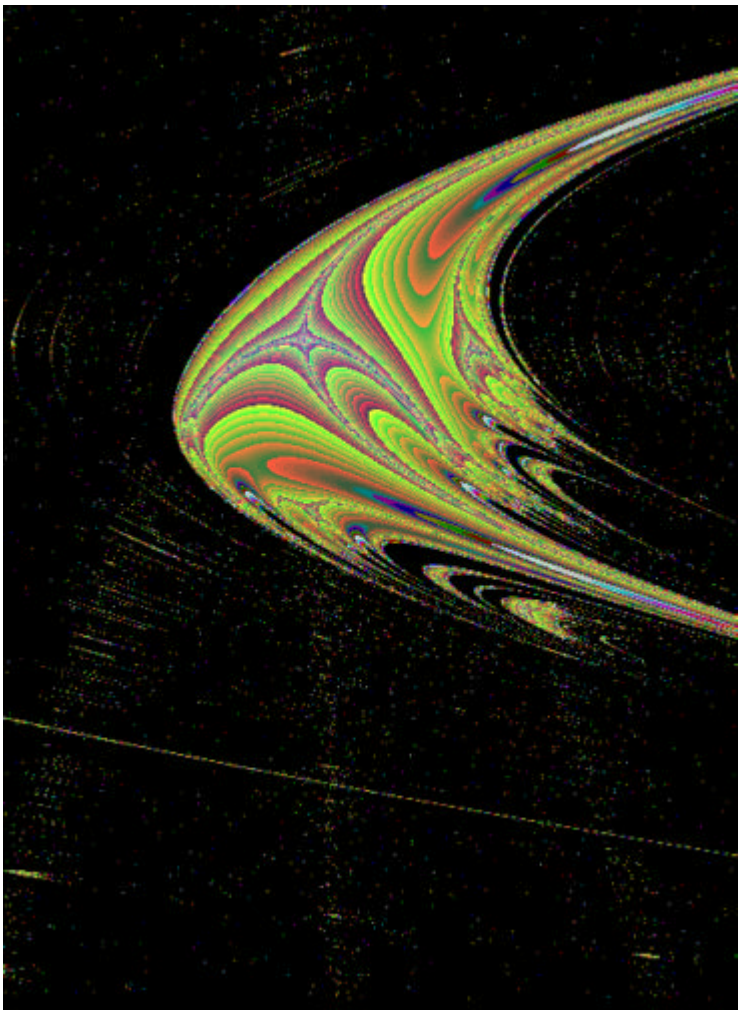
CHAPITRE 12

Classes



12.1 *Utilité de la construction "class"*

La classe C++ est la construction la plus importante de ce langage, relativement à C. Les classes sont typiquement utilisées pour introduire un certain degré d'abstraction, en définissant par exemple de nouveaux types, ou en ajoutant de nouvelles fonctionnalités à des types existant. La notion de classe peut remplacer la construction C struct, comme dans :

```
typedef struct cp_typ
     {
     float x, y;
     }    Point;

class     Point
     {
     float x, y;
     };
```

Dans les deux cas, il est possible de déclarer une variable de type Point comme suit :

```
Point          thisPoint;
```

Mais la classe recèle beaucoup plus de choses que la structure C et une déclaration de type. La classe possède (implicitement ou explicitement) 4 attributs associés :

1. Des **membres**. Le terme de membre indique simplement qu'il s'agit d'entités (code, données) faisant partie intégrante de la classe. Par rapport aux éléments x et y de la structure `struct` il n'y a pas de différence fondamentale avec les**membres** x et y de la classe `Point`. Le terme de membre indique simplement de manière plus précise l'idée d'appartenance. Chaque identificateur déclaré dans le cadre d'une classe est un **membre** de cette classe, ce qui signifie qu'il fait **partie intégrant**e de cette classe.

2. Zéro ou plusieurs **données membres** (*data members*). Ces membres constituent la représentation interne de la classe. Les données implémentent la classe.

2. Zéro ou plusieurs **fonctions membres** (*function members*). Celles-ci représentent l'ensemble des opérations qui peut être appliqué à un objet de cette classe. Elles représentent l'interface de la classe pour les utilisateurs. On appelle souvent ces fonctions membres **méthodes** (*methods*). Dans la jungle de la terminologie orientée objets, on parle aussi volontiers de **messages** (*message*).

3. Des niveaux d'accès programmatiques. Les membres d'une classe peuvent être définis comme **private**, **protected**, ou **public**. Ces niveaux contrôlent l'accès aux membres depuis le programme. Typiquement, les données membres sont déclarées `private`, et les fonctions membres `public`. Cette méthodologie est appelée dissimulation de l'implémentation, ou *information hiding*. Lorsque toute la représentation interne de la classe est déclarée `private`, on a réalisé une encapsulation. Nous reviendrons sur les membres de type `protected` plus

tard, dans le cadre de l'héritage.

4. Un nom de classe associé (*tag name*), servant de dénomination de type (*type specifier*) pour la classe définie par l'utilisateur. Un nom de classe défini par l'utilisateur peut apparaître partout où un identificateur de type prédéfini est autorisé.

12.2 *Un exemple : la classe Point*

Nous allons commencer par un court exemple destiné à illustrer pratiquement l'utilisation d'une classe. Il s'agit ici de définir le type **Point**, défini par ses coordonnées cartésiennes ou polaires.

12.2.1 *Définition et implémentation*

Une classe ayant une représentation interne de niveau `private` et un ensemble de fonctions membres de niveau public est par définition un type de données abstrait (*abstract data type*). Ainsi, la classe `Point`, dont le seul but est de définir le type de données `Point`, ou coordonnée dans un plan, pourrait se définir comme suit :

```
/************************************
File : Point.h
Author : /users/pro/mjn (Markus Jaton)
Date : 05.07.1995
Location : telec2.einev.ch
Mail : jaton@einev.ch
************************************/

// Inclusion de fichiers librairie standards

typedef  boolean       int;
#include    <iostream.h>

class   Point
     {
     private :
               // Ces données sont inaccessibles pour l'application.
          double  x, y;
     public :
          Point()           ;    // constructeur par defaut
                                 // Constructeur d'assignation,
                                 // ou d'affectation
          Point(double xx, double yy);
                                 // constructeur de copie
          Point(const Point& coord);
                                 // Destructeur (ne fait rien)
          ~Point() {;}
                                 // Coordonnées polaires
          Point(double module, long angleInMilliDegrees);
                                 // Opérateur de test d'égalité
          boolean operator==(const Point& cx) const;
                                 // Opérateur d'assignation
          void    operator=(const Point& cx);
                                 // Fonction permettant l'impression
                                 // sur un ostream.
          void    print(ostream &os);
     };
```

est un type de données abstrait. L'application peut l'utiliser sans connaître aucun des détails d'implémentation. Un passage à un système de coordonnées polaires peut s'effectuer sans que l'application ne soit touchée. Essayons d'analyser d'un peu plus près cette définition de classe très simple, pour parvenir à comprendre les règles de base de définition d'une classe.

La définition commence par le mot réservé **`private`**, suivi de la déclaration de deux valeurs de type double. Ceci est la représentation interne de la classe. On représente la classe Point par ses coordonnées cartésiennes, représentées par des nombres en virgule flottante de double précision. On pourrait également la représenter par son module et son argument: l'application n'a en principe pas à s'occuper de ceci. D'ailleurs, toute tentative de la part d'une application, d'adresser le membre x de la variable z, elle-même de type Point, par exemple, générerait de la part du compilateur l'erreur suivante :

```
myProg.C, line XXX : error : Cannot access z.x : private member //HPC++
```

Suit une partie précédée par le mot reservé **`public`**. C'est ici que commence l'interface de la classe pour les utilisateurs. En principe, les utilisateurs normaux de la classe ne voyent que cette partie. Cette partie liste les **opérations** qu'un utilisateur peut demander à un objet de la classe `Point`. En jargon OO, on dira que cette partie spécifie les messages auxquels peut répondre un objet de type `Point`.

La déclaration que nous avons donnée çi-dessus représente l'**interface** de la classe. Cet interface est (du point de vue de sa signification) équivalent à un module de définition en MODULA-2, ou à une spécification en ADA. Cet interface est habituellement stocké dans un fichier séparé, un *header file* (**fichier en-tête**). Le fichier en-tête ne contient pas de code[1], mais uniquement des définitions, ainsi que le veut le bon usage en C et en C++.

Le code est contenu dans un fichier séparé, le **fichier implémentation** (*implementation file*). Cette convention correspond exactement à la convention en langage C traditionnel, pour définir les librairies externes. La définition de classes au moyen de C++ par l'intermédiaire de fichiers en-tête est néanmoins plus puissante que l'utilisation habituelle de fichiers en-tête en C, parceque les concepts utilisés par le langage permettent une plus grande abstraction que la simple définition de prototypes de fonction.

Concentrons-nous maintenant sur l'implémentation de cette classe Point. Nous allons tout d'abord la donner telle quelle, sans explications préalables de la syntaxe, et nous reviendrons ensuite progressivement sur les divers éléments de cette implémentation.

```
/*************************************
File : Point.C
Author : /users/pro/mjn (Markus Jaton)
Date : 05.07.1995
Location : telec2.einev.ch
```

1. En réalité, le fichier en-tête peut contenir du code, comme nous le verrons par la suite. Ce code est alors appelé **«en-ligne implicite»** (*implicit inline code*). Il est fortement déconseillé de publier le code de cette façon, sauf pour certaines fonctions particulièrement triviales. Il est préférable de dissimuler toujours l'implémentation, au besoin en la déclarant comme «`inline`» de manière explicite dans le fichier implémentation.

```
Mail : jaton@einev.ch
************************************/
#include <math.h>
#include «Point.h»
const    double   pi = 3.1415926;
const    double   RadToDeg = 360 / 2 * pi;
Point::Point()

     {
     // Constructeur par défaut
     x = y = 0;
     }

Point::Point(double xx, double yy)

     {
     // Constructeur normal
     x = xx; y = yy;
     }

Point::Point(const Point& coord)

     {
     // Constructeur de copie
     x = coord.x; y = coord.y;
     }

Point::Point(double module, long angleInMilliDegrees)

     {
     // Constructeur par coordonnées polaires
     x = module * cos(angleInMilliDegrees * RadToDeg);
     y = module * sin(angleInMilliDegrees * RadToDeg);
     }

boolean Point::operator==(const Point& cx)

     {
     // Test d'égalité des deux coordonnées :
     return   ((x == cx.x) && (y == cx.y));
     }

Point&    Point::operator=(const Point& cx)

     {
                                 // Opérateur d'assignation
     x = cx.x;
     y = cx.y;
     return *this;
     }

void    Point::print(ostream &os)

     {
               // Fonction permettant l'impression sur un ostream.
     os<<x<<",”<<y<<endl;
     }
```

Notons la syntaxe particulière utilisée pour dénoter un membre : la méthode print devient, dans le fichier implémentation **`void Point::print()`**. On exprime par là qu'il s'agit d'une fonction -d'une méthode- propre à Point, donc appartenant au contexte de Point. On pourrait, dans le même fichier, définir une autre fonction print() dans un contexte différent sans qu'il n'y ait conflit.

