Programmation orientée objet en C++

Département Automatisation et Informatisation - Bertrand Cottenceau

Ce support de cours de C++ a été réalisé à partir d'un document de Serge Tahé écrit en 1993 pour l'ISTIA. Je tiens donc à le remercier ici pour la base de travail que son document a fournie.

Les exemples donnés dans ce document ont été évalués avec le compilateur de Visual C++ 6.

Pour toute documentation supplémentaire :

« Programmer en langage C++ »
C. Delannoy - Eyrolles (597 pages).

« Comprendre et utiliser C++ pour programmer objets »
G. Clavel I. Trillaud et L.Veillon – Masson (234 pages)

« Pont entre C et C++ »
P.-N. Lapointe – Addison-Wesley (184 pages)

Bertrand Cottenceau

ISTIA
Bureau 311
Email : bertrand.cottenceau@istia.univ-angers.fr
URL :http://www.univ-angers.fr/~cottence

TABLE DES MATIERES

1 INTRODUCTION

Le langage C++ constitue une sur-couche du langage C. Il utilise la syntaxe du langage C à laquelle a été ajoutée la notion de *classe* qui représente un modèle d'objets, de même qu'un type représente un modèle de variables. Il s'agit d'un langage dit *orienté objet* (et non objet pur).

En C++, on peut sans problème mélanger programmation dite procédurale et programmation objet. De ce fait, le caractère objet d'un programme dépendra essentiellement de la bonne volonté du programmeur à se conformer aux concepts de la programmation objet qu'on présente brièvement dans ce chapitre.

1.1 Rappels sur le langage C

Puisque le C++ est greffé sur le langage C, il est important de maîtriser certaines notions de base du C. On utilisera notamment dans ce cours :

- **les types scalaires du C :**

```
int, float, long, double, short, char, …
```

- **les tableaux :**

```
char chaine[26]= "bonjour";
int tab[3]={1,2,5};
```

- **les types structurés** (la notion de classe généralise d'ailleurs la notion de structure) **:**

```
struct fiche
{
   char nom[30];
   char prenom[30];
   unsigned int age;
};
```

Après cette définition de type, l'identificateur **`fiche`** représente un type (et non une variable). On peut ensuite déclarer une variable du type **`fiche`** avec la syntaxe

```
   fiche f;       // f  est une variable de type fiche
   f.age = 15;    // on accède au champ age de la variable f
```

- **la notion de variable pointeur**

```
int var = 12;     // var est une variable de type entier
int * ptr;        // p est une variable de type pointeur sur entier
ptr = &var;       // on affecte à ptr l'adresse de var
*ptr = 14;        // l'opérateur * (opérateur d'indirection) désigne la variable de type
                  // int dont l'adresse  figure dans le pointeur ptr.
                  // Ici, cette variable est donc var.
```

Une variable *pointeur* est une variable qui permet de stocker l'adresse d'un objet. L'opérateur d'indirection permet de désigner l'objet dont l'adresse figure dans une variable pointeur.

Remarque (l'opérateur ->) **:** lorsqu'un pointeur est initialisé avec l'adresse d'une structure, l'opérateur **->** a un sens particulier.

```
fiche f;             // f  est une variable de type fiche
f.age=15;
fiche * ptr = &f ;   // ptr est un pointeur sur une fiche, initialisé avec l'adresse de f

ptr->age=16;         // équivalent à (*ptr).age = 16.
                     // équivaut ici à f.age = 16
```

• les structures de contrôle du C.

La structure **if(…){…}else{…}**

```
int entier;
printf("Saisir un entier:\n");
scanf("%d",&entier);

if((entier%2)==0)
{
   printf("Entier pair");
}
else
{
   printf("Entier impair");
}
```

La structure **while(…){…}** ou **do{…}while(…);**

```
int entier;

do
{
   printf("Saisir un entier impair :\n");
   scanf("%d",&entier);
}
while((entier%2)==0);      //boucle tant que entier pair
```

La structure **for(… ;… ;…){…}**

```
int tab[10];   // réserve un tableau de 10 entiers (non initialisés)
int indice;
tab[0]=0;      // initialise le premier élément à 0

for(indice=1;indice<10;indice++)
{
   tab[indice]=tab[indice-1]+indice;   // tab[indice]=indice + (indice-1)+...
}
```

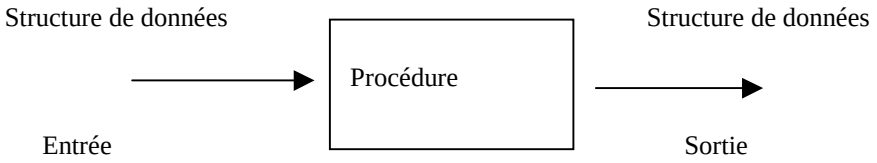
1.2 Introduction aux concepts de la Programmation Orientée Objet

Il convient tout d'abord de se rappeler en quoi consiste la programmation dite *procédurale* pour comprendre ce qui diffère en *Programmation Orientée Objet* (POO).

1.2.1 La programmation procédurale

Avec des langages tels que le C ou le Pascal, la résolution d'un problème informatique passe généralement (mais pas nécessairement) par l'analyse descendante[1]. On décompose ainsi un programme en un ensemble de sous-programmes appelés *procédures* qui coopèrent pour la résolution d'un problème.

Les procédures et fonctions sont généralement des outils qui produisent et/ou modifient des structures de données.



La programmation procédurale suit l'équation de Wirth :

ALGORITHMES + STRUCTURES DE DONNEES = PROGRAMME

Autrement dit, on distingue clairement les structures de données des algorithmes qui les manipulent.

Inconvénient de la programmation procédurale

L'évolution d'une application développée suivant ce modèle n'est pas évidente. En effet, la moindre modification des structures de données d'un programme conduit à la révision de toutes les procédures manipulant ces données. En outre, pour de très grosses applications, le simple fait de répertorier toutes les fonctions manipulant une structure de données peut déjà être problématique.

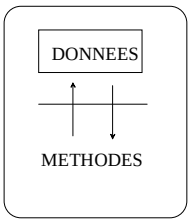
1.2.2 Les idées de la POO

1.2.2.1 Objet = méthodes + données

L'approche objet consiste à mettre en relation directement les structures de données avec les algorithmes qui les manipulent. Un objet regroupe à la fois des *données* et des *algorithmes* de traitement de ces données. Au sein d'un objet, les algorithmes sont généralement appelés des *méthodes.*

Ainsi, par analogie avec l'équation de Wirth, on peut dire que la POO suit l'équation

METHODES + DONNEES = OBJET



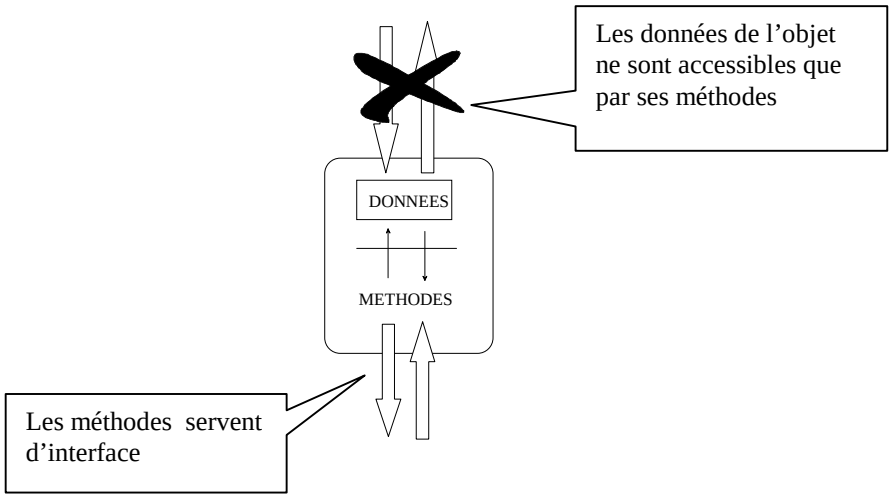
Premier avantage

On maîtrise plus facilement l'incidence d'une évolution de la structure de données d'un objet. Les répercussions principales se situent au niveau des méthodes de l'objet.

[1] L'analyse descendante est en quelque sorte la mise en pratique du Discours de la Méthode de Descartes appliqué aux algorithmes. Elle consiste à décomposer un problème en sous-problèmes et à décomposer ceux-ci, eux-mêmes, en sous-problèmes, et ainsi de suite jusqu'à descendre au niveau des actions dites primitives.

1.2.2.2 L'encapsulation des données

Au sein d'un objet, les données sont protégées. Celles-ci ne sont pas accessibles directement depuis l'extérieur de l'objet. L'accès aux données d'un objet se fait toujours *via* les méthodes de l'objet qui, de ce fait, jouent le rôle d'*interface* entre les données (internes) et l'extérieur.



Cette protection des données au sein d'un objet s'appelle l'*encapsulation de données*. En raison de cette encapsulation des données, on peut dire qu'un objet réalise une « abstraction de données » dans ce sens qu'un utilisateur de l'objet ne peut pas savoir précisément comment les données sont implémentées.

A quoi sert l'encapsulation de données ?

L'encapsulation assure un meilleur contrôle des structures de données et de leur intégrité. Pour expliquer ceci, on va tout d'abord analyser un exemple en C.

```
#include <iostream.h>
void main()
{
   int tab[4]={1,2,3,4};
   char chaine1[8]="abcdefg";
   chaine1[12]='a';
   for(int i=0;i<4;i++)
   {
      printf("%d ",tab[i]);
   }
}
```

Résultat

```
1  97  3  4
```

On réserve ici deux tableaux : le premier de 4 entiers intialisés, le second de 8 caractères également intialisés. Dans cet exemple, rien ne nous empêche d'écrire la ligne suivante.

```
chaine1[12]='a';
```

Et pourtant, il est évident que cette chaîne ne peut pas stocker plus de 8 caractères ! Autrement dit, le langage C nous autorise à écrire n'importe où en mémoire, dans le voisinage du tableau `chaine1[]`. En l'occurrence, on écrit ici directement dans le tableau d'entiers `tab[]`. Lorsque l'on écrit à l'emplacement `chaine1[12]`, on modifie en fait l'entier `tab[1]` (Il faut noter que 97 est le code ASCII de la lettre 'a'). En langage C, c'est à la charge du programmeur de gérer les indices correctement pour que ceux-ci ne dépassent pas la taille du tableau initialement réservé.

Grâce à l'encapsulation, l'approche objet palie ce problème en implémentant les tableaux de la manière suivante :

Données encapsulées :

`Ptr`	adresse de base d'une zone de stockage d'une taille donnée
`N`	taille de la zone de stockage

Méthodes :

`Ecriture(valeur,i)`	méthode d'écriture d'une valeur dans le tableau à l'indice i
`Lecture(i)`	méthode de lecture de l'élément d'indice i

Puisqu'en programmation objet l'accès aux données se fait via l'interface de l'objet (ses méthodes), il suffit ici de vérifier dans chacune des méthodes de lecture et d'écriture que l'indice `i` est bien inférieur à `N` pour autoriser l'accès aux données du tableau. Grâce à ce filtre, l'utilisateur de l'objet ne peut pas écrire n'importe où dans le voisinage proche de la zone mémoire réservée pour le tableau.

1.2.2.3 La classe

La classe est un *modèle* pour générer des objets. Une classe permet de spécifier, pour tous les objets faits à partir de cette classe, comment les données sont structurées et quelles méthodes permettent de manipuler ces données. Ensuite, la valeur des données peut évidemment différer d'un objet à l'autre d'une même classe.

La notion de *classe* généralise ainsi la notion de *type* et la notion d'*objet* généralise celle de *variable*. On dira également qu'un objet est une *instance* d'une classe ou, plus simplement, qu'un objet est une variable d'une classe.

1.2.2.4 L'héritage

L'*héritage* correspond à la possibilité de créer une classe d'objets, dite *classe dérivée*, à partir d'une classe existante, dite *classe de base*.

Dans le processus d'héritage, la classe dérivée « hérite » des caractéristiques (structures de données + méthodes) de la classe de base. En plus de cet héritage, on peut ajouter dans la classe dérivée des données et des fonctionnalités nouvelles. L'héritage conduit ainsi à une spécialisation de la classe de base.

Cette caractéristique des langages objets améliore la *réutilisabilité logicielle*. En effet, l'expérience montre qu'il est plus facile de réutiliser des classes que de réutiliser des bibliothèques de fonctions qui ont, la plupart du temps, été écrites pour des cas précis, et donc des structures de données bien particulières.

1.3 Exemples de classes en C++

On anticipe un peu sur les chapitres suivants pour pouvoir illustrer ce qui a été présenté dans ce premier chapitre.

1.3.1 Une classe `point`

```
// point.h

#include<stdio.h>

class point
{
private :
   float x;
   float y;
public:
   void  Init(float abs,float ord){ x = abs;    y = ord;}
   float GetX()   {  return x;   }
   float GetY()   {  return y;   }
   void  Affiche(){  printf("( %f , %f )\n", x , y ); }
};
```

Utilisation de la classe :

```
#include "point.h"

void main()
{
   point p1;
   p1.Init(10,20);
   p1.Affiche();
   printf("abscisse de p1 : %f\n", p1.GetX());
}
```

Sortie écran

```
(10,20)
abscisse de p1 : 10
```

Ce premier exemple en C++ définit une classe appelée `point`. Les objets de cette classe représentent tout simplement des points du plan $\mathbb{R}^2$.

Les données membres : les champs `x` et `y` de type `float` représentent les données membres des objets de cette classe. Ces champs permettront simplement de stocker des coordonnées. L'attribut `private` indique que ces champs ne sont pas accessibles depuis l'extérieur de l'objet (les données sont encapsulées).

Les méthodes : les fonctions `Init(), GetX(), GetY()` et `Affiche()` sont les méthodes de cette classe. Ces fonctions joueront le rôle d'interface.

`Init()` : permet l'initialisation des coordonnées d'un point
`GetX(), GetY()` : retournent respectivement l'abscisse et l'ordonnée d'un point
`Affiche()` : permet l'affichage à l'écran d'un point sous la forme « (x,y) »

L'accès à un membre d'un objet se fait par l'opérateur « . », comme pour les types structurés. Par exemple, la syntaxe

```
p1.Affiche();
```

correspond à l'appel de la méthode `Affiche()` sur l'objet `p1`, ce qui permet à l'objet `p1` de « s'afficher » à l'écran. Cela revient en quelque sorte à envoyer le message « *Affiche-toi* ! » à l'objet `p1`.

Remarque : ce changement de rôle est toujours déroutant au début. En C, on aurait écrit une fonction `Affiche(point p)` qui reçoit l'objet `p1` comme argument. Ici, c'est l'inverse. C'est au sein même de l'objet que le mécanisme d'affichage est prévu.

Notons que les données sont déclarées privées, la ligne suivante n'est donc pas compilée avec succès :

```
p1.x = 12;
```

Il n'est pas possible d'atteindre directement les champs privés. Dans cet exemple, les modifications des champs `x` et `y` ne peuvent donc s'effectuer que par une seule méthode : la méthode `Init()`.

1.3.2 Une seconde classe `point`

Spécification : on souhaite cette fois-ci disposer d'une classe de points pour laquelle tous les objets ont des coordonnées *entières* et uniquement dans l'ensemble $[0, 1023] \times [0,767]$. Cette classe nous permettra de représenter les pixels d'un moniteur vidéo.

L'encapsulation autorise un meilleur contrôle des données. On montre dans l'exemple suivant quel traitement effectuer pour s'assurer que les données restent toujours conformes à la spécification.

Définition de la classe `point` :

```
//point.h

#include<stdio.h>
class point
{
private:
   int x;
   int y;
public:
   void Init(int abs,int ord)
   {
      if(abs >= 0 && abs<1024) x = abs;  // contrôle de la validité de l'abscisse
      else
      {
         if(abs>=1024) x=1023;
         else x= 0;
      }
      if(ord >= 0 && ord<768) y = ord; // contrôle de la validité de l'ordonnée
      else
      {
         if(ord>=768) y=767;
         else y= 0;
      }
   }
   void Affiche(){ printf("(%d,%d)\n",x,y); }
};
```

Utilisation de la classe `point` :

```
#include "point.h"

void main()
{
   point p1,p2,p3;

   p1.Init(45,89);
   p2.Init(2045,96);

   p1.Affiche();
   p2.Affiche();
}
```

Sortie écran

```
(45,89)
(1023,96)
```

La méthode `Init()`, qui est la seule méthode permettant la modification des données de l'objet, réalise un filtre interdisant à l'utilisateur de la classe de corrompre les données de l'objet. Même si l'utilisateur de la classe cherche à initialiser un point avec des coordonnées non valides, la méthode réalise une troncature des coordonnées.

2 EXTENSIONS AU LANGAGE C

Nous présentons dans ce chapitre les nouveautés apportées par le C++ en dehors des objets. Certaines d'entre elles améliorent la facilité d'utilisation du langage et peuvent justifier à elles-seules l'utilisation du C++ à la place du C. L'aspect objet du langage sera abordé ultérieurement.

2.1 Les opérations d'entrées/sorties

En C, il existe une grande variété de fonctions de lecture et d'écriture d'informations :

. pour la lecture : `scanf(), gets(), cgets(), getchar(), getch(), getche()`…

. pour l'écriture : `printf(), puts(), putchar(), putch()` ...

Le C++ offre la possibilité de faire ces opérations à l'aide d'objets appelés *flots*. Le flot `cin` permet de lire les informations provenant du fichier `stdin` (clavier normalement). Le flot `cout` permet d'écrire des informations sur le fichier `stdout` (écran normalement).

2.1.1 Le flot `cout`

Le flot `cout` permet d'écrire des expressions de n'importe quel type comme le fait `printf()` mais sans avoir à préciser de format d'écriture. La syntaxe est la suivante :

```
cout << expr1 << expr2 << ... << exprn ;
```

pour écrire les expressions *expri*. Le flot `cout` est défini dans le fichier `iostream.h` qui devra donc faire l'objet d'un `#include`.

```
#include <iostream.h>           /* nécessaire pour gérer les flots cin et cout */
#include <conio.h>

void main(void)
{
   int i=-3;
   unsigned int ui=4;
   char c='A';
   char chaine[10]="Martin";
   long l = 100000;
   unsigned long ul=111111;
   float f=3.5;
   double d=4.0;
   long double ld=165.7e+10;
   int *adr=&i;

   clrscr();
   cout << "Valeur de i :" << i << "\n";
   cout << "Valeur de ui :" << ui << "\n";
   cout << "Valeur de c :" << c << "\n";
   cout << "Valeur de chaine :" << chaine << "\n";
   cout << "Valeur de l :" << l << "\n";
   cout << "Valeur de ul :" << ul << "\n";
   cout << "Valeur de f :" << f << "\n";
   cout << "Valeur de d :" << d << "\n";
   cout << "Valeur de ld :" << ld << "\n";
   cout << "Valeur de adr :" << adr << "\n";
}
```

Résultats (sortie stdout)

```
Valeur de i :-3
Valeur de ui :4
Valeur de c :A
Valeur de chaine :Martin
Valeur de l :100000
Valeur de ul :111111
Valeur de f :3.5
Valeur de d :4
Valeur de ld :1.657e+12
Valeur de adr :0x8ed1fff4
```

Exemple 1. Utilisation du flot `cout`.

2.1.2 Le flot `cin`

Le flot `cin` permet de lire des données de tout type comme le fait `scanf()`, avec cependant les différences suivantes:

. Il n'y a pas lieu de préciser de format de lecture
. Il n'y a pas lieu de donner l'adresse de la variable lue.

La syntaxe est la suivante :

```
cin >> var1 >> var2 >> ... >> varn ;
```

pour lire *n* données et les affecter aux *n* variables *vari*.

```
#include <iostream.h>           /* nécessaire pour gérer les flots cin et cout */
void main(void)
{
   int i;
   unsigned int ui;
   char c;
   char chaine[10];
   long l;
   unsigned long ul;
   float f;
   double d;
   long double ld;
   cout << "Valeur de i (int) :";
   cin >> i;
   cout << "Valeur de ui (unsigned) :";
   cin >> ui;
   cout << "Valeur de c (char) :";
   cin >> c;
   cout << "Valeur de chaine (char *) :";
   cin >> chaine;
   cout << "Valeur de l (long) :";
   cin >> l;
   cout << "Valeur de ul (unsigned long) :";
   cin >> ul;
   cout << "Valeur de f (float) :";
   cin >> f;
   cout << "Valeur de d (double) :";
   cin >> d;
   cout << "Valeur de ld (long double) :";
   cin >> ld;
   cout << "             Valeurs lues \n\n\n";
   cout << "Valeur de i :" << i << "\n";
   cout << "Valeur de ui :" << ui << "\n";
   cout << "Valeur de c :" << c << "\n";
   cout << "Valeur de chaine :" << chaine << "\n";
   cout << "Valeur de l :" << l << "\n";
   cout << "Valeur de ul :" << ul << "\n";
   cout << "Valeur de f :" << f << "\n";
   cout << "Valeur de d :" << d << "\n";
   cout << "Valeur de ld :" << ld << "\n";
}
```

Résultats

```
Valeur de i (int) :-5
Valeur de ui (unsigned) :40
Valeur de c (char) :B
Valeur de chaine (char *) :CHARON
Valeur de l (long) :100000
Valeur de ul (unsigned long) :2222222
Valeur de f (float) :4.5e+10
Valeur de d (double) :5e+50
Valeur de ld (long double) :6e+100


        Valeurs lues


Valeur de i :-5
Valeur de ui :40
Valeur de c :B
Valeur de chaine :CHARON
Valeur de l :100000
Valeur de ul :2222222
Valeur de f :4.5e+10
Valeur de d :5e+50
Valeur de ld :6e+100
```

Exemple 2. Utilisation du flot `cin`.

2.2 Nouvelle forme de commentaire

Le commentaire s'écrit en C sous la forme

```
/*  commentaire .... */
```

En C++, un commentaire en fin de ligne peut être écrit sous la forme :

```
// commentaire ...
```

A la rencontre du double caractère //, le compilateur considère le reste de la ligne comme un commentaire.

2.3 Emplacement des déclarations et portée des variables

La déclaration d'une donnée en C++ peut se faire n'importe où dans le programme pour peu qu'elle soit faite avant l'utilisation de la donnée à déclarer. Sa *portée* est celle du bloc où se trouve la déclaration.

```
#include <iostream.h>

void main(void)
{
   int x=30;
   int y=40;

   cout << x << y << endl;

      {
         int aux = x; // la variable aux n'est définie qu'à l'intérieur de ce bloc
         x = y;
         y = aux;
      }

   cout << x << y << endl;

   cout << aux << endl;    // << cette ligne produirait une erreur de compilation

}
```

Exemple 3. Déclaration et portée des variables en C++

Note : le modificateur de flot `endl` produit un passage au début de la ligne suivante (comme `cout << "\n" ;`)

Commentaire sur l'exemple 3 :

. les variables `x` et `y` sont locales à la fonction `main()`. Leur portée va du début de la fonction `main()` jusqu'à la dernière accolade.
. la variable `aux` est déclarée dans le bloc allant de la deuxième accolade «{ » à la première accolade « }». Sa portée est donc limitée à ce bloc. En dehors de ce bloc, la variable aux n'existe pas.

2.4 La notion de référence

En C++, la notion de *référence* consiste à pouvoir donner plusieurs identificateurs (noms) à une même variable. Cette caractéristique, présente également dans la langage PASCAL, facilitera par la suite la définition des paramètres de sortie (c'est-à-dire modifiés), ou des paramètres d'entrée/sortie (c'est-à-dire utilisés puis modifiés), pour les fonctions.

2.4.1 La référence comme nouveau type de données

La notation

```
T var1;
T &var2 = var1;
```

fait de `var2` une *référence* à l'objet `var1` de type `T`. Cela signifie que les identificateurs `var1` et `var2` désignent le même objet. Manipuler l'un, revient à manipuler l'autre. On peut dire que les identificateurs `var1` et `var2` sont des synonymes, ou bien que `var2` est un alias de `var1`.

L'initialisation d'une référence lors de sa déclaration est obligatoire (dans la mesure où c'est l'alias d'une variable existante)

```
#include <iostream.h>

void main(void)
{
   int I=3;
   int &refI = I;    // refI est une référence à I

   cout << "I = " << I << endl;
   cout << "refI = " << refI << endl;

   cout << "adresse de I : " << &I << endl;
   cout << "adresse de refI : " << &refI << endl;

   cout << "Incrément de I (I++)" << endl;
   I++;
   cout << "refI = " << refI << endl;

   cout << "Incrément de refI (refI++)" << endl;
   refI++;
   cout << "I = " << I << endl;
}
```

Résultats

```
I = 3
refI = 3
adresse de I : 0x0065FDF4
adresse de refI : 0x0065FDF4

Incrément de I (I++)
refI = 4
Incrément de refI (refI++)
I = 5
```

Exemple 4. Notion de référence en C++

Commentaire sur l'exemple 4 :

. on constate que les variables `I` et `refI` ont la même adresse**.** Elles désignent donc bien le même objet.
. puisque `I` et `refI` sont des synonymes, modifier l'un revient à modifier l'autre**.**

2.4.2 Transmission d'arguments par référence

L'utilisation de la notion de référence faite dans l'exemple 4 ne présente pas, sous cette forme, d'intérêt particulier. L'utilisation la plus courante en C++ concerne l'échange d'informations avec une fonction. Cette partie du cours est essentielle et doit faire l'objet d'une attention toute particulière

Rappels sur la tansmission d'arguments en langage C

En langage C, il n'y a qu'une façon de transmettre des informations à une fonction : la copie. Le paramètre formel d'une fonction peut être considéré comme une variable locale à la fonction, différente du paramètre d'appel, et qui prend pour valeur, au début de l'exécution de la fonction, celle du paramètre d'appel. L'exemple (classique) suivant illustre cette particularité.

```
#include <iostream.h>
         void f(int);
void main(void)
{
   int pAppel=5;
   cout << "pAppel = " << pAppel << endl;
   cout << "f(pAppel)" << endl;

   f(pAppel);
   cout << "Après la fonction… " << endl ;
   cout << "pAppel = "<< pAppel << endl;
}

void f(int pFormel)
{
   cout << "pFormel = "<< pFormel << endl;
   cout << "Incrément de pFormel " << endl;
   pFormel++;
   cout << "pFormel = "<< pFormel << endl;
}
```

Résultats

```
pAppel = 5
f(pAppel)
pFormel = 5
Incrément de pFormel
pFormel = 6
Après la fonction…
pAppel = 5
```

Exemple 5. La transmission d'arguments en langage C.

Commentaire sur l'exemple 5 :

La variable `pAppel` désigne le paramètre avec lequel on appelle la fonction dans le programme principal et `pFormel` représente formellement le paramètre dans la fonction.

Au moment de l'appel de la fonction `f(pAppel)`, le paramètre formel reçoit une *copie* du paramètre d'appel. On peut considérer qu'il se passe une déclaration de variable locale du style : `int pFormel = pAppel.` Dans notre exemple, au début de l'exécution de la fonction, `pFormel` contient donc la valeur 5. On peut d'ailleurs incrémenter `pFormel` au sein de la fonction et constater son évolution. Malgré tout, cette modification du paramètre formel n'a aucune incidence sur le paramètre d'appel. Par conséquent, après l'appel de la fonction, la variable `pAppel` contient toujours la valeur qu'elle avait avant l'appel de la fonction.

D'un point de vue algorithmique, cette caractéristique ne pose aucun problème pour les procédures ayant des paramètres d'entrée[2]. En revanche, lorsqu'une procédure a des paramètres de sortie, la technique appelée souvent *passage par adresse* s'impose. Lorsqu'une fonction du C doit modifier une variable, on adopte la convention suivante :

. côté programme principal , on passe à cette fonction (une copie de) l'*adresse* de cette variable.

. côté fonction, puisqu'on récupère une adresse, on sait modifier la variable que cette adresse désigne grâce à l'opérateur d'indirection `*`.

Par exemple, si l'on souhaite que la fonction `f()` de l'exemple 5 modifie la variable `pAppel`, on adopte la démarche suivante :

. l'argument de la fonction doit alors être de type pointeur sur entier : le prototype devient `void f(int *)`

. lors de l'appel de la fonction, on ne transmet pas `pAppel` mais l'adresse de `pAppel`, c'est-à-dire `&pAppel`.

[2] On appelle

- *paramètre d'entrée* (ou donnée), d'une fonction ou d'une procédure, un paramètre dont la valeur est utilisée par la procédure mais n'est pas modifiée par celle-ci.
- *paramètre de sortie* (ou résultat) un paramètre dont la valeur n'est pas utilisée par la procédure mais qui contient, en sortie de la procédure, une valeur produite par la procédure.
- *paramètre d'entrée/sortie*, un paramètre dont la valeur est utilisée par la procédure, par exemple pour initialiser unn calcul, puis finalement éventuellement modifiée pour contenir une nouvelle valeur.

Cette convention conduit à l'exemple 6.

```
#include <iostream.h>

   void f(int *);

void main(void)
{
   int pAppel=5;
   cout << "pAppel = " << pAppel << endl;
   cout << "f(&pAppel)" << endl;

   f(&pAppel);       // l'appel est modifié
   cout << "Après la fonction... " << endl ;
   cout << "pAppel = "<< pAppel << endl;
}

void f(int * pFormel)
{
   cout << "pFormel = "<< pFormel << endl;
   cout << "variable désignée par pFormel = "<< *pFormel << endl;
   cout << "Incrément de la variable pointée par pFormel " << endl;
   (*pFormel)++;
   cout << "variable désignée par pFormel = "<< *pFormel << endl;
}
```

Résultats

```
pAppel = 5
f(&pAppel)
pFormel = 0x0065FDF4
variable désignée par pFormel=5
Incrément de la variable pointée par
pFormel
variable désignée par pFormel=6
Après la fonction…
pAppel = 6
```

Exemple 6. Le passage par adresse.

Intérêt des références dans la transmission d'arguments

La notion de référence va nous affranchir, dans beaucoup de cas, de la technique de transmission par adresse rappelée précédemment. Examinons l'exemple suivant :

```
#include <iostream.h>

         void f(int&);
void main(void)
{
   int pAppel=5;
   cout << "pAppel = " << pAppel << endl;
   cout << "f(pAppel)" << endl;

   f(pAppel);

   cout << "Après la fonction... " << endl ;
   cout << "pAppel = "<< pAppel << endl;
}
void f(int &pFormel)
{
   cout << "pFormel = "<< pFormel << endl;
   cout << "Incrément de pFormel " << endl;
   pFormel++;
   cout << "pFormel = "<< pFormel << endl;
}
```

Résultats

```
pAppel = 5
f(pAppel)
pFormel = 5
Incrément de pFormel
pFormel=6
Après la fonction…
pAppel = 6
```

`pFormel` est une référence au paramètre d'appel.

Exemple 7. Passage par référence.

Commentaire sur l'exemple 7 :

Il convient de remarquer la syntaxe au niveau du prototype de la fonction :

```
void f(int &);
```

Par cette syntaxe, on indique que le paramètre de cette fonction est du type référence à un entier. Autrement dit, le paramètre formel `pFormel` de la fonction `f()` sera un alias du paramètre d'appel. Dans ce cas, manipuler le paramètre formel `pFormel` aura désormais une incidence sur le paramètre d'appel, puisqu'il s'agit précisément du paramètre d'appel, mais avec un identificateur différent.

Par la suite, on appellera ce mode de transmission *passage par référence*. Au moment du passage d'argument, le passage par référence de l'exemple 7 peut s'interpréter de la manière suivante : lorsque l'on écrit `f(pAppel)`, on peut imaginer qu'au sein du bloc de la fonction, on définit une nouvelle référence de la manière suivante :

```
int & pFormel = pAppel;
```

Le passage d'argument par référence autorise donc la modification du paramètre d'appel au sein de la fonction. Ce mode de passage convient donc aux paramètres de sortie et aux paramètres d'entrée/sortie. En outre, l'utilisation d'une référence est moins lourde que la technique du passage d'argument par adresse du C.

