Exercices en langage C++

C. Delannoy

AVANT-PROPOS

La maîtrise d'un langage de programmation passe obligatoirement par la pratique, c'est-à-dire la recherche personnelle d'une solution à un problème donné. Cette affirmation reste vraie pour le programmeur chevronné qui étudie un nouveau langage. C'est dans cette situation que se trouve généralement une personne qui aborde le C++ : elle connaît déjà le langage C sur lequel s'appuie effectivement C++ ; toutefois, ce dernier langage introduit suffisamment de possibilités supplémentaires et surtout de nouveaux concepts (en particulier, ceux de la Programmation Orientée Objet) pour que son apprentissage s'apparente à celui d'un nouveau langage.

Ce livre vous propose d'accompagner votre étude du C++ et de la prolonger, ceci à l'aide d'exercices appropriés, variés et de difficulté croissante.

Les différents chapitres, à l'exception du dernier, correspondent à une progression classique d'un "cours de C++" : incompatibilités entre C et C++ ; les spécificités de C++ ; notions de classe, constructeur et destructeur ; propriétés des fonctions membre ; construction, destruction et initialisation des objets ; les fonctions amies ; la surdéfinition d'opérateurs ; les conversions de type définies par l'utilisateur ; la technique de l'héritage ; les fonctions virtuelles ; les flots d'entrée et de sortie, les patrons de fonctions et les patrons de classes. Le dernier chapitre, enfin, propose des "exercices de synthèse".

Chaque chapitre débute par un "rappel détaillé"[1] des connaissances nécessaires pour aborder les exercices correspondants (naturellement, un exercice d'un chapitre donné peut faire intervenir des points résumés dans les chapitres précédents).

Au sein de chaque chapitre, les exercices proposés vont d'une application immédiate du cours à des réalisations de classes relativement complètes. Au fil de votre progression dans l'ouvrage, vous réaliserez des classes de plus en plus "réalistes et opérationnelles" et ayant un intérêt général ; citons, par exemple :

- les ensembles,
- les vecteurs dynamiques,
- les tableaux dynamiques à plusieurs dimensions,
- les listes chaînées,
- les tableaux de bits,
- les (vraies) chaînes de caractères,
- les piles,
- les complexes,
- etc.

Naturellement, tous les exercices sont "corrigés". Pour la plupart, la solution proposée ne se limite pas à une simple liste d'un programme (laquelle ne représente finalement qu'une rédaction possible parmi d'autres). Vous y trouverez une analyse détaillée du problème et, si besoin, les justifications de certains choix. Des commentaires viennent, le cas échéant, éclairer les parties quelque peu délicates. Fréquemment, vous trouverez des suggestions de prolongement ou de généralisation du problème abordé.

---

[1] Le cours complet correspondant à ces résumés peut être trouvé dans "Programmer en langage C++" ou dans "Programmer en Turbo C++", du même auteur, aux Editions Eyrolles.

Outre la maîtrise du langage C++ proprement dit, les exercices proposés vous permettront de vous forger une "méthodologie de conception de vos propres classes". Notamment, vous saurez :

- décider du bien-fondé de la surdéfinition de l'opérateur d'affectation ou du constructeur par recopie,
- exploiter, lorsque vous jugerez que cela est opportun, les possibilités de "conversions automatiques" que le compilateur peut mettre en place,
- comment tirer parti de l'héritage (simple ou multiple) et quels avantages présente la création d'une "bibliothèque de classes", notamment par le biais du "typage dynamique des objets" qui découle de l'emploi des fonctions virtuelles.

De surcroît, vous aurez rencontré un certain nombre de techniques fondamentales telles que : la réalisation d'un "itérateur" ou les "classes conteneurs" et l'utilisation des possibilités de "fonctions génériques" (patrons de fonctions) et de "classes génériques" (patrons de classes).

N.B.

En attendant la normalisation de C++, ce sont des publications de AT&T qui servent de "références" : version 1.1 en 86, 1.2 en 87, 2.0 en 89, 3.0 en 91 (cette dernière servant de base au travail du comité ANSI). Les rares différences existant entre les versions sont clairement mentionnées dans les résumés des différents chapitres.

Les exemples de programmes fournis en solution ont été testés avec le compilateur du logiciel Turbo C++ pour Windows, version 3.0.

TABLE DES MATIERES

CHAPITRE I :
INCOMPATIBILITES ENTRE C ET C++

RAPPELS

C++ est "presque" un sur-ensemble du C, tel qu'il est défini par la norme ANSI. Seules existent quelques "incompatibilités" dont nous vous rappelons ici les principales.

Déclarations de fonctions

En C++, toute fonction utilisée dans un fichier source doit obligatoirement avoir fait l'objet :

- soit d'une déclaration sous forme d'un **prototype** (il précise à la fois le nom de la fonction, le type de ses arguments éventuels et le type de sa valeur de retour), comme dans cet exemple :

```
float fexp (int, double, char *) ;
```

- soit d'une définition préalable au sein du même fichier source (ce dernier cas étant d'ailleurs peu conseillé, dans la mesure où des problèmes risquent d'apparaître dès lors qu'on sépare ladite fonction du fichier source en question).

En C, une fonction pouvait ne pas être déclarée (auquel cas, on considérait, par défaut, que sa valeur de retour était de type *int*), ou encore déclarée partiellement (sans fournir le type de ses arguments), comme dans :

```
float fexp () ;
```

Fonctions sans arguments

En C++, une fonction sans argument se définit (au niveau de l'en-tête) et se déclare (au niveau du prototype) en fournissant une "liste d'arguments vide" comme dans :

```
float fct () ;
```

En C, on pouvait indifféremment utiliser cette notation ou faire appel au mot clé *void* comme dans :

```
float fct (void)
```

Fonctions sans valeur de retour

En C++, une fonction sans valeur de retour se définit (en-tête) et se déclare (prototype) obligatoirement à l'aide du mot clé *void* comme dans :

```
void fct (int, double) ;
```

En C, l'emploi du mot clé *void* était, dans ce cas, facultatif.

Le qualificatif const

En C++, un symbole global déclaré avec le qualificatif *const* :

- a une portée limitée au fichier source concerné, tandis qu'en C il pouvait éventuellement être utilisé dans un autre fichier source (en utilisant le mot clé *extern*),

- peut être utilisé dans une "expression constante" (expression calculable au moment de la compilation), alors qu'il ne pouvait pas l'être en C ; ce dernier point permet notamment d'utiliser de tels symboles pour définir la taille d'un tableau (en C, il fallait obligatoirement avoir recours à une "définition" de symboles par la directive *#define*).

Le type void *

En C++, un pointeur de type *void* *ne peut pas être converti "implicitement" lors d'une affectation en un pointeur d'un autre type ; la chose était permise en C. Bien entendu, en C++, il reste possible de faire appel à l'opérateur de "*cast*".

Exercice I.1

Enoncé

Quelles erreurs seront détectées par un compilateur C++ dans ce fichier source qui est accepté par un compilateur C ?

```
main()
{
   int a=10, b=20, c ;
   c = g(a, b) ;
   printf ("valeur de g(%d,%d) = %d", a, b, c) ;
}
g(int x, int y)
{
   return (x*x + 2*x*y + y*y) ;
}
```

Solution

1) La fonction g doit obligatoirement faire l'objet d'une déclaration (sous forme d'un prototype) dans la fonction *main*. Par exemple, on pourrait introduire (n'importe où avant l'appel de g) :

```
int g (int, int) ;
```

ou encore :

```
int g (int x, int y) ;
```

Rappelons que, dans ce dernier cas, les noms *x* et *y* sont fictifs : ils n'ont aucun rôle dans la suite et ils n'interfèrent nullement avec d'autres variables de même nom qui pourraient être déclarées dans la même fonction (ici *main*).

2) La fonction *printf* doit, elle aussi, comme toutes les fonctions C++ (le compilateur n'étant pas en mesure de distinguer les fonctions de la bibliothèque des "fonctions définies par l'utilisateur"), faire l'objet d'un prototype. Naturellement, il n'est pas nécessaire de l'écrire explicitement : il est obtenu par incorporation du fichier en-tête correspondant :

```
#include <stdio.h>
```

Notez que certains compilateurs C refusent déjà l'absence de prototype pour une fonction de la bibliothèque standard telle que *printf* (mais la norme ANSI n'imposait rien à ce sujet!).

Exercice I.2

Ecrire correctement en C ce programme qui est correct en C++ :

```
#include <stdio.h>
const int nb = 10 ;
const int exclus = 5 ;
main()
{
   int valeurs [nb] ;
   int i, nbval = 0 ;
   printf ("donnez %d valeurs :\n", nb) ;
   for (i=0 ; i<nb ; i++) scanf ("%d", &valeurs[i]) ;
   for (i=0 ; i<nb ; i++)
      switch (valeurs[i])
       { case exclus-1 :
         case exclus   :
         case exclus+1 : nbval++ ;
       }
   printf ("%d valeurs sont interdites", nbval) ;

}
```

Solution

En C, les symboles *nb* et *exclus* ne sont pas utilisables dans des "expressions constantes". Il faut donc les définir soit comme des variables, soit à l'aide d'une directive *#define* comme suit :

```
#include <stdio.h>
#define NB 10
#define EXCLUS 5
```

```
main()
{
   int valeurs [NB] ;
   int i ;
   int nbval=0 ;
   printf ("donnez %d valeurs :\n", NB) ;
   for (i=0 ; i<NB ; i++) scanf ("%d", &valeurs[i]) ;
   for (i=0 ; i<NB ; i++)
      switch (valeurs[i])
       { case EXCLUS-1 :
         case EXCLUS   :
         case EXCLUS+1 : nbval++ ;
       }
   printf ("%d valeurs sont interdites", nbval) ;

}
```

CHAPITRE II :
LES SPECIFICITES DE C++

RAPPELS

C++ dispose, par rapport au C ANSI, d'un certain nombre de spécificités qui ne sont pas véritablement axées sur la Programmation Orientée Objet.

Nouvelles possibilités d'entrées-sorties

C++ dispose de nouvelles facilités d'entrées-sorties. Bien qu'elles soient fortement liées à des aspects P.O.O. (surdéfinition d'opérateur en particulier), elles sont parfaitement utilisables en dehors de ce contexte. C'est tout particulièrement le cas des possibilités d'entrées-sorties conversationnelles (clavier, écran) qui remplacent avantageusement les fonctions *printf* et *scanf*. Ainsi :

cout << expression$_1$ << expression$_2$ << ..... << expression$_n$

affiche sur le "flot" *cout* (connecté par défaut à la sortie standard *stdout*) les valeurs des différentes expressions indiquées, suivant une présentation adaptée à leur type (depuis la version 2.0 de C++, on sait distinguer les attributs de signe et on peut afficher des valeurs de pointeurs). De même :

cin >> lvalue$_1$ >> lvalue$_2$ >> ..... >> lvalue$_n$

lit sur le "flot" *cin* (connecté par défaut à l'entrée standard *stdin*) des informations de l'un des types *char*, *short*, *int*, *long*, *float*, *double* ou *char** (depuis la version 2.0 de C++, on sait distinguer les attributs de signe ; en revanche, les pointeurs ne sont pas admis). Les conventions d'analyse des caractères lus sont comparables à celles de *scanf* avec cette principale différence que la lecture d'un caractère commence par sauter les séparateurs.

Nouvelle forme de commentaire

Les deux caractères // permettent d'introduire des "commentaires de fin de ligne" : tout ce qui suit ces caractères, jusqu'à la fin de la ligne, est considéré comme un commentaire.

Emplacement libre des déclarations

En C++, il n'est plus nécessaire de regrouper les déclarations en début de fonction ou en début de bloc. Il devient ainsi possible d'employer des expressions dans des initialisations comme dans cet exemple :

```
int n ;
   .....
n = ... ;
   .....
int q = 2*n - 1 ;
   .....
```

La transmission par référence[1]

En faisant précéder du symbole & le nom d'un argument dans l'en-tête (et dans le prototype) d'une fonction, on réalise une "transmission par référence". Cela signifie que les éventuelles modifications effectuées au sein de la fonction porteront sur l'argument effectif de l'appel et non plus sur une copie.

Les arguments par défaut

Dans la déclaration d'une fonction (prototype), il est possible de prévoir pour un ou plusieurs arguments (obligatoirement les derniers de la liste) des "valeurs par défaut" : elles sont indiquées par le signe =, à la suite du type de l'argument comme dans :

```
float fct (char, int = 10, float = 0.0) ;
```

Ces valeurs par défaut seront alors utilisées en cas d'appel de ladite fonction avec un nombre d'arguments inférieur à celui prévu. Par exemple, avec la précédente déclaration, l'appel *fct ('a')* sera équivalent à *fct ('a', 10, 0.0)* ; de même, l'appel *fct ('x', 12)* sera équivalent à *fct ('x', 12, 0.0)*. En revanche, l'appel *fct ()* sera illégal.

Surdéfinition de fonctions

En C++, il est possible, au sein d'un même programme, que plusieurs fonctions possèdent le même nom. Dans ce cas, lorsque le compilateur rencontre l'appel d'une telle fonction, il effectue le choix de la "bonne fonction" en tenant compte de la nature des arguments effectifs. Les règles utilisées dans une telle situation ont évolué avec les différentes versions de C++, généralement dans le sens d'un "affinement" : prise en compte de l'attribut de signe depuis la version 1.2, distinction des types *char*, *short* et *int* depuis la version 2.0, prise en compte de l'attribut *const* pour les pointeurs depuis la version 2.0 et pour les références depuis la version 3. D'une manière générale, si les règles utilisées par le compilateur pour sa recherche sont assez intuitives, leur énoncé précis est assez complexe et nous ne le rappellerons pas ici[2]. Signalons simplement qu'elles peuvent faire intervenir toutes les conversions usuelles (promotions numériques et conversions standards - éventuellement "dégradantes" depuis la version 2.0), ainsi que les "conversions définies par l'utilisateur" en cas d'argument de type classe, à condition qu'aucune ambiguïté n'apparaisse.

---

[1] La notion de référence est plus générale que celle de transmission d'arguments. C'est toutefois dans cette dernière situation qu'elle est la plus employée.

[2] On le trouvera, par exemple, dans l'annexe A de notre ouvrage "Programmer en langage C++", ou dans le chapitre V de "Programmer en Turbo C++", publiés également par les Editions EYROLLES.

Remarque :

Avant la version 2, toute définition ou utilisation d'une fonction "ordinaire" (c'est-à-dire autre qu'une "fonction membre d'une classe") surdéfinie devait être signalée au compilateur par une déclaration préalable de la forme : *overload nom_fonction*.

Gestion dynamique de la mémoire

En C++, les fonctions *malloc, calloc*... et *free* sont remplacées avantageusement par les "opérateurs" **new** et **delete**.

Si *type* représente la description d'un type absolument quelconque et si n représente une expression d'un type entier[3]

new type [n]

alloue l'emplacement nécessaire pour **n éléments** du type indiqué et fournit en résultat un pointeur (de type *type* *) sur le premier élément. On obtient un pointeur nul si l'allocation a échoué. L'indication *n* est facultative : avec **new type**, on obtient un emplacement pour **un élément** du type indiqué.

delete adresse

libère un emplacement préalablement alloué par *new* à l'adresse indiquée. Il n'est pas nécessaire de répéter le nombre d'éléments, du moins lorsqu'il ne s'agit pas d'objets, même lorsque celui-ci est différent de 1. Le cas des tableaux d'objets est examiné dans le chapitre V.

En outre, vous pouvez définir une fonction de votre choix et demander qu'elle soit appelée en cas de manque de mémoire. Il vous suffit d'en transmettre l'adresse en argument à la fonction **set_new_handler**.

Les fonctions "en ligne"

Une fonction "en ligne" (on dit aussi "développée") est une fonction dont les instructions sont incorporées par le compilateur (dans le module objet correspondant) à chaque appel. Cela évite la perte de temps nécessaire à un appel usuel (changement de contexte, copie des valeurs des arguments sur la "pile", ...) ; en revanche, les instructions en question sont générées plusieurs fois. Les fonctions "en ligne" offrent le même intérêt que les macros, sans présenter de risques "d'effets de bord". Une fonction en ligne est nécessairement définie en même temps qu'elle est déclarée (elle ne peut plus être compilée séparément) et son en-tête est précédé du mot clé *inline* comme dans :

```
inline fct ( ...)
{
   .....
}
```

Exercice II.1

---

[3] Généralement long ou unsigned long.

Enoncé

Ecrire le programme suivant (correct en C comme en C++), en ne faisant appel qu'aux nouvelles possibilités d'entrées-sorties de C++, donc en remplaçant les appels à *printf* et *scanf* :

```
#include <stdio.h>
main()
{
   int n ; float x ;
   printf ("donnez un entier et un flottant\n") ;
   scanf ("%d %e", &n, &x) ;
   printf ("le produit de %d par %e\n'est : %e", n, x, n*x) ;
}
```

Solution

```
#include <iostream.h> // pour pouvoir utiliser les flots cin et cout
                      // suivant les implémentations, on pourra rencontrer
                      //    h, hxx ...
main()
{
   int n ; float x ;
   cout << "donnez un entier et un flottant\n" ;
   cin >> n >> x ;
   cout << "le produit de " << n << " par " << x << "\n'est : " << n*x ;
}
```

Exercice II.2

Enoncé

Ecrire une fonction permettant d'échanger les contenus de 2 variables de type *int* fournies en argument :

- en transmettant l'adresse des variables concernées (seule méthode utilisable en C),
- en utilisant la transmission par référence.

Dans les deux cas, on écrira un petit programme d'essai (*main*) de la fonction.

Solution

a) avec la transmission des adresses des variables

```
#include <iostream.h>
main()
{
```

```
  void echange (int *, int *) ;        // prototype de la fonction echange
  int n=15, p=23 ;
  cout << "avant : " << n << " " << p << "\n" ;
  echange (&n, &p) ;
  cout << "après : " << n << " " << p << "\n" ;
}
void echange (int * a, int * b)
{ int c ;   // pour la permutation
  c = *a ;
  *a = *b ;
  *b = c ;
}
```

b) Avec une transmission par référence

```
#include <iostream.h>
main()
{
  void echange (int &, int &) ;        // prototype de la fonction echange
  int n=15, p=23 ;
  cout << "avant : " << n << " " << p << "\n" ;
  echange (n, p) ;                     // attention n et non &n, p et non &p
  cout << "après : " << n << " " << p << "\n" ;
}
void echange (int & a, int & b)
{ int c ;   // pour la permutation
  c = a ;
  a = b ;
  b = c ;
}
```

Exercice II.3

Enoncé

Soit le modèle de structure suivant :

```
struct essai
{ int n ;
  float x ;
} ;
```

Ecrire une fonction nommée *raz* permettant de remettre à zéro les 2 champs d'une structure de ce type, transmise en argument :

- par adresse,
- par référence.

Dans les deux cas, on écrira un petit programme d'essai de la fonction ; il affichera les valeurs d'une structure de ce type, après appel de ladite fonction.

Solution

a) Avec une transmission d'adresse

```
#include <iostream.h>
struct essai
{ int n ;
  float x ;
} ;
void raz (struct essai * ads)
{ ads->n = 0 ;         // ou encore (*ads).n = 0 ;
  ads->x = 0.0 ;       // ou encore (*ads).x = 0.0 ;
}
main()
{ struct essai s ;
  raz (&s) ;
  cout << "valeurs après raz : " << s.n << " " << s.x ;
}
```

b) Avec une transmission par référence

```
#include <iostream.h>
struct essai
{ int n ;
  float x ;
} ;
void raz (struct essai & s)
{ s.n = 0 ;
  s.x = 0.0 ;
}
main()
{ struct essai s ;
  raz (s) ;               // notez bien s et non &s !
  cout << "valeurs après raz : " << s.n << " " << s.x ;
}
```

Exercice II.4

Enoncé

Soient les déclarations (C++) suivantes :

```
overload fct             // inutile depuis la version 2.0
int fct (int) ;          // fonction I
int fct (float) ;        // fonction II
void fct (int, float) ;  // fonction III
void fct (float, int) ;  // fonction IV
```

```
int n, p ;
float x, y ;
char c ;
double z ;
```

Les appels suivants sont-ils corrects et, si oui, quelles seront les fonctions effectivement appelées et les conversions éventuellement mises en place ?

```
a)  fct (n) ;
b)  fct (x) ;
c)  fct (n, x) ;
d)  fct (x, n) ;
e)  fct (c) ;
f)  fct (n, p) ;
g)  fct (n, c) ;
h)  fct (n, z) ;
i)  fct (z, z) ;
```

Solution

Les cas a, b, c et d ne posent aucun problème, quelle que soit la version de C++ utilisée. Il y a respectivement appel des fonctions I, II, III et IV, sans qu'aucune conversion d'argument ne soit nécessaire.

e) Appel de la fonction I, après conversion de la valeur de c en *int*.

f) Appel incorrect, compte tenu de son ambiguïté ; deux possibilités existent en effet : conserver n, convertir p en *fbat* et appeler la fonction III ou, au contraire, convertir n en *fbat*, conserver p et appeler la fonction IV.

g) Appel de la fonction III, après conversion de c en *fbat*.

h) Appel de la fonction III, après conversion (dégradante) de z en *fbat* (du moins, depuis la version 2.0, car auparavant, on aboutissait à une erreur de compilation liée à l'ambiguïté de l'appel).

i) Appel incorrect, compte tenu de son ambiguïté ; deux possibilités existent en effet : convertir le premier argument en *fbat* et le second en *int* et appeler la fonction III ou, au contraire, convertir le premier argument en *int* et le second en *fbat* et appeler la fonction IV. Notez que, dans le deux cas, il s'agit de conversions dégradantes (dans les versions antérieures à la 2.0, ces dernières n'étaient pas acceptées, et l'appel était rejeté parce qu'alors aucune correspondance n'était trouvée !).

Exercice II.5

Enoncé

Ecrire plus simplement en C++ les instructions suivantes, en utilisant les opérateurs *new* et *delete* :

```
int * adi ;
double * add ;
   .....
adi = malloc (sizeof (int) ) ;
add = malloc (sizeof (double) * 100 ) ;
```

Solution

```
int * adi ;
double * add ;
   .....
adi = new int ;
add = new double [100] ;
```

On peut éventuellement tenir compte des possibilités de déclarations dynamiques offertes par C++ (c'est-à-dire que l'on peut introduire une déclaration à n'importe que lemplacement d'un programme), et écrire, par exemple :

```
int * adi = new int ;
double * add = new double [100] ;
```

Exercice II.6

Enoncé

Ecrire plus simplement en C++, en utilisant les spécificités de ce langage, les instructions C suivantes :

```
double * adtab ;
int nval ;
   .....
printf ("combien de valeurs ? ") ;
scanf ("%d", &nval) ;
adtab = malloc (sizeof (double) * nval) ;
```

Solution

```
double * adtab ;
int nval ;
   .....
cout << "combien de valeurs ? " ;
cin >> nval ;
adtab = new double [nval] ;
```

Exercice II.7

Enoncé

Ecrire un programme allouant des emplacements pour des tableaux d'entiers dont la taille est fournie en donnée. Les allocations se poursuivront jusqu'à ce que l'on aboutisse à un débordement mémoire. L'exécution se présentera ainsi :

```
Taille souhaitée ? 6000
Allocation bloc numéro : 1
Allocation bloc numéro : 2
Allocation bloc numéro : 3
Allocation bloc numéro : 4
Allocation bloc numéro : 5
Mémoire insuffisante - arrêt exécution
```

On proposera deux solutions, l'une ne faisant pas appel à *set_new_handler* (il faudra donc vérifier la valeur fournie par *new*), l'autre faisant appel à *set_new_handler*.

Solution

a) Sans utiliser set_new_handler

On se contente de répéter l'allocation par *new* de blocs ayant la taille indiquée, jusqu'à ce que l'adresse fournie en retour soit nulle (*NULL*) :

```
#include <stdlib.h>         // pour exit
#include <iostream.h>
main()
{
   long taille ;
   int * adr ;
   int nbloc ;

   cout << "Taille souhaitée ? " ;
   cin >> taille ;

   for (nbloc=1 ; ; nbloc++)
      { adr = new int [taille] ;
        if (adr) cout << "Allocation bloc numéro : " << nbloc << "\n" ;
            else { cout << "Mémoire insuffisante - arrêt exécution \n " ;
                   exit (1) ;
                 }
      }
}
```

b) En utilisant set_new_handler

Cette fois, on commence par appeler la fonction *set_new_handler*, à laquelle on précise l'adresse d'une fonction (nommée ici *deborde*). Cette dernière sera appelée automatiquement dès qu'une allocation mémoire aura échoué, de sorte qu'il n'est plus nécessaire de prévoir de test à chaque appel de *new*.

```
#include <stdlib.h>          // pour exit
#include <iostream.h>
main()
{
   void deborde () ;        // proto fonction appelée en cas manque mémoire
   void set_new_handler ( void (*) () ) ; // proto de set_new_handler
   set_new_handler (&deborde) ;           // précise quelle sera la fonction
                                          // appelée en cas de manque mémoire
   long taille ;
   int * adr ;
   int nbloc ;

   cout << "Taille souhaitée ? " ;
   cin >> taille ;
   for (nbloc=1 ; ; nbloc++)
      { adr = new int [taille] ;
        cout << "Allocation bloc numéro : " << nbloc << "\n" ;
      }
}


void deborde ()       // fonction appelée en cas de manque mémoire
{
   cout << "Mémoire insuffisante - arrêt exécution \n " ;
   exit (1) ;
}
```

Exercice II.8

Enoncé

a) Transformer le programme suivant pour que la fonction *fct* devienne une fonction "en ligne".

```
#include <iostream.h>
main()
{
   int fct (char, int) ;            // déclaration (prototype) de fct
   int n = 150, p ;
   char c = 's' ;
   p = fct ( c , n) ;
   cout << "fct (\'" << c << "\', " << n << ") vaut : " << p ;
}
int fct (char c, int n)             // définition de fct
{
   int res ;
   if (c == 'a')      res = n + c ;
   else if (c == 's') res = n - c ;
   else               res = n * c ;
   return res ;
}
```

b) Comment faudrait-il procéder si l'on souhaitait que la fonciton *fct* soit compilée séparément?

Solution

a) Nous devons donc d'abord déclarer (et définir en même temps) la fonction *fct* comme une fonction "en ligne". Le programme *main* s'écrit de la même manière, si ce n'est que la déclaration de *fct* n'y est plus nécessaire puisqu'elle apparaît déjà auparavant.

```
#include <iostream.h>
inline int fct (char c, int n)
{   int res ;
    if (c == 'a')      res = n + c ;
    else if (c == 's') res = n - c ;
    else               res = n * c ;
    return res ;
}
main ()
{   int n = 150, p ;
    char c = 's' ;
    p = fct (c, n) ;
    cout << "fct (\'" << c << "\', " << n << ") vaut : " << p ;
}
```

b) Il s'agit en fait d'une question piège. En effet, la fonction *fct* étant "en ligne", elle ne peut plus être compilée séparément. Il est cependant possible de la conserver dans un fichier d'extension h et d'incorporer simplement ce fichier par *#include* pour compiler le *main*. Cette démarche se rencontrera d'ailleurs fréquemment dans le cas de classes comportant des fonctions "en ligne". Dans ce cas, dans un fichier d'extension h, on trouvera la déclaration de la classe en question, à l'intérieur de laquelle apparaîtront les "déclarations-définitions" des fonctions "en ligne".

CHAPITRE III :
NOTIONS DE CLASSE,
CONSTRUCTEUR ET DESTRUCTEUR

RAPPELS

Les possibilités de Programmation Orientée Objet de C++ reposent sur le concept de classe. Une classe est la généralisation de la notion de type défini par l'utilisateur, dans lequel se trouvent associées à la fois des données (on parle de "membres donnée") et des fonctions (on parle de "fonctions membre" ou de méthodes). En P.O.O. pure, les données sont "encapsulées", ce qui signifie que leur accès ne peut se faire que par le biais des méthodes. C++ vous autorise à n'encapsuler qu'une partie seulement des données d'une classe[1].

Déclaration et définition d'une classe

La **déclaration** d'une classe précise quels sont les membres (données ou fonctions) publics (c'est-à-dire accessibles à l'utilisateur de la classe) et quels sont les membres privés (inaccessibles à l'utilisateur de la classe). On utilise pour cela le mot clé **public**, comme dans cet exemple dans lequel la classe *point* comporte deux membres donnée privés *x* et *y* et trois fonctions membres publiques *initialise*, *deplace* et *affiche* :

```
       /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
    int x ;
    int y ;
              /* déclaration des membres publics */
public :
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
```

La **définition** d'une classe consiste à fournir les définitions des fonctions membre. Dans ce cas, on indique le nom de la classe correspondante, à l'aide de l'opérateur de résolution de portée (::). Au sein de la

[1] En toute rigueur, C++ vous permet également d'associer des fonctions membre à des structures, des unions ou des énumérations. Dans ce cas, toutefois, aucune encapsulation n'est possible (ce qui revient à dire que tous les membres sont "publics").

définition même, les membres (privés ou publics - données ou fonctions) sont directement accessibles sans qu'il soit nécessaire de préciser le nom de la classe. Voici, par exemple, ce que pourrait être la définition de la fonction *initialise* de la classe précédente :

```
void point::initialise (int abs, int ord)
{
   x = abs ; y = ord ;
}
```

Ici, x et y représentent implicitement les membres x et y d'un objet de la classe *point*.