12.2.2 *Instanciation d'un objet de la classe Point*

Cette définition et l'implémentation ayant été dûment enregistrée, comment peut-on utiliser la classe et le code associés? La classe Point n'est en fait pas grand'chose d'autre qu'une déclaration de type. Pour instancier une variable d'un type donné, en C comme en C++, il suffit d'écrire

```
<TYPE_DONNE> uneVariable;
```

Ainsi, pour une variable de type int, on écrira :

```
int       uneVariableDeTypeInt;
```

De manière semblable, on peut définir une variable de type Point par :

```
Point     uneVariableDeTypePoint;
```

En C++, on utilisera volontiers les expressions orientées objets; ainsi on dira de la ligne çi-dessus qu'il s'agit de l'**instanciation** d'un **objet** de type `Point`. `uneVariableDeTypePoint` est ainsi une instance de la classe `Point`.

12.2.3 *Membres privés*

La classe Point définit deux membres privés : x et y. En principe, l'application n'a pas à connaître l'existence de ces deux membres, puisqu'elle n'y a pas accès. Alors, pourquoi publier leur existence dans le fichier Point.h ? C'est une question justifiée; du point de vue théorique, il est effectivement peu judicieux de faire connaître l'existence de x et de y à une application qui ne peut de toutes façons pas y accéder. D'un point de vue pratique, on est ici limité par le choix qui a été fait - pour des raisons de compatibilité avec C - d'utiliser le mécanisme d'inclusion de fichiers (`#include`) du préprocesseur pour définir une classe. Pour calculer la taille de la classe en mémoire, le compilateur a besoin de la définition **complète** (membres privés inclus) de la classe. Soit on lui fournit cette définition au moyen de fichiers de spécification précompilés (qui ne peuvent alors plus être importés simplement par #include !), soit on définit la classe au niveau source, ce qui permettra au compilateur de la traiter comme une `struct` normale. C++ a choisi la deuxième possibilité, évitant ainsi la définition de mots réservés supplémentaires (du style `import`), au détriment d'une plus grande consistance du langage.

Notons que d'une manière tout à fait générale, les membres privés peuvent être constitués de code, de données, ou d'un mélange arbitraire des deux.

Les membres de la classe ont quant à eux un accès tout à fait libre aux membres privés.

12.2.4 *Membres publics*

Dans la classe Point, tous les autres membres sont déclarés de type `public`. Ceci signifie qu'ils sont tous accessibles depuis l'extérieur de la classe, donc qu'ils sont accessibles depuis une application, ou depuis une autre classe, par exemple. Ainsi, pour accéder au membre print() d'une instance de type Point, on pourrait utiliser le segment de code suivant :

```
int  main(int, char**)
     {
     Point cc;
     cc.print();   // Appel du membre public print() de l'instance cc
     }
```

Il n'y a pas de différence syntactique entre l'accès à un membre public d'une classe, et l'accès à un élément d'une `struct`.

12.2.5 *Constructeurs*

La classe Point définit trois constructeurs. Un constructeur sert à initialiser correctement la classe, ou plus exactement toute **instance** de la classe. Un constructeur permet de garantir, par l'utilisation de code judicieux, l'état initial d'une instance de la classe considérée. C++ permet de définir un nombre arbitraire de constructeurs pour une classe donnée, pour autant que leur signature diffère. Ainsi, compte tenu de la définition et de l'implémentation de la classe Point çi-dessus, un programme pourrait effectuer les déclarations suivantes :

```
int  main(int, char**)

     {
     Point cx;          // Appel du constructeur par défaut
     Point cy(1.0, 4.23);        // Constructeur normal
                                 // (initialisation)
     Point cz(cy);      // Appel du constructeur par copie.
     }
```

L'appel du constructeur par défaut (`Point cx;`) aura pour effet la génération d'une instance de la classe Point appelée cx, dont les valeurs des membres privés x et y valent 0. Ces valeurs sont assignées par le constructeur par défaut.

L'appel du constructeur "normal" (`Point cy(1.0, 4.23);` parfois appelé constructeur d'assignation par certains auteurs, appelation tout aussi abusive que le "normal" utilisé ici) a pour effet d'imposer `cy.x = 1.0` et `cy.y = 4.23`.

L'appel du constructeur de copie (`Point cz(cy);`) a pour effet de faire de cz une copie de cy. Ainsi, on aura `cz.x = 1.0` et `cz.y = 4.23.`

Le constructeur de coordonnées polaires permet de définir une coordonnée à l'aide de son module et de son argument. L'implémentation montre que le constructeur va convertir les

paramètres passés en coordonnées cartésiennes, de manière à garantir l'uniformité du comportement de la classe, quelle que soit la manière dont l'instanciation est faite (ou, si vous préférez, le constructeur utilisé pour générer une instance de la classe Point).

12.2.6 *Destructeur*

La classe Point définit un destructeur qui ne fait rien de particulier (destructeur composé d'une instruction vide). D'une manière plus générale, un destructeur fait pendant au constructeur: un destructeur est censé défaire tout ce que le constructeur -voire d'autres méthodes de la classe, au cours de la durée de vie d'une instance de cete classe- a fait. Il doit permettre, lors de la destruction d'une instance de la classe Point, de s'assurer que toutes les ressources que l'instance considérée aurait pu occuper au cours de sa durée de vie seront libérées. Dans notre cas, aucune ressource particulière n'est occupée, et nous pouvons de ce fait nous contenter d'une implémentation minimaliste de destructeur.

12.2.7 *Méthodes*

Les méthodes de la classe Point sont portion congrue, puisqu'elles se réduisent au nombre de une (sans compter les méthodes particulières que sont les opérateurs). Seule la méthode print() a été définie ici, qui imprime la valeur des deux membres privés. Notons que cette méthode peut accéder aux membres privés, mais uniquement parcequ'elle est un membre de la classe. Ainsi, le code suivant :

```
Point          cx(1.0, 2.3);
cx.print();
...
```

a pour effet :

```
1.0, 2.3
```

Notons que la méthode print() ne prend aucun paramètre. Beaucoup d'adeptes de la programmation en PASCAL (ou autre langage structuré traditionnel) s'en étonnent: ils voudraient pouvoir écrire quelque chose comme `print(cx)`. Ce serait effectivement indispensable si **cx n'était pas une instance de Point**. cx étant une instance, **c'est la méthode print() de cx qui va être appelée**. print() appartient à l'objet cx au même titre que les membres privés x ou y; l'indication d'appartenance à cx est implicitement contenu dans la notation cx.print().

12.2.8 *Opérateurs*

Notre classe Point définit de plus deux opérateurs: le test d'égalité et l'assignation.Curieusement, semble-t-il, ces deux opérateurs ne prennent qu'un seul argument, bien qu'en toute logique, l'opérateur ==, par exemple, doive s'effectuer sur deux opérandes ! Il n'y a pourtant pas d'erreur, et la raison de cet apparent manque d'opérande est similaire à l'apparent manque de paramètre discuté au paragraphe12.2.7, page175.

La définition de l'opérateur d'assignation (=) est la suivante :

```
Point&    Point::operator=(const Point& cx);
```

L'opérande donné comme paramètre est toujours **l'opérande de droite** dans le cas d'un opérateur membre binaire. L'opérande de gauche est -implicitement- l'instance de la classe à laquelle l'opérateur est appliqué. Ainsi, dans le segment de code suivant :

```
int  main(int, char**)

     {
     Point cx(2.4, 3.1);
     Point cy;
     cy = cx;
     }
```

L'expression `cy = cx` signifie "appliquer l'opérateur = de l'instance cy avec pour paramètre cx". L'opérande de gauche est donc cy, l'opérande de droite cx. On a bien deux opérandes comme habituellement, mais le fait que l'opérateur soit membre d'une classe implique que le membre de gauche est passé implicitement. Une autre manière, parfaitement correcte, d'invoquer l'opérateur "=" de cy serait

```
cy.operator=(cx);
```

Il en va de même pour les opérateurs unaires, sauf que le seul opérande qu'ils traitent est passé par défaut.

12.3 *Constructeurs et destructeurs*

Lors de l'examen de notre petite classe Point, nous avons abordé les constructeurs (qui servent à créer une instance d'une classe) et le destructeur servant à détruire une instance. Bien que l'on puisse définir plusieurs constructeurs, on ne peut fort logiquement pas appeler explicitement un constructeur pour un membre déjà crée (se qui équivaudrait à construire quelque chose de déjà construit!). Par contre, on ne peut définir qu'un seul destructeur.

12.3.1 *Initialisation par le constructeur*

Lors de l'instanciation d'une classe définie par l'utilisateur, le compilateur effectue automatiquement la réservation de mémoire nécessaire aux membres de la classe considérée. Cette réservation se fera au moyen des fonctions standard malloc, calloc, etc... Lorsque cette place mémoire est réservée, cette zone mémoire devra être initialisée. C++ permet à l'utilisateur de définir pour ce faire un constructeur, qui est appelé lors de l'instanciation, après que l'espace pour les membres ait été reservé avec succès. L'initialisation effective des valeurs dans la zone mémoire réservée se fait soit au moyen d'**instructions explicites d'assignation** aux membres privés, soit au moyen de la **liste d'initialisation** que l'on peut fournir avec le constructeur, soit encore au moyen d'un mélange des deux techniques :

```
class     A
     {
     private :
          int x, y;
     public :
          A(int b1, int b2) : x(b1), y(b2) {}
               // Utilisation de la liste d'initialisation
     };

class     A
     {
     private :
          int x, y;
     public :
          A(int b1, int b2) { x = b1; y = b2; }
               // Assignation explicite
     };

class     A
     {
     private :
          int x, y;
     public :
          A(int b1, int b2) : x(b1) { y = b2; }
               // Initialisation + Assignation explicite
     };
```

Quelle différence entre les deux techniques ? Pratiquement, et pour les exemples très simples qui nous préoccupent maintenant, aucune. En fait, la différence fondamentale réside dans l'ordre d'exécution : la liste d'initialisation est exécutée **avant** l'instanciantion d'une ob-

jet de la classe A, alors que le code d'initialisation ne peut s'effectuer qu'après instanciation de l'objet. En d'autres termes, dans le premier cas de figure, A est généré avec des valeurs de x = b1 et y = b2; dans le deuxième cas, A est généré avec des valeurs x et y quelconques, et ensuite ces valeurs quelconques sont surécrites par le code utilisateur.

