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.

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.

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.

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.

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**.

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).

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.

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.

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`.