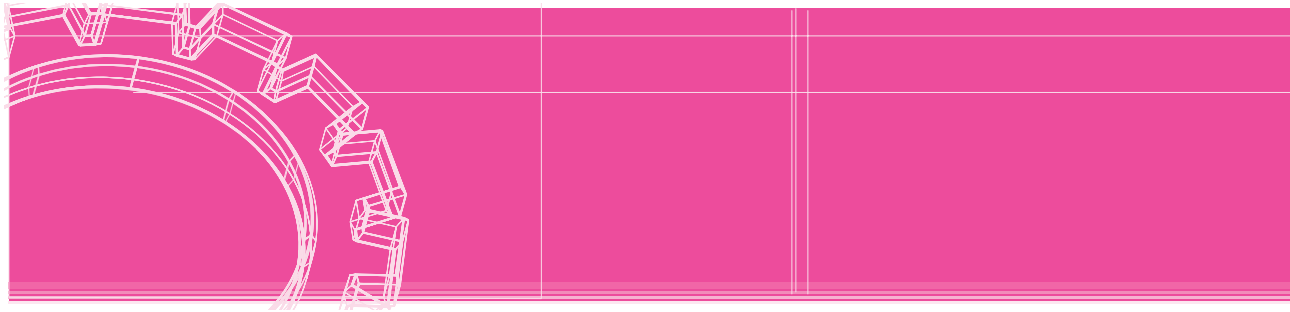
Chapitre 9

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 lorsqu'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.

© Éditions Eyrolles

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 qu'il ne sera plus utile. Par exemple, si une classe `point` possède le constructeur `point (float, float)` et si `a` est de type `point`, nous pouvons écrire :

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

Remarque Ne confondez pas une telle affectation avec une initialisation d'un objet lors de sa déclaration.

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`.

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, 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) ; // création d'un objet de type point, par
                              // appel d'un constructeur à deux arguments
```

Remarque Comme pour les variables ordinaires, on peut préciser un nombre d'objets, mais, des restrictions apparaissent alors, qui portent sur le constructeur qu'il est possible d'appeler (voyez ci-après la rubrique « tableaux d'objets »).

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`. (Dans le cas d'un tableau d'objets, la syntaxe sera différente, voir un peu plus loin.)

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 :

© Éditions Eyrolles

```
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 mentionnés 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

En C++, on parle d'initialisation d'un objet dans des situations telles que :

```
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
```

Le deuxième cas correspond à l'initialisation d'un objet à l'aide d'un autre objet de même type, qu'on nomme parfois « initialisation par recopie ». L'opération est réalisée par appel de ce que l'on nomme un « constructeur par recopie ».

Mais il existe d'autres situations, plus courantes, qui mettent en œuvre un tel mécanisme, à savoir :

- la transmission d'un objet par valeur en argument d'appel d'une fonction ;
- la transmission d'un objet par valeur en valeur de retour d'une fonction.

Dans toutes ces situations d'initialisation par recopie, le constructeur par recopie employé est :

- soit un constructeur de la forme `type (type &)` ou `type (const type &)` s'il en existe un ; ce dernier doit alors prendre en charge la recopie de tous les membres de l'objet, y compris ceux qui sont des objets ; il peut cependant s'appuyer sur les possibilités de transmission d'information entre constructeurs, présentée auparavant ;

Attention

La transmission par référence est obligatoire ici. D'autre part, le fait d'utiliser l'attribut `const` permet d'appliquer le constructeur à un objet constant ou à une expression ; rappelons que, dans ce cas, il y création d'un objet temporaire dont on transmet la référence au constructeur.

- soit, dans le cas contraire, ce que l'on nomme un « constructeur de recopie par défaut », qui recopie les différents membres de l'objet. Si certains de ces membres sont eux-mêmes des objets, la recopie sera réalisée par appel de son propre constructeur par recopie (qui pourra être soit un constructeur par défaut, soit un constructeur défini dans la classe correspondante).

Remarque

Dans les très anciennes versions (antérieures à 2.0), la recopie se faisait de façon globale ; autrement dit, la « valeur » d'un objet était reportée (bit par bit) dans un autre, sans tenir compte de sa structure. Les choses étaient alors relativement peu satisfaisantes...

