INTRODUCTION

Ce cours suppose le langage C connu.

On trouvera le listing du corrigé des exercices à la fin de chaque chapitre.
Une disquette, contenant le programme source de ces exercices est fournie avec le polycopié.
Ces programmes ont été mis au point sous environnement BORLAND C++ sous WINDOWS.

Pour avancer un peu plus vite et aborder l'essentiel de la Programmation Orientée Objet (P.O.O.), on pourra étudier les chapitres et paragraphes marqués de ***, dans un deuxième temps.

En principe, selon le principe de la compatibilité «ascendante», tout programme écrit en C fonctionne sous un environnement C++.

Bibliographie:

- Le C++ sous TURBO BORLAND C++ - Claude DELANNOY - EYROLLES
- Programmation Windows en Turbo C++ et Borland C++ - Gérard LEBLANC - EYROLLES
- Programmer Windows avec Turbo C++ - Claude DELANNOY - EYROLLES

CHAPITRE 1
INCOMPATIBILITES ENTRE C ET C++
NOUVELLES POSSIBILITES DU C++

Dans ce premier chapitre sont exposés les «plus» et les différences du C++ par rapport au C, sans toutefois aborder la programmation orientée objet (P.O.O.) qui sera vue lors d'un prochain chapitre.

I- COMMENTAIRES

En C++, les commentaires s'écrivent derrière un // et se terminent avec la fin de la ligne.

Exemple:
```
void main()   // ceci est un commentaire
{             // noter que le qualificatif «void» est obligatoire
int n=2;      // devant «main()»
}
```

II- LE QUALIFICATIF « CONST »

En C++, de telles déclarations de tableaux sont autorisées:

```
const int taille = 5;
char tab1[taille];
float tab2[taille + 1][taille *2];
```

III- DECLARATION DES VARIABLES

En C++, on peut déclarer les variables LOCALES au moment où on en a besoin. Si une variable locale est déclarée au début d'un bloc, sa portée est limitée à ce bloc.

Exemple:

```
void main()
{
for ( int i = 0;i<10;i++ )      // la variable i est connue de toute la fonction
        {                       // «main()»
    int j;                      // la variable j n'est connue que du bloc
    ...;
    ...;
    }
}
```

Ce exemple est équivalent à:

```
void main()
{
int i;
for ( i = 0;i<10;i++ )  // la variable i est connue de toute la fonction
        {                       // « main() »
        int j;          // la variable j n'est connue que du bloc
        ...;
        ...;
        }
}
```

Cette possibilité autorise une gestion plus précise de la mémoire, mais peut nuire à la structuration des programmes.

IV- LES NOUVELLES POSSIBILITES D'ENTREES/SORTIES

On peut utiliser en C++ les fonctions d'entrées/sorties classiques du C (printf, scanf, puts, gets, putc, getc ...), à condition de déclarer le fichier d'en-tête stdio.h.

Il existe de nouvelles possibilités en C++, à condition de déclarer le fichier d'en-tête **iostream.h**.
Ces nouvelles possibilités ne nécessitent pas de FORMATAGE des données.

Sortie sur écran: l'opérateur *cout*:

Exemples:

```
cout << "BONJOUR";        // équivalent à puts("BONJOUR");

int n = 25;
cout << "Valeur: ";       // équivalent à puts("Valeur");
cout << n;                // équivalent à printf("%d",n);
```

On peut encore écrire directement:

```
cout <<"Valeur:" << n;

cout <<"\n ";       // pour aller à la ligne

char c = 'a';
cout << c;          // affiche le caractère, il faut utiliser
                    // printf (formatage), pour obtenir le code
```

ASCII

Cette notation sera justifiée lors du chapitre sur les flots.
L'opérateur *cout* permet d'afficher des nombres entiers ou réels, des caractères, des chaînes de caractères, des pointeurs autre que de type *char (valeur de l'adresse).

Exemple (à tester) et exercice I-1:

```
#include <iostream.h>
#include <conio.h>
void main()
{
int i,n=25,*p;
char *tc="On essaie cout !";
float x = 123.456;
cout<<"BONJOUR\n";
cout<<tc<<"\n";
cout<<"BONJOUR\n"<<tc<<"\n";
cout<<"n= "<<n<<"  x= "<<x<<"  p= "<<p<<"\n";
getch() ;
}
```

Saisie clavier: l'opérateur *cin*:

Exemples:
```
int n;
cout<<"Saisir un entier:  ";
cin >> n;                  // équivalent à scanf("%d",&n);

int a, b;
cin >> a >> b; // saisie de 2 entiers séparés par un Retour Charriot
```

Cette notation sera justifiée lors du chapitre sur les flots.
L'opérateur *cin* permet de saisir des nombres entiers ou réels, des caractères, des chaînes de caractères.

Exemple (à tester) et exercice I-2:

Tester cet exemple plusieurs fois, notamment en effectuant des saisies erronées de sorte d'évaluer les « anomalies » de fonctionnement de *cin.*

```
#include <iostream.h>
#include <conio.h>
void main()
{
int n;
char tc[30],c;
float x;
cout<<"Saisir un entier:";
cin>>n;
cout<<"Saisir un réel:";
cin>>x;
cout<<"Saisir une phrase:";
cin>>tc;
cout<<"Saisir une lettre:";
cin>>c;
cout<<"Relecture: "<<n<<" "<<x<<" "<<tc<<" "<<c<<"\n";  getch() ;
}
```

V- LES CONVERSIONS DE TYPE

Le langage C++ autorise les conversions de type entre variables de type ***char, int, float, double:***

Exemple:

```
void main()
{
char c=0x56,d=25,e;
int i=0x1234,j;
float r=678.9,s;
j = c;                  // j vaut 0x0056
j = r;                  // j vaut 678
s = d;                  // s vaut 25.0
e = i;                  // e vaut 0x34
}
```

Une conversion de type float --> int ou char est dite *dégradante*
Une conversion de type int ou char --> float est dite *non dégradante*

VI- LES CONVERSIONS DE TYPE LORS D'APPEL A FONCTION

Le C++, contrairement au C, autorise, dans une certaine mesure, le non-respect du type des arguments lors d'un appel à fonction: le compilateur opère alors une conversion de type.

Exemple:

```
double ma_fonction(int u, float f)
{
.....;                  // fonction avec passage de deux paramètres
.....;
}

void main()
{
char c; int i, j; float r; double r1, r2, r3, r4;
r1 = ma_fonction( i, r );       // appel standard
r2 = ma_fonction( c, r);        // appel correct, c est converti en int
r3 = ma_fonction( i, j);        // appel correct, j est converti en float
r4 = ma_fonction( r, j);        // appel correct, r est converti en int
                                // et j est converti en float
}
```

Exercice I-3:

Ecrire une fonction **float puissance(float x,int n)** qui renvoie x^n. La mettre en oeuvre en utilisant les propriétés de conversion de type.

VII- LES ARGUMENTS PAR DEFAUT

En C++, on peut préciser la valeur prise par défaut par un argument de fonction. Lors de l'appel à cette fonction, si on omet l'argument, il prendra la valeur indiquée par défaut, dans le cas contraire, cette valeur par défaut est ignorée.

Exemple:

```
void f1(int n = 3)      // par défaut le paramètre n vaut 3
{
....;
}

void f2(int n, float x = 2.35)  // par défaut le paramètre x vaut 2.35
{
....;
}

void f3(char c, int n = 3, float x = 2.35)      // par défaut le paramètre
n vaut 3
{                                               // et le paramètre x vaut
2.35
....;
}

void main()
{
char a = 0; int i = 2; float r = 5.6;
f1(i);          // l'argument n vaut 2, l'initialisation par défaut est
ignorée
f1();           // l'argument n prend la valeur par défaut
f2(i,r);        // les initialisations par défaut sont ignorées
f2(i);          //  le second paramètre prend la valeur par défaut
// f2();   interdit
f3(a, i, r);    //  les initialisations par défaut sont ignorées
f3(a, i);       //  le troisième paramètre prend la valeur par défaut
f3(a);          //  le deuxième et la troisième paramètres prennent les
valeurs         }               //  par défaut
```

Remarque:

Les arguments, dont la valeur est fournie par défaut, doivent OBLIGATOIREMENT se situer en fin de liste.
La déclaration suivante est interdite:

```
void f4(char c=2, int n)
{
....;
}
```

Exercice I-4:

Reprendre l'exercice précédent. Par défaut, la fonction puissance devra fournir x^4.

VIII- LA SURDEFINITION DES FONCTIONS

Le C++ autorise la définition de fonctions *différentes* et portant *le même nom.* Dans ce cas, il faut les différencier par le type des arguments.

Exemple (à tester) et exercice I-5:

```
#include <iostream.h>
#include <conio.h>
void test(int n = 0,float x = 2.5)
{
cout <<"Fonction N°1 :  ";
cout << "n= "<<n<<"  x="<<x<<"\n";
}

void test(float x = 4.1,int n = 2)
{
cout <<"Fonction N°2 :  ";
cout << "n= "<<n<<"  x="<<x<<"\n";
}

void main()
{
int i = 5; float r = 3.2;
test(i,r);        // fonction N°1
test(r,i);        // fonction N°2
test(i);          // fonction N°1
test(r); // fonction N°2

// les appels suivants, ambigüs, sont rejetés par le compilateur
// test();
// test (i,i);
// test (r,r);
// les inialisations par défaut de x à la valeur 4.1
// et de n à 0 sont inutilisables
getch() ;}
```

Exemple (à tester) et exercice I-6:

```
#include <iostream.h>
#include <conio.h>
void essai(float x,char c,int n=0)
{cout<<"Fonction N°1: x = "<<x<<" c = "<<c<<" n = "<<n<<"\n";}

void essai(float x,int n)
{cout<<"Fonction N°2: x = "<<x<<" n = "<<n<<"\n";}
```

```
void main()
{
char l='z';int u=4;float y = 2.0;
essai(y,l,u);     // fonction N°1
essai(y,l);       // fonction N°1
essai(y,u);       // fonction N°2
essai(u,u);       // fonction N°2
essai(u,l);       // fonction N°1
// essai(y,y); rejet par le complateur
essai(y,y,u);     // fonction N°1
getch() ;}
```

Exercice I-7:

Ecrire une fonction **void affiche (float x, int n = 0)** qui affiche x^n (avec en particulier $x^0 = 1$ et donc, $0^0 = 1$).
Ecrire *une autre* fonction **void affiche(int n, float x=0)** qui affiche x^n (avec en particulier $0^n = 0$ et donc, $0^0 = 0$).
Les mettre en oeuvre dans le programme principal, en utilisant la propriété de surdéfinition.

Remarque: Cet exemple conduit à une erreur de compilation lors d'appel de type m^n avec m et n entiers.

IX- LES OPERATEURS *new* ET *delete*

Ces deux opérateurs remplacent *malloc* et *free* (que l'on peut toujours utiliser). Ils permettent donc de réserver de la place en mémoire, et d'en libérer.

Exemples:
```
int *ad;          // déclaration d'un pointeur sur un entier
ad = new int;     // réservation de place en mémoire pour un entier
```

On aurait pu déclarer directement **int *ad = new int;**

```
char *adc;
adc = new char[100]; // réservation de place en mémoire pour 100 caractères
```

On aurait pu déclarer directement **char *adc = new char[100];**

```
int *adi;
adi = new int[40]; // réservation de place en mémoire pour 40 entiers
```

On aurait pu déclarer directement **int *adi = new int[40];**

```
delete ad;        // libération de place
delete adc;
delete adi;
```

Exemple (à tester) et exercice I-8:

```
#include <iostream.h>   // new et delete
#include <conio.h>      // verifier en testant que cin et cout posent les
                        // memes pb que scanf et printf (flux d'E-S)
void main()
{
int *ad = new int;
char *adc;
adc = new char[25];

cout<<"Entrer un nombre:";
cin>>*ad;
cout<<"Voici ce nombre:"<<*ad;

cout<<"\nEntrer une phrase:";
cin>>adc;
cout<<"Voici cette phrase:"<<adc;

delete ad;
delete adc;
getch() ;}
```

Exercice I-9:

Déclarer un tableau de 5 réels. Calculer et afficher leur moyenne.

Remarques:

- Il ne faut pas utiliser conjointement *malloc et delete* ou bien *new* et *free.*
- En TURBO C++, l'opérateur *new* permet de réserver au maximum 64 Koctets en mémoire; la fonction *set_new_handler* permet de gérer cette limite.

X- NOTION DE REFERENCE

En C, la notation *&n* signifie « l'adresse de la variable n »
En C++, cette notation possède deux significations:

- Il peut toujours s'agir de l'adresse de la variable n
- Il peut aussi s'agir de la *référence* à n

Seul le contexte du programme permet de déterminer s'il s'agit de l'une ou l'autre des deux significations.

Exemple:

```
int n;
int &p = n;     // p est une référence à n
                // p occupe le même emplacement mémoire que n
n = 3;
cout<< p;       // l'affichage donnera 3
```

XI -PASSAGE DE PARAMETRE PAR REFERENCE

Rappel:

En C, comme en C++, un sous-programme ne peut modifier la valeur d'une variable locale passée en argument de fonction. Pour se faire, en C, il faut passer *l'adresse* de la variable.

Exemple (à tester) et exercices I-10 et I-11:

```
#include <iostream.h>        // passage par valeur
#include <conio.h>
void echange(int a,int b)
{
int tampon;
tampon = b; b = a; a = tampon;
cout<<"Pendant l'échange: a = "<<a<<" b = "<<b<<"\n";
}
void main()
{
int u=5,v=3;
cout<<"Avant echange: u = "<<u<<" v = "<<v<<"\n";
echange(u,v);
cout<<"Apres echange: u = "<<u<<" v = "<<v<<"\n";
getch() ;}
```

L'échange n'a pas lieu.

```
#include <iostream.h>                // passage par adresse
#include <conio.h>
void echange(int *a,int *b)
{
int tampon;
tampon = *b; *b = *a; *a = tampon;
cout<<"Pendant l'échange: a = "<<*a<<" b = "<<*b<<"\n";
}
void main()
{
int u=5,v=3;
cout<<"Avant echange: u = "<<u<<" v = "<<v<<"\n";
echange(&u,&v);
cout<<"Apres echange: u = "<<u<<" v = "<<v<<"\n";
getch() ;}
```

L'échange a lieu.

En C++, on préférera *le passage par référence:*

Exemple (à tester) et exercice I-12:

```
#include <iostream.h>        //passage par référence
#include <conio.h>
void echange(int &a,int &b)    // référence à a et b
{
int tampon;
tampon = b; b = a; a = tampon;
cout<<"Pendant l'échange: a = "<<a<<" b = "<<b<<"\n";
}

void main()
{
int u=5,v=3;
cout<<"Avant echange: u = "<<u<<" v = "<<v<<"\n";
echange(u,v);
cout<<"Apres echange: u = "<<u<<" v = "<<v<<"\n";
getch() ;}
```

L'échange a lieu. Le compilateur prend en charge le passage par adresse si celui-ci est nécessaire.
Remarquer la simplification de l'écriture de la fonction.