La liste d'initialisation devrait, en règle générale, être préférée pour initialiser des valeurs (logiquement). En revanche, elle ne peut pas être utilisée pour exécuter du code devant être associé à la création d'un objet (comme par exemple une ouverture d'un fichier). Dans ce dernier cas, il est obligatoire d'utiliser le code du constructeur.

Il est permis de définir plusieurs constructeurs différents (comme nous l'avons fait pour notre classe Point): le compilateur choisira le bon en fonction de la signature du constructeur et de la manière d'instancier la classe :

```
Point         z; // Point::Point() utilisé

Point         z1(0.5, 0.2); //Point::Point(double re, double im)
Point         z2(z);   //Point::Point(const Point& cmplx)
```

Le constructeur est une méthode particulière qui permet d'initialiser une instance de la classe considerée. En C, la définition d'un type a l'aide de la construction typedef n'a pas d'effet direct. Ainsi, definir

```
typedef struct { double x, y; }Point;
```

introduit un synonyme: le type Point est défini comme équivalent à une structure sur deux variables en virgule flottante en double precision. Si on essaie d'utiliser une variable de type Point, comme par exemple dans :

```
Point         z;
```

les variables z.x et z.y n'ont pas de valeur définie. L'utilisation de leurs valeurs comme dans l'exemple suivant :

```
double module = sqrt(z.x * z.x + z.y * z.y);
```

conduit a un resultat indéfini, du point de vue de la norme ANSI. C'est le programmeur qui utilise le type Point qui est responsable d'initialiser convenablement les variables qu'il définit. L'inconvénient de cette manière de faire est évident : il implique que ledit programmeur connaisse parfaitement les détails d'implémentation du type qu'éventuellement un autre a défini. Utiliser une implémentation d'un système de coordonnées en mode polaire implique des changements majeurs chez les utilisateurs de ce type.

Le constructeur permet d'atteindre effectivement l'abstraction souhaitée. Le constructeur définit comment doit etre initialisée toute nouvelle instance du type (de la classe) considéré. Dans notre exemple de type Point, nous avons défini plusieurs constructeurs dans la partie publique de la déclaration de la classe. On peut en définir en principe autant que l'on veut. Géneralement, on définira le constructeur par défaut (*default constructor*), le construc-

teur de copie (*copy constructor*), et le constructeur d'assignation (*assignement constructor*). A l'aide de ces trois constructeurs, on peut dès lors effectuer les déclarations suivantes :

```
Point          z;                  // Constructeur par defaut
Point          zz(1.0, 3.7);       // Constructeur d'assignation.
                                   // zz.re = 1.0, zz.im = 3.7
Point          z2(zz);             // Constructeur de copie.
                                   // z2.re = zz.re, z2.im = zz.im
```

L'utilisateur du type abstrait Point n'a plus à s'inquiéter de la manière dont est implementé ce type: en fait, il n'en a même plus la possibilité. On a substitué a cela la notion beaucoup plus génerale de définir la manière par laquelle on peut définir une coordonnée ayant des particularités données.

12.3.2 *Constructeurs générés par défaut*

Il est parfaitement légal de définir une classe sans constructeur: le compilateur introduira, en l'absence de constructeur, un constructeur par défaut qui se comportera grosso modo comme un appel au bon vieux malloc de la librairie C, et un constructeur de copie qui effectuera une simple copie membre à membre. Implicitement, les deux constructeurs suivants sont définis:

```
Point::Point(const Point&); // Constructeur de copie
Point::Point();             // Constructeur par défaut
```

Le constructeur de copie est toujours généré, à moins que l'utilisateur en définisse un lui-même. Le constructeur par défaut n'est généré que si aucun constructeur n'a été défini par le programmeur. Ne définir aucun constructeur est parfaitement légal, mais devrait être considéré comme une habitude dangereuse. Il est préférable de définir explicitement des constructeurs plutôt que de laisser le compilateur en définir pour soi.

Dans notre classe Point, les implémentations par défaut seraient suffisantes. Le constructeur par défaut ne fait que réserver de la place mémoire alors que le constructeur par copie, en plus de réserver de la place, copie les membres un à un d'une instance source vers l'instance nouvellement crée. Examinons par contre l'exemple suivant :

```
// Fichier  SimpleString.h

class     SimpleString
     {
     private :
          char *stringContent;
          int  maxLength;
     public :
          SimpleString(char *content, int maximumAllowableLength);
          SimpleString(char *content);


               // autres méthodes permettant de travailler
```

```
               // avec des SimpleString

     };

// Fichier SimpleString.C

#include “SimpleString.h”
#include <string.h>

SimpleString::SimpleString(char *content, int maxAllowableLength)
     {
     stringContent = new char [maximumAllowableLength];
     maxLength = maximumAllowableLength;
     if (content == NULL) return;
     if (maxLength > strlen(content) strcpy(stringContent, content);
     else strncpy(stringContent, content, maximumAllowableLength);
     }

SimpleString::SimpleString(char *content)
     {
     maxLength = strlen(content)
     stringContent = new char [maxLength + 1];
     strcpy(stringContent, content);
     }
```

Cette classe ne définit que deux constructeurs, qui tous deux réservent une certaine place pour des chaînes de caractères, et copient une chaîne donnée comme initialisation dans la place ainsi réservée. Le constructeur par défaut ne sera donc en aucun cas généré par le compilateur. Par contre, le constructeur de copie va être généré, avec des effets peu souhaitables. Considérons le code suivant :

```
...
SimpleString  s1(“TOTO”, 256);   // Constructeur défini explicitement
SimpleString  s2(s1);            // Appel du constructeur de copie défini
                                 // implicitement.
....
```

`s2.stringContent` est un pointeur dont la valeur est égale à `s1.stringContent`, et `s2.maxLength` vaut 256. **Ceci ne correspond pas à ce que l’on désirerait** ! s1 et s2 devraient bien sûr logiquement avoir des contenus identiques, mais être différents; or ici, toute modification du contenu de s1 entraîne une modification identique dans s2. Il y a pire : si au cours du programme s1.stringContent vient à être libéré par un appel à `delete []`, s2.stringContent sera détruit également, sans que rien ne vienne en informer s2 ! Tôt ou tard, cette situation conduira à une erreur fatale.

Dans ce cas, il est indispensable de redéfinir le constructeur copie, de manière à s'assurer que le compilateur n'utilisera pas celui qu'il aurait autrement généré automatiquement. Un exemple de définition du constructeur de copie pour la classe SimpleString serait :

```
SimpleString::SimpleString(const SimpleString& str)

     {
     stringContent = new char [str.maxLength + 1];
```

```
     strcpy(stringContent, str.stringContent);
     }
```

12.3.3 *Le constructeur de copie (quelques règles)*

On peut également empêcher un utilisateur de la classe d'utiliser le constructeur de copie pour une classe donnée, en définissant un constructeur copie vide comme membre privé ou protégé:

```
class     NoCopyConstructor
     {
     private :
          .....
          NoCopyConstructor(const NoCopyConstructor&) {;}
     public :
          .....
     };
```

Ainsi, le compilateur ne va pas redéfinir le constructeur de copie, puisqu'il y en a un de défini, mais les utilisateurs ne peuvent pas non plus l'utiliser, puisqu'il s'agit d'un membre privé. Cette technique est utile lorsqu'on ne désire pas implémenter ce constructeur (pour des raisons techniques, généralement, liées à des fichiers et à des organes physiques), et qu'on veut également s'assurer qu'aucun utilisateur n'écrira

```
NoCopyConstructor ncc;
NoCopyConstructor ncc2(ncc);// Error : private member
```

Néanmoins, bloquer l'utilisation du constructeur de copie est une technique qui devrait être utilisée avec beaucoup de parcimonie, car elle produit des effets très gênants. Ainsi, le code suivant ne peut plus se compiler pour la classe çi-dessus :

```
void      aFunction(NoCopyConstructor x)
     {
     }
```

Passer un paramêtre par valeur (par opposition au passage par référence ou au passage d'un pointeur sur le paramètre) requiert que le programme crée une copie de l'instance dans la pile. Comme il n'existe pas de constructeur par copie accessible (puisqu'on l'a rendu privé), le compilateur ne peut générer de copie, et émettra logiquement le message d'erreur correspondant (`Cannot access NoCopyConstructor(const NoCopyConstructor&) : private member`).

Quand faut-il impérativement redéfinir le constructeur copie? En règle générale, toute classe qui requiert un destructeur nécessite également la définition d'un constructeur copie. En particulier :

- Lorsque l'instance de la classe considérée peut, au cours du temps, réserver de la mémoire en créant de nouveaux objets.
- Lorsque l'instance de la classe ouvre des fichiers.
- Lorsque la classe manipule des ressources système (pipes, sémaphores, queues de messages, mémoire partagée, etc...)

- Lorsque la classe s'interface avec des protocoles de communication (modems, protocoles OSI, réseau TCP-IP, interfaces MIDI, interfaces de bus de mesure GPIB, bus de terrain, etc...)

  La décision de savoir quels constructeurs vont être définis pour une classe donnée est importante, et doit faire l'objet d'un soin particulier.