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 une syntaxe particulière de l'opérateur `new`. Par exemple, pour détruire le tableau précédent, on écrira :

```
delete [] adcourbe ;
```


© Éditions Eyrolles

Remarque

En théorie, 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. Ces valeurs doivent être constantes pour les tableaux de classe statique ; il peut s'agir d'expressions pour les tableaux de classe automatique. On peut ne pas préciser de valeurs pour les derniers éléments du tableau.

On notera qu'un tel initialiseur ne peut pas être utilisé avec des tableaux dynamiques (il en va de même pour les tableaux ordinaires !).

Exercice 74

Énoncé

Comment concevoir le type classe `chose` de façon 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
```

© Éditions Eyrolles

Exercice 75

Énoncé

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>
using namespace std ;
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)           // ou demo::demo (const 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) ;
```


© Éditions Eyrolles

```
   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 que fournit le programme (ici, nous avons utilisé le compilateur C++ Builder X ; ce point n'ayant d'importance que pour les objets temporaires, dans le cas des implémentations qui ne respecteraient pas la norme quant à l'instant de leur destruction).

La plupart des lignes sont assorties d'explications complémentaires présentées conventionnellement sous forme de commentaires et qui précisent les instructions concernées.

```
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, ...)   */
                                  /*   ce qui crée un objet temporaire            */
entrée fct
destruction              : 3 3   /* delete add ;   (dans fct)                    */
sortie fct
destruction              : 1 0   /* destruction objet temporaire créé pour       */
                                  /*  l'appel de fct                              */
constructeur I           : 4 4   /* demo d = demo(4, 4)                          */
constructeur I           : 5 5   /* c = demo(5, 5) (contruction objet temporaire) */
destruction              : 5 5   /* destruction objet temporaire précédent       */
fin main
destruction              : 4 4   /* destruction d */
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 objet est effectivement détruit aussitôt après. Mais il existe certaines implémentations qui ne respectent pas la norme et où cela peut se produire plus tard.

© Éditions Eyrolles

Par ailleurs, l'appel de `fct` a entraîné la construction d'un objet temporaire, par appel du constructeur par recopie. Cet objet est ici libéré dès la sortie de la fonction. Là encore, dans certaines implémentations, cela peut se produire plus tard.

Exercice 76

Énoncé

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...). Le constructeur affichera le numéro du point créé et le destructeur affichera le numéro du point détruit. Écrire 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. Comme tout membre statique, `nb_points` devra être initialisé.

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

```
#include <iostream>
using namespace std ;
class point
{   int num;
    static int nb_points ;
  public :
    point ()
       {  num = ++nb_points ;
          cout << "création point numéro    : " << num << "\n" ;
       }
    ~point ()
       {  cout << "Destruction point numéro : " << num << "\n" ;
       }
} ;
int point::nb_points=0 ;     // initialisation obligatoire
main()
{  point * adcourb = new point [4] ;
   delete [] adcourb ;
}
```


© Éditions Eyrolles

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

```
création point numéro    : 1
création point numéro    : 2
création point numéro    : 3
création point numéro    : 4
Destruction point numéro : 4
Destruction point numéro : 3
Destruction point numéro : 2
Destruction point numéro : 1
```

Exercice 77

Énoncé

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. Écrire, en outre, un programme (`main`) utilisant la classe `set_int` pour déterminer le nombre d'entiers différents contenus dans 20 entiers lus en données.

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 ?

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

Solutions

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=0 ;
     // on examine si nb appartient déjà à l'ensemble
     // (si ce n'est pas le cas, i vaudra nele en fin de boucle)
   while ( (i<nelem) && (adval[i] != nb) ) i++ ;
   return (i<nelem) ;
}
int set_int::cardinal ()
{  return nelem ;
}
```


© Éditions Eyrolles

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 que l'utilisateur soit 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 "setint1.h"
#include <iostream>
using namespace std ;
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" ;
}
```

À titre indicatif, voici un exemple d'exécution :

```
donnez 20 entiers
0 2 5 2 8 5 1 8 2 0 7 2 5 5 4 5 0 0 4 5
il y a : 7 entiers différents
```

3. Telle 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 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 pourraient se poser si la fonction était amenée à modifier le contenu de l'ensemble ; en effet, on modifierait alors le tableau d'entiers original, chose à laquelle on ne s'attend pas dans le cas de la transmission par valeur.

