Classes et instances

L'objectif poursuivi par Bjarne Stroustrup était, rappelons-le, d'implémenter sur un compilateur C les classes décrites par le langage Simula. Ces deux derniers langages étant radicalement opposés dans leur approche, il fallait identifier une double continuité, notamment du côté du C.

Il fut aisé de remarquer que la programmation serait grandement simplifiée si certaines fonctions pouvaient migrer à l'intérieur des structures du C. De ce fait, il n'y aurait plus de structure à passer à ces fonctions puisqu'elles s'appliqueraient évidemment aux champs de la structure.

Toutefois, il fallait conserver un moyen de distinguer deux instances et c'est pour cette raison que l'on a modifié la syntaxe de l'opérateur point :

Programmation fonctionnelle	Programmation orientée objet

`struct Point`
`{`
`int x,y;`
`};`
`void afficher(Point p)`
`{`
`printf("%d,%d\n",p.x,p.y);`
`}`
	`struct Point`
	`{`
	`int x,y;`
	`void afficher()`
	`{`
	`printf("%d,%d\n",x,y);`
	`}`
	`};`

`Point p1;` `afficher(p1);`	`Point p1;` `p1.afficher();`

Cette différence d'approche a plusieurs conséquences positives pour la programmation. Pour commencer, le programmeur n'a plus à effectuer un choix parmi les différents modes de passage de la structure à la fonction `afficher()` . Ensuite, nous

allons pouvoir opérer une distinction entre les éléments (champs, fonctions) de premier plan et de second plan. Ceux de premier plan seront visibles, accessibles à l'extérieur de la structure. Les autres seront cachés, inaccessibles.

Ce procédé garantit une grande indépendance dans l'implémentation d'un concept, ce qui induit également une bonne stabilité des développements.

1. Définition de classe

Une classe est donc une structure possédant à la fois des champs et des fonctions. Lorsque les fonctions sont considérées à l'intérieur d'une classe, elles reçoivent le nom de méthodes.

L'ensemble des champs et des méthodes est désigné sous le terme de membres. Nous ne recommandons pas la désignation des champs à l'aide du terme attribut, car il peut prendre un sens très particulier en langage C++ managé ou en langage C#.

Pour le lecteur qui passe de Java à C++, il faut faire attention à terminer la déclaration d'une classe par le caractère point-virgule, une classe étant la continuité du concept de structure :

```
class Point
{
int x,y;  // deux champs
   // une méthode
   void afficher()
   {
      printf("%d,%d\t",x,y);
   }
}; // point-virgule
```

La classe suit globalement les mêmes règles que la structure pour ce qui est de son utilisation : instanciation / allocation, copie, passage à une fonction...

a. Les modificateurs d'accès

Nous allons à présent opérer une distinction entre les méthodes (fonctions) accessibles

depuis l'extérieur de la classe et celles qui n'ont qu'une utilité algorithmique. De même, certains champs sont exposés à l'extérieur de la classe, leur accès en lecture et en modification est autorisé, alors que d'autres doivent être protégés contre des accès intempestifs.

Cette distinction laisse une grande latitude dans l'organisation d'une classe, la partie cachée pouvant évoluer sans risque de remettre en question le reste du programme, la partie accessible étant au contraire considérée comme stable.

Le langage C++ offre plusieurs niveaux d'accessibilité et la palette de possibilités est assez large, si bien que certains langages qui lui succèdent dans l'ordre des publications ne les retiennent pas tous.

Modificateur	Accessibilité
`public`	Complète. Le membre - champ ou méthode - est visible à l'intérieur de la classe comme à l'extérieur.
`private`	Très restreinte. Le membre n'est accessible qu'à l'intérieur de la classe.
`protected`	Restreinte à la classe courante et aux classes dérivées.
`friend`	Restreinte à une liste de fonctions identifiées comme étant amies.

Donnons maintenant un exemple de visibilité. Sachant que la visibilité par défaut dans une classe est `private`, nous allons choisir pour chaque membre un modificateur d'accès :

```
class Point
{
private:
  int x,y;
public:
```

```
    void afficher()
    {
      printf("%d,%d\t",x,y);
    }
    void positionner(int X,int Y)
    {
      x=X;
      y=Y;
    }
    int couleur;
};
```

Avant d'utiliser cette classe, dressons un tableau des visibilités pour chaque membre de la classe `Point` :

`x`	`private` (caché)
`y`	`private` (caché)
`afficher`	`public`
`positionner`	`public`
`couleur`	`public`

Les champs `x` et `y` sont bien présents pour chaque instance, mais ils ne sont lisibles/modifiables que par des méthodes appartenant à la classe, comme `afficher()` et `positionner()`. Les autres fonctions, même si elles appartiennent à des classes, n'ont pas d'accès direct à `x` et `y`. Elles doivent passer par des méthodes publiques de la classe `Point`.