XII- CORRIGE DES EXERCICES

Exercice I-3:

```
#include <iostream.h>
#include <conio.h>
float puissance(float x,int n)
{
float resultat=1;
for(int i=1;i<=n;i++)resultat = resultat * x;
return resultat;
}

void main()
{
char c=5;int i=10,j=6; float r=2.456,r1,r2,r3,r4,r5;
r1 = puissance(r,j);
r2 = puissance(r,c);
r3 = puissance(j,i);
r4 = puissance(j,r);
r5 = puissance(0,4);
cout << "r1 = " <<r1<<"\n";
cout << "r2 = " <<r2<<"\n";
cout << "r3 = " <<r3<<"\n";
cout << "r4 = " <<r4<<"\n";
cout << "r5 = " <<r5<<"\n";
getch() ;}
```

Exercice I-4:

```
#include <iostream.h>
#include <conio.h>
float puissance(float x,int n=4)
{
float resultat=1;
for(int i=1;i<=n;i++)resultat = resultat * x;
return resultat;
}
void main()
{
int j=6; float r=2.456,r1,r2,r3,r4,r5;
r1 = puissance(r,j);
r2 = puissance(r);
r3 = puissance(1.4,j);
r4 = puissance(1.4);
cout << "r1 = " <<r1<<"\n";
cout << "r2 = " <<r2<<"\n";
cout << "r3 = " <<r3<<"\n";
cout << "r4 = " <<r4<<"\n";
getch() ;}
```

Exercice I-7:

```
#include <iostream.h>
#include <conio.h>
void affiche(float x,int n=0)
{int i = 1;float resultat = 1;
for(;i<=n;i++)resultat = resultat * x;
cout << "x = "<<x<< " n = " << n << " resultat = " << resultat <<"\n";}

void affiche(int n,float x=0)
{int i = 1;float resultat = 1;
if (n!=0){for(;i<=n;i++)resultat = resultat * x;}
else (resultat = 0);
cout << "n = "<<n<< " x = " << x << " resultat = " << resultat <<"\n";
}
void main()
{
int u=4,v=0;float y = 2.0,z=0;
affiche(u);
affiche(y);
affiche(y,u);
affiche(u,y);
affiche(v,z);
affiche(z,v);
getch() ;}
```

Exercice I-9:

```
#include <iostream.h>
#include <conio.h>
void main()
{
float moyenne =0,*tab = new float[5];
int i=0;
for(;i<5;i++)
        {
        cout<<"Entrer un nombre: ";
        cin>>tab[i];
        moyenne = moyenne + tab[i];
        }
moyenne = moyenne/5;
cout <<"Moyenne = "<<moyenne<<"\n";
delete tab;
getch() ;}
```

CHAPITRE 2

PROGRAMMATION ORIENTE OBJET: NOTION DE CLASSE

I- INTRODUCTION

On attend d'un programme informatique
- l'exactitude (réponse aux spécifications)
- la robustesse (réaction correcte à une utilisation « hors normes »)
- l'extensibilité (aptitude à l'évolution)
- la réutilisabilité (utilisation de modules)
- la portabilité (support d'une autre implémentation)
- l'efficience (performance en termes de vitesse d'exécution et de consommation mémoire)

Les langages évolués de type C ou PASCAL, reposent sur le principe de la programmation structurée (algorithmes + structures de données)
Le C++ et un langage orienté objet. Un langage orienté objet permet la manipulation de *classes.* Comme on le verra dans ce chapitre, la classe généralise la notion de structure.
Une classe contient des variables (ou « données ») et des fonctions (ou « méthodes ») permettant de manipuler ces variables.
Les langages « orientés objet » ont été développés pour faciliter l'écriture et améliorer la qualité des logiciels en termes de modularité.
Un langage orienté objet sera livré avec une bibliothèque de classes. Le développeur utilise ces classes pour mettre au point ses logiciels.

Rappel sur la notion de prototype de fonction:

En C++, comme en C, on a fréquemment besoin de déclarer des *prototypes* de fonctions.
Par exemple, dans les fichiers d'en-tête (de type *.h), sont déclarés les *prototypes* des fonctions appelées par le programme.
Le *prototype* d'une fonction est constitué du nom de la fonction, du type de la valeur de retour, du type des arguments à passer

Exemples: **void ma_fonction1()**

void ma_fonction2(int n, float u)	// prototype «complet»
void ma_fonction2(int, float)	// prototype «réduit»
int ma_fonction3(char *x)	// prototype «complet»
int ma_fonction3(char *)	// prototype «réduit»
int ma_fonction4(int &u)	// prototype «complet»
int ma_fonction4(int &)	// prototype «réduit»

On utilise indifféremment, dans les fichiers d'en-tête, le prototype complet ou le prototype réduit.

II- NOTION DE CLASSE

Exemple (à tester) et exercice II-1 :
(il faut ajuster la temporisation aux performances de la machine utilisée)

```
#include <iostream.h>       // les classes
#include <conio.h>
class point
{
int x,y;
public:  void initialise(int,int);
         void deplace(int,int);
         void affiche();
};

void point::initialise(int abs,int ord)
{x = abs; y = ord;}

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}

void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

void main()
{
point a,b;
a.initialise(1,4);
a.affiche();
tempo(10);
a.deplace(17,10);
a.affiche();
b = a;        // affectation autorisee
tempo(15);
clrscr();
b.affiche();
getch() ;}
```

«point» est une ***classe***. Cette classe est constituée des ***données*** *x* et y et des ***fonctions membres*** **(*ou méthodes*) «** initialise », « deplace », « affiche ». On déclare la classe en début de programme (données et prototype des fonctions membres), puis on définit le contenu des fonctions membres.
Les données x et y sont dites ***privées.*** Ceci signifie que l'on ne peut les manipuler qu'au travers des fonctions membres. On dit que le langage C++ réalise ***l'encapsulation des données.***

a et b sont des ***objets*** de classe «point», c'est-à-dire des variables de type «point». On a défini ici un nouveau type de variable, propre à cet application, comme on le fait en C avec les structures.

Suivant le principe dit de « l'encapsulation des données », la notation **a.x** est interdite.

Exercice II-2:

Utiliser la classe « point » précédente. Ecrire une fonction de prototype **void test()** dans laquelle on déclare un point u, on l'initialise, on l'affiche , on le déplace et on l'affiche à nouveau. Le programme principal *main* ne contient que l'appel à *test*.

Exercice II-3:

Ecrire une fonction de prototype **void test(point &u)** (référence) similaire. Ne pas déclarer de point local dans *test.* Déclarer un point local a dans le programme principal *main* et appeler la fonction *test* en passant le paramètre a.

Exercice II-4:

Ecrire une fonction de prototype **point test()** qui retourne un point. Ce point sera initialisé et affiché dans *test* puis déplacé et à nouveau affiché dans *main.*

III- NOTION DE CONSTRUCTEUR

Un constructeur est une fonction membre *systématiquement exécutée* lors de la déclaration d'un objet statique, automatique, ou dynamique.

On ne traitera dans ce qui suit que des objets automatiques.

Dans l'exemple de la classe *point,* le constructeur remplace la fonction membre *initialise.*

Exemple (à tester) et exercice II-5:

```
#include <iostream.h>       // constructeur
#include <conio.h>

class point
{
int x,y;
public:  point(); // noter le type du constructeur (pas de "void")
         void deplace(int,int);
         void affiche();
};

point::point() // initialisation par default
{x = 20; y = 10;}  // grace au constructeur
```

```
void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}

void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

void main()
{
point a,b;    // les deux points sont initialisés en 20,10
a.affiche();
tempo(10);
a.deplace(17,10);
a.affiche();
tempo(15);
clrscr();
b.affiche();
getch() ;}
```

Exemple (à tester) et exercice II-6:

```
#include <iostream.h>       // constructeur
#include <conio.h>

class point
{
int x,y;
public:  point(int,int); // noter le type du constructeur (pas de "void")
         void deplace(int,int);
         void affiche();
};

point::point(int abs,int ord) // initialisation par default
{x = abs; y = ord;}  // grace au constructeur, ici paramètres à passer

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}

void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

void main()
{
```

```
point a(20,10),b(30,20);  // les deux points sont initialises:a en 20,10  b en 30,20
a.affiche();
tempo(10);
a.deplace(17,10);
a.affiche();
tempo(15);
clrscr();
b.affiche();
getch() ;}
```

Exercice II-7: Reprendre l'exercice II-2, en utilisant la classe de l'exercice II-6

Exercice II-8: Reprendre l'exercice II-3, en utilisant la classe de l'exercice II-6

Exercice II-9: Reprendre l'exercice II-4, en utilisant la classe de l'exercice II-6

IV- NOTION DE DESTRUCTEUR

Le destructeur est une fonction membre *systématiquement exécutée* «à la fin de la vie » d'un objet statique, automatique, ou dynamique.
On ne peut pas passer de paramètres par le destructeur.

On ne traitera dans ce qui suit que des objets automatiques.

Exemple (à tester) et exercice II-10:

```
#include <iostream.h>       // destructeur
#include <conio.h>

class point
{
int x,y;
public:  point(int,int);
          void deplace(int,int);
          void affiche();
         ~point();        // noter le type du destructeur
};

point::point(int abs,int ord) // initialisation par default
{x = abs; y = ord;}  // grace au constructeur, ici paramètres à passer

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}
```

```
point::~point()
{cout<<"Frapper une touche...";getch();
cout<<"destruction du point x ="<<x<<" y="<<y<<"\n";}

void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

void test()
{
point u(3,7);
u.affiche();
tempo(20);
}

void main()
{point a(1,4);a.affiche();tempo(20);
test();
point b(5,10);b.affiche();
getch() ;}
```

V- ALLOCATION DYNAMIQUE

Lorsque les membres données d'une classe sont des pointeurs, le constructeur est utilisé pour l'allocation dynamique de mémoire sur ce pointeur.
Le destructeur est utilisé pour libérer la place.

Exemple (à tester) et exercice II-11:

```
#include <iostream.h>    // Allocation dynamique de données membres
#include <stdlib.h>
#include <conio.h>

class calcul
{int nbval,*val;
public:          calcul(int,int);  // constructeur
         ~calcul();    // destructeur
         void affiche();
};

calcul::calcul(int nb,int mul)  //constructeur
{int i;
nbval = nb;
val = new int[nbval];  // reserve de la place
for(i=0;i<nbval;i++)val[i] = i*mul;
}

calcul::~calcul()
{delete val;}  // abandon de la place reservee
```

```
void calcul::affiche()
{int i;
for(i=0;i<nbval;i++)cout<<val[i]<<" ";
cout<<"\n";
}

void main()
{
clrscr();

calcul suite1(10,4);
suite1.affiche();
calcul suite2(6,8);
suite2.affiche();
getch() ;}
```

VI- EXERCICES RECAPITULATIFS

Exemple (à tester) et exercice II-12:

Cet exemple ne fonctionne qu'en environnement DOS. Il utilise les fonctions graphiques classiques du TURBO C. On crée une classe « losange », les fonctions membres permettent de manipuler ce losange.

```
 #include <graphics.h>
#include <conio.h>
#include <iostream.h>
#include <dos.h>

class losange
{
int x,y,dx,dy,couleur;
public:
losange();
void deplace(int,int,int);
void affiche();
void efface();
};

losange::losange() // constructeur
{x=100;y=100;dx=60;dy=100;couleur=BLUE;}

void losange::deplace(int depx,int depy,int coul)
{x=x+depx;y=y+depy;couleur=coul;}

void losange::affiche()
{int tab[10];
tab[0]=x;tab[1]=y;tab[2]=x+dx/2;tab[3]=y+dy/2;
```

```
tab[4]=x;tab[5]=y+dy;tab[6]=x-dx/2;tab[7]=y+dy/2;
tab[8]=x;tab[9]=y;
setfillstyle(SOLID_FILL,couleur);
fillpoly(5,tab);
}

void losange::efface()
{int tab[10];
tab[0]=x;tab[1]=y;tab[2]=x+dx/2;tab[3]=y+dy/2;
tab[4]=x;tab[5]=y+dy;tab[6]=x-dx/2;tab[7]=y+dy/2;
tab[8]=x;tab[9]=y;
setcolor(getbkcolor());  // pour effacer le contour
setfillstyle(SOLID_FILL,getbkcolor());
fillpoly(5,tab);
}

void init_graph()
{
int gd,gm;
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"c:\\cplus\\bgi");
setbkcolor(YELLOW);
}

void main()
{
losange l;
init_graph();
l.affiche();
getch();closegraph();
}
```

Exercice II- 13: Modifier le programme principal de sorte de faire clignoter le losange tant que l'utilisateur n'a pas appuyé sur une touche.

Exercice II- 14: Modifier le programme principal de sorte de déplacer le losange d'une position à une autre position, tant que l'utilisateur n'a pas appuyé sur une touche.

VII- CORRIGE DES EXERCICES

Exercice II-2:

```
#include <iostream.h>       // les classes
#include <conio.h>

class point
{
int x,y;
```

```
public:  void initialise(int,int);
         void deplace(int,int);
         void affiche();
};

void point::initialise(int abs,int ord)
{x = abs; y = ord;}

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}


void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

void test()
{
point u;
u.initialise(1,4);u.affiche();
tempo(10);
u.deplace(17,10);u.affiche();
}

void main()
{test();getch() ;}
```

Exercice II-3:

```
#include <iostream.h>       // les classes
#include <conio.h>

class point
{
int x,y;
public:  void initialise(int,int);
         void deplace(int,int);
         void affiche();
};

void point::initialise(int abs,int ord)
{x = abs; y = ord;}

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}
```

```
void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

void test(point &u)
{
u.initialise(1,4);u.affiche();
tempo(10);
u.deplace(17,10);u.affiche();
}

void main()
{point a;test(a);getch() ;}
```

Exercice II-4:

```
#include <iostream.h>       // les classes
#include <conio.h>

class point
{
int x,y;
public:  void initialise(int,int);
         void deplace(int,int);
         void affiche();
};

void point::initialise(int abs,int ord)
{x = abs; y = ord;}

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}


void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

point test()
{point u;
u.initialise(1,4);u.affiche();return u;}

void main()
{point a;
a = test();
tempo(10);
a.deplace(17,10);a.affiche();getch() ;}
```

Exercice II-7:

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

class point
{
int x,y;
public:  point(int,int); // noter le type du constructeur (pas de "void")
         void deplace(int,int);
         void affiche();
};

point::point(int abs,int ord) // initialisation par default
{x = abs; y = ord;}  // grace au constructeur, ici paramètres à passer

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}

void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

void test()
{
point u(1,4);
u.affiche();
tempo(10);
u.deplace(17,10);
u.affiche();
}

void main()
{test();getch() ;}
```

Exercice II-8:

```
#include <iostream.h>      // les classes
#include <conio.h>

class point
{
int x,y;
public:  point(int,int); // noter le type du constructeur (pas de "void")
         void deplace(int,int);
         void affiche();
};
```

```
point::point(int abs,int ord) // initialisation par default
{x = abs; y = ord;}  // grace au constructeur, ici paramètres à passer

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}

void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

void test(point &u)
{
u.affiche();
tempo(10);
u.deplace(17,10);u.affiche();
}

void main()
{point a(1,4);test(a);getch() ;}
```

Exercice II-9:

```
#include <iostream.h>       // les classes
#include <conio.h>

class point
{
int x,y;
public:  point(int,int); // noter le type du constructeur (pas de "void")
         void deplace(int,int);
         void affiche();
};

point::point(int abs,int ord) // initialisation par default
{x = abs; y = ord;}  // grace au constructeur, ici paramètres à passer

void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<"  "<<y<<"\n";}

void tempo(int duree)
{float stop ;stop = duree*10000.0;for(;stop>0;stop=stop-1.0);}

point test()
```

```
{point u(5,6);
u.affiche();return u;}

void main()
{point a(1,4);
a.affiche();
tempo(15);
a = test();
tempo(10);
a.deplace(17,10);a.affiche();}
```

Exercice II-13:

```
void main()
{
losange l;
init_graph();
while(!kbhit())
{
l.affiche(); delay(500);
l.efface(); delay(500);
}
getch();closegraph();
}
```

Exercice II-14:

```
void main()
{
losange l;
init_graph();
while(!kbhit())
{
l.affiche(); delay(500);l.efface();
l.deplace(150,150,RED);
l.affiche();delay(500);l.efface();
l.deplace(-150,-150,BLUE);
}
getch();closegraph();
}
```

CHAPITRE 3

PROPRIETES DES FONCTIONS MEMBRES

I- SURDEFINITION DES FONCTIONS MEMBRES

En utilisant la propriété de surdéfinition des fonctions du C++, on peut définir plusieurs constructeurs, ou bien plusieurs fonctions membres, différentes, mais portant le même nom.

Exemple (à tester) et exercice III-1: Définition de plusieurs constructeurs:

```
#include <iostream.h>       // Surdefinition de fonctions
#include <conio.h>

class point
{
int x,y;
public:  point();   // constructeur 1
         point(int);// constructeur 2
         point(int,int);// constructeur 3
         void affiche();
};

point::point()  // constructeur 1
{x=0;y=0;}

point::point(int abs)   // constructeur 2
{x = abs; y = abs;}

point::point(int abs,int ord)    // constructeur 3
{x = abs; y = ord;}

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<" "<<y<<"\n";}

void main()
{
clrscr();
point a,b(5);
a.affiche();
b.affiche();
point c(3,12);
c.affiche();
getch() ;}
```

Exercice III-2: Surdéfinition d'une fonction membre
Ecrire une deuxième fonction **affiche** de prototype **void point::affiche(char *message)**

Cette fonction donne la possibilité à l'utilisateur d'ajouter, à la position du point, un message sur l'écran.

II- FONCTIONS MEMBRES « EN LIGNE »

Le langage C++ autorise la description des fonctions membres dès leur déclaration dans la classe. On dit que l'on écrit une fonction « inline ».
Il s'agit alors d'une « macrofonction »: A chaque appel, il y a génération du code de la fonction et non appel à un sous-programme.
Les appels sont donc plus rapides mais cette méthode génère davantage de code.

Exemple (à tester) et exercice III-3:
Comparer la taille des fichiers exIII_1.obj et exIII_3.obj

```
#include <iostream.h>       // Surdefinition de fonctions
#include <conio.h>

class point
{
int x,y;
public:  point(){x=0;y=0;}  // constructeur 1
         point(int abs){x=abs;y=abs;}// constructeur 2
         point(int abs,int ord){x=abs;y=ord;}// constructeur 3
         void affiche();
};

void point::affiche()
{gotoxy(x,y);cout<<"Je suis en "<<x<<" "<<y<<"\n";}

void main()
{
point a,b(5);
a.affiche();
b.affiche();
point c(3,12);
c.affiche();
getch() ;
}
```

III- INITIALISATION DES PARAMETRES PAR DEFAUT

Exemple (à tester) et exercice III-4:

```
#include <iostream.h>       // Fonctions membres « en ligne »
#include <conio.h>

class point
```

```
{
int x,y;
public:  point(int abs=0,int ord=2){x=abs;y=ord;}// constructeur
         void affiche(char* = "Position du point"); // argument par defaut
};

void point::affiche(char *message)
{gotoxy(x,y-1);cout<<message;
gotoxy(x,y);cout<<"Je suis en "<<x<<" "<<y<<"\n";}

void main()
{
point a,b(40);
a.affiche();
b.affiche("Point b");
char texte[10]="Bonjour";
point c(3,12);
c.affiche(texte);
getch() ;
}
```

IV- OBJETS TRANSMIS EN ARGUMENT D'UNE FONCTION MEMBRE

Quand on passe comme paramètre à une fonction membre ... un objet de la classe à laquelle appartient cette fonction:

1- Passage par valeur

Exemple (à tester) et exercice III-5:

```
#include <iostream.h>
#include <conio.h>              // objets transmis en argument d'une fonction
membre

class point
{
int x,y;
public:  point(int abs = 0,int ord = 2){x=abs;y=ord;}// constructeur
         int coincide(point);
};

int point::coincide(point pt)
{if ((pt.x == x) && (pt.y == y)) return(1);else return(0);}
// noter la dissymetrie des notations pt.x et x

void main()
{
int test1,test2;
```

```
point a,b(1),c(0,2);
test1 = a.coincide(b);
test2 = b.coincide(a);
cout<<"a et b:"<<test1<<" ou "<<test2<<"\n";
test1 = a.coincide(c);
test2 = c.coincide(a);
cout<<"a et c:"<<test1<<" ou "<<test2<<"\n";
getch() ;
}
```

Noter que l'on rencontre la notation « pt.x » ou « pt.y » pour la première fois. Elle n'est autorisée qu'à l'intérieur d'une fonction membre (x et y membres privés de la classe).

On verra plus tard que le passage d'un objet par valeur pose problème si certains membres de la classe sont des pointeurs. Il faudra alors prévoir une allocation dynamique de mémoire via un constructeur.

2- Passage par adresse

Exercice III-6: Modifier la fonction membre **coincide** de l'exercice prcédent de sorte que son prototype devienne **int point::coincide(point *adpt).** Ré-écrire le programme principal en conséquence.

3- Passage par référence

Exercice III-7: Modifier à nouveau la fonction membre **coincide** de sorte que son prototype devienne **int point::coincide(point &pt).** Ré-écrire le programme principal en conséquence.

V- EXERCICES RECAPITULATIFS

On définit la classe **vecteur** comme ci-dessous:

```
class vecteur
{float x,y;
public: vecteur(float,float);
        void homotethie(float);
        void affiche();
};

vecteur::vecteur(float abs =0.,float ord = 0.)
{x=abs;y=ord;}

void vecteur::homotethie(float val)
{x = x*val;  y = y*val;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}
```

Exercice III-8: La mettre en oeuvre dans **void main(),** en ajoutant une fonction membre **float det(vecteur)** qui retourne le déterminant des deux vecteurs (celui passé en paramètre et celui de l'objet).

Exercice III-9: Modifier la fonction **déterminant** de sorte de passer le paramètre par adresse.

Exercice III-10: Modifier la fonction **déterminant** de sorte de passer le paramètre par référence.

VI- OBJET RETOURNE PAR UNE FONCTION MEMBRE

Que se passe-t-il lorsqu'une fonction membre retourne elle-même un objet ?

1- Retour par valeur

Exemple (à tester) et exercice III-11: (la fonction concernée est la fonction **symetrique**)

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

// La valeur de retour d'une fonction est un objet
// Transmission par valeur

class point
{
int x,y;
public:  point(int abs = 0,int ord = 0){x=abs;y=ord;}// constructeur
         point symetrique();
         void affiche();
};

point point::symetrique()
{point res;
res.x = -x;  res.y = -y;
return res;
}

void point::affiche()
{cout<<"Je suis en "<<x<<" "<<" "<<y<<"\n";}

void main()
{point a,b(1,6);
a=b.symetrique();a.affiche();b.affiche();
getch() ;}
```

2- Retour par adresse (***)

Exemple (à tester) et exercice III-12:

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

// La valeur de retour d'une fonction est un objet
// Transmission par adresse

class point
{
int x,y;
public:  point(int abs = 0,int ord = 0){x=abs;y=ord;}// constructeur
         point *symetrique();
         void affiche();
};

point *point::symetrique()
{point *res;
res = new point;
res->x = -x;  res->y = -y;
return res;
}

void point::affiche()
{cout<<"Je suis en "<<x<<" "<<y<<"\n";}

void main()
{
point a,b(1,6);
a = *b.symetrique();a.affiche();b.affiche();
getch() ;}
```

3- Retour par référence (***)

La valeur retournée l'est par référence. On en verra l'usage dans un prochain chapitre.

Exemple (à tester) et exercice III-13:

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

// La valeur de retour d'une fonction est un objet
// Transmission par reference

class point
{
int x,y;
public:  point(int abs = 0,int ord = 0){x=abs;y=ord;}// constructeur
         point &symetrique();
         void affiche();
};
```

```
point &point::symetrique()  // La variable res est obligatoirement static
{static point res;          // Pour passer par reference
res.x = -x;  res.y = -y;
return res;
}

void point::affiche()
{cout<<"Je suis en "<<x<<" "<<y<<"\n";}

void main()
{
point a,b(1,6);
a=b.symetrique();a.affiche();b.affiche();
getch() ;}
```

Remarque: « res » et « b.symetrique »occupent le même emplacement mémoire (car « res » est une référence à « b.symetrique ». On déclare donc « res » comme variable static, sinon, cet objet n'existerait plus après être sorti de la fonction.

VII- EXERCICES RECAPITULATIFS

Exercice III-14: Reprendre la classe **vecteur.** Modifier la fonction **homotéthie**, qui retourne le vecteur modifié. (prototype: **vecteur vecteur::homotethie(float val)**).

Exercice III-15 (***): Même exercice, le retour se fait par adresse.

Exercice III-16 (***): Même exercice, le retour se fait par référence.

VIII- LE MOT CLE « THIS »

Ce mot désigne l'adresse de l'objet invoqué. Il est *utilisable uniquement* au sein d'une fonction membre.

Exemple (à tester) et exercice III-17:

```
#include <conio.h>          // le mot cle THIS: pointeur sur l'objet l'ayant appel
#include <iostream.h>       // utilisable uniquement dans une fonction membre

class point
{int x,y;
public:
point(int abs=0,int ord=0) // constructeur en ligne
{x=abs;y=ord;}
void affiche();
};
```

```
void point::affiche()
{cout<<"Adresse: "<<this<<" - Coordonnees: "<<x<<" "<<y<<"\n";}

void main()
{point a(5),b(3,15);
a.affiche();b.affiche();
getch() ;}
```

Exercice III-18: Remplacer, dans l'exercice III-6, la fonction **coincide** par la fonction suivante:

```
int point::coincide(point *adpt)
{if ((this->x == adpt->x) && (this->y == adpt->y))
 return(1);else return(0);}
```

IX- EXERCICE RECAPITULATIF

Exercice III-19: Reprendre la classe **vecteur**, munie du constructeur et de la fonction d'affichage. Ajouter
- Une fonction membre **float vecteur::prod_scal(vecteur)** qui retourne le produit scalaire des 2 vecteurs.
- Une fonction membre **vecteur vecteur::somme(vecteur)** qui retourne la somme des 2 vecteurs.

XI- CORRIGE DES EXERCICES

Exercice III-2:

```
#include <iostream.h>       // Surdefinition de fonctions
#include <conio.h>

class point
{
int x,y;
public:  point();   // constructeur 1
         point(int);// constructeur 2
         point(int,int);// constructeur 3
         void affiche();
         void affiche(char *); // argument de type chaine
};

point::point()  // constructeur 1
{x=0;y=0;}

point::point(int abs)   // constructeur 2
{x=y=abs;}

point::point(int abs,int ord)    // constructeur 3
```

```
{x = abs; y = ord;}

void point::affiche()            // affiche 1
{gotoxy(x,y);cout<<"Je suis en "<<x<<" "<<y<<"\n";}

void point::affiche(char *message)            // affiche 2
{gotoxy(x,y-1);cout<<message;
gotoxy(x,y);cout<<"Je suis en "<<x<<" "<<y<<"\n";}

void main()
{point a,b(5);
a.affiche();
b.affiche("Point b:");
point c(3,12);
char texte[10] = "Bonjour";
c.affiche(texte);
getch() ;}
```

Exercice III-6:

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

// objets transmis en argument d'une fonction membre - transmission de
l'adresse

class point
{
int x,y;
public:  point(int abs = 0,int ord = 2){x=abs;y=ord;}// constructeur
         int coincide(point *);
};

int point::coincide(point *adpt)
{if ((adpt->x == x) && (adpt->y == y)) return(1);else return(0);}
// noter la dissymetrie des notations pt->x et x

void main()
{point a,b(1),c(0,2);
int test1,test2;
test1 = a.coincide(&b);
test2 = b.coincide(&a);
cout<<"a et b:"<<test1<<" ou "<<test2<<"\n";
test1 = a.coincide(&c);
test2 = c.coincide(&a);
cout<<"a et c:"<<test1<<" ou "<<test2<<"\n";
getch() ;}
```

Exercice III-7:

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

// objets transmis en argument d'une fonction membre - transmission par
reference

class point
{
int x,y;
public:  point(int abs = 0,int ord = 2){x=abs;y=ord;}// constructeur
         int coincide(point &);
};

int point::coincide(point &pt)
{if ((pt.x == x) && (pt.y == y)) return(1);else return(0);}
// noter la dissymetrie des notations pt.x et x

void main()
{point a,b(1),c(0,2);
int test1,test2;
test1 = a.coincide(b);
test2 = b.coincide(a);
cout<<"a et b:"<<test1<<" ou "<<test2<<"\n";
test1 = a.coincide(c);
test2 = c.coincide(a);
cout<<"a et c:"<<test1<<" ou "<<test2<<"\n";
getch() ;}
```

Exercice III-8:

```
#include <iostream.h>
#include <conio.h>
        // Classe vecteur - Fonction membre determinant - Passage par valeur
class vecteur
{float x,y;
public: vecteur(float,float);
        void homotethie(float);
        void affiche();
        float det(vecteur);};

vecteur::vecteur(float abs =0.,float ord = 0.)
{x=abs;y=ord;}

void vecteur::homotethie(float val)
{x = x*val;  y = y*val;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

float vecteur::det(vecteur w)
```

```
{float res;
res = x * w.y - y * w.x;
return res;}

void main()
{vecteur v(2,6),u(4,8);
v.affiche();v.homotethie(2);v.affiche();
cout <<"Determinant de (u,v) = "<<v.det(u)<<"\n";
cout <<"Determinant de (v,u) = "<<u.det(v)<<"\n";getch() ;}
```

Exercice III-9:

```
#include <iostream.h>
#include <conio.h>
        // Classe vecteur - Fonction membre determinant - Passage par adresse
class vecteur
{float x,y;
public: vecteur(float,float);
        void homotethie(float);
        void affiche();
        float det(vecteur *);};

vecteur::vecteur(float abs =0.,float ord = 0.)
{x=abs;y=ord;}

void vecteur::homotethie(float val)
{x = x*val;  y = y*val;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

float vecteur::det(vecteur *w)
{float res;
res = x * w->y - y * w->x;
return res;}

void main()
{vecteur v(2,6),u(4,8);
v.affiche();v.homotethie(2);v.affiche();
cout <<"Determinant de (u,v) = "<<v.det(&u)<<"\n";
cout <<"Determinant de (v,u) = "<<u.det(&v)<<"\n";getch() ;}
```

Exercice III-10:

```
#include <iostream.h>
#include <conio.h>
        // Classe vecteur - Fonction membre determinant - Passage par reference
class vecteur
{float x,y;
public: vecteur(float,float);
```

```
        void homotethie(float);
        void affiche();
        float det(vecteur &);};

vecteur::vecteur(float abs =0,float ord = 0)
{x=abs;y=ord;}

void vecteur::homotethie(float val)
{x = x*val;  y = y*val;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

float vecteur::det(vecteur &w)
{float res;
res = x * w.y - y * w.x;
return res;}

void main()
{vecteur v(2,6),u(4,8);
v.affiche();v.homotethie(2);v.affiche();
cout <<"Determinant de (u,v) = "<<v.det(u)<<"\n";
cout <<"Determinant de (v,u) = "<<u.det(v)<<"\n";getch() ;}
```