Remarque :

Depuis la version 1.2 de C++, on peut également utiliser le mot clé **private**. Il n'est alors plus nécessaire de scinder en deux parties (publique et privée) les membres d'une classe. Une déclaration telle que la suivante est tout à fait envisageable :

```
class X
{     private :
       ...
      public :
       ...
      private :
       ...
      public :
       ...
} ;
```

Utilisation d'une classe

On déclare un "objet" d'un type classe donné en faisant précéder son nom du nom de la classe, comme dans l'instruction suivante qui déclare deux objets a et b de type *point* :

```
point a, b ;
```

On peut accéder à n'importe quel membre public (donnée ou fonction) d'une classe en utilisant l'opérateur . (point). Par exemple :

```
a.initialise (5, 2) ;
```

appelle la fonction membre *initialise* de la classe à laquelle appartient l'objet *a*, c'est-à-dire, ici, la classe *point*.

Affectation entre objets

C++ autorise l'affectation d'un objet d'un type donné à un autre objet de même type. Dans ce cas, il y a (tout naturellement) recopie des valeurs des champs de données (qu'ils soient publics ou privés). Toutefois, si, parmi ces champs, se trouvent des pointeurs, les emplacements pointés ne seront pas soumis à cette recopie. Si un tel effet est nécessaire (et il le sera souvent!), il ne pourra être obtenu qu'en "surdéfinissant" l'opérateur d'affectation pour la classe concernée (voyez le chapitre consacré à la surdéfinition d'opérateurs).

Constructeur et destructeur

Une fonction membre portant le même nom que sa classe se nomme un **constructeur**. Dès lors qu'une classe comporte un constructeur (au moins un), il n'est plus possible de déclarer un objet du type correspondant, sans fournir des valeurs pour les arguments requis par ce constructeur (sauf si ce dernier ne possède aucun argument). Le constructeur est appelé **après** l'allocation de l'espace mémoire destiné à l'objet.

Par définition même, un constructeur ne renvoie pas de valeur (aucune indication de type, pas même *void*, ne doit figurer devant sa déclaration ou sa définition).

Une fonction membre portant le même nom que sa classe, précédé du symbole tilda (~), se nomme un **destructeur**. Le destructeur est appelé avant la libération de l'espace mémoire associé à l'objet. Par définition même, un destructeur ne peut pas comporter d'arguments et il ne renvoie pas de valeur (aucune indication de type ne doit être prévue).

Membres donnée statiques

Un membre donnée déclaré avec l'attribut **static** est partagé par tous les objets de la même classe. Il existe même lorsque aucun objet de cette classe n'a été déclaré. Un membre donnée statique est initialisé par défaut à zéro (comme toutes les variables statiques!). Il peut être initialisé explicitement, à l'extérieur de la classe (même s'il est privé), en utilisant l'opérateur de résolution de portée (::) pour spécifier sa classe.

Exploitation d'une classe

En pratique, à l'utilisateur d'une classe (on dit souvent le "client"), on fournira :

- un fichier en-tête contenant la déclaration de la classe : l'utilisateur l'employant devra l'inclure dans tout programme faisant appel à la classe en question,
- un module objet résultant de la compilation du fichier source contenant la définition de la classe, c'est-à-dire la définition de ses fonctions membre.

Exercice III.1

Enoncé

Réaliser une classe *point* permettant de manipuler un point d'un plan. On prévoira :

- un constructeur recevant en arguments les coordonnées (*float*) d'un point,
- une fonction membre *deplace* effectuant une translation définie par ses deux arguments (*float*),
- une fonction membre *affiche* se contentant d'afficher les coordonnées cartésiennes du point.

Les coordonnées du point seront des membres donnée privés.

On écrira séparément :

- un fichier source constituant la *déclaration* de la classe,
- un fichier source correspondant à sa *définition*.

Ecrire, par ailleurs, un petit programme d'essai (*main*) déclarant un point, l'affichant, le déplaçant et l'affichant à nouveau.

Solution

a) Fichier source (nommé point.h) contenant la déclaration de la classe

```
          /*       fichier POINT1.H         */
          /* déclaration de la classe point */
class point
{
   float x, y ;                  // coordonnées (cartésiennes) du point
 public :
   point (float, float) ;        // constructeur
   void deplace (float, float) ; // déplacement
   void affiche () ;             // affichage
} ;
```

b) Fichier source contenant la définition de la classe

```
          /* définition de la classe point */
#include "point1.h"
#include <iostream.h>
point::point (float abs, float ord)
{  x = abs ; y = ord ;
}
void point::deplace (float dx, float dy)
{  x = x + dx ; y = + dy ;
}
void point::affiche ()
{  cout << "Mes coordonnées cartésiennes sont " << x << " " << y << "\n" ;
}
```

Notez que, pour compiler ce fichier source, il est nécessaire d'inclure le fichier source, nommé ici *point1.h*, contenant la déclaration de la classe *point*.

c) Exemple d'utilisation de la classe

```
     /* exemple d'utilisation de la classe point */
#include <iostream.h>
#include "point1.h"
main ()
{
   point p (1.25, 2.5) ;  // construction d'un point de coordonnées 1.25 2.5
   p.affiche () ;         // affichage de ce point
   p.deplace (2.1, 3.4) ; // déplacement de ce point
   p.affiche () ;         // nouvel affichage
}
```

Bien entendu, pour pouvoir exécuter ce programme, il sera nécessaire d'introduire, lors de l'édition de liens, le module objet résultant de la compilation du fichier source contenant la définition de la classe *point*.

Notez que, généralement, le fichier *point.h* contiendra des directives conditionnelles de compilation, afin d'éviter les risques d'inclusion multiple. Par exemple, on pourra procéder ainsi :

```
#ifndef POINT_H
#define POINT_H
     .....

     déclaration de la classe.....

     .....

#endif
```

Exercice III.2

Enoncé

Réaliser une classe *point*, analogue à la précédente, mais ne comportant pas de fonction *affiche*. Pour respecter le principe d'encapsulation des données, prévoir deux fonctions membre publiques (nommées *abscisse* et *ordonnee*) fournissant en retour respectivement l'abscisse et l'ordonnée d'un point. Adapter le petit programme d'essai précédent pour qu'il fonctionne avec cette nouvelle classe.

Solution

Il suffit d'introduire deux nouvelles fonctions membre *abscisse* et *ordonnée* et de supprimer la fonction *affiche*. La nouvelle déclaration de la classe est alors :

```
           /*       fichier POINT2.H         */
           /* déclaration de la classe point */
class point
{
   float x, y ;                  // coordonnées (cartésiennes) du point
 public :
   point (float, float) ;        // constructeur
   void deplace (float, float) ; // déplacement
   float abscisse () ;           // abscisse du point
   float ordonnee () ;           // ordonnée du point
} ;
```

Voici sa nouvelle définition :

```
           /* définition de la classe point */
#include "point2.h"
#include <iostream.h>
point::point (float abs, float ord)
{  x = abs ; y = ord ;
}
```

```
void point::deplace (float dx, float dy)
{  x = x + dx ; y = + dy ;
}
float point::abscisse ()
{ return x ;
}
float point::ordonnee ()
{ return y ;
}
```

Et le nouveau programme d'essai :

```
      /* exemple d'utilisation de la classe point */
#include <iostream.h>
#include "point2.h"
main ()
{
   point p (1.25, 2.5) ;                                  // construction
                                                          // affichage
   cout << "Coordonnées cartésiennes : " << p.abscisse () << " "
                                        << p.ordonnee () << "\n" ;
   p.deplace (2.1, 3.4) ;                                 // déplacement
                                                          // affichage
   cout << "Coordonnées cartésiennes : " << p.abscisse () << " "
                                        << p.ordonnee () << "\n" ;
}
```

Discussion

Cet exemple montre comment il est toujours possible de respecter le principe d'encapsulation en introduisant ce que l'on nomme des "fonctions d'accès". Il s'agit de fonctions membre destinées à accéder (aussi bien en consultation - comme ici - qu'en modification) aux membres privés. L'intérêt de leur emploi (par rapport à un accès direct aux données qu'il faudrait alors rendre publiques) réside dans la souplesse de modification de l'implémentation de la classe qui en découle. Ainsi, ici, il est tout à fait possible de modifier la manière dont un point est représenté (par exemple, en utilisant ses coordonnées polaires plutôt que ses coordonnées cartésiennes), sans que l'utilisateur de la classe n'ait à se soucier de cet aspect. C'est d'ailleurs ce que vous montrera l'exercice III.4 ci-après.

Exercice III.3

Enoncé

Ajouter à la classe précédente (comportant un constructeur et 3 fonctions membre *deplace*, *abscisse* et *ordonnee*) de nouvelles fonctions membre :

- *homothetie* qui effectue une homothétie dont le rapport est fourni en argument,
- *rotation* qui effectue une rotation dont l'angle est fourni en argument,
- *rho* et *theta* qui fournissent en retour les **coordonnées polaires** du point.

Solution

La déclaration de la nouvelle classe *point* découle directement de l'énoncé :

```
            /*       fichier POINT3.H         */
            /* déclaration de la classe point */
   class point
   {
      float x, y ;                  // coordonnées (cartésiennes) du point
    public :
      point (float, float) ;        // constructeur
      void deplace (float, float) ; // déplacement
      void homothetie (float) ;     // homothetie
      void rotation (float) ;       // rotation
      float abscisse () ;           // abscisse du point
      float ordonnee () ;           // ordonnée du point
      float rho () ;                // rayon vecteur
      float theta () ;              // angle
   } ;
```

Sa définition mérite quelques remarques. En effet, si *homothetie* ne présente aucune difficulté, la fonction membre *rotation*, quant à elle, nécessite une transformation intermédiaire des coordonnées cartésiennes du point en coordonnées polaires. De même, la fonction membre *rho* doit calculer le rayon vecteur d'un point dont on connaît les coordonnées cartésiennes tandis que la fonction membre *theta* doit calculer l'angle d'un point dont on connaît les coordonnées cartésiennes.

Le calcul de rayon vecteur étant simple, nous l'avons laissé figurer dans les deux fonctions concernées (*rotation* et *rho*). En revanche, le calcul d'angle a été réalisé par ce que nous nommons une "fonction de service", c'est-à-dire une fonction qui n'a d'intérêt que dans la définition de la classe elle-même. Ici, il s'agit d'une fonction indépendante mais, bien entendu, on peut prévoir des fonctions de service sous forme de fonctions membre (elles seront alors généralement privées).

Voici finalement la définition de notre classe *point* :

```
        /**************** déclarations de service *****************/
   #include "point3.h"
   #include <iostream.h>
   #include <math.h>                     // pour sqrt et atan
   const float pi = 3.141592653 ;        // valeur de pi
   float angle (float, float) ;          // fonction de service (non membre)

        /*************** définition des fonctions membre **********/
   point::point (float abs, float ord)
   {  x = abs ; y = ord ;
   }
   void point::deplace (float dx, float dy)
   {  x += dx ; y += dy ;
   }
   void point::homothetie (float hm)
   {  x *= hm ; y *= hm ;
   }
```

```
void point::rotation (float th)
{  float r = sqrt (x*x + y*y) ;       // passage en
   float t = angle (x, y) ;          //   coordonnées polaires
   t += th ;                         // rotation th
   x = r * cos (t) ;                 // retour en
   y = r * sin (t) ;                 //   coordonnées cartésiennes
}

float point::abscisse ()
{ return x ;
}
float point::ordonnee ()
{ return y ;
}
float point::rho ()
{ return sqrt (x*x + y*y) ;
}
float point::theta ()
{ return angle (x, y) ;

}        /********** définition des fonctions de service **********/
/* fonction de calcul de l'angle correspondant aux coordonnées     */
/*           cartésiennes fournies en argument                     */
/* On choisit une détermination entre -pi et +pi (0 si x=0)       */

float angle (float x, float y)
{  float a = x ? atan (y/x) : 0 ;
   if (y<0) if (x>=0) return a + pi ;
                 else return a - pi ;
   return a ;
}
```

Exercice III.4

Enoncé

Modifier la classe *point* précédente, de manière à ce que les données (privées) soient maintenant les coordonnées polaires d'un point, et non plus ses coordonnées cartésiennes. On évitera de modifier la déclaration des membres publics, de sorte que l'interface de la classe (ce qui est visible pour l'utilisateur) ne change pas.

Solution

La déclaration de la nouvelle classe découle directement de l'énoncé :

```
/*       fichier POINT4.H         */
/* déclaration de la classe point */
```

```
class point
{
   float r, t ;                  // coordonnées (polaires) du point
 public :
   point (float, float) ;        // constructeur
   void deplace (float, float) ; // déplacement
   void homothetie (float) ;     // homothétie
   void rotation (float) ;       // rotation
   float abscisse () ;           // abscisse du point
   float ordonnee () ;           // ordonnée du point
   float rho () ;                // rayon vecteur
   float theta () ;              // angle
} ;
```

En ce qui concerne sa définition, il est maintenant nécessaire de remarquer :

- que le constructeur reçoit toujours en argument les coordonnées cartésiennes d'un point ; il doit donc opérer les transformations appropriées,

- que la fonction *deplace* reçoit un déplacement exprimé en coordonnées cartésiennes ; il faut donc tout d'abord déterminer les coordonnées cartésiennes du point après déplacement, avant de repasser en coordonnées polaires.

En revanche, les fonctions *homothetie* et *rotation* s'expriment très simplement.

Voici la définition de notre nouvelle classe (nous avons fait appel à la même "fonction de service" *angle* que dans l'exercice précédent) :

```
#include "point4.h"
#include <iostream.h>
#include <math.h>                    // pour cos, sin, sqrt et atan
const int pi = 3.141592635 ;         // valeur de pi



    /********** définition des fonctions de service **********/
/* fonction de calcul de l'angle correspondant aux coordonnées     */
/*           cartésiennes fournies en argument                     */
/* On choisit une détermination entre -pi et +pi (0 si x=0)        */
float angle (float x, float y)
{  float a = x ? atan (y/x) : 0 ;
   if (y<0) if (x>=0) return a + pi ;
            else return a - pi ;
   return a ;
}
    /********** définition des fonctions membre ****************/
point::point (float abs, float ord)
{  r = sqrt (abs*abs + ord*ord) ;
   t = atan (ord/abs) ;
}
void point::deplace (float dx, float dy)
{  float x = r * cos (t) + dx ;   // nouvelle abscisse
   float y = r * sin (t) + dy ;   // nouvelle ordonnée
   r = sqrt (x*x + y*y) ;
   t = angle (x, y) ;
}
void point::homothetie (float hm)
```

```
{  r *= hm ;
}
void point::rotation (float th)
{  t += th ;
}
float point::abscisse ()
{ return r * cos (t) ;
}
float point::ordonnee ()
{ return r * sin (t) ;
}
float point::rho ()
{ return r ;
}
float point::theta ()
{ return t ;
}
```

Exercice III.5

Enoncé

Soit la classe *point* créée dans l'exercice III.1, et dont la déclaration était la suivante :

```
class point
{
   float x, y ;
  public :
   point (float, float) ;
   void deplace (float, float) ;
   void affiche () ;
}
```

Adapter cette classse, de manière à ce que la fonction membre *affiche* fournisse, en plus des coordonnées du point, le nombre d'objets de type *point*.

Solution

Il faut donc définir un compteur du nombre d'objets existant à un moment donné. Ce compteur doit être incrémenté à chaque création d'un nouvel objet, donc par le constructeur *point*. De même, il doit être décrémenté à chaque destruction d'un objet, donc par le destructeur de la classe *point* ; il faudra donc ajouter ici une fonction membre nommée ~ *point*.

Quant au compteur proprement dit, nous pourrions certes en faire une variable globale, définie par exemple, en même temps que la classe ; cette démarche présente toutefois des risques d'effets de bord (modification

accidentelle de la valeur de cette variable, depuis n'importe quel programme utilisateur). Il est plus judicieux d'en faire un membre privé statique[2].

Voici la nouvelle déclaration de notre classe *point* :

```
          /*       fichier POINT5.H         */
          /* déclaration de la classe point */
class point
{
   static nb_pts ;               // compteur du nombre d'objets créés
   float x, y ;                  // coordonnées (cartésiennes) du point
 public :
   point (float, float) ;        // constructeur
   ~point () ;                   // destructeur
   void deplace (float, float) ; // déplacement
   void affiche () ;             // affichage
} ;
```

Et voici sa nouvelle définition :

```
#include "point5.h"
#include <iostream.h>
point::point (float abs, float ord)       // constructeur
{  x = abs ; y = ord ;
   nb_pts++ ;                             //    actualisation nb points
}
point::~point ()                          // destructeur
{  nb_pts-- ;                             //    actualisation nb points
}
void point::deplace (float dx, float dy)
{  x = x + dx ; y = + dy ;
}
void point::affiche ()
{  cout << "Je suis un point parmi " << nb_pts
    << " de coordonnées "<< x << " " << y << "\n" ;
}
```

Remarque :

Il pourra être judicieux de munir notre classe *point* d'une fonction membre fournissant le nombre d'objets de type *point* existant à un moment donné. C'est ce que nous vous proposerons dans un exercice du prochain chapitre.

Exercice III.6

---

[2] En toute rigueur, il reste toutefois possible d'initialiser un tel membre depuis l'extérieur de la classe.

Enoncé

Réaliser une classe nommée *set_char* permettant de manipuler des ensembles de caractères. On devra pouvoir réaliser sur un tel ensemble les opérations classiques suivantes : lui ajouter un nouvel élément, connaître son "cardinal" (nombre d'éléments), savoir si un caractère donné lui appartient.

Ici, on n'effectuera **aucune allocation dynamique d'emplacements mémoire**. Il faudra donc prévoir, en membre donnée, un tableau de taille fixe.

Ecrire, en outre, un programme (*main*) utilisant la classe *set_char* pour déterminer le nombre de caractères différents contenus dans un mot lu en donnée.

Solution

Compte tenu des contraintes imposées par l'énoncé (pas de gestion dynamique), une solution consiste à prévoir un tableau dans lequel un élément de rang i précise si le caractère de code i appartient ou non à l'ensemble. Notez qu'il est nécessaire que i soit positif ou nul ; on travaillera donc toujours sur des caractères non signés. La taille du tableau doit être égale au nombre de caractères qu'il est possible de représenter dans une implémentation donnée (généralement 256).

Le reste de la déclaration de la classe découle de l'énoncé.

```
          /*         fichier SETCHAR1.H        */
          /* déclaration de la classe set_char */
#define N_CAR_MAX 256       // on pourrait utiliser UCHAR_MAX défini
                            //    dans <limits.h>

class set_char
{
   unsigned char ens [N_CAR_MAX] ;
                     // tableau des indicateurs (présent/absent)
                     // pour chacun des caractères possibles
 public :
   set_char () ;                    // constructeur
   void ajoute (unsigned char) ;    // ajout d'un élément
   int appartient (unsigned char) ; // appartenance d'un élément
      int cardinal () ;             // cardinal de l'ensemble
} ;
```

La définition de la classe en découle assez naturellement :

```
     /* définition de la classe set_char */
#include "setchar1.h"
set_char::set_char ()
{  int i ;
   for (i=0 ; i<N_CAR_MAX ; i++) ens[i] = 0 ;
}

void set_char::ajoute (unsigned char c)
{   ens[c] = 1 ;
}

int set_char::appartient (unsigned char c)
```

```
{   return ens[c] ;
}

int set_char::cardinal ()
{   int i, n ;
    for (i=0, n=0 ; i<N_CAR_MAX ; i++) if (ens[i]) n++ ;
    return n ;
}
```

Il en va de même pour le programme d'utilisation :

```
     /* utilisation de la classe set_char */
#include <iostream.h>
#include <string.h>
#include "setchar1.h"
main()
{  set_char ens ;
   char mot [81] ;
   cout << "donnez un mot " ;
   cin >> mot ;
   int i ;
   for (i=0 ; i<strlen(mot) ; i++) ens.ajoute (mot[i]) ;
   cout << "il contient " << ens.cardinal () << " caractères différents" ;
}
```

Remarque :

Si l'on avait déclaré de type *char* les arguments de *ajoute* et *appartient*, on aurait alors pu aboutir, soit au type *unsigned char*, soit au type *signed char*, selon l'environnement utilisé. Dans le dernier cas, on aurait couru le risque de transmettre, à l'une des fonctions membre citées, une valeur négative, et partant d'accéder à l'extérieur du tableau *ens*.

Discussion

Le tableau *ens [N_CHAR_MAX]* occupe un octet par caractère ; chacun de ces octets ne prend que l'une des valeurs 0 ou 1 ; on pourrait économiser de l'espace mémoire en prévoyant seulement 1 bit par caractère. Les fonctions membre y perdraient toutefois en simplicité, ainsi qu'en vitesse.

Bien entendu, beaucoup d'autres implémentations sont possibles ; c'est ainsi, par exemple, que l'on pourrait fournir au constructeur un nombre maximal d'éléments, et allouer dynamiquement l'emplacement mémoire correspondant ; toutefois, là encore, on perdrait le bénéfice de la correspondance immédiate entre un caractère et la position de son indicateur. Notez, toutefois, que ce sera la seule possibilité réaliste lorsqu'il s'agira de représenter des ensembles dans lesquels le nombre maximal d'éléments sera très grand.

Exercice III.7

Enoncé

Modifier la classe *set_char* précédente, de manière à ce que l'on dispose de ce que l'on nomme un "itérateur" sur les différents éléments de l'ensemble. On nomme ainsi un mécanisme permettant d'accéder séquentiellement aux différents éléments. On prévoira trois nouvelles fonctions membre : *init*, qui initialise

le processus d'exploration : *prochain*, qui fournit l'élément suivant lorsqu'il existe et *existe*, qui précise s'il existe encore un élément non exploré.

On complétera alors le programme d'utilisation précédent, de manière à ce qu'il affiche les différents caractères contenus dans le mot fourni en donnée.

Solution

Compte tenu de l'implémentation de notre classe, la gestion du mécanisme d'itération nécessite l'emploi d'un pointeur (que nous nommerons *courant*) sur un élément du tableau *ens*. Nous conviendrons que *courant* désigne le premier élément de *ens* non encore traité dans l'itération, c'est-à-dire non encore renvoyé par la fonction membre *suivant* (nous aurions pu adopter la convention contraire, à savoir que *courant* désigne le dernier élément traité).

En outre, pour nous faciliter la reconnaissance de la fin de l'itération, nous utiliserons un membre donnée supplémentaire (*fin*) valant 0 dans les cas usuels, et 1 lorsque aucun élément ne sera disponible (pour *suivant*).

Le rôle de la fonction *init* sera donc de faire pointer *courant* sur la première valeur non nulle de *ens* s'il en existe une ; dans le cas contraire, *fin* sera placé à 1.

La fonction *suivant* fournira en retour l'élément pointé par *courant* lorsqu'il existe (*fin* non nul) ou la valeur 0 dans le cas contraire (il s'agit là d'une convention destinée à protéger l'utilisateur ayant appelé cette fonction, alors qu'aucun élément n'était plus disponible). Dans le premier cas, *suivant* recherchera le prochain élément de l'ensemble (en modifiant la valeur de *fin* lorsqu'un tel élément n'existe pas). Notez bien que la fonction *suivant* doit renvoyer, non pas le prochain élément, mais l'élément courant.

Enfin la fonction *existe* se contentera de renvoyer la valeur de *fin* puisque cette dernière indique l'existence ou l'inexistence d'un élément courant.

Voici la nouvelle définition de la classe *set_char* :

```
          /*         fichier SETCHAR2.H        */
          /* déclaration de la classe set_char */
#define N_CAR_MAX 256       // on pourrait utiliser UCHAR_MAX défini
                            //    dans <limits.h>

class set_char
{
   unsigned char ens [N_CAR_MAX] ;
             // tableau des indicateurs (présent/absent)
             // pour chacun des caractères possibles
   int courant ;  // position courante dans le tableau ens
   int fin ;      // indique si fin atteinte
 public :
   set_char () ;                    // constructeur
   void ajoute (unsigned char) ;    // ajout d'un élément
   int appartient (unsigned char) ; // appartenance d'un élément
   int cardinal () ;                // cardinal de l'ensemble
   void init () ;                   // initialisation itération
   unsigned char suivant () ;       // caractère suivant
   int existe () ;                  //
} ;
```

Voici la définition des trois nouvelles fonctions membre *init, suivant* et *existe* :

```
void set_char::init ()
{  courant=0 ; fin = 0 ;
   while ( (++courant<N_CAR_MAX) && (!ens[courant]) ) ;
     // si la fin de ens est atteinte, courant vaut N_CAR_MAX
   if (courant>=N_CAR_MAX) fin = 1 ;
}

unsigned char set_char::suivant ()
{   if (fin) return 0 ;  // au cas où on serait déjà en fin de ens
    unsigned char c = courant ;   // conservation du caractère courant
       // et recherche du suivant s'il existe
    while ( (++courant<N_CAR_MAX) && (!ens[courant]) )  ;
       // si la fin de ens est atteinte, courant vaut N_CAR_MAX
    if (courant>=N_CAR_MAX) fin = 1 ;   // s'il n'y a plus de caractère
    return c ;
}

int set_char::existe ()
{   return (!fin) ;
}
```

Voici, enfin l'adaptation du programme d'utilisation :

```
#include <iostream.h>
#include <string.h>
#include "setchar2.h"
main()
{  set_char ens ;
   char mot [81] ;
   cout << "donnez un mot " ;
   cin >> mot ;
   int i ;
   for (i=0 ; i<strlen(mot) ; i++) ens.ajoute (mot[i]) ;
   cout << "il contient " << ens.cardinal () << " caractères différents"
        << " qui sont :\n" ;
   ens.init() ; // init itération sur les caractères de l'ensemble
   while (ens.existe())
     cout << ens.suivant () ;
}
```

CHAPITRE IV :
PROPRIETES DES FONCTIONS MEMBRE

RAPPELS

Surdéfinition des fonctions membre et arguments par défaut

Il s'agit simplement de la généralisation aux fonctions membre des possibilités déjà offertes par C++ pour les "fonctions ordinaires".

Fonctions membre en ligne

Il s'agit également de la généralisation aux fonctions membre d'une possibilité offerte pour les fonctions ordinaires, avec une petite nuance concernant sa mise en oeuvre ; pour rendre "en ligne" une fonction membre, on peut :

- soit fournir directement la définition de la fonction dans la déclaration même de la classe ; dans ce cas le qualificatif *inline* n'a pas à être utilisé, comme dans cet exemple :

```
class truc
{   ...
    int fctenlig (int, float)
      {    définition de fctenlig
      }
    .....
} ;
```

- soit procéder comme pour une fonction "ordinaire", en fournissant une définition en dehors de la déclaration de la classe ; dans ce cas, le qualificatif *inline* doit apparaître, à la fois devant la déclaration et devant l'en-tête.

Cas des objets transmis en argument d'une fonction membre

Une fonction membre reçoit implicitement l'adresse de l'objet l'ayant appelé. Mais, en outre, il est toujours possible de lui transmettre explicitement un argument (ou plusieurs arguments) du type de sa classe, ou même du type d'une autre classe. Dans le premier cas, la fonction membre aura accès aux membres privés de l'argument en question (car, en C++, l'unité d'encapsulation est la classe elle-même et non l'objet !). En revanche, dans le second cas, la fonction membre n'aura accès qu'aux membres publics de l'argument.

Un tel argument peut être transmis classiquement par valeur, par adresse ou par référence. Avec la transmission par valeur, il y a recopie des valeurs des membres donnée dans un emplacement local à la fonction appelée. Des problèmes peuvent surgir dès lors que l'objet transmis en argument contient des pointeurs sur des parties dynamiques. Ils seront réglés par l'emploi d'un "constructeur par recopie" (voir chapitre suivant).

Remarque :

Bien que ce soit d'un usage plus limité, une fonction ordinaire peut également recevoir un argument de type classe. Bien entendu, dans ce cas, elle n'aura accès qu'aux membres publics de cet argument.

Cas des fonctions membre fournissant un objet en retour

Une fonction membre peut fournir comme valeur de retour un objet du type de sa classe ou d'un autre type classe (dans ce dernier cas, elle n'accèdera bien sûr qu'aux membres publics de l'objet en question). La transmission peut, là encore, se faire par valeur, par adresse ou par référence.

La transmission par valeur implique une recopie qui pose donc les mêmes problèmes que ceux évoqués ci-desssus pour les objets comportant des pointeurs sur des parties dynamiques. Quant aux transmissions par adresse ou par référence, elles doivent être utilisées avec beaucoup de précautions, dans la mesure où, dans ce cas, on renvoie (généralement) l'adresse d'un objet alloué automatiquement, c'est-à-dire dont la durée de vie coïncide avec celle de la fonction.

Auto-référence : le mot clé this

Au sein d'une fonction membre, **this** représente un pointeur sur l'objet ayant appelé ladite fonction membre.

Fonctions membre statiques

Lorsqu'une fonction membre a une action indépendante d'un quelconque objet de sa classe, on peut la déclarer avec l'attribut *static*. Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (::).