Le champ `couleur` est lui public, ce qui signifie qu'il est complètement exposé.

Essayons maintenant d'appliquer ces règles de visibilité à la classe sus-décrite.

```
Point p;
 p.x=3;  // erreur, x est privé
 p.y=2;  // erreur, y est privé
 p.positionner(89,2);  // ok, positionner() est publique
 p.afficher();  // ok, afficher() est publique
 p.couleur=0x00FF00FF;  // ok, couleur est public
```

Lorsque les commentaires indiquent erreur, cela signifie que le compilateur relèvera la ligne comme n'étant pas valide. Le message circonstanciel peut avoir la forme "champ inaccessible" ou "champ privé", voire "champ invisible". Parfois le compilateur est plus explicite en indiquant que la portée privée (`private`) d'un membre ne convient pas à l'usage que l'on en fait.

Pour découvrir des exemples de champs `protected` ou de fonctions amies, nous devrons attendre encore quelques pages.

Pour le programmeur qui débute, le choix d'une portée n'est pas facile à opérer. Et il faut éviter de suivre le conseil suivant qui se révèle trop stéréotypé, un peu grossier : tous les champs sont privés (`private`) et toutes les méthodes sont publiques. Cette règle peut convenir dans certains cas, mais sans doute pas dans tous les cas de figure. D'abord, elle favorise trop la notion d'interface au détriment de l'algorithmie (la programmation fonctionnelle). Or une classe ne se résume pas à une interface, sans quoi l'encapsulation - la réunion d'une structure et de fonctions - n'aurait aucun intérêt. Les détails d'implémentation ont besoin d'être cachés pour évoluer librement sans remettre en question le reste du programme, fort bien. Ce qui nous conduit à déclarer certaines méthodes avec un niveau de visibilité inférieur à public. De même, certains champs ont un typage particulièrement stable, et si aucun contrôle n'est nécessaire quant à leur affectation, il n'y a aucune raison de les déclarer de manière privée.

Bien que le choix d'une visibilité puisse être décidé par le concepteur qui s'aidera d'un système de modélisation comme UML, le programmeur à qui incomberait cette responsabilité peut s'aider du tableau suivant pour décider quelle visibilité choisir. En l'absence de dérivation (héritage), le nombre de cas de figure est assez limité, et il faut bien reconnaître que les programmes contenant un nombre élevé de dérivations ne sont pas légion, surtout sans l'emploi d'UML.

champ algorithmique	`private` ou `protected`
champ de structure	`public`
champ en lecture seule	`private` ou `protected` avec une méthode `getXXX()` publique
champ en écriture seule	`private` ou `protected` avec une méthode `setXXX()` publique
champ en lecture et écriture avec contrôle des accès	`private` ou `protected`, avec deux méthodes publiques `getXXX()` et `setXXX()`
champ "constante" de classe	champ public, statique et sans doute déclaré `const`
méthode caractérisant les opérations accessibles à un objet	Publique. La méthode fait alors partie de l'interface
Méthode destinée à porter l'algorithmie	`private` ou `protected`

Nous constatons dans ce tableau que les fonctions amies n'y figurent pas. Il s'agit d'un concept propre à C++, donc très peu portable, qui a surtout de l'utilité pour la surcharge des opérateurs lorsque le premier argument n'est pas du type de la classe (reportez-vous à la section Autres aspects de la POO - Surchage d'opérateurs sur la surcharge pour d'autres détails).

Également, il est fait plusieurs fois mention du terme "interface". Une interface est une classe qui ne peut être instanciée. C'est un concept. Il y a deux façons d'envisager l'interface, selon qu'on la déduit d'une classe ordinaire ou bien que l'on construit une

classe ordinaire à partir d'une interface. Dans le premier cas, on déduit l'interface d'une classe en créant une liste constituée des méthodes publiques de la classe. Le concept est bâti à partir de la réalisation concrète. Dans le second cas, on crée une classe dérivant (héritant) de l'interface. Le langage C++ n'a pas de terme spécifique pour désigner les interfaces, bien qu'il connaisse les classes abstraites. On se reportera à la section Héritage - Méthodes virtuelles et méthodes virtuelles pures sur les méthodes virtuelles pures pour terminer l'étude des interfaces.