12.3.4 *Le constructeur et son rôle de conversion de type*

Le constructeur, au delà de son rôle lors de l'initialisation d'une classe, permet de définir des conversions de type implicites, comme dans l'exemple suivant :

```
classA
     {
     private :
          int a;
     public :
          A(int x) : a(x) {}
          ...
     };
void f(A anArgument)

     {
     ...// some user code
     }
int  main(char*, int)

     {
     int  k = 10;
     f(k);          // Ok. Conversion A -> int.
     }
```

Que se passe-t-il en réalité ? Le compilateur ne trouvant pas de signature correspondant à f(int), va chercher un constructeur de A prenant un int comme argument. Si il le trouve (ce qui est le cas ici), il remplacera le int dans l'appel de f par une instance de la classe A générée au moyen de k, avec le constructeur de A prenant un entier comme paramètre. Ce qui réalise une conversion implicite d'un entier en une classe A.

Du fait de cette possibilité de conversion implicite, le code suivant se compilera correctement :

```
extern    void      aFunction(const SimpleString s&);

aFunction("Hello World");
```

Ce qui se passe en réalité, c'est que le compilateur a detecté, dans la définition de notre classe SimpleString, la présence du constructeur

```
SimpleString::SimpleString(char *content);
```

Ce constructeur va lui permettre de définir une conversion implicite entre un char* et un SimpleString. Lors de la recherche d'une signature correspondante, le compilateur va donc automatiquement essayer cette nouvelle conversion implicite, et va de ce fait obtenir une correspondance. Lors de l'appel, un SimpleString temporaire sera généré à partir du char*, et celui-ci sera passé correctement à la fonction aFunction.

Il est également possible de définir des conversions implicites à l'aide de l'opérateur conversion (). Les conversions implicites constituent un mécanisme très performant et très puissant pour alléger l'écriture d'un programme, et nous reviendrons en détail sur ces conversions lors de la discussion détaillée des opérateurs d'une classe.

12.3.5 *Destructeurs*

Le destructeur fait pendant au constructeur, en ce sens qu'il permet de détruire proprement l'instance de l'objet crée. Reprenons l'exemple de SimpleString çi-dessus :

Le constructeur de SimpleString alloue de la memoire dynamique a un membre privé. Cette allocation a lieu lors de l'exécution du programme, hors de contrôle du compilateur. Lorsque la mémoire occupée par SimpleString doit etre liberée, lors de la fin de la vie d'une instance de SimpleString, il n'existe pas d'automatisme qui pourrait également libérer la memoire reservée pour le membre stringContent. Dans l'implémentation actuelle, ce n'est pas le cas, et la fin de vie d'un objet de type SimpleString va également laisser une zone mémoire

Il est de ce fait nécessaire de définir, par correspondance au constructeur, un destructeur, qui sera appelé en fin de vie de l'instance, et qui se chargera de libérer proprement les ressources qui ont pu être allouées pour les membres de la classe lors de la durée de vie de l'objet. Ces ressources ne consistent pas uniquement en mémoire dynamique reservée par des membres privés, comme dans l'exemple SimpleString çi-dessus. Il peut également s'agir de fichiers ouverts, de boîtes de dialogue à refermer, voire de sessions de protocole de communication à terminer, etc... Dans notre cas, on implémentera le destructeur de SimpleString à peu près comme suit :

```
SimpleString::~SimpleString() { delete [] stringContent; }
```

En l'absence de destructeur déclaré explicitement, le compilateur se charge de définir un destructeur par défaut. Ce destructeur se contente de libérer au besoin la mémoire occupée par les membres donnée de la classe. Il va de soi que toute allocation mémoire effectuée par du code utilisateur au cours de la durée de vie de l'instance d'une classe(en particulier par le code associé au constructeur) n'est pas liberée implicitement, mais doit être liberée par le code associé au destructeur. Il en va de même pour toutes les ressources utilisées par la classe dont le compilateur ne peut pas contrôler l'usage (ressources externes au langage, réservations en cours d'exécution).

D'une manière générale, toute classe utilisant à un moment ou l'autre de la vie d'une

de ses instances des ressources externes (mémoire, fichiers, périphériques, communications) doit **impérativement** définir un destructeur.

12.4 *Membres*

Les membres d'une classe sont les éléments constitutifs de la classe. Ces éléments constitutifs peuvent être des **données**, des **opérateurs**, ou des **méthodes**. Lors de l'instanciation d'une classe, les membres données sont générés (à l'exception de certains membres particuliers, dits statiques, que nous aborderons plus loin) par le biais d'un appel au constructeur de la classe. A la fin de la durée de vie d'une instance, ces membres sont détruits (la place qu'ils occupaient est libérée) par le biais d'un appel au destructeur de cette classe.

Les membres peuvent être spécifiés avec des accès programmatiques, comme `private`, ou `public`.

```
class     C
     {
     private :
          int x, y;// membre donnée privé
          void aFunction(int);// méthode privée
     public :
          C(int xx, int yy) : x(xx), y(yy) {} // Constructeur
          ~C();                          // Destructeur
          boolean operator==(const C&);// Opérateur égalité
          void setX(int xx); // Méthode
          void setY(int yy); // Méthode
          int  getX() const; // Méthode const
          int  getY() const; // Méthode const
          void someFunction();
     };
```

L'accès à un membre se fait au moyen de la syntaxe '.' (ou '->' dans le cas de pointeurs sur des membres), comme pour un élément d'une `struct` du langage C habituel :

```
C    c1(10, 11);
const     C c2(1, 2);
C    *c3;
c3 = new C(5, 7);
int k = c3->getX();
c1.setX(12);            // Ok, c1 n'est pas constant
int xx = c2.getX();     // Ok, getX() est un membre const
c2.setX(4);             // Erreur, setX() n'est pas un membre const
```

L'utilisation de méthodes associées à un type est fréquemment source de confusions pour les novices de la programmation orientée objet. Ils tendent à confondre une fonction normale avec une fonction membre (donc, une méthode), et ne comprennent pas la manière dont les paramètres sont passés. Ainsi, un novice tendra à définir une méthode C::getX() plutôt comme C::getX(const C& cx). Ceci ne constitue pas une erreur au sens du compilateur, mais au niveau de la compréhension de ce qu'est une méthode. Une méthode est une fonction qui

ne peut s'appliquer qu'à une instance de la classe pour laquelle la méthode est définie: il est donc inutile de lui passer une instance de manière explicite. A l'intérieur de la méthode, on peut accéder aux membres x et y (ainsi qu'aux autres, privés ou non) directement, sans avoir besoin de spécifier de quelle instance on veut parler, pour autant que ce soit la même instance que celle pour laquelle on a invoqué la méthode. Ainsi la méthode getX() çi-dessus peut s'implémenter de la manière suivante :

```
int  C::getX() const
     {
     return x;// Accès au membre de l'instance
               // pour laquelle getX() a été invoqué
     }
```

En pratique, il est parfois nécessaire de disposer explicitement d'un pointeur ou d'une référence à l'instance actuelle, et ceci à l'intérieur d'une méthode. Un cas typique est l'appel d'une fonction non membre requérant une instance du type consideré comme paramètre depuis une méthode. Il existe pour ceci le mot reservé `this`, qui est un pointeur sur l'instance actuelle. Conceptuellement, this, dans le cas de la classe C définie plus haut, pourrait se définir comme `const C* this,` ou plus généralement, pour une classe de type TYPE, `const TYPE* this`;

```
extern    void aFunction(C& itsArgument);

void C::someFunction()
     {
     aFunction(*this);
     }
```

Un membre peut être dénoté comme constant, (`intC::getX() const`) ce qui, comme nous l'avons vu précedemment ne signifie pas que ce membre est effectivement constant, mais que l'instance à laquelle il appartient **ne sera pas modifiée**. En particulier, on peut invoquer cette méthode pour un objet que l'on a lui-même spécifié comme `const`. En revanche, appeler une méthode non spécifiée comme `const` pour une instance déclarée comme `const` provoquera une erreur à la compilation.

12.4.1 *Membres privés*

Les membres privés constituent l'implémentation de la classe : en ce sens, ils ne doivent pas être accessibles a l'utilisateur (pour ne pas rendre ce dernier dépendant de ladite implémentation, justement). La clause private rend ces membres inaccessibles, par les moyens légaux, a une application.

Comme on l'a vu plus haut, de par l'implémentation du langage, il est difficile de cacher effectivement l'implémentation a un utilisateur. Un utilisateur malveillant pourrait, fort de cette connaissance, essayer de violer la règle de non-visibilité des membres privés au moyen de pointeurs:

```
Point         z(1.0, 2.3);
```

```
double        *badPointer = (double*) &z;
cout<<"Abcisse "<<*z<<" Ordonnée "<<*(++z)<<endl;
```

Cet exemple ne figure ici que pour vous montrer ce qu'il ne faut faire en aucun cas! En effet, l'accès aux membres privés viole tout d'abord les sécurites qu'avait placées ici l'implémentateur de la classe, puisqu'il est désormais possible de modifier les membres privés hors du contrôle du concepteur de la classe. Si la définition de la classe change, il va sans dire que le bricolage indiqué çi-dessus se trouve aussitôt invalidé. Enfin, l'adresse des membres privés n'est en aucune facon définie dans le standard C++ et peut varier d'un compilateur à un autre, ou d'une machine à une autre. Le "code" çi-dessus a de fortes chances de fournir des résultats aléatoires (voire une plantée magistrale) lors du portage sur une autre installation.

Malheureusement, l'implémentation de C++ permet ce genre d'abus. D'un autre coté, il est de notorieté publique que tout ce qui n'est pas formellement interdit est consideré comme autorisé. Ce genre de méthode de programmation est effectivement utilisé par certains programmeurs peu soucieux de la qualité de leur code, ou soumis à des pressions excessives par leurs chefs, pour obtenir des effets non prévus par le concepteur d'une classe, lorsque ce concepteur n'est pas accessible (classes achetées, par exemple). Quiconque est un jour amené a commettre une telle héresie est vivement encouragé a encapsuler toutes les parties de code illicites dans un module separé et à mettre le source de ce module sous clé: ce genre de pratique est pour le concepteur de la classe un motif suffisant à l'annulation de toutes prestations de service en cas d'erreurs de logiciel.

En tant que concepteur d'une classe, vous pouvez néanmoins protéger votre propre implémentation contre ce genre d'abus, en recourant à une classe privée. Une classe privée est une classe comme toutes les autres, à l'exception du fait que sa définition ne figure pas dans un fichier en-tête, mais seulement dans un fichier source. Cette définition n'est donc pas visible par les utilisateurs de votre classe, puisque les seules choses que vous publiez sont un fichier en-tête et le fichier objet déjà compilé, mais pas le source.