Fonctions membre constantes

On peut déclarer des objets constants (à l'aide du qualificatif *const*). Dans ce cas, seules les fonctions membre déclarées (et définies) avec ce même qualificatif (exemple de déclaration : *void affiche () const*) peuvent recevoir (implicitement ou explicitement) en argument un objet constant.

Exercice IV.1

Enoncé

On souhaite réaliser une classe *vecteur3d* permettant de manipuler des vecteurs à trois composantes. On prévoit que sa déclaration se présente ainsi :

```
class vecteur3d
{   float x, y, z ;      // pour les 3 composantes (cartésiennes)
    .....
} ;
```

On souhaite pouvoir déclarer un vecteur, soit en fournissant explicitement ses trois composantes, soit en en fournissant aucune, auquel cas le vecteur créé possèdera trois composantes nulles. Ecrire le ou les constructeur(s) correspondant(s) :

a) en utilisant des fonctions membre surdéfinies,

b) en utilisant une seule fonction membre,

c) en utilisant une seule fonction "en ligne".

Solution

Il s'agit de simples applications des possibilités de surdéfinition, d'arguments par défaut et d'écriture "en ligne" des fonctions membres.

a)

```
          /* déclaration de la classe vecteur3d */
class vecteur3d
{
    float x, y, z ;
   public :
    vecteur3d () ;                    // constructeur sans arguments
    vecteur3d (float, float, float) ;  // constructeur 3 composantes
    .....
} ;

    /* définition des constructeurs de la classe vecteur3d */
vecteur3d::vecteur3d ()
{ x = 0 ; y = 0 ; z = 0 ;
}
vecteur3d::vecteur3d (float c1, float c2, float c3)
{ x = c1 ; y = c2 ; z = c3 ;
}
```

b)

```
          /* déclaration de la classe vecteur3d */
class vecteur3d
{
    float x, y, z ;
   public :
    vecteur3d (float=0.0, float=0.0, float=0.0) ; // constructeur (unique)
    .....
} ;
    /* définition du constructeur de la classe vecteur3d */
```

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

On notera toutefois qu'avec ce constructeur il est possible de déclarer un point en fournissant non seulement 0 ou 3 composantes, mais éventuellement seulement une ou deux. Cette solution n'est donc pas rigoureusement équivalente à la précédente.

c)

```
          /* déclaration de la classe vecteur3d */
class vecteur3d
{   float x, y, z ;
   public :
      // constructeur unique "en ligne"
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0) ;
    ..{ x = c1 ; y = c2 ; z = c3 ;
      }.
   .....
} ;
```

Ici, il n'y a plus aucune définition de constructeur, puisque ce dernier est "en ligne".

Exercice IV.2

Enoncé

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 ;
      }
     ...
} ;
```

Introduire une fonction membre nommée *coincide* permettant de savoir si deux vecteurs ont mêmes composantes :

a) en utilisant une transmission par valeur,

b) en utilisant une transmission par adresse,

c) en utilisant une transmission par référence.

Si v1 et v2 désignent 2 vecteurs de type *vecteur3d*, comment s'écrit le test de coïncidence de ces 2 vecteurs, dans chacun des 3 cas considérés ?

Solution

La fonction *coincide* est membre de la classe *vecteur3d* ; elle recevra donc implicitement l'adresse du vecteur l'ayant appelée. Elle ne possédera donc qu'un seul argument, lui-même de type *vecteur3d*. Nous supposerons qu'elle fournit une valeur de retour de type *int* (1 pour la coïncidence, 0 dans le cas contraire).

a) La déclaration de *coincide* pourra se présenter ainsi :

```
int coincide (vecteur3d) ;
```

Voici ce que pourrait être sa définition :

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

b) La déclaration de *coincide* devient :

```
int coincide (vecteur3d *) ;
```

Et sa nouvelle définition pourrait être :

```
int vecteur3d::coincide (vecteur3d * adv)
{   if ( (adv->x == x) && (adv->y == y) && (adv->z == z) )
                return 1 ;
           else return 0 ;
}
```

Remarque :

En utilisant *this*, la définition de *coincide* pourrait faire moins de distinction entre ses deux arguments (l'un implicite, l'autre explicite) :

```
int vecteur3d::coincide (vecteur3d * adv)
{  if ( (adv->x == this->x) && (adv->y == this->y) && (adv->z == this->z) )
                return 1 ;
           else return 0 ;
}
```

c) La déclaration de *coincide* devient :

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

Et sa nouvelle définition est :

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

Notez que le corps de la fonction est resté le même qu'en a.

Voici les 3 appels de *coincide* correspondant respectivement aux 3 définitions précédentes :

a) `v1.coincide (v2) ;` ou `v2.coincide (v1) ;`

b) `v1.coincide (&v2)` ou `v2.coincide (&v1) ;`

c) `v1.coincide (v2)` ou `v2.coincide (v1) ;`

Discussion

La surdéfinition d'opérateur offrira une mise en oeuvre plus agréable de ce test de coïncidence de deux vecteurs. C'est ainsi qu'il sera possible de surdéfinir l'opérateur de comparaison == (pour la classe *vecteur3d*) et, partant, d'exprimer ce test sous la simple forme : *v1 == v2*.

Exercice IV.3

Enoncé

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 ;
      }
     .....
} ;
```

Introduire, dans cette classe, une fonction membre nommée *normax* permettant d'obtenir, parmi deux vecteurs, celui qui a la plus grande norme. On prévoira deux situations :

a) le résultat est renvoyé par valeur,

b) le résultat est renvoyé par référence, l'argument (explicite) étant également transmis par référence.

c) le résultat est renvoyé par adresse, l'argument (explicite) étant également transmis par adresse.

Solution

a) La seule difficulté réside dans la manière de renvoyer la valeur de l'objet ayant appelé une fonction membre, à savoir **this*. Voici la définition de la fonction *normax* (la déclaration en découle immédiatement) :

```
vecteur3d vecteur3d::normax (vecteur3d v)
{  float norm1 = x*x + y*y + z*z ;
   float norm2 = v.x*v.x + v.y*v.y + v.z*v.z ;
   if (norm1>norm2) return *this ;
               else return v ;
```

```
}
```

Voici un exemple d'utilisation (on suppose que *v1*, *v2* et *w* sont de type *vecteur3d)* :

```
w = v1.normax (v2) ;
```

Notez bien que l'affectation ne pose aucun problème ici, puisque notre classe ne comporte aucun pointeur sur des parties dynamiques.

b) Aucun nouveau problème ne se pose. Il suffit de modifier ainsi l'en-tête de notre fonction, sans en modifier le corps :

```
vecteur3d & vecteur3d::normax (vecteur3d & v)
```

La fonction *normax* s'utilise comme précédemment.

c) Il faut, cette fois, adapter en conséquence l'en-tête et le corps de la fonction :

```
vecteur3d *  vecteur3d::normax (vecteur3d * adv)
{  float norm1 = x * x + y * y + z * z ;
   float norm2 = adv->x * adv->x + adv->y * adv->y + adv->z * adv->z ;
   if (norm1>norm2) return this ;
               else return  adv ;
}
```

Ici, l'utilisation de la fonction nécessite quelques précautions. En voici un exemple (*v1*, *v2* et *w* sont toujours de type *vecteur3d*) :

```
w = *(v1.normax(&v2))  ;
```

Discussion

En ce qui concerne la transmission de l'unique argument explicite de *normax*, il faut noter qu'il est impossible de la prévoir par valeur, dès lors que *normax* doit restituer son résultat par adresse ou par référence. En effet, dans ce cas, on obtiendrait en retour l'adresse ou la référence d'un vecteur alloué automatiquement au sein de la fonction. Notez qu'un tel problème ne se pose pas pour l'argument implicite (*this*), car il correspond toujours à l'adresse d'un vecteur (transmis automatiquement par référence), et non à une valeur.

Exercice IV.4

Enoncé

Réaliser une classe *vecteur3d* permettant de manipuler des vecteurs à 3 composantes (de type *fbat*). On y prévoira :

- un constructeur, avec des valeurs par défaut (0),
- une fonction d'affichage des 3 composantes du vecteur, sous la forme :

```
< composante1, composante2, composante3 >
```

- une fonction permettant d'obtenir la somme de 2 vecteurs,
- une fonction permettant d'obtenir le produit scalaire de 2 vecteurs.

On choisira les modes de transmission les mieux appropriés. On écrira un petit programme utilisant la classe ainsi réalisée.

Solution

La fonction membre calculant la somme de deux vecteurs (nous la nommerons *somme*) reçoit implicitement (par référence) un argument de type *vecteur3d*. Elle comportera donc un seul argument, lui aussi de type *vecteur3d*. On peut, a priori, le transmettre par adresse, par valeur ou par référence. En fait, la transmission par adresse, en C++, n'a plus guère de raison d'être, dans la mesure où la transmission par référence fait la même chose, moyennant une écriture plus agréable.

Le choix doit donc se faire entre transmission par valeur ou par référence. Lorsqu'il s'agit de transmettre un objet (comportant plusieurs membres donnée), la transmission par référence est plus efficace (en temps d'exécution). Qui plus est, la fonction *somme* reçoit déjà implicitement un vecteur par référence, de sorte qu'il n'y a aucune raison de lui transmettre différemment le second vecteur.

Le même raisonnement s'applique à la fonction de calcul du produit scalaire (que nous nommons *prodscal*).

En ce qui concerne la valeur de retour de *somme*, laquelle est également de type *vecteur3d*, il n'est, en revanche, pas possible de la tranmettre par référence. En effet, ce "résultat" (de type *vecteur3d*) sera créé au sein de la fonction elle-même, ce qui signifie que l'objet correspondant sera de classe automatique, donc détruit à la fin de l'exécution de la fonction. Il faut donc absolument en transmettre la valeur.

Voici ce que pourrait être la déclaration de notre classe *vecteur3d* (ici, seul le constructeur a été prévu "en ligne") :

```
       /* déclaration de la classe vecteur3d */
class vecteur3d
{   float x, y, z ;
  public :
    vecteur3d (float c1=0, float c2=0, float c3=0)  // constructeur
      {  x=c1 ; y=c2 ; z=c3;
      }
    vecteur3d somme (vecteur3d &) ;      // somme (résultat par valeur)
    float prodscal (vecteur3d &) ;       // produit scalaire
    void affiche () ;                    // affichage composantes
} ;
```

Voici sa définition :

```
       /* définition de la classe vect3d */
#include <iostream.h>
vecteur3d vecteur3d::somme (vecteur3d & v)
{   vecteur3d res ;
    res.x = x + v.x ;
    res.y = y + v.y ;
    res.z = z + v.z ;
    return res ;
}
float vecteur3d::prodscal (vecteur3d & v)
{   return ( v.x * x + v.y * y + v.z * z) ;
}
void vecteur3d::affiche ()
```

```
{   cout << "< " << x << ", " << y << ", " << z << ">" ;
}
```

Voici un petit programme d'essai de la classe *vecteur3d*, accompagné des résultats produit par son exécution :

```
      /* programme d'essai de la classe vecteur3d */
main()
{  vecteur3d v1 (1,2,3), v2 (3,0, 2), w ;
   cout << "v1 = " ; v1.affiche () ; cout << "\n" ;
   cout << "v2 = " ; v2.affiche () ; cout << "\n" ;
   cout << "w = " ;  w.affiche () ;  cout << "\n" ;
   w = v1.somme (v2) ;
   cout << "w = " ;  w.affiche () ;  cout << "\n" ;
   cout << "V1.V2 = " << v1.prodscal (v2) << "\n" ;
}
```

```
v1 = < 1, 2, 3>
v2 = < 3, 0, 2>
w = < 0, 0, 0>
w = < 4, 2, 5>
V1.V2 = 9
```

Exercice IV.5

Enoncé

Comment pourrait-on adapter la classe *point* créée dans l'exercice III.5, pour qu'elle dispose d'une fonction membre *nombre* fournissant le nombre de points existant à un instant donné ?

Solution

On pourrait certes introduire une fonction membre classique. Toutefois, cette solution présenterait l'inconvénient d'obliger l'utilisateur à appliquer une telle fonction à un objet de type *point*. Que penser alors d'un appel tel que (*p* étant un *point*) *p.compte()* pour connaître le nombre de points ? Qui plus est, comment faire appel à *compte* s'il n'existe aucun point ?

La solution la plus agréable consiste à faire de *compte* une "fonction statique". On la déclarera donc ainsi :

```
static int compte () ;
```

Voici sa définition :

```
int point::compte ()
{  return nb_pts ;
}
```

Voici un exemple d'appel de *compte* au sein d'un programme dans lequel la classe *point* a été déclarée :

```
cout << "il y a " << point::compte () << "points\n" ;
```

CHAPITRE V :
CONSTRUCTION, DESTRUCTION ET INITIALISATION DES OBJETS

RAPPELS

Appels du constructeur et du destructeur

Dans tous les cas (objets statiques, automatiques ou dynamiques), s'il y a appel du **constructeur**, celui-ci a lieu après l'allocation de l'emplacement mémoire destiné à l'objet. De même, s'il existe un **destructeur**, ce dernier est appelé avant la libération de l'espace mémoire associé à l'objet.

Les objets automatiques et statiques

Les objets **automatiques** sont créés par une déclaration soit dans une fonction, soit au sein d'un bloc. Ils sont créés au moment de l'exécution de la déclaration (laquelle, en C++, peut apparaître n'importe où dans un programme). Ils sont détruits lorsque l'on sort de la fonction ou du bloc.

Les objets **statiques** sont créés par une déclaration située en dehors de toute fonction ou par une déclaration précédée du mot clé *static* (dans une fonction ou dans un bloc). Ils sont créés avant l'entrée dans la fonction *main* et détruits après la fin de son exécution.

Les objets temporaires

L'appel explicite, au sein d'une expression, du constructeur d'un objet provoque la création d'un objet temporaire (on n'a pas accès à son adresse) qui pourra être automatiquement détruit dès lors qu'il ne sera plus utile (mais le langage ne précise pas quand aura effectivement lieu cette destruction!). Par exemple, si une classe *point* possède le constructeur *point(float, float)* et si *a* est de type *point*, nous pouvons écrire[1] :

```
a = point (1.5, 2.25) ;
```

Cette instruction provoque la création d'un objet temporaire de type *point* (avec appel du constructeur concerné), suivie de l'affectation de cet objet à *a*.

[1] Ne confondez pas une telle affectation avec une initialisation d'un objet lors de sa déclaration,

Les objets dynamiques

Ils sont créés par l'opérateur **new**, auquel on doit fournir, le cas échéant, les valeurs des arguments destinés à un constructeur[2], comme dans cet exemple (qui suppose qu'il existe une classe *point* possédant le constructeur *point(float, float)*) :

```
point * adp ;
   .....
adp = new point (2.5, 5.32) ;
```

L'accès aux membres d'un objet dynamique est réalisé comme pour les variables ordinaires. Par exemple, si *point* possède une méthode nommée *affiche*, on pourra l'appeler par *(*adp).affiche ()* ou encore par *adp->affiche ()*.

Les objets dynamiques n'ont pas de durée de vie définie a priori. Ils sont détruits à la demande en utilisant l'opérateur **delete** comme dans : *delete adr*[3].

Initialisation d'objets

En C++, il est très important de distinguer les deux opérations que sont l'initialisation et l'affectation.

Il y a initialisation d'un objet dans l'une des trois situations suivantes :

- déclaration d'un objet avec *initialiseur*, comme dans :

```
point a = 5 ;   // il doit exister un constructeur à un argument de type entier
point b = a ;   // il doit exister un constructeur à un argument de type point
```

- transmission d'un objet par valeur en argument d'appel d'une fonction,

- transmission d'un objet par valeur en valeur de retour d'une fonction.

L'initialisation d'un objet provoque toujours l'appel d'un constructeur particulier, nommé **constructeur par recopie** ; plus précisément :

- si aucun constructeur de l'une des formes *type (type &)* ou *type (const type &)* (*type* désignant le type de l'objet) n'a été défini, il y aura appel d'un "constructeur de recopie par défaut" ; ce dernier recopie les valeurs des différents membres donnée de l'objet, comme le fait l'affectation. Des problèmes peuvent alors se poser dès lors que l'objet contient des pointeurs sur des parties dynamiques.

- si un constructeur de la forme *type (type &)* ou, mieux, *type (const type &)* existe, il sera appelé. Notez qu'alors le constructeur par recopie par défaut n'est plus appelé. C'est donc au constructeur par recopie de prévoir la recopie de tous les membres de l'objet.

Remarque : une exception

Lorsque l'on initialise un objet par un simple d'appel d'un constructeur comme dans (en supposant que le type *point* dispose d'un constructeur approprié) :

```
point a = point (1.3, 4.5)
```

il n'y a pas d'appel d'un constructeur par recopie, précédée de la création d'un objet temporaire. La déclaration précédente est rigoureusement équivalente à :

```
point a (1.3, 4.5) ;
```

---

[2] Comme pour les variables ordinaires, on peut préciser un nombre d'objets mais des restrictions apparaissent alors quant au constructeur qu'il est possible d'appeler (voyez ci-dessous la rubrique "tableaux d'objets").

[3] Dans le cas d'un tableau d'objets, on devra préciser, en plus, son nombre d'éléments.

Les tableaux d'objets

Si *point* est un type objet possédant un constructeur sans argument (ou, situation généralement déconseillée, sans constructeur), la déclaration :

```
point courbe [20] ;
```

crée un tableau *courbe* de 20 objets de type *point* en appelant, le cas échéant, le constructeur pour chacun d'entre eux. Notez toutefois que *courbe* n'est pas lui-même un objet.

De même :

```
point * adcourbe = new point [20] ;
```

alloue l'emplacement mémoire nécessaire à vingt objets (consécutifs) de type *point*, en appelant, le cas échéant, le constructeur pour chacun d'entre eux, puis place l'adresse du premier dans *adcourbe*.

La destruction d'un tableau d'objets se fait en utilisant l'opérateur **new** (en précisant, dans les versions inférieures à la 3.0, le nombre d'éléments du tableau). Par exemple, pour détruire le tableau précédent, on écrira :

```
delete [] adcourbe ;                  // depuis la version 3.0
```

ou :

```
delete [20] adcourbe ;               // versions antérieures à la 3.0
```

Remarque :

En théorie, depuis la version 2.0 de C++, on peut compléter la déclaration d'un tableau d'objets par un initialiseur contenant une liste de valeurs (elles peuvent éventuellement être de types différents). Chaque valeur est alors transmise à un constructeur approprié. Cette faculté ne peut toutefois pas s'appliquer aux tableaux dynamiques (il en va de même pour les tableaux ordinaires !).

Construction d'objets contenant des objets membre

Une classe peut posséder un membre donnée qui est lui-même de type classe. En voici un exemple ; si nous avons défini :

```
class point
{   float x, y ;
   public :
    point (float, float) ;
    .....
} ;
```

nous pouvons définir une classe *pointcol*, dont un membre est de type *point* :

```
class pointcol
{   point p ;
    int couleur ;
   public :
    pointcol (float, float, int) ;
    .....
```

```
} ;
```

Dans ce cas, lors de la création d'un objet de type *pointcol*, il y aura tout d'abord appel d'un constructeur de *pointcol*, puis appel d'un constructeur de *point* ; ce dernier recevra les arguments qu'on aura mentionné dans l'en-tête de la définition du constructeur de *pointcol* ; par exemple, avec :

```
pointcol::pointcol (float abs, float ord, int coul) : p (abs, ord)
{   ...
}
```

on précise que le constructeur du membre *p* recevra en argument les valeurs *abs* et *ord*.

Si l'en-tête de *pointcol* ne mentionnait rien concernant p, il faudrait alors que le type *point* possède un constructeur sans argument pour que cela soit correct.

Initialisation d'objets contenant des objets membre

Depuis la version 2.0 de C++, le constructeur par recopie par défaut procède "membre par membre" ; cela signifie que si l'un des membres est lui-même un objet, il est recopié en appelant son propre constructeur par recopie (qui pourra être soit un constructeur par défaut, soit un constructeur défini dans la classe correspondante). Dans les versions antérieures, la recopie se faisait de façon globale ; autrement dit, la "valeur" d'un objet était reportée ("bit par bit") dans une autre, sans tenir compte de sa structure.

Les différences entre ces deux modes de recopie seront particulièrement sensibles dans la situation suivante :

- objet membre comportant des pointeurs sur des emplacements dynamiques et ayant défini son constructeur par recopie ;
- objet principal n'ayant pas défini de constructeur par recopie.

Depuis la version 2.0, la construction par recopie d'un tel objet fonctionnera convenablement (s'il ne contient pas lui-même de pointeur), puisqu'il y aura bien appel du constructeur par recopie de l'objet membre. Dans les versions antérieures, en revanche, ce constructeur par recopie, bien qu'existant, ne sera pas utilisé et les choses seront beaucoup moins satisfaisantes.

Exercice V.1

Enoncé

Comment concevoir le type classe *chose* de façon à ce que ce petit programme :

```
main()
{
   chose x ;
   cout << "bonjour\n" ;
}
```

fournisse les résultats suivants :

```
création objet de type chose
bonjour
destruction objet de type chose
```

Que fournira alors l'exécution de ce programme (utilisant le même type *chose*) :

```
main()
{
   chose * adc = new chose
}
```

Solution

Il suffit de prévoir dans le constructeur de *chose*, l'instruction :

```
cout << "création objet de type chose\n" ;
```

et dans le destructeur, l'instruction :

```
cout << "destruction objet de type chose\n" ;
```

Dans ce cas, le deuxième programme fournira simplement à l'exécution (puisqu'il crée un objet de type *chose* sans jamais le détruire) :

```
création objet de type chose
```

Exercice V.2

Enoncé

Quels seront les résultats fournis par l'exécution du programme suivant (ici, la déclaration de la classe *demo*, sa définition et le programme d'utilisation ont été regroupés en un seul fichier) :

```
#include <iostream.h>

class demo
{  int x, y ;
  public :
   demo (int abs=1, int ord=0)   // constructeur I (0, 1 ou 2 arguments)
     { x = abs ;  y = ord ;
       cout << "constructeur I            : " << x << " " << y << "\n" ;
     }
   demo (demo&) ;                // constructeur II (par recopie)
   ~demo () ;                    // destructeur
} ;

demo::demo (demo & d)
{  cout << "constructeur II (recopie) : " << d.x << " " << d.y << "\n" ;
   x = d.x ;  y = d.y ;
}
demo::~demo ()
{  cout << "destruction              : " << x << " " << y << "\n" ;
}
```

```
main ()
{
   void fct (demo, demo *) ;        // proto fonction indépendante fct
   cout << "début main\n" ;
   demo a ;
   demo b = 2 ;
   demo c = a ;
   demo * adr = new demo (3,3) ;
   fct (a, adr) ;
   demo d = demo (4,4) ;
   c = demo (5,5) ;
   cout << "fin main\n" ;
}

void fct (demo d, demo * add)
{   cout << "entrée fct\n" ;
    delete add ;
    cout << "sortie fct\n" ;
}
```

Solution

Voici les résultats (commentés) que fournit le programme (ici, nous avons utilisé le compilateur Turbo C++ [4]) :

```
début main
constructeur I            : 1 0      demo a ;
constructeur I            : 2 0      demo b = 2 ;
constructeur II (recopie) : 1 0      demo c = a ;
constructeur I            : 3 3      new demo (3,3)
constructeur II (recopie) : 1 0      recopie de la valeur de a dans fct(a, ..)
                                          (crée un objet temporaire)
entrée fct
destruction               : 3 3      delete add ;  (dans fct)
sortie fct
constructeur I            : 4 4      demo d = demo (4, 4) ;
constructeur I            : 5 5      c = demo (5,5)  (construction objet temporaire)
fin main
destruction               : 5 5      destruction objet temporaire  (peut se
                                          faire à un autre moment
destruction               : 4 4      destruction d
destruction               : 1 0      destruction objet temporaire créé pour l'appel
                                          de fct (peut apparaître ailleurs)
destruction               : 5 5      destruction c
destruction               : 2 0      destruction b
destruction               : 1 0      destruction a
```

Notez bien que l'affectation *c = demo (5,5)* entraîne la création d'un objet temporaire par appel du constructeur de *demo* (arguments 5 et 5) ; cet objet est ensuite affecté à *a*. On constate d'ailleurs que cet

[4] Ce point n'a en fait d'importance que pour les objets temporaires.

objet est effectivement détruit (ici, après l'exécution du *main*, mais, dans d'autres implémentations, cela peut se produire plus tôt).

Par ailleurs, l'appel de *fct* a entraîné la construction d'un objet temporaire, par appel du constructeur par recopie. Nous aurions pu penser que cet objet serait libéré à la sortie de la fonction. Ici, ce n'est pas le cas mais cela pourrait l'être dans d'autres implémentations.

Discussion

Cet exemple devrait vous mettre en garde contre le fait que de nombreux objets temporaires peuvent prendre naissance dans un programme, sans même qu'on s'en rende compte ; des problèmes d'encombrement de la mémoire peuvent apparaître dans les implémentations qui ne libèrent pas les emplacement temporaires, dès que cela est devenu possible.

Exercice V.3

Enoncé

Créer une classe *point* ne contenant qu'un constructeur sans arguments, un destructeur et un membre donnée privé représentant un numéro de point (le premier créé portera le numéro 1, le suivant le numéro 2, et ainsi de suite...). Le constructeur affichera le numéro du point créé et le destructeur affichera le numéro du point détruit. Ecrire un petit programme d'utilisation créant dynamiquement un tableau de 4 points et le détruisant.

Solution

Pour pouvoir numéroter nos points, il nous faut pouvoir compter le nombre de fois où le constructeur a été appelé, ce qui nous permettra bien d'attribuer un numéro différent à chaque point. Pour ce faire, nous définissons, au sein de la classe *point*, un membre donnée statique *nb_points*. Ici, il sera incrémenté par le constructeur mais le destructeur n'aura pas d'action sur lui.

Voici la déclaration (définition) de notre classe, accompagnée du programme d'utilisation demandé :

```
#include <iostream.h>
class point
{   int num;
    static nb_points ;
  public :
    point ()
       {  num = ++nb_points ;
          cout << "création point numéro    : " << num << "\n" ;
       }
    ~point ()
       {  cout << "Destruction point numéro : " << num << "\n" ;
       }
} ;

main()
{  point * adcourb = new point [4] ;
```

```
   int i ;
   delete [4] adcourb ;
}
```

Exercice V.4

Enoncé

1) Réaliser une classe nommée *set_int* permettant de manipuler des ensembles de nombres entiers. On devra pouvoir réaliser sur un tel ensemble les opérations classiques suivantes : lui ajouter un nouvel élément, connaître son "cardinal" (nombre d'éléments), savoir si un entier donné lui appartient.

Ici, on conservera les différents éléments de l'ensemble dans un tableau alloué dynamiquement par le constructeur. Un argument (auquel on pourra prévoir une valeur par défaut) lui précisera le nombre maximal d'éléments de l'ensemble.

2) Ecrire, en outre, un programme (*main*) utilisant la classe *set_int* pour déterminer le nombre d'entiers différents contenus dans un tableau d'entiers lus en donnée.

3) Que faudrait-il faire pour qu'un objet du type *set_int* puisse être transmis par valeur, soit comme argument d'appel, soit comme valeur de retour d'une fonction.

Solution

1) La déclaration de la classe découle de l'énoncé :

```
            /*         fichier SETINT1.H        */
            /* déclaration de la classe set_int */
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
   void ajoute (int) ;        // ajout d'un élément
   int appartient (int) ;     // appartenance d'un élément
   int cardinal () ;          // cardinal de l'ensemble
} ;
```

Le membre donnée *adval* est destiné à pointer sur le tableau d'entiers qui sera alloué par le constructeur. Le membre *nmax* représentera la taille de ce tableau, tandis que *nelem* fournira le nombre effectif d'entiers stockés dans ce tableau. Ces entiers seront, cette fois, rangés dans l'ordre où ils seront fournis à *ajoute*, et non plus à un emplacement prédéterminé, comme nous l'avions fait pour les caractères (dans les exercices du chapitre précédent).

Comme la création d'un objet entraîne ici une allocation dynamique d'un emplacement mémoire, il est raisonnable de prévoir la libération de cet emplacement lors de la destruction de l'objet ; cette opération doit donc être prise en charge par le destructeur, d'où la présence de cette fonction membre.

Voici la définition de notre classe :

```
#include "setint1.h"
set_int::set_int (int dim)
{  adval = new int [nmax = dim] ;    // allocation tableau de valeurs
   nelem = 0 ;
}
set_int::~set_int (
{  delete adval ;                    // libération tableau de valeurs
}

void set_int::ajoute (int nb)
{   // on examine si nb appartient déjà à l'ensemble
    //   en utilisant la fonction membre appartient
    // s'il n'y appartient pas et si l'ensemble n'est pas plein
    //   on l'ajoute
   if (!appartient (nb) && (nelem<nmax))  adval [nelem++] = nb ;
}
int set_int::appartient (int nb)
{  int i ;
    // 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) ;
}
int set_int::cardinal ()
{  return nelem ;
}
```

Notez que, dans la fonction membre *ajoute*, nous avons utilisé la fonction membre *appartient* pour vérifier que le nombre à ajouter ne figurait pas déjà dans notre ensemble.