2.4.3 Fonction retournant une référence

En C++, une fonction peut retourner une référence à une variable. Il s'agit d'une convention d'écriture. Dans l'exemple 8, la syntaxe `float & Element(float *, int)` signifie que la fonction retourne une référence à une variable de type `float`.

```
#include <iostream.h>
   float & Element(float*, int);

void main(void)
{
   float tab1[5]={1.2, 2, 3.8, 4.5, 5.7};

   // affichage de tab1[]
   cout << "tab1[]=";
   for(int i=0;i<5;i++) cout << tab1[i] <<" ";
   cout << endl;

   cout << "Element(tab1,2) ="<< Element(tab1,2)<<endl;

   // Modification du tableau via la référence retournée par la fonction
   Element(tab1,3)=5;

   // affichage de tab1[]
   for(i=0;i<5;i++) cout << tab1[i] <<" ";
   cout << endl;
}

float &  Element(float * tableau,int indice)
{
   return tableau[indice];
}
```

Résultats

```
tab1[]= 1.2 2 3.8 4.5 5.7
Element(tab1,2) = 3.8
tab1[]= 1.2 2 3.8 5 5.7
```

Exemple 8. Renvoi d'une référence.

Commentaire sur l'exemple 8 :

La fonction `Element()` reçoit deux arguments : l'adresse d'un tableau d'éléments de type `float` et un entier qui servira d'indice dans le tableau. Cette fonction retourne une référence à la variable `tableau[indice]`. Qu'est-ce que cela signifie exactement ?

Lorsque dans le programme on écrit

```
Element(tab1,3)=5;
```

`Element(tab1,3)`, qui représente ce que renvoie la fonction, est une référence à `tab1[3]`. Autrement dit, si l'on modifie `Element(tab1,3)`, on modifie par conséquent l'élément `tab1[3]`, c'est-à-dire le contenu du tableau. Puisque `Element(tab1,3)` est une référence à une variable, on peut placer cette expression à gauche d'une affectation. Il faut admettre que c'est d'ailleurs assez déroutant que l'appel d'une fonction soit à gauche de l'affectation !

Attention : on ne peut retourner une référence à une variable qu'à la condition que la variable existe à l'extérieur du bloc de la fonction.

Le compilateur MS Visual C++ produit une mise en garde (justifiée) lors de la compilation du source suivant.

```
#include <iostream.h>

   int & f();

void main(void)
{
      f()=3;
}

int &  f()
{
   int A=12;
   return A;   // << c'est sur cette ligne que le compilateur génère un warning !!
}
```

Exemple 9. Une fonction ne doit pas retourner une référence à une variable locale.

Le compilateur indique : `warning C4172 : returning address of local variable or temporary`

Explication : la variable `A` de type entier est locale à la fonction `f()` (cf. la portée des variables). Autrement dit, en dehors du corps de la fonction, cette variable n'existe plus.

Par conséquent, lorsque l'on écrit

```
f()=3;
```

on tente de modifier une variable qui n'existe plus, ou plus très longtemps ! L'erreur de programmation qui consiste à retourner une référence à une variable locale à la fonction conduit à des comportements très étranges lors de l'exécution.

2.5 Arguments d'une fonction avec valeur par défaut

En C, l'appel d'une fonction doit comporter autant de paramètres effectifs qu'il y a de paramètres formels dans la définition de la fonction. En C++, certains paramètres effectifs peuvent être omis. Dans ce cas, les paramètres formels de la fonction prennent une *valeur par défaut*.

```
#include <iostream.h>

   float Addition(float, float =3);

void main(void)
{

   cout << Addition(10,5) << endl;

   cout << Addition(7) << endl;

}

float Addition(float a, float b)
{
   return a+b;
}
```

le second argument vaut 3 s'il fait défaut

Résultats

```
15
10
```

Exemple 10. Arguments avec valeur par défaut.

En cas d'absence d'un second paramètre d'appel pour la fonction `Addition()`, le second argument prend pour valeur 3.

Note : les arguments ayant des valeurs par défaut doivent tous figurer en fin de liste des arguments. Voici quelques exemples, bons et mauvais, de prototypes :

```
float Addition(float=2, float =3);       // OK
float Addition(float=2, float);          // NON !!!
float Addition(float=1, float, float =3);// NON PLUS !!!
float Addition(float, float=5, float =3);// OK
```

Remarque : la valeur qu'un paramètre doit prendre par défaut doit figurer au niveau du prototype mais pas au niveau de l'en-tête de la fonction.

2.6 Surcharge (ou surdéfinition) de fonctions

Définition (Signature) : on appelle **signature** d'une fonction la combinaison de sa classe (si elle est membre d'une classe) de son identificateur et de la suite des types de ses paramètres.

Les 3 fonctions suivantes ont des signatures différentes :

```
float Addition(float);
float Addition(int,float);
float Addition(float,float );
```

Note : le type de la valeur de retour n'appartient pas à la signature.

En C++, il est possible de définir plusieurs fonctions avec le même identificateur (même nom) pour peu qu'elles aient des signatures différentes. On appelle cela *la surcharge de fonction*.

L'exemple suivant présente un programme où deux fonctions `Addition()` sont définies. Il faut noter qu'il s'agit bien de fonctions différentes et qu'elles ne réalisent donc pas le même traitement. Le compilateur réalise l'appel de la fonction appropriée à partir du type des paramètres d'appel.

```
#include<iostream.h>

   int Addition(int, int=4);
   float Addition(float, float =3);

void main(void)
{
   float fA=2.5;
   float fB=3.7;
   int iA=2;
   int iB=3;

   cout << Addition(fA,fB) << endl;
   cout << Addition(iA,iB) << endl;

   cout << Addition(fA) << endl;
   cout << Addition(iA) << endl;
}

float Addition(float a, float b)
{
   return a+b;
}

int Addition(int a,int b)
{
   const int offset=12;
   return a+b+offset;
}
```

Résultats

```
6.2
17
5.5
18
```

Exemple 11. Surcharge de fonctions.

2.7 Les opérateurs de gestion de mémoire

En C, on demande de la mémoire avec la fonction `malloc()` et on la libère avec la fonction `free()`. En C++, on peut faire la même chose avec deux opérateurs (et non des fonctions) appelés `new` et `delete`.

2.7.1 L'opérateur `new`

La syntaxe d'utilisation de l'opérateur `new` est la suivante :

```
1  new TYPE
2  new TYPE[n]
```

où `TYPE` est un identificateur de type quelconque, `n` est une expression entière quelconque.

La syntaxe 1 réserve de la place pour 1 élément ayant le type `TYPE`, alors que la syntaxe 2 réserve de la place pour n éléments de type `TYPE`. Dans les deux cas, le résultat de l'opération est

. l'adresse de l'espace mémoire réservé sous la forme d'un pointeur de type `(TYPE *)`
. le pointeur `NULL` si l'espace demandé n'a pu être obtenu.

On a donc quelque chose de très proche de ce que fait `malloc()` avec cependant une syntaxe plus simple.

2.7.2 L'opérateur `delete`

La syntaxe d'utilisation de `delete` est la suivante :

```
1  delete pointeur
2  delete [] pointeur
```

où `pointeur` est un pointeur ayant obtenu sa valeur actuelle par `new`.

La syntaxe 1 libère l'espace mémoire pointé par `pointeur`. La syntaxe 2 sert à libérer l'espace occupé par un tableau d'objets. L'intérêt de la seconde syntaxe n'apparaîtra qu'après avoir présenté les constructeurs/destructeurs.

Remarque : le fonctionnement du programme est indéterminé si `pointeur` pointe sur une zone déjà libérée ou si la valeur de `pointeur` a été obtenue autrement que par `new` (`malloc()` par exemple). Il est déconseillé de mélanger l'utilisation des opérateurs `new/delete` avec celle de `malloc()/free()`. Durant ce cours, on utilisera exclusivement les opérateurs de gestion de mémoire `new/delete`.

```
#include <iostream.h>

   void AlloueTableau ( int * & ptr, int taille)
   {
      ptr = new int[taille];  // alloue « taille » entiers et place l'adresse de la zone
                              // dans ptr
   }

void main()
{
   int * ptrInt;
   AlloueTableau( ptrInt, 12 );

   for(int i=0; i<12; i++) ptrInt[i]=i;

   int * ptrTemp = ptrInt;
   AlloueTableau(ptrInt, 5);

   delete ptrInt;
   delete ptrTemp;
}
```

Exemple 12. Les opérateurs de gestion de mémoire.

Commentaire sur l'exemple 12 :

La fonction `AlloueTableau()` réserve une zone d'entiers et place l'adresse de cette zone dans `ptr`. On remarquera que, logiquement, `ptr` est un pointeur sur des éléments de type entier, mais puisque ce pointeur doit être modifié par la fonction (c'est un paramètre de sortie), l'argument `ptr` est passé par référence.

2.8 Incompatibilités entre C++ et C

Un programme écrit en C pur n'est pas toujours accepté en C++. Suivent quelques cas où Turbo C++ de Borland n'accepte pas une forme du langage Turbo C du même éditeur ou l'inverse.

2.8.1 Prototypes de fonctions

C On peut rencontrer dans un programme l'appel à une fonction avant d'avoir rencontré sa définition ou son prototype.
C++ Ce n'est plus possible

Le compilateur C rencontre l'appel à `printf()` sans avoir rencontré sa définition ou son prototype, et ne peut alors vérifier la validité de l'appel. Cependant aucune erreur n'est signalée. En C++, cela n'est plus autorisé afin que le compilateur puisse vérifier la validité de l'appel. Le prototype de `printf()` étant dans le fichier `stdio.h`, on aura en C++ le programme suivant :

```
// en C++
#include <stdio.h>      // obligatoire pour prototype de printf

void main(void)
{  .....
   printf("coucou");
   .....
}
```

2.8.2 Conversion d'un pointeur de type `void *`

En **C,** la valeur d'un pointeur de type `void*` peut être affectée directement à un pointeur de type `T*` différent. En C++, ce n'est plus autorisé et il faut convertir explicitement le pointeur de type `void*` en un pointeur de type `T*`.

```
/* en langage C */
#include <alloc.h>
void main(void)
{
   int *tableau;
   tableau=malloc(100);
}
```

La fonction `malloc()` rend un pointeur de type `void *` qui est affecté à un pointeur de type `int *`. Aucune erreur n'est signalée.

En C++, une erreur est signalée et le programme doit être corrigé comme suit :

```
// en C++
#include <alloc.h>
void main(void)
{ int *tableau;
  tableau=(int *)(malloc(100));    // conversion explicite
}
```

L'écriture `(int *)(malloc(100))` permet une conversion du type de retour de `malloc()`, c'est-à-dire `void *`, vers le type `int*`. Une telle conversion explicite d'un type donné vers un autre type est également appelée *transtypage* ou *cast.*

2.8.3 Expressions constantes

En langage C, une variable déclarée constante par le modificateur `const`, ne peut pas servir à dimensionner un tableau. En C++, c'est rendu possible.

```
/* en C */
#include <stdio.h>

void main(void)
{ const N=100;
  const P=2*N+1;
  int tableauN[N],   << Erreur lors de la compilation
      tableauP[P];   << Erreur lors de la compilation
}
```

Ce programme génère une erreur pour chacune des deux déclarations de tableaux. Le même programme en C++ est accepté.

Remarque : en C, `tableauN` pourra être dimensionné par `N` si cet identificateur fait l'objet d'un `#define N 100`. Il n'y a pas de solution pour `tableauP`, si ce n'est de dire explicitement que `P` vaut 201 à l'aide d'un `#define P 201`.

2.8.4 Identification d'une structure

Considérons le programme suivant :

```
void main(void)
{
   struct chaine
   {
      char *valeur;
      unsigned taille;
   };

  chaine nom;
}
```

Ce programme est accepté par le compilateur C++ mais pas par le compilateur C. En **C++**, le type **`struct chaine`** précédent peut être identifié par `struct chaine` ou plus simplement par `chaine`. Ce n'est pas le cas en C, où il faudrait écrire :

```
struct chaine nom;
```

2.9 Le modificateur `const` et les références constantes

2.9.1 Le modificateur `const`

On verra au fil du cours que le modificateur `const` est, pour diverses raisons qu'on ne peut pas encore exposer, d'emploi très fréquent en C++.

La première utilisation de ce modificateur apparaît, comme nous l'avons déjà vu, lors de la déclaration de *constantes* dans un programme. Ces constantes sont souvent des valeurs scalaires, mais peuvent également être des tableaux ou des structures.

A quoi correspondent ces constantes ? Elles représentent des éléments de programmation dont la valeur est fixée pour l'ensemble de la durée d'exécution du programme (on peut penser par exemple à la valeur de π).

A quoi cela sert-il ? Le compilateur empêche toute détérioration de la valeur d'une constante (normal). On évite ainsi certaines erreurs de programmation. Pour illustrer ceci, examinons le programme de l'exemple 13.

Ce programme est censé demander à l'utilisateur de saisir une lettre différente de la lettre `'n'`. Tant que l'utilisateur persiste à saisir un `'n'`, le programme boucle jusqu'à ce que l'utilisateur saisisse une autre lettre. On déclare donc

une variable `test` qui représente ce fameux caractère `'n'`. Le programme suivant compile parfaitement mais ne s'exécute pas normalement. En effet, il boucle sans fin sur la saisie d'un caractère. Pourquoi ? Parcequ'en C++, comme en C, le test d'égalité s'effectue avec un double == et que dans l'état actuel, il s'agit d'une affectation ! Puisque le caractère saisi sera toujours de valeur numérique non nulle, le programme boucle sans fin.

```
#include<iostream.h>
void main()
{
   char test='n';
   char saisie;
   do
   {
      cout << "saisissez une lettre différente de n : "<<endl;
      cin>>saisie;
   }while(test=saisie);    //erreur classique de programmation
}
```

Exemple 13. Erreur classique : confusion de l'affectation = avec le test d'égalité ==

Une attitude de programmation différente nous aurait mis la puce à l'oreille. Reprenons ce programme et déclarons la variable `test` comme une constante (ce qui paraît d'ailleurs assez naturel). Que se passe-t-il ?

```
#include<iostream.h>
void main()
{
   const char test='n';
   char saisie;
   do
   {
      cout << "saisissez une lettre différente de n : "<<endl;
      cin>>saisie;
   }while(test=saisie);    << le compilateur indique qu'il est impossible d'affecter
                           << quelque chose à une constante !!!
}
```

Exemple 14. Utilisation d'une constante

Pour ce programme, le compilateur génère un message qui attire notre attention sur cette erreur classique. Cette démarche de déclarer dès que nécessaire un objet comme étant constant évite donc quelques « pièges » du C/C++.

2.9.2 Les références constantes

Considérons l'exemple suivant.

```
#include<iostream.h>
void main()
{
   int val=12;
   int &ref=val;            //référence à val
   const int& constref=val; //référence constante à val
   val++;
   cout << "val = " << val<< " ref= " << ref << " constref = " << constref<< endl;
   ref++;
   cout << "val = " << val<< " ref= " << ref << " constref = " << constref << endl;
   // constref++; << impossible de modifier la variable "val" via sa reference constante
}
```

Résultats

```
val =13 ref =13 constref =13
val =14 ref =14 constref =14
```

Dans le programme précédent, on déclare tout d'abord une variable `val` de type entier. Il faut bien comprendre que c'est d'ailleurs la seule variable du programme. Les deux autres déclarations ne sont que des références à cette variable, c'est-à-dire des alias. Cela explique pourquoi l'affichage du triplet `val-ref-constref` fournit systématiquement trois sorties identiques.

La première référence (`ref`) est, comme nous l'avons déjà vu, une référence « classique » à la variable `val`. Il est donc possible de modifier la variable `val` via sa référence (ce qui est fait en exécutant `ref++`).

La seconde référence (`constref`) est appelée *référence constante*. La syntaxe est

```
const int & nom_de_la_reference = nom_de_la_variable,
```

ou, de manière plus générale

```
const TYPE & nom_de_la_reference = nom_de_la_variable.
```

Une référence constante est avant tout une référence, avec en plus la particularité suivante : elle interdit toute modification de la variable à laquelle elle fait référence. Il est donc impossible ici d'écrire ~~`constref++`~~.

A quoi les références constantes servent-elles ?

Leur intérêt apparaît, là encore, dans le cadre du passage d'arguments à une fonction. La règle générale à retenir est la suivante : *le passage d'arguments par référence est plus rapide que le passage par valeur.*

```
#include<iostream.h>

struct personne
{
   char nom[40];
   char prenom[40];
   int age;
};

      void affiche(personne);

void main()
{
   personne jean={"Martin","Jean",45},pierre={"Dubecque","Pierre",24};
   affiche(jean);
   affiche(pierre);
}

void affiche(personne p)
{
   cout << p.prenom <<" "<< p.nom <<", " <<p.age << " ans"<<endl;
}
```

Résultats

```
Jean Martin, 45 ans
Pierre Dubecque, 24 ans
```

Exemple 15. Passage d'une structure à une fonction (par copie)

Commentaire : dans cet exemple, le seul argument de la fonction `affiche`() est un paramètre d'entrée. Il peut donc être passé par valeur. En revanche, ce seul argument est une structure contenant au moins 82 octets (80 caractères pour les chaînes et au moins 2 octets pour l'entier). Toute la structure est copiée à chaque appel de telle sorte que le paramètre formel `p` n'est qu'une copie du paramètre d'appel. Il y a donc 82 octets copiés à chaque appel de `affiche()` !

Si l'on adopte un passage par référence, il n'y a plus ce problème de copie.

```
#include<iostream.h>
struct personne
{
   char nom[40];
   char prenom[40];
   int age;
};
      void affiche(personne &);  // passage par référence
void main()
{
   personne jean={"Martin","Jean",45};
   affiche(jean);
}
void affiche(personne & p)
{
   cout << p.prenom <<" "<< p.nom <<", " <<p.age << " ans"<<endl;
}
```

Exemple 16. Passage d'une structure à une fonction (par référence)

Commentaire : cette fois-ci, le paramètre formel `p` n'est pas une copie du paramètre d'appel, mais simplement une référence à ce paramètre d'appel. Il n'y a donc pas duplication des structures `jean` et `pierre` lors des appels. En pratique, la fonction reçoit seulement l'adresse de chacune de ces structures (le passage par référence est en fait un passage par adresse transparent pour le programmeur**).** Le passage par référence est donc plus rapide que le passage par valeur (par copie). On évite ici la copie de 80 octets.

Pourquoi passage par référence constante et pas uniquement passage par référence ?

Dans notre exemple, l'argument de la fonction est un paramètre d'entrée. En effet, la fonction utilise les valeurs des champs de la structure passée en argument, sans les modifier. Et puisque la structure est désormais passée par référence on a, au sein de la fonction affiche, la possibilité de modifier la structure argument. Ceci est à éviter ! En effet, il n'est pas exclu que par inadvertance (pensez à la confusion possible entre l'affectation et le test d'égalité) on modifie le paramètre `p` qui devrait normalement n'être qu'un paramètre d'entrée. Aussi, pour éviter ce genre d'erreur, on peut utiliser le passage d'argument par référence constante.

```
#include<iostream.h>
struct personne{
   char nom[40];
   char prenom[40];
   int age;
};
      void affiche(const personne &);  //passage par référence constante
void main()
{
   personne jean={"Martin","Jean",45};
   affiche(jean);
}

void affiche(const personne & p)
{
   cout << p.prenom <<" "<< p.nom <<", " <<p.age << " ans"<<endl;
}
```

Exemple 17. Passage d'une structure à une fonction (par référence constante)

Dans ce dernier exemple, l'appel de la fonction `affiche(jean)` revient à faire du paramètre formel `p` une référence constante à la structure `jean`. Puisque `p` est une référence constante, toute tentative de modification de `p` au sein de la fonction `affiche()` sera interdite par le compilateur. On ne risque donc plus de modifier la structure (qui ne doit pas l'être) par mégarde sans que le compilateur ne génère une erreur.

Pour résumer, le passage par référence constante

. évite une copie de l'argument lors de l'appel de la fonction
. empêche toute modification de l'argument au sein de la fonction

Ce passage d'argument est donc idéal lorsqu'une fonction reçoit des paramètres d'entrée occupant beaucoup de place (c-à-d beaucoup de mots mémoire). Ce sera donc recommandé lorsque les paramètres d'entrée d'une fonction seront des structures ou des objets avec beaucoup de données internes.

2.10 Conclusion

Qu'avons-nous appris ?

En C, lors de l'appel d'une fonction,

. on passe les paramètres d'entrée par valeur (c'est-à-dire qu'il y a duplication des arguments lors de l'appel).
. on passe les paramètres de sortie par adresse.

En C++,

. on peut passer les paramètres d'entrée par valeur (c'est-à-dire qu'il y a duplication des arguments lors de l'appel) *si ceux-ci n'occupent pas trop de mémoire* (sans quoi la duplication est coûteuse en temps).

. on passe les paramètres d'entrée par référence constante si ceux-ci sont volumineux

. on peut passer les paramètres de sortie par adresse (mais c'est devenu obsolète)

. on passe les paramètres de sortie par référence (non constante) car c'est aussi efficace que le passage par adresse avec une syntaxe plus légère.

3 LA SURCHARGE D'OPERATEURS EN C++

Il est possible, en C++, de surcharger la plupart des opérateurs. Cela signifie qu'on va pouvoir décrire quels traitements doivent réaliser certains opérateurs. Cette surcharge n'est toutefois possible que sur les types créés par le programmeur : il n'est pas possible de redéfinir les opérateurs agissant sur les types élémentaires tels que `int`, `float`, etc.

3.1 La pluralité des opérateurs du C++

Selon le nombre d'opérandes dont l'opérateur a besoin, l'opérateur sera qualifié d'opérateur *unaire* (1 seul opérande) ou d'opérateur *binaire* (2 opérandes).

Par exemple, l'opérateur « = » en C++ est un opérateur binaire. La syntaxe d'utilisation de cet opérateur étant :

```
Op1 = Op2;
```

L' opérateur « ++ » en revanche est un opérateur unaire. En effet, il s'applique à un seul opérande.

```
Op1++;
```

Remarque : il ya en fait deux opérateurs « ++ ». Celui qui pré-incrémente et qui s'utilise `++Op1` et celui qui post-incrémente et qui s'utilise `Op1++`. Idem pour les opérateurs de décrément «--» **.**

Enfin, de même qu'avec l'opérateur « ++ » où le même signe peut représenter deux opérateurs différents (la syntaxe d'utilisation permet au compilateur de les distinguer), certains opérateurs peuvent avoir une version unaire et une version binaire. C'est le cas par exemple de l'opérateur « - ».

```
#include<iostream.h>

void main()
{
   int a=4, b=5;

   cout << (a-b) << endl;     // opérateur - binaire
   cout << -a;                // opérateur - unaire
}
```

La syntaxe **`a-b`** utilise l'opérateur – binaire (soustraction) tandis que **`-a`** utilise l'opérateur unaire (opposé).

3.2 Les opérateurs membres d'une classe

Avant de présenter un tableau récapitulatif des opérateurs que l'on peut surcharger en C++, il faut savoir que tous ne peuvent pas être surchargés de la même manière. Certains ne peuvent être surchargés que s'ils appartiennent à une classe. Il s'agit alors de méthodes d'une classe. Dans la mesure où l'on n'a pas encore présenté les classes, la présentation faite ici n'est que partielle et se raffinera au cours des chapitres suivants.

3.3 Les opérateurs avec traitement par défaut

Lorsque l'on définit de nouveaux types, par exemple des types structurés, certains opérateurs réalisent un traitement par défaut. C'est le cas de l'opérateur « = » et de l'opérateur « & ».

3.3.1 Le traitement par défaut de l'opérateur =

Lorsque l'on définit un nouveau type structuré, le traitement réalisé par défaut pour l'opérateur « = » est une copie des champs d'une structure dans l'autre structure. C'est une copie membre-à-membre.

L'exemple ci-dessous illustre cette caractéristique.

```
#include<iostream.h>

struct S
{
   int a;
   float b;
};

void main()
{

   S x={1,2.4},y={3,4.5};  // x et y deux structures de type S initialisées

   x = y;   // utilise l'opérateur = par défaut

   cout << "x.a = "<<  x.a << " x.b =" << x.b<<endl;
}
```

Résultat

```
x.a = 3 x.b = 4.5
```

3.3.2 Le traitement par défaut de l'opérateur &

Lorsque l'on définit un nouveau type, l'opérateur & (opérateur unaire) retourne l'adresse de l'objet auquel il s'applique.

3.4 Les opérateurs que l'on peut surcharger en C++

Les différents opérateurs que l'on peut surcharger en C++ sont :

Unaire	`+ - ++ -- ! ~ * & new new[] delete (cast)`
Binaire	`* / % + -`
Binaire	`<< >>`
Binaire	`< > <= >= == !=`
Binaire	`& |`
Binaire	`&& ||`
Binaire	`+= -= *= /= %= &= ^= |= <<= >>=`
Binaire	`,`

Doivent être définis comme membres d'une classe :

Binaire	`=`	(affectation)
	`( )`	(opérateur fonction)
	`[ ]`	(opérateur d'indexation)
	`->`	

3.5 Le mécanisme de surdéfinition des opérateurs en C++

Lorsqu'un opérateur est utilisé sur un type défini par le programmeur, l'emploi de cet opérateur est équivalent à l'appel d'une fonction particulière.

Si l'on définit un nouveau type `T`, et deux variables `x` et `y` de type `T`, soit

```
T x,y;
```

les deux syntaxes suivantes sont équivalentes pour le compilateur C++ :

```
(1)   x+y;
(2)   operator+(x,y);
```

Autrement dit, les opérateurs sont vus comme des fonctions avec des identifiants particuliers : `operator` suivi du signe de l'opérateur. Dans le cas précédent, si l'on souhaite que l'opérateur + réalise un traitement particulier sur les variables de type `T`, il suffit de définir une fonction appelée `operator+()` qui accepte deux arguments de type `T`.

3.6 Exemple de surdéfinition d'opérateurs

On définit ici un type `mat2x2` qui est un type structuré représentant des matrices 2 lignes-2 colonnes. En réalité, la structure représente seulement les éléments de la matrice : `Mij` est l'élément de numéro de ligne i et de numéro de colonne j.

3.6.1 Surcharge des opérateurs + et * pour des structures

On va ici surcharger les opérateurs + et * pour qu'ils réalisent les opérations de somme et de produit matriciel sur les matrices de taille 2x2.

```
#include<iostream.h>

struct mat2x2
{
   float M11;
   float M12;
   float M21;
   float M22;
};


      void     affiche(const mat2x2 &);
      mat2x2   operator+(const mat2x2 &,const mat2x2 &);
      mat2x2   operator*(const mat2x2 &,const mat2x2 &);


void affiche(const mat2x2 & m)
{
   cout << "|"<< m.M11 << " " << m.M12 << "|"<< endl;
   cout << "|"<< m.M21 << " " << m.M22 << "|"<< endl;
}


mat2x2 operator+(const mat2x2 & A,const mat2x2 & B)
{
   mat2x2 resultat;
   resultat.M11=A.M11+B.M11;
   resultat.M12=A.M12+B.M12;
   resultat.M21=A.M21+B.M21;
   resultat.M22=A.M22+B.M22;
   return resultat;
}
```

```
mat2x2 operator*(const mat2x2 & A,const mat2x2 & B)
{
   mat2x2 resultat;
   resultat.M11=A.M11*B.M11 + A.M12*B.M21;
   resultat.M12=A.M11*B.M12 + A.M12*B.M22;
   resultat.M21=A.M21*B.M11 + A.M22*B.M21;
   resultat.M22=A.M21*B.M12 + A.M22*B.M22;
   return resultat;
}


void main()
{

   mat2x2 m1={1,2,3,4},m2={1,4,8,-1};

   cout << "m1 = " << endl;
   affiche(m1);
   cout << "m2 = " << endl;
   affiche(m2);

   cout << "m1+m2=" << endl;
   affiche(m1+m2);   // affiche ce que retourne l'opérateur +

   cout << "m1*m2=" << endl;
   affiche(m1*m2);   // affiche ce que retourne l'opérateur *
}
```

Résultat

```
m1=
| 1 2 |
| 3 4 |
m2=
| 1 4 |
| 8 –1|
m1+m2=
| 2 6 |
| 11 3 |
m1*m2=
| 17 2 |
| 35 8 |
```

Exemple 18. Surcharge d'opérateurs

Il faut garder à l'esprit que lorsqu'on écrit `m1+m2` dans le programme, c'est interpreté comme `operator+(m1,m2)` par le compilateur. Cela appelle donc la fonction que l'on a définie.

3.6.2 Choix du paramétrage des opérateurs.

Lorsque l'on écrit `m1+m2` ou `m1*m2`, l'opérateur doit réaliser un calcul à partir des deux opérandes et retourner une structure correspondant au résultat du calcul. Mais, la somme ou le produit de `m1` et `m2` ne doit en aucun cas modifier les opérandes `m1` et `m2`. On en déduit :

Les arguments des opérateurs + et * sont des paramètres d'entrée de l'opérateur.

Il est donc important que ces arguments soient passés par référence constante (voir chapitre précédent).

Le prototype de ces opérateurs doit donc être (le type du retour reste à établir):

```
? operator+(const mat2x2 &, const mat2x2 &)
? operator*(const mat2x2 &, const mat2x2 &)
```

D'un autre côté, l'opérateur retourne le résultat du calcul. D'ailleurs, dans les deux opérateurs, ce calcul est réalisé grâce à une variable `resultat`, locale à l'opérateur. Puisque le calcul est stocké dans une variable locale à l'opérateur, il n'est pas possible de retourner une référence à cette variable. Le retour de l'opérateur doit donc être un retour par valeur.

Finalement, les prototypes de ces opérateurs doivent être :

```
mat2x2 operator+(const mat2x2 &, const mat2x2 &)
mat2x2 operator*(const mat2x2 &, const mat2x2 &)
```

Remarque : il est important de mener cette reflexion au moment du choix du paramétrage des fonctions et en particulier des opérateurs.

3.6.3 Utilisation de l'opérateur = par défaut.

Dans l'exemple de surcharge d'opérateurs précédent, nous n'avons jamais stocké le résultat rendu par les opérateurs + et * dans une variable. Le programme principal suivant fonctionne pourtant correctement :

```
void main()
{

   mat2x2 m1={1,2,3,4},m2={1,4,8,-1},m3;

   m3=m1+m2 ;
   affiche(m3);

   cout << endl ;

   m3=m1*m2;
   affiche(m3);
}
```

Résultat

```
| 2 6 |
| 11 3 |

| 17 2 |
| 35 8 |
```

Pourquoi cela fonctionne-t-il sans surcharger l'opérateur = ?

L'opérateur = réalise, si l'on ne le surcharge pas explicitement, une copie membre-à-membre. Or c'est exactement ce qui est nécessaire lorsque l'on souhaite faire l'affectation de deux matrices de taille 2x2.

Précisément, lorsque l'on écrit

```
m3 = m1+m2 ;
```

il y a, dans l'ordre

1) appel de la fonction `operator+(m1,m2)`
2) la structure retournée par cet opérateur est utilisée pour réaliser la copie membre-à-membre dans `m3`.

Ici, cela convient parfaitement. Mais pour certaines classes, le traitement par défaut ne convient pas du tout. Il est alors indispensable de redéfinir explicitement ce que doit réaliser l'opérateur d'affectation.

Remarque : lorsque l'on surcharge l'opérateur = (ce qui n'est possible que pour les classes), on annule le traitement par défaut.

3.7 Conclusion

On a commencé ici la présentation de la surcharge d'opérateurs en C++. Cette surcharge est en fait très simple à mettre en œuvre. Des précisions seront néanmoins données après avoir présenté la définition de classes d'objets. Notons que le point le plus délicat reste le choix du type des arguments de ces opérateurs (passage/retour par valeur, référence ou référence constante).

4 LES CLASSES D'OBJETS

Nous abordons dans ce chapitre la notion d'objet qui est sans nul doute l'apport essentiel de C++. Cependant, C++ n'est qu'un langage orienté objet. Destiné à être une sur-couche de C, il n'a pas été conçu dès l'origine comme un langage à objets pur. Il n'a donc pas toutes les caractéristiques d'un tel langage. Avant d'aborder la notion de classe, nous revenons tout d'abord sur les structures en C++.