Exercice III-14:

```
#include <iostream.h>
#include <conio.h>
                // Classe vecteur - Fonction homotethie - Retour par valeur
class vecteur
{float x,y;
public: vecteur(float,float); // Constructeur
        vecteur homotethie(float);
        void affiche();};

vecteur::vecteur(float abs =0,float ord = 0)
{x=abs;y=ord;}

vecteur vecteur::homotethie(float val)
{vecteur res;
res.x = x*val;  res.y = y*val;
return res;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

void main()
{vecteur v(2,6),u;
v.affiche();u.affiche();
u = v.homotethie(4);
```

```
u.affiche();getch() ;}
```

Exercice III-15:

```
#include <iostream.h>
#include <conio.h>
                // Classe vecteur - Fonction homotethie - Retour par adresse
class vecteur
{float x,y;
public: vecteur(float,float); // Constructeur
        vecteur  *homotethie(float);
        void affiche();};

vecteur::vecteur(float abs =0.,float ord = 0.)    // Constructeur
{x=abs;y=ord;}

vecteur *vecteur::homotethie(float val)
{vecteur *res;
res = new vecteur;
res->x = x*val;  res->y = y*val;
return res;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}


void main()
{vecteur v(2,6),u;
v.affiche();u.affiche();
u = *v.homotethie(4);
u.affiche();getch() ;}
```

Exercice III-16:

```
#include <iostream.h>
#include <conio.h>
                        // Classe vecteur -Fonction homotethie - Retour par
reference
class vecteur
{float x,y;
public: vecteur(float,float); // Constructeur
        vecteur &homotethie(float);
        void affiche();};

vecteur::vecteur(float abs =0,float ord = 0)
{x=abs;y=ord;}

vecteur &vecteur::homotethie(float val)
{static vecteur res;
res.x = x*val;  res.y = y*val;
```

```
return res;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

void main()
{vecteur v(2,6),u;
v.affiche();u.affiche();
u = v.homotethie(4);
u.affiche();getch() ;}
```

Exercice III-18:

```
#include <iostream.h>          // objets transmis en argument d'une fonction
membre
#include <conio.h>    // Utilisation du mot clé THIS

class point
{
int x,y;
public:  point(int abs = 0,int ord = 0){x=abs;y=ord;}// constructeur
          int coincide(point *);
};

int point::coincide(point *adpt)
{if ((this->x == adpt->x) && (this->y == adpt->y))
 return(1);else return(0);}

void main()
{point a,b(1),c(1,0);
cout<<"a et b:"<<a.coincide(&b)<<" ou "<<b.coincide(&a)<<"\n";
cout<<"b et c:"<<b.coincide(&c)<<" ou "<<c.coincide(&b)<<"\n";
getch() ;}
```

Exercice III-19:

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

// Creation d'une classe vecteur, avec constructeur, affichage
// Produit scalaire de 2 vecteurs

class vecteur
{float x,y;
public:vecteur(float,float);
vecteur somme(vecteur);
float prod_scal(vecteur);void affiche();};

vecteur::vecteur(float xpos=0,float ypos=0)
{x = xpos; y = ypos;}
```

```
float vecteur::prod_scal(vecteur v)  // tester le passage par reference &v
{float res;
res = (x * v.x) + (y * v.y);
return (res);}

vecteur vecteur::somme(vecteur v) // tester aussi le passage par reference &v
{vecteur res;
res.x = x + v.x;
res.y = y + v.y;
return res;}

void vecteur::affiche()
{cout<<"x= "<<x<<" y= "<<y<<"\n";}

main()
{vecteur a(3);a.affiche();vecteur b(1,2);b.affiche();
vecteur c(4,5),d;c.affiche();
cout<<"b.c = "<<b.prod_scal(c)<<"\n";
cout<<"c.b = "<<c.prod_scal(b)<<"\n";
c = a.somme(b);
d = b.somme(a);
cout<<"Coordonnees de a+b:";c.affiche();cout<<"\n";
cout<<"Coordonnees de b+a:";d.affiche();cout<<"\n";
getch() ;}
```

CHAPITRE 4 (***)

INITIALISATION, CONSTRUCTION, DESTRUCTION DES OBJETS

Dans ce chapitre, on va chercher à mettre en évidence les cas pour lesquels le compilateur cherche à exécuter un constructeur, et quel est ce constructeur et, d'une façon plus générale, on étudiera les mécanismes de construction et de destruction.

I- CONSTRUCTION ET DESTRUCTION DES OBJETS AUTOMATIQUES

Rappel: Une variable locale est appelée encore « automatique », si elle n'est pas précédée du mot « static ». Elle n'est alors pas initialisée et sa portée (ou durée de vie) est limitée au bloc où elle a été déclarée.

Exemple et exercice IV-1:

Exécuter le programme suivant et étudier soigneusement à quel moment sont créés puis détruits les objets déclarés. Noter l'écran d'exécution obtenu.

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

```
class point
{int x,y;
public:  point(int,int);
        ~point();
};

point::point(int abs,int ord)
{x = abs; y = ord;cout<<"Construction du point "<<x<<"  "<<y<<"\n";}

point::~point()
{cout<<"Destruction du point "<<x<<" "<<y<<"\n";}

void test()
{cout<<"Debut de test()\n";point u(3,7);cout<<"Fin de test()\n";}

void main()
{cout<<"Debut de main()\n";point a(1,4);
test();
point b(5,10);
for(int i=0;i<3;i++)point(7+i,12+i);
cout<<"Fin de main()\n";getch() ;}
```

II- CONSTRUCTION ET DESTRUCTION DES OBJETS STATIQUES

Exemple et exercice IV-2: Même étude avec le programme suivant:

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

class point
{int x,y;
public:  point(int,int);
        ~point();
};

point::point(int abs,int ord)
{x = abs; y = ord;cout<<"Construction du point "<<x<<"  "<<y<<"\n";}

point::~point()
{cout<<"Destruction du point "<<x<<" "<<y<<"\n";}

void test()
{cout<<"Debut de test()\n";
static point u(3,7);cout<<"Fin de test()\n";}

void main()
{cout<<"Debut de main()\n";point a(1,4);
test();
point b(5,10);
cout<<"Fin de main()\n";getch() ;}
```

III- CONSTRUCTION ET DESTRUCTION DES OBJETS GLOBAUX

Exemple et exercice IV-3: Même étude avec le programme suivant

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

class point
{int x,y;
public:  point(int,int);
        ~point();
};

point::point(int abs,int ord)
{x = abs; y = ord;cout<<"Construction du point "<<x<<"  "<<y<<"\n";}

point::~point()
{cout<<"Destruction du point "<<x<<" "<<y<<"\n";}

point a(1,4);  // variable globale

void main()
{cout<<"Debut de main()\n";
point b(5,10);
```

cout<<"Fin de main()\n";getch() ;}

IV- CONSTRUCTION ET DESTRUCTION DES OBJETS DYNAMIQUES

Exemple et exercice IV-4: Même étude avec le programme suivant

```
#include <iostream.h>
#include <conio.h>
class point
{int x,y;
public:  point(int,int);
        ~point();
};

point::point(int abs,int ord)
{x = abs; y = ord;cout<<"Construction du point "<<x<<" "<<y<<"\n";}

point::~point()
{cout<<"Destruction du point "<<x<<" "<<y<<"\n";}

void main()
{cout<<"Debut de main()\n";
point *adr;
adr = new point(3,7); // reservation de place en memoire
delete adr;  // liberation de la place
cout<<"Fin de main()\n";getch() ;}
```

Exécuter à nouveau le programme en mettant en commentaires l'instruction « delete adr ».

Donc, dans le cas d'un objet dynamique, le constructeur est exécuté au moment de la réservation de place mémoire (« new »), le destructeur est exécuté lors de la libération de cette place (« delete »).

V- INITIALISATION DES OBJETS

Le langage C autorise l'initialisation des variables dès leur déclaration:
Par exemple: **int i = 2;**

Cette initialisation est possible, et de façon plus large, avec les objets:
Par exemple: **point a(5,6); // constructeur avec arguments par défaut**

Et même: **point b = a;**
Que se passe-t-il alors à la création du point b ? En particulier, quel constructeur est-il exécuté?

Exemple et exercice IV-5: Tester l'exemple suivant, noter l'écran d'exécution obtenu et conclure

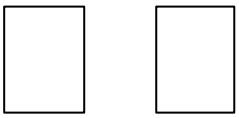
```
#include <iostream.h>
#include <conio.h>
class point
```

```
{int x,y;
public:  point(int,int);
 ~point();};
```

```
point::point(int abs,int ord)
{x = abs; y = ord;cout<<"Construction du point "<<x<<" "<<y;
cout<<" Son adresse: "<<this<<"\n";

point::~point()
{cout<<"Destruction du point "<<x<<" "<<y<<" Son adresse:"<<this<<"\n";}

void main()
{cout<<"Debut de main()\n";
point a(3,7);
point b=a;
cout<<"Fin de main()\n";clrscr() ;}
```

Sont donc exécutés ici:
- le constructeur pour a UNIQUEMENT
- le destructeur pour a ET pour b

Le compilateur affecte correctement des emplacements-mémoire différents pour a et b:



Exemple et exercice IV-6:

Dans l'exemple ci-dessous, la classe **liste** contient un membre privé de type pointeur. Le constructeur lui alloue dynamiquement de la place. Que se passe-t-il lors d'une initialisation de type: **liste a(3);**
liste b = a;

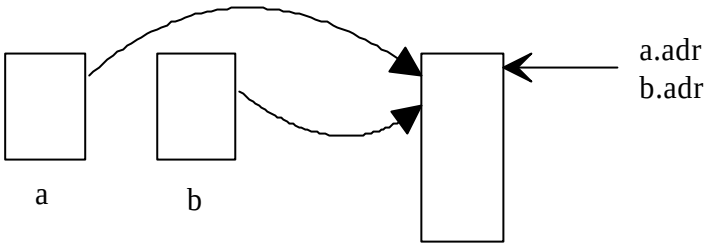
```
#include <iostream.h>
#include <conio.h>

class liste
{int taille;
float *adr;
public:  liste(int);
        ~liste();
};

liste::liste(int t)
{taille = t;adr = new float[taille];cout<<"Construction";
cout<<" Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";}

liste::~liste()
{cout<<"Destruction Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";
```

delete adr;}

```
void main()
{cout<<"Debut de main()\n";
liste a(3);
liste b=a;
cout<<"Fin de main()\n";getch() ;}
```

Comme précédemment, sont exécutés ici:
- le constructeur pour a UNIQUEMENT
- le destructeur pour a ET pour b

Le compilateur affecte des emplacements-mémoire différents pour a et b.
Par contre, les pointeurs **b.adr** et **a.adr** pointent sur la même adresse. La réservation de place dans la mémoire ne s'est pas exécutée correctement:



Exercice IV-7:

Ecrire une fonction membre **void saisie()** permettant de saisir au clavier les composantes d'une liste et une fonction membre **void affiche()** permettant de les afficher sur l'écran. Les mettre en oeuvre dans **void main()** en mettant en évidence le défaut vu dans l'exercice IV-6.

L'étude de ces différents exemples montre que, lorsque le compilateur ne trouve pas de constructeur approprié, il n'en n'exécute pas.

Exemple et exercice IV-8:

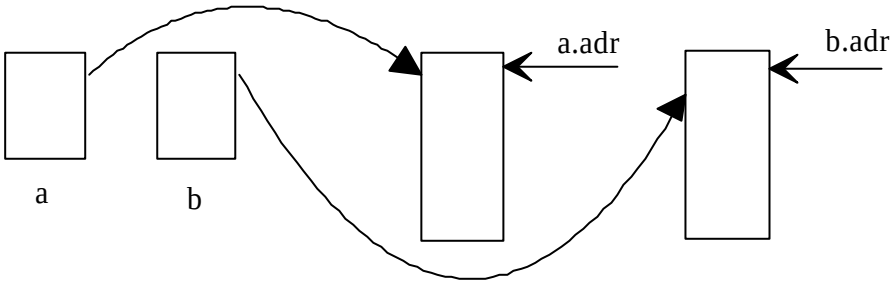
On va maintenant ajouter un constructeur de prototype **liste(liste &)** appelé encore « constructeur par recopie ». Ce constructeur sera appelé lors de l'exécution de **liste b=a;**

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

class liste
{
int taille;
float *adr;
public:  liste(int);
         liste(liste &);
         ~liste();
```

```
};
```

```
liste::liste(int t)
{
taille = t;adr = new float[taille];
cout<<"\nConstruction";
cout<<"  Adresse de l'objet:"<<this;
cout<<"  Adresse de liste:"<<adr<<"\n";
}

liste::liste(liste &v)  // passage par référence obligatoire
{
taille = v.taille;adr = new float[taille];
for(int i=0;i<taille;i++)adr[i] = v.adr[i];
cout<<"\nConstructeur par recopie";
cout<<"  Adresse de l'objet:"<<this;
cout<<"  Adresse de liste:"<<adr<<"\n";
}

liste::~liste()
{cout<<"\nDestruction Adresse de l'objet:"<<this;
cout<<"  Adresse de liste:"<<adr<<"\n";
delete adr;}

void main()
{cout<<"Debut de main()\n";
liste a(3);
liste b=a;
cout<<"\nFin de main()\n";getch() ;}
```

Ici, toutes les réservations de place en mémoire ont été correctement réalisées:



Exercice IV-9:

Reprendre l'exercice IV-7, et montrer qu'avec le « constructeur par recopie », on a résolu le problème rencontré.

VI- CONCLUSION

Il faut prévoir un « constructeur par recopie » lorsque la classe contient des données dynamiques.
Lorsque le compilateur ne trouve pas ce constructeur, aucune erreur n'est générée.

VII - ROLE DU CONSTRUCTEUR LORSQU'UNE FONCTION RETOURNE UN OBJET

On va étudier maintenant une autre application du « constructeur par recopie ».

Exemple et exercice IV-10:

On reprend la fonction membre **point symetrique()** étudiée dans l'exercice III-11.
Cette fonction retourne donc un objet.
Tester le programme suivant et étudier avec précision à quel moment les constructeurs et le destructeur sont exécutés.