Par ailleurs, l'énoncé ne prévoit rien pour le cas où l'on cherche à ajouter un élément à un ensemble déjà "plein" ; ici, nous nous sommes contenté de ne rien faire dans ce cas. Dans la pratique, il faudrait, soit prévoir un moyen pour l'utilisateur d'être prévenu de cette situation, soit, mieux, prévoir automatiquement l'agrandissement de la zone dynamique associée à l'ensemble.

2) Voici le programme d'utilisation demandé :

```
#include <iostream.h>
#include "setint1.h"
main()
{  set_int ens ;
   cout << "donnez 20 entiers \n" ;
   int i, n ;
   for (i=0 ; i<20 ; i++)
      { cin >> n ;
     ens.ajoute (n) ;
      }
   cout << "il y a : " << ens.cardinal () << " entiers différents\n" ;
}
```

3) Tel qu'est actuellement prévue notre classe *set_int*, si un objet de ce type est transmis par valeur, soit en argument d'appel, soit en retour d'une fonction, il y a appel du constructeur de recopie par défaut ; or, ce dernier se contente d'effectuer une copie des membres donnée de l'objet concerné. Ce qui signifie qu'on se retrouve alors en présence de deux objets contenant deux pointeurs différents sur un même tableau d'entiers. Un problème va donc se poser, dès lors que l'objet copié sera détruit (ce qui peut se produire, dès la sortie

de la fonction) ;en effet, dans ce cas, le tableau dynamique d'entiers sera détruit, alors même que l'objet d'origine continuera à "pointer" dessus.

Indépendamment de cela, d'autres problèmes similaires pourrait se poser si la fonction était amenée à modifier le contenu de l'ensemble : dans ce cas, en effet, on modifierait le tableau d'entiers original, chose à laquelle on ne s'attend pas dans le cas de transmission par valeur.

Pour régler ces problèmes, il est nécessaire de munir notre classe d'un constructeur par recopie approprié. Que doit-il faire ? Plusieurs solutions existent ; la plus élégante est certainement ce que l'on nomme la gestion d'un "compteur de référence". Nous la rencontrerons plus tard, associée, généralement à la surdéfinition de l'opérateur d'affectation (qui pose des problèmes analogues à ceux que nous venons d'évoquer).

Pour l'instant, nous utiliserons une démarche plus simple qui consiste à recopier totalement l'objet concerné, c'est-à-dire en tenant compte de sa "partie dynamique" (on parle parfois de "copie profonde"). Pour ce faire, on alloue un second emplacement pour un tableau d'entiers, dans lequel on recopie les valeurs du premier ensemble. Naturellement, il ne faut pas oublier alors de procéder également à la recopie des membres donnée, puisque celle-ci n'est plus assurée par le constructeur de recopie par défaut (lequel n'est plus appelé, dès lors qu'un constructeur par recopie a été défini).

Nous ajouterons donc, dans la déclaration de notre classe :

```
set_int (set_int &) ;          // constructeur par recopie
```

Et, dans sa définition :

```
set_int::set_int (set_int & e)
{  adval = new int [nmax = e.nmax] ;  // allocation nouveau tableau
   nelem = e.nelem ;
   int i ;
   for (i=0 ; i<nelem ; i++)          // copie ancien tableau dans nouveau
    adval[i] = e.adval[i] ;
}
```

Discussion

La méthode proposée a le mérite d'être facile à programmer et elle ne nécessite pas d'autres modifications que l'ajout d'une méthode supplémentaire. Elle a l'inconvénient de dupliquer totalement l'objet, ce qui entraîne une perte de mémoire. La méthode du compteur de référence (que nous rencontrerons plus tard) évite cette duplication ;en contre partie, sa mise en oeuvre est beaucoup plus délicate et elle ne se borne plus au simple ajout d'une méthode.

Quelle que soit la méthode employée, on constate que la surdéfinition du constructeur par recopie est quasi indispensable pour toute classe comportant un pointeur sur une partie dynamique. Dans l'état actuel de C++, il n'est pas possible d'interdire la transmission par valeur d'un objet qui n'en posséderait pas! Et il ne semble pas raisonnable de livrer à un "client" une telle objet, en lui demandant de ne jamais le transmettre par valeur! Ce qui signifie que la plupart des classes "intéressantes" devront définir un tel constructeur par recopie.

Nous verrons que les mêmes réflexions s'appliqueront à l'affectation entre objets et qu'elles conduiront à la conclusion que la plupart des classes "intéressantes" devront redéfinir l'opérateur d'affectation.

Exercice V.5

Enoncé

Modifier l'implémentation de la classe précédente (avec son constructeur par recopie) de façon à ce que l'ensemble d'entiers soit maintenant représenté par une **liste chaînée** (chaque entier est rangé dans une structure comportant un champ destiné à contenir un nombre et un champ destiné à contenir un pointeur sur la structure suivante). L'interface de la classe (la partie publique de sa déclaration) devra rester inchangée, ce qui signifie qu'un client de la classe continuera à l'employer de la même façon.

Solution

Comme nous le suggère l'énoncé, nous allons donc définir une structure que nous nommerons *élément* :

```
struct noeud
{  int valeur ;               // valeur d'un élément de l'ensemble
   noeud * suivant ;          // pointeur sur le noeud suivant de la liste
} ;
```

Notez que nous pouvons, en C++, écrire simplement *noeud *suivant*, là où C nous aurait imposé *struct noeud *suivant*.

Notre structure *noeud* peut être définie indifféremment dans la déclaration de la classe *set_int* ou en dehors.

En ce qui concerne les membres donnée privés de la classe, nous ne conserverons que *nelem* qui, bien que non indispensable, nous évitera de parcourir toute la liste pour déterminer le cardinal de notre ensemble.

En revanche, nous y introduirons un pointeur nommé *debut*, destiné à contenir l'adresse du premier élément de la liste, s'il existe (au départ, il sera initialisé à NULL).

En ce qui concerne le constructeur de *set_int*, nous lui conserverons son argument (de type *int*), bien qu'ici il n'ait plus aucun intérêt, et ceci dans le but de ne pas modifier l'interface de notre classe (comme le demandait l'énoncé).

Voici donc la nouvelle déclaration de notre classe :

```
           /*         fichier SETINT3.H        */
           /* déclaration de la classe set_int */
struct noeud
{  int valeur ;               // valeur d'un élément de l'ensemble
   noeud * suivant ;          // pointeur sur le noeud suivant de la liste
} ;

class set_int
{
   noeud * debut ;            // pointeur sur le début de la liste
   int nelem ;                // nombre courant d'éléments
 public :
   set_int (int = 20) ;       // constructeur  (argument inutile ici)
   set_int (set_int &) ;      // constructeur par recopie
   ~set_int () ;              // destructeur
   void ajoute (int) ;        // ajout d'un élément
   int appartient (int) ;     // appartenance d'un élément
   int cardinal () ;          // cardinal de l'ensemble
} ;
```

La définition du nouveau constructeur ne présente pas de difficulté. La fonction membre *ajoute* réalise une classique insertion d'un *noeud* en début de liste et incrémente le nombre d'éléments de l'ensemble. La fonction *appartient* effectue une exploration de liste, tant qu'elle n'a pas trouvé la valeur concernée ou atteint la fin de la liste. En revanche, le nouveau constructeur par recopie doit recopier la liste chaînée. Il réalise à la fois une exploration de la liste d'origine et une insertion dans la liste copiée. Quant au destructeur, il doit maintenant libérer systématiquement tous les emplacements des différents noeuds créés pour la liste.

Voici la nouvelle définition de notre classe :

```
#include <stdlib.h>          // pour NULL
#include "setint3.h"

set_int::set_int (int dim)   // dim est conservé pour la compatibilité
                        // avec l'ancienne classe
{  debut = NULL ;
   nelem = 0 ;
}

set_int::set_int (set_int & e)
{  nelem = e.nelem ;
       // création d'une nouvelle liste identique à l'ancienne
   noeud * adsource = e.debut  ;
   noeud * adbut ;
   debut = NULL ;
   while (adsource)
       { adbut = new noeud ;                // création nouveau noeud
         adbut->valeur = adsource->valeur ; // copie valeur
         adbut->suivant = debut ;           // insertion nouveau noeud
         debut = adbut ;                    //     dans nouvelle liste
         adsource = adsource->suivant ;     // noeud suivant ancienne liste
       }
}

set_int::~set_int ()
{   noeud * adn ;
    noeud * courant = debut ;
    while (courant)
    { adn = courant ;                 // libération de tous
      courant = courant->suivant ;    //    les noeuds
      delete adn ;                    //    de la liste
    }
}

void set_int::ajoute (int nb)
{
   if (!appartient (nb) )          // si nb n'appartient pas à la liste
     { noeud * adn = new noeud ;   // on l'ajoute en début de liste
       adn->valeur = nb ;
       adn->suivant = debut ;
       debut = adn ;
       nelem++ ;
     }
}

int set_int::appartient (int nb)
{  noeud * courant = debut ;
       // attention à l'ordre des deux conditions
```

```
    while (courant && (courant->valeur != nb) ) courant = courant->suivant ;
    return (courant != NULL) ;
}

int set_int::cardinal ()
{  return nelem ;
}
```

Notez que le programme d'utilisation proposé dans l'exercice V.4 reste valable ici, puisque nous n'avons précisément pas modifié l'interface de notre classe.

Par ailleurs, le problème évoqué à propos de l'ajout d'un élément à un ensemble "plein" ne se pose plus ici, compte tenu de la nouvelle implémentation de notre classe.

Exercice V.6

Enoncé

Modifier la classe *set_int* précédente (implémentée sous forme d'une liste chaînée - avec ou sans son constructeur par recopie) pour qu'elle dispose de ce que l'on nomme un "itérateur" sur les différents éléments de l'ensemble. Rappelons qu'on nomme ainsi un mécanisme permettant d'accéder séquentiellement aux différents éléments de l'ensemble. On prévoira trois nouvelles fonctions membre : *init*, pour initialiser le processus d'itération ; *prochain*, pour fournir l'élément suivant lorsqu'il existe et *existe*, pour tester s'il existe encore un élément non exploré.

On complétera alors le programme d'utilisation précédent (en fait, celui de l'exercice V.4), de manière à ce qu'il affiche les différents entiers contenus dans les valeurs fournies en donnée.

N.B. Cet exercice sera plus profitable s'il est traité après l'exercice du chapitre III qui proposait l'introduction d'un tel itérateur dans une classe représentant des ensembles de caractères (mais dont l'implémentation était différente de l'actuelle classe).

Solution

Ici, la gestion du mécanisme d'itération nécessite l'emploi d'un pointeur (que nous nommerons *courant*) sur un noeud de notre liste. Nous conviendrons qu'il pointe sur le premier élément non encore traité dans l'itération, c'est-à-dire dont la valeur correspondante n'a pas encore été renvoyée par la fonction *prochain*. Il n'est pas utile, ici, de prévoir un membre donnée pour indiquer si la fin de liste a été atteinte ; en effet, avec la convention adoptée, il nous suffit de tester la valeur de *courant* (qui sera égale à NULL, lorsque l'on sera en fin de liste).

Le rôle de la fonction *init* se limite à l'initialisation de *courant* à la valeur du pointeur sur le début de la liste (*debut*).

La fonction *suivant* fournira en retour la valeur entière associée au noeud pointé par *courant* lorsqu'il existe (*courant* différent de NULL) ou la valeur 0 dans le cas contraire (il s'agit, là encore, d'une convention destinée à protéger l'utilisateur ayant appelé cette fonction, alors que la fin de liste était déjà atteinte et, donc, qu'aucun élément de l'ensemble n'était disponible). De plus, dans le premier cas (usuel), la fonction *suivant* actualisera la valeur de *courant*, de manière à ce qu'il pointe sur le noeud suivant.

Enfin, la fonction *existe* examinera simplement la valeur de *debut* pour savoir s'il existe encore un élément à traiter.

Voici la déclaration complète de notre nouvelle classe :

```
            /*         fichier SETINT4.H        */
            /* déclaration de la classe set_int */
struct noeud
{  int valeur ;                // valeur d'un élément de l'ensemble
   noeud * suivant ;           // pointeur sur le noeud suivant de la liste
} ;

class set_int
{
   noeud * debut ;             // pointeur sur le début de la liste
   int nelem ;                 // nombre courant d'éléments
   noeud * courant ;           // pointeur sur noeud courant
 public :
   set_int (int = 20) ;        // constructeur
   set_int (set_int &) ;       // constructeur par recopie
   ~set_int () ;               // destructeur
   void ajoute (int) ;         // ajout d'un élément
   int appartient (int) ;      // appartenance d'un élément
   int cardinal () ;           // cardinal de l'ensemble
   void init () ;              // initialisation itération
   int prochain () ;           // entier suivant
   int existe () ;             // test fin liste
} ;
```

Voici la définition des trois nouvelles fonctions membre *init*, *suivant* et *existe* :

```
void set_int::init ()
{  courant = debut ;
}


int set_int::prochain ()
{  if (courant)
      { int val = courant->valeur ;
        courant = courant->suivant ;
        return val ;
      }
     else return 0 ;    // par convention
}

int set_int::existe ()
{  return (courant != NULL) ;
}
```

Voici le nouveau programme d'utilisation demandé :

```
      /* utilisation de la classe set_int */
#include <iostream.h>
#include "setint1.h"
main()
```

```
{  void fct (set_int) ;
   set_int ens ;
   cout << "donnez 20 entiers \n" ;
   int i, n ;
   for (i=0 ; i<20 ; i++)
      { cin >> n ;
        ens.ajoute (n) ;
      }
   cout << "il y a : " << ens.cardinal () << " entiers différents\n" ;
   cout << "Ce sont : \n" ;
   ens.init () ;
   while (ens.existe ())
      cout << ens.prochain() << " " ;
}
```

CHAPITRE VI :
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.

a) 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.

Généralement, la fonction membre *fct* possèdera un argument ou une valeur de retour de type A (ce qui justifiera sa déclaration d'amitié). Pour traduire sa définition, le compilateur devra posséder les caractéristiques de A, donc disposer de sa déclaration.

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

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

La fonction membre *fct* 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 ci-dessus, le compilateur devra simplement "savoir" que B est une classe. Si la définition de la classe B n'a pas encore été compilée, on lui précisera simplement cette information par la déclaration :

```
class A ;
```

Généralement la fonction membre *fct* possèdera 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), le compilateur devra, là encore, savoir simplement que A est une classe. En revanche, pour compiler la définition de *fct*, le compilateur devra posséder les caractéristiques de A, donc disposer de sa déclaration.

c) 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 A est une classe par *class A ;* quant à la déclaration de la classe B, elle nécessitera généralement (dès qu'une de ses fonctions membre possèdera un argument ou une valeur de retour de type A) la déclaration de la classe A.

Exercice VI.1

Enoncé

Soit la classe *point* suivante :

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

Ecrire 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*. Ecrire 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 tranmettre 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 &) ;
```

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 &) ;
    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 <iostream.h>
#include "point1.h"          // nécessaire pour compiler affiche
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 VI.2

Enoncé

Soit la classe *vecteur3d* définie dans l'exercice IV.2 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 ;
      }
     .....
} ;
```

Ecrire une fonction indépendante *coincide*, amie de la classe *vecteur3d*, permettant de savoir si deux vecteurs ont mêmes composantes (cette fonction remplacera la fonction membre *coincide* qu'on demandait d'écrire dans l'exercice IV.2).

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 &) ;
```

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 ;
       }
} ;
```

et 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 VI.3

Enoncé

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 3x3 ; elle comportera un constructeur avec un argument (adresse d'un tableau de 3x3 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. Ecrire 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
} ;
```

Notez que nous avons déclaré constants les arguments de la fonction *matrice*. Il s'agit simplement d'une précaution destinée à prévenir le risque d'une faute de programmation conduisant, au sein de la définition de *matrice*, à une modification desdits arguments.

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

```
#include <iostream.h>
#include "vect1.h"
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 <iostream.h>
#include "mat1.h"
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 :

```
#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 () ;
}
```

CHAPITRE VII :
LA SURDEFINITION D'OPERATEURS

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 deuxième 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, en page 93).

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.

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

PLURALITE	OPERATEURS	ASSOCIATIVITE
Binaire	()[3] [][3] ->[2][3]	->
Unaire	+ - ++[5] --[5] ! ~ * &[1] new[4] delete[4] (cast)	<-
Binaire	* / %	->
Binaire	+ -	->
Binaire	<< >>	->
Binaire	< <= >	->
Binaire	== !=	->
Binaire	&	->
Binaire	^	->
Binaire	||	->
Binaire	&&	->
Binaire	|	->
Binaire	=[1][3] += -= *= /= %= &= ^= |= <<= >>=	<-
Binaire	,[2]	->

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

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

(2) Depuis la version 2.0 seulement.

(3) Doit être défini comme fonction membre.
(4) A un "niveau global" avant la version 2.0. Depuis la version 2.0, il peut, en outre, être surdéfini pour une classe ; dans ce cas, il doit l'être comme fonction membre.
(5) Jusqu'à la version 3.0, on ne peut pas distinguer entre les notations "pré" et "post". Depuis la version 3.0, ces opérateurs (lorsqu'ils sont définis de façon unaire) 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".

Remarque:

Même lorsque l'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 VII.1

Enoncé

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 à ce qu'ils permettent de tester la coïncidence ou la non-coïncidence de deux points :

- en utilisant des fonctions membre,

- en utilisant des fonctions amies.

Solution

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 utilisant n'importe laquelle des deux classes *vecteur3d* que nous venons de définir.

```
#include "vecteur3d.h"
#include <iostream.h>
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" ;
}
```

Exercice VII.2

Enoncé

Soit la classe *vecteur3d* définie ainsi :

```
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. 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 VII.3

Enoncé

Soit la classe *vecteur3d* définie ainsi :

```
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 [], le C++ ne permet de le surdéfinir que sous 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:

A titre indicatif, voici un petit exemple de programme faisant appel à notre classe *vecteur3d* :

```
#include "vecteur3d.h"
#include <iostream.h>
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] << " " ;
}
```

Exercice VII.4

Enoncé

L'exercice V.4 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 (ce ne serait pas le cas si l'on cherchait à utiliser une technique de "compteur de référence"). 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 nous traiterons une affectation telle que b = a, dans le cas où b sera 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.

Dans le cas où 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 dans ce cas.

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 pourra 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)
// 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) 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 V.4 convient parfaitement).

2) 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 VII.5

Enoncé

Considérer à nouveau la classe *set_int* créée dans l'exercice V.4 (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 à ce 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 manière à ce 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 a ajouté l'élément voulu.

Nous pouvons indifféremment utiliser ici 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 (ici, c'est possible 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 (abrs 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 que le pourrait être la définition de nos deux opérateurs (notez que nous utilisons [] pour définir <) :

```
        /* 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) ;
}
```

Exercice VII.6

Enoncé

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 à ce que :

- il permette d'accéder "normalement" à un élément d'un objet non constant de type *vecteur3d*, et ce, 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, a priori, de leur appliquer une fonction membre publique, sauf si cette dernière a été déclarée avec le qualificatif *const* (auquel cas, une telle fonction 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, en fonction de la présence ou de 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] ;
}
```

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

```
#include "vecteur3d.h"
#include <iostream.h>
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 VII.7

Enoncé

Définir une classe *vect* permettant de représenter des "vecteurs dynamiques", c'est-à-dire des vecteurs dont la dimension peut ne pas être connue 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 à ce 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.

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 :

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

Exercice VII.8

Enoncé

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 des tableaux 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).

a) Dire pourquoi il n'est pas possible, ici, de surdéfinir ici l'opérateur [] pour accéder à des éléments d'un tel tableau.

b) Surdéfinir, de façon appropriée, l'opérateur () de manière à ce 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.

Solution

a) L'opérateur [] 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*.

Compte tenu de ce que, 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, afin de pouvoir, dans ce nouveau type classe, surdéfinir à nouveau [] pour qu'il fournisse un résultat de type *int*. Il n'est donc pas possible d'obtenir le résultat souhaité sans définir d'autres classes que *int2d*.

b) 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* :

```
#include <iostream.h>
#include "int2d.h"
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" ;
       }
}
```

Remarques :

1) Dans la pratique, on sera amené à définir deux opérateurs () comme nous l'avions fait dans l'exercice précédent pour l'opérateur [], à savoir : 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".

Exercice VII.9

Enoncé

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 (fbat min, fbat 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 de manière telle 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 de manière telle 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
    float * 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, de surcroît, nous devons 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.h>
      /********************* 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 float [nint-1] ;              // 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)
{   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.h>
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"  ;
}
```

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

Exercice VII.10

Enoncé

Réaliser une classe nommée *stack_int* permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un emplacement allbué 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) :

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

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 ce rôle ici.

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 subit 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 ; ici, nous prévoirons une duplication complète de l'objet (une autre démarche, plus compliquée, consisterait à employer un "compteur de référence", ce qui permettrait de ne plus dupliquer la partie dynamique).

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 <iostream.h>
#include <stdlib.h>
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
    void operator = (stack_int &) ; // affectation
    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
```

```
} ;
```

Notez que 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) ;
}
```

A 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.h>
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 VII.11

Enoncé

Adapter la classe *point* :

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

de manière à ce 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 <iostream.h>
#include <stddef.h>               // pour size_t
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] ;
    }
    void operator delete (void * dp)
     { nptd-- ;
       ::delete (dp) ;
     }
    static int npt_tot ()
     { return npt ;
     }
    static int npt_dyn ()
     { return nptd ;
     }
} ;
```

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* :

```
#include "point.h"
main()
{
   point * ad1, * ad2 ;
   point a(3,5) ;
   cout << point::npt_tot () << " " << point::npt_dyn () << "\n" ;
   ad1 = new point (1,3) ;
   point b ;
   cout << point::npt_tot () << " " << point::npt_dyn () << "\n" ;
   ad2 = new point (2,0) ;
   delete ad1 ;
   cout << point::npt_tot () << " " << point::npt_dyn () << "\n" ;
   point c(2) ;
   delete ad2 ;
   cout << point::npt_tot () << " " <<  point::npt_dyn () << "\n" ;
}
```

CHAPITRE VIII :
LES CONVERSIONS DE TYPE DEFINIES PAR L'UTILISATEUR

RAPPELS

C++ vous permet de définir des conversions d'un type classe vers un autre type classe ou un type de base. On parle de conversions définies par l'utilisateur (en abrégé : C.D.U.). Ces conversions peuvent alors être éventuellement mises en oeuvre, de façon "automatique" par le compilateur, afin de donner une signification à un appel de fonction ou à un opérateur (alors que, sans conversion, l'appel ou l'opérateur serait illégal). On retrouve ainsi des possibilités comparables à celles qui nous sont offertes par le C en matière de conversions implicites.

Deux sortes de fonctions (obligatoirement des fonctions membre) permettent de définir des C.D.U. :

- les constructeurs à un argument (quelque soit le type de cet argument) : un tel constructeur réalise une conversion du type de cet argument dans le type de sa classe,

- les opérateurs de "cast" : dans une classe A, on définira un opérateur de conversion d'un type x (quelconque, c'est-à-dire aussi bien un type de base qu'un autre type classe) en introduisant la fonction membre de prototype :

```
operator x () ;
```

notez que le type de la valeur de retour (obligatoirement défini par son nom) ne doit pas figurer dans l'en-tête ou le prototype d'une telle fonction.

Les règles d'utilisation des C.D.U. rejoignent celles concernant le choix d'une fonction surdéfinie :

- les C.D.U. ne sont mises en oeuvre que si cela est nécessaire,

- une seule C.D.U. peut intervenir dans une chaîne de conversions (d'un argument d'une fonction ou d'un opérande d'un opérateur),

- il ne doit pas y avoir d'ambiguïté, c'est-à-dire plusieurs chaînes de conversions conduisant au même type, pour un argument ou un opérande donné.

Exercice VIII.1

Enoncé

Soit la classe *point* suivante :

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

a) La munir d'un opérateur de "cast" permettant de convertir un *point* en un entier (corrrespondant à son abscisse).

b) Soient alors ces déclarations :

```
point p ;
int n ;
void fct (int) ;
```

Que font ces instructions :

```
n = p ;       //  instruction 1
fct (p) ;     //  instruction 2
```

Solution

a) Il suffit de définir une fonction membre, de nom *operator int*, sans argument et renvoyant la valeur de l'abscisse du point l'ayant appelé. Rappelons que le type de la valeur de retour (que C++ déduit du nom de la fonction - ici *int*) ne doit pas figurer dans l'en-tête ou le prototype. Voici ce que pourraient être la déclaration et la définition de cette fonction, ici réunies en une seule déclaration d'une fonction "en ligne" :

```
operator int ()
{  return x ;
}
```

b)

L'instruction 1 est traduite par le compilateur en une conversion de p en *int* (par appel de *operator int*), suivie d'une affectation du résultat à n. Notez bien qu'il n'y a pas d'appel d'un quelconque opérateur d'affection de la classe *point*, ni d'un constructeur par recopie (car le seul argument transmis à la fonction *operator int* est l'argument implicite *this*).

L'instruction 2 est traduite par le compilateur en une conversion de p en *int* (par appel de *operator int*), suivie d'un appel de la fonction *fct*, à laquelle on transmet le résultat de cette conversion. Notez qu'il n'y pas, là non plus, d'appel d'un constructeur par recopie, ni pour *fct* (puisqu'elle reçoit un argument d'un type de base), ni pour *operator int* (pour la même raison que précédemment).

Exercice VIII.2

Enoncé

Quels résultats fournira le programme suivant :

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs, int ord)              // constructeur 2 arguments
     { x = abs ; y = ord ;
     }
   operator int()                        // "cast" point --> int
     { cout << "*** appel int() pour le point " << x << " " << y << "\n" ;
       return x ;
     }
} ;
main()
{
   point a(1,5), b(2,8) ;
   int n1, n2, n3 ;
   n1 = a + 3 ;                          // instruction 1
   cout << "n1 = " << n1 << "\n" ;
   n2 = a + b ;                          // instruction 2
   cout << "n2 = " << n2 << "\n" ;

   double z1, z2 ;
   z1 = a + 3 ;                          // instruction 3
   cout << "z1 = " << z1 << "\n" ;
   z2 = a + b ;                          // instruction 4
   cout << "z2 = " << z2 << "\n" ;
}
```

Solution

Lorsque le compilateur rencontre, dans l'instruction 1, l'expression *a+ 3*, il cherche tout d'abord s'il existe un opérateur surdéfini correspondant aux types *point* et *int* (dans cet ordre). Ici, il n'en trouve pas. Il va alors chercher à mettre en place des conversions permettant d'aboutir à une opération existante ; ici, l'opérateur de cast (*operator int*) lui permet de se ramener à une addition d'entiers (par conversion de p en *int*), et c'est la seule possibilité. Le résultat est alors, bien sûr, de type *int* et il est affecté à *n1* sans conversion.

Le même raisonnement s'applique à l'instruction 2 ; l'évaluation de a+ b se fait alors par conversion de a et de b en *int* (seule possibilité de donner une signification à l'opérateur +). Le résultat est de type *int* et il est affecté sans conversion à *n2*.

Les expressions figurant à droite des affectations des instructions 3 et 4 sont évaluées exactement suivant les mêmes règles que les expressions des instructions 1 et 2. Ce n'est qu'au moment de l'affectation du résultat (de type *int*) à la variable mentionée que l'on opère une conversion *int* --> *double* (analogue à celles qui sont mises en place en langage C).

A titre indicatif, voici ce que fournit précisément l'exécution du programme :

```
*** appel int() pour le point 1 5
n1 = 4
```

```
*** appel int() pour le point 1 5
*** appel int() pour le point 2 8
n2 = 3
*** appel int() pour le point 1 5
z1 = 4
*** appel int() pour le point 1 5
*** appel int() pour le point 2 8
z2 = 3
```

Exercice VIII.3

Enoncé

Quels résultats fournira le programme suivant :

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs, int ord)             // constructeur 2 arguments
     { x = abs ; y = ord ;
     }
   operator int()                       // "cast" point --> int
     { cout << "** appel int() pour le point " << x << " " << y << "\n" ;
       return x ;
     }
} ;
void fct (double v)
{  cout << "$$ appel fct avec argument : " << v << "\n" ;
}
main()
{
   point a(1,4) ;
   int n1 ;
   double z1, z2 ;
   n1 = a + 1.75 ;                  // instruction 1
   cout << "n1 = " << n1 << "\n" ;
   z1 = a + 1.75 ;                  // instruction 2
   cout << "z1 = " << z1 << "\n" ;
   z2 = a ;                         // instruction 3
   cout << "z2 = " << z2 << "\n" ;
   fct (a) ;                        // instruction 4
}
```

Solution

Pour évaluer l'expression *a + 1.75* de l'instruction 1, le compilateur met en place une chaîne de conversions de *a* en *double* (*point --> int* suivie de *int --> double*) de manière à aboutir à l'addition de deux valeurs de type *double* ; le résultat, de type *double*, est ensuite converti pour être affecté à *n1* (conversion forcée par l'affectation, comme d'habitude en C).