Vous pouvez ainsi mettre tous les membres privés dans une classe locale, et ne mettre dans la classe que vous allez exporter qu'un pointeur sur la classe locale. De cette manière, l'implémentation est cachée aux autres utilisateurs. Voici comment on aurait pu implémenter la classe Point :

```
extern    class     _PointImplementation;
     // Simple déclaration d'identificateur

class     Point
     {
     private :
          _PointImplementation*        impl;
     public :
          ....
     };
```

La place à réserver pour un pointeur est connue par le compilateur, qui n'a donc pas besoin de la définition de _PointImplementation pour pouvoir compiler Point. La définition de _PointImplementation pourra figurer dans le module implémentation (.C, .CPP, etc...), sous

la forme, par exemple, de :

```
class     _PointImplementation
     {
     public :
          double    real, imag;
     };
```

Pas de scrupules à déclarer des membres données publics dans ce cas, puisque cette classe est purement locale et n'est aucunement destinée à l'exportation.

12.4.2 *Membres publics*

Les membres publics représentent l'interface de la classe avec ses utilisateurs. En principe, il peut s'agir aussi bien de membres données que de membres code. En réalité, et sauf cas bien particuliers, on ne déclare jamais publics des membres de données, car dans ce cas, il deviennent modifiables hors du contrôle de la classe. A tout le moins, on définira une méthode sous la forme d'une procedure "inline" pour accéder à un membre donné. Par exemple :

```
class     XYPoint
     {
     private :
          double x, y;
     public :
          XYPoint(double x = 0.0, double y = 0.0);
               // Les membres suivants ont une implémentation déclarée
               // dans le fichier en-tête; ces membres sont appelés
               // “implicit inline”, et seront implémentés par
               // le compilateur comme fonctions membre inline.
          void setXValue(double xs) { x = xs; }
          void setYValue(double ys) { y = ys; }
          double getXValue() { return x; }
          double getYValue() { return y; }
     };
```

D'aucuns diront qu'il n'y a pas beaucoup plus de sécurité dans cette écriture que dans la déclaration publique de x et y. C'est tout a fait vrai, puisque la classe ne se protège en fait aucunement contre des valeurs de x et y fantaisistes et absurdes. Peut-etre n'est-ce pas nécessaire de se protéger dans ce cas; mais même alors, il est absolument nécessaire de conserver privés x et y. La raison en est que si dans le futur, ou dans une autre implémentation (classes dérivées, par exemple), il devient nécessaire de contrôler la valeur de x ou y, la majeure partie du travail sera déjà faite, et avec un peu de chance, on n'aura pas à modifier l'application, alors que dans le premier cas, l'application doit subir d'importantes modifications. D'autre part, l'accès a des membres privés au travers de membres "`inline`" ne pénalise pas l'exécution: la majeure partie des compilateurs génère dans ce genre de cas une assignation directe, comme si la fonction n'existait pas.

Lorsque l'on invoque un membre public, généralement une méthode, le paramètre indiquant de quelle instance il s'agit est passé implicitement à la fonction qui implémente le mem-

bre. Ainsi :

```
XYPoint xy;
xy.setXValue(1.0);
xy.setYValue(2.0);
...
```

les méthodes invoquées reçoivent, comme paramètre caché, un pointeur sur l'instance xy. Ce pointeur peut être examiné par le code implémentant la méthode incriminée, au moyen du mot reservé `this`. `this` retourne un pointeur sur l'instance courante de l'objet. Ceci permet, au besoin, d'utiliser l'instance courante explicitement, comme dans :

```
class    X;
X&   X::someUserProc(X& arg)
     {
     ...
     return *this;
     }
```

12.4.3 *Membres protégés*

Les membres protégés constituent un mi-chemin très important entre membres privés et membres publics. Ils interviennent dans le cadre de classes derivées, pour autoriser certaines classes derivées à accéder à des membres autrement privés. Ils permettent d'exprimer que l'accès à ces membres est autorisé pour autant que la classe qui y accède est une classe dérivée.

Le mécanisme des membres protégés est utile pour construire des hiérarchies de classes sans avoir à gonfler les interfaces publics. Sans les membres protégés, il serait nécessaire de prévoir des méthodes d'accès aux membres privés des classes de base pour que les classes dérivées puissent fonctionner correctement, tout en garantissant une encapsulation suffisante des données. L'interface public se trouve de ce fait inutilement élargi de méthodes dont une application n'a que faire, et que l'on désirait complètement cacher puisque faisant partie de l'implémentation. Déclarer des membres protégés évite cet inconvénient, tout en garantissant néanmoins que l'application ne puisse pas accéder à ces membres, du moins par des moyens légaux.

12.5 *Opérateurs*

Un aspect intéressant de notre type Point très simple est que, pour atteindre à l'abstraction souhaitée, il redéfinit des opérateurs (ici, par souci de simplicité, on s'est limité à l'assignation et au test d'égalité). Redéfinir un opérateur est intéressant, puisque cela permet une écriture très naturelle du code. Nous avons parlé brièvement de la redéfinition d'opérateurs au niveau global au cours d'un chapitre précédent. Un opérateur peut également être défini comme un membre de la classe.

L'exemple çi-dessous définit pour une classe A hypothétique un opérateur global "addition" (+). Noter que l'implémentation de l'opérateur utilise une fonction membre pour accéder à la valeur du membre privé, sans quoi il n'a pas la possibilité de lire x directement.

```
class     A
     {
     private :
          int x;
     public :
          A(int s) : x(s);
          int getVal() { return x; }
     };

A    operator+(const A& a1, const A& a2)
     // Operateur global.
     {
     A a3(a1.getVal()+ a2.getVal());
     return a3;
     }
     // Il eut en realite ete plus efficace d'implementer
     // cette addition de la maniere suivante :
     //   return (a1.getVal() + a2.getVal());
     // Savez-vous pourquoi ?
```

Implémenter l'opérateur + à l'aide d'un membre s'écrit de la manière suivante. Noter que l'opérateur membre peut accéder de manière directe au membre privé, et de ce fait est beaucoup plus efficace. Il n'est dès lors plus nécessaire de recourir à une méthode permettant de retourner la valeur du membre privé.

```
class     A
     {
     private :
          int x;
     public :
          A(int s) : x(s);
          A operator+(const A& a2);
          A& operator=(const A& a2);
     };

A    A::operator+(const A& a2)
     {
     A a3(x + a2.x);
```

```
     return a3;
     }

const     A&   A::operator=(const A& a2)
     {
     x = a2.x;
     return *this;
     }
```

Mais où est donc passé l'opérateur de gauche dans la déclaration? Dans l'en-tête ne figure effectivement plus qu'un opérande, on pourrait donc croire qu'il en manque un. En réalité, le membre de gauche est passé implicitement: c'est l'instance à laquelle on applique l'opérateur concerné. Comme pour une fonction, il est implicite dans le cas d'une méthode, et un opérateur n'est qu'une méthode un peu particulière. On peut écrire, dans le cas de l'opérateur membre, l'opération addition de deux façons équivalentes :

```
A    a1(1), a2(3);
A    a3 = a1 + a2;
A    a4 = a1.operator+(a2);// identique à la ligne précédente
```

L'opérateur de gauche est donc `this`. Dans le cas d'opérateurs unaires, le seul opérande passé (implicitement !) est `this`.

12.5.1 *Opérateurs globaux / opérateurs membres*

Est-il préférable d'utiliser des opérateurs globaux ou membres ? Pour l'utilisateur de l'opérateur, du point de vue de l'écriture, rien ne semble changer. Mais ce n'est vrai qu'en apparence : reprenons l'exemple de la classe SimpleString, dont nous rappelons çi-dessous la définition:

```
class     SimpleString
     {
     private :
          char *stringContent;
          int  maxLength;
     public :
          SimpleString(int maxLgt = 80);
          SimpleString(char *content);
          SimpleString(const SimpleString &str);
          SimpleString(char *content, int maxLength = 80);
          ~SimpleString();
          ...
     };
```

Complétons-la pour lui ajouter l'opérateur "+" signifiant la concaténation de deux chaînes de caractères. Nous désirons obtenir les possibilités suivantes de cet opérateur :

```
SimpleString + SimpleString
SimpleString + "Chaîne de caractères explicite"
```

```
"Chaîne de caractères explicite" + SimpleString

SimpleString::operator+(const SimpleString&);// Membre
SimpleString::operator+(const char*);// Membre
operator+(const char*, const SimpleString&);// Opérateur global
```

En fait, les deux premiers opérateurs sont redondants. Le premier opérateur peut également traiter le second cas de figure, si nous nous souvenons de la manière dont nous avons défini la classe SimpleString. Elle comporte, entre autres, un constructeur comme suit :

```
SimpleString(char *content);
```

Ce qui signifie que nous avons défini une conversion implicite entre un char* et un SimpleString. Le cas de figure

```
SimpleString + "Chaine de caracteres immediate"
```

est donc traitable par l'opérateur

```
SimpleString + SimpleString
```

car le compilateur, lors de la recherche d'un opérateur correspondant, va essayer la conversion char* --> SimpleString, cette conversion lui ayant été définie comme implicite par la définition de la classe SimpleString.

Notre dernier cas de figure, par contre, nécessite un opérateur global, car l'opérande de gauche d'un opérateur membre doit obligatoirement être un membre explicite. Il ne peut pas y avoir de conversion automatique par le compilateur dans ce cas-là. Cet opérateur est donc nécessaire. Que serait-il advenu si nous avions d'emblée défini un opérateur global, de la manière suivante :

```
operator+(const SimpleString&, const SimpleString&);
```

Cette écriture résoud tous nos cas de figure, sans utilisation de fonctions membres, car le compilateur peut dès lors effectuer une conversion implicite tant sur l'opérande de gauche que celui de droite. Mais attention! Il faut encore implémenter cet opérateur. L'opérateur étant non-membre, il n'a pas accès aux membres privés de la classe SimpleString (ici, stringContent). Il me faut donc encore définir un opérateur membre qui puisse, lui accéder aux membres privés, par exemple :

```
SimpleString SimpleString::operator+(const SimpleString&)
     {
     ... // Opérateur membre
     }
SimpleString operator+(const SimpleString& s1, const SimpleString& s2)
     {
     // Opérateur global, appelle l'opérateur membre de s1
     // L'opérateur membre de s2 convient également
     return s1.operator+(s2);
```

```
}
```

Il est donc utile de bien considérer les opérateurs à définir, pour ne pas devoir écrire des opérateurs superflus.