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

class point
{
int x,y;
public:  point(int,int);
         // point(point &); // constructeur par recopie
          point symetrique();
       void affiche(){cout<<"x="<<x<<" y="<<y<<"\n";}
       ~point();
};

point::point(int abs=0,int ord=0)
{x = abs; y = ord;cout<<"Construction du point "<<x<<"  "<<y;
cout<<" d'adresse "<<this<<"\n";}

point::point(point &pt)
{x = pt.x; y = pt.y;
cout<<"Construction par recopie du point "<<x<<" "<<y;
cout<<" d'adresse "<<this<<"\n";}

point point::symetrique()
{point res;
cout<<"*****************************************\n";
res.x = -x; res.y = -y;
cout<<"#########################################\n";
return res;}

point::~point()
{cout<<"Destruction du point "<<x<<" "<<y;
cout<<" d'adresse "<<this<<"\n";}

void main()
{cout<<"Debut de main()\n";
point a(1,4), b;
cout<<"Avant appel à symetrique\n";
b = a.symetrique();
```

```
b.affiche();
cout<<"Après appel à symetrique et fin de main()\n";getch() ;}
```

Il y a donc création d'un objet temporaire, au moment de la transmission de la valeur de « res » à « b ». Le constructeur par recopie et le destructeur sont exécutés.
Il faut insister sur le fait que la présence du construteur par recopie n'était pas obligatoire ici: l'exercice III-1 a fonctionné correctement ! et se rappeler ce qui a été mentionné plus haut:

Lorsqu'un constructeur approprié existe, il est exécuté. S'il n'existe pas, aucune erreur n'est générée. Selon le contexte ceci nuira ou non au bon déroulement du programme.
Il faut prévoir un constructeur par recopie lorsque l'objet contient une partie dynamique.

Tester éventuellement le programme IV-10, en supprimant le constructeur par recopie.

Exemple et exercice IV-11:

On a écrit ici, pour la classe **liste** étudiée précédemment, une fonction membre de prototype **liste oppose()** qui retourne la liste de coordonnées opposées.
Exécuter ce programme et conclure.

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

class liste
{int taille;
float *adr;
public:  liste(int);
         liste(liste &);
         void saisie();
         void affiche();
         liste oppose();
         ~liste();
};

liste::liste(int t)
{taille = t;adr = new float[taille];
cout<<"Construction";
cout<<" Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";}

liste::liste(liste &v)  // passage par référence obligatoire
{
taille = v.taille;
adr = new float[taille];
for(int i=0;i<taille;i++)adr[i]=v.adr[i];
cout<<"Constructeur par recopie";
cout<<" Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";
}
```

```
liste::~liste()
{cout<<"Destruction Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";
delete adr;}
```

```
void liste::saisie()
{int i;
for(i=0;i<taille;i++)
{cout<<"Entrer un nombre:";cin>>*(adr+i);}
}

void liste::affiche()
{int i;
for(i=0;i<taille;i++)cout<<*(adr+i)<<" ";
cout<<"adresse de l'objet: "<<this<<" adresse de liste: "<<adr<<"\n";}

liste liste::oppose()
{liste res(taille);
for(int i=0;i<taille;i++)res.adr[i] = - adr[i];
for(i=0;i<taille;i++)cout<<res.adr[i]<<" ";
cout<<"\n";
return res;}

void main()
{cout<<"Debut de main()\n";
liste a(3),b(3);
a.saisie();a.affiche();
b = a.oppose();b.affiche();
cout<<"Fin de main()\n";getch() ;}
```

Solution et exercice IV-12:

On constate donc que l'objet local **res** de la fonction **oppose()** est détruit AVANT que la transmission de valeur ait été faite. Ainsi, la libération de place mémoire a lieu trop tôt.
Ré-écrire la fonction **oppose()** en effectuant le retour par référence (cf chapitre 3) et conclure sur le rôle du retour par référence.

VIII- EXERCICES RECAPITULATIFS

Exercice IV-13:

Ecrire une classe **pile_entier** permettant de gérer une pile d'entiers, selon le modèle ci-dessous.

```
class  pile_entier
{int *pile,taille,hauteur;  // pointeur de pile, taille maximum, hauteur actuelle
public:
pile_entier(int); // constructeur, taille de la pile, 20 par defaut, initialise la
hauteur à 0
                  // alloue dynamiquement de la place mémoire
~pile_entier(); // destructeur
void empile(int);  // ajoute un élement
```

```
int depile(); // retourne la valeur de l'entier en haut de la pile, la hauteur
                        // diminue de 1 unité
int pleine(); // retourne 1 si la pile est pleine, 0 sinon
int vide(); // retourne 1 si la pile est vide, 0 sinon
};
```

Mettre en oeuvre cette classe dans main(). Le programme principal doit contenir les déclarations suivantes:

```
void main()
 { pile_entier a,b(15);   // pile automatique
  pile_entier *adp;        // pile dynamique
```

Exercice IV-14:

Ajouter un constructeur par recopie et le mettre en oeuvre.

IX- LES TABLEAUX D'OBJETS

Les tableaux d'objets se manipulent comme les tableaux classiques du langage C
Avec la classe **point** déjà étudiée on pourra par exemple déclarer:

```
point courbe[100];    // déclaration d'un tableau de 100 points
```

La notation **courbe[i].affiche()** a un sens.
La classe **point** doit OBLIGATOIREMENT posséder un **constructeur** sans argument (ou avec des arguments par défaut). *Le constructeur est exécuté pour chaque élément du tableau.*
La notation suivante est admise:

```
class point
{int x,y;
public:        point(int abs=0,int ord=0)
               {x=abs;y=ord;}
};

void main()
{point courbe[5]={7,4,2};}
```

On obtiendra les résultats suivants:

	x	y
courbe[0]	7	0
courbe[1]	4	0
courbe[2]	2	0
courbe[3]	0	0
courbe[4]	0	0

On pourra de la même façon créer un tableau dynamiquement:

```
point *adcourbe = newpoint[20];
```

et utiliser les notations ci-dessus. Pour détruire ce tableau, on écrira **delete []adcourbe;**
Le destructeur sera alors exécuté pour chaque élément du tableau.

Exercice IV-15:

Reprendre par exemple l'exercice III-8 (classe **vecteur**), et mettre en oeuvre dans main() un tableau de vecteurs.

X - CORRIGE DES EXERCICES

Exercice IV-7:

```
#include <iostream.h>
#include <conio.h>
class liste
{
int taille;
float *adr;
public:  liste(int);
          void saisie();
         void affiche();
         ~liste();
};

liste::liste(int t)
{taille = t;adr = new float[taille];cout<<"Construction";
cout<<" Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";}

liste::~liste()
{cout<<"Destruction Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";
delete adr;}

void liste::saisie()
{
int i;
for(i=0;i<taille;i++)
{cout<<"Entrer un nombre:";cin>>*(adr+i);}
}

void liste::affiche()
{
int i;
for(i=0;i<taille;i++)cout<<*(adr+i)<<" ";
cout<<"\n";
}

void main()
{cout<<"Debut de main()\n";
liste a(3);
liste b=a;
a.saisie();a.affiche();
b.saisie();b.affiche();a.affiche();
cout<<"Fin de main()\n";
getch() ;}
```

Exercice IV-9:

Même programme qu'au IV-7, en ajoutant le « constructeur par recopie » du IV-8.

Exercice IV-12:

```
#include <iostream.h>
#include <conio.h>
class liste
{
int taille;
float *adr;
public:  liste(int);
           liste(liste &);
           void saisie();
           void affiche();
         liste &oppose();
           ~liste();
};

liste::liste(int t)
{taille = t;adr = new float[taille];
cout<<"Construction";
cout<<" Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";}

liste::liste(liste &v)  // passage par référence obligatoire
{
taille = v.taille;
adr = new float[taille];
for(int i=0;i<taille;i++)adr[i]=v.adr[i];
cout<<"Constructeur par recopie";
cout<<" Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";
}

liste::~liste()
{cout<<"Destruction Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";
delete adr;}

void liste::saisie()
{
int i;
for(i=0;i<taille;i++)
{cout<<"Entrer un nombre:";cin>>*(adr+i);}
}

void liste::affiche()
{
int i;
for(i=0;i<taille;i++)cout<<*(adr+i)<<" ";
cout<<"Adresse de l'objet: "<<this<<" Adresse de liste: "<<adr<<"\n";
}
```

```
liste &liste::oppose()
{static liste res(taille);
for(int i=0;i<taille;i++)*(res.adr+i) = - *(adr+i);
for(i=0;i<taille;i++)cout<<*(res.adr+i);
cout<<"\n";
return res;
}

void main()
{cout<<"Debut de main()\n";
liste a(3),b(3);
a.saisie();a.affiche();
b = a.oppose();b.affiche();
cout<<"Fin de main()\n";
getch() ;}
```

Exercice IV-13:

```
#include <iostream.h>   // Gestion d'une pile d'entiers
#include <conio.h>

class  pile_entier
{int *pile,taille,hauteur;
public:
pile_entier(int); // constructeur, taille de la pile
~pile_entier(); // destructeur
void empile(int);  // ajoute un element
int depile();  // depile un element
int pleine(); // 1 si vrai   0 sinon
int vide();  // 1 si vrai  0 sinon
};

pile_entier::pile_entier(int n=20)  // taille par defaut: 20
{taille = n;
pile = new int[taille];  // taille de la pile
hauteur = 0;
cout<<"On a fabrique une pile de "<<taille<<" elements\n";}

pile_entier::~pile_entier()
{delete pile;}  // libere la place

void pile_entier::empile(int p)
{*(pile+hauteur) = p;hauteur++;}

int pile_entier::depile()
{int res; hauteur--; res = *(pile+hauteur); return res;}

int pile_entier::pleine()
{if(hauteur==taille)return 1;else return 0;}
```

```
int pile_entier::vide()
{if(hauteur==0)return 1;else return 0;}


 void main()
 { pile_entier a,b(15);   // pile automatique
   a.empile(8);
 if(a.vide()==1) cout<<"a vide\n";else cout<<"a non vide\n";

 pile_entier *adp;          // pile dynamique
 adp = new pile_entier(5);  // pointeur sur une pile de 5 entiers
 for(int i=0;adp->pleine()!=1;i++) adp->empile(10*i);
 cout<<"\nContenu de la pile dynamique:\n";
 for(int i=0;i<5;i++)if(adp->vide()!=1)cout<<adp->depile()<<"\n";
 getch();
 }
```

Exercice IV-14:

```
 #include <iostream.h>  // constructeur par recopie
#include <conio.h>

class  pile_entier
{int *pile,taille,hauteur;
public:
pile_entier(int); // constructeur, taille de la pile
pile_entier(pile_entier &);  // constructeur par recopie
~pile_entier(); // destructeur
void empile(int);  // ajoute un element
int depile();  // depile un element
int pleine(); // 1 si vrai   0 sinon
int vide();  // 1 si vrai  0 sinon
};

pile_entier::pile_entier(int n=20)  // taille par defaut: 20
{taille = n;
pile = new int[taille];  // taille de la pile
hauteur = 0;
cout<<"On a fabrique une pile de "<<taille<<" elements\n";
cout<<"Adresse de la pile: "<<pile<<" et de l'objet: "<<this<<"\n";}

pile_entier::pile_entier(pile_entier &p)
{taille = p.taille; hauteur = p.hauteur;
pile=new int[taille];
for(int i=0;i<hauteur;i++)*(pile+i) = p.pile[i];
cout<<"On a fabrique une pile de "<<taille<<" elements\n";
cout<<"Adresse de la pile: "<<pile<<" et de l'objet: "<<this<<"\n";}

pile_entier::~pile_entier()
{delete pile;}  // libere la place
```

```
void pile_entier::empile(int p)
{*(pile+hauteur) = p;hauteur++;}

int pile_entier::depile()
{int res;hauteur--;res=*(pile+hauteur);return res;}

int pile_entier::pleine()
{if(hauteur==taille)return 1;else return 0;}

int pile_entier::vide()
{if(hauteur==0)return 1;else return 0;}

void main()
{ cout<<"Pile a:\n";pile_entier a(10);
for(int i=0;a.pleine()!=1;i++)a.empile(2*i);
cout<<"Pile b:\n";
pile_entier b = a;
while(b.vide()!=1)cout<<b.depile()<<"  ";
getch();}
```

Exercice IV-15:

```
#include <iostream.h>
#include <conio.h>
                              // Tableau de vecteurs

class vecteur
{float x,y;
public: vecteur(float,float);
        void homotethie(float);
        void affiche();
  float det(vecteur);
};

vecteur::vecteur(float abs =5.0,float ord = 3.0)
{x=abs;y=ord;}

void vecteur::homotethie(float val)
{x = x*val;  y = y*val;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

float vecteur::det(vecteur w)
{
float res;
res = x * w.y - y * w.x;
return res;
}
```

```
void main()
{vecteur v[4]={17,9},*u;
u = new vecteur[3];  // tableau de 3 vecteurs
for(int i=0;i<4;i++)v[i].affiche();
v[2].homotethie(3);v[2].affiche();
cout <<"Determinant de (u1,v0) = "<<v[0].det(u[1])<<"\n";
cout <<"Determinant de (v2,u2) = "<<u[2].det(v[2])<<"\n";
delete []u;
getch();}
```

CHAPITRE 5

SURDEFINITION DES OPERATEURS

I- INTRODUCTION

Le langage C++ autorise l'utilisateur à étendre la signification d'opérateurs tels que l'addition (+), la soustraction (-), la multiplication (*), la division (/), le ET logique (&) etc...

Exemple:

On reprend la classe vecteur déjà étudiée et on surdéfinit l'opérateur somme (+) qui permettra d'écrire dans un programme:

```
vecteur v1, v2, v3;
v3 = v2 + v1;
```

Exercice V-1:

Etudier et tester le programme suivant:

```
#include <iostream.h>
#include <conio.h>
                              // Classe vecteur
                              // Surdefinition de l'operateur +
class vecteur
{float x,y;
public: vecteur(float,float);
        void affiche();
        vecteur operator + (vecteur);   // surdefinition de l'operateur somme
                                        // on passe un parametre vecteur
                              // la fonction retourne un vecteur
};

vecteur::vecteur(float abs =0,float ord = 0)
```

```
{x=abs;y=ord;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

vecteur vecteur::operator+(vecteur v)
{ vecteur res;
res.x = v.x + x;
res.y = v.y + y;
return res;}
```

```
 void main()
{vecteur a(2,6),b(4,8),c,d,e,f;
c = a + b; c.affiche();
d = a.operator+(b); d.affiche();
e = b.operator+(a); e.affiche();
f = a + b + c; f.affiche();
getch() ;}
```

Exercice V-2:

Ajouter une fonction membre de prototype **float operator*(vecteur)** permettant de créer l'opérateur « produit scalaire », c'est à dire de donner une signification à l'opération suivante:

```
vecteur v1, v2;
float prod_scal;
prod_scal = v1 * v2;
```

Exercice V-3:

Ajouter une fonction membre de prototype **vecteur operator*(float)** permettant de donner une signification au produit d'un réel et d'un vecteur selon le modèle suivant :

```
vecteur v1,v2;
float h;
v2 =  v1 * h ;  // homotethie
```

Les arguments étant de type différent, cette fonction peut cohabiter avec la précédente.

Exercice V-4:

Sans modifier la fonction précédente, essayer l'opération suivante et conclure.

```
vecteur v1,v2;
float h;
v2 =  h * v1;  // homotethie
```

Cette appel conduit à une erreur de compilation. L'opérateur ainsi créé, n'est donc pas symétrique. Il faudrait disposer de la notion de « fonction amie » pour le rendre symétrique.

II- APPLICATION: UTILISATION D'UNE BIBLIOTHEQUE

TURBO C++ possède une classe « complex », dont le prototype est déclaré dans le fichier **complex.h**.

Voici une partie de ce prototype:

```
class complex
{
double re,im;  // partie reelle et imaginaire du nombre complexe

complex(double reel, double imaginaire = 0);  // constructeur

// complex manipulations
   double real(complex);   // retourne la partie réelle
   double  imag(complex);   // retourne la partie imaginaire
   complex conj(complex);  // the complex conjugate
   double norm(complex);   // the square of the magnitude
   double arg(complex);    // the angle in radians

// Create a complex object given polar coordinates
   complex polar(double mag, double angle=0);

// Binary Operator Functions
    complex  operator+(complex);

    friend complex  operator+(double, complex); // donnent une signification aux
deux
    friend complex  operator+(complex  , double); // notations « complex +
double »
                                                  // et «double + complex »
                  // la notion de « fonction amie » sera étudiée lors du prochain
chapitre

    complex  operator-(complex);

    friend complex  operator-(double, complex);     // idem avec la soustraction
    friend complex  operator-(complex  , double);

    complex  operator*(complex);

    friend complex  operator*(complex  , double);   // idem avec la multiplication
    friend complex  operator*(double, complex);

    complex  operator/(complex);

    friend complex  operator/(complex  , double);     // idem avec la division
    friend complex  operator/(double, complex);

    int  operator==(complex);  // retourne 1 si égalité
    int  operator!=(complex  , complex);  // retourne 1 si non égalité
    complex  operator-();  // oppose du vecteur
};

// Complex stream I/O
ostream    operator<<(ostream  , complex);   // permet d'utiliser cout avec un
complexe
```

istream operator>>(istream , complex); // permet d'utiliser cin avec un complexe

Exercice V-5:

Analyser le fichier **complex.h** pour identifier toutes les manipulations possibles avec les nombres complexes en TURBO C++.
Ecrire une application qui utilise notamment cin et cout.