Notez bien qu'il n'est pas question pour le compilateur de prévoir la conversion en *int* de la valeur 1.75 (de façon à se ramener à l'addition de deux *int*, après conversion de a en *int*) car il s'agit là d'une "conversion dégradante" qui n'est jamais mise en oeuvre de manière implicite dans un calcul d'expression. Il n'y a donc pas d'autre choix possible (notez que s'il y en avait effectivement un autre, il ne s'agirait pas pour autant d'une situation d'ambiguïté dans la mesure où le compilateur appliquerait alors les règles habituelles de choix d'une fonction surdéfinie).

L'instruction 2 correspond à un raisonnement similaire avec cette seule différence que le résultat de l'addition (de type *double*) peut être affecté à *z1* sans conversion.

Enfin les instructions 3 et 4 entraînent une conversion de *point* en *double*, par une suite de conversions *point* --> *int* et *int* --> *double*.

Voici le résultat de l'exécution du programme :

```
** appel int() pour le point 1 4
n1 = 2
** appel int() pour le point 1 4
z1 = 2.75
** appel int() pour le point 1 4
z2 = 1
** appel int() pour le point 1 4
$$ appel fct avec argument : 1
```

Exercice VIII.4

Enoncé

Que se passerait-il si, dans la classe *point* du précédent exercice, nous avions introduit, en plus de l'opérateur *operator int*, un autre opérateur de cast *operator double*.

Solution

Dans ce cas, les instructions 1 et 2 auraient conduit à une situation d'ambiguïté. En effet, le compilateur aurait disposé de deux chaînes de conversions permettant de convertir un point en *double* : soit *point* --> *double* soit *point* --> *int* suivie de *int* -> *double*. En revanche, les instructions 3 et 4 auraient toujours été acceptées.

Exercice VIII.5

Enoncé

Considérer la classe suivante :

```
class complexe
{  double x, y ;
```

```
  public :
   complexe (double r=0, double i=0) ;
   complexe (complexe &) ;
} ;
```

Dans un programme contenant les déclarations :

```
complexe z (1,3) ;
void fct (complexe) ;
```

que produiront les instructions suivantes :

```
z = 3.75 ;          // instruction 1
fct (2.8) ;         // instruction 2
z = 2 ;             // instruction 3
fct (4) ;           // instruction 4
```

Solution

L'instruction 1 conduit une conversion de la valeur *double* 3.75 en un *complexe* par appel du constructeur à un argument (compte tenu des arguments par défaut), suivie d'une affectation à z.

L'instruction 2 conduit à une conversion de la valeur *double* 2.8 en un *complexe* (comme précédemment par appel du constructeur à un argument) ;il y aura ensuite création d'une copie, par appel du constructeur par recopie de la classe *complexe*, copie qui sera transmise à la fonction *fct*.

Les instructions 3 et 4 jouent le même rôle que les deux précédentes, avec cette seule différence qu'elles font intervenir des conversions *int* --> *complexe* obtenues par une conversion *int* --> *double* suivie d'une conversion *double* --> *complexe*.

Exercice VIII.6

Enoncé

Quels résultats fournira le programme suivant :

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)         // constructeur 0, 1 ou 2 arguments
     { x = abs ; y = ord ;
       cout << "$$ construction point : " << x << " " << y << "\n" ;
     }
   friend point operator + (point, point) ;      // point + point --> point
   void affiche ()
     { cout << "Coordonnées : " << x << " " << y << "\n" ;
     }
} ;
point operator+ (point a, point b)
```

```
{  point r ;
   r.x = a.x + b.x ; r.y = a.y + b.y ;
   return r ;
}
main()
{
   point a, b(2,4) ;
   a = b + 6 ;          // affectation 1
   a.affiche() ;
   a = 6 + b ;          // affectation 2
   b.affiche() ;
}
```

Que se passerait-il si l'opérateur + avait été surdéfini comme une fonction membre et non plus comme une fonction amie ?

Solution

L'évaluation de l'expression *b+ 6* de la première affectation se fait en utilisant l'opérateur + surdéfini pour la classe *point*, après avoir converti l'entier 6 en un *point* (par appel du constructeur à un argument). Le résultat, de type *point*, est alors affecté à a.

L'instruction d'affectation 2 se déroule de façon similaire (conversion de l'entier 6 en un *point*).

En revanche, si l'opérateur + avait été surdéfini par une fonction membre, la seconde instruction d'affectation aurait été rejetée à la compilation ; en effet, elle aurait été interprétée comme :

```
6.operator + (b)
```

On voit ici que seule la fonction amie permet de traiter de façon identique les deux opérandes d'un opérateur binaire, notamment en ce qui concerne les possibilités de conversions implicites.

Exercice VIII.7

Enoncé

Soient les deux classes suivantes :

```
class A
{    // ...
   friend operator + (A, A) ;
 public :
   A (int) ;     // constructeur à un argument entier
   A (B) ;       // constructeur à un argument de type B
     // ...
} ;

class B
{    // ...
 public :
   B (int) ;     // constructeur à un argument entier
```

```
    // ...
} ;
```

a) Dans un programme contenant les déclarations :

```
A a1, a2, a3 ;
B b1, b2, b3 ;
```

les instructions suivantes seront-elles correctes et, si oui, que feront-elles ?

```
a1 = b1 ;         // instruction 1
b1 = a1 ;         // instruction 2
a3 = a1 + a2 ;    // instruction 3
a3 = b1 + b2 ;    // instruction 4
b3 = a1 + a2 ;    // instruction 5
```

b) Comment obtenir le même résultat, sans définir, dans A, le constructeur A(B) ?

Solution

a)

Instruction 1 :

Il faut distinguer 2 cas :

Si A n'a pas surdéfini d'opérateur d'affectation de la forme *A& operator = (B)*, il y aura conversion de b1 dans le type de A (par appel du constructeur A(B)), suivie d'une affectation à a1 (en utilisant soit l'opérateur d'affectation par défaut de A, soit éventuellement celui qui aurait pu y être surdéfini).

En revanche, si A a surdéfini un opérateur d'affectation de la forme *A& operator = (B)* (permettant d'affecter un objet de type B à un objet de type A), ce dernier sera utilisé pour réaliser l'instruction 1, et, dans ce cas, il n'y aura donc pas d'appel de constructeur de A, ni d'éventuel autre opérateur d'affectation. Ce comportement s'explique par le fait que les C.D.U. ne sont mises en oeuvre que lorsque cela est nécessaire.

Instruction 2 :

Là encore, il faut distinguer les deux cas précédents. Si B n'a pas surdéfini d'opérateur d'affectation de la forme *A& operator = (B)*, l'instruction 2 conduira à une erreur de compilation puisqu'il n'existera aucune possibilité de convertir un objet de type A en un objet de type B. En revanche, si l'opérateur d'affectation en question existe, il sera tout simplement utilisé.

Instruction 3 :

Elle ne pose aucun problème particulier.

Instruction 4 :

Ici, b1 et b2 seront convertis dans le type A en utilisant le constructeur A(B) avant d'être transmis à l'opérateur + (de A). Le résultat, de type A, sera affecté à a3, en utilisant l'opérateur d'affectation de A - soit celui par défaut, soit celui éventuellement surdéfini.

Instruction 5 :

L'expression a1 + a2 ne pose pas de problème puisqu'elle est évaluée à l'aide de l'opérateur + de la classe A ; elle fournit un résultat de type A. En revanche, pour l'affectation du résultat à b3, il faut à nouveau distinguer les 2 cas déjà évoqués. Si B a surdéfini un opérateur d'affectation de la forme *A & operator = (B)*, il sera utilisé ; si un tel opérateur n'a pas été surdéfini, l'instruction sera rejetée par le compilateur (puisqu'il n'existera alors aucune possibilité de conversion de B en A).

b) En introduisant, dans la classe B, un opérateur de "cast" permettant la conversion de B en A, de la forme :

```
operator B ()
```

CHAPITRE IX :
LA TECHNIQUE DE L'HERITAGE

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 dans le chapitre III), 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, moyennant l'utilisation de 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 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
                   // (possible uniquement grâce aux règles de compatibilité
                   //    entre classe de base et classe dérivée)
```

- 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 obtiendra 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 dans le cas où 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.

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 IX.1

Enoncé

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 *pointa*, 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* ?

Solution

a) Il suffit de prévoir, dans la déclaration de *point*, 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 définitions de *point* et 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
   }
```

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.

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 "protégés", 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
```

d) Un objet de type *point* 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 IX.2

Enoncé

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

```
#include <iostream.h>
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 *col*, de type *int*, destiné à contenir 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) ;
```

Solution

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* y accède 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"
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*),

- du constructeur de *pointcol*.

Exercice IX.3

Enoncé

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"
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, compte tenu de ce que p est un membre privé de *pointcol*.

Exercice IX.4

Enoncé

Soit une classe *point* définie ainsi (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
}  ;
```

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) Même question, en supposant, cette fois, que l'opérateur + a été défini au sein de *point*, sous forme d'une fonction membre.

Solution

a) Les 5 instructions proposées sont correctes. D'une manière générale, *x == y* est interprétée comme *operator == (x, y)*. Si x et y sont de type *point*, aucun problème ne se pose bien sûr. Si l'un des opérandes (ou les deux) est de type *pointcol*, il sera (ils seront) 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 les coordonnées. Pour un entier, elle revient à le considérer comme un *point* ayant cet entier pour abscisse et une ordonnée nulle.

b) Cette fois, *x == y* est interprétée 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 est ramené au 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.

Exercice IX.5

Enoncé

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 supppose 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, dans la mesure où 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 IX.6

Enoncé

Soit une classe *int2d* (telle que celle créée dans l'exercice VII.8) permettant de manipuler des "tableaux dynamiques" d'entiers à deux dimensions dont la déclaration (fournie dans le fichier *int2d.h*) se présente ainsi :

```
      /****** fichier int2d.h :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"
} ;
```

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)     // rédéfinition 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 IX.7

Enoncé

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*, de manière à ce que l'affectation et la transmission par valeur d'objets de type *vectok* se déroule convenablement. Pour faciliter l'utilisation de cette nouvelle classe, introduire une fonction membre *taille* fournissant la dimension d'un vecteur.

Ecrire 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 VII.4.

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)
{  int i ;
   for (i=0 ; i<nelem ; i++)  adr[i] = v.adr[i] ;
}
     /***** définition de l'affectation entre vectok *****/
vectok & vectok::operator = (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) ;
}
```

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)
{  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).

Voici un exemple de programme utilisant la classe *vectok* :

```
#include <iostream.h>
```

```
#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] << " " ;
}
```

CHAPITRE X :
L'HERITAGE MULTIPLE

RAPPELS

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

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

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

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

Appel des constructeurs et des destructeurs

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

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

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

Classes virtuelles

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

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

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

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

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

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

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

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

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

Exercice X.1

Enoncé

Quels seront les résultats fournis par ce programme :

```
#include <iostream.h>
class A
{   int n ;
    float x ;

  public :
    A (int p = 2)
      { n = p ; x = 1 ;
        cout << "** construction objet A : " << n << " " << x << "\n" ;
      }
} ;

class B
{   int n ;
    float y ;
  public :
    B (float v = 0.0)
      { n = 1 ;  y = v ;
        cout << "** construction objet B : " << n << " " << y << "\n" ;
      }
} ;

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

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

Solution

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

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

```
** construction objet C : 12 21
```

Exercice X.2

Enoncé

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

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

Solution

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

```
** construction objet B : 1 0
** construction objet A : 2 1
** construction objet C : 3 3
** construction objet B : 1 5
** construction objet A : 2 1
** construction objet C : 12 21
```

Exercice X.3

Enoncé

Même question que dans l'exercice X.1, en supposant que l'en-tête du constructeur de C est la suivante :

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

Solution

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

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

Exercice X.4

Enoncé

Quels seront les résultats fournis par ce programme :

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

class B : public A
{   float xb ;
  public :
    B (float xx=0.0)
      { xb = xx ;
        cout << "$$construction objet B " << xb << "\n" ;
      }
} ;

class C : public A
{   int nc ;
  public :
    C (int nn= 2) : A (2*nn+1)
      { nc = nn ;
        cout << "$$construction objet C " << nc << "\n" ;
      }
} ;

class D : public B, public C
{   int nd ;
  public :
    D (int n1, int n2, float x) : C (n1), B (x)
     { nd = n2 ;
       cout << "$$construction objet D " << nd << "\n" ;
     }
} ;

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

Solution

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

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

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

Exercice X.5

Enoncé

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

Solution

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

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

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

Celle de C est transformée de façon analogue :

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

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

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

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

```
$$construction objet A 21
$$construction objet B 5
```

```
$$construction objet C 10
$$construction objet D 20
```

Exercice X.6

Enoncé

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

dessin à reprendre dans "Programmer en Turbo C++ "

(C. DE LANNOY) page 321

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

```
struct element                          // structure d'un élément de liste
{ element * suivant ;                   // pointeur sur l'élément suivant
  void * contenu ;                      // pointeur sur un objet quelconque
} ;
class liste
{ element * debut ;                     // pointeur sur premier élément
         // autres membres données éventuels
 public :
  liste () ;                            // constructeur
  ~liste () ;                           // destructeur
  void ajoute (void *) ;                // ajoute un élément en début de liste
  void * premier () ;                   // positionne sur premier élément
  void * prochain () ;                  // positionne sur prochain élément
  int fini () ;
} ;
```

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

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

- *prochain*, qui fournira l'adresse de l'information associée au "prochain noeud" ;des appels successifs de *prochain* devront permettre de parcourir la liste (sans qu'il soit nécessaire d'appeler une autre fonction),

- *fini*, qui permettra de savoir si la fin de liste est atteinte ou non.

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

2) Soit la classe *point* suivante :

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

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

- ajouter un *point* en début d'une telle liste,

- disposer d'une fonction membre *affiche* affichant les informations associées à chacun des points de la liste de points.

3) Ecrire un petit programme d'essai.

Solution

1) Manifestement, les fonctions *premier* et *prochain* nécessitent un "pointeur sur un élément courant". Il sera membre donnée de la classe *liste*.

Nous conviendrons (classiquement) que la fin de liste est matérialisée par un noeud comportant un pointeur "nul". La classe *liste* devra disposer d'un constructeur dont le rôle se limitera à l'initialiser à une "liste vide", ce qui s'obtiendra simplement en plaçant un pointeur nul comme adresse de début de liste (cette façon de procéder simplifie grandement l'algorithme d'ajout d'un élément en début de liste, puisqu'elle évite d'avoir à distinguer des autres le cas de la liste vide).

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

Voici ce que pourra être notre classe *liste* complète :

```
struct element                          // structure d'un élément de liste
{ element * suivant ;                   // pointeur sur l'élément suivant
  void * contenu ;                      // pointeur sur un objet quelconque
} ;
class liste
{ element * debut ;                     // pointeur sur premier élément
  element * courant ;                   // pointeur sur élément courant
 public :
  liste ()                              // constructeur
```

```
    { debut = NULL ;
      courant = debut ;                  //   par sécurité
    }
  ~liste () ;                            // destructeur
  void ajoute (void *) ;                 // ajoute un élément en début de liste
  void * premier ()                      // positionne sur premier élément
    { courant = debut ; return (courant->contenu) ; }
  void * prochain ()                     // positionne sur prochain élément
    { if (courant != NULL) courant = courant->suivant ;
      return (courant->contenu) ;
    }
  int fini () { return (courant == NULL) ; }
} ;
liste::~liste ()
{ element * suiv ;
  courant = debut ;
  while (courant != NULL )
   { suiv = courant->suivant ; delete courant ; courant = suiv ; }
}
void liste::ajoute (void * chose)
{ element * adel = new element ;
  adel->suivant = debut ;
  adel->contenu = chose ;
  debut = adel ;
}
```

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

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

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

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

- aux fonctions *premier*, *prochain* et *fini* de la classe *liste* pour le parcours de la liste de points,
- à la fonction *affiche* de la classe *point* pour l'affichage d'un point.

Nous aboutissons à ceci :

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

3) Exemple de programme d'essai :

```
#include "listepts.h"
main()
{ liste_points l ;
  point a(2,3), b(5,9), c(0,8) ;
```

```
  l.ajoute (&a) ; l.affiche () ; cout << "---------\n" ;
  l.ajoute (&b) ; l.affiche () ; cout << "---------\n" ;
  l.ajoute (&c) ; l.affiche () ; cout << "---------\n" ;
}
```

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

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

CHAPITRE XI :
LES FONCTIONS VIRTUELLES

RAPPELS

Typage statique des objets (ou liaison dynamique des fonctions)

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

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

A * pta ;
B * ptb ;
```

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

Les fonctions virtuelles

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

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

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

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

N.B. : il peut y avoir liaison dynamique, même en dehors de l'utilisation de pointeurs (voyez, par exemple, l'exercice XI.2).

Règles

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

Les fonctions virtuelles pures

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

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

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

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

Exercice XI.1

Enoncé

Quels résultats produira ce programme :

```
#include <iostream.h>
class point
{ protected :            // pour que x et y soient accessibles à pointcol
   int x, y ;
```

```
  public :
   point (int abs=0, int ord=0) { x=abs ; y=ord ; }
   virtual void affiche ()
     { cout << "Je suis un point \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y << "\n" ;
     }
} ;

class pointcol : public point
{  short couleur ;
  public :
   pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
     { couleur = cl ;
     }
   void affiche ()
     { cout << "Je suis un point coloré \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y ;
       cout << "   et ma couleur est :    " << couleur << "\n" ;
     }
} ;
main()
{  point p(3,5) ; point * adp = &p ;
   pointcol pc (8,6,2) ; pointcol * adpc = &pc ;
   adp->affiche () ; adpc->affiche () ;      // instructions 1
   cout << "-----------------\n" ;
   adp = adpc ;
   adp->affiche () ; adpc->affiche () ;      // instructions 2
}
```

Solution

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

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

Voici les résultats complets fournis par le programme :

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

Exercice XI.2

Enoncé

Quels résultats produira ce programme :

```
#include <iostream.h>
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  virtual void identifie ()
    { cout << "Je suis un point \n" ;
    }
  void affiche ()
    { identifie () ;
      cout << "Mes coordonnées sont : " << x << " " << y << "\n" ;
    }
} ;

class pointcol : public point
{ short couleur ;
 public :
  pointcol (int abs=0, int ord=0, int cl=1 ) : point (abs, ord)
    { couleur = cl ;
    }
  void identifie ()
    { cout << "Je suis un point coloré de couleur : " << couleur << "\n" ;
    }
} ;



main()
{ point p(3,4) ;
  pointcol pc(5,9,5) ;
  p.affiche () ;
  pc.affiche () ;
  cout << "---------------\n" ;
  point * adp = &p ;
  pointcol * adpc = &pc ;
  adp->affiche () ; adpc->affiche () ;
  cout << "---------------\n" ;
  adp = adpc ;
  adp->affiche () ; adpc->affiche () ;
}
```

Solution

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

de type *point* appelle *affiche*, cette dernière fera toujours appel à la fonction *identifie* de *point* (le même résultat serait obtenu sans liaison dynamique).

Voici les résultats complets fournis par notre programme :

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

Exercice XI.3

Enoncé

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

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

La classe *ens_heter* disposera des fonctions membre suivantes :

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

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

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

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

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

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

Solution

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

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

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

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

Les mêmes réflexions s'appliquent à la valeur de retour de la fonction *suivant*.

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

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

 public :
   ens_heter (int=20) ;       // constructeur
   ~ens_heter () ;            // destructeur
   void ajoute (base &) ;     // ajout d'un élément
   int appartient (base &) ;  // appartenance d'un élément
   int cardinal () ;          // cardinal de l'ensemble
   void init () ;             // initialisation itération
   base & suivant () ;        // passage élément suivant
   int existe () ;            //
   void liste () ;            // affiche les "valeurs" des différents éléments
} ;
```

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

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

```
} ;
```

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

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

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

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

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

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

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

```
#include <iostream.h>
#include "ens_heter.h"
#include "base.h"
```

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

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


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

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

Remarque :

Si l'on cherchait à résoudre les problèmes posés par l'affectation et la transmission par valeur d'objets de type *ensemble*, on serait amené à choisir entre deux méthodes :

- effectuer des "copies complètes" (on dit souvent "profondes") de l'objet, c'est-à-dire en tenant compte, non seulement de ses membres, mais aussi de ses parties dynamiques,

- utiliser un "compteur de référence" associé à chaque partie dynamique de façon à en éviter la duplication.

Or, la première méthode conduirait effectivement à recopier la partie dynamique de l'ensemble, c'est-à-dire le tableau de pointeurs d'adresse *base* *. Mais, que faudrait-il faire pour les éléments eux-mêmes (pointés par les différentes valeurs du dit tableau)? Même si l'on admet qu'il faut également les dupliquer, il n'est pas possible de le faire sans connaître la "structure" des objets concernés, à moins d'imposer à chaque objet de disposer de fonctions appropriées permettant d'en réaliser une copie profonde.

La deuxième méthode pose des problèmes voisins en ce qui concerne le traitement à appliquer aux objets qui sont des éléments d'un ensemble de type *ensemble*. Disposent-ils d'un compteur de référence ? Si oui, comment le mettre à jour ?

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

CHAPITRE XII :
LES FLOTS D'ENTREE ET DE SORTIE

RAPPELS

Un flot est un "canal recevant (flot "d'entrée") ou fournissant (flot de "sortie") de l'information. Ce canal est associé à un périphérique ou à un fichier. Un flot d'entrée est un objet de type *istream* tandis qu'un flot de sortie est un objet de type *ostream*. Le flot *cout* est un flot de sortie prédéfini, connecté à la sortie standard *stdout* ;de même, le flot *cin* est un flot d'entrée prédéfini, connecté à l'entrée standard *stdin*. Les informations fournies ici sont valables à partir de la version 2.0.

La classe ostream

Elle surdéfinit l'opérateur < < sous la forme d'une fonction membre :

```
ostream & operator << (expression)
```

L'expression correspondant à son deuxième opérande peut être d'un type de base quelconque, y compris *char*, *char* *(on obtient la chaîne pointée) ou un pointeur sur un type quelconque autre que *char* (on obtient la valeur du pointeur) ;pour obtenir la valeur de l'adresse d'une chaîne, on la convertit artificiellement en un pointeur de type *void* *.

Fonctions membre :

ostream & put(char c) : transmet au flot correspondant le caractère c.

ostream & write (void *adr, int lng) : envoie *lng* caractères, prélevés à partir de l'adresse *adr*.

La classe istream

Elle surdéfinit l'opérateur > > sous forme d'une fonction membre :

```
istream & operator >> (& type_de_base)
```

Le *type_de_base* peut être quelconque, pour peu qu'il ne s'agisse pas d'un pointeur (*char** est cependant accepté - il correspond à l'entrée d'une chaîne de caractères, et non d'une adresse).

Les "espaces_blancs" (espace, tabulation horizontale \t ou verticale \v, fin de ligne \n, retour chariot \r et changement de page \f) servent de "délimiteurs" (comme dans *scanf*), y compris pour les chaînes de caractères.

Principales fonctions membres :

istream & get(char & c) : extrait un caractère du flot d'entrée et le range dans c.

int get() : extrait un caractère du flot d'entrée et en renvoie la valeur (sous forme d'un entier) ; fournit *EOF* en cas de fin de fichier.

istream & read (void *adr, int taille) lit *taille* caractères sur le flot et les range à partir de l'adresse *adr*.

La classe iostream

Elle est dérivée de *istream* et *ostream*. Elle permet de réaliser des entrées sorties "conversationnelles".

Le statut d'erreur d'un flot

A chaque flot est associé un ensemble de bits d'un entier formant le "statut d'erreur du flot".

Les bits d'erreur :

La classe *ios* (dont dérivent *istream* et *ostream*) définit les constantes suivantes :

eofbit : fin de fichier (le flot n'a plus de caractères disponibles),

failbit : la prochaine opération sur le flot ne pourra pas aboutir,

badbit : le flot est dans un état irrécupérable,

goodbit (valant, en fait 0) : aucune des errreurs précédentes.

Une opération sur le flot a réussi lorsque l'un des bits *goodbit* ou *eofbit* est activé. La prochaine opération sur le flot ne pourra réussir que si *goodbit* est activé (mais il n'est pas certain qu'elle réussisse !).

Lorsqu'un flot est dans un état d'erreur, aucune opération ne peut aboutir tant que la condition d'erreur n'a pas été corrigée et que le bit d'erreur correspondant n'a pas été remis à zéro (à l'aide de la fonction *clear*).

Accès aux bits d'erreur : *ios* contient 5 fonctions membre :

eof() : valeur de *eofbit*,

bad() : valeur de *badbit*,

fail() : valeur de *failbit*,

good () : 1 si aucun bit du statut d'erreur n'est activé,

rdstate () : valeur du statut d'erreur (entier).

Modification du statut d'erreur :

void clear (int i=0) donne la valeur i au statut d'erreur. Pour activer un seul bit (par exemple *badbit*), on procédera ainsi (*fl* étant un flot) :

```
fl.clear (ios::badbit | fl rdstate() ) ;
```

Surdéfinition de ( ) et de !

Si fl est un flot, **(fl)** est vrai si aucun des bits d'erreur n'est activé (c'est-à-dire si *good* est vrai) ; de même, **!fl** est vrai si un des bits d'erreur précédents est activé (c'est-à-dire si *good* est faux).

Surdéfinition de < < et > > pour des types classe

On surdéfinira < < et > > pour une classe quelconque, sous forme de fonctions amies, en utilisant ces "canevas" :

```
ostream & operator << (ostream sortie, type_classe objet[1])
{
     // Envoi sur le flot sortie des membres de objet en utilisant
     // les possibilités classiques de << pour les types de base
     // c'est-à-dire des instructions de la forme :
     //     sortie << ..... ;
   return sortie ;
}

istream & operator >> (istream & entree, type_de_base & objet)
{
     // Lecture des informations correspondant aux différents membres de objet
     // en utilisant les possibilités classiques de >> pour les types de base
     // c'est-à-dire des instructions de la forme :
     //     entree >> ..... ;
   return entree ;
}
```

Le mot d'état du statut de formatage

A chaque flot, est associé un "statut de formatage" constitué d'un mot d'état et de 3 valeurs numériques (gabarit, précision et caractère de remplissage).

Voici (page ci-conre) les principaux bits du mot d'état :

NOM DE CHAMP (s'il existe)	NOM DU BIT	SIGNIFICATION (quand activé)
ios::basefield	ios::dec	conversion décimale
	ios::oct	conversion octale
	ios::hex	conversion hexadécimale

[1] Ici, la transmission peut se faire par valeur ou par référence.

	ios::showbase	affichage indicateur de base (en sortie)
	ios::showpoint	affichage point décimal (en sortie)
ios::floatfield	ios::scientific	notation "scientifique"
	ios::fixed	notation "point fixe"

Le mot d'état du statut de formatage (partie I)

Action sur le statut de formatage

On peut utiliser, soit des "manipulateurs" qui peuvent être "simples" ou "paramétriques", soit des fonctions membre.

a) Les manipulateurs non paramétriques

Ils s'emploient sous la forme :

```
flot << manipulateur

flot >> manipulateur
```

Les principaux manipulateurs non paramétriques sont présentés en page suivante :

MANIPULATEUR	UTILISATION	ACTION
dec	Entrée/Sortie	Active le bit de conversion décimale
hex	Entrée/Sortie	Active le bit de conversion hexadécimale
oct	Entrée/Sortie	Active le bit de conversion octale
endl	Sortie	Insère un saut de ligne et vide le tampon
ends	Sortie	Insère un caractère de fin de chaîne (\0)

Les principaux manipulateurs non paramétriques

b) Les manipulateurs paramétriques

Ils s'utilisent sous la forme :

```
istream & manipulateur (argument)
```

```
ostream & manipulateur (argument)
```

Voici les principaux :

MANIPULATEUR	UTILISATION	ROLE
setbase (int)	Entrée/Sortie	Définit la base de conversion
setprecision (int)	Entrée/Sortie	Définit la précision des nombres flottants
setw (int)	Entrée/Sortie	Définit le gabarit. Il retombe à 0 après chaque opération

Les principaux manipulateurs paramétriques

Association d'un flot à un fichier

La classe *ofstream*, dérivant de *ostream* permet de créer un flot de sortie associé à un fichier :

```
ofstream flot (char * nomfich, mode_d_ouverture)
```

La fonction membre *seekp (déplacement, origine)* permet d'agir sur le pointeur de fichier.

De même, la classe *ifstream*, dérivant de *istream*, permet de créer un flot d'entrée associé à un fichier :

```
ifstream flot (char * nomfich, mode_d_ouverture)
```

La fonction membre *seekg (déplacement, origine)* permet d'agir sur le pointeur de fichier

Dans tous les cas, la fonction *close* permet de fermer le fichier.

L'utilisation des classes *ofstream* et *ifstream* demande l'inclusion du fichier *fstream.h*.

Modes d'ouverture d'un fichier

BIT DE MODE D'OUVERTURE	ACTION
ios::in	Ouverture en lecture (obligatoire pour la classe ifstream)
ios::out	Ouverture en écriture (obligatoire pour la classe ofstream)
ios::app	Ouverture en ajout de données (écriture en fin de fichier)
ios::ate	Se place en fin de fichier après ouverture
ios::trunc	Si le fichier existe, son contenu est perdu (obligatoire si ios::out est activé sans ios::ate ni ios::app
ios::nocreate	Le fichier doit exister
ios::noreplace	Le fichier ne doit pas exister (sauf si ios::ate ou ios::app est activé)

Les différents modes d'ouverture d'un fichier

Exercice XI.1

Enoncé

Ecrire un programme qui lit un nombre réel et qui en affiche le carré sur un "gabarit" minimal de 12 caractères, de 22 façons différentes :

- en "point fixe", avec un nombre de décimales variant de 0 à 10,
- en notation scientifique, avec un nombre de décimales variant de 0 à 10.

Dans tous les cas, on affichera les résultats avec cette présentation :

```
précision de xx chiffres : ccccccccccc
```

Solution

Il faut donc activer, d'abord le bit *fixed*, ensuite le bit *scientific* du champ *fbatfield*. Nous utiliserons la fonction *setf*, membre de la classe *ios*. Notez bien qu'il faut éviter d'écrire, par exemple :

```
setf (ios::fixed) ;
```

ce qui aurait pour effet d'activer le bit *fixed*, sans modifier les autres donc, en particulier, sans modifier les autres bits du champ *fbatfield*.

Le gabarit d'affichage est déterminé par le manipulateur *setw*. Notez qu'il faut transmettre ce manipulateur au fbt concerné, juste avant d'afficher l'information voulue.