4.1 Les structures

La structure en C, ne contient que des données. En C++, elle peut désormais contenir également des fonctions appelées *méthodes*. En voici un exemple :

Définition de la structure dans un fichier `personne.h` **:**

```
// personne.h
#ifndef __PERSONNE__
#define __PERSONNE__

#include <iostream.h>
#include <string.h>

struct personne
{
      char nom[20];          // nom de la personne
      unsigned age;          // son âge
      void identifie(void);  // fonction permettant de l'identifier
};

void personne :: identifie(void)   // définition de la méthode
{
   cout << "Je m'appelle "<< nom << " et j'ai " << age << " ans\n";
}
#endif
```

Consulter l'annexe pour l'explication de ces directives de compilation conditionnelle

Utilisation de cette structure :

```
#include "personne.h"
void main(void)
{
   personne agent;

   strcpy(agent.nom,"Jacques");  // on donne un nom à l'agent
   agent.age=20;                 // un âge
   agent.identifie();            // on cherche à l'identifier
}
```

Sortie écran

```
Je m'appelle Jacques et j'ai 20 ans
```

Exemple 19. Type structuré contenant des fonctions membre

Le programme précédent utilise une structure définie par :

```
struct personne{
    char nom[20];          // nom de la personne
    unsigned age;          // son âge
    void identifie(void);  // fonction permettant de l'identifier
    };
```

Ce qui est nouveau ici, c'est la possibilité de mettre une fonction dans une structure. En fait, on peut mettre soit la définition de la fonction, soit son prototype. C'est ce dernier cas qui est présenté ici. La définition de `identifie()` se trouve à l'extérieur de la structure :

```
// définition de la méthode identifie de la structure personne
void personne::identifie(void)
{
   cout << "Je m'appelle "<< nom << " et j'ai " << age << " ans\n";
}
```

Plusieurs structures différentes pourraient avoir une méthode appelée `identifie()`. Pour résoudre cet éventuel conflit d'identificateurs, l'entête de la méthode est précédé de l'identificateur de la structure qui la contient, selon la syntaxe:

```
   identificateur propriétaire::identificateur méthode(…)
```

On a donc ici l'entête :

```
void personne::identifie(void)
```

Notez qu'on n'a pas écrit

```
~~void struct personne :: identifie(void)~~
```

qui est une déclaration erronée en C++. Le propriétaire d'une méthode ne pouvant être qu'une structure (`struct`) ou une classe (`class`), cette règle entraîne qu'une structure ne pourra porter le même nom qu'une classe.

Revenons au corps de notre méthode `personne::identifie()` :

```
// définition de la méthode identifie de la structure personne
void personne::identifie(void)
{
   cout << "Je m'appelle "<< nom << " et j'ai " << age << " ans\n";
}
```

On remarque que la méthode `identifie()` fait référence à des variables appelées `nom` et `age` non déclarées dans `identifie()` et qui ne sont pas non plus des variables globales (ce sont les champs d'une structure donc d'un type de donnée). On ne sait donc pas ce que référence `identifie()` avec ces idenficateurs `nom` et `age`.

On en apprend un peu plus en regardant l'appel qui est fait dans le programme d'utilisation :

```
void main(void)
{
   personne agent;         // une variable de type struct personne

   strcpy(agent.nom,"Jacques");  // on donne un nom à l'agent
   agent.age=20;                 // un âge
   agent.identifie();            // on cherche à l'identifier
}
```

Une variable `agent` de type `personne` est définie puis initialisée. La méthode `identifie` est appelée par :

```
   agent.identifie();
```

L'appel donne donc une information qu'on n'avait pas dans la définition de la méthode : on appelle la méthode `identifie()` de la variable `agent`. Les champs `nom` et `age` référencés par `identifie()` sont alors ceux de cette variable `agent`.

En réalité, l'appel `agent.identifie()` appelle la fonction `identifie()` en lui passant l'adresse de la variable `agent`, ce qui permet à `identifie()` de travailler sur les champs `nom` et `age` de cette variable.

Dans l'appel `agent.identifie()`, on dira que `agent` est *l'objet courant* manipulé par la méthode `identifie()`**.**

Remarque : dans l'appel `agent.identifie()` tout se passe comme si l'on avait un appel `identifie(&agent)`. Autrement dit, même quand une méthode n'a pas d'argument explicite, elle reçoît toujours l'objet courant comme argument implicite.

Qu'avons-nous appris ?

. Une structure peut définir, outre des données, des fonctions membres (ou méthodes).
. L'appel à une méthode se fait selon la syntaxe

```
propriétaire.méthode( .... )
```

. La définition de la méthode peut référencer des champs propriétaire sans nommer celui-ci. La méthode reçoit automatiquement son adresse lors de l'appel.

4.2 Les classes

Un objet associe données et méthodes qui les manipulent dans un même moule (voir chapitre 1). En C++, les structures sont en quelque sorte des objets. Cependant, les structures n'empêchent pas l'accès direct à leurs champs de données, ce qui n'est normalement pas autorisé en langage objet pur. Autrement dit, les structures n'encapsulent pas les données**.** Ainsi on a pu écrire dans l'exemple précédent :

```
personne agent;              // une variable de type personne
strcpy(agent.nom,"Jacques"); // on donne un nom à l'agent
agent.age=20;                // un âge
```

On a pu modifier directement les champs `nom` et `age` de l'objet `agent`, sans passer par les méthodes. Or, un objet n'est vraiment sûr que dans la mesure où l'on interdit l'accès direct à ses champs de données. Par exemple, si le programmeur manipule directement le champ `nom` et y met des caractères en oubliant le caractère de fin de chaîne, la méthode `identifie()` va avoir un comportement aberrant.

Pour éviter ce problème, il nous faut interdire l'accès direct aux champs de l'objet. La structure ne nous permet pas de le faire, puisque cela remettrait en cause le type `struct` du C. Le C++ a donc introduit la *classe*.

Reprenons le même exemple que précédemment dans lequel nous remplaçons la structure par une classe :

```
// personne.h
#ifndef __CLS_PERSONNE__
#define __CLS_PERSONNE__
#include <iostream.h>
#include <string.h>

class personne
{
private:
      char nom[20];        // nom de la personne
      unsigned age;        // son âge
public:
      void identifie(void);
      void initialise(char *,unsigned);
};

void personne::identifie(void)
{
   cout << "Je m'appelle "<< nom ;
   cout << " et j'ai " << age << " ans\n";
}
void personne::initialise( char *nom_p, unsigned age_p)
{        nom[19]='\0';
         strncpy(nom,nom_p,19);
         age=age_p;
}
#endif
```

Partie Privée de la classe

Il s'agit ici des données membres :

`nom` : tableau de caractères
`age` : entier non signé

Partie Publique de la classe

Il s'agit ici des méthodes de la classe.

`initialise()`
`identifie()`

Programme d'utilisation de cette classe :

```
#include "personne.h"
void main(void)
{
   personne agent,standardiste;  // agent, standardiste  = objets de la classe personne

   agent.initialise("Jacques",20);
   standardiste.initialise("Pierre",25);
   agent.identifie();
   standardiste.identifie();
}
```

Exemple 20. Premier exemple de classe

Commençons par la définition de la classe `personne` (dans le fichier `personne.h`):

```
class personne
{
private :
      char nom[20];
      unsigned age;
public :
      void identifie(void);
      void initialise(char *,unsigned);
};
```

Par cette syntaxe, l'identificateur `personne` désigne une classe, c'est-à-dire en quelque sorte un type. Autrement dit, toute variable du type `personne` sera un *objet*. Par exemple, lorsque l'on écrit

```
personne agent, technicien, standardiste;
```

`agent`, `technicien` et `standardiste` sont trois objets de la classe `personne`. Parce que la notion de classe généralise celle de type, on dira indifféremment :

. `standardiste` est un objet de la classe `personne`
. `standardiste` est une instance de la classe `personne`
. `standardiste` est du type `personne`
. `standardiste` est une variable de la classe `personne`

Les *membres* (ou champs) d'un objet peuvent être des données ou des méthodes (fonctions). Ces champs peuvent avoir l'un des trois attributs suivants :

privé Un champ privé (`private`) d'un objet n'est accessible que par les seules méthodes de l'objet, par les méthodes d'un autre objet de sa classe et par certaines fonctions dites amies (`friend`).

public Un champ `public` est accessible par toute fonction définie ou non au sein de l'objet.

protégé Un champ protégé (`protected`) n'est accessible que par les seules méthodes internes de l'objet ou d'un objet dérivé (voir ultérieurement le concept d'héritage) et par les fonctions amies (`friend`).

La syntaxe de définition d'une classe est la suivante :

```
class nom_classe{
   private :
      ..... données/méthodes privées
   protected :
      ..... données/méthodes protégées
   public :
      ..... données/méthodes publiques
};    // <- ne pas oublier le « ; » ici
```

Remarques :

. L'ordre de déclaration des attributs `private`, `protected` et `public` est quelconque.
. Un même attribut peut apparaître plusieurs fois dans la déclaration de l'objet.
. Des données ou méthodes précédées d'aucun attribut sont déclarées *privées*.

Quel est le rôle d'`initialise()` ?

Maintenant que `nom` et `age` sont devenues des données privées de la classe `personne`, les instructions

```
strcpy(agent.nom,"Jacques");
agent.age=20;
```

pour initialiser l'objet `agent` deviennent illégales : les références `agent.nom` et `agent.age` sont interdites car elles désignent des données privées de l'instance `agent`. Il nous faut donc une méthode pour accéder aux données privées, méthode publique de plus car elle doit pouvoir être référencée à l'extérieur de l'objet personne : c'est le rôle de la méthode `initialise()`.

```
void personne::initialise( char *nom_p, unsigned age_p)
{     nom[19]='\0';
      strncpy(nom,nom_p,19);     // copie nom_p dans nom
      age=age_p;                 //  copie age_p dans age
}
```

La méthode `initialise()` initialise les champs `nom` et `age` d'un objet avec les valeurs de `nom_p` et `age_p` qu'on lui passe en paramètres. L'appel devient alors celui-ci :

```
void main(void)
{
   personne agent;                    // une variable de type personne
   agent.initialise("Jacques",20);    // on initialise agent
   agent.identifie();                 // on cherche à l'identifier
}
```

Qu'avons-nous gagné ?

En rendant privées les données de la classe `personne`, nous avons rendu ce dernier plus sûr. Dans notre exemple, le champ `nom` d'un objet `personne` est un tableau de 20 caractères. Les chaînes de caractères sont une source fréquente d'erreurs en C : il y a risque de débordement du tableau qui leur est alloué. Voyons ce qui se passe dans les deux cas (données publiques/données privées) lorsqu'un programmeur veut affecter la chaîne `"Ceci est un très long nom"` au champ `nom` :

Si le champ `nom` est une donnée publique : le programmeur a accès au champ `nom` de l'objet `agent` et rien ne l'empêche de faire son initialisation par `strcpy(agent.nom,"Ceci est un très long nom");`
Il y aura alors débordement du tableau `nom` et « plantage » à court terme du programme.

Si le champ `nom` est une donnée privée : le programmeur est obligé de passer par la méthode `initialise()`, pour initialiser le champ `nom` de l'objet `agent`. Il écrira donc :

```
agent.initialise("Ceci est un très long nom", 20);
```

Or, si on regarde la méthode `initialise()`

```
void personne::initialise( char *nom_p, unsigned age_p)
{
      strncpy(nom,nom_p,19);       // copie nom_p dans nom
      age=age_p;                   // copie age_p dans age
}
```

elle ne copie que 19 caractères au plus dans le champ `nom` de l'objet `agent` : il n'y aura donc pas débordement du tableau, mais simplement troncature du nom : le programme peut continuer normalement.

Conclusion

Un objet bien construit**,** assurant l'intégrité de ses données, facilite le travail du programmeur qui l'utilise car une source importante d'erreurs est ainsi éliminée. Le contrôle étant géré au niveau même de l'objet. Le terme « bien construit » est important : si le concepteur de l'objet `agent` avait utilisé dans la méthode `initialise()` l'instruction

```
strcpy(nom,nom_p);
```

au lieu de

```
strncpy(nom,nom_p,19);
```

on aurait eu un objet mal construit car n'évitant pas le débordement du tableau `nom`.

En général, les données d'un objet sont déclarées privées alors que ses méthodes sont déclarées publiques. Cela signifie que l'utilisateur[3] de l'objet n'aura pas accès directement aux données de l'objet, mais pourra faire appel aux méthodes de l'objet et notamment à celles qui donneront accès aux données privées.

Notons qu'en C++ l'encapsulation de données n'est malgré tout pas imposée par le langage (on n'est pas obligé de déclarer les données comme des membres privés), mais est vivement recommandée.

4.3 Les méthodes d'une classe

Les méthodes d'une classe sont des fonctions comme les autres et supportent donc les points suivants :

. le passage d'arguments par référence
. l'utilisation de paramètres par défaut
. la surcharge

Voici un exemple illustrant ces trois points :

```
// personne.h
#ifndef __CLS_PERSONNE_
#define __CLS_PERSONNE_
#include <iostream.h>
#include <string.h>

class personne
{
  private :
   char nom[20];
   unsigned age;
  public :
   void identifie() const;
   void initialise(char * ="?",unsigned =0);
   void initialise(const personne &);
};

void personne::identifie() const
{
   cout << "Je m'appelle "<< nom << " et j'ai " << age << " ans\n";
}

void personne::initialise(char *nom_p, unsigned age_p)
{     nom[19]='\0';
      strncpy(nom,nom_p,19);
       age=age_p;
}
void personne::initialise(const personne &individu)
{   nom[19]='\0';
    strncpy(nom,individu.nom,19);
    age=individu.age;
}
#endif
```

La présence de ce « `const` » fera l'objet d'une explication ultérieurement.

[3] On distingue celui qui a écrit la classe, que l'on appelle concepteur de classe, de celui qui l'utilise. Il peut s'agir de deux individus différents. Dans la fonction `void main()` du programme, on se situe côté utilisateur de la classe. Grâce à l'encapsulation, l'utilisateur ne sait pas nécessairement comment est implémentée la classe.

```
#include "personne.h"
void main(void)
{

   personne agent1,agent2,agent3;

   agent1.initialise("Jacques",20);
   agent1.identifie();
   agent2.initialise(agent1);    // initialisation de agent2 avec agent1
   agent2.identifie();
   agent3.initialise();          // initialisation de agent3 avec les valeurs par défaut
   agent3.identifie();
}
```

Sortie écran

```
Je m'appelle Jacques et j'ai 20 ans
Je m'appelle Jacques et j'ai 20 ans
Je m'appelle ? et j'ai 0 ans
```

Exemple 21. Surcharge des méthodes d'une classe, valeurs par défaut, passage par référence

Dans l'exemple précédent la méthode `initialise()` est surchargée, a des paramètres par défaut (version 1) et accepte un paramètre par référence constante (version 2). Revenons sur cette version :

```
void personne::initialise(const personne &individu)
{   nom[19]='\0';
    strncpy(nom,individu.nom,19);
    age=individu.age;
}
```

L'appel a été le suivant :

```
   personne agent1,agent2;
   agent2.initialise(agent1);    // on initialise agent2 avec agent1
```

Dans l'appel précédent, `agent1` est un paramètre d'entrée de la méthode `initialise()` : ses champs vont être lus mais non modifiés par la méthode. On pouvait donc passer cet argument par valeur. Comme nous l'avons vu dans le chapitre précédent, le passage par valeur implique une recopie de toutes les données de l'objet. On préfère donc dans ce cas le passage par référence constante.

On remarque que la méthode `initialise()` fait référence directement aux champs des deux objets de la classe `personne` impliqués :

`nom` et `individu.nom`
`age` et `individu.age`

On en déduit qu'une méthode d'un objet a accès aux champs, même privés, de tous les objets de sa classe, et non aux seuls champs de l'objet courant.

4.4 Constructeur et destructeur d'une classe

Il existe des méthodes particulières pour un objet : ses *constructeurs* et son *destructeur*.

Définition (constructeur) : un *constructeur* est une méthode qui porte le même nom que la classe. Cette méthode est appelée juste après la création de l'objet. On s'en sert généralement pour l'initialiser. C'est une méthode qui peut accepter des arguments mais qui ne rend aucun résultat. Son prototype ou sa définition ne sont précédés d'aucun type (pas même void).

Si une classe `maClasse` a un constructeur acceptant `n` arguments `argi`, la déclaration d'un objet de cette classe se fera sous la forme :

```
maClasse unObjet(arg1,arg2, ... argn);
```

C'est-à-dire que l'utilisateur d'une classe avec constructeur doit obligatoirement initialiser les objets lors de leur instanciation (sinon le compilateur signalera une erreur).

. Un constructeur peut être surchargé et avoir des arguments par défaut.

Définition (destructeur) : le *destructeur* est une méthode de la classe qui porte comme nom celui de la classe précédé d'un tilde « ~ ». Cette méthode est appelée juste avant la destruction de l'objet.

Un destructeur n'accepte aucun argument et ne rend aucun résultat. Son prototype ou sa définition ne sont précédés d'aucun type (pas même void).

. Un destructeur ne peut pas être surchargé.

4.5 Exemple 1

Reprenons notre exemple sur la classe `personne`, créons un constructeur chargé d'initialiser l'objet lors de sa création. Nous introduisons également un destructeur. Le constructeur et le destructeur font des écritures afin de révéler les moments où ils sont appelés.

```
// personne_v1.h
#ifndef __CLS_PERSONNE_V1_
#define __CLS_PERSONNE_V1_
#include <iostream.h>
#include <string.h>

class personne
{
 private :
      char nom[20];
      unsigned age;
 public :
      void identifie(void) const;
      personne(char*,unsigned );   // constructeur pour initialiser l'objet personne
      ~personne(void);             // destructeur
};

void personne::identifie(void)  const{
   cout << "Je m'appelle "<< nom << " et j'ai " << age << " ans\n";
}

personne::personne(char *nom_p, unsigned age_p){
   strncpy(nom,nom_p,19);
   nom[19]='\0';
   age=age_p;
   cout << "Le constructeur de la personne (" << nom << "," << age << ") a été appelé\n";
 }

personne::~personne (void){
  cout << "Le destructeur de la personne (" << nom << "," << age << ") a été appelé\n";
}
#endif
```

```
#include "personne_v1.h"
void main(void)
{
   personne agent1("Jacques",20),
   agent2("Louise",18);

}
```

Sortie écran

```
Le constructeur de la personne (Jacques,20) a été appelé
Le constructeur de la personne (Louise,18) a été appelé
Le destructeur de la personne (Louise,18) a été appelé
Le destructeur de la personne (Jacques,20) a été appelé
```

Exemple 22. Classe `personne` version 1

Dans le programme précédent, rien n'est fait dans la fonction `main()` et pourtant il y a des affichages écran !

Explication : `agent1` et `agent2` sont des variables locales ou automatiques de la fonction `main()` et sont donc créées lorsque l'exécution de `main()` commence. Leur constructeur est donc automatiquement appelé. Il a deux arguments :

```
personne::personne(char *,unsigned );
```

Les variables `agent1` et `agent2` doivent donc être déclarées avec deux arguments qui seront transmis au constructeur, ce qui explique la syntaxe suivante:

```
class personne agent1("Jacques",20), agent2("Louise",18);
```

Le constructeur initialise chaque variable `agent1` et `agent2` avec les arguments passés et fait une écriture : ceci explique les deux premiers affichages écran :

Le constructeur de la personne (Jacques,20) a été appelé
Le constructeur de la personne (Louise,18) a été appelé

Ensuite, le corps de la fonction `main()` se termine (il n'y a pas d'autre instruction). Les variables locales à `main()` sont donc détruites. Comme la classe `personne` a un destructeur, celui-ci est automatiquement appelé et réalise également une écriture. Ceci conduit aux affichages :

Le destructeur de la personne (Louise,18) a été appelé
Le destructeur de la personne (Jacques,20) a été appelé

4.6 Exemple 2

Dans l'exemple précédent les objets automatiques `agent1` et `agent2` étaient créés dans la pile et leur durée de vie était celle de la fonction `main()`. Ici, nous créons les objets `agent1` et `agent2` dans le tas. C'est-à-dire que l'on va créer dynamiquement des objets. Leur durée de vie est alors contrôlée par les opérateurs `new` et `delete` et n'est plus dépendante de la durée de vie de `main()`.

Nous rappelons que pour créer un objet de type `T_objet` dans le tas la syntaxe est alors :

```
T_objet * ptr = new T_objet;
```

Pour restituer la zone allouée pour cet objet , on écrit

```
delete ptr;
```

Si l'objet créé a un constructeur avec des arguments, les arguments peuvent être passés au constructeur au moment de l'allocation de mémoire. La syntaxe devient :

```
T_objet * ptr = new T_objet(arg1,arg2,…,argN);
```

On passe ici les arguments au constructeur de l'objet.

La libération de la mémoire quant à elle ne change pas.

```
#include "personne_v1.h"
void main()
{
   personne *ptr1, *ptr2;

   cout << "instant t1"<<endl;

   ptr1 = new personne("jean",25);

   cout << "instant t2"<<endl;

   ptr2 = new personne("jacques",18);

   cout << "instant t3"<<endl;

   delete ptr1;

   cout << "instant t4"<<endl;

}
```

Résultats écran

```
instant t1
Le constructeur de la personne (jean,25) a été appelé
instant t2
Le constructeur de la personne (jacques,18) a été appelé
instant t3
Le destructeur de la personne (jean,25) a été appelé
instant t4
```

Exemple 23. Allocation dynamique d'objets de type `personne`.

On remarque les points suivants :

La syntaxe suivante

```
personne * ptr1 = new personne("jean",25);
```

crée un objet personne dans le tas. Cet objet est pointé par **`ptr1`** et est initialisé avec les arguments ("jean",25). Le constructeur est donc appelé et est à l'origine de l'affichage de :

Le constructeur de la personne (jean,25) a été appelé

Il est à noter que, contrairement aux objets automatiques, les objets dynamiques ne sont construits qu'au moment de l'allocation de la mémoire. En outre, la déclaration d'une variable pointeur sur un objet de la classe `personne` ne crée pas d'objet de cette classe.

La syntaxe

```
delete ptr1;
```

détruit l'objet pointé par `ptr1`. Puisqu'il y a destruction d'objet, il y a appel du destructeur qui affiche alors :

Le destructeur de la personne (Jean,25) a été appelé

4.7 Exemple 3

Dans les exemples précédents, le destructeur n'avait aucun intérêt. Les exemples 1 et 2 ne servent qu'à la compréhension du mécanisme d'appel des constructeurs/destructeur.

L'exemple suivant utilise un objet pour lequel un destructeur en présente un : celui de restituer de la mémoire obtenue par le constructeur. La différence essentielle réside dans le fait que la chaîne de caractères n'est plus stockée dans un tableau de taille fixée par la déclaration de la classe, mais allouée dynamiquement selon les besoins de chaque objet de la classe.

```
//personne_v2.h - Deuxième version de la classe personne
#ifndef __CLS_PERSONNE_V2_
#define __CLS_PERSONNE_V2_
#include <iostream.h>
#include <string.h>
#include <process.h>

class personne
{
private:
      char * ptrNom;    // pointeur sur une chaîne de caractères
      unsigned age;
public:
      void identifie() const;
      personne(char *,unsigned );
      ~personne(void);
};

void personne::identifie(void)   const
{
   cout << "Je m'appelle "<< ptrNom << " et j'ai " << age << " ans\n";
}

personne::personne(char *nom_p, unsigned age_p)
{
   ptrNom=new char[strlen(nom_p)+1];        // demande de la mémoire pour le nom
   if(ptrNom==NULL)
   {
      cout << "Mémoire insuffisante ... Abandon ...\n";
      exit(1);
   }
   strcpy(ptrNom,nom_p);                    // initialise la chaîne pointée par ptrNom
   age=age_p;                               // initialise age
   cout << "Le constructeur de la personne (" << ptrNom << "," << age << ") a été
appelé\n";
}

personne::~personne()
{
  cout << "Le destructeur de la personne (" << ptrNom << "," << age << ") a été appelé\n";
  delete ptrNom; // restitue la mémoire pointée par ptrNom
}

#endif
```

```
// programme d'utilisation de la classe personne (deuxième version)
#include "personne_v2.h"

void main(void)
{
   personne *ptrAgent= new personne("Jacques",20);
   personne agent("Louise",18);
   delete ptrAgent;
```

Résultats écran

```
Le constructeur de la personne (Jacques,20) a été appelé
Le constructeur de la personne (Louise,18) a été appelé
Le destructeur de la personne (Jacques,20) a été appelé
Le destructeur de la personne (Louise,18) a été appelé
```

```
}
```

Exemple 24. Deuxième version de la classe `personne`

Commentaires

En comparant à la première version de la classe, la définition de la classe `personne` a ici véritablement changé. La structure de chacun des objets instanciés à partir de cette classe est résolument différente.

En effet, là où nous avions auparavant un tableau de caractères appelé **`nom`** dans la première version de la classe, nous n'avons plus qu'un pointeur sur une chaîne de caractères.

Où sera stockée la chaîne ? Le tableau de caractères nécessaire au stockage des caractères du nom de la personne est désormais alloué dynamiquement par le constructeur de l'objet.

Le constructeur de cette classe est donc:

```
personne::personne ( char *nom_p, unsigned age_p)
{
   ptrNom=new char[strlen(nom_p)+1];        // demande de la mémoire pour le nom
   if(ptrNom==NULL)
   {
      cout << "Mémoire insuffisante ... Abandon ...\n";
      exit(1);
   }
   strcpy(ptrNom,nom_p);  // initialise la chaîne pointée par ptrNom
   age=age_p;          // initialise age
   cout << "Le constructeur de la personne (" << ptrNom << "," << age << ") a été
appelé\n";
}
```

Avant d'affecter un nom à l'objet de la classe `personne`, le constructeur demande de l'espace pour loger ce nom et fait pointer le champ `ptrNom` dessus. Un test d'insuffisance mémoire est également fait. Le destructeur évolue alors comme suit :

```
personne::~personne ()
{
      cout << "Le destructeur de la personne (" << ptrNom << "," << age << ") a été
appelé\n";
      delete ptrNom;            // restitue la mémoire pointée par ptrNom
}
```

Rappelons-nous que le destructeur est appelé avant la destruction définitive de l'objet. Ici, il prend soin de restituer l'espace mémoire pointé par le champ `ptrNom` devenu inutile puisque l'objet va être détruit. Le constructeur et le destructeur participent à la sûreté d'utilisation d'un objet. Leur présence assure que l'objet est toujours initialisé et détruit proprement.

4.8 Objets avec des données en profondeur

On discute ici les différences fondamentales entre les deux versions de la classe `personne` présentées précédemment. Le lecteur est donc invité à se reporter aux deux définitions de la classe `personne` faites respectivement p.49 et p.52.

Les exemples précédents mettent en évidence deux techniques très différentes pour gérer des données liées à un objet. On va ici illustrer et comparer ces méthodes.

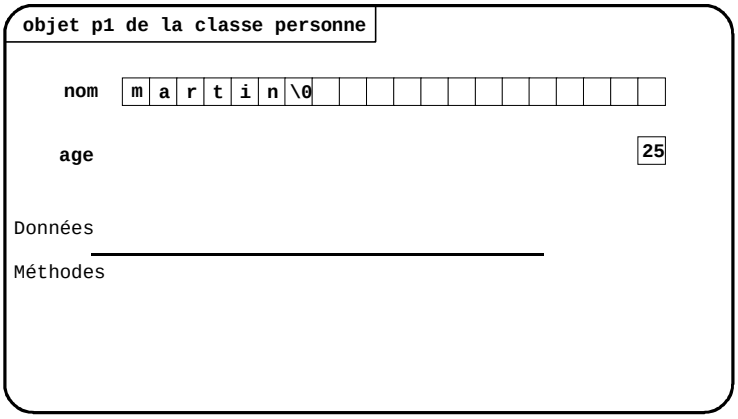
Première version de la classe personne

```
class personne
{
private :
      char nom[20];        // tableau de 20 caractères
      unsigned age;
public:

   // …
};
```

En déclarant un objet automatique de cette première version de la classe,

```
personne p1("martin",25);
```

on obtient la structure suivante :



On peut en effet voir un objet comme une structure avec plusieurs champs de données. Avec la première version de la classe, chaque objet contient 20 caractères + 1 entier non signé. Toutes ces données sont stockées au sein même de l'objet.

Inconvénients de cette version de la classe :

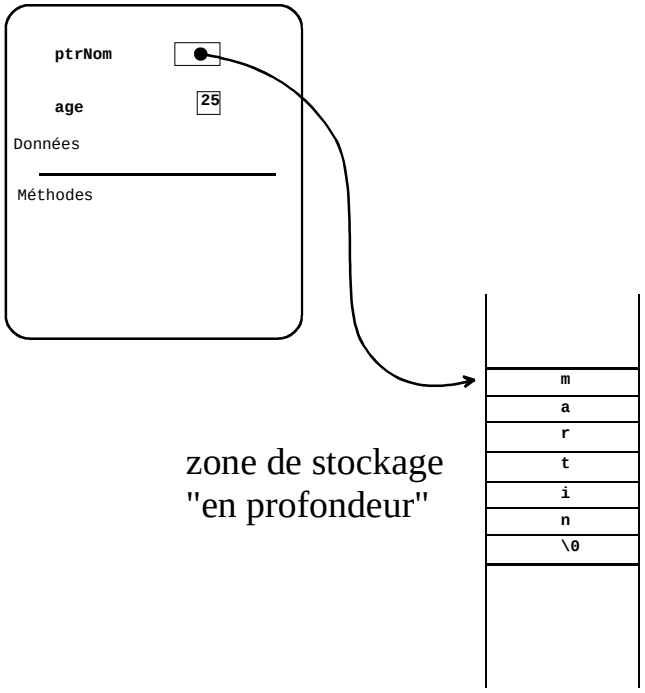
. les noms stockés sont forcément limités à la taille du tableau. Ici, les noms doivent faire au plus 19 caractères.
. en outre, si le nom stocké est très court, une grande partie du tableau n'est pas utilisée. Dans le cas de l'objet `p1`, 13 caractères sont réservés inutilement.

Avec cette technique de stockage des données, soit on manque de place dans l'objet, soit on gaspille des mots mémoire inutilement.

Seconde version de la classe personne avec *données en profondeur*

```
class personne
{
private:
      char * ptrNom;    // pointeur sur une chaîne de caractères
      unsigned age;
public:

   // …
};
```

Les objets instanciés à partir de la seconde version de la classe ont la structure suivante :



Un objet de cette version de la classe `personne` ne contient plus directement les données liées au nom d'une personne. Chaque objet de cette classe ne contient que l'adresse où est stockée la chaîne de caractères en mémoire. La zone de stockage de ces caractères est allouée par le constructeur de l'objet et désallouée par le destructeur de l'objet. On dit que la chaîne de caractère est stockée *en profondeur*. En effet, à la différence de la première version, on voit bien que les caractères ne sont plus vraiment dans l'objet, mais dans une zone mémoire distante. On retrouve néanmoins sans problème les données en profondeur par l'intermédiaire du pointeur qui, lui, est tout de même dans l'objet (`ptrNom`).

Intérêt de cette gestion des données

> En stockant les données en profondeur (c'est-à-dire dynamiquement), la zone mémoire réservée pour l'objet correspond effectivement précisément au besoin de l'objet. Il n'y a donc plus de limitation de la taille des noms stockés et, à l'inverse, un nom court ne gaspille pas de la mémoire.

Au vu de cette comparaison entre les deux versions de la classe, il apparaît que la première version de la classe `personne` n'est pas une bonne solution, bien que plus simple à mettre en oœuvre. Aussi, dès que les objets d'une classe n'auront pas tous des données de même taille, il conviendra de stocker ces données en profondeur, et donc d'utiliser la technique présentée dans l'exemple 3 de ce chapitre.

4.9 Les méthodes constantes

Nous allons ici analyser un nouveau motif d'emploi du modificateur `const`. Réexaminons la classe `personne` à laquelle nous avons ajouté une méthode de modification de l'âge `setAge()`.

```
//personne_v3.h – version 3 de la classe personne
#ifndef __CLS_PERSONNE_V3_
#define __CLS_PERSONNE_V3_
#include <iostream.h>
#include <string.h>
#include <process.h>

class personne
{
private:
      char * ptrNom;
      unsigned age;
public:
      personne(char *,unsigned );
      ~personne(void);

      void identifie() const;
      void setAge(unsigned);
      unsigned getAge() const;
};

void personne::identifie(void) const
{
   cout << "Je m'appelle "<< ptrNom << " et j'ai " << age << " ans\n";
}

void personne::setAge(unsigned a)
{
   age=a;
}

unsigned personne::getAge() const
{
   return age ;
}

personne::personne ( char *nom_p, unsigned age_p)
{
   ptrNom=new char[strlen(nom_p)+1];
   if(ptrNom==NULL)
   {
      cout << "Mémoire insuffisante ... Abandon ...\n";
      exit(1);
   }
   strcpy(ptrNom,nom_p);
   age=age_p;
   cout << "Le constructeur de la personne (" << ptrNom << "," << age << ") a été
appelé\n";
}
```

Ce `const` indique qu'il s'agit d'une méthode *constante*. Une méthode constante peut être appelée sur un objet variable ou constant.

Cette méthode, qui n'est pas constante, ne peut pas être appelée sur un objet constant.

