Chapitre 3

Classes et objets

Nous abordons (enfin) la programmation orientée objet proprement dite en commençant par la notion essentielle : celle de *classe*, les *objets* n'étant rien d'autre que les instances d'une classe. Les autres notions essentielles, l'*héritage*, la *surcharge* et le *polymorphisme*, seront abordés dans les chapitres suivants.

3.1 La programmation orientée objet

La *programmation orientée objet* est une nouvelle philosophie de programmation ayant pour but de mettre un peu d'ordre dans la gestion des sous-programmes. Les premiers langages de programmation étaient très liés au fonctionnement des ordinateurs, même les langages évolués, en particulier à cause du GOTO ; la programmation structurée a permis de mieux structurer les programmes. Cependant avec la programmation modulaire, d'une part, on arrive à de nombreux sous-programmes dont on ne sait plus très bien ce qu'ils font et, d'autre part, les arguments d'un sous-programme peuvent ne pas être clairement explicités comme arguments (quand ce sont des variables globales). La programmation orientée objet veut éviter ces deux difficultés.

3.1.1 Notion d'objet

L'idée fondamentale de la programmation orientée objet est que l'on considère des **objets**, à savoir une association de données et de fonctions qui les concernent. Les fonctions sont appelées les **méthodes** de l'objet, les données ses **attributs**.

Exemple.- On veut manipuler des points du plan. On considèrera chaque point comme un objet. Un point sera tout naturellement défini par deux coordonnées, donc deux données. Les méthodes que l'on peut vouloir à propos d'un point sont, par exemple, de l'*initialiser*, de le *déplacer* et de l'*afficher*. L'initialiser peut consister à lui donner des coordonnées dans un repère donné. Le déplacer consiste à le translater d'un vecteur donné, caractérisé également par ses deux coor-

données. L'afficher est soit l'afficher à l'écran, soit faire afficher les valeurs de ses coordonnées.

3.1.2 Principe d'encapsulation des données

Cette notion d'objet permet de rassembler certaines fonctions agissant sur les données de l'objet. On veut aller plus loin : on ne doit pouvoir agir sur ces données que grâce aux méthodes de l'objet, autrement dit les attributs de l'objet sont cachés au monde extérieur et on ne peut y accéder qu'à l'aide des méthodes. Quand ceci est réalisé on parle de l'**encapsulation des données**.

Dans le vocabulaire de la programmation orientée objet, on dit que l'appel d'une méthode est l'envoi d'un **message à l'objet**.

3.1.3 Principe d'abstraction des données

Un autre grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les *spécifications* de ses méthodes ; la manière dont les données et les méthodes sont réellement implémentées étant sans importance. On dit que l'on a réalisé une **abstraction des données**.

L'encapsulation des données facilite grandement la maintenance : une modification éventuelle de la structure des données d'un objet ne sera pas aperçue par les utilisateurs.

Remarquons qu'en programmation structurée une fonction pouvait déjà être entièrement caractérisée par sa spécification ; par contre la structure des données devait être connue des utilisateurs.

3.1.4 Notion de classe

La notion de **classe** est la généralisation de la notion de type pour une donnée : c'est le type d'un objet. Bien entendu si les objets d'un même type ont des données différentes, par contre les méthodes sont communes.

Exemple.- Pour la manipulation des objets points définis précédents, on peut considérer la classe des points ayant les trois méthodes d'initialisation, de déplacement et d'affichage.

3.2 Mise en place des objets en C++

En programmation orientée objet pure, les données sont encapsulées et leur accès ne peut se faire que par le biais des méthodes. Le langage C++ autorise de n'encapsuler qu'une partie des données d'une classe ; on parle alors de **langage hybride**. Il existe même en C++ un constructeur de types particulier, généralisation du constructeur des types structurés du langage C, qui permet d'associer des données et des méthodes (alors que les méthodes n'existaient pas en C), mais sans aucune encapsulation. Nous commencerons par étudier ce constructeur de types. Nous verrons ensuite la mise en place des classes proprement dites.