```
#include <iostream.h>
#include <iomanip.h>       //pour les "manipulateurs paramétriques"
main()
{  float val, carre ;
   cout << "donnez un nombre réel : " ;
   cin  >> val ;
   carre = val*val ;
   cout << "Voici son carré : \n" ;
   int i ;
   cout << "  en notation point fixe : \n" ;
   cout.setf (ios::fixed, ios::floatfield) ;   // met à 1 le bit ios::fixed
                                               // du champ ios::floatfield
   for (i=0 ; i<10 ; i++)
      cout << "      précision de " << setw (2) << i << " chiffres : "
           << setprecision (i) << setw (12) << carre << "\n" ;
   cout << "  en notation scientifique : \n" ;
   cout.setf (ios::scientific, ios::floatfield) ;
   for (i=0 ; i<10 ; i++)
      cout << "      précision de " << setw (2) << i << " chiffres : "
           << setprecision (i) << setw (12) << carre << "\n" ;
}
```

Exercice XII.2

Enoncé

Soit la classe *point* suivante :

```
class point
{   int x, y ;
   public :
    // fonctions membre
} ;
```

Surdéfinir les opérateurs << et >> , de manière à ce qu'il soit possible de lire un point sur un flot d'entrée ou d'écrire un point sur un flot de sortie. On prévoira qu'un tel point soit représenté sous la forme :

```
<entier, entier>
```

avec éventuellement des séparateurs "espaces_blancs" supplémentaires, de part et d'autre des nombres entiers.

Solution

Nous devons donc surdéfinir les opérateurs << et >> pour qu'ils puissent recevoir, en deuxième opérande, un argument de type *point*. Il ne pourra s'agir que de fonctions amies, dont les prototypes se présenteront ainsi :

```
ostream & operator << (ostream &, point) ;

istream & operator >> (istream &, point) ;
```

L'écriture de *operator* << ne présente pas de difficultés particulières : on se contente d'écrire, sur le flot concerné, les coordonnées du point, accompagnées des symboles < et> .

En revanche, l'écriture de *operator* >> nécessite un peu plus d'attention. En effet, il faut s'assurer que l'information se présente bien sous la forme requise et, si ce n'est pas le cas, prévoir de donner au flot concerné l'état *bad*, afin que l'utilisateur puisse savoir que l'opération s'est mal déroulée (en testant "nature lement" l'état du flot).

Voici ce que pourrait être la déclaration de notre classe (nous l'avons simplement munie d'un contructeur) et la définition des deux fonctions amies voulues :

```
#include <iostream.h>

class point
{  int x, y ;
  public :
   point (int abs=0, int ord=0)
     { x = abs ; y = ord ; }
   int abscisse () { return x ; }
   friend ostream & operator << (ostream &, point) ;
   friend istream & operator >> (istream &, point &) ;
```

```
} ;

ostream & operator << (ostream & sortie, point p)
{ sortie << "<" << p.x << "," << p.y << ">" ;
  return sortie ;
}



istream & operator >> (istream & entree, point & p)
{  char c = '\0' ;
   float x, y ;
   int ok = 1 ;
   entree >> c ;
   if (c != '<') ok = 0 ;
      else
       { entree >> x >> c ;
         if (c != ',') ok = 0 ;
            else
               { entree >> y >> c ;
                 if (c != '>') ok = 0 ;
               }
       }
   if (ok) { p.x = x ; p.y = y ; }         // on n'affecte à p que si tout est OK
      else entree.clear (ios::badbit | entree.rdstate () ) ;
   return entree ;
}
```

A titre indicatif, voici un petit programme d'essai, accompagné d'un exemple d'exécution :

```
main()
{ char ligne [121] ;
  point a(2,3), b ;
  cout << "point a : " << a << "  point b : " << b << "\n" ;
  do
   { cout << "donnez un point : " ;
     if (cin >> a)  cout << "merci pour le point : " << a << "\n" ;
             else { cout << "** information incorrecte \n" ;
                    cin.clear () ;
                    cin.getline (ligne, 120, '\n') ;
                  }
   }
  while ( a.abscisse () ) ;
}
```

```
point a : <2,3>  point b : <0,0>
donnez un point : 4,5
** information incorrecte
donnez un point : <4,5<
** information incorrecte
donnez un point : <4,5>
merci pour le point : <4,5>
donnez un point : < 8,   9     >
merci pour le point : <8,9>
```

```
donnez un point : bof
** information incorrecte
donnez un point : <0,0>
merci pour le point : <0,0>
```

Exercice XII.3

Enoncé

Ecrire un programme qui enregistre (sous forme "binaire", et non pas formatée), dans un fichier de nom fourni par l'utilisateur, une suite de nombres entiers fournis sur l'entrée standard. On conviendra que l'utilisateur fournira la valeur 0 (qui ne sera pas enregistrée dans le fichier) pour préciser qu'il n'a plus d'entiers à entrer.

Solution

Si *nomfich* désigne une chaîne de caractères, la déclaration :

```
ofstream sortie (nomfich, ios::out) ;
```

permet de créer un flot de nom *sortie*, de l'associer au fichier dont le nom figure dans *nomfich* et d'ouvrir ce fichier en écriture.

L'écriture dans le fichier en question se fera par la fonction *write*, appliquée au flot *sortie*.

Voici le programme demandé :

```
const int LGMAX = 20 ;
#include <stdlib.h>                        // pour exit
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier à créer : " ;
   cin >> setw (LGMAX) >> nomfich ;
   ofstream sortie (nomfich, ios::out) ;
   if (!sortie) { cout << "création impossible \n" ;
                  exit (1) ;
                }
   do
      { cout << "donnez un entier : " ;
        cin >> n ;
        if (n) sortie.write ((char *)&n, sizeof(int) ) ;
      }
```

```
   while (n && sortie) ;

   sortie.close () ;
}
```

Notez que *if(!sortie)* est équivalent à *if(!sortie.good())* et que *while (n && sortie)* est équivalent à *while (n && sortie.good())*.

Exercice XII.4

Enoncé

Ecrire un programme permettant de lister (sur la sortie standard) les entiers contenus dans un fichier tel que celui créé par l'exercice précédent.

Solution

```
const int LGMAX = 20 ;
#include <stdlib.h>                        // pour exit
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier à lister : " ;
   cin >> setw (LGMAX) >> nomfich ;
   ifstream entree (nomfich, ios::in) ;
   if (!entree) { cout << "ouverture impossible \n" ;
                  exit (1) ;
                }
   while ( entree.read ( (char*)&n, sizeof(int) ) )
        cout << n << "\n" ;

   entree.close () ;
}
```

Exercice XII.5

Enoncé

Ecrire un programme permettant à un utilisateur de retrouver, dans un fichier tel que celui créé dans l'exercice XII.3, les entiers dont il fournit le "rang". On conviendra qu'un rang égal à 0 signifie que l'utilisateur souhaite mettre fin au programme.

Solution

```
const int LGMAX_NOM_FICH = 20 ;
#include <stdlib.h>               // pour exit
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
main()
{
  char nomfich [LGMAX_NOM_FICH + 1] ;
  int n, num ;
  cout << "nom du fichier à consulter : " ;
  cin >> setw (LGMAX_NOM_FICH) >> nomfich ;
  ifstream entree (nomfich, ios::in) ;
  if (!entree) { cout << "Ouverture impossible\n" ;
                 exit (1) ;
               }
  do
    { cout << "Numéro de l'entier recherché : " ;
      cin >> num ;
      if (num)
         { entree.seekg (sizeof(int) * (num-1) , ios::beg ) ;
           entree.read ( (char *) &n, sizeof(int) ) ;
           if (entree) cout << "-- Valeur : " << n << "\n" ;
              else {  cout << "-- Erreur\n" ;
                      entree.clear () ;
                   }
         }
    }
  while (num) ;
  entree.close () ;
}
```

CHAPITRE XIII :
LES PATRONS DE FONCTIONS
(Depuis la Version 3 seulement)

RAPPELS

Introduite par la version 3, la notion de patron de fonctions permet de définir ce qu'on nomme souvent des "fonctions génériques". Plus précisément, à l'aide d'une unique définition comportant des "paramètres de type", on décrit toute une famille de fonctions ; le compilateur "fabrique" (on dit aussi instancie) la ou les fonctions nécessaires à la demande (on nomme souvent ces instances des fonctions patron).

La version 3 limite les paramètres d'un patron de fonctions à des paramètres de type ; beaucoup d'implémentations toutefois acceptent également des "paramètres expression" et cette possibilité sera probablement reconnue de la future norme ANSI de C++. Toutefois, pour plus de clarté, nous la présenterons séparément.

Définition d'un patron de fonctions

On précise les paramètres (muets) de type, en faisant précéder chacun du mot (relativement arbitraire) *class* sous la forme *template* < *class* ..., *class* ..., ...> . La définition de la fonction est classique, hormis le fait que les paramètres muets de type peuvent être employés n'importe où un type effectif est permis. Par exemple :

```
template <class T, class U> void fct (T a, T * b, U c)
{
   T x ;              // variable locale x de type T
   U *  adr ;         // variable locale adr de type U *
   ...
   adr = new T [10] ;  // allocation tableau de 10 éléments de type T
   ...
   n = sizeof (T) ;    // une instruction utilisant le type T
   ...
}
```

Remarque :

Une instruction telle que (T désignant un type quelconque) :

```
T x (3) ;
```

est légale même si T n'est pas un type classe ; dans ce dernier cas, elle est simplement équivalente à :

```
T x = 3 ;
```

Instanciation d'une fonction patron

A chaque fois qu'on utilise une fonction ayant un nom de patron, le compilateur cherche à utiliser ledit patron pour créer (instancier) une fonction adéquate. Pour ce faire, il cherche à réaliser une **correspondance exacte** des types (aucune conversion, qu'il s'agisse de promotion numérique ou de conversion standard n'est permise)[1].

Voici des exemples utilisant notre patron précédent :

```
int n, p ; float x ; char c ;
int * adi ; float * adf ;
class point ; point p ; point * adp ;

fct (n, adi, x) ;       // instancie la fonction void fct (int, int *, float)
fct (n, adi, p)         // instancie la fonction void fct (int, int *, int)
fct (x, adf, p) ;       // instancie la fonction void fct (float, float *, int)
fct (c, adi, x) ;       // erreur char et int * ne correspondent pas à T et T*
                        //         ( pas de conversion)
fct (&n, &adi, x) ;     // instancie la fonction void fct (int *, int * *, float)
fct (p, adp, n) ;       // instancie la fonction void fct (point, point *, int)
```

D'une manière générale, il est nécessaire que **chaque paramètre de type** apparaisse **au moins une fois dans l'en-tête** du patron.

Remarque :

La définition d'un patron de fonctions ne peut pas être compilée seule ; de toute façon, elle doit être connue du compilateur pour qu'il puisse instancier la bonne fonction patron. En général, les définitions de patrons de fonctions figureront dans des fichiers d'extension h, de façon à éviter d'avoir à en fournir systématiquement la liste.

Les paramètres expression d'un patron de fonctions

Bien que cette notion n'apparaisse pas dans la version 3, elle sera très probablement introduite dans la norme ANSI de C++. Un paramètre expression d'un patron de fonctions se présente comme un argument usuel de fonction ; il n'apparaît pas dans la liste de paramètres de type (*template*) et il doit apparaître dans l'en-tête du patron. Par exemple :

```
template <class T> int compte (T * tab, int n)
{  // ici, on peut se servir de la valeur de l'entier n
   // comme on le ferait dans n'importe quelle fonction ordinaire
}
```

[1] - A priori, même, la version 3 n'autorise même pas les conversions dites triviales (telles que T en T& ou, mieux, T[] en T*) ; toutefois, il est probable que celles-ci seront acceptées par la norme ANSI de C++ (au même titre qu'elles sont acceptées dans les correspondances exactes pour les fonctions surdéfinies).

Lors de l'instanciation d'une fonction patron, il est nécessaire de réaliser une correspondance exacte sur les paramètres expression (aucune conversion n'est possible, contrairement à ce qui se produit dans le cas de surdéfinition de fonction).

Surdéfinition de patrons de fonctions et spécialisation de fonctions de patrons

On peut définir plusieurs patrons de même nom, possédant des paramètres (de type ou expression) différents. La seule règle à respecter dans ce cas est que l'appel d'une fonction de ce nom ne doit pas conduire à une ambiguïté : un seul patron de fonctions doit pouvoir être utilisé à chaque fois.

Par ailleurs, il est possible de fournir la définition d'une ou plusieurs fonctions particulières qui seront utilisées en lieu et place de celle instanciée par un patron. Par exemple, avec :

```
template <class T> T min (T a, T b)      // patron de fonctions
{ ... }
char * min (char * cha, char * chb)      // version spécialisée pour le type char *
{ ... }
int n, p;
char * adr1, * adr2 ;

min (n, p)            // appelle la fonction instanciée par le patron général
                      //  soit ici  :    int min (int, int)
min (adr1, adr2)      // appelle la fonction spécialisée
                      //                 char * min (char *, char *)
```

Algorithme d'instanciation ou d'appel d'une fonction

Précisons comment doivent être aménagées les règles de recherche d'une fonction surdéfinie, dans le cas où il existe un ou plusieurs patrons de fonctions.

Lors d'un appel de fonction, le compilateur recherche tout d'abord une correspondance exacte avec les fonctions "ordinaires". S'il y a ambiguïté, la recherche échoue (comme à l'accoutumée). Si aucune fonction "ordinaire" ne convient, on examine alors tous les patrons ayant le nom voulu. Si une seule correspondance exacte est trouvée, la fonction correspondante est instanciée[2] et le problème est résolu. S'il y en a plusieurs, la recherche échoue..

Enfin, si aucun patron de fonction ne convient, on examine à nouveau toutes les fonctions "ordinaires" en les traitant cette fois comme de simples fonctions surdéfinies (promotions numériques, conversions standards[3]...).

Exercice XIII.1

Enoncé

Créer un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque (le résultat possèdera le même type). Ecrire un petit programme utilisant ce patron.

---

2 - Du moins, si elle ne l'a pas déjà été.

3 - Revoyez éventuellement le paragraphe 5.3 du chapitre 4.

Solution

Ici, notre patron ne comportera qu'un seul paramètre de type (correspondant à la fois à l'unique argument et à la valeur de retour de la fonction). Sa définition ne pose pas de problème particulier.

```
#include <iostream.h>
template <class T> T carre (T a)
  { return a * a ;
  }
main()
{ int n = 5 ;
  float x = 1.5 ;
  cout << "carre de " << n << " = " << carre (n) << "\n" ;
  cout << "carre de " << x << " = " << carre (x) << "\n" ;
}
```

Exercice XIII.2

Enoncé

Soit cette définition de patron de fonctions :

```
template <class T, class U> T fct (T a, U b, T c)
{ .....
}
```

Avec les déclarations suivantes :

```
int n, p, q ;
float x ;
char t[20] ;
char c ;
```

Quels sont les appels corrects et, dans ce cas, quels sont les prototypes des fonctions instanciées ?

```
  fct (n, p, q) ;          // appel I
  fct (n, x, q) ;          // appel II
  fct (x, n, q) ;          // appel III
  fct (t, n, &c) ;         // appel IV
```

Solution

l'appel I est correct ; il instancie la fonction :

```
int fct (int, int, int)
```

l'appel II est correct ; il instancie la fonction :

```
int fct (int, float, int)
```

l'appel III est incorrect.

l'appel IV est théoriquement incorrect dans la version 3 car *char** n'est pas considéré comme une correspondance exacte pour un *char [20]* ; il sera probablement accepté dans la norme ANSI et il instanciera alors la fonction :

```
char * fct (char *, int, char *)
```

Exercice XIII.3

Enoncé

Créer un patron de fonctions permettant de calculer la somme d'un tableau d'éléments de type quelconque, le nombre d'éléments du tableau étant fourni en paramètre (on supposera que l'environnement utilisé accepte les "paramètres expression"). Ecrire un petit programme utilisant ce patron.

Solution

```
// définition du patron de fonctions
template <class T> T somme (T * tab, int nelem)
{ T som ;
  int i ;
  som = 0 ;
  for (i=0 ; i<nelem ; i++)  som = som + tab[i] ;
  return som ;
}

// exemple d'utilisation
#include <iostream.h>
main()
{ int ti[] = {3, 5, 2, 1} ;
  float tf [] = {2.5, 3.2, 1.8} ;
  char tc[] = { 'a', 'e', 'i', 'o', 'u' } ;
  cout << somme (ti, 4) << "\n"  ;
  cout << somme (tf, 3) << "\n"  ;
  cout << somme (tc, 5) << "\n"  ;
}
```

Remarques :

1) Tel qu'il a été conçu, le patron *somme* ne peut être appliqué qu'à un type T pour lequel:

- l'opération d'addition a un sens ; cela signifie donc qu'il ne peut pas s'agir d'un type pointeur ; il peut s'agir d'un type classe, à condition que cette dernière ait surdéfini l'opérateur d'addition,

- la déclaration *T som* est correcte ; cela signifie que si T est un type classe, il est nécessaire qu'il dispose d'un constructeur sans argument,

- l'affectation *som =0* est correcte ; cela signifie que si T est un type classe, il est nécessaire qu'il ait surdéfini l'affectation.

A ce propos, notons, qu'il est possible d'initialiser *somme* lors de sa déclaration, en procédant ainsi :

```
T somme (0) ;
```

Cela est équivalent à *T somme =0* si T est un type prédéfini ; en revanche, si T est de type classe, cela provoque l'appel d'un constructeur à 1 argument de T, en lui transmettant la valeur 0 ; le problème relatif à l'affectation *somme =0* ne se pose plus alors.

2) L'exécution de l'exemple proposé fournit des résultats peu satisfaisants dans le cas où l'on applique *somme* à un tableau de caractères, compte tenu de la capacité limitée de ce type. On pourrait améliorer la situation en "spécialisant" notre patron pour les tableaux de caractères (en prévoyant, par exemple, une valeur de retour de type *int*).

Exercice XIII.4

Enoncé

Soient les définitions suivantes de patrons de fonctions :

```
template <class T, class U> void fct (T a, U b)   { ... } // patron I
template <class T, class U> void fct (T * a, U b) { ... } // patron II
template <class T>          void fct (T, T, T)    { ... } // patron III
void fct (int a, float b) { .....}                        // fonction IV
```

Avec ces déclarations :

```
int n, p, q ;
float x, y ;
double z ;
```

Quels sont les appels corrects et, dans ce cas, quels sont les patrons utilisés et les prototypes des fonctions instanciées ?

```
fct (n, p) ;      // appel I
fct (x, y ) ;     // appel II
fct (n, x) ;      // appel III
fct (n, z) ;      // appel IV
fct (&n, p) ;     // appel V
fct (&n, x) ;     // appel VI
fct (&n, &p, &q) // appel VII
```

Solution

Ici, on fait appel à la fois à une surdéfinition de patrons (patrons I, II et III) et à une spécialisation de patron (fonction IV).

```
I)      patron I       void fct (int, int) ;
II)     patron I       void fct (float, float) ;
III)    fonction IV    void fct (int, float) ;
IV)     patron I       void fct (int, double) ;
V)      erreur : ambigüité entre fct (T, U) et fct (T*, U)
VI)     erreur : ambigüité entre fct (T, U) et fct (T*, U)
VII)    patron III     void fct (int *, int *, int *) ;
```

Remarque :

Le patron II ne peut jamais être utilisé ; en effet, à chaque fois qu'il pourrait l'être, le patron I peut l'être également, de sorte qu'il y a ambiguïté. Le patron II est donc, ici, parfaitement inutile.

Notez que si nous avions défini simultanément les deux patrons :

```
template <class T, class U> void fct (T a, U b) ;
template <class T>          void fct (T a, T b)
```

le même phénomène d'ambiguïté (entre ces deux patrons) serait apparu lors d'appels tels que *fct(n,p)* ou *fct(x,y)*.

Rappelons que l'ambiguïté n'est détectée que lorsque le compilateur doit instancier une fonction et non simplement au vu des définitions de patrons elles-mêmes : ces dernières restent donc acceptées tant que l'ambiguïté n'est pas mise en évidence par un appel la révélant.

CHAPITRE XIV
LES PATRONS DE CLASSES
(Depuis la version 3 seulement)

RAPPELS

Introduite par la version 3, la notion de patron de classes permet de définir ce que l'on nomme aussi des "classes génériques". Plus précisément, à l'aide d'une seule définition comportant des paramètres de type et des paramètres expression[1], on décrit toute une famille de classes ; le compilateur fabrique (instancie) la ou les classes nécessaires à la demande (on nomme souvent ces instances des classes patron).

Définition d'un patron de classes

On précise les paramètres de type en les faisant précéder du mot-clé *class* et les paramètres expression en mentionnant leur type dans une liste de paramètres introduite par le mot *template* (comme pour les patrons de fonctions, avec cette différence qu'ici, tous les paramètres - type ou expression - apparaissent). Par exemple :

```
template <class T, class U, int n> class gene
{ // ici, T désigne un type quelconque, n une valeur entière quelconque
} ;
```

Si une fonction membre est définie (ce qui est le cas usuel) à l'extérieur de la définition du patron, il faut rappeler au compilateur la liste de paramètres (*template*) et préfixer l'en-tête de la fonction membre du nom du patron accompagné de ses paramètres[2]. Par exemple, pour un constructeur de notre patron de classes précédent :

```
template <class T, class U, int n> gene <T, U, n>::gene (...)
{ ..... }
```

---

1. Cette fois, ces paramètres sont déjà prévus par la version 3 (à brs que, dans le cas des patrons de fonctions, ils risquaient d'être introduits par la norme ANSI.

2. En toute rigueur, il s'agit d'une redondance, constatée, mais non justifiée, par le fondateur du langage lui-même (Stroustrup).

Instanciation d'une classe patron

On déclare une classe patron en fournissant à la suite du nom de patron un nombre de paramètres effectifs (noms de types ou expressions) correspondant aux paramètres figurant dans la liste (*template*). Les paramètres expression doivent obligatoirement être des expressions constantes du même type (exact[3]) que celui figurant dans la liste. Par exemple, avec notre précédent patron (on suppose que *pt* est une classe) :

```
class gene <int, float, 5> c1 ;      // T = int, U = float,  n = 5
class gene <int, int, 12>  c2 ;      // T = int, U = int,    n = 12
const int NV=100 ;
class gene <pt, double, NV> c3 ;     // T = pt,  U = double, n=100
int n = 5 ;
class gene <int, double, n> c4 ;     // erreur : n n'est pas constant
const char C = 'e' ;
class gene <int, double, C> c5 ;     // erreur : C de type char et non int
```

Un paramètre de type effectif peut lui-même être une classe patron. Par exemple, si nous avons défini un patron de classes *point* par :

```
template <class T> class point { ..... } ;
```

Voici des instances possibles de *gene* :

```
class gene <point<int>, float, 10> c5 ;        // T=point<int>, U=float, n=10
class gene <point<char>, point<float>, 5> c6 ; // T=point<int>, U=point<float>, n=5
```

Un patron de classes peut comporter des membres (donnée ou fonction) statiques ; dans ce cas, chaque instance de la classe dispose de son propre jeu de membres statiques.

Spécialisation d'un patron de classes

Un patron de classes ne peut pas être surdéfini (on ne peut pas définir deux patrons de même nom). En revanche, on peut spécialiser un patron de classes de différentes manières :

- en spécialisant une fonction membre :

Par exemple, avec ce patron :

```
template <class T, int n> class tableau { ..... } ;
```

Nous pourrons écrire une version spécialisée de constructeur pour le cas où T est le type *point* et où n vaut 10 en procédant ainsi :

```
tableau <point, 10>:: tableau (...) { ..... }
```

- en spécialisant une classe (dans ce dernier cas, on peut éventuellement spécialiser tout ou une partie des fonctions membre, mais ce n'est pas nécessaire). Par exemple, avec ce patron :

---

[3]. Dans la norme ANSI, les conversions triviales seront probablement acceptées.

```
template <class T> class point { ..... } ;
```

nous pouvons fournir une version spécialisée pour le cas où T est le type *char* en procédant ainsi :

```
class point <char>
{ // nouvelle définition de la classe point pour les caractères
} ;
```

Identité de classes patron

On ne peut affecter entre eux que deux objets de même type. Dans le cas d'objets d'un type classe patron, on considère qu'il y a identité de type lorsque leurs paramètres de types sont identiques et que les paramètres expression ont les mêmes valeurs.

Classes patron et héritage

On peut "combiner" de plusieurs façons l'héritage avec la notion de patron de classes :

- **Classe "ordinaire" dérivée d'une classe patron** ;par exemple, si A est une classe patron définie par *template < class T> A* :

```
class B : public A <int>    // B dérive de la classe patron A<int>
```

On obtient une seule classe nommée B

- **Patron de classes dérivé d'une classe "ordinaire"**, par exemple (A étant une classe ordinaire) :

```
template <class T> class B : public A
```

On obtient une famille de classes (de paramètre de type T).

- **Patron de classes dérivé d'un patron de classes**. Par exemple, si A est une classe patron définie par *template < class T> A*, on peut :

  *définir une nouvelle famille de fonctions dérivées par :

```
template <class T> class B : public A <T>
```

  Dans ce cas, il existe autant de classes dérivées possibles que de classes de base possibles.

  *définir une nouvelle famille de fonctions dérivées par :

```
template <class T, class U> class B : public A <T>
```

  Dans ce cas, on peut dire que chaque classe de base possible peut engendrer une famille de classes dérivées (de paramètre de type U).

Exercice XIV.1

Enoncé

Soit la définition suivante d'un patron de classes :

```
template <class T, int n> class essai
{  T tab [n] ;
```

```
  public :
   essai (T) ;    // constructeur
} ;
```

a) Donnez la définition du constructeur *essai*, en supposant :

- qu'elle est fournie "à l'extérieur" de la définition précédente,
- que le constructeur recopie la valeur reçue en argument dans chacun des éléments du tableau *tab*.

b) Disposant ainsi de la définition précédente du patron *essai*, de son constructeur et de ces déclarations :

```
const int n = 3 ;
int p = 5 ;
```

Quelles sont les instructions correctes et les classes instanciées : on en fournira (dans chaque cas) une définition équivalente sous forme d'une "classe ordinaire", c'est-à-dire dans laquelle la notion de paramètre a disparu.

```
essai <int, 10> ei (3) ;              // I
essai <float, n> ef (0.0) ;           // II
essai <double, p> ed (2.5) ;          // III
```

Solution

a) La définition du constructeur est analogue à celle que l'on aurait écrite "en ligne" ; il faut simplement "préfixer" son en-tête d'une liste de paramètres introduite par *template*. De plus, il faut préfixer l'en-tête de la fonction membre du nom du patron accompagné de ses paramètres (bien que cela soit redondant) :

```
template <class T, int n> essai<T,n>::essai(T a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

b) Appel I : correct

```
class essai
{  int tab [10] ;
  public :
   essai (int) ;    // constructeur
} ;

essai::essai (int a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

b) Appel II : correct

```
class essai
{  float tab [n] ;
  public :
   essai (float) ;    // constructeur
```

```
} ;

essai::essai (float a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

c) Appel III : incorrect car p n'est pas une expression constante.

Exercice XIV.2

Enoncé

a) Créer un patron de classes nommé *pointcol*, tel que chaque classe instanciée permette de manipuler des points colorés (deux coordonnées et une couleur) pour lesquels on puisse "choisir" à la fois le type des coordonnées et celui de la couleur. On se limitera à deux fonctions membre : un constructeur possédant trois arguments (sans valeur par défaut) et une fonction *affiche* affichant les coordonnées et la couleur d'un "point coloré".

b) Dans quelles conditions peut-on instancier une classe patron *pointcol* pour des paramètres de type classe.