Le `const` doit figurer ici également. Il fait partie de la signature d'une méthode.

```
personne::~personne()
{
  cout << "Le destructeur de la personne (" << ptrNom << "," << age << ") a été appelé\n";
  delete ptrNom;
}
#endif
```

Exemple 25. Les méthodes constantes d'une classe.

La présence d'un modificateur `const` à la suite de la liste des arguments indique qu'il s'agit d'une *méthode constante*, c'est-à-dire une méthode qui peut être invoquée indifféremment sur un objet variable ou constant. En revanche, une méthode non constante ne peut être appelée que pour un objet variable.

La tentative d'appel d'une méthode non constante sur un objet constant produit une erreur de compilation !

Remarque : notons que les constructeurs et destructeurs peuvent toujours être appelés sur les objets constants ou variables sans qu'il soit nécessaire de le préciser.

Dans la classe précédente, les méthodes `identifie()` et `getAge()` sont des méthodes constantes, tandis que la méthode `setAge()` ne l'est pas. Quelle en est la conséquence ?

Dans l'exemple suivant d'utilisation de cette classe, l'objet `agentV` est un objet variable et `agentC` un objet constant .

```
#include personne_v3.h
void main()
{
   personne agentV("Jean",23);           // objet variable
   const personne agentC("Gerard",18);   // objet constant

   agentC.identifie() ;
   agentV.identifie() ;

   cout << " âge de agentV : " << agentV.getAge() << endl ;
   cout << " âge de agentC : " << agentC.getAge() << endl ;

   agentV.setAge(13) ;

   agentC.setAge(15) ;  << erreur de compilation.
}
```

Tous les appels sont légaux sauf l'appel de la méthode `setAge()` sur un objet constant.

```
   agentC.setAge(15) ;  << erreur de compilation.
```

Pourquoi avoir déclaré la méthode `setAge()` comme méthode constante ?

La méthode `setAge()` modifie les données de l'objet. Aussi, si un objet est déclaré constant, il n'est pas souhaitable que ses données puissent être modifiées au travers d'une méthode. Par conséquent, on empêche la méthode `setAge()` d'être appelée sur un objet constant en ne la déclarant pas comme méthode constante.

On adoptera donc la discipline de programmation suivante :

. méthode d'accès aux données ou « accesseur » : il s'agit des méthodes qui retournent ou affichent des données de l'objet sans les modifier. Dans l'exemple précédent, les méthodes `identifie()` et `getAge()` sont des accesseurs. Les accesseurs doivent être des méthodes constantes.

. méthode de modification des données ou « modificateur » ou « mutateur » : il s'agit des méthodes qui modifient les données de l'objet. Un modificateur ne doit donc pas être appelé sur un objet constant. Dans l'exemple précédent, `setAge()` est un modificateur. Par conséquent, un modificateur ne doit surtout pas être déclaré comme méthode constante.

4.10 Le pointeur `this`

En C++, tous les objets possèdent un champ privé particulier nommé `this`, sans qu'il soit nécessaire de le déclarer. Ce champ contient l'adresse de l'objet courant.

```
#include  <iostream.h>

class C
{
public:
   C* GetAdress(){ return this;}
};

void main()
{
   C obj1,obj2;

   cout << "adresse de l'objet obj1 : " << &obj1 << endl;
   cout << "adresse de l'objet obj2 : " << &obj2 << endl;

   cout << "adresse de l'objet obj1 via le pointeur this : " << obj1.GetAdress() << endl;
   cout << "adresse de l'objet obj2 via le pointeur this : " << obj2.GetAdress() << endl;
}
```

Résultat

```
adresse de l'objet obj1 :0x0065FDF4
adresse de l'objet obj2 :0x0065FDF0
adresse de l'objet obj1 via le pointeur this :0x0065FDF4
adresse de l'objet obj2 via le pointeur this :0x0065FDF0
```

Exemple 26. Le pointeur `this`.

La classe `C` ne contient qu'une seule méthode retournant, pour chaque objet, la valeur du pointeur `this`, c'est-à-dire l'adresse de l'objet qui appelle cette méthode. Comme on pouvait s'y attendre, on obtient la même adresse en écrivant soit `&obji`, soit `obji.GetAdress()`.

4.11 Affectation d'un objet à un autre objet

Soient deux objets de même type `T` :

```
T obj1, obj2;
```

L'affectation

```
obj1 = obj2;
```

est légale (mais pas nécessairement souhaitable comme on le verra ici).

Cette affectation utilise le *traitement par défaut* de l'opérateur = pour les types définis par le programmeur (voir chapitre sur les opérateurs). Comme pour les structures, l'affectation entre objets de même classe est une copie membre-à-membre (ou champ par champ) des champs de données de l'objet. Ici, tous les champs de donnée de l'objet `obj2` sont simplement recopiés dans ceux de l'objet `obj1`.

Que cela signifie-t-il exactement ? Que contient un objet ?

Soit une classe d'objets qui, pour schématiser, est déclarée ainsi

```
class C
{
   private:
            //données
   public:
            //méthodes
};
```

et deux objets déclarés par :

```
C obj1,obj2;
```

Chaque objet `obji` contient ses propres données. En revanche, les méthodes sont les mêmes pour tous les objets de la classe `C` et n'existent qu'en un seul exemplaire en mémoire. Dans certains cas, l'objet `obji` possède un pointeur sur ses méthodes. Mais ce n'est pas le cas le plus fréquent : le compilateur sait en général quelle méthode est appelée et où elle se trouve en mémoire. Il n'a donc nul besoin de ce pointeur qui aurait la même valeur pour tous les objets d'une même classe. Nous verrons cependant que dans certains cas d'héritage, il y a incertitude, pour le compilateur, sur la méthode réellement appelée. Il ne peut alors générer lui-même l'appel à la méthode. Dans ce cas, chaque objet `obji` aura un pointeur sur les méthodes de sa classe.

Retenons simplement ici, qu'un objet ne contient que des données et pas de méthodes.

Revenons à l'affectation entre objets d'une même classe : affecter un objet `obj2` à un autre objet `obj1` d'une même classe entraîne la recopie des données de l'objet `obj2` dans l'objet `obj1`. Voyons-en les conséquences pour les deux versions de la classe `personne`, selon qu'elle a ou pas des données en profondeur.

4.11.1 Affectation d'un objet à un autre objet d'une même classe lorsque les objets n'ont pas de données en profondeur

La première version de la classe personne, celle ne disposant pas de données en profondeur, est déclarée ainsi

```
class personne
{
 private :
      char nom[20];
      unsigned age;
public :
      // méthodes…
};
```

Ecrire

```
personne p1("louis",15),p2(" jean",23) ;
p1 = p2;
```

permet de copier les données de `p1` dans `p2`, c'est-à-dire tous les caractères du tableau `p2.nom[]` dans le tableau `p1.nom[]`, ainsi que `p2.age` dans `p1.age`.

Vérifions-le sur un exemple :

```
#include "personne_v1.h"
void main(void)
{
   personne agent1("Jacques",20);
   personne agent2("Louise",18);

   agent1.identifie();

   agent2.identifie();

   agent2 = agent1;              // affectation de agent1 à agent2

   agent1.identifie();
   agent2.identifie();
}
```

Résultat

```
Le constructeur de la personne (Jacques,20) a été appelé
Le constructeur de la personne (Louise,18) a été appelé
Je m'appelle Jacques et j'ai 20 ans
Je m'appelle Louise et j'ai 18 ans
Je m'appelle Jacques et j'ai 20 ans
Je m'appelle Jacques et j'ai 20 ans
Le destructeur de la personne (Jacques,20) a été appelé
Le destructeur de la personne (Jacques,20) a été appelé
```

Exemple 27. L'affectation par défaut est une copie membre-à-membre

Il n'y a donc aucun problème dans ce cas (et même dans tous les cas où les données sont toutes stockées au sein de l'objet).

4.11.2 Affectation d'un objet à un autre objet d'une même classe lorsque les objets ont des données en profondeur

Maintenant voyons ce qui se passe avec la seconde version de la classe `personne` :

```
class personne
{
private :
      char * ptrNom;
      unsigned age;
public :
      // méthodes
};
```

On sait que les données sont ici en profondeur. Nous avons vu ce que cela entraînait : un constructeur qui demande de la mémoire pour stocker le nom et un destructeur qui la restituait. Voyons maintenant ce qu'implique l'affectation suivante :

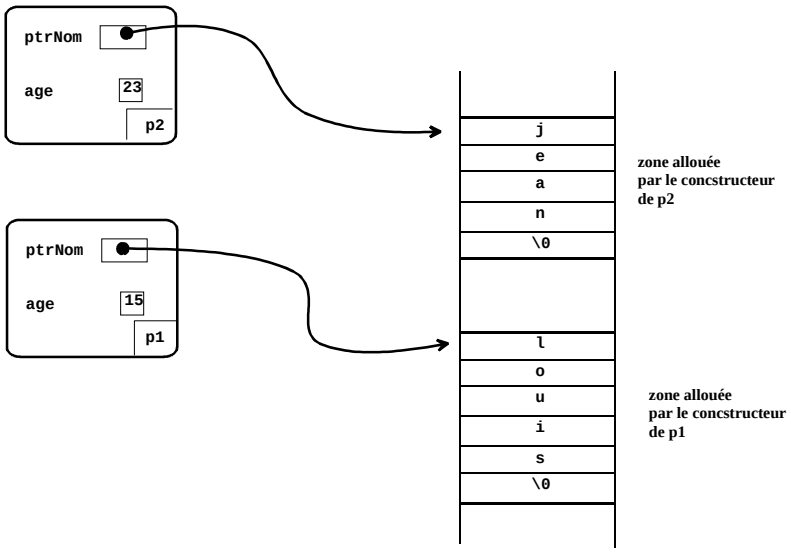
```
personne p1("louis",15),p2(" jean",23) ;
p1 = p2;
```

Le traitement réalisé par l'opérateur = est toujours le même : il réalise une copie membre-à-membre. Les données de `p1` sont recopiées dans `p2`. Cela entraîne que `p2.ptrNom` est recopié dans `p1.ptrNom` ! Autrement dit, les deux pointeurs `p1.ptrNom` et `p2.ptrNom` contiennent maintenant la même adresse, c'est-à-dire pointent sur le même espace mémoire.

Illustrons ce qui se passe lors de cette affectation

1ère étape : construction des objets `p1` et `p2`

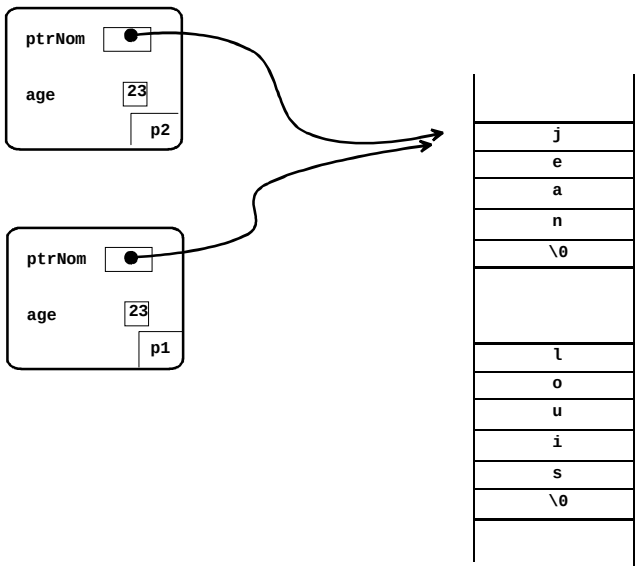
```
personne p1("louis",15),p2("jean",23) ;
```



Chaque objet dispose de sa propre zone de données en profondeur. Jusqu'ici, tout va bien.

2nde étape : affectation par défaut

```
p1 = p2;
```



Comme prévu, les deux pointeurs `p1.ptrNom` et `p2.ptrNom` contiennent la même adresse. Celle de la zone contenant la chaîne `"jean"`.

Quels problèmes cela pose-t-il ?

1) La zone allouée par le constructeur de `p1`, celle qui contient la chaîne `"louis"`**,** ne pourra jamais être rendue au gestionnaire de mémoire. En effet, on a perdu son adresse au cours de l'affectation !

   Cette situation se produit sur le programme suivant.

```
#include "personne_v2.h"

void main()
{
   personne p1("louis",15),p2(" jean",23) ;
   p1 = p2;
}
```

   Cela posera très vite des problèmes lorsque beaucoup d'objets seront construits et que beaucoup d'affectations auront lieu au sein d'un programme. La mémoire sera vite épuisée par ces blocs qui ne sont jamais restitués.

2) Plus grave, supposons que l'objet `p1` soit détruit mais pas `p2`. Au moment de la destruction de `p1`, le destructeur de `p1` restitue la zone mémoire pointée par `p1.ptrNom`. Le pointeur `p2.ptrNom`, qui est égal à `p1.ptrNom`, conserve néanmoins l'adresse de cette zone. L'objet `p2` conserve donc l'adresse d'une zone qui ne lui est plus réservée et qui sera donc utilisée par la suite par d'autres objets.

   Cette situation se produit sur le programme suivant.

```
#include "personne_v2.h"

void main()
{
   personne p2("jean",23) ;

   {
      personne p1("louis",15) ;
      p1 = p2;
   }

}
```

Ici, `p1` est détruit, mais pas `p2`!

4.12 Redéfinition de l'opérateur = pour des objets avec données en profondeur.

4.12.1 Paramétrage de l'opérateur d'affectation « = »

Au vu de ce qui précède, on usera avec prudence de l'affectation par défaut entre objets. Notamment, elle ne convient pas lorsque des objets ont des données en profondeur. Dans ce cas, nous devons redéfinir ce que doit faire l'opérateur d'affectation.

Comme nous le disions dans le chapitre sur les opérateurs (chapitre 3), l'opérateur d'affectation = doit être surchargé dans la classe, c'est-à-dire en tant que fonction membre.

Si l'opérateur = est une fonction membre de la classe `personne`, lorsque l'on écrit `p1 = p2` le compilateur interprète cet appel comme l'appel de la méthode `operator=()` sur l'objet `p1` soit

```
p1.operator=(p2);
```

autrement dit, il cherche une méthode membre de la classe `personne` avec le nom `operator=` et acceptant un argument de type `personne`.

Paramétrage de cet opérateur

. l'opérateur = sera une fonction membre de la classe `personne` et d'identifiant `operator=`
. l'opérateur = recevra un objet de type `personne` comme argument
. l'argument est un paramètre d'entrée (l'affectation ne modifie jamais l'objet à droite du =)

Après cette petite analyse, on en déduit que l'opérateur aura l'entête suivant:

surtout pas de modificateur `const` ici. Ne doit en aucun cas être une méthode constante puisqu'elle modifie l'objet qui l'appelle.

```
void personne::operator=(const personne &p)
```

paramètre d'entrée de type `personne`

la fonction s'appelle operator=

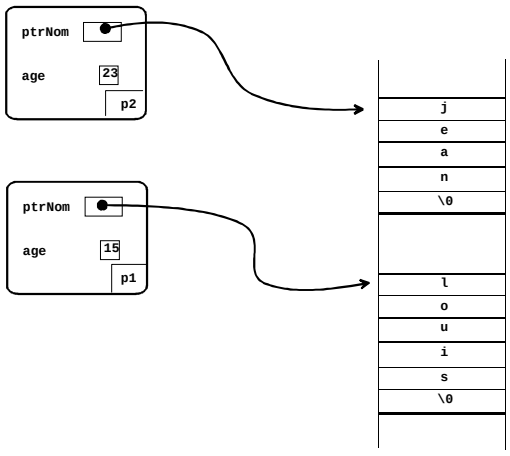
elle appartient à la classe `personne`

4.12.2 Etapes nécessaires à la copie d'objets avec données en profondeur.

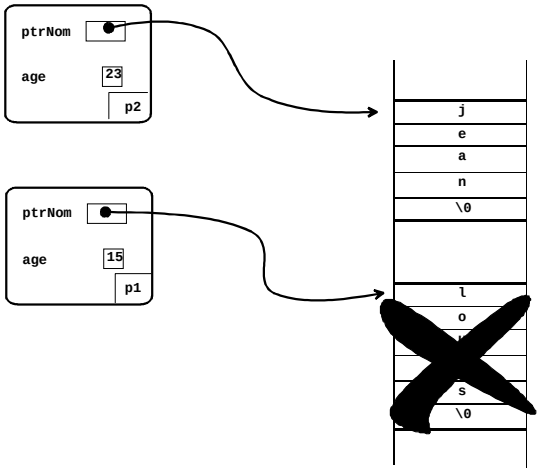
On décrit le traitement que doit réaliser l'opérateur = pour la ligne suivante:

```
p1.operator=(p2);       // qui équivaut à p1=p2;
```

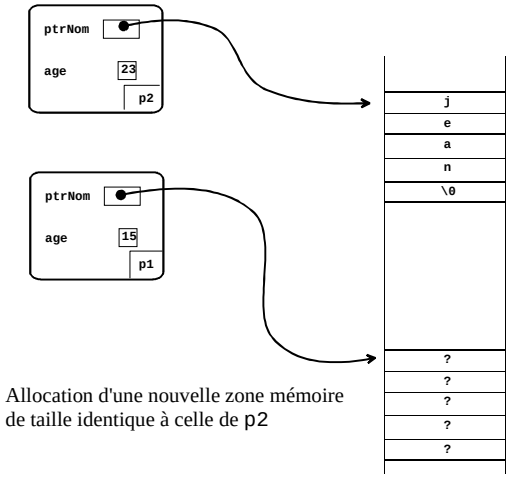
1ère étape : Les objets avant la copie



2ième étape: L'objet « récepteur », c'est-à-dire l'objet courant (`p1`), doit libérer sa zone mémoire

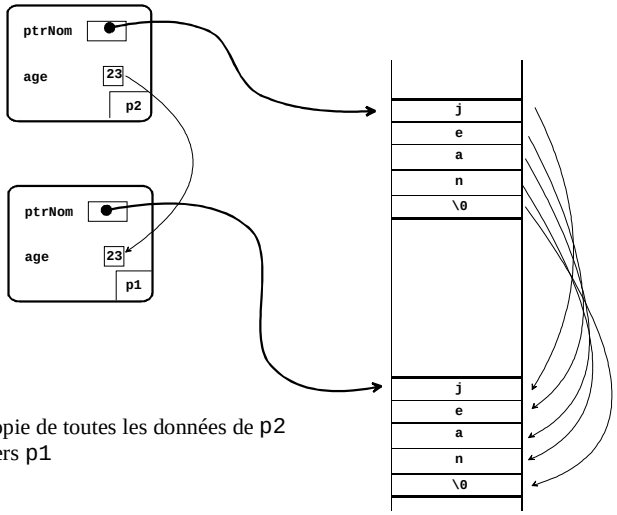


3ième étape : L'objet « récepteur » doit allouer une zone de la même taille que celle de l'objet « donneur »



Allocation d'une nouvelle zone mémoire de taille identique à celle de `p2`

4ième étape : copie de toutes les données, celles situées dans l'objet et celles en profondeur.



copie de toutes les données de `p2` vers `p1`

Cette analyse nous conduit à modifier la classe comme suit :

```
//personne_v4.h
#ifndef __CLS_PERSONNE_V4_
#define __CLS_PERSONNE_V4_

#include <iostream.h>
#include <string.h>
#include <process.h>

class personne
{
private:
      char * ptrNom;
      unsigned age;

public:
      personne(char *,unsigned );
      ~personne(void);

      void operator=(const personne &);

      void identifie() const;
      void setAge(unsigned);
      unsigned getAge() const;
};
```

```
void personne::operator=(const personne &p)
{
   if(this!=&p)                    si l'objet courant est différent de l'objet argument
   {
      delete ptrNom;                          // étape 2
      ptrNom=new char[strlen(p.ptrNom)+1];  // étape 3
      if(ptrNom==NULL)
      {
         cout << "Mémoire insuffisante ... Abandon ...\n";
         exit(1);
      }
      strcpy(ptrNom,p.ptrNom);                  // étape 4 (données en profondeur)
      age=p.age;                                // étape 4 (données dans l'objet)
   }
}

// le reste est identique.
#endif
```

Exemple 28. Surcharge de l'opérateur d'affectation

Pourquoi avoir ajouté `if(this != &p)` ?

En effet, dans cet opérateur, tout le traitement n'est réalisé que si l'adresse `this` (l'adresse de l'objet courant) est différente de l'adresse de l'objet argument. Autrement dit, on ne réalise ce traitement que si l'objet courant est différent de l'objet argument. Pourquoi ?

Si l'on ne conditionne pas le traitement par le test `if(this!=&p)`, en écrivant

```
p1 = p1;
```

l'étape 2 pose problème, puisqu'on supprime en même temps les données de l'objet « donneur » et de l'objet « récepteur » (il n'y a dans ce cas qu'un seul objet !).

4.13 Les objets temporaires

Dans une expression, on peut faire appel explicitement au constructeur d'un objet. Un objet est alors construit. Il s'agit d'un objet temporaire construit pour les besoins d'évaluation de l'expression. Il sera détruit à un moment décidé par le compilateur.

```
#include "personne_v4.h"

void main(void)
{

   cout << " Affichage test 1 ";
   cout <<  endl;

   personne("Sylvie",20).identifie();

   cout << " Affichage test 2 " <<  endl;

}
```

Résultat

```
Affichage test 1
Constructeur de la personne (Sylvie,20)
Je m'appelle Sylvie et j'ai 20 ans
Destructeur de la personne (Sylvie,20)
Affichage test 2
```

Exemple 29. Les objets temporaires

Remarques

La syntaxe `personne("Sylvie",20)` construit un objet temporaire (sans nom). On peut néanmoins accéder à ses méthodes, notamment à la méthode `identifie()`.

La personne ("Sylvie",20) n'a été construite que pour permettre l'évaluation de l'expression

```
personne("Sylvie",20).identifie();
```

On voit que cet objet a été détruit dès que l'évaluation de l'expression a été menée à son terme.

4.14 L'initialisation d'un objet à sa déclaration à l'aide du signe =

On s'intéresse ici à une déclaration d'objet faite sous la forme :

```
type_classe objet = valeur;
```

Il s'agit d'une convention d'écriture (d'ailleurs assez ambiguë). La syntaxe suivante est équivalente :

```
type_classe objet(valeur);
```

Les deux syntaxes permettent la construction de l'objet `objet` en appelant un constructeur à 1 argument de la classe `type_classe`, et qui accepte un argument du type de `valeur`.

Attention : la première syntaxe fait appel à un constructeur et n'utilise donc pas l'opérateur = de la classe `type_classe` !

4.14.1 Exemple

Le programme suivant déclare et utilise une classe appellée `Entier`. Cet exemple vous permet en outre de vérifier que vous maîtrisez quelques points : surcharge des constructeurs, valeurs par défaut, notions de méthode accesseur (qui doit être une méthode constante) et modificateur (qui ne doit surtout pas être une méthode constante).

```
#include <iostream.h>
class Entier
{
   int valeur;
public:
   Entier(int arg=0);   // constructeur (int)
   Entier(double arg);  // constructeur (double)
   ~Entier();           // destructeur
   void Set(int arg);   // mutateur
   int Get() const;     // accesseur
};

Entier::Entier(int arg)
{
   cout << "constructeur Entier(int=0) objet:"<< this <<endl;
   valeur=arg;
}
Entier::Entier(double arg)
{
   cout << "constructeur Entier(double) objet:"<< this <<endl;
   valeur=(int) arg; // opération de conversion de type float->int
}
```

```
Entier::~Entier()
{
   cout << "destructeur ~Entier() objet:"<< this <<endl;
}
void Entier::Set(int arg)
{
   valeur=arg;
}
int Entier::Get() const
{
   return valeur;
}

void main()
{
   Entier i1;
   cout << "i1 = " << i1.Get() << endl;
   Entier i2(3);
   cout << "i2 = " << i2.Get() << endl;
   Entier i3=4.5;
   cout << "i3 = " << i3.Get() << endl;
   Entier i4=2;
   cout << "i4 = " << i4.Get() << endl;
}
```

Résultat

```
constructeur Entier(int=0) objet :0x0066FDE8
i1 = 0
constructeur Entier(int=0) objet :0x0066FDE4
i2 = 3
constructeur Entier(double) objet :0x0066FDE0
i3 = 4
constructeur Entier(int=0) objet :0x0066FDDC
i4 = 2
destructeur ~Entier() objet : 0x0066FDDC
destructeur ~Entier() objet : 0x0066FDE0
destructeur ~Entier() objet : 0x0066FDE4
destructeur ~Entier() objet : 0x0066FDE8
```

Exemple 30. Initialisation des objets avec la syntaxe `TYPE obj=valeur`.

Remarques

On vérifie facilement que les deux syntaxes sont en effet équivalentes. Le constructeur appelé dépend à chaque fois du type de l'argument placé derrière le signe =.

4.14.2 Notion de constructeur par recopie dit aussi « constructeur-copie »

Parmi les constructeurs acceptant un seul argument dont peut disposer une classe, il en est un particulier. Celui qui accepte un argument du type de la classe. C'est le constructeur qui est appelé par exemple lorsque le compilateur rencontre une déclaration du genre :

```
classe objet2(objet1);
```
// où `objet1` est un objet de la classe `classe`.

D'après ce qui vient d'être dit, on s'attend à ce que le compilateur cherche un constructeur de prototype :

```
classe::classe(classe)
```

ou encore (puisque l'argument d'un constructeur est toujours un paramètre d'entrée)

```
classe::classe(const classe &)
```

ou encore des variantes avec des valeurs par défaut pour les paramètres autres que le premier.

En fait, la première syntaxe `classe::classe(classe)` est interdite : on ne peut pas utiliser un tel constructeur. Ceci sera expliqué ultérieurement.

Le constructeur

```
classe::classe(const classe &)
```

porte un nom particulier : le constructeur par recopie**,** encore appelé *constructeur-copie*. Il est chargé de dupliquer la valeur d'un objet existant dans un objet en cours de création.

4.14.3 Le constructeur-copie par défaut

Si la classe `classe` ne définit pas explicitement un constructeur-copie, alors la déclaration

```
classe objet2(objet1);
```

sera néanmoins acceptée. L'objet `objet1` sera recopié membre-à-membre dans `objet2`. Ceci rappelle énormément ce que réalise l'opérateur = par défaut. On dit que cette forme de construction est opérée par le constructeur-copie par défaut. Bien évidemment[4], cette solution ne sera satisfaisante que si les objets ne comportent pas de données en profondeur. Sans quoi, la copie membre-à-membre nous ramène dans la situation où deux objets ont un pointeur sur une même zone de mémoire.

En résumé, si une classe possède des données en profondeur, le constructeur-copie doit être défini explicitement : il aura pour mission de réserver une zone de mémoire de même taille que celle de l'objet dont il sera la copie, puis de réaliser la copie des données (en profondeur et en surface).

4.14.4 Exemple : ajout d'un constructeur-copie à la classe `personne`

On va, une fois de plus, compléter la classe `personne` (avec données en profondeur) pour illustrer ce point.
On ne rappelle pas toute la définition. On a néanmoins laissé le code de l'opérateur = pour montrer les différences qui existent entre le constructeur-copie et l'opérateur d'affectation (on fait trop souvent l'amalgame) .

Différences entre constructeur-copie et opérateur=

. lorsque l'on construit un objet, celui-ci n'a encore alloué aucune zone mémoire. Il n'y a pas de désallocation dans un constructeur-copie

. il n'est pas nécessaire de vérifier si l'argument et l'objet courant sont identiques puisque la ligne suivante n'est pas compilée

```
personne p1(p1);
```

Ce qui est identique pour les deux méthodes

. allocation d'une zone mémoire de même taille que celle de l'argument

. copie des données (en surface et en profondeur) de l'objet argument vers l'objet courant

```
//personne_v5.h

#ifndef __CLS_PERSONNE_V5_
#define __CLS_PERSONNE_V5_

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

[4] si ce n'est pas une évidence, je vous invite à revoir ce que l'on a dit sur l'opérateur = par défaut pour des objets avec des données en profondeur (§ 4.12).

```
class personne
{
private:
      char * ptrNom;
      unsigned age;
public:
      personne(char *,unsigned );

      personne(const personne & );       // constructeur-copie

      ~personne(void);
      void operator=(const personne &);
      void identifie() const;
      void setAge(unsigned);
      unsigned getAge() const;
};

personne::personne(const personne &p)    //constructeur-copie
{
      ptrNom=new char[strlen(p.ptrNom)+1];  // allocation
      if(ptrNom==NULL)
      {
         cout << "Mémoire insuffisante ... Abandon ...\n";
         exit(1);
      }
      strcpy(ptrNom,p.ptrNom);              // copie (données en profondeur)
      age=p.age;                            // copie (données en surface)
      cout << "Le constructeur-copie de la personne (" << ptrNom << "," << age << ") a été
appelé\n";
}

void personne::operator=(const personne &p) //opérateur=
{
   if(this!=&p)
   {
      delete ptrNom;
      ptrNom=new char[strlen(p.ptrNom)+1];
      if(ptrNom==NULL)
      {
         cout << "Mémoire insuffisante ... Abandon ...\n";
         exit(1);
      }
      strcpy(ptrNom,p.ptrNom);
      age=p.age;
   }
}

// … le reste est identique.

#endif
```

Exemple 31. Définition d'un constructeur-copie pour la classe `personne`

Cette modification autorise désormais l'exécution correcte et fiable du programme suivant

```
#include "personne_v5.h"

void main(void)
{
   personne agent1("Jacques",20) ;
   personne agent2=agent1;      // utilise le constructeur-copie explicitement défini
}
```

4.15 Passage/retour d'un objet à/par une fonction

Nous allons observer dans cette partie quels mécanismes le C++ met en œuvre lorsque des objets sont passés à des fonctions ou retournés par des fonctions.

Pour tout ce qui suit, nous allons utiliser une nouvelle version de la classe `personne` complétée par certaines méthodes : `setName()`, `getName()`. La méthode d'affichage d'un objet, nommée `identifie()` jusqu'à présent, a été renommée `affiche()` dans cette version.