Pour régler ces problèmes, il est nécessaire de munir notre classe d'un constructeur par recopie approprié, c'est-à-dire tenant compte de la « partie dynamique » de l'objet (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 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)  // ou set_int::set_int (const 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 surdéfinition du constructeur par recopie est quasiment indispensable pour toute classe comportant un pointeur sur une partie dynamique. En 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 » un tel objet, en lui demandant de ne jamais le transmettre par valeur ! Cela 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 78

Énoncé

Modifier l'implémentation de la classe précédente (avec son constructeur par recopie) de façon 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.

© Éditions Eyrolles

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 nœud suivant de la liste
} ;
```

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 cela 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 nœud 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 nœuds 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)    // ou :  set_int::set_int (const 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 nœud
         adbut->valeur = adsource->valeur ; // copie valeur
         adbut->suivant = debut ;           // insertion nouveau nœud
         debut = adbut ;                    //     dans nouvelle liste
         adsource = adsource->suivant ;     // nœud suivant ancienne liste
       }
}

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


© Éditions Eyrolles

```
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 26 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 (hormis un éventuel manque de mémoire).

Exercice 79

Énoncé

Modifier la classe `set_int` précédente (implémentée sous la 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'il s'agit d'un mécanisme permettant d'accéder séquentiellement aux différents éléments de l'ensemble. On prévoira trois nouvelles fonctions membre : `init`, pour initialiser le processus d'itérationþ; `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 26), de manière qu'il affiche les différents entiers contenus dans les valeurs fournies en donnée.

© Éditions Eyrolles

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici. D'autre part, cet exercice sera plus profitable s'il est traité après l'exercice du chapitre 3 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 nœud 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 nœud 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 qu'il pointe sur le nœud 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 nœud 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 nœud 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
```


© Éditions Eyrolles

```
   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 "setint1.h"
#include <iostream>
using namespace std ;
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" ;
   cout << "Ce sont : \n" ;
   ens.init () ;
   while (ens.existe ())
      cout << ens.prochain() << " " ;
}
```

À titre indicatif, voici un exemple d'exécution :

```
donnez 20 entiers
0 2 1 5 4 1 2 0 2 1 4 5 1 2 0 2 1 4 5 2
il y a : 5 entiers différents
Ce sont :
4 5 1 2 0
```

Exercice 80

Énoncé

Quels seront les résultats fournis par l'exécution de ce programme :

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
 public :
  point ()
    { x=0 ; y=0 ;
      cout << "** constructeur 0 argument\n" ;
    }
  point (int abs)
    { x=abs ; y=0 ;
      cout << "** constructeur 1 argument\n" ;
    }
  point (int abs, int ord)
    { x=abs ; y=ord ;
      cout << "** constructeur 2 arguments\n" ;
    }
  point (point & p)
    { x=p.x ; y=p.y ;
      cout << "**constructeur par recopie\n" ;
    }
  void affiche ()
    { cout << "point de coordonnees : " << x << " " << y << "\n" ;
    }
} ;
main()
{ point a(10,20) ;
  point b(30,40) ;
  point courbe[6] = { 4, a, 0, b} ;
  for (int i=0 ; i<6 ; i++) courbe[i].affiche() ;
}
```


© Éditions Eyrolles

Solution

```
** constructeur 2 arguments                // construction (classique) de a
** constructeur 2 arguments                // construction (classique) de b
** constructeur 1 argument                 // construction courbe[0]
**constructeur par recopie                 // construction courbe[1]
** constructeur 1 argument                 // construction courbe[2]
**constructeur par recopie                 // construction courbe[3]
** constructeur 0 argument                 // construction courbe[4]
** constructeur 0 argument                 // construction courbe[5]
point de coordonnees : 4 0
point de coordonnees : 10 20
point de coordonnees : 0 0
point de coordonnees : 30 40
point de coordonnees : 0 0
point de coordonnees : 0 0
```

Après la construction classique des points `a` et `b`, il y a création d'un tableau de 6 objets de type `point`. Les quatre premiers points sont construits par :

- appel d'un constructeur à un argument de type `int` (valeur `4`) pour le premier pointþ;
- appel d'un constructeur à un argument de type `point` (valeur `a`) pour le deuxièmeþ;
- appel d'un constructeur à un argument de type `int` (valeur `0`) pour le troisièmeþ;
- appel d'un constructeur à un argument de type `point` (valeur `b`) pour le quatrième.

Les deux derniers points du tableau ne disposent pas d'initialiseur ; ils sont donc construits par appel d'un constructeur sans argument.