Solution

a) Voici ce que pourrait être la définition du patron demandé, en prévoyant les fonctions membre "en ligne" :

```
template <class T, class U> class pointcol
{  T x, y ;    // coordonnees
   U coul ;    // couleur
  public :
   pointcol (T abs, T ord, U cl)
     { x = abs ; y = ord ; coul = cl ;
     }
   void affiche ()
     { cout << "point colore - coordonnees " << x << " " << y
            << " couleur " << coul << "\n" ;
     }
}
```

A titre indicatif, voici un exemple d'utilisation (on y suppose que la définition précédente figure dans *pointcol.h* :

```
#include <iostream.h>
#include "pointcol.h"

main()
{ pointcol <int, short int > p1 (5, 5, 2) ;  p1.affiche () ;
  pointcol <float, int> p2 (4, 6, 2) ; p2.affiche () ;
  pointcol <double, unsigned short> p3 ; p3.affiche () ;
```

```
}
```

b) Il suffit que le type classe en question ait convenablement surdéfini l'opérateur <<, afin d'assurer convenablement l'affichage sur *cout* des informations correspondantes.

Exercice XIV.3

Enoncé

On a défini le patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
     { cout << "Coordonnees : " << x << " " << y << "\n" ;
     }
```

a) Que se passe-t-il avec ces instructions :

```
point <char> p (60, 65) ;
p.affiche () ;
```

b) Comment faut-il modifier la définition de notre patron pour que les instructions précédentes affichent bien :

```
Coordonnees : 60 65
```

Solution

a) On obtient l'affichage des caractères de code 60 et 65 (c'est-à-dire dans une implémentation utilisant le code ASCII : < et A) et non les nombres 60 et 65.

b) Il faut spécialiser notre patron *point* pour le cas où le type T est le type *char*. Pour ce faire, on peut :

- soit fournir une définition complète de *point<char>*, avec ses fonctions membre,
- soit, puisqu'ici seule la fonction *affiche* est concernée, se contenter de surdéfinir la fonction *point<char>::affiche*, ce qui conduit à cette nouvelle définition de notre patron :

```
    // définition generale du patron point
template <class T> class point
{  T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
```

```
      { cout << "Coordonnees : " << x << " " << y << "\n" ;
      }
     // version specialisee de la fonction affiche pour le type char
void point<char>::affiche ()
  { cout << "Coordonnees : " << (int)x << " " << (int)y << "\n" ;
  }
```

Remarque :

Bien que cela ne soit pas totalement formalisé, beaucoup d'implémentations n'acceptent pas que l'on spécialise une fonction membre définie en ligne ; c'est d'ailleurs pour cette raison que, dans notre énoncé, nous avions défini *affiche* sous forme d'une fonction indépendante.

Exercice XIV.4

Enoncé

Créer un patron de classes permettant de représenter des "vecteurs dynamiques" c'est-à-dire des vecteurs dont la dimension peut ne pas être connue lors de la compilation (ce n'est donc pas obligatoirement une expression constante comme dans le cas de tableaux usuels). On prévoira que les éléments de ces vecteurs puissent être de type quelconque.

On surdéfinira convenablement l'opérateur [] pour qu'il permette l'accès aux éléments du vecteur (aussi bien en consultation qu'en modification) et on s'arrangera pour qu'il n'existe aucun risque de "débordement d'indice". En revanche, on ne cherchera pas à régler les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets du type concerné.

Solution

En généralisant ce qui a été fait dans l'exercice VII.7 (mais sans toutefois initialiser les éléments du vecteur lors de sa construction), nous aboutissons au patron de classes suivant[4] :

```
template <class T> class vect
{  int nelem ;    // nombre d'elements
   T * adr ;      // adresse zone dynamique contenant les elements
  public :
   vect (int) ;              // constructeur
   ~vect () ;                // destructeur
   T & operator [] (int) ;   // operateur d'acces a un element
} ;

template <class T> vect<T>::vect (int n)
{ adr = new T [nelem = n] ;
}


template <class T> vect<T>::~vect ()
```

4. La définition serait plus simple si les fonctions membre étaient "en ligne".

```
{ delete adr ;
}

template <class T> T & vect<T>::operator [] (int i)
{ if ( (i<0) || (i>nelem) ) i = 0 ;     // protection indice hors limites
  return adr [i] ;
}
```

Notez que, ici encore, nous avons fait en sorte qu'une tentative d'accès à un élément situé en dehors du vecteur conduise à accéder à l'élément de rang 0. Dans la pratique, on aura intérêt à utiliser des protections plus élaborées.

A titre indicatif, voici un petit programme utilisant ce patron (dont on suppose que la définition figure dans *vectgen.h* :

```
#include <iostream.h>
#include "vectgen.h"
main()
{ vect<int> vi (10) ;
  vi[5] = 5 ; vi[2] = 2 ;
  cout << vi[2] << " " << vi[5] << "\n" ;
  vect<double> vd (3) ;
  vd[0] = 0.0 ; vd[1] = 0.1 ; vd[2] = 0.2 ;
  cout << vd[0] << " " << vd[1] << " " << vd[2] << "\n" ;
  cout << vd[12] ; vd[12] = 1.2 ; cout << vd[12] << " " << vd[0] ;
}
```

Remarque :

Notre patron *vect* permet d'instancier des vecteurs dynamiques dans lesquels les éléments sont de type absolument quelconque, en particulier de type classe (pourvu que ladite classe dispose d'un constructeur sans argument). Il n'en serait pas allé ainsi si nous avions initialisé les éléments du tableau lors de leur construction par :

```
int i ;
for (i=0 ; i<nelem ; i++) adr[i] = 0 ;
```

En effet, dans ce cas, ces instructions auraient convenablement fonctionné pour n'importe quel type de base (par conversion de l'entier 0 dans le type voulu). En revanche, pour être applicable à des éléments de type classe, il aurait fallu, en outre, que la classe concernée dispose d'une conversion d'un *int* dans ce type classe, c'est-à-dire d'un constructeur à un argument de type numérique.

Exercice XIV.5

Enoncé

Comme dans l'exercice précédent, réaliser un patron de classes permettant de manipuler des vecteurs dont les éléments sont de type quelconque mais pour lesquels la dimension, supposée être cette fois une expression constante, apparaîtra comme un paramètre (expression) du patron. Hormis cette différence, les "fonctionnalités" du patron resteront les mêmes.

Solution

Il n'est donc plus nécessaire d'allouer un emplacement dynamique pour notre vecteur qui peut donc figurer directement dans les membres donnée de notre patron. Le constructeur n'est plus nécessaire (voir toutefois la remarque ci-dessous), pas plus que le destructeur. Voici ce que pourrait être la définition de notre patron[5] :

```
template <class T, int n> class vect
{  T v [n] ;      // vecteur de n elements de type T
  public :
   T & operator [] (int) ;   // operateur d'acces a un element
} ;

template <class T, int n> T & vect<T,n>::operator [] (int i)
{ if ( (i<0) || (i>n) ) i = 0 ;     // protection indice hors limites
  return v [i] ;
}
```

Voici, toujours à titre indicatif, ce que deviendrait le petit programme d'essai :

```
#include "vectgen1.h"
#include <iostream.h>
main()
{ vect<int, 10> vi  ;
  vi[5] = 5 ; vi[2] = 2 ;
  cout << vi[2] << " " << vi[5] << "\n" ;
  vect<double, 3> vd ;
  vd[0] = 0.0 ; vd[1] = 0.1 ; vd[2] = 0.2 ;
  cout << vd[0] << " " << vd[1] << " " << vd[2] << "\n" ;
  cout << vd[12] << " " ; vd[12] = 1.2 ; cout << vd[12] << " " << vd[0] ;
}
```

Remarque :

Ici, nous n'avons pas eu besoin de faire du nombre d'éléments un membre donnée de nos classes patron : en effet, lorsqu'on en a besoin, on l'obtient comme étant la valeur du second paramètre fourni lors de l'instanciation. Si nous avions voulu conserver ce nombre d'éléments sous forme d'un membre donnée, il aurait été nécessaire de prévoir un constructeur, par exemple :

```
template <class T, int n> class vect
{  int nelem ;    // nombre d'elements
   T v [n] ;      // vecteur de n elements de type T
  public :
   vect () ;
   T & operator [] (int) ;   // operateur d'acces a un element
} ;
template <class T, int n> vect<T,n>::vect ()
{ nelem = n ;
}
```

Exercice XIV.6

[5]. La définition serait plus simple si les fonctions membre étaient "en ligne".

Enoncé

On dispose du patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche ()
     { cout << "Coordonnees : " << x << " " << y << "\n" ;
     }
} ;
```

a) Créer, par dérivation, un patron de classes *pointcol* permettant de manipuler des "points colorés" dans lesquels les coordonnées et la couleur sont de même type. On redéfinira convenablement les fonctions membre en réutilisant les fonctions membre de la classe de base.

b) Même question, mais en prévoyant que les coordonnées et la couleur puissent être de deux types différents.

c) Toujours par dérivation, créer cette fois une "classe ordinaire" (c'est-à-dire une classe qui ne soit plus un patron de classes, autrement dit qui ne dépende plus de paramètres...) dans laquelle les coordonnées sont de type *int*, tandis que la couleur est de type *short*.

Solution

a) Aucun problème particulier ne se pose ; il suffit de faire dériver *pointcol*< *T*> de *point*< *T*> . Voici ce que peut être la définition de notre patron (ici, nous avons laissé le constructeur "en ligne" mais nous avons défini *affiche* en dehors de la classe) :

```
template <class T> class pointcol : public point<T>
{  T cl ;
  public :
   pointcol (T abs, T ord, T coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche () ;
} ;

template <class T> void pointcol<T>::affiche ()
{  point<T>::affiche () ;
   cout << "    couleur : " << cl << "\n" ;
}
```

Voici un petit exemple d'utilisation (il nécessite les déclarations appropriées ou l'incorporation de fichiers .h correspondants) :

```
main()
{ pointcol <int> p1 (2, 5, 1) ; p1.affiche () ;
  pointcol <float> p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

b) Cette fois, la classe dérivée dépend de deux paramètres (nommés ici T et U). Voici ce que pourrait être la définition de notre patron (avec, toujours, un constructeur en ligne et une fonction *affiche* définie à l'extérieur de la classe) :

```
template <class T, class U> class pointcol : public point<T>
{  U cl ;
  public :
   pointcol (T abs, T ord, U coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche () ;
} ;
template <class T, class U> void pointcol<T, U>::affiche ()
{  point<T>::affiche () ;
   cout << "    couleur : " << cl << "\n" ;
}
```

Voici un exemple d'utilisation (on suppose qu'il est muni des déclarations appropriées) :

```
main()
{ pointcol <int, short> p1 (2, 5, 1) ; p1.affiche () ;
  pointcol <float, int> p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

c) Cette fois, *pointcol* est une simple classe, ne dépendant plus d'aucun paramètre. Voici ce que pourrait être sa définition :

```
class pointcol : public point<int>
{  short cl ;
  public :
   pointcol (int abs, int ord, short coul) : point<int> (abs, ord)
     { cl = coul ;
     }
   void affiche ()
     { point<int>::affiche () ;
       cout << "    couleur : " << cl << "\n" ;
     }
} ;
```

Et un petit exemple d'utilisation :

```
main()
{ pointcol  p1 (2, 5, 1) ; p1.affiche () ;
  pointcol  p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

Exercice XIV.7

Enoncé

On dispose du même patron de classes que précédemment :

```
template <class T> class point
{  T x, y ;    // coordonnees
```

```
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche ()
     { cout << "Coordonnees : " << x << " " << y << "\n" ;
     }
} ;
```

a) Lui ajouter une version spécialisée de *affiche* pour le cas où T est le type caractère.

b) Comme dans la question a de l'exercice précédent, créer un patron de classes *pointcol* permettant de manipuler des "points colorés" dans lesquels les coordonnées et la couleur sont de même type. On redéfinira convenablement les fonctions membre en réutilisant les fonctions membre de la classe de base et l'on prévoira une version spécialisée de *affiche* de *pointcol* dans le cas du type caractère.

Solution

a) Pour définir une version spécialisée d'une fonction membre, il est généralement nécessaire que cette fonction membre soit définie extérieurement au patron de classes (revoyez la remarque de l'exercice XIV.3). Voici ce que pourrait être la nouvelle définition (complète) de notre patron *point* :

```
template <class T> class point
{  T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche () ;
} ;

template <class T> void point<T>::affiche ()
{ cout << "Coordonnees : " << x << " " << y << "\n" ;
}
void point<char>::affiche ()
{ cout << "Coordonnees : " << (int)x << " " << (int)y << "\n" ;
}
```

b) Pour les mêmes raisons que précédemment, la fonction *affiche* de *pointcol* doit être définie à l'extérieur du patron. Voici ce que pourrait être cette définition :

```
template <class T> class pointcol : public point<T>
{  T cl ;
  public :
   pointcol (T abs, T ord, T coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche () ;
} ;

template <class T> void pointcol<T>::affiche ()
{  point<T>::affiche () ;
   cout << "    couleur : " << cl << "\n" ;
}

void pointcol<char>::affiche ()
{  point<char>::affiche () ;
   cout << "    couleur : " << (int)cl << "\n" ;
}
```

Remarque :

Seule la question a de l'exercice XIV.6 se prêtait à une spécialisation pour le type caractère. En effet, pour la classe demandée en c, n'ayant plus affaire à un patron de classes, la question n'aurait aucun sens. En ce qui concerne la classe demandée en b, en revanche, on se trouve en présence d'une classe dérivée dépendant de 2 paramètres T et U. Il faudrait alors pouvoir spécialiser une classe, non plus pour des valeurs données de tous les (deux) paramètres, mais pour une valeur donnée (*char*) de l'un d'entre eux ; cette notion de famille de spécialisation n'est pas possible, du moins dans l'état actuel de la définition de C++.

Exercice XIV.8

Enoncé

On dispose du patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnees
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche ()
     { cout << "Coordonnees : " << x << " " << y << "\n" ;
     }
} ;
```

On souhaite créer un patron de classes *cercle* permettant de manipuler des cercles, définis par leur centre (de type *point*) et un rayon. On n'y prévoira, comme fonctions membre, qu'un constructeur et une fonction *affiche* se contentant d'afficher les coordonnées du centre et la valeur du rayon.

a) Le faire par héritage (un cercle est un point qui possède un rayon),

b) Le faire par composition d'objets membre (un cercle possède un point et un rayon).

Solution

a) La démarche est analogue à celle de la question a de l'exercice XIV.6. On a affaire à un patron dépendant de deux paramètres (ici T et U).

```
template <class T, class U> class cercle : public point<T>
{  U r ;       // rayon
  public :
   cercle (T abs, T ord, U ray) : point<T> (abs, ord)
     { r = ray ;
     }
   void affiche ()
     { point<T>::affiche () ;
       cout << "    rayon : " << r ;
     }
```

```
} ;
```

b) Le patron dépend toujours de deux paramètres (T et U) mais il n'y a plus de notion d'héritage :

```
template <class T, class U> class cercle
{  point<T> c ;  // centre
   U r ;         // rayon
  public :
   cercle (T abs, T ord, U ray) : c(abs, ord)     // pourrait r(ray)
     { r = ray ;
     }
   void affiche ()
     { c.affiche () ;
       cout << "    rayon : " << r ;
     }
} ;
```

Notez que, dans la définition du constructeur *cercle*, nous avons transmis les arguments *abs* et *ord* à un constructeur de *point* pour le membre c. Nous aurions pu utiliser la même notation pour r, bien que ce membre soit d'un type de base et non d'un type classe ; cela nous aurait conduit au constructeur suivant (de corps vide) :

```
cercle (T abs, T ord, U ray) : c(abs, ord), r(ray)
  { }
```

CHAPITRE XV :
EXERCICES DE SYNTHESE

Exercice XV.1

Enoncé

Réaliser une classe nommée *set_int* permettant de manipuler des ensembles de nombres entiers. Le nombre maximal d'entiers que pourra contenir l'ensemble sera précisé au constructeur qui allouera dynamiquement l'espace nécessaire. On prévoira les opérateurs suivants (*e* désigne un élément de type *set_int* et *n* un entier :

- **<<**, telque *e* << *n* ajoute l'élément *n* à l'ensemble *e*,
- **%**, telque *n*%*e* vaut 1 si *n* appartient à *e* et 0 sinon,
- **<<**, telque *flot* << *e* envoie le contenu de l'ensemble *e* sur le flot indiqué, sous la forme :

```
[entier1, entier2, ... entiern]
```

La fonction membre *cardinal* fournira le nombre d'éléments de l'ensemble. Enfin, on s'arrangera pour que l'affectation ou la transmission par valeur d'objets de type *set_int* ne pose aucun problème (on acceptera la duplication complète d'objets).

Solution

Naturellement, notre classe comportera, en membres donnée, le nombre maximal (*nmax*) d'éléments de l'ensemble, le nombre courant d'éléments (*nelem*) et un pointeur sur l'emplacement contenant les valeurs de l'ensemble.

Compte tenu de ce que notre classe comporte une partie dynamique, il est nécessaire, pour que l'affectation et la transmission par valeur se déroulent convenablement, de surdéfinir l'opérateur d'affectation et de munir notre classe d'un constructeur par recopie. Les deux fonctions membre (*operator =* et *set_int*) devront prévoir une "copie profonde" des objets. Nous utiliserons pour cela une méthode que nous avons déjà rencontrée et qui consiste à considérer que deux objets différents disposent systématiquement de deux parties dynamiques différentes, même si elles possèdent le même contenu.

L'opérateur % doit être surdéfini obligatoirement sous forme d'une fonction amie, puisque son premier opérande n'est pas de type classe. L'opérateur de sortie dans un flot doit, lui aussi, être surdéfini sous forme d'une fonction amie, mais pour une raison différente : son premier argument est de type *ostream*.

Voici la déclaration de notre classe *set_int* :

```
          /*         fichier SETINT.H         */
          /* déclaration de la classe set_int */
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 (set_int &) ;                    // constructeur par recopie
   set_int & operator = (set_int &) ;       // opérateur d'affectation
   ~set_int () ;                            // destructeur
   int cardinal () ;                        // cardinal de l'ensemble
   set_int & operator << (int) ;            // ajout d'un élément
   friend int operator % (int, set_int &) ;  // appartenance d'un élément
                                    // envoi ensemble dans un flot
   friend ostream & operator << (ostream &, set_int &) ;
} ;
```

Voici ce que pourrait être la définition de notre classe (les points délicats sont commentés au sein même des instructions) :

```
#include <iostream.h>
#include "setint.h"

     /*************** constructeur ******************/
set_int::set_int (int dim)
{  adval = new int [nmax = dim] ;    // allocation tableau de valeurs
   nelem = 0 ;
}

     /****************** destructeur *****************/
set_int::~set_int ()
{  delete adval ;                   // libération tableau de valeurs
}

     /********** constructeur par recopie ************/
set_int::set_int (set_int & e)
{  adval = new int [nmax = e.nmax] ;   // allocation nouveau tableau
   nelem = e.nelem ;
   int i ;
   for (i=0 ; i<nelem ; i++)           // copie ancien tableau dans nouveau
    adval[i] = e.adval[i] ;
}

     /************ opérateur d'affectation ************/
set_int & set_int::operator = (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 ;
}


     /************ fonction membre cardinal **********/
int set_int::cardinal ()
{  return nelem ;
}

     /************ opérateur d'ajout << ***************/
set_int & set_int::operator << (int nb)
{        // on examine si nb appartient déjà à l'ensemble
         //   en utilisant l'opérateur %
         // s'il n'y appartient pas, et s'il y a encore de la place
         //   on l'ajoute à l'ensemble
   if ( ! (nb % *this) && nelem < nmax )  adval [nelem++] = nb ;
   return (*this) ;
}

     /*********** opérateur d'appartenance % **********/
int operator % (int nb, set_int & e)
{  int i=0 ;
         // on examine si nb appartient déjà à l'ensemble
         //  (dans ce cas i vaudra nele en fin de boucle)
   while ( (i<e.nelem) && (e.adval[i] != nb) ) i++ ;
   return (i<e.nelem) ;
}

     /****** opérateur << pour sortie sur un flot *****/
ostream & operator << (ostream & sortie, set_int & e)
{  sortie << "[ " ;
   int i ;
   for (i=0 ; i<e.nelem ; i++)
      sortie << e.adval[i] << " " ;
   sortie << "]" ;
   return sortie ;
}
```

Notez qu'ici il n'est pas possible d'agrandir l'ensemble au-delà de la limite qui lui a été impartie lors de sa construction. Il serait assez facile de remédier à cette lacune en modifiant sensiblement la fonction d'ajout d'un élément (*operator* < <). Il suffirait, en effet, qu'elle prévoie, lorsque la limite est atteinte, d'allouer un nouvel emplacement dynamique, par exemple d'une taille double de l'emplacement existant, d'y recopier l'actuel contenu et de libérer l'ancien emplacement (en actualisant convenablement les membres donnée de l'objet).

Voici un exemple de programme utilisant la classe *set_int*, accompagné du résultat fourni par son exécution :

```
     /************* test de la classe set_int *********/
#include <iostream.h>
#include "setint.h"

main()
```

```
{  void fct (set_int) ;
   void fctref (set_int &) ;
   set_int ens ;
   cout << "donnez 10 entiers \n" ;
   int i, n ;
   for (i=0 ; i<10 ; i++)
      { cin >> n ;
        ens << n ;
      }
   cout << "il y a : " << ens.cardinal () << " entiers différents\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   fct (ens) ;
   cout << "au retour de fct, il y en a " << ens.cardinal () << "\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   fctref (ens) ;
   cout << "au retour de fctref, il y en a " << ens.cardinal () << "\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   cout << "appartenance de -1  : " <<  -1 % ens << "\n" ;
   cout << "appartenance de 500 : " << 500 % ens << "\n" ;
   set_int ensa, ensb ;
   ensa = ensb = ens ;
   cout << "ensemble a : " << ensa << "\n" ;
   cout << "ensemble b : " << ensb << "\n" ;
}

void fct (set_int e)
{  cout << "ensemble reçu par fct : " << e << "\n" ;
   e << -1 << -2 << -3 ;
}

void fctref (set_int & e)
{  cout << "ensemble reçu par fctref : " << e << "\n" ;
   e << -1 << -2 << -3 ;
}
```

```
donnez 10 entiers
3 5 3 1 8 5 1 7 7 3
il y a : 5 entiers différents
qui forment l'ensemble : [ 3 5 1 8 7 ]
ensemble reçu par fct : [ 3 5 1 8 7 ]
au retour de fct, il y en a 5
qui forment l'ensemble : [ 3 5 1 8 7 ]
ensemble reçu par fctref : [ 3 5 1 8 7 ]
au retour de fctref, il y en a 8
qui forment l'ensemble : [ 3 5 1 8 7 -1 -2 -3 ]
appartenance de -1  : 1
appartenance de 500 : 0
ensemble a : [ 3 5 1 8 7 -1 -2 -3 ]
ensemble b : [ 3 5 1 8 7 -1 -2 -3 ]
```

Exercice XV.2

Enoncé

Créer une classe *vect* permettant de manipuler des "vecteurs dynamiques" d'entiers, c'est-à-dire des tableaux d'entiers dont la dimension peut être définie au moment de leur création (une telle classe a déjà été partiellement réalisée dans l'exercice VII.7). Cette classe devra disposer des opérateurs suivants :

[] pour l'accès à une des composantes du vecteur, et ce là aussi bien au sein d'une expression qu'à gauche d'une affectation (mais cette dernière situation ne devra pas être autorisée sur des "vecteurs constants"),

==, tel que si *v1* et *v2* sont deux objets de type *vect*, *v1==v2* prenne la valeur 1 si *v1* et *v2* sont de même dimension et ont les mêmes composantes et la valeur 0 dans le cas contraire,

<<, tel que *flot<<v* envoie le vecteur *v* sur le flot indiqué, sous la forme :

```
<entier1, entier2, ... , entiern>
```

De plus, on s'arrangera pour que l'affectation et la transmission par valeur d'objets de type *vect* ne pose aucun problème ;pour ce faire, on acceptera de dupliquer complètement les objets concernés.

Solution

Rappelons que lorsque l'on définit des objets constants, 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, une telle fonction peut indifféremment être utilisée avec des objets constants ou non constants). Pour obtenir l'effet demandé de l'opérateur [], lorsqu'il est appliqué à un vecteur constant, il est nécessaire d'en prévoir deux définitions dont l'une s'applique aux vecteurs constants ;pour éviter qu'on ne puisse, dans ce cas, l'utiliser à gauche d'une affectation, il est nécessaire qu'elle renvoie son résultat par valeur (et non par adresse comme le fera la fonction applicable aux vecteurs non constants).

Voici la déclaration de notre classe :

```
class vect
{  int nelem ;                  // nombre de composantes du vecteur
   int * adr ;                  // pointeur sur partie dynamique
 public :
   vect (int n=1) ;             // constructeur "usuel"
   vect (vect & v) ;            // constructeur par recopie
   ~vect () ;                   // destructeur
   friend ostream & operator << (ostream &, vect &) ;
   vect operator = (vect & v) ;  // surdéfinition opérateur affectation
   int & operator [] (int i) ;   // surdef [] pour vect non constants
   int   operator [] (int i) const ;   // surdef [] pour vect constants
} ;
```

Voici la définition des différentes fonctions :

```
#include <stdio.h>
#include "a:\synthese\vect.h"
vect::vect (int n)                  // constructeur "usuel"
     { adr = new int [nelem = n] ;
     }

vect::vect (vect & v)               // constructeur par recopie
     { adr = new int [nelem = v.nelem] ;
       int i ;
       for (i=0 ; i<nelem ; i++)
           adr[i] = v.adr[i] ;
```

```
     }

vect::~vect ()                    // destructeur
     { delete adr ;
     }

vect vect::operator = (vect & v)    // surdéfinition opérateur affectation
     { if (this != &v)             // on ne fait rien pour a=a
         { delete adr ;
           adr = new int [nelem = v.nelem] ;
           int i ;
           for (i=0 ; i<nelem ; i++)
             adr[i] = v.adr[i] ;
         }
       return * this ;
     }

int & vect::operator [] (int i)   // surdéfinition opérateur []
     { return adr[i] ;
     }

int   vect::operator [] (int i) const // surdéfinition opérateur [] pour cst
     { return adr[i] ;
     }

ostream & operator << (ostream & sortie, vect & v)
     { sortie << "<" ;
       int i ;
       for (i=0 ; i<v.nelem ; i++) sortie << v.adr[i] << " " ;
       sortie << ">" ;
       return sortie ;
     }
```

A titre indicatif, voici un petit programme utilisant la classe *vect* :

```
#include <vect.h>
#include <iostream.h>
main()
{  int i ;
   vect v1(5), v2(10) ;
   for (i=0 ; i<5 ; i++) v1[i] = i ;
   cout << "v1 = " << v1 << "\n" ;
   for (i=0 ; i<10 ; i++) v2[i] = i*i ;
   cout << "v2 = " << v2 << "\n" ;
   v1 = v2 ;
   cout << "v1 = " << v1 << "\n" ;
   vect v3 = v1 ;
   cout << "v3 = " << v3 << "\n" ;
   vect v4 = v2 ;
   cout << "v4 = " << v4 << "\n" ;
   // const vect w(3) ; w[2] = 5 ;  // conduit bien à erreur compilation
}
```

Exercice XV.3

Enoncé

En langage C++, comme en langage C, il n'existe pas, a priori, de véritable type chaîne, mais simplement une "convention" de représentation des chaînes (suite de caractères, terminée par un caractère de code nul). Un certain nombre de fonctions utilisant cette convention permettent les "manipulations classiques" (en particulier la concaténation).

Créer une classe nommée *string*, offrant des possibilités plus proches d'un véritable type chaîne. On devra y trouver les opérateurs suivants :

- **+**, tel que *ch1+ch2* fournisse en résultat la chaîne obtenue par concaténation des chaînes *ch1* et *ch2* (sans modifier les chaînes *ch1* et *ch2*),,
- **==**, tel que *ch1==ch2* prenne la valeur 1 si les deux chaînes *ch1* et *ch2* sont égales et la valeur 0 dans le cas contraire,
- **[]**, tel que *ch[i]* représente le caractère de rang *i* de la chaîne *ch* ; cet opérateur devra également pouvoir être utilisé à gauche d'une affectation (*ch[i] = ...*),
- **<<**, tel que *flot << ch* transmette au flot indiqué le contenu de la chaîne *ch*.

Une fonction membre *long* fournira la longueur courante d'une chaîne.

Par ailleurs, l'affectation et la transmission par valeur de chaînes devra pouvoir se faire sans problème (on acceptera de dupliquer les chaînes de même contenu).

Enfin, si *ch* est de type *string*, on devra pouvoir accepter des expressions de la forme suivante, et dont le résultat sera, lui aussi de type *string* :

```
ch + "hello"
"hello" + ch
```

Solution

Manifestement, il faudra conserver le contenu d'une chaîne (suite de caractères) dans un emplacement dynamique, de manière à pouvoir en faire évoluer aisément le contenu. Dans les membres donnée, on trouvera donc un pointeur (*char **) sur un tel emplacement. On peut se demander s'il est nécessaire de conserver un caractère de fin de chaîne dans l'emplacement dynamique en question. En fait, à partir du moment où l'on décide de conserver la longueur (courante) d'une chaîne dans un membre donnée, ce caractère de fin n'est plus indispensable ; néanmoins, sa présence facilite les traitements à opérer sur les caractères de la chaîne, dans la mesure où il reste alors possible de faire appel aux fonctions classiques du langage C[1].