```
//personne_v6.h

#ifndef __CLS_PERSONNE_V6
#define __CLS_PERSONNE_V6

#include <iostream.h>
#include <string.h>
#include <process.h>

//déclaration/définition de la classe

class personne
{
private:
      char* ptrNom;
      unsigned age;
public:
      personne(const char*="",unsigned=0);
      personne(const personne &);

      ~personne(void);

      void operator=(const personne &);

      void affiche() const;

      void setAge(unsigned);     //modificateur
      unsigned getAge() const;   // accesseur

      void setName(const char *);   //modificateur
      const char * getName() const; //accesseur
};
```

```
//définition des fonctions membre
personne::personne ( const char *nom_p, unsigned age_p)
{
   ptrNom=new char[strlen(nom_p)+1];
   if(ptrNom==NULL)
   {
      cout << "Mémoire insuffisante ... Abandon ...\n";
      exit(1);
   }
   strcpy(ptrNom,nom_p);
   age=age_p;
   //sortie :personne(const char*="",unsigned=0)  objet: adresse  (nom,age)
   cout << "personne(const char*=\"\",unsigned=0)";
   cout << "\t objet : " << this << " \t("<< ptrNom << "," << age << ")"<<endl;
}

personne::personne(const personne &p)
{
      ptrNom=new char[strlen(p.ptrNom)+1];
      if(ptrNom==NULL)
      {
         cout << "Mémoire insuffisante ... Abandon ...\n";
            exit(1);
      }
      strcpy(ptrNom,p.ptrNom);
      age=p.age;
      //sortie : personne(const personne &) objet: adresse  (nom,age)
      cout << "personne(const personne &)";
      cout << "\t\t objet : " << this;
      cout << " \t("<< ptrNom << "," << age << ")"<<endl;
}

void personne::operator=(const personne &p)
{
   if(this!=&p)
   {
      delete ptrNom;
      ptrNom=new char[strlen(p.ptrNom)+1];
      if(ptrNom==NULL)
      {
         cout << "Mémoire insuffisante ... Abandon ...\n";
            exit(1);
      }
      strcpy(ptrNom,p.ptrNom);
      age=p.age;
   }
}

void personne::affiche(void)  const
{
   //sortie : affiche() objet: adresse (nom,age)
   cout << "affiche() \t\t \t \t objet :"<<this;
   cout << "\t("<< ptrNom <<","<< age << ")"<<endl;
}
```

```
void personne::setAge(unsigned a)
{
   age=a;
}

unsigned personne::getAge() const
{
   return age;
}

void personne::setName(const char * str)
{
   delete this->ptrNom;
   this->ptrNom=new char[strlen(str)+1];
   strcpy(ptrNom,str);
}

const char * personne::getName() const
{
   return ptrNom;
}

personne::~personne()
{
   // ~personne()       objet : adresse
   cout << "~personne() \t\t\t\t objet :"<<this << endl;
   delete ptrNom;
}

#endif
```

Exemple 32. Sixième version de la classe `personne`

4.15.1 Passage d'un objet par valeur à une fonction

Observons l'exécution du programme suivant :

```
#include "personne_v6.h"

   void recopie(personne);

void main()
{
   personne agent("Louis",15);
   recopie(agent);
   agent.affiche();
}



void recopie (personne p)
{
   p.setName("Jean");
   p.affiche();
}
```

Résultats écran

```
personne(const char*="",unsigned =0) objet :0x0065FDE4 (Louis,15)
personne (const personne &)          objet :0x0065FD88 (Louis,15)
affiche()                            objet :0x0065FD88  (Jean,15)
~personne                            objet :0x0065FD88
affiche()                            objet :0x0065FDE4 (Louis,15)
~personne                            objet :0x0065FDE4
```

Exemple 33. Passage d'un objet par valeur

Ce programme utilise une fonction `recopie()` dont le seul argument est un objet de la classe `personne` passé par valeur.

L'exemple précédent montre le mécanisme de recopie de l'objet `agent`, variable locale de `main()`, dans l'objet `p`, variable locale de la fonction `recopie()`. L'objet `p` est initialisé avec la valeur de `agent`.

Pourquoi ces différents affichages ?

Il y a 6 sorties écran en tout.

1ière sortie écran : l'objet `agent` est construit en passant deux arguments au constructeur. Le premier de type `char *` et le second de type `unsigned`. Le premier affichage est donc produit par le constructeur

```
personne::personne(const char*="",unsigned=0)
```

2nde sortie écran : l'objet `agent` est passé à la fonction `recopie`(). Puisqu'il s'agit d'un passage par valeur, le paramètre formel `p` de la fonction `recopie()` est différent du paramètre d'appel `agent`. En fait, il faut voir le paramètre formel `p` comme un objet local à la fonction `recopie()`. L'objet `p` est donc contruit à chaque appel de la fonction. En outre, puisque `p` doit être une copie du paramètre d'appel (car passage par valeur), c'est le constructeur-copie qui est ici appelé pour dupliquer l'objet agent dans p. La seconde sortie écran est donc la construction de l'objet p comme copie de l'objet agent. Il y a donc appel du constructeur

```
personne::personne(const personne &)
```

On remarque d'ailleurs que l'adresse de p (0x0065FD88) est différente de l'adresse de agent (0x0065FDE4)

3ième sortie écran : au sein de la fonction `recopie()`, l'objet `p` (local à la fonction) est tout d'abord modifié par l'instruction `p.setName("Jean")` puis affiché. La 3ième sortie écran correspond donc à l'affichage de `p`, c'est-à-dire à l'appel de la méthode

```
void personne::affiche() const
```

sur l'objet `p`.

4ième sortie écran : lorsque l'exécution de la fonction `recopie()` se termine, l'objet `p` (local à la fonction) est détruit. Il y a donc appel du destructeur de `p`

```
personne::~personne()
```

5ième sortie écran : on affiche l'objet `agent`. Très naturellement, ce dernier n'a pas changé.

6ième sortie écran : l'objet `agent` est détruit à la fin de la fonction `main`.

Que retenir de cet exemple ?

Lorsqu'un objet est passé à une fonction par valeur, il y a appel du constructeur-copie pour créer le paramètre manipulé par la fonction (local à la fonction).

Remarque

Si le constructeur-copie n'est pas explicitement défini, c'est le constructeur-copie par défaut qui est appelé. Il est donc indispensable de redéfinir le constructeur-copie lorsque les objets ont des données en profondeur, sans quoi tous les passages d'objets par valeur conduiront à des fonctionnements erratiques.

4.15.2 Passage d'un objet à une fonction par référence

Afin d'illustrer la différence entre *passage par valeur* et *passage par référence* pour un objet, nous reprenons l'exemple précédent en effectuant un passage par référence. La fonction `pas_de_recopie(personne &)` reçoit un argument de type `personne` passé par référence.

```
#include "personne_v6.h"
   void pas_de_recopie(personne &);
void main()
{
   personne agent("Louis",15);
   pas_de_recopie(agent);
   agent.affiche();
}

void pas_de_recopie (personne & p)
{
   p.setName("Jean");
   p.affiche();
}
```

Résultats écran

```
personne(const char*="",unsigned=0)  objet :0x0065FDE4  (Louis,15)
affiche()                            objet :0x0065FDE4  (Jean,15)
affiche()                            objet :0x0065FDE4  (Jean,15)
~personne                            objet :0x0065FDE4
```

Exemple 34. Passage d'un objet par référence

Remarque : clairement, il n'y a qu'un seul objet de construit au cours de ce programme : il s'agit de l'objet `agent`. Dans le passage par référence, l'objet `p` n'est qu'une référence. Il n'engendre donc pas de construction de nouvel objet comme c'est le cas lors d'un passage par valeur. De plus, on voit bien que la modification de la référence `p` par la méthode `setName()` a modifié l'objet `agent`, puisqu'il s'agit du même objet.

Remarque : l'argument de la fonction `pas_de_recopie()` est ici un paramètre de sortie, puisque l'objet passé à la fonction est effectivement modifié par celle-ci.

4.15.3 Retour d'un objet par valeur

Un autre cas "invisible" d'initialisation d'objet est celui où une fonction rend un objet par valeur. Examinons l'exemple suivant :

```
#include "personne_v6.h"
           personne recopie(personne &);
void main()
{
   personne agent("Louis",15);
   recopie(agent);
   agent.affiche();
}
personne recopie (personne & p)
{
   p.setName("Charles");
   p.affiche();
   return p;
}
```

Résultats écran

```
personne(const char*="",unsigned=0) objet :0x0065FDE4 (Louis,15)
affiche()                           objet :0x0065FDE4  (Charles,15)
personne (const personne &)         objet :0x0065FDDC  (Charles,15)
~personne                           objet :0x0065FDDC
affiche()                           objet :0x0065FDE4  (Charles,15)
~personne                           objet :0x0065FDE4
```

Exemple 35. Retour d'un objet par valeur

La fonction `recopie()` reçoit une référence : le paramètre effectif `agent` et le paramètre formel `p` vont donc désigner la même personne. La fonction `recopie()` change la valeur de `p`, donc `d'agent`, par la méthode `setName()`, puis rend la valeur de `p`.

Pourquoi ces sorties écran ?

1ière sortie écran : l'objet `agent` est construit en passant deux arguments au constructeur.

2nde sortie écran : la référence `p`, c'est-à-dire l'objet `agent`, est modifiée puis affichée.

On remarque d'ailleurs que l'adresse de `p` (0x0065FDE4) est identique à celle de `agent`.

3ième et 4ième sortie écran : l'objet `p` est ici retourné par valeur. Ce n'est donc pas vraiment l'objet `p` qui est retourné mais une copie de `p`. C'est pourquoi un objet est construit. Il s'agit de l'objet retourné par la fonction. Il y a donc construction, puis destruction, d'un objet dont l'adresse est 0x0065FDDC. Il y a ici appel du constructeur-copie pour créer cet objet qui est une copie de l'objet `p` (et donc de l'objet `agent`).

Remarque : même si l'objet retourné par la fonction n'est pas utilisé, il est néanmoins construit puis détruit.

5ième sortie écran : on affiche l'objet `agent`. On vérifie ainsi qu'il a bien été modifié au sein de la fonction.

6ième sortie écran : l'objet `agent` est détruit à la fin de la fonction `main()`.

4.15.4 Utilisation d'un objet retourné par valeur par une fonction

Dans l'exemple précédent, nous n'utilisions pas la valeur retournée par la fonction `recopie()`. Nous pourrions par exemple l'utiliser de la façon suivante :

```
#include "personne_v6.h"

   personne recopie(personne &);

void main()
{
   personne agent("Louis",15);

   recopie(agent).affiche();

   agent.affiche();
}

personne recopie (personne & p)
{
   p.setName("Charles");
   p.affiche();
   return p;
}
```

Résultats écran

```
personne (const char*="",unsigned =0) objet :0x0065FDE4 (Louis,15)
affiche()                             objet :0x0065FDE4   (Charles,15)
personne (const personne &)           objet :0x0065FDDC   (Charles,15)
affiche()                             objet :0x0065FDDC   (Charles,15)
~personne                             objet :0x0065FDDC
affiche()                             objet :0x0065FDE4   (Charles,15)
~personne                             objet :0x0065FDE4
```

On voit qu'il y a un affichage supplémentaire : celui de l'objet retourné par la fonction `recopie()`.

4.15.5 Retour d'un objet par référence

Nous reprenons l'exemple précédent en rendant l'objet *par référence***.** Le programme devient celui-ci :

```
#include "personne_v6.h"

   personne & pas_de_recopie(personne &);

void main()
{
   personne agent("Louis",15);
   pas_de_recopie(agent).setName("Jean");
   agent.affiche();
}

personne& pas_de_recopie (personne & p)
{
   p.setName("Charles");
   p.affiche();
   return p;
}
```

Résultats écran

```
personne (const char*="",unsigned =0) objet :0x0065FDE4    (Louis,15)
affiche()                             objet :0x0065FDE4    (Charles,15)
affiche()                             objet :0x0065FDE4    (Jean,15)
~personne                             objet :0x0065FDE4
```

Exemple 36. Retour d'un objet par référence

Remarque

Il n'y a plus qu'un objet de construit au cours du programme : l'objet `agent`. Ce dernier est tout d'abord construit. D'où le premier affichage

personne (const char="",unsigned =0) objet :0x0065FDE4 (Louis,15)*

puis modifié au sein de la fonction `pas_de_recopie()`,

affiche() objet :0x0065FDE4 (Charles,15)

Mais, l'objet `agent` est également modifié au moment du retour de la fonction. La ligne

```
pas_de_recopie(agent).setName("Jean");
```

signifie que l'on modifie le nom de l'objet retourné par la fonction `pas_de_recopie()`. Puisqu'il s'agit d'un retour par référence, la fonction retourne une référence à `p` qui est elle-même une référence à `agent`. Par transitivité, la fonction retourne donc une référence à `agent`, ce qui permet de modifier l'objet `agent`.

Le troisième affichage confirme que l'objet `agent` a bel et bien été modifié.

4.15.6 Pourquoi l'argument du constructeur-copie ne peut-il pas être passé par valeur ?

Pourquoi le constructeur-copie de la classe personne ne peut-il pas être déclaré de la manière suivante ?

```
personne::personne(personne p){…}
```

Au vu de ce que l'on a dit précédemment, si tel était le cas, le passage de l'argument par valeur au constructeur-copie déclencherait un autre appel du constructeur-copie pour faire la copie de l'argument, et ainsi de suite récursivement.

Autrement dit, une telle déclaration conduit à un nombre d'appels infini du constructeur-copie. En pratique, cela s'arrête quand la pile déborde !

4.15.7 Conclusion

Soit une classe `C`. Le constructeur de prototype

```
C(const C &)
```

est appelé le *constructeur de recopie* ou *constructeur-copie*. S'il existe, il est appelé dans les cas suivants :

. initialisation d'un objet par un autre objet dans une déclaration.

. initialisation d'un paramètre objet formel par un paramètre objet effectif lors du passage par valeur d'un objet à une fonction.

. retour d'un objet par valeur.

S'il n'est pas explicitement défini, dans les deux cas précédents, une recopie bit à bit est effectuée par le constructeur-copie par défaut. Ce mécanisme par défaut n'est toutefois pas convenable si les objets contiennent des données en profondeur.

4.16 Les tableaux d'objets

4.16.1 Déclaration et initialisation d'un tableau d'objets

La classe généralise la notion de type en C++. Il est donc possible de déclarer des tableaux d'objets. Se pose néanmoins la difficulté de la construction et de la destruction des objets d'un tableau. Illustrons ce point par un exemple.

En s'appuyant sur la version 6 de la classe `personne`, examinons l'exécution du programme suivant :

```
#include "personne_v6.h"

void main()
{
   personne tableau[3];
```

Résultats écran

```
personne (const char*="",unsigned =0) objet :0x0065FDE0    (,0)
personne (const char*="",unsigned =0) objet :0x0065FDE8    (,0)
personne (const char*="",unsigned =0) objet :0x0065FDF0    (,0)
~personne                             objet :0x0065FDF0
~personne                             objet :0x0065FDE8
~personne                             objet :0x0065FDE0
```

```
}
```

La fonction `main()` ne contient que la déclaration d'un tableau de 3 objets de type `personne`.

On remarque que tous les objets du tableau font appel à un constructeur**.** Ici, en l'absence d'informations supplémentaires, seul le constructeur pouvant être appelé sans argument est utilisé. En C++, tout objet fait nécessairement appel à un constructeur, y compris dans un tableau On remarque que naturellement tous les objets du tableau sont également détruits lorsque la fonction `main()` se termine.

Remarque : pour pouvoir construire un tableau avec cette syntaxe, il est donc indispensable que la classe contienne un constructeur pouvant être appelé sans argument. Ici, c'est rendu possible par la présence d'arguments avec valeurs par défaut.

Une syntaxe différente, également utilisée en C, permet de passer des informations aux constructeurs de chacun des objets du tableau.

```
#include "personne_v6.h"

void main()
{
   personne agent("Louis",15);
   personne tableau[3]={"Louis",personne("Charles",9),agent};
```

Résultats écran

```
personne (const char*="",unsigned =0) objet :0x0065FDE4    (Louis,15)
personne (const char*="",unsigned =0) objet :0x0065FDCC    (Louis,0)
personne (const char*="",unsigned =0) objet :0x0065FDD4    (Charles,19)
personne (const personne & )          objet :0x0065FDDC    (Louis,15)
~personne                           objet :0x0065FDDC
~personne                           objet :0x0065FDD4
~personne                           objet :0x0065FDCC
~personne                           objet :0x0065FDE4
```

```
}
```

Le premier affichage résulte de la construction de l'objet `agent`.

Pour les trois affichages suivants, il s'agit de l'appel des constructeurs des 3 objets du tableau.

L'objet `tableau[0]` est construit avec le constructeur à deux arguments, le second argument prenant pour valeur la valeur par défaut 0.

L'objet `tableau[1]` est construit avec le constructeur à deux arguments `"Charles"` et 9.

L'objet `tableau[2]` est construit avec le constructeur-copie. Il est construit comme une copie de l'objet `agent`.

4.16.2 Utilisation de la syntaxe `delete[]`

Lorsque l'on a présenté l'opérateur `delete`, nous avons précisé qu'il existait une seconde syntaxe `delete[]` d'utilisation de cet opérateur, sans pouvoir en expliquer l'utilisation.

```
#include "personne_v6.h"
void main()
{
   personne * ptr;
   ptr=new personne[3];
   delete ptr;
}
```

Résultats écran

```
personne (const char*="",unsigned =0) objet :0x0065FD84    (,0)
personne (const char*="",unsigned =0) objet :0x0065FD8C    (,0)
personne (const char*="",unsigned =0) objet :0x0065FD94    (,0)
~personne                           objet :0x0065FD84
```

Dans cet exemple, on déclare un pointeur sur un objet de type `personne`. En réalité, on y stocke l'adresse d'un tableau de 3 objets. Puisqu'aucune information ne peut être passée au constructeur de chacun de ces 3 objets, c'est le constructeur qui peut être appelé sans argument qui est utilisé.

Remarque : pour pouvoir utiliser la syntaxe `new personne[n]`, la classe doit avoir un constructeur sans argument.

Que remarquons-nous ? Lorsque l'on fait `delete ptr`, seul un objet est correctement détruit. Les autres objets ne sont pas détruits.

Pourquoi ?

La ligne `delete ptr;` rend la mémoire allouée par la commande `new personne[3]`, c'est-à-dire rend au gestionnaire de mémoire la zone occupée par les données des 3 objets. Par contre, seul un des destructeurs de ces trois objets est appelé, celui de l'objet `ptr[0]` (le premier objet du tableau). Autrement dit, la mémoire allouée pour les données en profondeur des objets `ptr[1]` et `ptr[2]` n'est pas restituée.

Si l'on utilise la seconde syntaxe de l'opérateur `delete`, il n'y a plus de problème.

```
#include "personnev6.h"

void main()
{
   personne * ptr;
   ptr=new personne[3];

   delete [] ptr;
}
```

Résultats écran

```
personne (const char*="",unsigned =0) objet :0x0065FD84    (,0)
personne (const char*="",unsigned =0) objet :0x0065FD8C    (,0)
personne (const char*="",unsigned =0) objet :0x0065FD94    (,0)
~personne                             objet :0x0065FD94
~personne                             objet :0x0065FD8C
~personne                             objet :0x0065FD84
```

Exemple 37. Restitution d'un tableau d'objets alloué dynamiquement par la syntaxe `delete []`

5 ELABORATION D'UNE CLASSE DE CHAINES DE CARACTERES.

5.1 Introduction

Dans ce chapitre, nous présentons l'élaboration d'une classe. Ce chapitre va servir de synthèse des éléments présentés précédemment. Cette élaboration permettra également de revenir un peu sur la surdéfinition d'opérateurs en tant que membres de classes.

Nous souhaitons créer une classe de chaînes de caractères qui aurait les avantages des chaînes de caractères des langages Pascal et C mais pas leurs inconvénients. Présentons les uns et les autres :

Les chaînes de caractères en Pascal

Inconvénient : les chaînes sont limitées en taille à 255 caractères.

Avantages : elles sont d'un usage simple et sûr.

- En Pascal, on peut réaliser l'opération

```
chaine1=chaine2
```

qui recopie `chaine1` dans `chaine2`. Si `chaine2` déborde de `chaine1`, elle est tronquée.

- On peut comparer directement deux chaînes avec les opérateurs habituels : `<, <=, >, >=, =, <>`
- On peut concaténer deux chaînes par l'opération

```
chaine1+chaine2
```

Les chaînes de caractères en C

Avantage : les chaînes ne sont pas limitées en taille.

Inconvénients :

- Aucune vérification de débordement n'est faite.
- La copie de chaînes nécessite l'appel d'une fonction

```
strcpy(chaine1,chaine2)
```

- Si `chaine2` est plus longue que `chaine1`, il y aura débordement.
- On ne peut comparer deux chaînes avec les opérateurs habituels : il faut passer par les fonctions `strcmp` (string compare).
- On ne peut concaténer deux chaînes par l'opération

```
chaine1+chaine2
```

Il faut passer par la fonction `strcat`. Il y a là aussi risque de débordement.

- La lecture d'une chaîne par la fonction standard `gets` (la plus utile puisqu'elle accepte les blancs dans la chaîne, ce que ne fait pas `scanf`) peut là encore entraîner un débordement.

5.2 L'utilisation et l'interface de la classe

La conception d'une classe commence généralement par spécifier quelle interface on en attend. On présente donc tout d'abord un programme d'utilisation de la classe (qui n'est toutefois pas encore écrite).

```
#include "chaine.h"

void main()
{

   chaine s1;                // chaîne vide
   chaine s2="une chaine ";  // chaîne initialisée par une chaîne du C
   chaine s3=s2;             // chaîne initialisée par copie

   cout << s3 << endl;       // affichage d'une chaîne à l'écran

   chaine s4;

   s4=chaine("de caracteres");  // affectation entre deux objets de type chaine
   s2=s4;
   cout << s2 << endl;

   s2 = s3 + s4;                // concaténation de deux chaines
   cout << s2 << endl;

   s2[0]='C';                   // une chaîne sera vue également comme un tableau
   cout << s2 << endl;

   s3=s2;
   if(s3==s1) cout << "chaines identiques" << endl;
}
```

Exemple 38. Spécification pour la classe `chaine`

- On doit pouvoir déclarer :
  - une chaîne vide (objet `s1`)
  - une chaîne initialisée par une chaîne du langage C (objet `s2`)
  - une chaîne initialisée par un autre objet de type `chaine` (objet `s3`)

- On va écrire la classe `chaine` de façon à pouvoir afficher une chaîne de manière homogène avec celle utilisée par le C++, c'est-à-dire en surdéfinissant l'opérateur `<<`.

```
cout << s3 << endl;       // affichage d'une chaîne à l'écran
```

- On va définir l'opérateur = pour l'affectation et l'opérateur + pour la concaténation de chaînes.

```
s2=s4;                    // affectation entre chaînes

s2 = s3 + s4;             // concaténation de deux chaines
```

- On va munir les chaines d'un accès direct, comme s'il s'agissait aussi de tableaux.

- On va munir la classe `chaine` d'un opérateur de test d'égalité entre deux chaînes (opérateur ==).

```
if(s3==s1) cout << "chaines identiques" << endl;
```

5.3 Les données de la classe `chaine`.

On analyse ici quelles données doivent être stockées et comment les stocker.

Il est clair que chaque objet de la classe `chaine` doit disposer d'une zone de mémoire (un tableau) permettant de stocker de manière contiguë un ensemble de caractères. Au vu de ce que l'on a déjà rencontré dans les chapitres précédents, il y a deux solutions :

. chaque objet dispose d'un tableau de taille identique pour tous les objets

. chaque objet se charge d'allouer/restituer une zone de données de taille en adéquation avec les besoins de l'objet.

La première solution conduirait à une déclaration du type

```
class chaine
{
private:
   char str[n];   // avec n une constante
public:
   // etc.
};
```

Cette première solution n'est absolument pas satisfaisante pour les raisons suivantes :

. une chaîne aurait nécessairement une taille limitée (c'est le problème des chaînes en Pascal que l'on voulait résoudre)
. on gaspille de l'espace mémoire lorsque les chaînes stockées sont de faible taille.

Seule la deuxième solution est satisfaisante. Elle requiert pour chaque objet d'allouer de la mémoire au moment de la construction et de la restituer au moment de la destruction. Ainsi, les chaînes n'auront plus d'autres limites de taille que celle imposée par la mémoire disponible dans le tas.

Chaque objet doit donc disposer d'une variable pointeur permettant de mémoriser l'adresse d'un tableau de caractères. En outre, le caractère de fin de chaîne du C (caractère '\0' de valeur numérique 0) n'est plus indispensable si l'on mémorise également la taille de la chaîne.

La déclaration de la classe `chaine` est alors

```
class chaine
{
private:
   char * _ptrChaine;      // adresse du tableau de caractères
   int _taille;            // taille du tableau
public:
};
```

Note : certains programmeurs identifient les données par des noms commençant pas le caractère « _ ». Ceci permet d'utiliser le même nom sans caractère « _ » comme nom de méthode : ici on pourra appeler `taille()` la méthode renvoyant le nombre de caractères d'une chaîne.

5.4 Les constructeurs et le destructeur de la classe `chaine`

Le programme d'utilisation donné en début de chapitre oriente la façon d'écrire les constructeurs. Il doit y avoir un constructeur sans argument (ou avec des arguments ayant des valeurs par défaut), un constructeur avec argument de type `char *`, un constructeur-copie et un destructeur.

Ceci conduit à l'ébauche suivante :

```
class chaine
{
private:
   char * _ptrChaine;
   int _taille;
public:
   chaine();
   chaine(const char *);
   chaine(const chaine &);
   ~chaine();
};
```

Remarque : on pourrait également définir un constructeur avec un argument ayant une chaîne vide comme argument par défaut :

```
class chaine
{
private:
   char * _ptrChaine;
   int _taille;
public:
   chaine(const char *="");
   chaine(const chaine &);
   ~chaine();
};
```

Nous ne développerons pas cette dernière version qui ne présente toutefois aucune difficulté supplémentaire.

Les définitions des constructeurs et du destructeur sont les suivantes :

Le constructeur sans argument (construction d'une chaîne vide)

```
chaine::chaine()
{
   _ptrChaine=NULL;  // facultatif
   _taille=0;
}
```

Le constructeur avec un argument de type pointeur sur caractère.

```
chaine::chaine(const char * str)
{
   _taille=strlen(str);
   _ptrChaine=new char[_taille];   // allocation d'un tableau

   if(!_ptrChaine) // si _ptrChaine==0
   {
      cerr << "Manque de mémoire..." << endl;
      exit(1);
   }
   memmove(_ptrChaine,str,_taille*sizeof(char));  // copie de la chaîne
}
```

Remarque :

. l'argument de ce constructeur est un paramètre d'entrée (d'où l'emploi du `const`)

. si l'allocation n'est pas possible, l'opérateur `new` retourne le pointeur `NULL` (valeur 0). Si tel est le cas, on quitte le programme en affichant un message d'erreur sur le flot d'erreur `cerr`.

. la fonction `memmove()` permet la copie d'une zone mémoire vers une autre. En effet, puisque l'on a supprimé le caractère de fin de chaîne, il est exclus de pouvoir utiliser la fonction `strcpy()`.

Le constructeur-copie

```
chaine::chaine(const chaine & c)
{
   _taille=c._taille;
   _ptrChaine=new char[_taille];

   if(!_ptrChaine)
   {
      cerr << "Manque de mémoire..." << endl;
      exit(1);
   }
   memmove(_ptrChaine,c._ptrChaine,_taille*sizeof(char));
}
```

Le destructeur

```
chaine::~chaine()
{
   delete _ptrChaine;
}
```

5.5 Factorisation d'un traitement.

On voit dans la définition des méthodes précédentes que les traitements réalisés par les constructeurs présentent certaines similitudes. Aussi, il paraît intéressant de regrouper tous les traitements semblables au sein d'une même fonction dont l'accès sera restreint à la classe. On appelle cette technique une *factorisation de code*.

La factorisation concerne ici l'allocation plus la copie d'une chaîne de caractères. On ajoute donc une méthode `allouеAffecte()` à la classe. Il s'agit d'une méthode privée car seuls les objets de la classe en ont besoin. Il serait même risqué d'en laisser l'accès public.

Ceci conduit aux modifications suivantes :

```
class chaine
{
private:
   char * _ptrChaine;
   int _taille;
   void allouеAffecte(const char *,int); // allocation + copie
public:
   chaine();
   chaine(const char *);
   chaine(const chaine &);
   ~chaine();
};
```

Les définitions des fonctions membres évoluent de la manière suivante

```
// factorisation de l'allocation + la copie (méthode privée)
void chaine::alloueAffecte(const char * p,int t)
{
   _ptrChaine = new char [_taille=t];
   if(_ptrChaine==NULL)
   {
      cerr << "erreur d'allocation ";
      exit(1);
   }
   memmove(_ptrChaine,p,t);
}

//constructeur sans argument
chaine::chaine()
{
   _ptrChaine=NULL;
   _taille=0;
}

// constructeur avec un argument de type pointeur sur char
chaine::chaine(const char * str)
{
   alloueAffecte(str,strlen(str));
}

// constructeur-copie
chaine::chaine(const chaine & c)
{
   alloueAffecte(c._ptrChaine,c._taille);
}

// destructeur
chaine::~chaine()
{
   delete _ptrChaine;
}
```

5.6 Les opérateurs membres ou non membres d'une classe.

On fait ici une petite parenthèse sur les opérateurs pour pouvoir poursuivre l'écriture de la classe `chaine`.

On sait qu'il est possible de surcharger les opérateurs pour les nouveaux types. On l'a par exemple appliqué sur des structures et cela se généralise sans problème aux classes. Dans le chapitre 3, les opérateurs apparaissaient comme des fonctions non membres d'une classe.

Lorsque des opérateurs s'appliquent à des objets d'une classe, il est possible de les faire apparaître comme fonctions membres de celle-ci. On a d'ailleurs illustré ceci en surchargeant l'opérateur = pour l'affectation entre objets de type `personne`.

En pratique, mis à part les quelques opérateurs qu'il est nécessaire de faire apparaître comme membres d'une classe (voir chap. 3), on a toujours *le choix* de surcharger un opérateur *soit comme fonction membre* d'une classe, *soit comme fonction non membre*. (c'est-à-dire une fonction « classique » ne dépendant d'aucune classe). Il convient de prendre un exemple pour comprendre ceci.

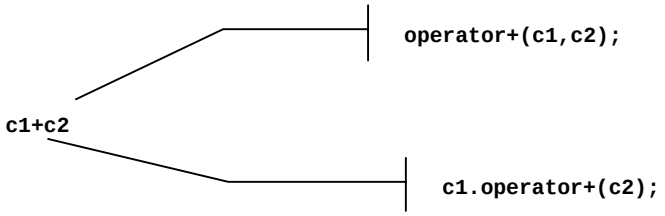
Par exemple, si `C` est un type classe et que `c1` et `c2` sont deux objets de la classe `C`,

```
C c1,c2;
```

lorsque le compilateur rencontre une ligne telle que

```
c1+c2;
```

celui-ci a deux façons de pouvoir interpréter cet appel d'opérateur.



Première possibilité : il s'agit d'une fonction `operator+( , )` à deux arguments de type `C` (ou référence à `C`) puisque l'opérateur + est ici un opérateur binaire. Cette fonction n'est membre d'aucune classe.

Seconde possibilité : il s'agit d'une fonction membre de la classe `C` et qui est appelée sur l'objet `c1` (l'opérande de gauche). Cette méthode n'a plus qu'un seul argument (le 2^{ième} opérande). Le premier opérande étant alors l'objet courant.

Le concepteur de la classe C choisira donc entre ces deux possibilités.

Ce qui est dit ici vaut également pour tous les opérateurs binaires qu'il n'est pas nécessaire de déclarer comme membres d'une classe.

Comment choisir entre ces deux possibilités ?

Dans la pratique, comme les opérateurs ont généralement besoin d'accéder aux données de l'objet, les opérateurs sont autant que possible déclarés comme fonctions membres de la classe à laquelle ils s'appliquent. C'est dans cet esprit que l'on va continuer l'écriture de la classe `chaine`.

Néanmoins, la surcharge de certains opérateurs, comme l'opérateur << pour l'affichage, conduit à des situations où l'opérateur ne peut pas être défini comme membre d'une classe.

5.7 Les opérateurs de la classe `chaine`

5.7.1 L'opérateur = pour l'affectation entre objets de type `chaine`

Il y a ici peu de nouveauté par rapport au cas traité dans les chapitres précédents. Puisque l'objet contient des données en profondeur, l'affectation doit être redéfinie, et ce de la manière suivante.

Tout d'abord, au sein de la classe le prototype de l'opérateur est :

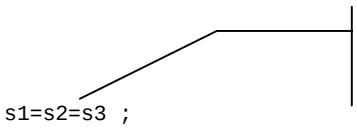
```
class chaine
{
public:
   //…
   chaine&  operator=(const chaine &);
   //…
} ;
```

Sa définition est ensuite :

```
chaine & chaine::operator =(const chaine & c)
{
   if(this!=&c)
   {
      delete _ptrChaine;
      allouеAffecte(c._ptrChaine,c._taille);
   }
   return *this;
}
```

Seule nouveauté : l'opérateur retourne un objet de type `chaine`**. Pourquoi ?**

Tout simplement pour que la ligne suivante soit compilée et exécutée avec succès :



```
s1=s2=s3 ;
```

Ce qui est interprété comme
`s1.operator=(s2.operator=(s3));`

L'objectif est de garder une certaine homogénéité avec le C++ où de telles affectations en cascade sont possibles. Il est donc nécessaire que l'opérateur = retourne l'objet courant. L'opérateur doit donc se terminer par

```
chaine & chaine::operator =(const chaine & c)
{
   //…

   return *this;  // retourne l'objet courant
}
```

De plus, puisque cet objet n'est pas local à la fonction, on peut le retourner par référence et éviter ainsi une duplication de l'objet au moment du retour.

5.7.2 « Les » opérateurs d'indexation `[]`

Nous souhaitons pouvoir conserver l'accès direct aux caractères d'une chaîne, comme s'il s'agissait d'un tableau. Il convient pour cela de redéfinir l'opérateur `[]`.

Nous souhaitons pouvoir écrire

```
chaine s1="abcd";
s1[2]='e';
```

Cette dernière ligne est interprétée par le compilateur comme l'appel suivant :

```
s1.operator[](2)='e';
```

Autrement dit, l'opérateur d'indexation, qui d'ailleurs ne peut être surchargé qu'à la condition d'être membre de la classe, doit avoir un prototype du style

```
type_retour chaine::operator[](int indice);
```

Il s'agit donc d'une fonction membre de la classe `chaine` dont le seul argument correspond à l'indice figurant entre les crochets.

Il reste à régler le problème du retour de l'opérateur.

Il faut distinguer deux cas. L'opérateur ne doit pas proposer les mêmes services pour les objets constants et les objets variables.