3.2.1 Les structures en C++

Les structures du langage C++ constituent une généralisation des *structures* du langage C, la différence avec ces dernières étant que des fonctions peuvent être membres d'une telle structure. Les structures telles qu'elles sont définies en langage C existent toujours ; nous les appellerons **structures simples**.

3.2.1.1 Les structures simples

Imaginons que l'on veuille définir un type structuré pour les points du plan. On déclare le type :

```
struct point
       {
       float x,y ;
       } ;
```

Les champs, ici `x` et `y`, s'appellent des **membres** en C++.
On déclare des points, par exemple `A,B,C` de la façon suivante :

```
struct point A,B,C ;
```

Pour initialiser ces points, on écrira par exemple :

```
A.x = 2.34 ;
A.y = 5.1 ;
```

3.2.1.2 Les structures avec fonctions membre en ligne

Introduction.- La nouveauté en C++ est que l'on peut avoir comme champs des fonctions, que l'on appelle **fonctions membre**. Commençons par le cas le plus simple dans lequel les fonctions membre sont des fonctions en ligne.

Exemple.- Pour l'ensemble des points vu ci-dessus, on peut déclarer la structure `point` du programme ci-dessous :

```
// enligne.cpp
#include <iostream.h>
// declaration du type point
     struct point
     {
     // declaration des attributs
     float x, y ;
     // declaration des methodes
```

```
     void initialise(float abs, float ord)
          {
          x = abs ;
          y = ord ;
          }
     void deplace(float dx, float dy)
          {
          x = x + dx ;
          y = y + dy ;
          }
     void affiche(void)
          {
          cout << "Le point est en (" << x << "," << y <<")\n" ;
          }
     } ;
// Programme principal
     void main(void)
     {
     struct point A, B ;
     A.initialise(2.34, 5.1) ;
     A.deplace(1.03, 3.3) ;
     A.affiche() ;
     B.affiche() ;
     }
```

Membres attributs et fonctions membre.- Une telle structure comporte donc des membres classiques, ici `x` et `y`, appelés **membres attributs** et des *fonctions membre*. Les membres attributs s'utilisent de la façon habituelle, il n'y a donc rien de plus à en dire.

Seule la définition des fonctions membre en ligne est permise à l'intérieur d'une structure (ou d'une classe). Il est donc inutile de spécifier qu'une telle fonction est en ligne en la faisant précéder du mot réservé `inline`.

On remarquera que l'on peut utiliser, dans la définition des fonctions en ligne, les membres attributs (ainsi que les fonctions membre) de la structure, telles que `x` et `y` dans notre exemple, sans déclaration préalable.

Utilisation des méthodes.- Pour utiliser, par exemple, la méthode `initialise()` de la structure `point` pour la variable `A` de type `point` on écrit, de façon habituelle pour les structures, `A.initialise()` en utilisant l'opérateur point, comme on le voit sur l'exemple ci-dessus.

Remarque.- La dernière instruction de notre programme est là pour bien démontrer qu'il n'y a pas d'initialisation des données par défaut : les valeurs affichées sembleront aléatoires.

3.2.1.3 Les structures avec fonctions membre

Introduction.- Lorsqu'on veut qu'un membre d'une structure soit une fonction (qui n'est pas une fonction en ligne), seule la déclaration de la fonction membre apparaît dans la structure. La fonction elle-même est définie en dehors de la définition de la structure, en utilisant l'*opérateur de résolution de portée.*

Exemple.- Pour le problème vu ci-dessus, on peut déclarer la structure `point` du programme

ci-dessous :

```
// membre.cpp
#include <iostream.h>
// declaration du type point
     struct point
     {
     // declaration des attributs
     float x, y;
     // declaration des methodes
     void initialise(float, float);
     void deplace(float, float);
     void affiche(void);
     };
// definition des fonctions membre
     void point::initialise(float abs, float ord)
          {
          x = abs;
          y = ord;
          }
     void point::deplace(float dx, float dy)
          {
          x = x + dx;
          y = y + dy;
          }
     void point::affiche(void)
          {
          cout << "Le point est en (" << x << "," << y <<")\n";
          }
// Programme principal
     void main(void)
          {
          struct point A, B;
          A.initialise(2.34, 5.1);
          A.deplace(1.03, 3.3);
          A.affiche();
          B.affiche();
          }
```

Déclaration des fonctions membre.- Les fonctions membre se déclarent dans la structure, sans définition, uniquement sous forme de prototype.

