stack_int::stack_int (int n)
{   nmax = n ;
    adv = new int [nmax] ;
    nelem = 0 ;
}
stack_int::~stack_int ()
{   delete adv ;
}

stack_int::stack_int (stack_int & p)
{   nmax = p.nmax ; nelem = p.nelem ;
    adv = new int [nmax] ;
    int i ;
    for (i=0 ; i<nelem ; i++)
       adv[i] = p.adv[i] ;
}
void stack_int::operator = (stack_int & p)
{   cout << "*** Tentative d'affectation entre piles - arrêt exécution ***\n" ;
    exit (1) ;
}
stack_int & stack_int::operator << (int n)
{   if (nelem < nmax) adv[nelem++] = n ;
    return (*this) ;
}
stack_int & stack_int::operator >> (int & n)
{   if (nelem > 0) n = adv[--nelem] ;
    return (*this) ;
}
```

```
int stack_int::operator ++ ()
{   return  (nelem == nmax) ;
}
int stack_int::operator -- ()
{   return (nelem == 0) ;
}
```

À titre indicatif, voici un petit programme d'utilisation de notre classe, accompagné d'un exemple d'exécution :

```
  /************ programme d'essai de stack_int *********/
#include "stack_int.h"
#include <iostream>
using namespace std ;
main()
{   void fct (stack_int) ;
    stack_int pile (40) ;
    cout << "pleine : " << pile++ << " vide : " << pile-- << "\n" ;
    pile << 1 << 2 << 3 << 4 ;
    fct (pile) ;
    int n, p ;
    pile >> n >> p ;     // on dépile 2 valeurs
    cout << "haut de la pile au retour de fct : " << n << " " << p << "\n" ;
    stack_int pileb (25) ;
    pileb = pile ;       // tentative d'affectation
}
void fct (stack_int pl)
{   cout << "haut de la pile reçue par fct : " ;
    int  n, p ;
    pl >> n >> p ;  // on dépile 2 valeurs
    cout << n << " " << p << "\n" ;
    pl << 12 ;      // on en ajoute une
}
```

```
pleine : 0 vide : 1
haut de la pile reçue par fct : 4 3
haut de la pile au retour de fct : 4 3
*** Tentative d'affectation entre piles - arrêt exécution ***
```

Exercice 94

Énoncé

Adapter la classe `point` :

```
class point
{
  int x, y ;
    // .......
  public :
    point (int abs=0, int ord=0)                        // constructeur
    // .....
} ;
```

de façon qu'elle dispose de deux fonctions membre permettant de connaître, à tout instant, le nombre total d'objets de type `point`, ainsi que le nombre d'objets dynamiques (c'est-à-dire créés par `new`) de ce même type.

Solution

Ici, il n'est plus possible de se contenter de comptabiliser les appels au constructeur et au destructeur. Il faut, en outre, comptabiliser les appels à `new` et `delete`. La seule possibilité pour y parvenir consiste à surdéfinir ces deux opérateurs pour la classe `point`. Le nombre de points total et le nombre de points dynamiques seront conservés dans des membres statiques (n'existant qu'une seule fois pour l'ensemble des objets du type `point`). Quant aux fonctions membre fournissant les informations voulues, il est préférable d'en faire des fonctions membre statiques (déclarées, elles aussi, avec l'attribut `static`), ce qui permettra de les employer plus facilement que si on en avait fait des fonctions membre ordinaires (puisqu'on pourra les appeler sans avoir à les faire porter sur un objet particulier).

Voici ce que pourrait être notre classe `point` (déclaration et définition) :

```
#include <stddef.h>                 // pour size_t
#include <iostream>
using namespace std ;
class point
{   static int npt ;                // nombre total de points
    static int nptd ;               // nombre de points dynamiques
    int x, y ;
  public :
    point (int abs=0, int ord=0)            // constructeur
        { x=abs ; y=ord ;
          npt++ ;
        }
    ~point ()                               // destructeur
        { npt-- ;
        }
```

```
    void * operator new (size_t sz)         // new surdéfini
    { nptd++ ;
      return ::new char[sz] ;               // appelle new prédéfini
    }
    void operator delete (void * dp)
     { nptd-- ;
       ::delete (dp) ;                      // appelle delete prédéfini
     }
    static int npt_tot ()
     { return npt ;
     }
    static int npt_dyn ()
     { return nptd ;
     }
} ;

int point::npt = 0 ;     // initialisation des membres statiques de point
int point::nptd = 0 ;
```

Notez que, dans la surdéfinition de `new` et `delete`, nous avons fait appel aux opérateurs prédéfinis (par emploi de `::`) pour ce qui concerne la gestion de la mémoire.

Voici un exemple de programme utilisant notre classe `point`, accompagné du résultat fourni par son exécution :

```
#include "point.h"
#include <iostream>
using namespace std ;
main()
{
   point * ad1, * ad2 ;
   point a(3,5) ;
   cout << "A : " << point::npt_tot () << " " << point::npt_dyn () << "\n" ;
   ad1 = new point (1,3) ;
   point b ;
   cout << "B : " << point::npt_tot () << " " << point::npt_dyn () << "\n" ;
   ad2 = new point (2,0) ;
   delete ad1 ;
   cout << "C : " << point::npt_tot () << " " << point::npt_dyn () << "\n" ;
   point c(2) ;
   delete ad2 ;
   cout << "D : " << point::npt_tot () << " " <<  point::npt_dyn () << "\n" ;
}
```

```
A : 1 0
B : 3 1
C : 3 1
D : 3 0
```