```
chaine svar ="modifiable ";
const chaine sconst = "immuable";

cout << svar[1] << endl ;
cout << sconst[1] << endl ;

svar[2]='c' ;

sconst[2]='c' ; // << doit être interdit par le compilateur.
```

Il doit y avoir en fait *deux* opérateurs [ ] au sein de la classe. L'un pour traiter les objets constants, l'autre les objets variables.

L'opérateur [ ] pour les objets variables.

L'opérateur d'indexation traitant les objets variables doit pouvoir être appelé à gauche d'une affectation. Dans le jargon du C++, ce doit être un `left-operand`.

```
svar[2]='c';   // équivaut à svar.operator[](2)='c';
```

Autrement dit, il est nécessaire que le retour de l'opérateur soit un retour par référence. L'opérateur d'indexation doit retourner une référence à un caractère du tableau. Pour l'opérateur d'indexation non-constant l'entête est donc

```
char & chaine::operator[](int indice){…}
```

L'opérateur [ ] pour les objets constants.

L'opérateur d'indexation traitant les objets constants ne doit pas pouvoir être appelé à gauche d'une affectation.

```
sconst[2]='c';    // << doit être interdit par le compilateur
sconst.operator[](2)='c'; // << interdit
```

autrement dit, il suffit que le retour de l'opérateur soit un retour par valeur. L'opérateur d'indexation constant doit donc retourner un caractère du tableau par valeur.

Pour l'opérateur d'indexation constant, l'entête est donc :

```
char chaine::operator[](int indice) const {…}
```

Contrôle des indices

Puisque l'accès aux caractères d'un objet `chaine` passe par l'appel d'un opérateur que l'on maîtrise, on peut ajouter un contrôle supplémentaire vérifiant la validité de l'indice. En effet, si l'indice est négatif ou supérieur à `_taille-1`, le programme doit s'arrêter et afficher un message d'erreur indiquant à l'utilisateur que le programme contient certainement un problème d'ordre algorithmique.

Finalement, les deux opérateurs sont définis de la manière suivante.

```
class chaine
{
public:
   char&    operator[](int);
   char     operator[](int) const;

   //…
};

// opérateur [] (pour objets variables)
char & chaine::operator [](int indice)
{
   if(indice<0 || indice>=_taille) // contrôle d'indice
   {
      cerr << "indice hors de la plage hors de la plage d’indices"<<endl;
      exit(2);
   }
   else  return _ptrChaine[indice];
}

// opérateur [] (pour objets constants)
char chaine::operator [](int indice) const
{
   if(indice<0 || indice>=_taille) // contrôle d'indice
   {
      cerr << "indice hors de la plage d’indices"<< endl;
      exit(2);
   }
   else  return _ptrChaine[indice];
}
```

5.7.3 L’opérateur de concaténation (opérateur +)

La concaténation de chaînes de caractères doit être rendue possible par la syntaxe

```
s3 = s1+s2;
```

ce qui est en fait interprété comme

```
s3.operator=(s1.operator+(s2));
```

dans le cas où, comme nous l’avons dit, nous choisissons de définir l’opérateur + comme membre de la classe `chaine`.

Compte-tenu de nos habitudes de programmation, lorsque l’on lit la ligne suivante

```
s3 = s1+s2;
```

on comprend que, d’une part, `s3` va contenir la concaténation des chaînes `s1` et `s2`, mais aussi que ni `s1` ni `s2` n’est modifiée par cette concaténation. En ce qui concerne l’opérateur +, on en déduit que ni l’objet courant, ni l’objet argument n’est modifié par cet opérateur.

L'ébauche de l'entête de cet opérateur est donc

```
type_retour chaine::operator+(const chaine &) const {…}
```

Quel est le retour de cet opérateur ? Quel en est le type ?

Clairement, cet opérateur doit retourner un objet de type `chaine` réalisant la concaténation de l'objet courant avec l'objet argument.

Est-ce un retour par valeur ou par référence ?

Pour que le retour soit un retour par référence, il faut que l'objet retourné ne soit pas local à l'opérateur +. Or, ici nous n'avons pas d'autre choix que de créer l'objet résultat au sein même de l'opérateur (puisque le retour n'est ni l'objet courant, ni l'objet argument). Le retour est donc ici forcément un retour par valeur.

L'entête complet de cet opérateur de concaténation est donc

```
chaine chaine::operator+(const chaine &) const
```

La déclaration et la définition de cet opérateur sont donc :

```
class chaine
{
public:
   chaine   operator+(const chaine &) const;
   //..
};

chaine chaine::operator+(const chaine & c) const
{
   chaine resultat;        //chaine résultat (initialement vide)

   resultat._taille=_taille + c._taille;
   resultat._ptrChaine = new char[resultat._taille]; // allocation de mémoire pour stocker
                                                     // la concaténation
   if(resultat._ptrChaine==NULL)
   {
      cerr << "erreur d'allocation" << endl;
      exit(1);
   }
   memmove(resultat._ptrChaine,_ptrChaine,_taille);
   memmove(resultat._ptrChaine+_taille,c._ptrChaine,c._taille);

   return resultat;
}
```

5.7.4 L'opérateur <<

On souhaite pouvoir écrire

```
cout << s1 << endl;
```

pour afficher un objet `chaine` à l'écran. La ligne précédente peut être interprétée de deux manières selon que l'opérateur est membre ou non d'une classe.

Si l'opérateur << est membre d'une classe, ce doit être un membre de la classe dont est issu l'objet `cout`. L'appel est alors interprété de la manière suivante par le compilateur

```
cout.operator<<(s1).operator<<(endl);
```

Si l'opérateur << n'est pas membre d'une classe, l'appel de l'opérateur est interprété

```
operator<<(operator<<(cout,s1),endl);
```

L'opérateur << ne peut pas être surchargé dans la classe de `cout`.

L'objet `cout` est un objet de la classe `ostream` (out stream = flot de sortie) qui est définie dans `iostream.h`.

On ne peut pas définir l'opérateur << pour afficher des objets `chaine` dans la classe `ostream` puisque l'on n'en pas le fichier source. Il est donc impossible ici de surcharger l'opérateur pour l'affichage comme membre de la classe `ostream`.. Il ne reste plus que l'autre solution. La ligne

```
cout << s1 << endl;
```

sera interprétée

```
operator<<(operator<<(cout,s1),endl);
```

Le premier opérande de cet opérateur est un objet de la classe `ostream`. Il faut considérer qu'une écriture modifie le flot `cout`. Aussi, le premier opérande est un paramètre de type entrée-sortie. Le passage par référence s'impose.

Le second opérande est un objet de la classe `chaine` qui lui ne sera pas modifié par l'affichage. Le second opérande peut donc être passé par référence constante.

L'opérateur doit retourner le flot `cout` !

Attention ! L'opérateur doit retourner le flot de type `ostream` par référence pour autoriser des insertions successives dans le flot du style

```
cout << s1 << s2 << endl;
```

Le prototype de cet opérateur est donc

```
ostream & operator<<(ostream &, const chaine & );
```

et sa définition est

```
ostream & operator<<(ostream & flot,const chaine & c)
{
   for(int i=0;i<c.taille();i++)
   {
      flot << c[i];
   }
   return flot;
}
```

5.8 La classe `chaine` au complet

Après avoir détaillé, méthode par méthode, l'écriture de cette classe, on donne ici le résultat de cette étude. On a en outre ajouté des méthodes qui ne doivent plus, à ce stade, poser de problème.

Déclaration / Définition de la classe

```
//chaine.h
#ifndef __CLASSE_CHAINE__
#define __CLASSE_CHAINE__

#include<iostream.h>
#include<string.h>
#include<process.h>
#include<memory.h>

class chaine
{
private:
   char * _ptrChaine;
   int _taille;
   void alloueAffecte(const char *,int);

public:

   chaine();
   chaine(const char *);
   chaine(const chaine &);

   ~chaine();

   chaine&  operator=(const chaine &);

   int taille() const;

   chaine   operator+(const chaine &) const;

   char&    operator[](int);
   char  operator[](int) const;

   bool  operator==(const chaine &) const;
};

   ostream & operator<<(ostream & ,const chaine & );

#endif
```

Remarque

En pratique, le fichier de définition d'une classe (ici `chaine.h`) est très utile. Il donne suffisamment d'informations pour pouvoir utiliser la classe. On y voit, entre autres, quels constructeurs sont disponibles (ici 3 constructeurs) ou encore si l'on peut réaliser des affectations avec l'opérateur =.

Même si cela n'a pas été fait ici, il est donc judicieux de bien commenter ce fichier (sans le rendre illisible) en expliquant comment fonctionne l'interface de la classe, puisque c'est tout ce dont a besoin l'utilisateur.

Définition des fonctions membres

```
// fichier chaine.cpp
#include "chaine.h"

// fonction à accès privé
void chaine::alloueAffecte(const char * p,int t)
{
   _ptrChaine = new char [_taille=t];
   if(_ptrChaine==NULL)
   {
      cerr << "erreur d'allocation ";
      exit(1);
   }
   memmove(_ptrChaine,p,t);
}

//constructeur sans argument
chaine::chaine()
{
   _ptrChaine=NULL;
   _taille=0;
}

// constructeur avec un argument de type pointeur sur char
chaine::chaine(const char * str)
{
   alloueAffecte(str,strlen(str));
}

// constructeur-copie
chaine::chaine(const chaine & c)
{
   alloueAffecte(c._ptrChaine,c._taille);
}

// destructeur
chaine::~chaine()
{
   delete _ptrChaine;
}
int chaine::taille() const
{
   return _taille;
}

// opérateur =
chaine & chaine::operator =(const chaine & c)
{
   if(this!=&c)
   {
      delete _ptrChaine;
      alloueAffecte(c._ptrChaine,c._taille);
   }
   return *this;
}
```

```
chaine chaine::operator+(const chaine & c) const
{
   chaine resultat;

   resultat._taille=_taille+c._taille;
   resultat._ptrChaine = new char[resultat._taille];

   if(resultat._ptrChaine==NULL)
   {
      cerr << "erreur d'allocation" << endl;
      exit(1);
   }

   memmove(resultat._ptrChaine,_ptrChaine,_taille);
   memmove(resultat._ptrChaine+_taille,c._ptrChaine,c._taille);

   return resultat;
}

// opérateur [] : accès à un caractère d'une chaîne
char & chaine::operator [](int indice)
{
   if(indice<0 || indice>=_taille) // contrôle d'indice
   {
      cerr << "indice hors de la plage"<<endl;
      exit(2);
   }
   else  return _ptrChaine[indice];
}

// opérateur [] (pour objets constants)
char chaine::operator [](int indice) const
{
   if(indice<0 || indice>=_taille) // contrôle d'indice
   {
      cerr << "indice hors de la plage"<< endl;
      exit(2);
   }
   else  return _ptrChaine[indice];
}

// opérateur == : retourne true si chaîne courante = chaîne argument
bool chaine::operator==(const chaine & c) const
{
   if(_taille==c._taille)
   {
      if(_taille==0) return true;
      else  return !strncmp(_ptrChaine,c._ptrChaine,_taille);
   }
   else  return false;
}
```

```
// opérateur << pour cout << chaine("bonjour");
ostream & operator<<(ostream & flot,const chaine & c)
{
   for(int i=0;i<c.taille();i++)
   {
      flot << c[i];
   }
   return flot;
}
```

6 AMITIE : FONCTIONS AMIES, CLASSES AMIES.

6.1 Introduction

La programmation objet impose l'encapsulation de données, ne rendant celles-ci accessibles qu'aux membres de la classe. En C++, l'**unité de protection est la classe**, ce qui signifie qu'une fonction membre d'une classe peut accéder à la partie privée de tout objet de cette classe.

```
class A
{
   private:
      int data;
   public:
      int EstEgal(A x)
      {
         return (data==x.data);
      }
};

void main()
{
   A obj1,obj2;
   //…
   if(obj1.EstEgal(obj2))
   {
      //…
   }
}
```

Dans cet exemple, la fonction `EstEgal()` appelée sur l'objet `obj1` a accès également à la donnée privée `data` de `obj2`.

En revanche, l'encapsulation interdit à une fonction membre d'une classe d'accéder à des membres privés d'une autre classe. Cette contrainte peut-être gênante dans certaines circonstances, par exemple lorsque l'on surcharge l'opérateur << pour l'affichage d'un objet à l'écran.

La notion d'amitié va permettre à certaines fonctions non membres d'une classe d'avoir néanmoins accès à la partie privée d'une classe.

Une fonction `f()` amie d'une classe `A` a accès à la partie privée de tout objet de la classe `A`.

Comment se déclare l'amitié ?

Lors de la déclaration/définition d'une classe, il est possible d'y déclarer qu'une ou plusieurs fonctions, extérieures à la classe, sont *amies* de la classe.

Remarque : puisque l'amitié est déclarée dans la classe, une fonction quelconque ne peut pas « se proclamer » amie d'une classe. C'est la classe qui attribue ce droit.

6.2 Fonction amie d'une classe.

Il y a plusieurs situations d'amitié. La première consiste à rendre une fonction extérieure à toute classe amie d'une ou de plusieurs classes. Chaque lien d'amitié est déclaré dans une classe.

Dans l'exemple suivant, la fonction `f()` acceptant un argument de type `A` est déclarée amie de la classe `A`.

```
class A;

      int f(A);   // prototype

class A
{
   friend int f(A);  // la fonction f est déclarée amie de la classe A

   private:
      int data;
   public:
      //…
};

int f(A x)
{
   return x.data; // OK puisque f est amie de A, elle a accès à x.data
}
```

Exemple 39. Fonction amie d'une classe

6.3 Fonction membre d'une classe amie d'une autre classe.

Cette deuxième situation présente une fonction d'une classe `B` amie d'une classe `A`.

Soient deux classes `A` et `B`. Dans la classe `B`, une fonction membre a l'entête suivant

```
int B::f(char c,A x)
{
   //…
}
```

On souhaite que cette fonction `f()` ait accès aux membres privés des objets de la classe `A`. Il faut pour cela qu'elle soit amie de la classe `A`. La déclaration d'amitié dans la classe `A` est la suivante :

```
class A; // déclaration anticipée

class B
{
   private:
      //…
   public:
      int f(char, A);
};

class A
{
   friend int B::f(char,A);   // la fonction B::f() est déclarée amie de la classe A
   //…
};
```

Ici, le compilateur a besoin de savoir que A est une classe, d'où la déclaration anticipée.

Exemple 40. Fonction d'une classe amie d'une autre classe

6.4 Fonction amie de plusieurs classes.

Soient deux classes `A` et `B` et une fonction

```
int f(A x,B y)
{
   //…
}
```

Il est souvent pratique qu'une telle fonction `f()` ait accès aux membres privés de `A` et `B` sans restriction. Il suffit de déclarer `f()` amie de chacune des classes `A` et `B`. Il y a donc deux déclarations d'amitié.

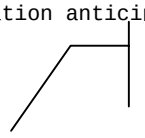
```
class B;

class A
{
   friend int f(A,B);   // la fonction f est déclarée amie de la classe A
   //…
};

class B
{
   friend int f(A,B);   // la fonction f est aussi déclarée amie de la classe B
   //…
};

int f(A x ,B y)
{
   // les champs privés des objets x et y sont ici accessibles.
}
```

Exemple 41. Fonction amie de deux classes.

6.5 Classe amie d'une autre classe.

Pour déclarer que toutes les fonctions membres d'une classe `B` sont amies d'une classe `A`, on écrit

```
class B; // déclaration anticipée de la classe B

class A
{
   friend class B;
};
```



Toutes les fonctions de la classe B sont amies de la classe A.

6.6 Exemple : l'opérateur << de la classe `chaine`

Dans la classe `chaine` décrite au chapitre précédent, comme l'opérateur << était extérieur à la classe `chaine`, il lui était impossible d'accéder à la partie privée de la classe `chaine`.

On aurait pu modifier la classe de la manière suivante.

Déclaration / Définition de la classe

```
//chaine.h
#ifndef __CLASSE_CHAINE__
#define __CLASSE_CHAINE__

#include<iostream.h>
#include<string.h>
#include<process.h>
#include<memory.h>

class chaine
{
private:
   char * _ptrChaine;
   int _taille;
   void alloueAffecte(const char *,int);

public:
   chaine();
   chaine(const char *);
   chaine(const chaine &);
   ~chaine();
   int taille() const;
   chaine&  operator=(const chaine &);
   chaine   operator+(const chaine &) const;
   char&    operator[](int);
   char  operator[](int) const;
   bool  operator==(const chaine &) const;

   friend ostream & operator<<(ostream &, const chaine &);
};
#endif
```

L'opérateur << est déclaré ami de la classe chaine.

```
#include "chaine.h"

// …

// opérateur << pour cout << chaine("bonjour");
// fonction amie de la classe chaine
ostream & operator<<(ostream & flot,const chaine & c)
{
   for(int i=0;i < c._taille;i++)
   {
      flot << c._ptrChaine[i];
   }
   return flot;
}
```

_taille et _ptrChaine ne seraient pas accessibles sans la déclaration d'amitié.

7 LES CHANGEMENTS DE TYPE.

7.1 Introduction

En langage C, il existe des situations où des variables d'un certain type sont attendues par des fonctions ou des opérateurs et que d'autres types leur sont passés.

```
int i;
float f=3.2 ;

i=f;  // conversion implicite float->int
```

Dans l'exemple précédent, il y a une conversion d'un type `float` vers le type `int`. Ce changement de type est d'ailleurs dit dégradant dans la mesure où une partie de l'information est perdue en route : la variable `i` se voit affecter la partie entière de la variable `f`.

Dans le cas précédent, on dit qu'il y a eu une *conversion de type implicite* puisque le programmeur ne spécifie rien et le compilateur met néanmoins une règle de conversion en oeuvre.

Une syntaxe particulière permet de demander explicitement au compilateur une conversion de type lorsqu'une telle conversion a un sens.

```
int i;
float f=3.2 ;

i=(int)f;   // conversion explicite float -> int
```

Nous cherchons ici à explorer ce qu'amènent les classes dans les changements de type. Nous étudions deux sortes de conversions :

. les conversions entre un type classe et un type de base
. les conversions d'un type classe vers un autre type classe

7.2 Conversion d'un type de base vers un type classe par un constructeur

Reprenons la classe `chaine` élaborée au chapitre 5.

```
class chaine
{
private:
   char * _ptrChaine;
   int _taille;
   void alloueAffecte(const char *,int);
public:
   chaine();
   chaine(const char *);
   chaine(const chaine &);
   ~chaine();
   chaine&  operator=(const chaine &);
   //…
};
```

Partant de cette classe, les lignes suivantes sont compilées et exécutées sans problème :

```
chaine s1;
s1="bonjour";
```

Pourtant il apparaît clairement dans la définition de la classe que l'opérateur = attend un objet de type `chaine` à droite.

Que se passe-t-il ?

Effectivement, un objet de type `chaine` est attendu à droite de l'opérateur =. Puisqu'on passe un argument de type `char *`, le compilateur cherche s'il est possible de convertir une chaîne de caractères en un objet de type `chaine`. Or, il est effectivement possible de construire un objet de type `chaine` à partir d'une chaîne de caractères du langage C.

Chronologiquement, sur la ligne

```
s1="bonjour";
```

il se déroule la même chose qu'en écrivant

```
s1.operator=(chaine("bonjour"));
```

c'est-à-dire

. construction d'un *objet temporaire* de type `chaine` avec l'argument `"bonjour"` de type `char *`
. appel de `s1.operator=()` avec cet objet temporaire comme argument

La règle suivante s'applique en C++ : lorsqu'un objet de type classe `CLS` (ici `chaine`) est attendu, et qu'une variable `autre` de type `autreType` est fournie, le compilateur cherche si parmi les constructeurs de la classe `CLS` il en exite un qui accepte un seul argument de type `autreType`. Si un tel constructeur exite, un objet temporaire est construit avec ce constructeur et est utilisé à la place de la variable `autre`.

Autrement dit :

Les constructeurs à un argument définissent une règle de conversion.

Par exemple, le constructeur

```
chaine::chaine(const char *)
```

définit la conversion d'un type `const char *` (ou `char *`) vers le type `chaine`.

Remarque : ce rôle des constructeurs doit être bien compris car en pratique beaucoup de conversions de type sont réalisées *tacitement* sans que le programmeur en soit tenu informé.

On peut donner un autre exemple utilisant la classe `chaine` et illustrant ces conversions implicites. Pour les mêmes raisons que précédemment, la ligne suivante est compilée sans aucun problème.

```
s1=s1+"ajout";
```

Elle produit le même comportement que si l'on avait écrit : `s1=s1.operator+(chaine("ajout"));`

En effet, là où l'opérateur + attend un objet de type `chaine`, on lui passe une variable de type `char *`. Le compilateur cherche donc s'il est possible de convertir la chaîne `"ajout"` en un objet de type `chaine`.

7.3 Conversion d'un type classe vers un type de base

On peut également définir des règles permettant de convertir un objet d'un type classe en un type de base. Il s'agit en fait de définir le traitement réalisé par un *opérateur de cast.*

Considérons l'exemple suivant

```
chaine s1="abc";
int i;
i=(int)s1; // conversion explicite du type chaine en type int
```

Sur cette ligne, le compilateur C++ cherche si une fonction membre appelée `operator int` a été définie dans la classe `chaine`.

Pour que les lignes précédentes soient compilées avec succès, il est donc nécessaire que l'ajout de l'opérateur `int` soit effectué à la classe `chaine`.

```
class chaine
{
private:
   char * _ptrChaine;
   int _taille;
   void alloueAffecte(const char *,int);
public:
   //…
   operator int() {return _taille;}
};
```

Ici, nous avons défini la conversion `chaine -> int` de sorte de convertir un objet `chaine` en sa taille.

Note : il n'est pas très évident de comprendre toujours précisément quelles conversions le compilateur réalise implicitement. Il peut même y avoir plusieurs conversions en cascade. Aussi, il est recommandé de ne pas abuser de ces opérateurs de conversion.

7.4 Exemple

On va illustrer toutes ces règles de conversions sur une classe `complexe` pour laquelle nous avons défini l'opérateur + pour la somme de deux nombres complexes.

```
//complexe.h
#ifndef __COMPLEXE__
#define __COMPLEXE__
#include<iostream.h>

class complexe
{
private:
   double re;  // partie réelle
   double im;  // partie imaginaire
public:
   complexe(double r=0,double i=0)
   {
      cout << "complexe(double=0,double=0) \t objet : "<<this << endl;
      re=r;
      im=i;
   }

   ~complexe()
   {
      cout << "~complexe() \t\t\t objet : "<<this << endl;
   }
```

```
   complexe operator+(const complexe & c) const
   {
      cout << "complexe::operator+() \t obj:"<<this<<" + obj:" <<&c<<endl;
      return complexe(re+c.re,im+c.im);
   }

   operator double()
   {
      cout << "complexe::operator double() \t obj:"<<this<<endl;
      return re;
   }

   friend ostream & operator<<(ostream & flot,const complexe & c);
};

ostream & operator<<(ostream & flot, const complexe & c)
{
   flot << c.re;
   if(c.im)
   {
      flot << "+"<<c.im<<"i";
   }
   return flot;
}
#endif
```

Le programme d'utilisation est le suivant :

```
#include"complexe.h"

void main()
{
   complexe c1,c2(1,3);

   c1=2.4;
   cout << c1 << endl;

   c2=c1+c2;
   cout << c2 << endl;

   c1=c1+(complexe)3.5;
   cout << c1 << endl;

   c1=(double)c1+3.5;
   cout << c1 << endl;

   double d;
   d=c1;
   cout << d << endl;

   // c1=c1+3.5;  // << error c2666 : 2 overloads have similar conversions
}
```

Résultats

```
complexe(double=0,double=0)  objet : 0x0066FDDC           1
complexe(double=0,double=0)  objet : 0x0066FDCC           2
complexe(double=0,double=0)  objet : 0x0066FDB4           3
~complexe()                  objet : 0x0066FDB4           4
2.4                                                       5
complexe::operator+()  obj :0x0066FDDC + obj :0x0066FDCC  6
complexe(double=0,double=0)  objet : 0x0066FDA4           7
~complexe()                  objet : 0x0066FDA4           8
3.4+3i                                                    9
complexe(double=0,double=0)  objet : 0x0066FD94           10
complexe::operator+()  obj :0x0066FDDC + obj :0x0066FD94  11
complexe(double=0,double=0)  objet : 0x0066FD84           12
~complexe()                  objet : 0x0066FD84           13
~complexe()                  objet : 0x0066FD94           14
5.9                                                       15
complexe::operator double()  obj :0x0066FDDC              16
complexe(double=0,double=0)  objet : 0x0066FD74           17
~complexe()                  objet : 0x0066FD74           18
9.4                                                       19
complexe::operator double()  obj :0x0066FDDC              20
9.4                                                       21
~complexe()                  objet : 0x0066FDCC           22
~complexe()                  objet : 0x0066FDDC           23
```

Les lignes de la sortie écran ont été numérotées : il y a 23 affichages en tout.

1) construction de `c1` (objet d'adresse 0x0066FDDC)
2) construction de `c2` (objet d'adresse 0x0066FDCC)

```
c1=2.4 ;        //c1.operator=(complexe(2.4));
```

3) construction d'un objet temporaire, à partir de l'argument 2.4, qui est passé à l'opérateur = . L'adresse de cet objet temporaire est 0x0066FDB4.

Note : puisque l'opérateur = n'a pas été explicitement défini, c'est l'opérateur = par défaut qui est utilisé. Ce dernier fait une copie membre-à-membre qui convient parfaitement puisque les objets de la classe `complexe` n'ont pas de données en profondeur.

4) Cet objet est détruit après l'exécution de l'opérateur =
5) Affichage du contenu de `c1` grâce à la surdéfinition de l'opérateur <<

```
c2=c1+c2 ;      //c2.operator=(c1.operator+(c2));
```

6) L'opérateur + de la classe `complexe` est appelé.
7) Un objet temporaire d'adresse 0x0066FDA4 est construit pour retourner le résultat de l'opérateur car c'est un retour par valeur. C'est d'ailleurs cet objet temporaire qui est utilisé comme argument de l'opérateur =.
8) Destruction de l'objet retourné par l'opérateur +
9) Affichage de `c2`

```
c1=c1+(complexe)3.5 ;       //c1.operator=(c1.operator+(complexe(3.5)));
```

10) On utilise ici la syntaxe d'une conversion de type explicite. La valeur 3.5 doit être convertie en un objet `complexe`. Il y a donc construction d'un objet de la classe `complexe` d'adresse 0x0066FD94.
11) Appel de l'opérateur + de la classe `complexe`. On remarque que l'argument de l'opérateur est bien l'objet d'adresse 0x0066FD94
12) L'opérateur + retourne un objet par valeur. Il y a construction d'un objet d'adresse 0x0066FD84.
13) Après l'affectation à l'objet `c1`, cet objet temporaire est détruit.
14) L'objet de la classe `complexe` construit pour convertir 3.5 en `complexe` est détruit.
15) Affichage de `c1`

```
c1=(double)c1+3.5 ;     //c1.operator=(complexe((double)c1+3.5));
```

16) Là encore, il ya une conversion explicite. On souhaite convertir `c1` en type `double`, ce qui fait appel à l'opérateur `double` de la classe `complexe` sur l'objet `c1` (dont l'adresse est 0x0066FDDC)

Note : par conséquent, c'est l'opérateur + correspondant au type `double` qui est utilisé ici, et non celui du type `complexe`.

17) Puisque `(double)c1+3.5` est de type `double`, il doit y a voir une conversion implicite `double-> complexe` avant l'affectation. Il y a donc construction d'un objet temporaire d'adresse 0x0066FD74.
18) Cet objet est passé à l'opérateur = puis détruit.
19) Affichage de `c1`

```
d=c1;    // d =(double) c1;
```

20) Il y a conversion implicite du type `complexe` vers le type `double`. Cette conversion implicite utilise l'opérateur `double` de la classe `complexe` sur l'objet `c1`.
21) Affichage de `c1`
22) Destruction de `c2` (car fin de `main()`)
23) Destruction de `c1` (car fin de `main()`)

7.5 Conclusion

Ces conversions de type constituent un sujet sensible. L'utilisation de ces moyens de conversions peut conduire à des situations où le compilateur ne sait pas quelle conversion mettre en œuvre. C'est le cas pour la ligne suivante :

```
c1=c1+3.5;  // c1=c1+(complexe)3.5 ?  ou c1=(complexe)((double)c1+3.5) ?
```

Le compilateur ne sait pas si l'on souhaite que 3.5 soit converti en `complexe` ou bien que `c1` soit converti en `double`.

Le compilateur indique alors le message d'erreur :

```
error c2666 : 2 overloads have similar conversions
```

Notons que cela ne conduit d'ailleurs pas au même résultat !

8 UTILISATION DE CLASSES EXISTANTES PAR COMPOSITION

8.1 Introduction

On présente dans ce chapitre une première manière de réutiliser des classes existantes pour concevoir de nouvelles classes.

La composition : on parle de composition lorsque les données membres d'un objet sont elles-mêmes des instances d'une classe.

Par exemple dans cette ébauche de la classe `cercle`, le centre du cercle est un objet de la classe `point`.

```
class cercle
{
private:
   point centre;
   int rayon ;
public:
   //…
};
```

8.2 Ecriture d'une nouvelle classe `personne`

Créer une classe au sein de laquelle les champs sont eux-mêmes des objets ne pose pas de problème majeur. La seule nouveauté concerne *la liste d'initialisation* pour les constructeurs de la classe composée. Cette présentation s'appuie simplement sur une réécriture de la classe `personne` décrite dans la première partie de ce document.

Dans le chapitre 4, nous avions conçu une classe `personne` de la manière suivante

```
class personne
{
   private:
      char * ptrNom ;
      int age ;
   public :
      //…
};
```

Après avoir développé une classe de chaînes de caractères au chapitre 5, il paraît plus naturel de concevoir la classe `personne` de la manière suivante :

```
class personne
{
   private:
      chaine _nom ;
      chaine _prenom ;
      int _age ;
   public:
      //…
};
```

Chaque objet de la classe personne sera *composé* de deux objets de type `chaine` pour les noms et prénoms et d'un champ `_age` de type `entier`.

Note : on a également ajouté un champ `_prenom` qui ne figurait pas dans la version du chapitre 4.

8.2.1 Définition de la classe `personne`

```
// personne.h

#ifndef __PERSONNE__
#define __PERSONNE__

#include<iostream.h>
#include "chaine.h"

class personne
{
   chaine _nom;
   chaine _prenom;
   int _age;

public:

   personne();
   personne(const char *,const char *,int);
   personne(const chaine &,const chaine &,int);
   personne(const personne &);

   ~personne();

   personne & operator=(const personne &);


   chaine getNom() const;
   chaine getPrenom() const;
   int getAge() const;


   void setNom(const chaine &);
   void setPrenom(const chaine &);
   void setAge(int);


   friend ostream & operator<<(ostream &,const personne &);

};

#endif
```

```
#include "personne.h"

personne::personne():_nom("neant"),_prenom("neant"),_age(0)
{
}

personne::personne(const char * strN,const char * strP,int a):_nom(strN),
                                                              _prenom(strP),_age(a)
{
}

personne::personne(const chaine & N,const chaine & P,int A):_nom(N),_prenom(P),_age(A)
{
}

personne::personne(const personne & p):_nom(p._nom),_prenom(p._prenom),_age(p._age)
{
}

personne::~personne()
{
}

personne & personne::operator=(const personne & p)
{
   if(this!=&p)
   {
      _nom=p._nom;
      _prenom=p._prenom;
      _age=p._age;
   }
   return *this;
}

chaine personne::getNom() const
{
   return _nom;
}

chaine personne::getPrenom() const
{
   return _prenom;
}

int personne::getAge() const
{
   return _age;
}

void personne::setNom(const chaine & N)
{
   _nom=N;
}

void personne::setPrenom(const chaine & P)
{
   _prenom=P;
}

void personne::setAge(int A)
{
   _age=A;
}

ostream & operator<<(ostream & flot,const personne & p)
{
   flot << p._prenom << ' '<< p._nom << " " << p._age << " ans";
   return flot;
}
```

liste d'initialisation

8.2.2 Les constructeurs de la classe personne et la liste d'initialisation

La nouveauté se trouve au niveau de la syntaxe des constructeurs de cette classe. En effet, construire un objet de la classe `personne` requiert également la construction des deux objets de type `chaine` qui le composent. Il faut donc spécifier quelque part comment ces objets membres doivent être initialisés. C'est le rôle de la liste d'initialisation.

Liste d'initialisation : il s'agit de la liste qui suit la liste des arguments du constructeur.