Définition des fonctions membre.- La définition d'une fonction membre se fait en dehors de la définition de la structure, de façon classique. Mais encore faut-il indiquer à quelle structure elle se rapporte. Pour cela on utilise l'**opérateur de résolution de portée** indiqué par deux fois deux points '::'. Ainsi la fonction `initialise()` de la structure `point` sera-t-elle désignée, pour sa définition, par `point::initialise()`.

D'autre part on peut utiliser les membres attributs de la structure, tels que `x` et `y` dans notre exemple, sans indication supplémentaire (les indications de la résolution de portée sont suffisantes).

3.2.2 Membres publics et membres privés

Introduction.- Le concept de structure en C++ est amélioré de façon à permettre à certains membres d'être **privés**, c'est-à-dire qu'on ne peut pas y accéder de l'extérieur, ce qui met en place la notion d'encapsulation.

Mise en place.- Les membres privés sont précédés du mot clé `private :` alors que les membres publics sont précédés du mot clé `public :`. Par défaut les membres sont publics. Lorsqu'on a utilisé un de ces deux mots clés tous les membres qui suivent sont de la nature indiquée, d'où une suite de la forme suivante :

```
private :
- - - - -
public :
- - - - -
private :
- - - - -
```

Exemple 1.- Dans les exemples précédents aucun membre n'était privé, on pouvait donc accéder à tous les membres. On peut le vérifier grâce au programme suivant :

```
// acces_d.cpp
#include <iostream.h>
// declaration du type point
struct point
   {
   // declaration des attributs
   float x, y;
   // declaration des methodes
   void initialise(float,float);
   void deplace(float,float);
   void affiche(void);
   };
// definition des fonctions membre
void point::initialise(float abs, float ord)
   {
   x = abs;
   y = ord;
   }
void point::deplace(float dx, float dy)
   {
   x = x + dx;
   y = y + dy;
   }
void point::affiche(void)
   {
   cout << "Le point est en (" << x << "," << y <<")\n";
   }
// Programme principal
void main(void)
   {
   struct point A, B;
```

```
   A.x = 2.34; A.y = 5.1; // Acces direct aux attributs
   A.deplace(1.03, 3.3);
   A.affiche();
   B.affiche();
   }
```

dans lequel la fonction d'initialisation n'est pas utilisée : on l'émule en travaillant directement sur les coordonnées.

Exemple 2.- Ce n'est pas ce que l'on veut, on a donc intérêt à réécrire la structure de la façon suivante :

```
//prive.cpp
#include <iostream.h>
// declaration du type point
struct POINT
   {
   // declaration des attributs
   private:
   float x, y;
   // declaration des methodes
   public:
   void initialise(float, float);
   void deplace(float, float);
   void affiche(void);
   };
// definition des fonctions membre
void point::initialise(float abs, float ord)
   {
   x = abs;
   y = ord;
   }
void point::deplace(float dx, float dy)
   {
   x = x + dx;
   y = y + dy;
   }
void point::affiche(void)
   {
   cout << "Le point est en (" << x << "," << y <<")\n";
   }
// Programme principal
void main(void)
   {
   struct point A, B;
   A.initialise(2.34, 5.1);
   A.deplace(1.03, 3.3);
   A.affiche();
   B.affiche();
   }
```

Remarques.- 1^o) C'est par hasard seuls les attributs sont privés et donc que toutes les fonctions sont publiques. Une fonction peut également être privée, par exemple une fonction auxiliaire servant à définir une fonction qui, elle, sera appelée et donc déclarée publique. Un attribut a par contre toujoursz intérêt à être privé pour respecter la philosophie de la programmation orientée objet, mais rien ne l'y oblige en C++.