Quoi qu'il en soit, c'est une habitude en programmation orientée objet de désigner l'ensemble des méthodes publiques d'une classe sous le terme d'interface.

b. Organisation de la programmation des classes

Il existe avec C++ une organisation particulière de la programmation des classes. Le type est défini dans un fichier d'en-tête `.h`, comme pour les structures, alors que l'implémentation est généralement déportée dans un fichier source `.cpp`. Nous nous rendrons compte par la suite, en étudiant les modules, que les notations correspondantes restent très cohérentes.

En reprenant la classe `Point`, nous obtiendrons deux fichiers, `point.h` et `point.cpp`. À la différence du langage Java, le nom de fichier n'a d'importance que dans les inclusions et les makefiles. Comme il est toujours explicité par le programmeur, aucune règle syntaxique n'impose de noms de fichiers pour une classe.

Toutefois, par souci de rigueur et de cohérence, il est d'usage de nommer fichier d'en-tête et fichier source à partir du nom de la classe, en s'efforçant de ne définir qu'une classe par fichier. Donc si la classe `PointColore` vient compléter notre programme, elle sera déclarée dans `PointColore.h` et définie dans `PointColore.cpp`.

Voici pour commencer la déclaration de la classe `Point` dans le fichier `point.h` :

```
class Point
{
private:
  int x,y;
public:
  void afficher();
```

```
  void positionner(int X,int Y);
  int couleur;
} ;
```

Nous remarquons que les méthodes sont uniquement déclarées à l'aide de prototypes (signature close par un point-virgule). Cela suffit aux autres fonctions pour les invoquer, peu importe où elles sont réellement implémentées.

Ensuite, nous implémentons (définissons) la classe `Point` dans le fichier `point.cpp` :

```
void Point::afficher()
{
printf("%d,%d\t",x,y);
}

void Point::positionner(int X,int Y)
{
x=X;
y=Y;
}
```

Dans cette écriture, il faut comprendre que l'identifiant de la fonction `afficher(...)` se rapporte à la classe `Point` (à son espace de noms, en fait, mais cela revient au même). C'est la même technique pour la méthode `positionner()` .

Nous avions déjà rencontré l'opérateur de résolution de portée `::` lorsque nous voulions atteindre une variable globale masquée par une variable locale à l'intérieur d'une fonction. En voilà une autre utilisation et ce n'est pas la dernière.

De nos jours, les compilateurs C++ sont très rapides et il n'y a plus de différence de performances à utiliser la définition complète de la classe lors de la déclaration ou bien la définition déportée dans un fichier `.cpp`. Les développeurs Java préféreront vraisemblablement la première version qui coïncide avec leur manière d'écrire une classe, mais l'approche déportée du C++, en plus d'être standard, offre comme avantage une lisibilité accrue si votre classe est assez longue. En effet, seule la connaissance des champs et de la signature des méthodes est importante pour utiliser une classe, alors que l'implémentation est à la charge de celui qui a créé la classe.

2. Instanciation

S'il est une règle impérative en science des algorithmes, c'est bien celle des valeurs consistantes pour les variables. Ainsi, l'écriture suivante est une violation de cette règle :

```
int x,y;
   y = 2*x;
```

La variable `x` n'étant pas initialisée, il n'est pas possible de prédire la valeur de `y`. Bien que certains compilateurs initialisent les variables à 0, ce n'est ni une règle syntaxique, ni très rigoureux. Donc, chaque variable doit recevoir une valeur avant d'être utilisée, et si possible dès sa création.

D'ailleurs, certains langages comme Java ou C# interdisent l'emploi du fragment de code ci-dessus, soulevant une erreur bloquante et interrompant le processus de compilation.

Que se passe-t-il à présent lorsqu'une instance de classe (structure) est créée ? Un jeu de variables tout neuf est disponible, n'attendant plus qu'une méthode vienne les affecter, ou bien les lire. Et c'est cette dernière éventualité qui pose problème. Ainsi, considérons la classe `Point` et l'instanciation d'un nouveau point, puis son affichage. Nous employons l'instanciation avec l'opérateur `new` car elle est plus explicite que l'instanciation automatique sur la pile.

```
Point*p;       // un point qui n'existe pas encore
 p=new Point;   // instanciation
 p->afficher(); // affichage erroné, trop précoce
```

En principe, la méthode `afficher()` compte sur des valeurs consistantes pour les champs de coordonnées `x` et `y`. Mais ces champs n'ont jusqu'à présent pas été initialisés. La méthode `afficher()` affichera n'importe quelles valeurs, violant à nouveau la règle énoncée ci-dessus.