```
personne::personne():_nom("neant"),_prenom("neant"),_age(0)
{
}
```

On spécifie après les « : » comment les différents membres doivent être initialisés. En ce qui concerne les objets membres, ça détermine quels constructeurs appeler. Dans l'exemple précdent, on indique que pour le constructeur sans argument de la classe `personne`, les objets membres `_nom` et `_prenom` seront construits en appelant le constructeur

```
chaine::chaine(const char *)
```

avec la chaîne `"neant"` comme argument.

Le champ `_age` quant à lui sera initialisé avec la valeur `0`.

Pour le constructeur

```
personne::personne(const chaine & N,const chaine & P,int A):_nom(N),_prenom(P),_age(A)
{
}
```

on voit que, dans la liste d'initialisation, c'est le constructeur-copie de la classe `chaine` qui est utilisé pour initialiser les membres `_nom` et `_prenom`.

Enfin, le constructeur suivant est tout simplement le constructeur copie de la classe `personne` :

```
personne::personne(const personne & p):_nom(p._nom),_prenom(p._prenom),_age(p._age)
{
}
```

Remarque : la liste d'initialisation ne s'applique qu'aux constructeurs d'une classe et pas aux autres méthodes de la classe.

Quelle différence y-a-t-il entre ces deux définitions du même constructeur ?

Version 1

```
personne::personne():_nom("neant"),_prenom("neant"),_age(0)
{
}
```

Version 2

```
personne::personne()
{
   _nom="neant";
   _prenom="neant";
   _age=0;
}
```

Bien que ces deux constructeurs conduisent au même résultat, ces deux versions ne réalisent pas le même traitement aussi efficacement.

Dans la première version (la plus rapide), les objets membres `_nom` et `_prenom` sont directement construits avec les bonnes valeurs en utilisant directement le constructeur adéquat.

Dans la seconde version (la moins efficace), le constructeur ne spécifie aucune information pour l'initialisation des objets membres. Les deux objets membres sont donc construits tout d'abord avec le constructeur sans argument de la classe `chaine`. Lorsque la première ligne du constructeur s'exécute, les objets membres existent déjà avec les valeurs prévues par le constructeur sans argument de la classe `chaine`. Les lignes suivantes utilisent donc l'opérateur = de la classe `chaine`.

```
_nom="neant";
_prenom="neant";
```

En synthèse :

version 1 : appel du constructeur `chaine::chaine(const char *)` pour les deux membres `_nom` et `_prenom`
version 2 : appel du constructeur `chaine::chaine()` pour les deux membres `_nom` et `_prenom` puis appel de l'opérateur `chaine & chaine::operator=(const chaine &)`

8.2.3 L'opérateur = de la classe personne

Cet opérateur d'affectation a été explicitement défini pour remplacer l'opérateur = par défaut.

```
personne & personne::operator=(const personne & p)
{
   if(this!=&p)
   {
      _nom=p._nom;
      _prenom=p._prenom;
      _age=p._age;
   }
   return *this;
}
```

Equivaut à

```
_nom.operator=(p._nom)
```

Il convient simplement de noter que cet opérateur utilise l'opérateur = de la classe `chaine` pour les deux membres `_nom` et `_prenom`.

8.2.4 Le constructeur-copie et l'opérateur = par défaut.

Il n'était pas indispensable ici de donner une définition explicite de l'opérateur = pour la classe `personne`. Dans ce cas précis, l'opérateur = par défaut suffisait.

En effet, le traitement par défaut de l'opérateur = d'une classe est de réaliser une copie membre-à-membre. Or, en présence d'objets membres, cette copie membre-à-membre utilise par défaut l'opérateur = de la classe à laquelle appartiennent ces objets membres. Ici, la copie des membres `_nom` et `_prenom` utiliserait donc par défaut l'opérateur = de la classe `chaine`.

Cett remarque vaut également pour le constructeur-copie. Le constructeur-copie par défaut d'une classe, dans le cas où la classe dispose d'objets membres, utilise pour les membres de type classe le constructeur-copie de la classe.

Ainsi, dans le cadre de cette classe `personne`, le constructeur-copie par défaut donnerait un résultat tout à fait satisfaisant dans la mesure où il réaliserait la construction des membres `_nom` et `_prenom` avec le constructeur-copie de la classe `chaine`.

Conseil : dans le doute, il est préférable de redéfinir ces deux méthodes explicitement plutôt que de ne pas bien comprendre ce que fait le traitement par défaut.

8.2.5 L'opérateur << pour l'affichage des objets de type personne

```
class personne
{
   //…
   friend ostream & operator<<(ostream &,const personne &);
   //…
};

ostream & operator<<(ostream & flot,const personne & p)
{
   flot << p._prenom << ' '<< p._nom << " " << p._age << " ans";
   return flot;
}
```

Rappelons qu'il s'agit nécessairement d'un opérateur non membre de la classe `personne`. Il est alors plus pratique de déclarer cet opérateur ami de la classe `personne` pour qu'il ait accès aux champs privés sans aucune restriction.

```
   flot << p._prenom << ' '<< p._nom << " " << p._age << " ans";
```

Il convient de remarquer que cet opérateur utilise la surcharge de l'opérateur << appliqué aux objets de type `chaine`. En effet, grâce à notre classe `chaine`, on peut écrire

```
   flot  << p._nom;
```

sachant que `flot` est une référence à l'objet `cout`.

9 UTILISATION DE CLASSES EXISTANTES PAR DERIVATION

9.1 Introduction

La dérivation : la seconde façon de réutiliser une classe consiste à la réutiliser en bloc en lui ajoutant des données/fonctions. On parle *d'héritage* ou de *dérivation*.

La classe dérivée *hérite* des fonctionnalités de la classe de base.

Exemple : dans le cadre de la représentation des pixels d'un moniteur vidéo, on souhaite créer une classe de points avec un attribut supplémentaire : la couleur. On peut réutiliser la classe `point` comme classe de base et ajouter un champ de couleur. On dit que la classe `pointC` dérive de la classe `point`.

```
class pointC : public point
{
private:
   int couleur;      // ajout d'un champ couleur
public:
   //…
};
```

La classe `pointC` dérive (ou hérite) de la classe `point`

9.2 classe `point`

Rappelons tout d'abord quelle classe `point` l'on utilise.

```
// point.h
#ifndef __POINT__
#define __POINT__
#include<iostream.h>

class point
{
private:
   float x,y;
public:
   point(float=0,float=0);
   point operator+(const point &) const;
   friend ostream & operator<<(ostream & flot, const point & p);
};
#endif
```

```
// point.cpp
#include"point.h"
point::point(float abs,float ord)
{
   x=abs;   y=ord;
}
point point::operator+(const point & p) const
{
   point local;
   local.x = x + p.x;
   local.y = y + p.y;
   return local;
}
ostream & operator<<(ostream & flot, const point & p)
{
   flot << '(' << p.x << ','<< p.y <<')';
   return flot;
}
```

9.3 Dérivation de la classe `point` en `pointC`

Dans cette première approche de la dérivation , on essaie de réutiliser la classe `point` en l'état.

Syntaxe : pour dériver la classe `point` en classe `pointC` (point coloré) on écrit simplement

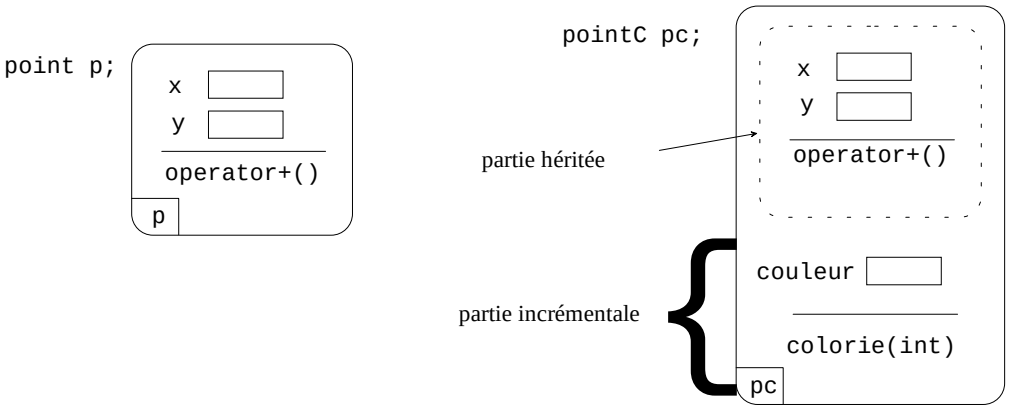
```
class pointC : public point
{
private:
   int couleur;
public:
   pointC(float =0,float=0,int=0);
   pointC(const point &, int=0);
   pointC(const pointC &);

   void colorie(int c);
};
```

La classe `pointC` dérive (ou hérite) de la classe `point`

Remarque : on se contente pour l'instant de traiter le cas de la dérivation publique (attribut `public`) qui est le mode de dérivation le plus couramment utilisé.

Dès lors que l'on définit la classe `pointC` comme dérivant de la classe `point`, il faut avoir à l'esprit que tout objet de la classe `pointC` contiendra nécessairement les données et les méthodes de la classe `point`. Autrement dit, une partie d'un objet de type `pointC` est hérité de la classe `point`.

Schématiquement, on peut voir les objets `point` et `pointC` comme ayant les structures suivantes :



Un objet de la classe `point` contient deux champs de données privés pour les coordonnées ainsi que des méthodes. Puisque la classe `pointC` est définie par dérivation (publique) de la classe `point`, tout objet de la classe `pointC` contiendra également ces fonctionnalités. C'est ce qui est décrit en pointillé sur la partie droite du schéma précédent. C'est *la partie héritée* d'un objet de type `pointC`.

Tout ce que l'on définit de nouveau dans la classe `pointC` vient s'ajouter à cette base. La partie ajoutée est appelée *partie incrémentale.*

9.4 Accès aux champs de l'objet père

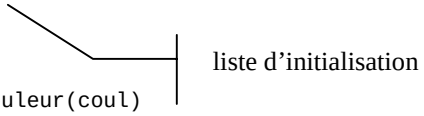
Un objet de type `pointC` contient des champs `x` et `y` hérités de la classe `point`. Néanmoins, il n'est pas possible pour l'objet fils d'accéder aux champs privés de l'objet père. Dans la classe `pointC`, les méthodes n'ont pas accès directement aux champs privés `x` et `y`.

Pourquoi ? Il suffirait de dériver une classe pour contourner l'encapsulation de données.

9.5 Définition de la classe `pointC`

```
// pointcol.h
#ifndef __POINTCOL__
#define __POINTCOL__

#include<iostream.h>
#include "point.h"

class pointC:public point
{
private:
   int couleur;
public:
   pointC(float =0,float=0,int=0);
   pointC(const point &, int=0);
   pointC(const pointC &);

   void colorie(int c);
};
#endif
```

```
// pointcol.cpp
#include "pointcol.h"

pointC::pointC(float abs,float ord,int coul):point(abs,ord),couleur(coul)
{
}

pointC::pointC(const point & p,int coul):point(p),couleur(coul)
{
}

pointC::pointC(const pointC & pc):point(pc),couleur(pc.couleur)
{
}

void pointC::colorie(int c)
{
   couleur=c;
}
```



9.6 Les constructeurs de la classe `pointC`

Le premier constructeur est défini de la manière suivante :

```
pointC::pointC(float abs,float ord,int coul):point(abs,ord),couleur(coul)
{


}
```

La partie héritée est construite en appelant le constructeur `point::point(float,float)`

La construction d'un objet de type `pointC` requiert avant tout la construction de la partie héritée qui, elle, est de type `point`. Aussi, dans le constructeur de la classe `pointC`, on doit spécifier comment la partie héritée est construite. Ceci est fait dans la liste d'initialisation. On peut également spécifier dans cette liste d'initialisation comment le champ `couleur` doit être initialisé.

Le second constructeur est défini de la manière suivante :

```
pointC::pointC(const point & p,int coul):point(p),couleur(coul)
{


}
```

La partie héritée est construite en appelant le constructeur `point::point(const point &)`

La liste d'initialisation indique que la partie héritée sera construite en utilisant le constructeur-copie de la classe `point`.

9.7 Compatibilité dérivée -> base : le constructeur-copie de la classe `pointC`

Le constructeur-copie de la classe `pointC` est défini comme suit :

```
pointC::pointC(const pointC & pc):point(pc),couleur(pc.couleur)
{
}
```

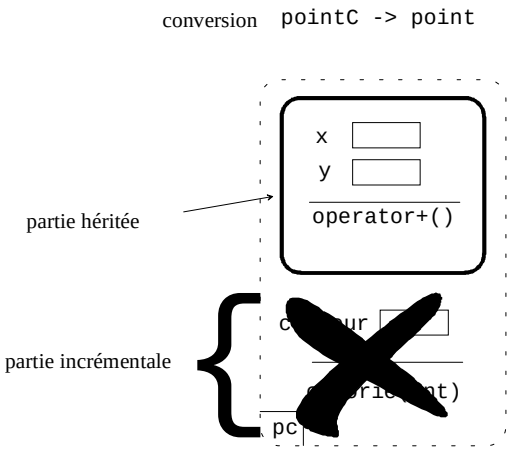
Il convient de relever une particularité : dans la liste d'initialisation, on spécifie que l'objet `pc` est passé au constructeur de la partie héritée. Or, aucun constructeur de la classe `point` n'est prévu pour recevoir un argument de type `pointC`.

Conversion classe dérivée -> classe de base

Il y a en fait une règle de conversion que le compilateur peut toujours mettre en œuvre : la conversion type classe dérivée vers classe de base (ou fils vers père). Ici, il est possible de convertir un objet de type `pointC` en un objet de type `point`.

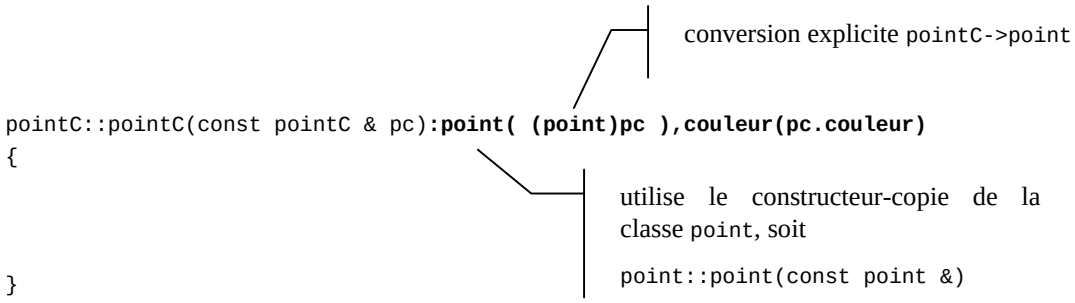
Quel est le sens de cette conversion de type ?

Il s'agit d'une conversion dégradante. Dans notre exemple, on peut considérer qu'un « point coloré » est avant tout un « point », si l'on oublie la couleur (c-à-d la partie incrémentale).



De manière générale, un objet de la classe dérivée peut être converti en un objet de la classe de base. Seule la partie héritée est préservée par cette conversion de type. La partie incrémentale est perdue lors de la conversion.

Revenons au constructeur de la classe `pointC` : l'écriture suivante est donc équivalente à celle donnée dans la définition de la classe (avec une conversion explicite)



9.8 Modification de la classe de base `point` et complément de la classe `pointC`

Modification de l'attribut des données membres de la classe `point`

Il peut être gênant que les méthodes de la classe `pointC` n'aient pas accès aux coordonnées `x` et `y` de la partie héritée. Ceci est dû à l'attribut `private` dans la classe `point`.

L'attribut `protected` offre les services suivants :

. un membre protégé est inaccessible de l'extérieur

(En l'absence de dérivation, un membre protégé a les mêmes propriétés qu'un membre privé).

. par contre, en dérivation publique, les membres protégés de la classe de base sont *accessibles par les méthodes de la classe dérivée.*

Ici, il est préférable de redéfinir les champs `x` et `y` comme membres protégés de la classe `point`.

```
// point.h
#ifndef __POINT__
#define __POINT__
#include<iostream.h>

class point                    modification de l'attribut
{
protected:
   float x,y;
public:
   point(float=0,float=0);

   point operator+(const point &) const;

   friend ostream & operator<<(ostream &,
                        const point & p);
};

#endif
```

```
// point.cpp

#include"point.h"
#include <iostream.h>

point::point(float abs,float ord)
{
   x=abs;
   y=ord;
}

point point::operator+(const point & p) const
{
   point local;

   local.x = x + p.x;
   local.y = y + p.y;

   return local;
}

ostream & operator<<(ostream & flot,
                     |    const point & p)
{
   flot << '(' << p.x << ','<< p.y <<')';
   return flot;
}
```

Opérateur + de la classe `pointC`

On ajoute un opérateur + dans la classe dérivée `pointC` : cet opérateur doit réaliser la somme des coordonnées et le maximum des couleurs.

Puisque les données de la classe de base ont désormais l'attribut `protected`, elles deviennent visibles dans la classe dérivée, ce qui rend possible l'écriture de cet opérateur.

On définit également un opérateur << pour afficher les points colorés.

```
#ifndef __POINTCOL__
#define __POINTCOL__
#include "point.h"
class pointC:public point
{
protected:
   int couleur;
public:
   pointC(float =0,float=0,int=0);
   pointC(const point &, int=0);
   pointC(const pointC &);
   void colorie(int c);
   pointC operator+(const pointC &) const;
   friend ostream & operator<<(ostream &, const pointC &);
};
#endif
```

```
#include "pointcol.h"
pointC::pointC(float abs,float ord,int coul):point(abs,ord),couleur(coul)
{
}
pointC::pointC(const point & p,int coul):point(p),couleur(coul)
{
}
pointC::pointC(const pointC & pc):point(pc),couleur(pc.couleur)
{
}
void pointC::colorie(int c)
{
   couleur=c;
}

pointC pointC::operator +(const pointC & pc) const
{
   pointC local;
   local.x = x + pc.x;
   local.y = y + pc.y;
   local.colorie(pc.couleur>couleur ? pc.couleur : couleur);
   return local;
}

ostream & operator<<(ostream & flot, const pointC & pc)
{
   flot << (point)pc << '['<<pc.couleur<<']';
   return flot;
}
```

La définition de l'opérateur + ne présente aucune originalité hormis l'utilisation de l'opérateur ternaire

```
op1 ? op2 : op3
```

pour réaliser le maximum de deux couleurs. Cet opérateur retourne `op2` si `op1` vaut `true` ou retourne `op3` sinon.

```
pointC pointC::operator +(const pointC & pc) const
{
   pointC local;
   local.x = x + pc.x;
   local.y = y + pc.y;
   local.colorie(pc.couleur>couleur ? pc.couleur : couleur);
   return local;
}
```

On peut également noter qu'au sein de la définition de l'opérateur

```
ostream & operator<<(ostream & flot, const pointC & pc)
```

il y a une conversion de type explicite du type `pointC` vers le type `point`.

```
ostream & operator<<(ostream & flot, const pointC & pc)
{
   flot << (point)pc << '['<<pc.couleur<<']';
   return flot;
}
```

conversion de type `pointC->point`

Ceci permet de réutiliser l'opérateur << qui a déjà fait l'objet d'une surcharge pour les objets de type `point`. Autrement dit, il est de cette façon possible d'afficher la partie héritée du point coloré, c'est-à-dire ses coordonnées, en utilisant l'opérateur << déjà surchargé pour la classe `point`. Il ne reste plus ensuite qu'à afficher la couleur.

9.9 Ordre d'appel des constructeurs de la classe de base et de la classe dérivée

Dans la contruction d'un objet de type `pointC`, il y a également la construction d'un objet de type `point`.

Nous redéfinissons deux classes `point` et `pointC` (simplifiées) pour observer l'ordre d'appel des constructeurs.

```
class point
{
protected:
   float x,y;
public:
   point(float abs=0,float ord=0)
   {
      cout << "point::point(float,float) obj: " << this << endl;
      x=abs;
      y=ord;
   }
   ~point()
   {
      cout << "point::~point() obj:" << this << endl;
   }
};
```

```
class pointC:public point
{
protected:
   int couleur;

public:
   pointC(float abs,float ord,int coul):point(abs,ord),couleur(coul)
   {
      cout<<"pointC::pointC(float,float,int) obj: "<< this << endl;
   }
   pointC(const point & p,int coul):point(p),couleur(coul)
   {
      cout<<"pointC::pointC(const point &,int) obj: "<< this << endl;
   }
   pointC(const pointC & pc):point(pc),couleur(pc.couleur)
   {
      cout<<"pointC::pointC(const pointC &) obj: "<< this << endl;
   }
   ~pointC()
   {
      cout << "pointC::~pointC() obj:" << this << endl;
   }
   void colorie(int c) { couleur=c;}
};
```

```
void main()
{
   pointC p1(10,20,15);

   pointC p2(p1);

}
```

Résultat

```
point::point(float,float)  obj :0x0066FDE0
pointC::pointC(float,float,int)  obj :0x0066FDE0
pointC::pointC(const pointC &)  obj :0x0066FDD4
pointC::~pointC()  obj :0x0066FDD4
point::~point()  obj :0x0066FDD4
pointC::~pointC()  obj :0x0066FDE0
point::~point()  obj :0x0066FDE0
```

Que remarquer ?

Lors de la construction d'un objet de la classe dérivée il y a chronologiquement

. appel du constructeur de la classe de base
. appel du constructeur de la classe dérivée

Inversement, lors de la destruction d'un objet de la classe dérivée il y a chronologiquement

. appel du destructeur de la classe dérivée
. appel du destructeur de la classe de base

Remarque

Pour l'objet `p2`, on ne voit pas d'appel au constructeur de la classe `point`. Pourquoi ? En fait, le constructeur-copie de la classe `pointC` utilise le constructeur-copie de la classe `point`. Or, ce dernier n'est pas explicitement défini. Il y a donc utilisation du constructeur-copie « par défaut » de la classe `point` qui, lui, ne produit pas de sortie écran.

9.10 Que retenir sur la dérivation publique ?

Considérons une classe `BASE` et une classe `DERIVEE` dérivant publiquement de la classe `BASE`.

```
class DERIVEE : public BASE
{

};
```

1) Conversion classe dérivée vers classe de base

La conversion du type `DERIVEE` vers le type `BASE` est toujours possible, qu'il s'agisse d'une conversion explicite avec la syntaxe (`BASE`) ou implicite.

Ex :

```
BASE b;
DERIVEE d;

b=d;        // conversion implicite
b=(BASE)d;  // conversion explicite
```

2) Conversion pointeur sur classe dérivée vers pointeur sur classe de base

Dans le même esprit, une conversion d'un pointeur sur objet de type `DERIVEE` vers pointeur sur objet de type `BASE` est toujours possible.

Ex :

```
BASE * ptrB;
DERIVEE *ptrD;

DERIVEE D;     // objet de la classe DERIVEE

ptrD = &D;        // aucune conversion de type
ptrB = ptrD;      // conversion implicite DERIVEE * vers BASE *
ptrB = &D;        // conversion implicite DERIVEE * vers BASE *

ptrB = (BASE*) ptrD; // conversion explicite par la syntaxe (BASE*)
```

Si l'on revient à la classe `pointC` qui dérive de la classe `point`, on peut donner l'exemple suivant.

Ex :

```
pointC pc(10,25,12);    // objet pc coordonnées (10,25) couleur [12]

point * ptrP ;       // pointeur ptrP de type point *
ptrP=&pc;            // conversion implicite pointC* -> point*

cout << (*ptrP);     // (*ptrP) est un objet de type point
                     // l'affichage est donc (10,25)
```

3) Surdéfinition possible, dans la classe dérivée, des méthodes de la classe de base

Dans le cas de la classe `pointC`, on surcharge une méthode `operator+()` déjà présente dans la classe de base. En réalité, les entêtes complets de ces deux méthodes sont différents.

Pour l'opérateur + de la classe `point`, l'entête est

```
point point::operator+(const point &) const
```

Pour l'opérateur + de la classe `pointC`, l'entête est

```
pointC pointC::operator+(const pointC &) const
```

Remarque : on se rappelle d'ailleurs que la signature d'une méthode est constituée, non seulement de l'identificateur de la méthode, mais aussi de la liste des types des arguments et de la classe propriétaire.

4) Constructeur-copie par défaut de la classe dérivée

Comme dans toute classe, une classe créée par dérivation d'une classe de base possède un constructeur-copie par défaut qui réalise une copie membre-à-membre.

En pratique, la partie héritée de la classe est considérée comme un membre à part entière. Plus précisément, **le constructeur-copie par défaut de la classe dérivée fait appel au constructeur-copie de la classe de base pour la construction de la partie héritée.**

```
class point
{
protected:
   float x,y;
public:
   point(float abs=0,
         float ord=0):x(abs),y(ord)
   {
   }
   point(const point & p)
   {
   cout << "point::point(const point &)";
   cout << " obj: " ;
   cout << this << endl;
      x=p.x;
      y=p.y;
   }
};
```

```
class pointC:public point
{
protected:
   int couleur;
public:
   pointC(float abs,
          float ord,
      int coul):point(abs,ord),couleur(coul)
   {
   }

   void colorie(int c) { couleur=c;}
};
```

Sur l'exemple suivant, on voit effectivement que le constructeur-copie de la classe `point` est appelé lors de la construction de l'objet `p2` par le constructeur-copie par défaut de la classe `pointC`.

```
void main()
{

   pointC p1(10,25,12);

   pointC p2(p1);

}
```

Résultat

```
point::point(const point & ) obj 0x0066FDE0
```

Conclusion : par conséquent, si la partie incrémentale de la classe dérivée ne possède pas de données en profondeur, il n'est pas indispensable de définir explicitement le constructeur-copie de la classe dérivée. Le traitement par défaut convient**.** Dans le doute, ne pas hésiter à redéfinir le constructeur-copie de la classe dérivée.

5) Opérateur = par défaut de la classe dérivée

De même que dans toute classe, une classe créée par dérivation possède un opérateur = par défaut qui réalise une affectation membre-à-membre. Dans le cadre de la dérivation, la partie héritée constitue un membre à part entière.

L'opérateur = par défaut de la classe dérivée utilise l'opérateur = de la classe de base pour réaliser l'affectation.

C'est ce que l'on voit sur l'exemple suivant.

```
class point
{
protected:
   float x,y;
public:
point(floatabs=0,float ord=0):x(abs),y(ord)
   {
   }
point & operator=(const point & p)
{
   cout << "point::op=()"<< endl;
   if(this!=&p)
   {
      x=p.x;
      y=p.y;
   }
   return *this;
}
};
```

```
class pointC:public point
{
protected:
  int couleur;
public:

pointC(float abs=0,float ord=0,
int coul=0):point(abs,ord),couleur(coul)
  {
  }
friend ostream & operator<<(ostream &, const
pointC &);
};

ostream & operator<<(ostream & flot,
                     const pointC & p)
{
  flot << '(' << p.x;
  flot << ',' << p.y;
  flot << ")["<<p.couleur<<']';
  return flot;
}
```

```
void main()
{
   pointC p1(10,20,56),p2;

   p2=p1;

   cout << p2;
}
```

Résultat

```
point : :op=()
(10,20)[56]
```

9.11 Formes canoniques

On a vu que certains traitements sont faits par défaut. Il est difficile (mis à part à l'usage) de mémoriser quels mécanismes sont mis en œuvre dans certaines situations. Il est donc plus sage de définir *systématiquement* quelques méthodes d'usage très fréquent.

On dit qu'une classe est *sous forme canonique* si elle contient au minimum un constructeur sans argument, un constructeur-copie explicitement défini (en remplacement du constructeur-copie par défaut), le destructeur et un opérateur = défini (en remplacement de l'opérateur = par défaut).

Pourquoi est-il recommandé de définir systématiquement ces 4 méthodes ?

Le constructeur sans argument : il est indispensable à la création d'un tableau d'objets alloué dynamiquement. Par exemple, dans l'expression

```
ptr = new BASE[10];
```

il est impossible de passer une information au constructeur des 10 objets de type `BASE` du tableau. Le constructeur sans argument est alors indispensable

Le constructeur-copie : Le constructeur-copie par défaut ne convient pas lorsqu'il y a des données en profondeur.

L'opérateur = : Idem. L'opérateur = par défaut ne convient pas lorsqu'il y a des données en profondeur.

Le destructeur : Dans le cas de classes avec données en profondeur, cette méthode restitue la mémoire.

Ce qui suit est un exemple type de classe de base sous forme canonique et de classe dérivée également sous forme canonique. La classe de base contient des données en profondeur. On considére que **`CLS`** est une classe qui contient des données en profondeur.

```
class Base
{
protected:
   CLS * ptr;
   int taille;
public:
   Base(int t=0);
   Base(const Base & B);
   ~Base();
   Base & operator=(const Base & B);
};

Base::Base(int t){
   ptr=new CLS[taille=t];
}
Base::Base(const Base &B){
   ptr=new CLS[taille=B.taille];
   for(int i=0;i<taille;i++)
   {
      ptr[i]=B.ptr[i];
   }
}
Base::~Base(){
   delete [] ptr;
}
Base & Base::operator=(const Base & B){
   if(this!=&B)
   {
      delete [] ptr;
      ptr=new CLS[taille=B.taille];
      for(int i=0;i<taille;i++)
      {
         ptr[i]=B.ptr[i];
      }
   }
   return *this;
}
```

```
class Derivee:public Base
{
protected:
   int ajout;
public:
   Derivee(int t=0,int aj=0);
   Derivee(const Derivee & D);
   ~Derivee();
   Derivee& operator=(const Derivee &);
};

Derivee::Derivee(int t,int aj):Base(t){
   ajout=aj;
}
Derivee::Derivee(const Derivee & D):Base(D){
   // initialisation de la partie
   // incrémentale
   ajout = D.ajout;
}
Derivee::~Derivee(){
// destruction partie incrémentale

   // Note :la partie héritée est détruite
   // par ~BASE()
}
Derivee& Derivee::operator=(const Derivee& D)
{
   if(this!=&D)
   {
      // affectation partie héritée
      Base * p1 = this;
      const Base * p2 = & D;
      (*p1)=(*p2);
      // affectation partie incrémentale
      ajout=D.ajout;
   }
   return *this;
}
```

A propos de la classe de base…

Puisque la classe `CLS` contient des données en profondeur, il est important de restituer la mémoire allouée par la classe `Base` avec la syntaxe `delete [] ptr;`

A propos de la classe dérivée…

Il convient de remarquer la syntaxe du constructeur-copie

```
Derivee::Derivee(const Derivee & D):Base(D)
{
   // etc.
}
```

construction de la partie héritée en utilisant le constructeur-copie de la classe de base

La construction de la partie héritée est faite en appelant le constructeur-copie de la classe de base. On utilise implicitement la conversion de type `Derivee-> Base`.

Il convient également de noter, dans l'opérateur =, comment est effectuée la copie de la partie héritée. On utilise ici la conversion de type `Derivee * -> Base *`.

```
Derivee& Derivee::operator=(const Derivee& D)
{
   if(this!=&D)
   {
      Base * p1 = this;
      const Base * p2 = & D;
      (*p1)=(*p2);      // utilise l'opérateur = de la classe Base

      // etc.
   }
   return *this;
}
```

Il ne reste plus ensuite qu'à copier la partie incrémentale.

9.12 Accès aux données et aux méthodes dans le cadre de l'héritage

Il s'agit ici de donner des précisions sur l'accès aux données et fonctions membres de la classe de base et de la classe dérivée dans le cadre d'une dérivation publique.

```
class Base
{
protected:
   int dataBase;
public:
Base()
{
   dataBase=8;
}

void show()
{
 cout << "Base::show() :" << dataBase;
 cout << " obj :" << this << endl;
}

void showB()
{
 cout << "Base::showB() :" << dataBase;
 cout <<" obj :" << this <<endl;
}
};
```

```
class Derivee:public Base
{
protected:
  float dataDerivee;
public:
Derivee()
{
  dataDerivee=12;
}

void show()
{
 cout << "Derivee::show() :" << dataBase;
 cout<< " "<< dataDerivee <<" obj :";
 cout << this <<endl;
}

void showD()
{
 cout << "Derivee::showD() :" <<dataBase<< " ";
 cout << dataDerivee<<" obj :" << this <<endl;
}
};
```

On s'appuie une fois de plus sur un exemple. On définit une classe appelée `Base` et une classe dérivée de cette classe. Chacune de ces classes possède des fonctions membres d'affichage des données membres.