Un opérateur membre a pour opérande de gauche un objet de sa classe. Si, pour quelque raison que ce soit, l'opérateur requiert un membre de gauche d'une autre type, il n'est plus possible d'utiliser un opérateur membre de la classe considérée. Prenons pour exemple le produit scalaire d'un vecteur :

```
class     Vector
     {
     public :
     ....
          Vector    operator*(float multiplier);// produit scalaire
          Vector&   operator=(Vector &Vect);
     };

Vector A, B, C;
float     mult;
B = A*mult;    // Ok, Vector::operator* used.
C = mult*A;    // Error, operator is undefined (ne compile pas)
CC: "myProg.C", line XX : error : bad operands for *: float * Vector (HPC++)
```

Nous avons défini un opérateur multiplication qui n'est pas commutatif. Une telle classe n'est pas utilisable, car elle redéfinit (surcharge) un opérateur dont les règles de base sont connues, et qui soudain se comporte différemment. Nous avons déjà discuté de ce cas précédemment, et avons rejeté son utilisation parceque non intuitive. Une possibilité est d'introduire, en lieu et place d'un opérateur, une fonction membre (méthode), comme par exemple :

```
Vector&   Vector::multiply(float coefficient);
```

C'est vrai, c'est moins joli. Mais cette notation a l'avantage de lever toute ambiguïté sans pour autant compliquer le code. Nous la rejetons néanmoins, parcequ'elle alourdit la définition du type Vector: définir la notation * pour le produit scalaire (ou vectoriel, pourquoi pas) semble naturel, et ne pas nécessiter de documentation particulière. Mais comment faire? Il vaudrait mieux, dans ce cas, définir un opérateur global, qui pourrait utiliser une fonction comme celle proposée çi-dessus, comme par exemple:

```
Vector    operator*(float multiplier, Vector& vect)
          { vect.multiply(multiplier); }
Vector    operator*(Vector& vect, float multiplier)
          { vect.multiply(multiplier); }
```

Pour rendre cette implémentation plus transparente, on peut cacher la méthode multiply() de l'interface en la rendant privée ou protégée, et déclarer les deux opérateurs çi-dessus amis (friend) de la classe, ce qui leur permet néanmoins l'accès à cette méthode.

Certains opérateurs ne peuvent être redéfinis que s'ils sont membres d'une classe. Il

s'agit des opérateurs d'assignation ("="), de souscription ("[]"), d'appel ("()") et de sélection de membre ("->"). Toute tentative de redéfinir l'un quelconque de ces opérateurs comme opérateur global résultera en une erreur de compilation.

Pour le reste, chaque programmeur est libre de définir des opérateurs globaux ou membres. On peut dire que les opérateurs symétriques, où le membre de gauche peut être ou ne pas être un membre de la classe considérée sont généralement plus faciles à définir comme opérateurs non-membres (globaux).

12.5.2 *Opérateur = (assignation)*

Certains opérateurs ne peuvent être définis que comme membres. Il en va ainsi, en particulier, de l'opérateur d'assignation "=". Lorsque l'on redéfinit l'opérateur =, il faut toujours tenir compte du cas particulier d'assignation d'un objet à lui-même :

```
SimpleString       s1("It is a SimpleString");
s1 = s1;
```

Avant d'assigner le contenu du buffer caractères, il faudra détruire son contenu précédent. En ce faisant, on détruit malheureusement le contenu de la chaîne de caractères que l'on devait recopier. En résumé, il faut se protéger contre l'assignation à soi-même, par exemple de la manière suivante :

```
operator=(const SimpleString& s)
     {
     if (&s == this) return;
     ....
     }
```

Cette manière de faire non seulement évite un risque potentiel d'erreurs, mais a de plus l'avantage d'éviter un travail inutile. Elle implique que this contient l'adresse de s, ce qui est vrai dans presque tous les cas. Presque. Dans le cas de l'héritage multiple, ce n'est plus forcément vrai, et le test çi-dessus peut ne pas fonctionner dans tous les cas. L'héritage multiple doit être traité séparément, et nous verrons pourquoi plus tard.

D'aucuns se sont peut-être étonnés, dans l'exemple de la classe A çi-dessus, que l'on définissait un opérateur = retournant une référence sur un objet de type A. Fréquemment, des programmeurs en C++ novices définissent des opérateurs = dont le type de retour est `void`. Ceci semble à priori raisonnable, jusqu'au moment où l'on se heurte au problème suivant :

```
class     X
     {
     public :
          X(int initialValue);
          void operator=(X& x);

     };
```

```
X    x1, x2, x3, x4(10);
x1 = x2 = x3 = x4;
CC : myFile.C, line XXX : bad operands for =: X = void // HPC++
```

Que s'est-il passé ? Si l'on considère l'associativité de l'opérateur =, la ligne çi-dessus peut s'écrire :

```
x1 = (x2 = (x3 = x4));
```

Le résultat de (x3 = x4) est, comme on l'a défini, un void, ce qui donne pour l'opérateur = appliqué à x2, x2 = void. Cet opérateur n'est bien sûr pas défini. On a donc violé la règle qui veut que lorsque l'on redéfinit un opérateur, celui-ci doit conserver les mêmes priorités que l'original, sous peine d'introduire de la confusion pour les utilisateurs de la classe considérée: l'opérateur = que nous venons de définir n'est pas associatif.

Que faire pour le rendre associatif ? Très simple. Modifions la déclaration de la manière suivante :

```
class     X
     {
     public :
          X(int initialValue);
          const     X&   operator=(X& x);// Now returns a reference type
     };
```

Et terminons l'implémentation de cet opérateur par la ligne :

```
return *this;
```

Notre opérateur = est devenu associatif.

En conclusion, l'opérateur = doit toujours retourner une référence à this.

12.5.3 *Opérateur []*

L'opérateur d'indexation permet de redéfinir l'indexation dans un tableau pour un type quelconque. Le problème est de savoir ce que doit retourner cet opérateur. Soit l'exemple suivant :

```
int  anArray[10];
for (int i = 0; i < 9; i++) anArray[i] = i;
for (int i = 0; i < 9; i++) cout<<anArray[i]<<" ";
cout<<endl;
```

l'opérateur d'indexation permet aussi bien d'assigner une valeur que de la consulter. Donc, l'implémentation normale de cet opérateur retourne une référence sur un élément du tableau. Ceci interdit l'utilisation de l'opérateur [] pour un tableau constant (sinon par le biais de cast déconseillés, parceque difficiles à documenter).

Une solution est de déclarer deux opérateurs [] différents, un pour des instances constantes, l'autre pour des instances non-constantes, comme dans l'exemple suivant :

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

class   Arr3
    {
    float arr[3];
    public :
        Arr3(float c1 = 0, float c2 = 0, float c3 = 0)
        { arr[0] = c1; arr[1] = c2; arr[3] = c3; }
        float operator [](int i) const
            { cout<<" float operator[] called "<< endl;
          return arr[i];}

        float& operator [](int i)
            { cout<<" float& operator[] called "<< endl;
            return arr[i]; }
    };

int     main(int , char* )

    {
    arr3 a;
    const arr3 b;
    a[0] = 1;
    cout<<"Index operation on const "<<b[0]<<endl;
    // b[0] = 1; // Error, does not compile !
    }
```

12.5.4 *Opérateur conversion*

Pour une classe donnée, il est possible de définir un opérateur conversion, dont la syntaxe est la suivante :
`nom-de-la-fonction-de-conversion : operator nom-du-type-de-conversion`

L'opérateur de conversion peut réaliser deux fonctionnalités impossibles à réaliser à l'aide du constructeur :

- Définir une conversion d'une classe en un type de base (le constructeur peut, au mieux, définir une conversion d'un type de base en une classe).
- Définir une conversion d'une classe en une autre sans nécessiter de modification de la déclaration de l'autre classe.

Voici un exemple d'utilisation d'un opérateur de conversion :

```
class    X
     {
     private :
          int i;
```

```
     public :
          X(int xx = 3) : i(xx) {}
          operator int() { return i; }
     };

void f(X a)
     {
     int i = int(a);
     i = (int) a;
     i = a;
     }
```

Dans les trois assignations contenues dans la fonction f, c'est le même opérateur (operator int()) qui est utilisé pour faire la conversion. L'opérateur ainsi défini a donc remplacé le casting que génèrerait le compilateur par une conversion sûre définie par l'utilisateur.

De plus, on a la possibilité de définir des conversions implicites entre des classes différentes. Ainsi, L++, la "librairie standard" de C++ (qui n'est pour l'instant pas encore un standard établi) définit-elle des chaînes de caractère dynamiques avec tous les opérateurs nécessaires pour passer du type String au type char* et vice-versa.

On notera la syntaxe un peu particulière de cet opérateur: c'est forcément un opérateur membre, et il n'a pas de type de retour, bien que l'on termine son implémentation en retournant une valeur. De même, il est illégal de passer un argument à un opérateur de conversion. Ainsi, les deux premières définitions de la classe suivante sont fausses :

```
class     Y
     {
     public :
          int  operator int*();// error
          operator void*(int);// error
          operator char*();// Ok Y -> char*
     };
```

Là encore, il convient de se méfier de l'abus possible de conversions implicites par un programmeur par trop zêlé. Une conversion de type doit dans la mesure du possible pouvoir se passer de documentation pour être utilisable. Dans le cas de conversions entre un type défini par l'utilisateur et un type de base, cette règle devrait même être considérée comme impérative.

12.6 *Opérateurs new et delete*

Tout comme les autres opérateurs, il est possible de redéfinir new et delete. L'opérateur new doit accepter un argument du type size_t, et retourner un pointeur sur un void. Cette redéfinition est utile si l'on veut se prémunir contre une fragmentation trop considérable de la mémoire, et prendre en mains la gestion de la mémoire pour un type donné. Ainsi, on pourrait définir un opérateur new pour des strings, qui au lieu de réserver systématiquement la place mémoire nécessitée par une chaîne de caractères donnée, alloue de la mémoire par blocs de N bytes, n'allouant un nouveau bloc que quand le bloc précédent est complètement occupé par les chaînes de caractère précédemment définies. La réservation effective de mémoire doit impérativement se faire au moyen de la fonction new globale, comme dans