- 2^o) Les fonctions membre d'un classe (structure plus généralement en langage C++) ont accès à l'ensemble des membres, publics ou privés, de la classe.

Exemple 3.- (**Encapsulation**)

On peut vérifier que, dans le cas où on a déclaré certains membres comme étant privés, essayer d'accéder à ces membres privés ne sera pas accepté par le compilateur :

```
// incorrec.cpp
#include <iostream.h>
// declaration du type point
struct point
   {
   // declaration des attributs
   private:
   float x, y;
   // declaration des methodes
   public:
   void initialise(float, float);
   void deplace(float, float);
   void affiche(void);
   };
// definition des fonctions membres
void point::initialise(float abs, float ord)
   {
   x = abs;
   y = ord;
   }
void point::deplace(float dx, float dy)
   {
   x = x + dx;
   y = y + dy;
   }
void point::affiche(void)
   {
   cout << "Le point est en (" << x << "," << y <<")\n";
   }
// Programme principal
void main(void)
   {
   struct point A, B;
   A.x = 2.34; A.y = 5.1; //utilisation d'attributs prives
   A.deplace(1.03, 3.3);
   A.affiche();
   B.affiche();
   }
```

en effet celui-ci indique l'erreur suivante :

```
member 'x' is a private member of class 'point'
```

3.2.3 Les classes en C++

Introduction.- Une classe au sens de la programmation objet correspond en C++ à une structure avec fonctions membre et des qualificatifs privé et public pour permettre l'encapsulation. La notion de classe déclarée comme telle existe cependant en C++. On remplace `struct` par `class`. La différence entre ces deux types structurés est que, par défaut, les membres sont publics pour une structure alors qu'ils sont privés pour une classe.

Exemple.- On peut réécrire la déclaration du type `point` de la façon suivante, le reste du programme étant inchangé :

```
// class.cpp
#include <iostream.h>
// declaration du type point
class point
   {
   // declaration des attributs
   float x, y;
   // declaration des methodes
   public:
   void initialise(float, float);
   void deplace(float, float);
   void affiche(void);
   };
// definition des fonctions membre
void point::initialise(float abs, float ord)
   {
   x = abs;
   y = ord;
   }
void point::deplace(float dx, float dy)
   {
   x = x + dx;
   y = y + dy;
   }
void point::affiche(void)
   {
   cout << "Le point est en (" << x << "," << y <<")\n";
   }
// Programme principal
void main(void)
   {
   point A, B;
   A.initialise(2.34, 5.1);
   A.deplace(1.03, 3.3);
   A.affiche();
   B.affiche();
```

```
   }
```

Remarques.- 1^o) On pourra vérifier que l'on ne peut pas accéder directement, par exemple, à `A.x`.

- 2^o) Il est inutile, en C++, de déclarer `struct point A,B` ou `class point A,B`, la déclaration `point A,B` suffit.

3.2.4 Affectation globale

Introduction.- En langage C il est possible d'affecter à une variable structurée la valeur d'une expression structurée de même type. En C++, cette possibilité s'étend aux objets de même type. Elle correspond à une recopie des valeurs des membres donnée, que ceux-ci soient publics ou non ; les fonctions ne sont évidemment pas concernées.

Un exemple.- Considérons le programme suivant qui affecte un point à un autre.

```
// affect.cpp
#include <iostream.h>
// declaration de la classe point
class point
   {
   // declaration des attributs
   float x, y;
   // declaration des methodes
   public:
   void initialise(float, float);
   void deplace(float, float);
   void affiche();
   };
// definition des fonctions membres
void point::initialise(float abs, float ord)
   {
   x = abs;
   y = ord;
   }
void point::deplace(float dx, float dy)
   {
   x = x + dx;
   y = y + dy;
   }
void point::affiche()
   {
   cout << "Le point est en (" << x << "," << y <<")\n;
   }
// Programme principal
void main(void)
   {
   point A, B;
   A.initialise(2.34, 5.1);
   B = A;
```

```
B.affiche();
}
```

3.3 Cas des fonctions non unaires

Jusqu'à maintenant les méthodes associées à une classe d'objets étaient des fonctions unaires, c'est-à-dire qu'elles ne faisaient intervenir qu'un seul objet (de la classe en question, qui plus est) comme argument. Les membres données étaient utilisés tels quels sans indiquer à quel objet de la classe on faisait référence. Dans le cas des fonctions binaires (et au-delà) il va bien falloir faire la différence entre deux objets. Le cas où on fait intervenir des objets de classes différentes est encore plus complexe.