III- REMARQUES GENERALES (***)

- Pratiquement tous les opérateurs peuvent être surdéfinis:
+ - * / = ++ -- new delete [] -> & | ^ && || % << >> etc ...
avec parfois des règles particulières non étudiées ici.
- Il faut se limiter aux opérateurs existants.
- Les règles d'associativité et de priorité sont maintenues.
- Il n'en est pas de même pour la commutativité (cf exercice V-3 et V-4).
- L'opérateur = peut-être redéfini. S'il ne l'est pas, une recopie est exécutée comme on l'a vu dans le chapitre II (cf exercice II-1, à re-tester).
Un risque de dysfonctionnement existe si la classe contient des données dynamiques.

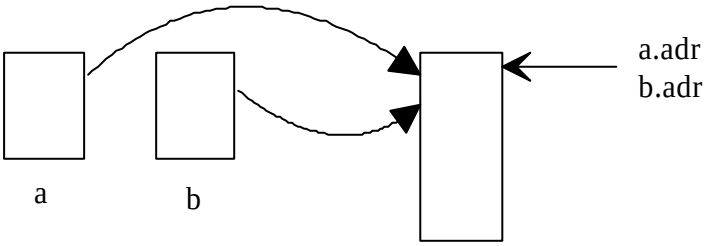
Exercice V-6:

Dans le programme ci-dessous, on surdéfinit l'opérateur =.
En étudier soigneusement la syntaxe, tester avec et sans la surdéfinition de l'opérateur =. Conclure.

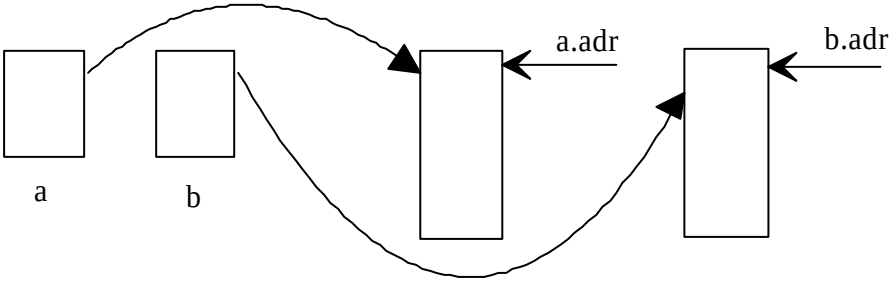
```
#include <iostream.h>
#include <conio.h>
class liste
{int taille;
float *adr;
public:  liste(int);  // constructeur
liste::liste(liste &); // constructeur par recopie
void saisie(); void affiche();
void operator=(liste &); // surdefinition de l'operateur =
~liste();  };

liste::liste(int t)
{taille = t;adr = new float[taille];cout<<"Construction";
cout<<" Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";}

liste::~liste()
{cout<<"Destruction Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";
delete adr;}

liste::liste(liste &v)
{taille = v.taille;adr = new float[taille];
for(int i=0;i<taille;i++)adr[i] = v.adr[i];
cout<<"\nConstructeur par recopie";
cout<<"  Adresse de l'objet:"<<this;
cout<<"  Adresse de liste:"<<adr<<"\n";}
```

```
void liste::saisie()
{int i;
for(i=0;i<taille;i++)
{cout<<"Entrer un nombre:";cin>>*(adr+i);}}

void liste::affiche()
{int i;
cout<<"Adresse:"<<this<<" ";
for(i=0;i<taille;i++)cout<<*(adr+i)<<" ";
cout<<"\n\n";}

void liste::operator=(liste &lis)// passage par reference pour
{int i;                  // eviter l'appel au constructeur par recopie
taille=lis.taille;       // et la double liberation d'un meme
delete adr;              // emplacement memoire
adr=new float[taille];
for(i=0;i<taille;i++)adr[i] = lis.adr[i];}

void main()
{cout<<"Debut de main()\n";
liste a(5);
liste b(2);
a.saisie();a.affiche();
b.saisie();b.affiche();
b=a;
b.affiche();a.affiche();
cout<<"Fin de main()\n";}
```

On constate donc que la surdéfinition de l'opérateur = permet d'éviter la situation suivante:



et conduit à:



Conclusion:
Une classe doit toujours posséder au minimum, un constructeur, un constructeur par recopie, un destructeur, la surdéfinition de l'opérateur =.

IV- EXERCICES RECAPITULATIFS (***)

Exercice V-7:

Reprendre la classe pile_entier de l'exercice IV-13 et remplacer la fonction membre « empile » par l'opérateur < et la fonction membre « depile » par l'opérateur >.
p < n ajoute la valeur n sur la pile p
p> n supprime la valeur du haut de la pile p et la place dans n.

Exercice V-8:

Ajouter à cette classe un constructeur par recopie et la surdéfinition de l'operateur =

Exercice V-9:

Ajouter à la classe **liste** la surdéfinition de l'opérateur [], de sorte que la notation a[i] ait un sens et retourne l'élément d'emplacement i de la liste a.
Utiliser ce nouvel opérateur dans les fonctions **affiche** et **saisie**
On créera donc une fonction membre de prototype ***float &liste::operator[](int i);***

Exercice V-10:

Définir une classe **chaine** permettant de créer et de manipuler une chaîne de caractères:
données:
- longueur de la chaine (entier)
- adresse d'une zône allouée dynamiquement (inutile d'y ranger la constante \0)

méthodes:
- constructeur **chaine()** initialise une chaîne vide
- constructeur **chaine(char *)** initialise avec la chaîne passée en argument
- constructeur par recopie **chaine(chaine &)**

- opérateurs affectation (=), comparaison (==), concaténation (+), accès à un caractère de rang donné ([])

V- CORRIGE DES EXERCICES

Exercice V-2:

```
#include <iostream.h>
#include <conio.h>  // Classe vecteur, surdefinition de l'operateur produit scalaire

class vecteur
{float x,y;
public: vecteur(float,float);
```

```
        void affiche();
        vecteur operator+(vecteur);   // surdefinition de l'operateur +
        float operator*(vecteur);     // surdefinition de l'operateur  produit
scalaire
};

vecteur::vecteur(float abs =0,float ord = 0)
{x=abs;y=ord;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

vecteur vecteur::operator+(vecteur v)
{ vecteur res;
res.x = v.x + x;
res.y = v.y + y;
return res;}

float vecteur::operator*(vecteur v)
{float res;res = v.x * x + v.y * y;return res;}

void main()
{vecteur a(2,6),b(4,8),c;
float prdscl1,prdscl2,prdscl3;
c = a + b;
c.affiche();
prdscl1 = a * b;
prdscl2 = a.operator*(b);
prdscl3 = b.operator*(a);
cout<<prdscl1<<" "<<prdscl2<<" "<<prdscl3<<"\n";
getch() ;}
```

Exercice V-3:

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

class vecteur
{float x,y;
public: vecteur(float,float);
        void affiche();
        vecteur operator+(vecteur);   // surdefinition de l'operateur +
        float operator*(vecteur);     // surdefinition de l'operateur
                                      // produit scalaire
        vecteur operator*(float);     // surdefinition de l'operateur homotethie
 };

vecteur::vecteur(float abs =0,float ord = 0)
{x=abs;y=ord;}
```

```
void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

vecteur vecteur::operator+(vecteur v)
{vecteur res; res.x = v.x + x; res.y = v.y + y; return res;}

float vecteur::operator*(vecteur v)
{float res;res = v.x * x + v.y * y;return res;}

vecteur vecteur::operator*(float f)
{vecteur res; res.x = f*x;   res.y = f*y;  return res;}

void main()
{vecteur a(2,6),b(4,8),c,d;
float prdscl1,h=2.0;
c = a + b; c.affiche();
prdscl1 = a * b;
cout<<prdscl1<<"\n";
d =  a * h; d.affiche();getch() ;}
```

Exercice V-5:

```
#include <iostream.h>          // Utilisation de la bibliotheque
#include <conio.h>             // de manipulation des nombres complexes
#include <complex.h>

#define PI 3.14159

void main()
{complex a(6,6),b(4,8),c(5);
float n =2.0,x,y;
cout<<"a = "<<a<<" b= "<<b<<" c= "<<c<<"\n" ;
c = a + b; cout<<"c= "<<c<<"\n" ;
c = a * b; cout<<"c= "<<c<<"\n" ;
c = n * a; cout<<"c= "<<c<<"\n" ;
c = a * n; cout<<"c= "<<c<<"\n" ;
c = a/b; cout<<"c= "<<c<<"\n" ;
c = a/n; cout<<"c= "<<c<<"\n" ;
x = norm(a); cout<<"x= "<<x<<"\n" ;
y = arg(a)*180/PI;       // Pour l'avoir en degrés
cout<<"y= "<<y<<"\n" ;
c = polar(20,PI/6);  // module = 20  angle = 30°
cout<<"c= "<<c<<"\n" ;
c = -a; cout<<"c= "<<c<<"\n" ;
c = a+n; cout<<"c= "<<c<<"\n" ;
cout<<(c==a)<<"\n" ;
cout<<"Saisir c sous la forme (re,im): ";
cin >> c;
cout<<"c= "<<c<<"\n" ;
getch() ;}
```

Exercice V-7:

```
 #include <iostream.h>   // Gestion d'une pile d'entiers
#include <conio.h>
class  pile_entier
{int *pile,taille,hauteur;
public:
pile_entier(int); // constructeur, taille de la pile
~pile_entier(); // destructeur
void operator < (int);  // ajoute un element
void operator >(int &);  // depile un element
int pleine(); // 1 si vrai   0 sinon
int vide();  // 1 si vrai  0 sinon
};

pile_entier::pile_entier(int n=20)  // taille par defaut: 20
{taille = n;
pile = new int[taille];  // taille de la pile
hauteur = 0;
cout<<"On a fabrique une pile de "<<taille<<" elements\n";}

pile_entier::~pile_entier()
{delete pile;}  // libere la place

void pile_entier::operator<(int x)
{*(pile+hauteur) = x;hauteur++;}

void pile_entier::operator >(int &x) // passage par reference obligatoire
{hauteur--;x = *(pile+hauteur); }    // pour modifier la valeur de l'argument

int pile_entier::pleine()
{if(hauteur==taille)return 1;else return 0;}

int pile_entier::vide()
{if(hauteur==0)return 1;else return 0;}


 void main()
 { pile_entier a ;
 int n = 8,m;
 a < n;
 if (a.vide()) cout<<"La pile est vide\n";
 else cout<<"La pile n'est pas vide\n";
 a > m;
 cout<<"m="<<m<<"\n";
 if (a.vide()) cout<<"La pile est vide\n";
 else cout<<"La pile n'est pas vide\n";
getch();}
```

Exercice V-8: On ajoute les éléments suivants:

```
#include <iostream.h>   // Gestion d'une pile d'entiers
#include <conio.h>
class  pile_entier
{int *pile,taille,hauteur;
public:
pile_entier(int); // constructeur, taille de la pile
pile_entier(pile_entier &); // constructeur par recopie
~pile_entier(); // destructeur
void operator < (int);  // ajoute un element
void operator >(int &);  // depile un element
void operator = (pile_entier &); // surdefinition de l'operateur =
int pleine(); // 1 si vrai   0 sinon
int vide();  // 1 si vrai  0 sinon
};

pile_entier::pile_entier(int n=20)  // taille par defaut: 20
{taille = n;
pile = new int[taille];  // taille de la pile
hauteur = 0;
cout<<"On a fabrique une pile de "<<taille<<" elements\n";}

pile_entier::pile_entier(pile_entier &p)  // constructeur par recopie
{taille = p.taille;
pile =  new int [taille] ;
hauteur =  p.hauteur;
for(int i=0;i<hauteur;i++)*(pile+i) = p.pile[i];
cout<<"On a bien recopie la pile\n"; }

pile_entier::~pile_entier()
{delete pile;}  // libere la place

void pile_entier::operator<(int x)
{*(pile+hauteur) = x;hauteur++;}

void pile_entier::operator >(int &x) // passage par reference obligatoire
{hauteur--;x = *(pile+hauteur); }    // pour modifier la valeur de l'argument

int pile_entier::pleine()
{if(hauteur==taille)return 1;else return 0;}

int pile_entier::vide()
{if(hauteur==0)return 1;else return 0;}


void pile_entier::operator = (pile_entier &p)
{int i;
taille = p.taille;
pile = new int [taille];
```

```
hauteur = p.hauteur;
for(i=0;i<hauteur;i++)*(pile+i)=p.pile[i];
cout<<"l'égalite, ca marche !\n";}


 void main()
 { pile_entier a,c(10); ;
 int i,n,m,r,s ;
 for(n=5;n<22;n++) a < n;   // empile 18 valeurs
 pile_entier b = a;
 for(i=0;i<3;i++){b>m;cout<<m<<" ";}     // depile 3 valeurs
 cout<<"\n";c = a;
 for(i=0;i<13;i++){c>r;cout<<r<<" ";}   // depile 13 valeurs
 cout<<"\n";for (i=0;i<4;i++){a>s;cout<< s<<" ";} //depile 4 valeurs
getch();}
```

Exercice V-9:

```
#include <iostream.h>
#include <conio.h>
class liste   // NOTER LA MODIFICATION DES FONCTIONS
          // saisie ET affiche QUI UTILISENT L'OPERATEUR []
{int taille;
float *adr;
public:  liste(int);
          liste::liste(liste &);
          void operator=(liste &); // surdefinition de l'operateur =
          float &operator[](int); // surdefinition de l'operateur []
          void saisie();
          void affiche();
         ~liste();};

liste::liste(int t)
{taille = t;adr = new float[taille];cout<<"Construction";
cout<<" Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";}

liste::~liste()
{cout<<"Destruction Adresse de l'objet:"<<this;
cout<<" Adresse de liste:"<<adr<<"\n";
delete adr;}

liste::liste(liste &v)
{taille = v.taille;adr = new float[taille];
for(int i=0;i<taille;i++)adr[i] = v.adr[i];
cout<<"\nConstructeur par recopie";
cout<<"  Adresse de l'objet:"<<this;
cout<<"  Adresse de liste:"<<adr<<"\n";}

void liste::operator=(liste &lis)// passage par reference pour
```

```
{int i;                 // eviter l'appel au constructeur par recopie
taille=lis.taille;      // et la double liberation d'un meme
delete adr;             // emplacement memoire
adr=new float[taille];
for(i=0;i<taille;i++)adr[i] = lis.adr[i];}

float &liste::operator[](int i) // surdefinition de []
{return adr[i];}

void liste::saisie()  // UTILISATION DE []
{int i;
for(i=0;i<taille;i++)
{cout<<"Entrer un nombre:";cin>>adr[i];}}

void liste::affiche()     // UTILISATION DE []
{int i;
cout<<"Adresse:"<<this<<" ";
for(i=0;i<taille;i++)cout<<adr[i]<<" ";
cout<<"\n\n";}

void main()
{cout<<"Debut de main()\n";
liste a(3);
a[0]=25;
a[1]=233;
cout<<"Saisir un nombre:";
cin>>a[2];a.affiche();
a.saisie();a.affiche();
cout<<"Fin de main()\n";
getch();}
```

Exercice V-10:

```
 #include <iostream.h>   // classe chaine
#include <conio.h>
class chaine
{int longueur; char *adr;
public:
chaine();chaine(char *);chaine(chaine &);  //constructeurs
~chaine();
void operator=(chaine &);
int operator==(chaine);
chaine &operator+(chaine);
char &operator[](int);
void affiche();};

chaine::chaine(){longueur = 0;adr = new char[1];}  //constructeur1

chaine::chaine(char *texte)   // constructeur2
{int i;
```

```
for(i=0;texte[i]!='\0';i++);
longueur = i;
adr = new char[longueur+1];
for(i=0;i!=(longueur+1);i++) adr[i] = texte[i];}

chaine::chaine(chaine &ch) //constructeur par recopie
{longueur = ch.longueur;
adr = new char[longueur];
for(int i=0;i!=(longueur+1);i++)adr[i] = ch.adr[i];}

void chaine::operator=(chaine &ch)
{  delete adr;
        longueur = ch.longueur;
        adr = new char[ch.longueur+1];
        for(int i=0;i!=(longueur+1);i++)adr[i] = ch.adr[i];
}

int chaine::operator==(chaine ch)
{int i,res=1;
for(i=0;(i!=(longueur+1))&&(res!=0);i++)if(adr[i]!=ch.adr[i])res=0;
return res;}

chaine &chaine::operator+(chaine ch)
{int i;static chaine res;
res.longueur = longueur + ch.longueur;
res.adr = new char[res.longueur+1];
for(i=0;i!=longueur;i++) res.adr[i] = adr[i];
for(i=0;i!=ch.longueur;i++)res.adr[i+longueur] = ch.adr[i];
res.adr[res.longueur]='\0';
return(res);}

char &chaine::operator[](int i)
{static char res='\0';
if(longueur!=0) res =  *(adr+i);
return res;}

chaine::~chaine(){delete adr;}

void chaine::affiche()
{int i;
for(i=0;i!=longueur;i++)cout<<adr[i];
cout<<"\n";}

void main()
{chaine a("Bonjour "),b("Maria"),c,d("Bonjour "),e;

if(a==b)cout<<"Gagne !\n";else cout<<"Perdu !\n";
if(a==d)cout<<"Gagne !\n";else cout<<"Perdu !\n";
cout<<"a: ";a.affiche();
cout<<"b: ";b.affiche();
```

```
cout<<"d: ";d.affiche();

c = a+b;
cout<<"c: ";c.affiche();

for(int i=0;c[i]!='\0';i++)cout<<c[i];
getch();}
```

CHAPITRE 6

FONCTIONS AMIES

Grâce aux fonctions amies, on pourra accéder aux membres privés d'une classe, autrement que par le biais de ses fonctions membres.

Il existe plusieurs situations d'amitié:

- Une fonction indépendante est amie d'une ou de plusieurs classes.
- Une ou plusieurs fonctions membres d'une classe sont amie d'une autre classe.

I- FONCTION INDEPENDANTE AMIE D'UNE CLASSE

Exemple (à tester) et exercice VI-1:

Dans l'exemple ci-dessous, la fonction *coïncide* est AMIE de la classe *point*. C'est une fonction ordinaire qui peut manipuler les membres privés de la classe *point*.

```
#include <iostream.h>          //fonction independante, amie d'une classe
#include <conio.h>
class point
{
int x,y;
public:
point(int abs=0,int ord=0){x=abs;y=ord;}

friend int coincide(point,point);      //declaration de la fonction amie
};

int coincide(point p,point q)
{if((p.x==q.x)&&(p.y==q.y))return 1;else return 0;}

void main()
{point a(4,0),b(4),c;
if(coincide(a,b))cout<<"a coincide avec b\n";
else cout<<"a est different de b\n";
if(coincide(a,c))cout<<"a coincide avec c\n";
else cout<<"a est different de c\n";
getch() ;}
```

Exercice VI-2:

Reprendre l'exercice III-8 dans lequel une fonction membre de la classe **vecteur** permettait de calculer le déterminant de deux vecteurs:
Définir cette fois-ci une fonction indépendante AMIE de la classe vecteur.

II- LES AUTRES SITUATIONS D'AMITIE

1- Dans la situtation ci-dessous, la fonction **fm_de_titi**, fonction membre de la classe TITI, a accès aux membres privés de la classe TOTO:

```
class TOTO
{
// partie privée
.....
// partie publique
friend int TITI::fm_de_titi(char, TOTO);
};

class TITI
{
.....
int fm_de_titi(char, TOTO);
};

int TITI::fm_de_titi(char c, TOTO t)
{ ... } // on pourra trouver ici une invocation des membres privés de l'objet t
```

Si toutes les fonctions membres de la classe TITI étaient amies de la classe TOTO, on déclarerait directement dans la partie publique de la classe TOTO: **friend class TITI;**

2- Dans la situation ci-dessous, la fonction **f_anonyme** a accès aux membres privés des classes TOTO et TITI:

```
class TOTO
{
// partie privée
.....
// partie publique
friend void f_anonyme(TOTO, TITI);
};

class TITI
{
// partie privée
.....
// partie publique
friend void f_anonyme(TOTO, TITI);
};

void f_anonyme(TOTO to, TITI ti)
{ ... } // on pourra trouver ici une invocation des membres privés des objets to et ti.
```

III- APPLICATION A LA SURDEFINITION DES OPERATEURS

Exemple (à tester) et exercice VI-3:

On reprend l'exemple V-1 permettant de surdéfinir l'opérateur + pour l'addition de 2 vecteurs.
On crée, cette fois-ci, une fonction AMIE de la classe **vecteur.**

```
#include <iostream.h>
#include <conio.h>
                                // Classe vecteur
// Surdefinition de l'operateur + par une fonction AMIE
class vecteur
{float x,y;
public: vecteur(float,float);
        void affiche();
        friend vecteur operator+(vecteur, vecteur);
};

vecteur::vecteur(float abs =0,float ord = 0)
{x=abs;y=ord;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

vecteur operator+(vecteur v, vecteur w)
{ vecteur res;
res.x = v.x + w.x;
res.y = v.y + w.y;
return res;}

void main()
{vecteur a(2,6),b(4,8),c,d;
c = a + b; c.affiche();
d = a + b + c; d.affiche();getch() ;}
```

Exercice VI-4:

Reprendre l'exercice VI-1: redéfinir l'opérateur == correspondant à la fonction **coïncide.**

Exercice VI-5:

Reprendre les exercices V-2, V-3 et V-4: En utilisant la propriété de surdéfinition des fonctions du C++, créer
- une fonction membre de la classe **vecteur** de prototype
float vecteur::operator*(vecteur); qui retourne le produit scalaire de 2 vecteurs
- une fonction membre de la classe **vecteur** de prototype
vecteur vecteur::operator*(float); qui retourne le vecteur produit d'un vecteur et d'un réel

(donne une signification à **v2 = v1 * h;**)

- une fonction AMIE de la classe **vecteur** de prototype
vecteur operator*(float, vecteur); qui retourne le vecteur produit d'un réel et d'un vecteur
(donne une signification à **v2 = h * v1;**)

On doit donc pouvoir écrire dans le programme:

```
vecteur v1, v2, v3, v4;
float h, p;
p = v1 * v2;
v3 = h * v1;
v4 = v1 * h;
```

Remarque:
On aurait pu remplacer la fonction membre de prototype **vecteur vecteur::operator*(float);**
par une fonction AMIE de prototype **vecteur operator*(vecteur, float);**

Exercice VI-6:

Etudier le listing du fichier d'en-tête **complex.h** fourni au chapitre V et justifier tous les prototypes des fonctions.

IV- CORRIGE DES EXERCICES

Exercice VI-2:

```
#include <iostream.h>
#include <conio.h>
        // Classe vecteur
        // Fonction AMIE permettant de calculer le déterminant de 2 vecteurs
class vecteur
{float x,y;
public: vecteur(float,float);
        void affiche();
      friend float det(vecteur, vecteur);
};

vecteur::vecteur(float abs =0.,float ord = 0.)
{x=abs;y=ord;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

float det(vecteur a, vecteur b)    // la fonction AMIE peut manipuler
{                  // les quantités b.x, b.y, a.x, a.y
float res;
res = a.x * b.y - a.y * b.x;
return res;
}
```

```
void main()
{vecteur u(2,6),v(4,8);
u.affiche(); v.affiche();
cout <<"Determinant de (u,v) = "<<det(u,v)<<"\n";
cout <<"Determinant de (v,u) = "<<det(v,u)<<"\n";getch() ;}
```

Exercice VI-4:

```
#include <iostream.h>          //Surdéfinition de l'opérateur ==
class point
{
int x,y;
public:
point(int abs=0,int ord=0){x=abs;y=ord;}
friend int operator==(point,point); //declaration de la fonction amie
};

int operator==(point p,point q)
{if((p.x==q.x)&&(p.y==q.y))return 1;else return 0;}

void main()
{
point a(4,0),b(4),c;
if(a==b)cout<<"a coincide avec b\n";
else cout<<"a est different de b\n";
if(a==c)cout<<"a coincide avec c\n";
else cout<<"a est different de c\n";
getch() ;}
```

Exercice VI-5:

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

class vecteur
{float x,y;
public: vecteur(float,float);
        void affiche();
        vecteur operator+(vecteur);   // surdefinition de l'operateur +
        float operator*(vecteur);     // surdefinition de l'operateur
                                      // produit scalaire
        vecteur operator*(float);     // surdefinition de l'homotethie
        friend vecteur operator*(float,vecteur);//surdefinition de l'homotethie
};

vecteur::vecteur(float abs =0,float ord = 0)
{x=abs;y=ord;}
```

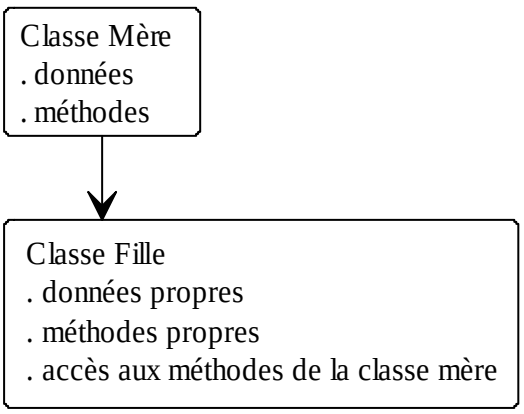
```
void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

vecteur vecteur::operator+(vecteur v)
{vecteur res; res.x = v.x + x; res.y = v.y + y; return res;}

float vecteur::operator*(vecteur v)
{float res;res = v.x * x + v.y * y;return res;}

vecteur vecteur::operator*(float f)
{vecteur res; res.x = f*x;   res.y = f*y;  return res;}

vecteur operator*(float f, vecteur v)
{vecteur res; res.x = f*v.x;   res.y = f*v.y;  return res;}

void main()
{vecteur a(2,6),b(4,8),c,d;
float p,h=2.0;
p = a * b;
cout<<p<<"\n";
c =  h * a;
c.affiche();
d =  a * h;
d.affiche();
getch() ;}
```

CHAPITRE 7

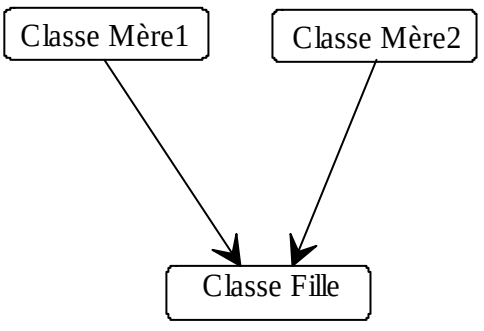
L'HERITAGE

La P.O.O. permet de définir de nouvelles classes (classes filles) dérivées de classes de base (classes mères), avec de nouvelles potentialités. Ceci permettra à l'utilisateur, à partir d'une bibliothèque de classes donnée, de développer ses propres classes munies de fonctionnalités propres à l'application.
On dit qu'une classe fille DERIVE d'une ou de plusieurs classes mères.

<u>Héritage simple:</u>



<u>Héritage multiple:</u>



La classe fille n'a pas accès aux données (privées) de la classe mère.

II- DERIVATION DES FONCTIONS MEMBRES

Exemple (à tester) et exercice VII-1:
L'exemple ci-dessous illustre les mécanismes de base :

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

class vecteur  // classe mère
{float x,y;
public: void initialise(float,float);
        void homotethie(float);
        void affiche();
};

void vecteur::initialise(float abs =0.,float ord = 0.)
{x=abs;y=ord;}

void vecteur::homotethie(float val)
{x = x*val;  y = y*val;}
```

```
void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

class vecteur3:public vecteur // classe fille
{float z;
public:
void initialise3(float,float,float);
void homotethie3(float);
void hauteur(float ha){z = ha;}
void affiche3();  } ;

void vecteur3::initialise3(float abs=0.,float ord=0.,float haut=0.)
{initialise(abs,ord); z = haut;}     // fonction membre de la classe vecteur

void vecteur3::homotethie3(float val)
{homotethie(val);  z = z*val;}   // fonction membre de la classe vecteur

void vecteur3::affiche3()
{affiche();cout<<"z = "<<z<<"\n";} // fonction membre de la classe vecteur

void main()
{vecteur3 v, w;
v.initialise3(5, 4, 3);v.affiche3();  // fonctions de la fille
w.initialise(8,2);  w.hauteur(7); w.affiche();  // fonctions de la mère
cout<<"*********************************\n";
w.affiche3(); w.homotethie3(6);w.affiche3(); // fonctions de la fille
getch() ;
}
```

L'exemple ci-dessus présente une syntaxe assez lourde. Il serait plus simple, pour l'utilisateur, de donner aux fonctions membres de la classe fille, le même nom que dans la classe mère, lorsque celles-ci jouent le même rôle (ici fonctions **initialise** et **homotethie**).
Ceci est possible en utilisant la propriété de ***surdéfinition*** des fonctions membres.

Exemple (à tester) et exercice VII-2:

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

class vecteur  // classe mère
{float x,y;
public: void initialise(float,float);
        void homotethie(float);
        void affiche();
};
void vecteur::initialise(float abs =0.,float ord = 0.)
{x=abs;y=ord;}
```

```
void vecteur::homotethie(float val)
{x = x*val;  y = y*val;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

class vecteur3:public vecteur // classe fille
{float z;
public:
void initialise(float,float,float);
void homotethie(float);
void hauteur(float ha){z = ha;}
void affiche();   } ;

void vecteur3::initialise(float abs=0.,float ord=0.,float haut=0.)
{vecteur::initialise(abs,ord); z = haut;}     // fonction membre de la classe vecteur

void vecteur3::homotethie(float val)
{vecteur::homotethie(val);   // fonction membre de la classe vecteur
z = z*val;}

void vecteur3::affiche()
{vecteur::affiche();    // fonction membre de la classe vecteur
cout<<"z = "<<z<<"\n";}

void main()
{vecteur3 v, w;
v.initialise(5, 4, 3);v.affiche();
w.initialise(8,2); w.hauteur(7);
w.affiche();
cout<<"*******************************\n";
w.affiche();
w.homotethie(6);w.affiche();
getch() ;}
```

Exercice VII-3:

A partir de l'exemple précédent, créer un projet. La classe mère sera considérée comme une bibliothèque. Définir un fichier **mere.h** contenant les lignes suivantes :

```
class vecteur  // classe mère
{float x,y;
public: void initialise(float,float);
        void homotethie(float);
        void affiche();};
```

Le programme utilisateur contiendra la définition de la classe fille, et le programme principal.