En ce qui concerne les constructeurs du type *string*, on peut naturellement prévoir un constructeur sans argument qui initialise la chaîne correspondante à une chaîne vide. Il faut également prévoir un constructeur par recopie.

Mais on peut également ajouter un constructeur qui fabrique une chaîne, à partir d'une chaîne classique (*char **) reçue en argument. En effet, dans ce cas, on disposera alors d'une conversion *char *--> string* qui permettra de donner un sens à la concaténation d'un objet de type *string* et d'une chaîne C classique. Toutefois, cela ne sera possible pour une expression de la forme *"hello" + ch* que si l'opérateur + a été surdéfini sous forme d'une fonction amie (son premier opérande pourra alors être soumis à d'éventuelles conversions) et non pas sous forme d'une fonction membre.

[1] Ce qui ne sera peut-être pas le cas si l'on vise une grande efficacité en temps d'exécution.

Par ailleurs, la valeur de retour de cet opérateur doit absolument être transmise par valeur, dans la mesure où le résultat devra être généré dans une chaîne de classe automatique (elle disparaîtra à la fin de l'exécution de la fonction correspondante).

Voici ce que peut être la déclaration de notre classe *string* :

```
      /* fichier chaine.h : déclaration de la classe chaine */
#include <iostream.h>
#include <string.h>
class string
{ int lg ;                // longueur actuelle de la chaîne
  char * adr ;            // adresse zone contenant la chaîne
 public :
  string () ;             // constructeur I
  string (char *) ;       // constructeur II
  string (string &) ;     // constructeur III (par recopie)
  ~string ()              // destructeur ("inline")
    { delete adr ;
    }
  string & operator = (string &) ;
  long length ()          // longueur courante ("inline")
    { return lg ;
    }
  int operator == (string &) ;
  char & operator [] (int) ;
  friend string operator + (string &, string &) ;
  friend ostream & operator << (ostream &, string &) ;
} ;
```

Voici la définition des différentes fonctions :

```
     /* définitions des fonctions membre de la classe string */
#include <chaine.h>
string::string ()              // constructeur I
   {  lg = 0 ; adr=0 ;
   }

string::string (char * adc)    // constructeur II (à partir d'une chaîne C)
   { lg = strlen (adc) ;
     adr = new char [lg+1]  ;
     strcpy (adr, adc) ;
   }

string::string (string & ch)   // constructeur III (par recopie)
   { lg = ch.lg ;
     adr = new char [lg+1] ;
     strcpy (adr, ch.adr) ;
   }

string & string::operator = (string & ch)
   { if (this != & ch)                 // on ne fait rien pour a=a
       { delete adr ;
         lg = ch.lg ;
         adr = new char [lg+1] ;
         strcpy (adr, ch.adr) ;
```

```
        }
      return * this ;             // pour pouvoir utiliser
   }                              // la valeur de a=b (affectations multiples)

int string::operator == (string & ch)
   { if (strcmp (adr, ch.adr) ) return 0 ;
                           else return 1 ;
   }

char & string::operator [] (int i)
   { return adr[i] ;                        // ici, on n'a pas prévu de
   }                                        // vérification de la valeur de i

     /* définition des fonctions amies de la classe string */
ostream & operator << (ostream & sortie, string & ch)
   { sortie << ch.adr ;
     return sortie ;
   }

string operator + (string & ch1, string & ch2) // attention : la valeur de retour
   { string res ;                               // est à transmettre par valeur
     res.adr = new char [res.lg = ch1.lg + ch2.lg] ;
       strcpy (res.adr, ch1.adr) ;
       strcpy (res.adr+ch1.lg, ch2.adr) ;
     return res ;
   }
```

Voici un programme d'essai de la classe *string*, accompagné du résultat de son exécution :

```
     /* programme d'essai */
main()
{ string a ;
     cout << "chaine a : " << a << " de longueur " << a.length() << "\n" ;
  string b("bonjour") ;
     cout << "chaine b : " << b << "\n" ;
  string c=b ;
     cout << "chaine c : " << c << "\n" ;
  string d("hello") ;
  a = b = d ;
     cout << "chaine b : " << b << " de longueur " << b.length() << "\n" ;
     cout << "chaine a : " << a << "\n" ;
     cout << "a == b   : " << (a == b) << "\n" ;
  string x("salut "), y("chère "), z("madame");
  a = x + y + z ;
     cout << "chaine a : " << a << "\n" ;
  a = a ;
     cout << "chaine a : " << a << "\n" ;
  a = a + " - Comment allez-vous ?\n" ;
     cout << "chaine a : " << a ;
  a = "*****" + a ;
     cout << "chaine a : " << a ;
  // a = "bon" + "jour" ;    // serait rejeté en compilation
  string e("xxxxxxxxxx") ;
  for (char cr='a', i=0 ; cr<'f' ; cr++, i++ ) e[i] = cr ;
     cout << "chaine e : " << e << "\n" ;
}
```

```
chaine a :  de longueur 0
chaine b : bonjour
chaine c : bonjour
chaine b : hello de longueur 5
chaine a : hello
a == b   : 1
chaine a : salut chère madame
chaine a : salut chère madame
chaine a : salut chère madame - Comment allez-vous ?
chaine a : *****salut chère madame - Comment allez-vous ?
chaine e : abcdexxxxx
```

Exercice XV.4

Enoncé

Réaliser une classe nommée *bit_array* permettant de manipuler des tableaux de bits (autrement dit des tableaux dans lesquels chaque élément ne peut prendre que l'une des deux valeurs 0 ou 1. La taille d'un tableau (c'est-à-dire le nombre de bits) sera définie lors de sa création (par un argument passé à son constructeur). On prévoira les opérateurs suivants :

+=, tel que *t+=n* mette à 1 le bit de rang *n* du tableau *t*,

-=, tel que *t-=n* mette à 0 le bit de rang *n* du tableau *t*,

[], tel que l'expression *t[i]* fournisse la valeur du bit de rang *i* du tableau *t* (on ne prévoira pas, ici, de pouvoir employer cet opérateur à gauche d'une affectation, comme dans *t[i] = ...*),

++, tel que *t++* mette à 1 tous les bits de *t*,

--, tel que *t--* mette à 0 tous les bits de *t*,

<<, tel que *fbt << t* envoie le contenu de *t* sur le flot indiqué, sous la forme

```
<* bit1, bit2, ... bitn *>
```

On fera en sorte que l'affectation et la transmission par valeur d'objets du type *bit_array* ne pose aucun problème. De plus, les opérateurs modifiant le contenu d'un tableau ne devront pas pouvoir être appliqués à des objets constants (attribut *const*).

Solution

Si l'on cherche à minimiser l'emplacement mémoire utilisé pour les objets de type *bit_array*, il est nécessaire de n'employer qu'un seul bit pour représenter un "élément" d'un tableau. Ces bits devront donc être regroupés, par exemple à raison de *CHAR_BIT* (défini dans *limits.h*) bits par caractère.

Manifestement, il faut prévoir que l'emplacement destiné à ces différents bits soit alloué dynamiquement en fonction de la valeur fournie au constructeur : pour n bits, il faudra n/CHAR_BIT+1 caractères.

En membre donnée, il nous suffit de disposer d'un pointeur sur l'emplacement dynamique en question, ainsi que du nombre de bits du tableau. Pour simplifier certaines des fonctions membre, nous prévoirons également de conserver le nombre de caractères correspondant.

Les opérateurs + =, - =, + + et -- peuvent être définis indifféremment sous forme d'une fonction membre ou d'une fonction amie. Ici, nous avons choisi des fonctions membre.

L'énoncé ne précise rien quant au résultat fourni par ces 4 opérateurs. En fait, on pourrait prévoir qu'ils restituent le tableau après qu'ils y ont effectué l'opération voulue, mais, en pratique, cela semble de peu d'intérêt. Ici, nous avons donc simplement prévu que ces opérateurs ne fourniraient aucun résultat.

Pour que [] ne soit pas utilisable dans une affectation de la forme *t[i]* = ..., il suffit de prévoir qu'il fournisse son résultat par valeur (et non par référence comme on a généralement l'habitude de le faire).

Les opérateurs + =, - =, + + et -- modifient la valeur de leur premier opérande. Ils ne doivent donc pas pouvoir agir sur un objet constant. En revanche, les autres, c'est-à-dire [] (ici) et < <, peuvent agir sans risque sur un objet constant ;pour qu'ils puissent y être autorisés, nous leur donnerons l'attribut *const*.

Naturellement, ici encore, l'énoncé nous impose de surdéfinir l'opérateur d'affectation et de prévoir un constructeur par recopie.

Voici ce que pourrait être la déclaration de notre classe *bit_array* :

```
    /* fichier bitarray.h : déclaration de la classe bit_array */
#include <iostream.h>
class bit_array
{   int nbits ;       // nombre courant de bits du tableau
    int ncar ;        // nombre de caractères nécessaires (redondant)
    char * adb ;      // adresse de l'emplacement contenant les bits
  public :
    bit_array (int = 16) ;      // constructeur usuel
    bit_array (bit_array &) ;   // constructeur par recopie
    ~bit_array () ;             // destructeur
                                // les opérateurs binaires
    bit_array & operator = (bit_array &) ;                  // affectation
    int operator [] (int) const ;                           // valeur d'un bit
    void operator += (int) ;                                // activation d'un bit
    void operator -= (int) ;                                // désactivation d'un bit
                                                            // envoi sur flot
    friend ostream & operator << (ostream &, bit_array &) const ;
                                // les opérateurs unaires
    void operator ++ () ;                                   // mise à 1
    void operator -- () ;                                   // mise à 0
    void operator ~ () ;                                    // complément à 1
} ;
```

Voici la définition des différentes fonctions.

```
     /* définition des fonctions de la classe bit_array */
#include "bitarray.h"
#include <limits.h>

bit_array::bit_array (int nb)
{  nbits = nb ;
   ncar = nbits / CHAR_BIT + 1 ;
   adb = new  char [ncar] ;
   int i ;
   for (i=0 ; i<ncar ; i++) adb[i] = 0 ;    // raz
}
```

```
bit_array::bit_array (bit_array & t)
{  nbits = t.nbits ; ncar = t.ncar ;
   adb = new  char [ncar] ;
   int i ;
   for (i=0 ; i<ncar ; i++) adb[i] = t.adb[i] ;
}
bit_array::~bit_array()
{  delete adb ;
}
bit_array & bit_array::operator = (bit_array & t)
{  if (this != & t)      //  on ne fait rien pour t=t
     { delete adb ;
       nbits = t.nbits ; ncar = t.ncar ;
       adb = new char [ncar] ;
       int i ;
       for (i=0 ; i<ncar ; i++)
          adb[i] = t.adb[i] ;
     }
   return *this ;
}

int bit_array::operator [] (int i) const
{       // le bit de rang i s'obtient en considérant le bit
        //   de rang i % CHAR_BIT du caractère de rang i / CHAR_BIT
    int carpos = i / CHAR_BIT ;
    int bitpos = i % CHAR_BIT ;
    return  ( adb [carpos] >> CHAR_BIT - bitpos -1 ) & 0x01 ;
}

void bit_array::operator += (int i)
{    int carpos = i / CHAR_BIT ;
     if (carpos < 0 || carpos >= ncar) return ;    // protection
     int bitpos = i % CHAR_BIT ;
     adb [carpos] |= (1 << (CHAR_BIT - bitpos - 1) ) ;
}

void bit_array::operator -= (int i)
{    int carpos = i / CHAR_BIT ;
     if (carpos < 0 || carpos >= ncar) return ;    // protection
     int bitpos = i % CHAR_BIT ;
     adb [carpos] &= ~(1 << CHAR_BIT - bitpos - 1) ;
}

ostream & operator << (ostream & sortie, bit_array & t) const
{   sortie << "<* " ;
    int i ;
    for (i=0 ; i<t.nbits ; i++)
       sortie << t[i] << " " ;
    sortie << "*>" ;
    return sortie ;
}

void bit_array::operator ++ ()
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = 0xFFFF ;
}
```

```
void bit_array::operator -- ()
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = 0 ;
}

void bit_array::operator ~ ()
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = ~ adb[i] ;
}
```

Voici un programme d'essai de la classe *bit_array*, accompagné du résultat fourni par son exécution :

```
     /* programme d'essai de la classe bit_array */
main ()
{ bit_array t1 (34) ;
  cout << "t1 = " << t1 << "\n" ;
  t1 +=3 ; t1 += 0 ; t1 +=8 ; t1 += 15 ; t1 += 33 ;
  cout << "t1 = " << t1 << "\n" ;
  t1-- ;
  cout << "t1 = " << t1 << "\n" ;
  t1++ ;
  cout << "t1 = " << t1 << "\n" ;
  t1 -= 0 ; t1 -= 3 ; t1 -= 8 ; t1 -= 15 ; t1 -= 33 ;
  cout << "t1 = " << t1 << "\n" ;
  cout << "t1 = " << t1 << "\n" ;
  bit_array t2 (11), t3 (17) ;
  cout << "t2 = " << t2 << "\n" ;
  t2 = t3 = t1 ;
  cout << "t3 = " << t3 << "\n" ;
}
```

```
t1 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 *>
t1 = <* 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 *>
t1 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 *>
t1 = <* 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 *>
t1 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 *>
t1 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 *>
t2 = <* 0 0 0 0 0 0 0 0 0 0 0 *>
t3 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 *>
```

Exercice XV.5

Enoncé

La capacité des nombres entiers est limitée par la taille du type *bngint*. Créer une classe *big_int* permettant de manipuler des nombres entiers de **valeur absolument quelconque**.

Pour ne pas alourdir l'exercice, on se limitera à des nombres sans signe et à l'opération d'addition ; on s'arrangera toutefois pour que des expressions mixtes (c'est-à-dire mélangeant des objets de type *bng_int* avec des entiers usuels) aient un sens.

On définira l'opérateur << pour qu'il permette d'envoyer un objet de type *big_int* sur un flot. Parmi les différents constructeurs, on en prévoira un avec un argument de type chaîne de caractères, correspondant aux chiffres d'un "grand entier".

On fera en sorte que l'affectation et la transmission par valeur d'objets de type *big_int* ne pose aucun problème.

Solution

Pour représenter un "grand entier", la démarche la plus naturelle (mais pas la plus économique en place mémoire!) consiste à conserver le nombre sous forme décimale, à chaque chiffre étant associé un caractère. Pour ce faire, on peut choisir de "coder" un tel chiffre par le caractère correspondant ('0' pour 0, '1' pour 1, ...) ; on peut aussi choisir de placer une valeur égale au chiffre lui-même (0 pour 0, 1 pour 1, ...). La dernière solution oblige à effecteur un "transcodage" lorsque l'on doit passer de la forme chaîne de caratères à la forme *big_int* (dans le constructeur correspondant, notamment) ou, inversement, lorsque l'on doit passer de la forme *big_int* à la forme suite de caractères (pour l'affichage). En revanche, elle simplifie quelque peu l'algorithme d'addition, et c'est elle que nous avons choisie.

L'emplacement permettant de conserver un grand entier sera alloué dynamiquement ; sa taille sera, naturellement, adaptée à la valeur du nombre qui s'y trouvera. On conservera également le nombre courant de chiffres de l'entier ; on pourrait, en toute rigueur, s'en passser mais nous verrons que sa présence simplifie quelque peu la programmation. En ce qui concerne l'ordre de rangement des chiffres au sein de l'emplacement correspondant, il y a manifestement deux possibilités. Chacune possède des avantages et des inconvénients ; nous avons ici choisi de ranger les chiffres dans l'ordre inverse de celui où on les écrit (unités, dizaines, centaines...).

Pour pouvoir accepter les expressions mixtes, on dispose de plusieurs solutions :

- soit surdéfinir l'opérateur + pour tous les cas possibles,
- soit surdéfinir + uniquement lorsqu'il porte sur des grands entiers et prévoir un constructeur recevant un argument de type *unsigned long* ; il permettra ainsi la conversion en *big_int* de n'importe quel type numérique.

C'est la deuxième solution que nous avons adoptée. Notez toutefois que, si elle a le mérite d'être la plus simple à programmer, elle n'est pas la plus efficace en temps d'exécution.

Par ailleurs, pour que les conversions envisagées s'appliquent au premier opérande de l'addition, il est nécessaire de surdéfinir l'opérateur + comme une fonction amie.

Voici la déclaration de notre classe *big_int* (la présence d'un constructeur privé à deux arguments entiers sera justifiée un peu plus loin) :

```
      /* fichier bigint.h : déclaration de la classe big_int */
#define NCHIFMAX 32    // nombre maxi de chiffres d'un entier (dépend de
                  //     l'implémentation
#include <iostream.h>
class big_int
{   int nchif ;                  // nombre de chiffres
    char * adchif ;              // adresse emplacement contenant les chiffres
    big_int (int, int) ;         // constructeur privé (à usage interne)
  public :
    big_int (unsigned long=0) ;  // constructeur à partir d'un nombre usuel
    big_int (char *) ;           // constructeur à partir d'une chaîne
    big_int (big_int &) ;                        // constructeur par recopie
    big_int & operator = (big_int &) ;                      // affectation
    friend big_int operator + (big_int &, big_int &) ;      // opérateur +
```

```
    friend ostream & operator << (ostream &, big_int &) ;   // opérateur <<
} ;
```

L'opérateur + commence par créer un emplacement temporaire pouvant recevoir un nombre comportant un chiffre de plus que le plus grand de ses deux opérandes (on ne sait pas encore combien de chiffres comportera exactement le résultat). On y calcule la somme suivant un algorithme calqué sur le processus manuel d'addition.

Puis on crée un objet de type *big_int* en utilisant un constructeur particulier : *big_int (int, int)*. En fait, nous avons besoin d'un constructeur créant un *big_int* comportant un nombre de chiffres donné, ce dont nous ne disposons pas dans les constructeurs publics. De plus, nous ne pouvons pas utiliser un constructeur de la forme *big_int (int)* car, alors, les additions mixtes faisant intervenir des entiers chercheraient à l'employer pour effectuer une conversion! C'est pourquoi nous avons prévu un constructeur à deux arguments, le second étant fictif ;de plus, nous l'avons rendu privé, dans la mesure où il n'a nullement besoin d'être accessible à un utilisateur de la classe.

Voici la définition des fonctions de la classe *big_int*

```
      /* définition des fonctions de la classe big_int */
#include <string.h>
#include <iostream.h>
#include "bigint.h"

big_int::big_int (int n, int p)    // l'argument p est fictif
{   nchif = n ;
    adchif = new char [nchif] ;
}

big_int::big_int (char * ch)
{
    nchif = strlen (ch) ;
    adchif = new char [nchif] ;
    int i ; char c ;
    for (i=0 ; i<nchif ; i++)
      { c = ch[i] - '0' ;
        if (c<0 || c>9) c=0 ;       // précaution
        adchif[nchif-i-1] = c ;     // attention à l'ordre des chiffres !
      }
}

big_int::big_int (unsigned long n)
{     // on crée le grand entier correspondant dans un emplacement temporaire
    char * adtemp = new char [NCHIFMAX] ;
    int i = 0 ;
    while (n)
      { adtemp [i++] = n % 10 ;
        n /= 10 ;
      }
       // ici i contient le nombre exact de chiffres
    nchif = i ;
    adchif = new char [nchif] ;
    for (i=0 ; i<nchif ; i++)
        adchif [i] = adtemp [i] ;
       // on libère l'emplacement temporaire
    delete adtemp ;
}
```

```
big_int::big_int (big_int & n)
{  nchif = n.nchif ;
   adchif = new char [nchif] ;
   int i ;
   for (i=0 ; i<nchif ; i++)
       adchif [i] = n.adchif [i] ;
}

big_int & big_int::operator = (big_int & n)
{  if (this != &n)
      { delete adchif ;
        nchif = n.nchif ;
        adchif = new char [nchif] ;
        int i ;
        for (i=0 ; i<nchif ; i++)
           adchif [i] = n.adchif [i] ;
      }
   return * this ;
}

big_int operator + (big_int & n, big_int & p)
{   int nchifmax = (n.nchif > p.nchif) ? n.nchif : p.nchif ;
    int ncar = nchifmax + 1 ;
       // préparation du résultat dans zone temporaire de taille ncar
    char * adtemp = new char [ncar] ;
    int i, s, chif1, chif2 ;
    int ret = 0 ;
    for (i=0 ; i<nchifmax ; i++)
      { chif1 = (i<n.nchif) ? n.adchif [i] : 0 ;
        chif2 = (i<p.nchif) ? p.adchif [i] : 0 ;
        s = chif1 + chif2 + ret ;
        if (s>=10) { s -= 10 ;
                     ret = 1 ;
                   }
              else ret = 0 ;
        adtemp [i] = s ;
      }
    if (ret == 1) adtemp [ncar-1] = 1 ;
             else ncar-- ;
      // construction d'un objet de type big_int où l'on recopie le résultat
    big_int res (ncar, 0) ;      // second argument fictif
    res.nchif = ncar ;
    for (i=0 ; i<ncar ; i++)
       res.adchif [i] = adtemp [i] ;
    delete adtemp ;
    return res ;
}

ostream & operator << (ostream & sortie, big_int & n)
{   int i ;
    for (i=n.nchif-1 ; i>=0 ; i--)       // attention à l'ordre !
       sortie << n.adchif [i] + '0' ;
    return sortie ;
}
```

Voici un petit programme d'utilisation de la classe *big_int*, accompagné du résultat fourni par son exécution :

```
     /* programme d'essai */
#include <iostream.h>
#include "bigint.h"
main()
{  big_int n1=12 ;
   big_int n2(35) ;
   big_int n3 ;
   n3 = n1 + n2 ;
   cout << n1 << " + " << n2 << " = " << n3 << "\n" ;
   big_int n4 ("1234567890123456789"), n5("9876543210987654321"), n6 ;
   n6 = n4 + n5 ;
   cout << n4 << " + " << n5 << " = " << n6 << "\n" ;
   cout << n6 << " + " << n1 << " = " << n6 + n1 << "\n" ;
}
```

```
12 + 35 = 47
1234567890123456789 + 9876543210987654321 = 11111111101111111110
11111111101111111110 + 12 = 11111111101111111122
```

Exercice XV.6

Enoncé

Créer un patron de classes nommé *stack*, permettant de manipuler des piles dont les éléments sont de type quelconque. Ces derniers seront conservés dans un emplacement alloué dynamiquement et dont la dimension sera fournie au constructeur (il ne s'agira donc pas d'un paramètre expression du patron). La classe devra comporter les opérateurs suivants :

- `<<`, telque `p<<n` ajoute l'élément `n` à la pile `p` (si la pile est pleine, il ne se passera rien),
- `>>`, telque `p>>n` place dans `n` la valeur du haut de la pile `p`, en la supprimant de la pile (si la pile est vide, il ne se passera rien),
- `++`, telque `++p` vaut 1 si la pile `p` est pleine et 0 dans le cas contraire,
- `--`, telque `--p` vaut 1 si la pile `p` est vide et 0 dans le cas contraire,
- `<<`, telque, `flot` étant un `flot` de sortie, `flot << p` affiche le contenu de la pile `p` sur le `flot` sous la forme : *// valeur_1 valeur_2... valeur_n //*.

On supposera que les objets de type *stack* ne seront jamais soumis à des transmissions par valeur ou à des affectations ; on ne cherchera donc pas à surdéfinir le constructeur par recopie ou l'opérateur d'affectation.

Solution

En fait, on peut s'inspirer de ce qui a été fait dans l'exercice VII.10 pour réaliser une pile d'entiers en faisant en sorte que *int* soit remplacé par un paramètre de type.

Voici ce que pourrait être la définition de notre patron de classes :

```
#include <iostream.h>
#include <stdlib.h>
```

```
template <class T> class stack
{   int nmax ;                  // nombre maximum de la valeurs de la pile
    int nelem ;                 // nombre courant de valeurs de la pile
    T * adv ;                   // pointeur sur les valeurs
  public :
    stack (int = 20) ;           // constructeur
    ~stack () ;                  // destructeur
    stack & operator << (T) ;   // opérateur d'empilage
    stack & operator >> (T &) ; // opérateur de dépilage (attention T &)
    int operator ++ () ;             // opérateur de test pile pleine
    int operator -- () ;             // opérateur de test pile vide
                                     // opérateur << pour flot de sortie
    friend ostream & operator << (ostream &, stack<T> &) ;
} ;
template <class T> stack<T>::stack (int n)
{   nmax = n ;
    adv = new T [nmax] ;
    nelem = 0 ;
}
template <class T> stack<T>::~stack ()
{   delete adv ;
}
template <class T> stack<T> & stack<T>::operator << (T n)
{   if (nelem < nmax) adv[nelem++] = n ;
    return (*this) ;
}
template <class T> stack<T> & stack<T>::operator >> (T & n)
{   if (nelem > 0) n = adv[--nelem] ;
    return (*this) ;
}
template <class T> int stack<T>::operator ++ ()
{   return  (nelem == nmax) ;
}
template <class T> int stack<T>::operator -- ()
{   return (nelem == 0) ;
}
template <class T> ostream & operator << (ostream & sortie, stack<T> & p)
{ sortie << "// " ;
  int i ;
  for (i=0 ; i<p.nelem ; i++) sortie << p.adv[i] << " " ;
  sortie << " " ;
}
```

A titre indicatif, voici un petit programme d'utilisation de notre patron de classes (dont on suppose que la définition figure dans *stack_gen.h*) :

```
     /************ programme d'essai de stack *********/
#include "stack_gen.h"
#include <iostream.h>
main()
{
    stack <int> pi(20) ;           // pile de 20 entiers maxi
    cout << "pi pleine : " << ++pi << " vide : " << --pi << "\n" ;
    pi << 2 << 3 << 12 ;
    cout << "pi = " << pi << "\n" ;
    stack <float> pf(10) ;         // pile de 10 flottants maxi
    pf << 3.5 << 4.25 << 2 ;       // 2 sera converti en float
```

```
    cout << "pf = " << pf << "\n" ;
    float x ;  pf >> x ;
    cout << "haut de la pile pf = " << x ;
    cout << "pf = " << pf << "\n" ;
}
```

Exercice XV.7

Enoncé

En vous inspirant de l'exercice XV.2, créer un patron de classes permettant de manipuler des vecteurs dynamiques dont les éléments sont de type quelconque.

Solution

Voici la déclaration de notre patron :

```
template <class T> class vect
{  int nelem ;                  // nombre de composantes du vecteur
   T * adr ;                    // pointeur sur partie dynamique
 public :
   vect (int n=1) ;             // constructeur "usuel"
   vect (vect & v) ;            // constructeur par recopie
   ~vect () ;                   // destructeur
   friend ostream & operator << (ostream &, vect <T> &) ;
   vect<T> operator = (vect<T> & v) ;  // surdéfinition opérateur affectation
   T & operator [] (int i) ;           // surdef [] pour vect non constants
   T   operator [] (int i) const ;     // surdef [] pour vect constants
} ;
```

Voici la définition des différentes fonctions membre :

```
#include "vectgen.h"
template <class T> vect<T>::vect (int n)      // constructeur "usuel"
     { adr = new T [nelem = n] ;
     }

template <class T> vect<T>::vect (vect<T> & v)   // constructeur par recopie
     { adr = new T [nelem = v.nelem] ;
       int i ;
       for (i=0 ; i<nelem ; i++)
           adr[i] = v.adr[i] ;
     }

template <class T> vect<T>::~vect ()
     { delete adr ;
     }

template <class T> vect<T>  vect<T>::operator = (vect<T> & v)
     { if (this != &v)              // on ne fait rien pour a=a
```

```
          { delete adr ;
            adr = new T [nelem = v.nelem] ;
            int i ;
            for (i=0 ; i<nelem ; i++)
              adr[i] = v.adr[i] ;
          }
        return * this ;
      }

template <class T> T & vect<T>::operator [] (int i)
      { return adr[i] ;
      }

template <class T> T   vect<T>::operator [] (int i) const
      { return adr[i] ;
      }

template <class T> ostream & operator << (ostream & sortie, vect<T> & v)
      { sortie << "<" ;
        int i ;
        for (i=0 ; i<v.nelem ; i++) sortie << v.adr[i] << " " ;
        sortie << ">" ;
        return sortie ;
      }
```

A titre indicatif, voici un petit programme utilisant notre patron :

```
#include <vectgen.h>
#include <iostream.h>
main()
{  int i ;
   vect <int> v1(5) ; vect <int> v2(10) ;
   for (i=0 ; i<5 ; i++) v1[i] = i ;
   cout << "v1 = " << v1 << "\n" ;
   for (i=0 ; i<10 ; i++) v2[i] = i*i ;
   cout << "v2 = " << v2 << "\n" ;
   v1 = v2 ;
   cout << "v1 = " << v1 << "\n" ;
   vect <int> v3  = v1 ;
// vect <double> v3 = v1 ;               //  serait rejete
   cout << "v3 = " << v3 << "\n" ;
// const vect <float> w(3) ; w[2] = 5 ;  // conduit bien a erreur compilation
// vect <float> v4(5) ; v4 = v1 ;        // conduit bien a erreur compilation
}
```