3.3.1 Lois de composition interne

Introduction.- Commençons par traiter le cas des lois de composition interne, c'est-à-dire d'opérations binaires (ou ternaires et ainsi de suite) sur une classe d'objets donnée.

Notation des opérateurs.- Le fait certainement le plus surprenant est qu'on n'utilise pas la *notation infixe*. Par exemple si `a`, `b` et `c` sont des objets d'une certaine classe munie de l'opération binaire `op` alors on n'utilise pas la notation infixe :

```
c = a op b
```

mais la notation :

```
c = a.op(b).
```

On attache donc l'opérateur à un des arguments, le premier, rompant ainsi la symétrie entre les deux opérandes `a` et `b`.

Remarquons que `c` n'a pas besoin d'être un objet ; il peut être d'un type non-objet. Par contre `a` est nécessairement un objet.

Mise en place en C++.- On attache l'opérateur au premier argument. Lors de la définition de la fonction, les données de cet argument ne seront pas préfixées par l'objet en question alors que ceci est indispensable pour les autres arguments.

Exemple.- Considérons la classe `vecteur` des vecteurs réels à deux coordonnées. On va définir les méthodes `initialise()` et `deplace()` comme dans le cas des points ainsi que l'opération d'addition de deux vecteurs (application de `vecteur`×`vecteur` dans `vecteur`, donc loi de composition interne proprement dite) et la fonction produit scalaire (application de `vecteur`×`vecteur` dans $\mathbb{R}$, ce qui n'est pas une loi de composition proprement dite).

```
// vecteur.cpp
#include <iostream.h>
// declaration de la classe vecteur
class vecteur
   {
   // declaration des attributs
   float x, y;
   // declaration des methodes
   public:
   void initialise(float, float);
```

```
   void affiche(void);
   vecteur add(vecteur b);
   float prod(vecteur b);
   };
// definition des fonctions membre
void vecteur::initialise(float abs, float ord)
   {
   x = abs;
   y = ord;
   }
void vecteur::affiche(void)
   {
   cout << "(" << x << "," << y <<")\n";
   }
vecteur vecteur::add(vecteur b)
   {
   vecteur c;
   c.x = x + b.x;
   c.y = y + b.y;
   return(c);
   }
float vecteur::prod(vecteur b)
   {
   float c;
   c = x*b.x + y*b.y;
   return(c);
   }
// Programme principal
void main(void)
   {
   vecteur u, v, w;
   float x;
   u.initialise(2.34, 5.1);
   v.initialise(1.03, 3.3);
   cout << "u = ";
   u.affiche();
   cout << "v = ";
   v.affiche();
   w = u.add(v);
   cout << "La somme de u et v est ";
   w.affiche();
   x = u.prod(v);
   cout << "Le produit scalaire de u et v est " << x << '\n';
   }
```

3.3.2 Lois de composition externe : les fonctions amies

Introduction.- Dans le cas d'une loi de composition externe dont les arguments sont d'un type classe (donc des objets), on a un problème pour les définir puisque les attributs de ces objets ne sont, en général, pas accessibles. Ce problème se régle, en langage C++, en déclarant certaines

fonctions comme étant **amies** de la classe : ceci permet, lors de la définition de ces fonctions, d'accéder aux attributs des objets de la classe dans laquelle on a déclaré ces fonctions comme étant amies.

Une telle fonction sera déclarée amie d'au moins deux classes puisqu'il y a au moins deux arguments.

Syntaxe.- On déclare la fonction dans chacune des classes desquelles on veut qu'elle soit amie en faisant précéder cette déclaration du mot clé `friend`.