Comment alors initialiser au plus tôt les champs d'une nouvelle instance pour éviter au programmeur une utilisation inopportune de ses classes ? Tout simplement en faisant coïncider l'instanciation et l'initialisation. Pour arriver à ce résultat, on utilise un

constructeur, méthode spéciale destinée à initialiser tous les champs de la nouvelle instance.

Nous compléterons l'étude des constructeurs au chapitre suivant et en terminons avec l'instanciation, qui se révèle plus évoluée que pour les structures.

Pour ce qui est de la syntaxe, la classe constituant un prolongement des structures, ces deux entités partagent les mêmes mécanismes :

- réservation par `malloc()` ;
- instanciation automatique, sur la pile ;
- instanciation à la demande avec l'opérateur `new`.

Pour les raisons qui ont été évoquées précédemment, la réservation de mémoire avec la fonction `malloc()` est à proscrire, car elle n'invoque pas le constructeur. Les deux autres modes, automatique et par l'opérateur `new`, sont, eux, parfaitement applicables aux classes.

```
Point p, m;  // deux objets instanciés sur la pile
Point* t = new Point; // un objet instancié à la demande
```

3. Constructeur et destructeur

Maintenant que nous connaissons mieux le mécanisme de l'instanciation des classes, nous devons définir un ou plusieurs constructeurs. Il est en effet assez fréquent de trouver plusieurs constructeurs, distingués par leurs paramètres (signature), entraînant l'initialisation des nouvelles instances en fonction de situations variées. Si l'on considère la classe `chaine`, il est possible de prévoir un constructeur dit par défaut, c'est-à-dire ne prenant aucun argument, et un autre constructeur recevant un entier spécifiant la taille du tampon à allouer pour cette chaîne.

a. Constructeur

Vous l'aurez compris, le constructeur est une méthode puisqu'il s'agit d'un groupe

d'instructions, nommé, utilisant une liste de paramètres. Quelles sont les caractéristiques qui le distinguent d'une méthode ordinaire ? Tout d'abord le type de retour, car le constructeur ne retourne rien, pas même `void`. Ensuite, le constructeur porte le nom de la classe. Comme C++ est sensible à la casse - il différencie les majuscules et les minuscules - il faut faire attention à nommer le constructeur `Point()` pour la classe `Point` et non `point()` ou `Paint()`.

Comme toute méthode, un constructeur peut être déclaré dans le corps de la classe et défini de manière déportée, en utilisant l'opérateur de résolution de portée.

```
class Chaine
{
private:
  char*buffer;
  int t_buf;
  int longueur;

public:
  // un constructeur par défaut
  Chaine()
  {
    t_buf = 100;
    buffer = new char[t_buf];
    longueur = 0;
  }

  // un autre constructeur, défini hors de la classe
  Chaine(int taille);
} ;

Chaine::Chaine (int taille)
{
  t_buf = taille;
  buffer = new char[t_buf];
  longueur = 0;
}
```

Vous aurez bien noté le type de retour du constructeur : aucun type n'est spécifié. Le choix d'une implémentation au sein même de la classe ou alors hors d'elle est le vôtre, il suit la

même logique que celle applicable aux méthodes ordinaires.

b. Le pointeur this

Il n'est pas rare qu'un constructeur reçoive un paramètre à répercuter directement sur un champ. Dans l'exemple précédent, l'argument `taille` a servi à initialiser le champ `t_buf`. Pourquoi alors s'embarrasser d'un nom différent si les deux variables désignent la même quantité ?