```
void main()
{
   Base B;
   Derivee D;

   B.showB(); // méthode de Base

   B.show();   // méthode de Base

   D.showD(); // méthode de Derivee
   D.showB(); // méthode de Base

   D.show();   // méthode de Derivee
   D.Base::show();   // méthode de Base
}
```

Résultats

```
Base::showB() :8 obj :0x0066FDF4
Base::show() :8 obj :0x0066FDF4
Derivee::showD() :8 12 obj :0x0066FDEC
Base::showB() :8 obj :0x0066FDEC
Derivee::show() :8 12 obj :0x0066FDEC
Base::show() :8 obj :0x0066FDEC
```

Remarque :

On peut placer dans la classe dérivée une méthode ayant même identificateur qu'une méthode de la classe de base. On voit qu'il y a alors un phénomène de *masquage*. Par exemple, lorsque l'on appelle la méthode `show()` sur un objet de la classe `Derivee`, la méthode `show()` de la classe de base n'est plus directement visible. Pour accéder à la méthode `show()` de la classe de base sur un objet de la classe dérivée, on doit ajouter le nom de la classe propriétaire au moment de l'appel, soit :

```
D.Base::show();
```

On a donc équivalence entre

```
B.show();   et    B.Base::show();
D.show();   et    D.Derivee::show();
```

9.13 Les différentes formes de dérivation

On présente ici les différentes formes de dérivation en C++. L'exposé est ici fait sous forme de tableaux de synthèse des droits d'accès selon les différentes formes de dérivation.

Rappel : les attributs possibles des membres (donnée ou fonction) d'une classe :

. membre public : visible par les membres de la classe et par l'utilisateur de la classe (« l'extérieur »)

. membre privé : visible par les fonctions membres et amies de la classe mais pas par un utilisateur de la classe (« l'extérieur ») ni par les fonctions membres d'une classe dérivée.

. membre protégé : visible par les fonctions membres et amies de la classe mais pas par l'utilisateur de la classe (« l'extérieur ») . A la différence des membres privés, les membres protégés sont visibles par les fonctions membres et amies des classes dérivées.

Trois formes de dérivation sont possibles : la dérivation publique, privée ou protégée.

La dérivation publique

```
class Derivee : public Base
{
};
```

Statut dans la classe de base	Accès aux fonctions membre et amies de la classe dérivée	Accès à un utilisateur de la classe dérivée	Nouveau statut dans la classe dérivée, en cas de nouvelle dérivation
Public	Oui	Oui	Public
Protégé	Oui	Non	Protégé
Privé	Non	Non	Privé

La dérivation privée

```
class Derivee : private Base
{
};
```

Dans ce mode de dérivation, les membres publics de la classe de base ne sont plus visibles par un utilisateur de la classe dérivée.

La dérivation protégée (à partir de la version 3 du C++)

```
class Derivee : protected Base
{
};
```

Dans ce mode de dérivation, les membres publics de la classe de base seront considérés comme protégés lors des dérivations ultérieures.

Synthèse **Accès FMA :** accès aux fonctions membres et amies de la classe
Nouveau statut : statut qu'aura ce membre dans une éventuelle classe dérivée

Classe de base			**Dérivée publique**		**Dérivée protégée**		**Dérivée privée**	
Statut initial	Accès FMA	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur
Public	Oui	Oui	Public	Oui	Protégé	Non	Privé	Non
Protégé	Oui	Non	Protégé	Non	Protégé	Non	Privé	Non
Privé	Oui	Non	Privé	Non	Privé	Non	Privé	Non

9.14 Le typage statique (ou ligature statique)

Reprenons l'exemple de dérivation suivant :

```
class Base
{
protected:
   int dataBase;
public:
   Base();
   void show();
   void showB();
};
```

```
class Derivee:public Base
{
protected:
   float dataDerivee;
public:
   Derivee();
   void show();
   void showD();
};
```

Nous savons qu'il y a les compatibilités suivantes : `Derivee->Base` et `Derivee*->Base*`

On peut donc écrire :

```
void main()
{
   Derivee D;
   Base * ptrB;
   Derivee * ptrD=&D;
   ptrB = ptrD;   // conversion Derivee*->Base *

   ptrB->show();  // appel de la fonction Base::show()
}
```

L'appel `ptrB->show()` correspond à l'appel de la méthode `show()` de la classe de base. Or, `ptrB` contient pourtant l'adresse d'un objet de type `Derivee`.

Par défaut, le langage C++ réalise une ligature statique. C'est-à-dire que le compilateur se fie au type du pointeur (ici `ptrB` est de type pointeur sur `Base`) pour déterminer la méthode à appeler.

On formule la même remarque lorsque l'on manipule des références :

```
void main()
{
   Derivee D;
   Base & refB=D;
   refB.show();   // appel de la fonction Base::show()
}
```

Le typage statique pose un problème puisque, même si l'on possède l'adresse d'un objet de type `Derivee` dans le pointeur `ptrB`, on ne peut atteindre que les méthodes de la classe de base (méthodes héritées) via ce pointeur.

Le typage dynamique offre une solution à ce problème.

9.15 Fonctions virtuelles : typage dynamique

Lorsque l'on déclare des méthodes avec l'attribut **`virtual`**, on indique au compilateur qu'une ligature dynamique doit être mise en œuvre pour ces fonctions.

Par exemple, on crée une ligature dynamique de la manière suivante pour la méthode `show()` :

```
class Base
{
protected:
   int dataBase;
public:
   Base();
   virtual void show();
   void showB();
};
```

```
class Derivee:public Base
{
protected:
   float dataDerivee;
public:
   Derivee();
   void show();
   void showD();
};
```

```
void main()
{
   Base B;
   Derivee D;

   Base * ptrB=&B;

   ptrB->show();

   ptrB=&D;

   ptrB->show();

}
```

Résultats

```
Base::show() :8 obj :0x0066FDF0
Derivee::show() :8 12 obj :0x0066FDE4
```

Quelle différence observe-t-on ?

Lors du premier appel

```
Base * ptrB=&B;
ptrB->show();
```

le pointeur `ptrB` contient l'adresse d'un objet de type `Base`. C'est donc la méthode `show()` de la classe de base qui est appelée.

Lors du second appel

```
ptrB=&D;
ptrB->show();
```

le pointeur `ptrB` contient l'adresse d'un objet de type `Derivee`. C'est donc la méthode `show()` de la classe dérivée qui est appelée.

Dans le cadre du typage dynamique, obtenu par la définition de méthodes virtuelles, le choix de la méthode appelée est effectué au moment de l'exécution et non pas pendant la phase de compilation.

9.16 Application du typage dynamique

Nous définissons une hiérarchie de classes. Les classes `point`, `cercle` et `rectangle` dérivent toutes d'une même classe de base appelée `forme`.

La définition de la classe `forme` est la suivante :

```
// classe abstraite
class forme
{
public:
   virtual void show()=0; // fonction virtuelle pure

   virtual ~forme(){};     // destructeur virtuel
};
```

Une fonction membre virtuelle dont la déclaration est suivie de `=0` est appelée *fonction virtuelle pure*. La fonction `show()` de cette classe est une fonction virtuelle pure.

Une classe contenant une fonction virtuelle pure est appelée *classe abstraite*.

Qu'est-ce qu'une classe abstraite ?

Une classe abstraite *ne peut pas créer d'objets*. Elle sert uniquement de modèle pour la dérivation. En effet, lorsqu'une classe de base contient une fonction virtuelle pure, celle-ci doit nécessairement être définie dans les classes dérivées. Cette déclaration impose donc une contrainte aux classes dérivées. Notons également qu'une fonction virtuelle pure n'a pas de définition dans la classe abstraite (pas de code).

Dans notre exemple, on définit une classe abstraite appelée `forme` servant de modèle à la dérivation de plusieurs classes représentant des formes géométriques en vue d'être affichées à l'écran. Puisque chaque forme devra s'afficher à l'écran, la fonction d'affichage est une fonction virtuelle pure dans la classe abstraite. Cette fonction doit donc être explicitement définie dans les classes dérivées.

Les définitions des autres classes sont les suivantes

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

   void show()
   {
      cout<< "point " <<'('<<x<<','<<y<<')';
   }
};
```

```
class cercle:public forme
{
protected:
   point centre;
   int rayon;
public:
   cercle(point c=point(10,10),int r=5):centre(c),rayon(r)
   {
   }

   void show()
   {
      cout << "cercle "<<'[';
      centre.show();
      cout <<rayon<<']';
   }
};
```

```
class rectangle:public forme
{
protected:
   point A;
   point B;
public:
   rectangle(point a=point(20,20),point b=point(40,40)):A(a),B(b)
   {
   }

   void show()
   {
      cout << "rectangle " << '[';
      A.show();
      B.show();
      cout << ']';
   }
};
```

Le programme principal utilisant ces différentes classes est le suivant

```
void main()
{

   forme * tabForme[5];

   tabForme[0]=new rectangle;
   tabForme[1]=new point(30,45);
   tabForme[2]=new cercle;
   tabForme[3]=new cercle(point(30,35),30);
   tabForme[4]=new cercle;

   for(int i=0;i<5;i++)
   {
      tabForme[i]->show();
      cout << endl;
   }

   for(i=0;i<5;i++)
   {
      delete tabForme[i];
   }
}
```

résultats

```
rectangle [point (20,20)point (40,40)]
point (30,45)
cercle [point (10,10)5]
cercle [point (30,35)30]
cercle [point (10,10)5]
```

On déclare un tableau de pointeurs sur des objets de type forme en écrivant

```
forme * tabForme[5];
```

Notons que la déclaration suivante n'aurait d'ailleurs pas été possible puisqu'il est impossible d'instancier des objets de type forme.

```
forme tabForme[5];   //<< impossible
```

Puisqu'il y a possibilité de convertir n'importe quel pointeur sur un objet de classe dérivée en un pointeur sur un objet de type classe de base, les différentes conversions `point *->forme *`, `cercle * -> forme *` et `rectangle * -> forme *` sont toutes possibles.

Les écritures suivantes sont donc légales.

```
tabForme[0]=new rectangle;    // rectangle * -> forme *
tabForme[1]=new point(30,45); // point * ->  forme *
tabForme[2]=new cercle;       // cercle * -> forme *
tabForme[3]=new cercle(point(30,35),30);
tabForme[4]=new cercle;
```

Grâce à la fonction virtuelle pure `show()` (c'est-à-dire avant tout virtuelle) dans la classe de base, il y une ligature dynamique sur l'appel de cette méthode.

Par conséquent, dans la boucle suivante, la méthode `show()` appelée à chaque itération dépend du type d'objet pointé par le pointeur `tabForme[i]`. Par conséquent, il y a appel de la méthode `show()` appropriée à chaque objet pointé, d'où l'affichage.

```
for(int i=0;i<5;i++)
   {
      tabForme[i]->show();
      cout << endl;
   }
```

La dernière boucle permet de restituer la mémoire allouée par chacun des objets.

```
for(i=0;i<5;i++)
   {
      delete tabForme[i];
   }
```

Destructeur virtuel

On doit noter que le destructeur de la classe `forme` est également une fonction virtuelle. Pourquoi ?

Lorsque l'on détruit les objets alloués dynamiquement, si le destructeur possède un lien statique, seul le destructeur de la classe `forme` est appelé. Pour que le destructeur de la classe à laquelle appartient réellement l'objet dynamique soit appelé, il convient de déclarer virtuel le destructeur de la classe de base. C'est ce qui est fait ici.

Cela est important si les classes dérivées possèdent des données en profondeur dans leur partie incrémentale. Dans l'exemple précédent, il n'y a aucune incidence.

Intérêt du typage dynamique

L'avantage du typage dynamique est de pouvoir traiter de la même manière (appel de méthodes ayant le même nom) des objets de types différents. Dans l'exemple précédent, le tableau de pointeurs contient en fait les adresses d'objets de types différents (`point`, `cercle` et `rectangle`). Cette caractéristique est appelée *polymorphisme.*

10 FONCTIONS GENERIQUES ET CLASSES GENERIQUES. PATRONS DE FONCTIONS ET PATRONS DE CLASSES

10.1 Introduction

Considérons l'exemple ultra classique d'une fonction de permutation des valeurs de deux variables de type `int`.

```
#include <iostream.h>
         void permute(int & a,int & b);
void main()
{
   int x=2,y=3;
   permute(x,y);
   cout << x << y << endl;
}

void permute(int & a,int & b)
{
   int c=a;
   a=b;
   b=c;
}
```

Nous voulons désormais, dans la même application, permuter deux valeurs de type `float`. Grâce à la surdéfinition de fonctions, on peut compléter le programme de la manière suivante

```
#include <iostream.h>
   void permute(int & a,int & b);
   void permute(float & a,float & b);
void main()
{
   int x=2,y=3;
   float X=2.3,Y=3.6;
   permute(x,y);
   permute(X,Y);
}
void permute(int & a,int & b){
   int c=a;
   a=b;
   b=c;
}
void permute(float & a,float & b){
   float c=a;
   a=b;
   b=c;
}
```

En quoi les deux fonctions diffèrent-elles ? La seule différence entre ces deux fonctions concerne les types utilisés. En effet, l'algorithme (si l'on peut parler d'algorithme) est rigoureusement identique. Il en serait de même pour toute fonction de permutation de deux variables de même type.

Si l'on pouvait paramétrer les types utilisés, on n'aurait qu'une seule définition à donner. C'est ce que permet la notion de `template` (patron).

10.2 Patron de fonctions : un exemple

Le C++ permet la définition de patrons de fonctions. Un patron est un modèle dans lequel un des paramètres représente un type, et non une variable. On parle également de fonction générique. On souhaite par exemple réaliser un patron de fonctions réalisant la somme de variables de même type.

Pour voir où apparaît le paramètre de type, on peut commencer par écrire deux fonctions réalisant la somme pour les types `int` et `double`.

On obtient pour ces deux types :

```
int somme(int a,int b)
{
   return a+b ;
}
```

```
double somme(double a, double b)
{
   return a+b ;
}
```

On voit alors clairement où apparaît le paramètre de type. Le patron de fonctions s'écrit de la manière suivante :

```
template <class T> T somme(T a,T b)
{
   return a+b;
}
```

La syntaxe `template <class T>` indique que `T` est un paramètre de type (et non une variable).

Un programme d'utilisation de ce patron est le suivant :

```
#include <iostream.h>

   template <class T> T somme(T a,T b);

void main()
{
   int u=2,v=5;
   double X=2.4,Y=3.7;

   cout << somme(u,v)<<endl;
   cout << somme(X,Y)<<endl;
}

// patron de fonctions
template <class T> T somme(T a,T b)
{
   return a+b;
}
```

Un patron de fonctions ne conduit pas directement à du code. Par exemple, s'il n'y avait pas d'appel de la fonction `somme()`, aucun code ne serait généré (contrairement à une fonction du C). Le code des fonctions créées à partir du patron n'est généré que s'il y a un appel de la fonction pour un type donné.

Dans l'exemple précédent, le compilateur crée deux fonctions à partir du patron. La première fonction est créée pour le paramètre de type `T=int` et la seconde pour le paramètre de type `T=double`. Le code des fonctions est généré selon le besoin.

10.3 Les limites de ce patron

Avec le patron défini dans la section précédente, la ligne suivante n'est pas compilée

```
int u=2,v=5;
double X=2.4,Y=3.7;

cout << somme(u,Y)<<endl;  //<< impossible
```

En raison de la définition du patron, qui spécifie que les deux arguments doivent être rigoureusement de même type, il n'est pas possible d'instancier une fonction avec deux arguments de types différents. Si l'on souhaitait que l'appel précédent puisse être pris en charge par le patron, il faudrait définir le patron avec un autre paramètre de type, par exemple de la manière suivante

```
template <class T,class U> T somme(T a,U b)
{
   return a+b;
}
```

Remarque : le type de la valeur retournée est alors systématiquement le même que celui du premier argument. Autrement dit, les deux appels suivants ne conduisent pas au même résultat :

```
int u=2;
double X=2.4;

cout << somme(u,X)<<endl;  // retourne 4
cout << somme(X,u)<<endl;  // retourne 4.4
```

Un patron de fonctions peut aussi s'appliquer à des objets. Néanmoins, dans l'exemple de patron donné ici, il est nécessaire que tout le code du patron puisse s'exécuter. En particulier, il est nécessaire ici que l'opérateur + soit surchargé pour le type `T`.

Par exemple, si la classe `point` est définie et que l'opérateur + est défini pour les objets de type `point`, on pourra écrire

```
point p1(10,23),p2(25,40);
cout << somme(p1,p2)<<endl;
```

10.4 Les paramètres expression

Dans un patron de fonctions, on peut aussi faire apparaître des arguments typés normalement. Ces paramètres sont parfois appelés paramètres expression. Sur le patron de fonctions `somme()`, rien ne nous interdit de fixer le type du second argument :

```
template <class T,class U> T somme(T a, int b)
{
   return a+b;
}
```

10.5 Spécialisation d'un patron de fonctions

Un patron de fonctions définit une famille de fonctions ayant toutes le même algorithme mais pour des types différents. Il reste néanmoins possible de définir explicitement le code d'une fonction qui devrait normalement être prise en charge par le patron.

Dans l'exemple suivant, nous réalisons un patron de fonctions calculant le minimum de deux variables de même type. La fonction `min()` obtenue à partir du patron a du sens pour tous les types élémentaires sauf pour le type

`char*`. Nous avons donc défini une spécialisation pour le type `char *`. Cette spécialisation remplace le code que génèrerait le patron.

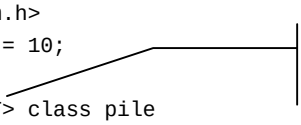
```
#include<iostream.h>
#include<string.h>

//patron de fonctions
template <class T> T min(T a,T b)
{
   return a<b ? a : b;
}

//spécialisation du patron pour le type char *
char * min(char * s1,char * s2)
{
   if(strcmp(s1,s2)<0)  return s1;
   else     return s2;
}

void main()
{
   int a=1,b=3;
   char chaine1[]="coucou",chaine2[]="salut";
   cout << min(a,b); //générée à partir du patron
   cout << min(chaine1,chaine2);  //spécialisation
}
```

10.6 Patron de classes

De même que l'on peut paramétrer certains types dans une fonction, on peut également donner des paramètres de type dans la définition d'un patron de classes. La présentation va une fois de plus reposer sur un exemple. On propose un patron de classes de piles.

Note : la syntaxe de définition d'un patron de classes n'est pas très intuitive. Il est donc important d'avoir un exemple sous les yeux lorsque vous ferez vos premiers essais

```
#include<iostream.h>
const defaultDim = 10;

template <class T> class pile
{
   protected :
      T* ptr;
      unsigned nbElem;
      unsigned dim;
   public :
      pile<T>(int taille = defaultDim);
      pile<T>(const pile<T> &);
      ~pile<T>();
      pile<T> & operator=(const pile<T> &);
      pile<T> & operator<(const T &);
      pile<T> & operator>(T &);

      friend ostream & operator<<(ostream &, const pile<T> &);
};
```



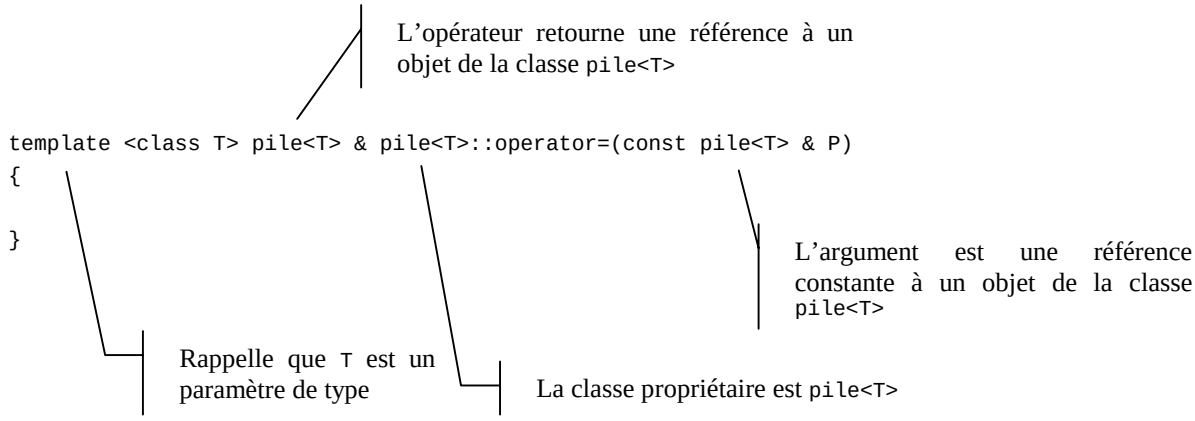
patron de classes où `T` est un paramètre de type.

```
template <class T> pile<T>::pile(int taille)
{
         ptr=new T[dim=taille];
         nbElem=0;
}

template <class T> pile<T>::~pile()
{
         delete [] ptr;
}

template <class T> pile<T>::pile(const pile<T> & P)
{
    ptr=new T[dim=P.dim];
    nbElem=P.nbElem;
    for(unsigned i=0;i<nbElem;i++) ptr[i]=P.ptr[i];

}

template <class T> pile<T> & pile<T>::operator=(const pile<T> & P)
{
   if(!(this==&P))
   {
      delete [] ptr;
      ptr=new T[dim=P.dim];
      nbElem=P.nbElem;
      for(unsigned i=0;i<nbElem;i++) ptr[i]=P.ptr[i];
   }
   return *this;
}

template <class T> pile<T> & pile<T>::operator<(const T & Source)
{
   if(nbElem<dim) ptr[nbElem++]=Source;
        return *this;
}

template <class T> pile<T> & pile<T>::operator>(T & Destination)
{
   if(nbElem>0) Destination=ptr[--nbElem];
   return *this;
}

template <class T> ostream & operator<<(ostream& flot,const pile<T>&P)
{
for(unsigned i=0; i<P.nbElem; i++) flot << P.ptr[i] << " ";
return flot;
}

void main()
{
   pile<int> p1(3);     // p1 est bun objet de la classe pile<int>
   pile<double> p2(4);  // p2 est un objet de la classe pile<double>

   p1<1<7<13;
   pile<pile<int> > p3(7); // p3 est une pile d'éléments de type pile d'entiers.
   p3<p1<p1;               // on peut donc empiler des piles d'entiers sur p3.
   cout << p3;
}
```

Au moment de le définition du patron de classes, `template <class T>` indique que `T` est un paramètre de type .

Une classe obtenue à partir du patron pour le type `X` est désignée par la syntaxe `pile<X>`. Par exemple, une classe de piles d'entiers sera désignée par l'expression `pile<int>`. Dans le programme d'utilisation du patron, l'objet `p1` est un objet de la classe `pile<int>` et `p2` un objet de la classe `pile<double>`.

Observons la définition de l'opérateur = de ce patron.



Remarque : on peut également spécialiser un patron ou faire apparaître des types par défaut.

10.7 Utilisation de patrons dans la cadre de l'héritage

Il est facile de combiner la notion de patron de classes avec celle d'héritage.

Classe ordinaire dérivée d'une classe patron (une instance particulière d'un patron)

La bibliothèque STL[5] (standard template library) fournit différents patrons de classes, dont un patron de listes doublement chaînées.

Une classe `list<X>` est une classe de listes chaînées pouvant stocker des éléments de type `X`.

L'exemple suivant montre comment l'on peut réutiliser une liste d'entiers pour faire une classe de piles d'entiers.
En effet, si l'on se contente d'ajouter des éléments et de les retirer à la même extrémité de la liste, une liste chaînée a un comportement de pile (gestion LIFO).

```
#include<list>       // définition du patron de listes
#include<iostream.h>

using namespace std;    // pour l'utilisation de la bibliothèque STL sous VisualC++ 6

class pile:private list<int>
{
public:
   void push(int);
   void pop(int &);
   void affiche();
};
```

La classe `pile` dérive d'une instance du patron `list<T>`

[5] Il s'agit d'une bibliothèque de patrons de classes développée chez Hewlett Packard et appartenant au domaine public. Cette bibliothèque est disponible notamment sur Visual C++ 6.

```
void pile::push(int val)
{
   list<int>::push_front(val);  // ajoute un élément en tête de liste
}
void pile::pop(int & val)
{
   if(!empty())
   {
      val = *begin();   // val ppv le premier élément de la liste
      pop_front();      // supprime le premier élément
   }
}
void pile::affiche()
{
   list<int>::iterator i;
   cout << endl;
   for(i=begin();i!=end();i++) cout<<*i; // affichage des éléments de la liste
}
```

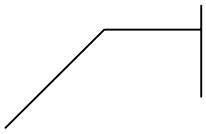
Remarque : le code des fonctions définies précédemment utilise des caractéristiques des classes de la bibliothèque STL, notamment la notion d'itérateur. Sans rentrer dans les détails, un itérateur généralise la notion de pointeur sur un élément. Par exemple, un itérateur sur une liste correspond à un pointeur sur un élément de la liste. Si `i` est un itérateur sur un élément de la liste, `*i` représente l'élément pointé et `i++` permet de passer à l'élément suivant de la liste.

```
void main()
{
   pile p;
   p.push(1);
   p.push(3);
   p.affiche();

   int temp;
   p.pop(temp);
   p.pop(temp);
   p.affiche();
}
```

Création d'un nouveau patron de classes par dérivation d'un patron existant

Dans le même esprit que précédemment, on peut définir un patron de piles pour différents types à partir d'un patron de listes chaînées. La définition devient la suivante :

```
#include<list>
#include<iostream.h>
using namespace std;

template <class T> class pile:private list<T>
{
public:
   void push(T);
   void pop(T &);
   void affiche();
};
```



`pile<T>` est un patron créé par dérivation du patron `list<T>`

```
template <class T> void pile<T>::push(T val)
{
   list<T>::push_front(val);
}

template <class T> void pile<T>::pop(T & val)
{
   if(!empty())
   {
      val = *begin();
      pop_front();
   }
}

template <class T> void pile<T>::affiche()
{
   list<int>::iterator i;
   cout << endl;
   for(i=begin();i!=end();i++) cout<<*i;
}

void main()
{
   pile<int> p;
   p.push(1);
   p.push(3);
   p.affiche();

   int temp;
   p.pop(temp);
   p.pop(temp);
   p.affiche();
}
```

10.8 Conclusion

La bibliothèque STL constitue une motivation particulière pour l'utilisation de patrons. En effet, cette bibliothèque met à disposition divers patrons de conteneurs (tableaux, listes, files, piles,…) qui peuvent faciliter le développement de certaines applications.

Il est à noter que d'autres bibliothèques de patrons sont parfois disponibles avec certains compilateurs. Par exemple, une biliothèque de patrons de conteneurs (listes, tableaux, tableaux associatifs) est fournie avec la bibliothèque MFC[6] (Microsoft Foundation Classes).

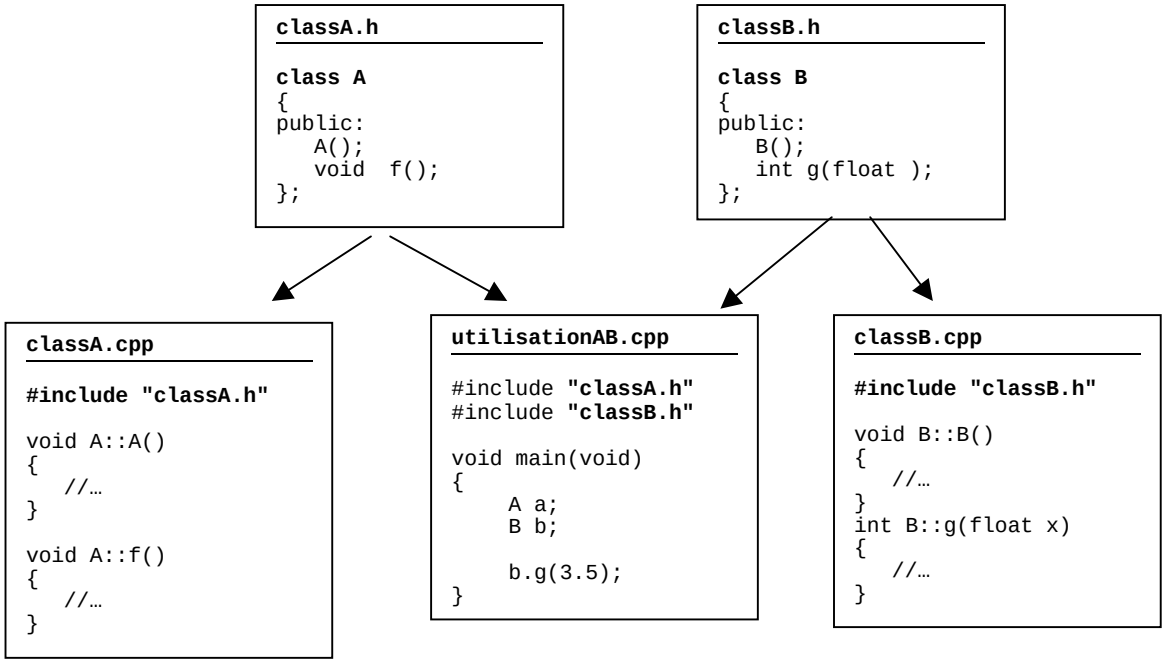
[6] Les MFC permettent de développer des applications Windows.

11 ANNEXE

11.1 Travailler sous forme de projet en C++

Pour les applications nécessitant beaucoup de classes, il est souhaitable que celles-ci figurent dans des fichiers différents. On travaille alors de la manière suivante :

- les déclarations des classes sont faites dans des fichiers d'extension `.h` (fichiers d'entête)
- les définitions des fonctions membres sont dans des fichiers d'extension `.cpp`
- l'utilisation de ces classes apparaît finalement dans un fichier d'extension **.**`cpp` contenant la fonction `void main(void)`.



On suppose ici que le projet contient `classA.cpp`**,** `classB.cpp et utilisationAB.cpp`**.**

• `#include "classA.h"` est une directive d'inclusion de fichier. Elle permet d'inclure tout le contenu du fichier `classA.h` (du répertoire courant) à l'emplacement de la directive, et ce juste avant la compilation. Si l'on compile par exemple uniquement le fichier `classA.cpp`, la déclaration de la classe `A` apparaît ainsi avant la définition de ses fonctions membre.

• Les trois fichiers `classA.cpp`, `classB.cpp` et `utilisationAB.cpp` appartiennent au même projet. C'est « un peu comme » si l'on avait écrit le contenu de ces trois fichiers dans un même fichier source. Que remarque-t-on alors ? Il y a deux fois la directive d'inclusion `#include "classA.h"` et deux fois `#include "classB.h"` ce qui revient à dire que l'on a défini deux fois la classe `A` et deux fois la classe `B`. Ceci ne peut pas fonctionner. En fait, on ne devrait voir apparaître qu'une seule fois la déclaration de chacune des classes, même si l'on utilise plusieurs fois la directive `#include "classA.h"` dans l'ensemble du projet.

11.2 Les directives de compilation conditionnelle.

Il est possible d'utiliser plusieurs fois une directive d'inclusion d'un fichier de déclaration de classe (.h) dans un projet sans qu'il y ait de problème si l'on munit la déclaration de classe de directives de *compilation conditionnelle.*

Dans l'exemple suivant, tout ce qu'il y a entre la directive `#ifndef MACHIN et #endif` n'est considéré par le compilateur que si l'identificateur `MACHIN` n'est pas défini par une directive du type `#define MACHIN.`

```
#ifndef MACHIN
   //si MACHIN n'est pas défini, voilà ce que l'on compile
   //on ne compile donc cette partie que si MACHIN n'est pas défini !
#endif
```

Que se passe-t-il lors de la compilation d'un fichier tel que le suivant ?

```
#ifndef MACHIN
#define MACHIN //si MACHIN n'est pas défini, on définit MACHIN
class A
{
public:
      A();
   void  f();
};
#endif

#ifndef MACHIN   // ici  MACHIN est défini, on ne compile donc pas la suite
#define MACHIN
class A
{
public:
   A();
 void f();
};
#endif
```

Réponse : grâce aux directives de compilation conditionnelle**,** la classe `A` n'est déclarée qu'une seule fois. C'est cette technique qui est utilisée pour l'écriture des fichiers de déclaration des classes. Pour que tout se passe bien, le fichier de déclaration d'une classe est généralement écrit ainsi :

classA.h

```
#ifndef __CLASS_A__
#define __CLASS_A__

class A
{
public:
   A();
 void    f();
};

#endif
```

classB.h

```
#ifndef __CLASS_B__
#define __CLASS_B__

class B
{
public:
   B();
   int   g(float );
};

#endif
```

Notons que le choix des noms **__CLASS_A__** et **__CLASS_B__** est arbitraire. Ensuite, la définition des fonctions membres peut se faire dans d'autres fichiers d'extension `.cpp`.

classA.cpp

```
#include "classA.h"

A::A()
{
   //…
}

void A::f()
{
   //…
}
```

classB.cpp

```
#include "classB.h"

B::B()
{
   //…
}

int B::g(float a)
{
   //…
}
```

INDEX