Lors de la définition de la fonction on n'utilise pas l'opérateur de portée de résolution (il y aurait ambiguïté sur la classe à laquelle il faudrait faire référence).

Puisque la fonction portera sur au moins deux classes il faudra commencer par **déclarer** une première classe, sans la définir, puis définir la seconde (qui fera appel à la première pour le type d'un des arguments de la fonction amie) puis enfin (dans le cas d'une loi binaire) définir la première classe.

Puisqu'il n'y a aucune raison de privilégier une classe par rapport à une autre, aucun argument ne sera implicite dans la fonction. Autrement dit l'appel de cette fonction se fait de façon traditionnelle et non comme membre d'un objet.

Exemple.- Considérons le problème de la définition des classes des vecteurs à deux coordonnées réelles et des matrices 2×2 à coefficients réels comprenant en particulier, parmi les fonctions, le produit d'une matrice par un vecteur.

On peut utiliser les déclarations suivantes en C++ :

```
// matrice.cpp
#include <iostream.h>
// declaration de la classe matrice
class matrice;
// definition de la classe vecteur
class vecteur
   {
   // declaration des attributs
   float x, y;
   // declaration des methodes
   public:
   void initialise(float, float);
   void affiche(void);
   friend vecteur mpy(matrice M, vecteur V);
   };
// definition de la classe matrice
class matrice
   {
   // declaration des attributs
   float a11, a12, a21, a22;
   // declaration des methodes
   public:
   void initialise(float, float, float, float);
   void affiche(void);
   friend vecteur mpy(matrice M, vecteur V);
   };
// definition des fonctions membre
```

```
void vecteur::initialise(float abs, float ord)
   {
   x = abs;
   y = ord;
   }
void vecteur::affiche(void)
   {
   cout << "(" << x << "," << y <<")\n";
   }
vecteur mpy(matrice M, vecteur V)
   {
   vecteur W;
   W.x = M.a11*V.x + M.a12*V.y;
   W.y = M.a21*V.x + M.a22*V.y;
   return(W);
   }
void matrice::initialise(float a, float b, float c, float d)
   {
   a11 = a; a12 = b;
   a21 = c; a22 = d;
   }
void matrice::affiche(void)
   {
   cout << "|" << a11 << "," << a12 <<"|\n";
   cout << "|" << a21 << "," << a22 <<"|\n";
   }
// Programme principal
void main(void)
   {
   vecteur V, W;
   matrice M;
   V.initialise(1, 2);
   M.initialise(1, 2, 3, 4);
   cout << "V = ";
   V.affiche();
   cout << "M = ";
   M.affiche();
   W = mpy(M, V);
   cout << "Le produit de M par V est ";
   W.affiche();
   }
```

3.4 Constructeur et destructeur

Introduction.- Lorsqu'on déclare un objet d'une classe donnée (ou, plus exactement, une variable de type cette classe), cet objet n'est pas initialisé. Ceci est particulièrement gênant pour les **objets dynamiques**, c'est-à-dire ceux qui réclament une allocation dynamique de mémoire, tels que les listes chaînées. Cette initialisation peut se faire de façon automatique grâce à une fonction membre particulière (ou, plus exactement, de nom particulier) appelée **constructeur**.

De même lorsqu'on n'a plus besoin de cet objet dynamique, c'est-à-dire en fin de bloc, on a intérêt à libérer la place qui lui était réservée ; ceci peut également se faire de façon automatique grâce à une fonction membre (de nom particulier) appelée **destructeur**.

Nous allons étudier sur un exemple la philosophie générale du constructeur et du destructeur. Remarquons que l'on pourrait se contenter bien souvent d'un constructeur sans destructeur.

Noms du destructeur et du constructeur.- Bien évidemment les noms des fonctions constructeur et destructeur doivent être liés au nom de la classe pour savoir quelles fonctions il faut mettre en oeuvre de façon automatique. Pour une classe `CL` la fonction membre constructeur se nomme également `CL` et la fonction destructeur se nomme ~CL, en faisant précéder le nom de la classe par un tilde '~'.