À cause du phénomène de masquage, répondrez-vous à juste titre :

```
Chaine::Chaine (int t_buf)
{
  t_buf = t_buf; // erreur
  buffer = new char[t_buf];
  longueur = 0;
}
```

La ligne normalement chargée d'initialiser le champ `t_buf` affecte l'argument `t_buf` en lui attribuant sa propre valeur. L'affaire se complique lorsqu'on apprend que l'opérateur de résolution de portée ne permet pas de résoudre l'ambiguïté d'accès à une variable de plus haut niveau comme nous l'avons fait pour le cas des variables globales. Il faut alors recourir au pointeur `this`, qui désigne toujours l'adresse de l'instance courante :

```
Chaine::Chaine (int t_buf)
{
  this->t_buf = t_buf;  // correct
  buffer=new char[t_buf];
  longueur=0;
}
```

Le pointeur `this` est toujours `Type*`, où `Type` représente le nom de la classe. Dans notre constructeur de classe `Chaine`, `this` est un pointeur sur `Chaine`, soit un `Chaine*`.

Signalons au passage que `this` peut atteindre n'importe quelle variable de la classe,

quelle que soit sa visibilité, mais pas les champs hérités de classes supérieures lorsqu'ils sont privés. Par ailleurs, il est inutile d'utiliser `this` lorsqu'il n'y a pas d'équivoque entre un paramètre, une variable locale et un champ de la classe.

Enfin, `this` peut être utilisé pour transmettre l'adresse de l'objet courant à une fonction ou une méthode. Comme application très indirecte, on peut demander au constructeur d'afficher l'adresse du nouvel objet :

```
printf("Adresse %x\n",(int) this);
```

c. Destructeur

Puisque le constructeur a un rôle d'initialisation, il semblait logique de prévoir une méthode chargée de rendre des ressources consommées par un objet avant sa destruction.

Rappelons que la destruction de l'objet intervient soit à la demande du compilateur, lorsque le flot d'exécution quitte la portée d'une fonction ou d'un bloc ayant alloué automatiquement des objets, soit lorsque le programme reçoit l'ordre de destruction au moyen de l'opérateur `delete`.

```
void destruction()
{
Point* p;
Point m;  // instanciation automatique
 p = new Point; // instanciation manuelle
 printf("deux objets en cours");
 delete p; // destruction de p
 return;   // destruction de m implicite
}
```

Le destructeur ne reçoit jamais d'arguments, ce qui est tout à fait normal. Comme il peut être invoqué automatiquement, quelles valeurs le compilateur fournirait-il à cette méthode ?

Comme le constructeur, il ne renvoie rien, pas même `void` et porte le nom de la classe. En

fait, il est reconnaissable par la présence du symbole ~ (tilde) placé avant son nom :

```
~Chaine()
{
  delete buffer;
}
```

Comme pour le constructeur, la syntaxe n'exige pas de doter chaque classe d'un destructeur. Sa présence est d'ailleurs liée à d'éventuelles réservations de mémoire ou de ressources système maintenues par l'objet.

La présence du constructeur n'est pas non plus imposée par la syntaxe, mais nous avons vu que son absence conduisait à un non-sens algorithmique.

Enfin, comme toute méthode, le destructeur peut être défini dans le corps de la classe ou bien de manière déportée, la syntaxe ressemblant alors à cela :

```
Chaine::~Chaine()
{
  delete buffer;
}
```

d. Destructeur virtuel

Bien que nous ne connaissions pas encore la dérivation (et l'héritage), ni les méthodes virtuelles, vous devez savoir que les destructeurs ont tout intérêt à être virtuels si la classe est héritée. Sans trop anticiper sur notre étude, prenez en compte le fait que les constructeurs sont toujours virtuels. Nous illustrerons le concept des méthodes virtuelles et des destructeurs virtuels un peu plus loin dans ce chapitre. Entre temps, voici la syntaxe de déclaration d'un tel destructeur :

```
virtual ~Chaine()
{
delete buffer;
}
```

Certains environnements de développement comme Visual C++ (Microsoft) déclarent systématiquement un constructeur et un destructeur virtuel pour une classe.

4. Allocation dynamique

Intéressons-nous maintenant à ce type de programmation appelé programmation dynamique. Comme les classes constituent un prolongement des structures, il est tout à fait possible de définir un pointeur de type `Classe*` pour construire des structures dynamiques, telles les listes, les arbres et les graphes. Nous vous proposons un exemple de classe `Liste` pour illustrer ce principe en même temps que les méthodes statiques.

D'autre part, vous devez faire attention lorsque vous allouez un tableau d'objets. Imaginons le scénario suivant :

```
Chaine*chaines;
 chaines=new Chaine[10];
```

Quel est le fonctionnement attendu par l'opérateur `new` ? Allouer 10 fois la taille de la classe `Chaine` ? Doit-il aussi appeler chaque constructeur pour chacune des 10 chaînes nouvellement créées ?