```
void*     operator new(size_t)
     {
     ...
     char *byteArray = ::new char [BLOCKSIZE];
     ...
     }
```

12.6.1 *Quand redéfinir new ?*

D'une manière générale, il est utile de redéfinir la fonction new chaque fois que l'on se trouve dans la situation où il est nécessaire, pour un type donné, de réserver souvent de très petites quantités de mémoire (de l'ordre de la dizaine de bytes). La gestion mémoire fournie par C++ est trop générale pour gérer efficacement de petites quantités de mémoire, et il vaut mieux la prendre en charge soi-même, par exemple en allouant, lorsque c'est nécessaire, de plus gros blocs de mémoire que l'on gère ensuite soi-même.

Si une classe redéfinit la fonction new, il est évident qu'elle doit également redéfinir delete.

La fonction C malloc retourne un pointeur vide (NULL pointer) lorsque la librairie C ne parvient pas à allouer la quantité de mémoire requise. Il existe une fonction équivalente en C++, définie dans new.h, et qui est set_new_handler. La déclaration correspond à :

```
extern    void (*set_new_handler(void(*)()))();
```

Cette fonction prend comme argument un pointeur sur une fonction qui de son coté, ne prend aucun argument et ne fournit aucun résultat. La fonction argument est fournie par l'application. Le seul effet de set_new_handler est de mettre le pointeur de la librairie standard new_handler à la valeur du pointeur sur la fonction utilisateur. Par défaut, ce pointeur vaut 0, donc aucune fonction de traitement de ce cas de figure n'est définie par défaut.

Au cas où l'appel à new résulte en un échec, la fonction définie par l'utilisateur sera dès lors appelée, laissant le soin à l'application de réagir à cette situation comme bon lui semble.

En principe, il est préférable d'utiliser la possibilité d'appel donnée par new_handler, plutot que de laisser le programme se planter lors de la tentative d'accès à de la mémoire inexistante. Quant à savoir exactement ce qu'il convient de faire, et ce qu'il est possible de faire raisonnablement dans une procédure de ce genre, c'est là un autre problème. Dans le cas le plus brutal, on pourrait imaginer une séquence de ce type :

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

void      myNewhandler()
     {
     cerr<<"Erreur lors de l'allocation mémoire"<<endl;
          // cerr est le flot (stream) erreurs
          // correspond à stderr de C
     exit(1);
     }
```

Bien que ce segment de code n'apporte aucune solution valable à l'état de fait, il évite que le programme continue à exécuter du code en situation d'erreur, et donne un diagnostic, aussi succinct soit-il. Le fait de continuer une exécution dans une condition d'erreur peut propager des données corrompues dans des zones mémoires permanentes (fichiers) où elles peuvent perturber plus tard d'autres programmes (propagation d'erreurs, pouvant se manifester beaucoup plus tard et de manière telle que la relation avec le programme original ne puisse plus être faite).

12.6.2 *Variantes*

Par défaut, new alloue de la mémoire depuis une zone appelée heap, qui désigne une zone de mémoire libre, et utilisable sur demande par un programme. Il n’est en principe pas possible de contrôler où se trouve, physiquement, ce heap, ce qui peut poser des problèmes à des applications de bas niveau désirant stocker des données dans une mémoire physique accessible par plusieurs processus, voire machines, séparées (en télécommunications, un exemple typique de ce genre de situation est le tampon entre deux couches OSI implémentées sur deux machines différentes, une configuration typique entre les couches 2 et 3 du modèle à couches).

Il est possible de dire à l’opérateur new où il doit prendre de la place pour ses réservations. La littérature anglo-saxonne parle de *placement new* et de *placement delete*. Cette possibilité peut être très intéressante pour des opérations de bas niveau, où il s’avère indispensable de contrôler la mémoire physique. Il devient ainsi plus aisé de se passer du langage assembleur, de plus en plus abandonné, il faut bien le dire.

Ainsi, le code suivant :

```
#include      <new.h>
char          buffer[8192]

// ...
```

```
MaClasse* pMC = new (buffer) MaClasse;
```

alloue de la mémoire de buffer, non depuis le heap commun à tout programme C++.

12.7 *Amis (friend)*

Dans certains cas, les règles d'accès aux membres privés ou publics peuvent s'avérer trop prohibitives pour permettre une implémentation utilisable d'une fonctionnalité donnée. Les règles de non-visibilité peuvent être sélectivement relaxées pour des éléments spécifiques (fonctions, opérateurs, classes) au moyen du mot-clé friend.

Prenons l'exemple suivant :

La librairie standard de C++ iostream définit deux opérateurs: >> et <<. >> est l'opérateur d'entrée; ainsi,

```
int anInteger;
cin >> anInteger;
```

a pour effet de lire la valeur de l'entier anInteger depuis le flot d'entrée standard (cin). De manière analogue,

```
int anInteger = 10;
cout << "Valeur de anInteger = " << anInteger << endl;
```

a pour effet l'impression, sur le flot de sortie standard cout de la chaîne de caractères suivantes :

```
Valeur de anInteger = 10
```

Je voudrais redéfinir, pour ma classe Point, les opérateurs << et >>, pour qu'ils puissent également traiter des instances de type Point. J'aimerais utiliser ces opérateurs de la manière suivante :

```
double x, y;
cout<<" Abcisse : "; cin >> x; cout << endl;
cout<<" Ordonnee : "; cin >> y; cout << endl;
Point         z(x, y);
cout << "Coordonnées du point = " << z << endl;
```

et obtenir comme sortie :

```
 Abcisse : 2.5
 Ordonnée : 3.4
Coordonnées du point = (2.5, 3.4)
```

Les opérateurs d'entrée et de sortie requièrent, respectivement, un objet de type istream et ostream comme opérandes de gauche. Tous deux retournent comme valeur l'objet sur lequel ils opèrent. Ceci autorise le chaînage des opérateurs input et output :

```
cout << "Valeur de anInteger = " << anInteger << endl;
(((cout << "Valeur de anInteger = ") << anInteger) << endl);
```

Chacune des expressions entre parenthèses retourne l'objet de type ostream cout, qui devient l'opérande de gauche de l'expression suivante (en l'occurence, l'opérateur de sortie suivant). D'un point de vue syntaxique, l'opérateur <<, pour des entiers, doit s'écrire :

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

Par analogie, je devrais pouvoir définir l'opérateur << pour un nombre complexe de la manière suivante :

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

L'implémentation de cet opérateur pourrait ressembler à quelque chose comme :

```
ostream&      operator<<( ostream& os, Point& z )
     {
     os<<“(“<<z.x<<“, ”<<z.y<<“)”;
     return os;
     }
```

Bien évidemment, cela ne peut pas marcher, car x et y sont des membres privés, et l'opérateur << ne peut y accéder que s'il est membre de la classe Point. A quoi ressemblerait l'implémentation de l'opérateur comme membre de la classe Point? Le paramètre Point de l'opérateur est passé implicitement, et devient l'opérande de gauche de l'opérateur. L'implémentation correspondante est :

```
classPoint
     {
     ...
     public :
          ostream& operator<<(ostream &);
     ...
     };
```

Pour utiliser cet opérateur, l'opérande de gauche est un Point, celui de droite un ostream, ce qui donne à l'utilisation:

```
Point z;
z << cout; // au lieu de cout<<z !!!
```

Définir un tel opérateur est au mieux une source de confusions pour le lecteur, donc également pour le programmeur qui, après quelques mois, sera redevenu un lecteur de ses propres programmes. D'autre part, cette manière de faire viole en partie une règle non écrite dans la définition des opérateurs : ne jamais redéfinir un opérateur en lui faisant réaliser des fonctions qui ne correspondent pas à celles réalisées par d'autres implémentations de cet opérateur. Dans le même ordre d'idées, redéfinir l'opérateur division / sur des complexes pour lui faire effectuer la division de l'opérande de droite par l'opérande de gauche (au lieu de la notation habi-

tuelle pour des réels) serait au mieux illisible.

Le mot-clé friend permet de résoudre le problème en autorisant l'opérateur << à accéder aux membres privés de la classe Point. Ce mot-clé ne peut apparaître qu'à l'intérieur d'une définition de classe. Les amis n'étant pas des membres, ils ne sont pas affectés par les mots-clés public, protected ou private. Il existe une convention stylistique largement adoptée qui est de grouper les déclarations d'amis immédiatement à la suite de l'en-tête de classe.

```
class    Point
     {
     friend   istream& operator>>( istream&, Point& );
     friend   ostream& operator<<( ostream&, Point& );

     ...


     };
```

Ayant défini la classe Point ainsi, et implémenté les opérateurs >> et <<, nous pouvons écrire :

```
Point z;
cout << z;
```

Il est non seulement possible de définir des fonctions ou des opérateurs comme amis, mais également des classes entières. Dans ce dernier cas, on déclare par ce biais que la classe amie a un accès sur tous les membres privés de la classe dont elle est l'amie. Notons qu'il est également possible de résoudre ce problème en définissant une méthode membre réalisant la fonction demandée, et en l'appelant depuis l'opérateur global, ceci au détriment d'un appel de fonction supplémentaire.

Les amis (friend) sont importants dans beaucoup de cas. Lors de l'utilisation de C++ pour définir des interfaces utilisateurs en se servant de GUI comme MOTIF, Windows, ou le Toolbox du Macintosh, on est souvent amené à se servir de callbacks, c'est-à-dire d'appels qui sont retournés par l'infrastructure du GUI vers l'application, suite par exemple à la pression, par l'utilisateur, d'un bouton "Ok" dans une boîte de dialogue. Il est en général plus simple, dans ces cas, de se servir d'amis que de fonctions membres. En effet, si il est facile de se faire retourner un pointeur sur une instance d'objet, il est difficile de faire en sorte qu'un infrastructure comme MOTIF se conforme aux règles de passage de paramètres de C++, en particulier en ce qui concerne `this`.

Dans les cas où l'on désire manipuler des objets de deux classes distinctes, il faudrait en principe que la fonction de manipulation soit membre des deux classes, ce qui est bien évidemment impossible. Séparer la fonction de manipulation en sous-fonctions, chacune membre d'une des deux classes est peu pratique et nuit considérablement à la lisibilité. La solution la plus courante est de rendre la fonction considérée membre d'une classe et amie de l'autre. Cette solution ne peut être recommandée que si la logique impose cette appartenance à une classe particulière. Sinon, il est plus correct de rendre cette fonction amie des deux classes. Lorsque ce cas se présente, on peut également se poser brièvement la question de savoir si on ne se trouve pas en présence d'un cas où l'héritage multiple pourrait être utilisé.