Mise en place.- Tout objet `x` de la classe `CL` possédant un constructeur se déclare de la façon suivante :

```
CL x(10, 20.1) ;
```

si, par exemple, la fonction constructeur possède deux arguments de types respectifs entier et réel.

Le destructeur, quant à lui, s'il existe, est automatiquement appelé lorsqu'on sort du bloc dans lequel la variable a été déclarée. Normalement on ne peut pas l'appeler directement.

Un exemple.- Nous avons vu qu'il n'existe pas de type chaîne de caractères en langage C. Créons une classe `string` ayant, lors de la déclaration, comme argument le nombre de caractères possibles, le constructeur initialisant la chaîne par le mot vide. On doit pouvoir lire une chaîne de caractères au clavier, l'afficher et la détruire.

```
// mot.cpp
#include <iostream.h>
#include <stdio.h> // pour utiliser getchar()
// definition de la classe string
class string
   {
   // declaration des attributs
   char *s;
   int max;
   // declaration des methodes
   public:
   string(int n);
   void lire(void);
   void affiche(void);
   ~string(void);
   };
// definition des fonctions membre
string::string(int n)
   {
   s = new char[n+1];
   max = n;
   *s = '\0';
   }
void string::lire(void)
   {
```

```
   char *tab;
   int i;
   char c;
   tab = s;
   i = 0;
   c = getchar();
   while ((c != '\r') && (i < max))
      {
      i++;
      *tab = c;
      tab++;
      c = getchar();
      }
   *tab = '\0';
   }
void string::affiche(void)
   {
   char *tab;
   int i;
   tab = s;
   i = 0;
   while ((*tab != '\0') && (i < max))
      {
      cout << *tab;
      i++;
      tab++;
      }
   }
string::~string(void)
   {
   delete [] s;
   }
// Programme principal
void main(void)
   {
   string s(40);
   cout << "Essai pour voir un mot initial : ";
   s.affiche();
   cout << "\nEntrer un mot de moins de 40 caracteres : ";
   s.lire();
   cout << "\nCe mot est : ";
   s.affiche();
   s.~string();
   cout << "\nEntrer un nouveau mot : ";
   s.lire();
   s.affiche();
   }
```

Le nouveau mot entré n'est pas affiché puisque le pointeur sur ce mot a été détruit.

Remarque.- Un constructeur (ou un destructeur) n'a pas de type de retour spécifié.

Exercice.- *Compléter la définition de la classe ci-dessus en ajoutant des fonctions pour la copie d'une chaîne de caractères dans une autre (de longueur moindre que la longueur possible), la concaténation et la détermination de la longueur effective d'une chaîne de caractères.*

3.5 Exercices

Exercice 1.- Écrire une classe `vecteur` de l'espace comportant comme membres donnée privés trois coordonnées de type `float` et comme fonctions membre publiques :

+ *initialise* pour attribuer des valeurs aux coordonnées,
+ *add* pour additionner deux tels vecteurs,
+ *homothetie* pour multiplier les coordonnées par un réel fourni en argument,
+ *scal* pour obtenir le produit scalaire de deux vecteurs,
+ *vect* pour obtenir le produit vectoriel de deux vecteurs,
+ *affiche* pour afficher les coordonnées du vecteur sous la forme d'un triplet.

Exercice 2.- Écrire une classe `complex` pour manipuler des nombres complexes, comportant comme membres donnée privés deux coordonnées de type float (la partie réelle et la partie imaginaire) et comme fonctions membre publiques :

+ *initialise* pour attribuer des valeurs aux parties réelles et imaginaires,
+ *affiche* sous la forme cartésienne $a + i.b$,
+ *trigo* pour afficher sous la forme trigonométrique $\rho.exp(i.\theta)$,
+ *add* pour additionner deux complexes,
+ *sub* pour soustraire deux complexes,
+ *mult* pour multiplier un nombre complexe par un nombre réel,
+ *prod* pour multiplier deux nombres complexes,
+ *div* pour diviser deux complexes (le second étant non nul),
+ *module* pour calculer le module d'un nombre complexe,
+ *conj* pour calculer le conjugué d'un nombre complexe.