Pour déterminer son comportement, ajoutons un message dans le corps du constructeur :

```
class Chaine
{
private:
 char* buffer;
 int t_buf;
 int longueur;
public:
 // un constructeur par défaut
 Chaine()
 {
  printf("Constructeur par défaut, Chaine()\n");
  t_buf = 100;
```

```
    buffer = new char[t_buf];
    longueur=0;
  }

  Chaine (int t_buf)
  {
    printf("Constructeur Chaîne(%d)\n", t_buf);
    this->t_buf = t_buf;
    buffer = new char[t_buf];
    longueur = 0;
  }
};

int main(int argc, char* argv[])
{
  Chaine*chaines;
  chaines=new Chaine[10];
  return 0;
}
```

Testons le programme et analysons les résultats : le constructeur est bien appelé 10 fois. Comment dès lors appeler le deuxième constructeur, celui qui prend un entier comme paramètre ?

Vous ne pouvez pas combiner la sélection d'un constructeur avec paramètre et la spécification d'un nombre d'objets à allouer avec l'opérateur `new`. Il faut procéder en deux étapes, puisque l'écriture suivante n'est pas valide :

```
Chaines = new Chaine(75)[10];
```

Optez alors pour la tournure suivante, même si elle paraît plus complexe :

```
Chaine** chaines;
  chaines = new Chaine*[10];
  for(int i=0; i<10; i++)
    chaines[i] = new Chaine(75);
```

Attention, vous aurez remarqué le changement de type de la variable `chaines` , passée de `Chaine*` à `Chaine**` . Ce changement diffère l'instanciation de l'allocation du tableau, devenu un simple tableau de pointeurs.

5. Constructeur de copie

Nous avons découvert lors de l'étude des structures que la copie d'une instance vers une autre, par affectation, avait pour conséquence la recopie de toutes les valeurs de la première instance vers la seconde. Cette règle reste vraie pour les classes.

```
Chaine s(20);
Chaine r;
Chaine x = s; // initialisation de x avec s
r = s;       // recopie s dans r
```

Nous devons également nous rappeler que cette copie pose des problèmes lorsque la structure (classe) possède des champs de type pointeur. En effet, l'instance initialisée en copie voit ses propres pointeurs désigner les mêmes zones mémoire que l'instance source. D'autre part, lorsque le destructeur est invoqué, les libérations à répétition des mêmes zones mémoire auront probablement un effet désastreux pour l'exécution du programme. Lorsqu'un bloc a été libéré, il est considéré comme perdu et libre pour une réallocation, donc il convient de ne pas insister.

Les classes de C++ autorisent une meilleure prise en charge de cette situation. Il faut tout d'abord écrire un constructeur dit de copie, pour régler le problème d'initialisation d'un objet à partir d'un autre. Ensuite, la surcharge de l'opérateur (cf. Autres aspects de la POO - Surcharge d'opérateurs) se charge d'éliminer le problème de la recopie par affectation.

```
class Chaine
{
// ... la déclaration ci-dessus
public:
  Chaine(const Chaine&);  // constructeur de copie
  Chaine& operator = (const Chaine&); // affectation copie
```

```
};
```

Commençons par le constructeur de copie :

```
Chaine::Chaine(const Chaine & ch)
{
  t_buf = ch.t_buf;
  longueur = ch.longueur;
  buffer = new char[ch.t_buf];   // ch référence
  for(int i=0; i<ch.longueur; i++)
    buffer[i]=ch.buffer[i];
}
```

Le constructeur de copie prend comme unique paramètre une référence vers un objet du type considéré.

Poursuivons avec la surcharge de l'opérateur d'affectation :

```
Chaine& Chaine::operator=(const Chaine& ch)
{
  if(this != &ch)
  {
    delete buffer;
    buffer = new char[ch.t_buf];   // ch référence
    for(int i=0; i<ch.longueur; i++)
      buffer[i] = ch.buffer[i];
    longueur = ch.longueur;

  }
  return *this;
}
```

Il faut faire attention à ne pas libérer par erreur la mémoire lorsque le programmeur procède à une affectation d'un objet vers lui-même :

```
s = s;
```

Pour une classe munie de ces deux opérations, les recopies par initialisation et par affectation ne devraient plus poser de problèmes. Naturellement, le programmeur a la charge de décrire l'algorithme approprié à la recopie.