12.8 *Membres statiques*

Il est parfois nécessaire de définir, pour tous les membres d'une classe, une variable unique, commune, ayant la même valeur pour toutes les instances de cette classe. La manière évidente de réaliser ceci est au travers d'une variable globale ou d'une constante dans le contexte fichier. La solution d'une constante n'est pas viable, car on ne peut pas faire évoluer sa valeur au cours du programme. La variable globale présente les inconvénients suivants :

Etant globale, elle est accessible en dehors du controle des membres de la classe. Cet effet peut être ou ne pas être souhaité: il est en tous cas indésirable.

Comme variable globale, l'identificateur est connu par toutes les composantes du programme: en particulier, cet identificateur peut entrer en collision avec un autre identificateur.

C'est dans ce genre de circonstances qu'un membre statique est intéressant. Un membre statique n'existe qu'à un seul exemplaire, identique pour toutes les instances de la classe considérée. Sa valeur est unique pour toutes ces instances, ce qui réduit par effet de bord la mémoire nécessitée par le programme. De plus, il est possible de déclarer ce membre privé, de manière à le protéger des accès de l'extérieur.

Un membre statique de données, même privé, peut être initialisé dans le contexte fichier. Le segment de code suivant implémente à l'aide de membres statiques un système de messages pour l'utilisateur. Le fichier dans lequel sont stockés les messages ne doit être ouvert qu'une fois, car c'est le même pour tous les messages. On se sert d'une variable statique pour détecter que ce fichier n'a encore jamais été ouvert, et d'une autre variable statique pour contenir la variable décrivant le fichier.

```
#include      <fstream.h>
#include      <codelibs/boolean.h>
     // codelibs est une librairie C++ fréquemment rencontrée
     // sur les systèmes UNIX, qui définit, entre autres, un type
     // boolean

const     char*     MsgFilename = "msgfile.txt";

class     Message
     {
     private :
          static    ifstream  msgFile(MsgFilename);
          static    boolean   msgFileOpened;
     public :
               // Si msgFileOpened == FALSE, cela signifie que msgFile
               // est fermé, il faut tout d'abord ouvrir
               // le fichier, et mettre msgFileOpened à TRUE avant de
               // continuer. Sinon, le message peut être affiché sans
               // autre.
          Message(int msgNo);// Display the message
          ~Message();   // Clear the message
     };
```

```
boolean  Message::msgFileOpened = FALSE;
```

Les restrictions d'accès aux membres privés d'une classe ne se rapportent qu'à la lecture et à l'écriture, non à l'initialisation, c'est pourquoi on peut accéder à msgFileOpened dans le contexte fichier. Le mécanisme çi-dessus permet d'ouvrir le fichier contenant les messages dès la première utilisation, et non par défaut à l'initialisation du programme (ce qui a souvent pour effet de ralentir le démarrage encore plus).

Une autre utilisation pour des membres statiques est le partage de variables devant évoluer de façon identique pour toutes les instances.

12.9 *Un exemple un peu plus complet*

Nous nous proposons de faire ici un exemple un peu plus complexe que la classe Point, comportant des opérateurs, des opérations de réservation de mémoire, etc... Il s'agit de cacher la représentation des chaînes de caractères en C++ par une classe. En même temps, on rendra ces chaînes de caractères un peu plus flexibles, leur permettant de s'étendre au besoin, de proposer des insertions de chaînes de caractères à des positions prédeterminées, etc...

Voici la définition de cette classe, et partant, le fichier Strings.h :

```
#ifndef_MJN_STRINGS_HDR
#define_MJN_STRINGS_HDR
//
//
// M. Jaton EINEV - Telecommunications
// Date : 14.06.95
// Tel. : 024.4232.381
// email : jaton@einev.ch
//
// Cours special C++ (1995)
// Compilateur(s) : Symantec C++ for Macintosh (Iostream C++ Project)
//                      HP C++ 3.1 (Cfront compliant)
//
//
// Définition de la classe Strings.
// (Simple classe à titre d'exemple de programmation)
//
//
// Un string peut être construit avec un char*,
//                                    avec un char
//                                    avec rien.
// Les opérateurs << et >> fonctionnent avec un object String
// sur un ostream (cout) ou un istream (cin)
// Les opérations d'addition de strings sont proposées, ainsi
// que divers utilitaires.
//

#include<iostream.h>

class Strings
    {
     friend   ostream&  operator<<(ostream& os, const Strings& s);
     friend   istream&  operator>>(istream& is, Strings& s);
     friend Strings     operator+(const Strings& s1, const Strings& s2);
     private :
          char     *str;
          void     overwrite(char *newContent);
    public:
          // Constructeurs
          // Les constructeurs définissent également des conversions
          // implicites.
         Strings(char *str);
```

```
        Strings(char c);
        Strings();
        //
        // Le constructeur de copie est ici mandatoire
        // (Réservation de mémoire dynamique)
        //
        Strings(const Strings &t);
        ~Strings();
        //
        // Assignation
        //
        const Strings &operator=(const Strings &);
            //
            // Dans le membre size() ci-dessous, le qualificateur
            // const place apres l'identite du membre specifie que
            // l'appel du membre ne modifie en rien l'objet. On pourrait
            // donc appeler ce membre avec une instance d'objet constante.
            //
        long size() const;
        //
        // Opérateurs relationnels :
        // (retournent zéro si la relation est FAUSSE)
        //
        int   operator==(const Strings& s) const;
        int   operator<(const Strings& s) const;
        int   operator>(const Strings& s) const;

        char &operator[](long e);
        //
        // Position d'un substring dans un string
        //
        long  position(const Strings& subString) const;
        //
        // Insertion de substring dans un string à
        // une position déterminée :
        //
        void  insert(const Strings& subString, int where);
   };

#endif
```

Voici l'implémentation de la classe Strings, dans le fichier Strings.C :

```
#include  "Strings.h"
#include  <string.h>
static    char aChar;

void Strings::overwrite(char *s)

     {
     if (str != NULL) delete [] str;
     if (s == NULL) return;
     str = new char [strlen(s) + 1];
     strcpy(str, s);
     }
```

```
Strings::Strings() { str = NULL; }

Strings::Strings(char *s)

     {
     str = NULL;
     overwrite(s);
     }

Strings::Strings(char c)

     {
     str = new char [2];
     str[0] = c;
     str[1] = '\0';
     }

Strings::Strings(const Strings& t)

    {
    str = NULL;
    overwrite(t.str);
    }

Strings::~Strings()

     {
     if (str != NULL) delete[] str;
     }


long Strings::size() const

     {
     return strlen(str);
     }


const     Strings &Strings::operator=(const Strings &a)

    {
    if (&a != this) overwrite(a.str);
     return *this;
    }

char&    Strings::operator[](long e)

    {
    aChar = '\0';
    if (str == NULL) return aChar;
    if (e <= 0 || e >= size()) return aChar;
    return str[e];
    }

Strings   operator+(const Strings& s1, const Strings& s2)
```

```
     {
     int k;

     char *buf = new char [ (k = s1.size()) + s2.size() + 1];
     if (s1.str != NULL)
          strcpy(buf, s1.str);
     if (s2.str != NULL)
          strcat(buf, s2.str);
     Strings temp(buf);
     delete [] buf;
     return temp;
     }

long Strings::position(const Strings& subString) const

     {
     if (str == NULL) return -1;
     char *p = strstr(str, subString.str);
     if (p == NULL) return -1;
     return (p - str);
     }

void Strings::insert(const Strings& subString, int where)

     {
     if (str == NULL)
          {
          overwrite(subString.str);
          return;
          }
     if (subString.str == NULL) return;
     if (where > size()) where = size();
     if (where < 0) where = 0;
     char *buf = new char [ size() + subString.size() ];

     strncpy(buf, str, where);
     buf[where] = '\0';
     strcat(buf, subString.str);
     strcat(buf, &str[where]);
     delete [] str;
     str = buf;
     }

int      Strings::operator==(const Strings& s) const

     {
     return (strcmp(str, s.str) == 0);
     }

int      Strings::operator<(const Strings& s) const

     {
     return (strcmp(str, s.str) < 0);
     }

int      Strings::operator>(const Strings& s) const
```

```
     {
     return (strcmp(str, s.str) > 0);
     }

ostream&  operator<<(ostream& os, const Strings& s)
     {
     os<<s.str;
     return os;
     }

istream&  operator>>(istream& is, Strings& s)

     {
     if (s.str != NULL) delete [] s.str;
     char c;
     do
          {
          is.get(c);
          if (c != '\n') s = s + c;
          }
     while (c != '\n');
     return is;
     }
```

12.10 *Test*

1. Quels sont les éléments constitutifs d'une classe ? Qu'est-ce qui distingue une classe d'une `struct` ?
2. La définition de classe suivante est-elle correcte, et satisfaisante ?

```
class     UneClasse
     {
     public :
          int x;
          double y;
          UneClasse();
          UneClasse(const UneClasse&);
          ~UneClasse();
     private :
          char*     str;
     public :
          void      uneMethode(int leParametre);
     };
```

3. Quel est le rôle du constructeur dans une classe ?
4. Quel est le rôle du destructeur dans une classe ?
5. Pourquoi ne peut-on avoir qu'un destructeur, alors qu'il est possible de définir plusieurs constructeurs ?
6. Par quel moyen publie-t-on une définition de classe, et comment l'implémente-t-on ? Illustrer ce schéma par la définition d'une classe TresSimple contenant un entier comme membre privé, un constructeur initialisant cet entier et une méthode lisant cet entier. Détailler les différentes étapes permettant de publier le code ainsi crée pour d'autres utilisateurs.
7. Quelle est l'utilité pour des membres friend ? Où figure la déclaration friend ?
8. Quelle est l'utilité de membres statiques ? Comment peut-on initialiser un membre statique, et comment peut-on modifier sa valeur dans une classe ?
9. Dans la définition de UneAutreClasse ci-dessous, ajouter l'opérateur *, qui a pour effet de multiplier le membre x par l'opérande de droite.

```
class     UneAutreClasse
     {
     private :
          int x;
     public :
          UneAutreClasse();
          UneAutreClasse(int xx);
          UneAutreClasse(const UneAutreClasse& cxx);
          int  lire();           // retourne la valeur du membre privé
     };
```