Exercice VII-4:

Dans le programme principal précédent, mettre en œuvre des pointeurs de **vecteur.**

Remarque :
L'amitié n'est pas transmissible: une fonction amie de la classe mère ne sera amie que de la classe fille que si elle a été déclarée amie dans la classe fille.

III- DERIVATION DES CONSTRUCTEURS ET DU DESTRUCTEUR

On suppose la situation suivante :

```
class A                          class B : public A
{ ...                            { ...
public :                         public :
A ( ..... ) ; // constructeur    B ( ..... ) ;  // constructeur
~A ( ) ;      // destructeur      ~B( ) ;        // destructeur
.....                            ........
} ;                              } ;
```

Si on déclare un objet B, seront exécutés
- Le constructeur de A, puis le constructeur de B,
- Le destructeur de B, puis le destructeur de A.

Exemple (à tester) et exercice VII-5:

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

class vecteur  // classe mère
{float x,y;
public:        vecteur(); // constructeur
               void affiche();
               ~vecteur();  // destructeur
};

vecteur::vecteur()
{x=1;y=2; cout<<"Constructeur mere\n";}

void vecteur::affiche() {cout<<"x = "<<x<<"  y = "<<y<<"\n";}
vecteur::~vecteur()  {cout<<"Destructeur mere\n";}

class vecteur3:public vecteur // classe fille
{float z;
public:
vecteur3();  // Constructeur
void affiche();
~vecteur3();} ;

vecteur3::vecteur3()
{z = 3;
cout<<"Constructeur fille\n";}
```

```
void vecteur3::affiche()
{vecteur::affiche();
cout<<"z = "<<z<<"\n";}

vecteur3::~vecteur3()
{cout<<"Destructeur fille\n";}

void main()
{vecteur3 v;
v.affiche();
getch();}
```

Lorsque il faut passer des paramètres aux constructeurs, on a la possibilité de spécifier au compilateur vers lequel des 2 constructeurs, les paramètres sont destinés :

Exemple (à tester) et exercice VII-6:
Modifier le programme principal, pour tester les différentes possibilités de passage d'arguments par défaut.

```
#include <iostream.h>
#include <conio.h>
                        // Héritage simple
class vecteur  // classe mère
{float x,y;
public:         vecteur(float,float); // constructeur
                void affiche();
                ~vecteur();  // destructeur
};

vecteur::vecteur(float abs=1, float ord=2)
{x=abs;y=ord;
cout<<"Constructeur mere\n";}
void vecteur::affiche() {cout<<"x = "<<x<<"  y = "<<y<<"\n";}
vecteur::~vecteur()  {cout<<"Destructeur mere\n";}

class vecteur3:public vecteur // classe fille
{float z;
public:
vecteur3(float, float, float);  // Constructeur
void affiche();
~vecteur3();} ;

vecteur3::vecteur3(float abs=3, float ord=4, float haut=5):vecteur(abs,ord)
{z = haut;                      // les 2 1ers paramètres sont
cout<<"Constructeur fille\n";}      // pour le constructeur de la classe mère

void vecteur3::affiche()
{vecteur::affiche();
cout<<"z = "<<z<<"\n";}
```

```
vecteur3::~vecteur3()
{cout<<"Destructeur fille\n";}

void main()
{vecteur u;
vecteur3 v, w(7,8,9);
u.affiche();v.affiche(); w.affiche();
getch();}
```

Cas du constructeur par recopie (***)

Rappel : Le constructeur par recopie est appelé dans 2 cas :
- Initialisation d'un objet par un objet de même type :

```
vecteur a (3,2) ;
vecteur b = a ;
```

- Lorsqu'une fonction retourne un objet par valeur :

```
vecteur a, b ;
b = a.symetrique() ;
```

Dans le cas de l'héritage, on peut définir un constructeur par recopie pour la classe fille, qui appelle le constructeur par recopie de la classe mère.

Exemple (à tester) et exercice VII-7:

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

class vecteur  // classe mère
{float x,y;
public:         vecteur(float,float); // constructeur
                vecteur(vecteur &); // constructeur par recopie
                void affiche();
                ~vecteur();  // destructeur
};

vecteur::vecteur(float abs=1, float ord=2)
{x=abs;y=ord; cout<<"Constructeur mere\n";}

vecteur::vecteur(vecteur &v)
{x=v.x;  y=v.y;
cout<<"Constructeur par recopie mere\n";}

void vecteur::affiche() {cout<<"x = "<<x<<"  y = "<<y<<"\n";}
vecteur::~vecteur()  {cout<<"Destructeur mere\n";}
```

```
class vecteur3:public vecteur // classe fille
{float z;
public:
vecteur3(float, float, float);  // Constructeur
vecteur3(vecteur3 &);  // Constructeur par recopie
void affiche();
~vecteur3();} ;

vecteur3::vecteur3(float abs=3, float ord=4, float haut=5):vecteur(abs,ord)
{z = haut;
cout<<"Constructeur fille\n";}

vecteur3::vecteur3(vecteur3 &v):vecteur(v)  // par recopie
{z = v.z;                    // appel au constructeurpar recopie de
cout<<"Constructeur par recopie fille\n";} // la classe vecteur

void vecteur3::affiche()
{vecteur::affiche();
cout<<"z = "<<z<<"\n";}

vecteur3::~vecteur3()
{cout<<"Destructeur fille\n";}

void main()
{vecteur3 v(5,6,7);
vecteur3 w = v;
v.affiche(); w.affiche();
getch();}
```

IV- LES MEMBRES PROTEGES

On peut donner à certaines données d'une classe mère le statut « protégé ».
Dans ce cas, les fonctions membres, et les fonctions amies de la classe fille auront accès aux données de la classe mère :

```
class vecteur  // classe mère
{
protected:     float x,y;
public:        vecteur(float,float); // constructeur
               vecteur(vecteur &); // constructeur par recopie
               void affiche();
               ~vecteur();  // destructeur
};

void vecteur3::affiche()
{cout<< "x = "<<x<<"  y= "<<y<< "  z = "<<z<<"\n";}
```

La fonction **affiche** de la classe **vecteur3** a accès aux données **x** et **y** de la classe **vecteur**.

Cette possibilité viole le principe d'encapsulation des données, on l'utilise pour simplifier le code généré.

V- EXERCICES RECAPITULATIFS

On dérive la classe **chaîne** de l'exercice V-10:

```
class chaine
{int longueur; char *adr;
public:
chaine();chaine(char *);chaine(chaine &);  //constructeurs
~chaine();
void operator=(chaine &);
int operator==(chaine);
chaine &operator+(chaine);
char &operator[](int);
void affiche();};
```

La classe dérivée se nomme **chaine_T**.

```
class chaine_T :public chaine
{int Type ;
float Val ;
public :
.......} ;
```

Type prendra 2 valeurs : 0 ou 1.
1 si la chaîne désigne un nombre, par exemple « 456 » ou « 456.8 », exploitable par **atof** la valeur retournée sera Val.
0 dans les autres cas, par exemple « BONJOUR » ou « XFLR6 ».

Exercice VII-8: Prévoir pour chaine_T
- un constructeur de prototype **chaine_T() ;** qui initialise les 3 nombres à 0.
- un constructeur de prototype **chaine_T(char *) ;** qui initialise les 3 nombres à 0 ainsi que la chaîne de caractères.
- une fonction membre de prototype **void affiche()** qui appelle la fonction **affiche** de **chaine** et qui affiche les valeurs des 3 nombres.

Exercice VII-9 (***): Prévoir un constructeur par recopie pour chaine_T , qui initialise les 3 nombres à 0.

Exercice VII-10: Déclarer « protected » la donnée **adr** de la classe **chaîne.** Ecrire une fonction membre pour chaine_T de prototype **void calcul()** qui donne les bonnes valeurs à Type, Val.

VI- CONVERSIONS DE TYPE ENTRE CLASSES MERE ET FILLE

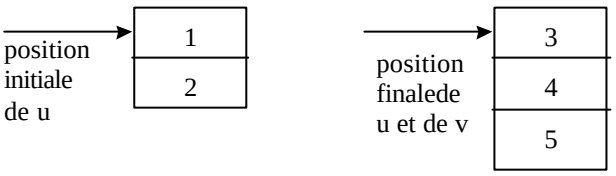
Règle : La conversion d'un objet de la classe fille en un objet de la classe mère est implicite.
La conversion d'un objet de la classe mère en un objet de la classe fille est interdite.

Autrement dit :
vecteur u ;
vecteur3 v ;
u = v ; **// est autorisée, il y a conversion fictive de v en un vecteur**
v = u; **// est interdite**

Exercice VII-11 : Tester ceci avec le programme VII-6

Avec des pointeurs :
vecteur *u ;
u = new vecteur ; // réservation de place, le constructeur de vecteur est exécuté
vecteur3 *v ;
v = new vecteur3 ; // réservation de place, le constructeur de vecteur3 est exécuté
u = v ; **// est autorisée, u vient pointer sur la même adresse que v**
v = u; **// est interdite**
delete u ;
delete v ;

On obtient la configuration mémoire suivante :



Exemple (à tester) et exercice VII-12 :
Reprendre l'exemple VII-11 avec le programme principal suivant :

void main()
{vecteur3 *u;
u = new vecteur3; u->affiche();

vecteur *v;
v = new vecteur; v->affiche();
v = u; v->affiche();
delete v ; delete u ;
getch();}

Conclusion : quelle est la fonction **affiche** exécutée lors du 2ème appel à **v->affiche()** ?
Le compilateur C++ a-t-il « compris » que v ne pointait plus sur un **vecteur** mais sur un **vecteur3** ?
Grâce à la notion de fonction virtuelle on pourra, pendant l'exécution du programme, tenir compte du type de l'objet pointé, indépendamment de la déclaration initiale.

VI- SURDEFINITION DE L'OPERATEUR D'AFFECTATION (***)

Rappels :
- Le C++ définit l'opérateur = par défaut.
- Il est souhaitable de le surdéfinir via une fonction membre, lorsque la classe contient des données de type pointeur.

A est la classe mère, B est la classe fille, on exécute les instructions suivantes :

```
B x, y ;
y = x ;
```

Dans ce cas :
a) Si ni A, ni B n'ont surdéfini l'opérateur = , le mécanisme d'affectation par défaut est mis en œuvre.
b) Si = est surdéfini dans A mais pas dans B, la surdéfinition est mise en œuvre pour les données A de B, le mécanisme d'affectation par défaut est mis en œuvre pour les données propres à B.
c) Si = est surdéfini dans B, cette surdéfinition doit prendre en charge la TOTALITE des données (celles de A et celles de B).

Exemple (à tester) et exercice VII-13 :

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

class vecteur
{float x,y;
public: vecteur(float,float);
void affiche();
void operator=(vecteur &);   // surdefinition de l'operateur =
};

vecteur::vecteur(float abs=1, float ord=2)
{x=abs;y=ord;
cout<<"Constructeur mere\n";}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}

void vecteur::operator=(vecteur &v)
{cout<<"operateur egalite mere\n"; x = v.x; y = v.y;}

class vecteur3:public vecteur // classe fille
{float z;
public:
vecteur3(float, float, float);  // Constructeur
void operator=(vecteur3 &);   // surdefinition de l'operateur =
void affiche();} ;
```

```
void vecteur3::operator=(vecteur3 &v)
{cout<<"operateur egalite fille\n";
vecteur *u, *w;
u = this;
w = &v;
*u = *w;
z = v.z;}

vecteur3::vecteur3(float abs=3, float ord=4, float haut=5):vecteur(abs,ord)
{z = haut;                          // les 2 1ers paramètres sont
cout<<"Constructeur fille\n";}      // pour le constructeur de la classe mère

void vecteur3::affiche()
{vecteur::affiche();
cout<<"z = "<<z<<"\n";}

void main()
{vecteur3 v1(6,7,8),v2;
v2 = v1;
v2.affiche();
getch();
}
```

VII- LES FONCTIONS VIRTUELLES

Les fonctions membres **de la classe mère** peuvent être déclarées ***virtual***.
Dans ce cas, on résout le problème invoqué dans le §V.

Exemple (à tester) et exercice VII-14 :

Reprendre l'exercice VII-12 en déclarant la fonction **affiche**, **virtual** :

```
class vecteur  // classe mère
{float x,y;
public:
vecteur(float,float); // constructeur
virtual void affiche();
~vecteur();  // destructeur
};
```

Quelle est la fonction **affiche** exécutée lors du 2ème appel à **v->affiche()** ?
Le programme a-t-il « compris » que v ne pointait plus sur un **vecteur** mais sur un **vecteur3** ?

Ici, le choix de la fonction à exécuter ne s'est pas fait lors de la compilation, mais ***dynamiquement*** lors de l'exécution du programme.

Exemple (à tester) et exercice VII-15 :

Modifier les 2 classes de l'exercice précédent comme ci-dessous :

```
class vecteur  // classe mère
{float x,y;
public:
vecteur(float,float); // constructeur
virtual void affiche();
void message()  {cout<<"Message du vecteur\n";}
  ~vecteur();  // destructeur
};

void vecteur::affiche()
{message();
cout<<"x = "<<x<<"  y = "<<y<<"\n";}

class vecteur3:public vecteur // classe fille
{float z;
public:
vecteur3(float, float, float);  // Constructeur
void affiche();
void message(){cout<<"Message du vecteur3\n";}
~vecteur3();} ;

void vecteur3::affiche()
{message();
vecteur::affiche();
cout<<"z = "<<z<<"\n";}
```

Remarques :
- Un constructeur ne peut pas être virtuel,
- Un destructeur peut-être virtuel,
- La déclaration d'une fonction membre virtuelle dans la classe mère, sera comprise par toutes les classes descendantes (sur toutes les générations).

Exercice VII-16 :

Expérimenter le principe des fonctions virtuelles avec le destructeur de la classe **vecteur** et conclure.

VIII- CORRIGE DES EXERCICES

Exercice VII-3: Le projet se nomme exvii_3, et contient les fichiers exvii_3.cpp et mere.cpp ou bien mere.obj.

Fichier exvii_3.cpp:

```
#include <iostream.h>
#include <conio.h>
#include "c:\bc5\cours_cpp\teach_cp\chap7\mere.h"

// Construction d'un projet à partir d'une classe mère disponible

class vecteur3:public vecteur // classe fille
{float z;
public:
void initialise(float,float,float);
void homotethie(float);
void hauteur(float ha){z = ha;}
void affiche();   } ;

void vecteur3::initialise(float abs=0.,float ord=0.,float haut=0.)
{vecteur::initialise(abs,ord); z = haut;}     // fonction membre de la classe vecteur

void vecteur3::homotethie(float val)
{vecteur::homotethie(val);   // fonction membre de la classe vecteur
z = z*val;}

void vecteur3::affiche()
{vecteur::affiche();    // fonction membre de la classe vecteur
cout<<"z = "<<z<<"\n";}

void main()
{vecteur3 v, w;
v.initialise(5, 4, 3);v.affiche();
w.initialise(8,2); w.hauteur(7);
w.affiche();
cout<<"*******************************\n";
w.affiche();
w.homotethie(6);w.affiche();}
```

Fichier mere.cpp:

```
#include <iostream.h>
#include <conio.h>
#include "c:\bc5\cours_cpp\teach_cp\chap7\mere.h"

void vecteur::initialise(float abs =0.,float ord = 0.)
{x=abs;y=ord;}

void vecteur::homotethie(float val)
{x = x*val;  y = y*val;}

void vecteur::affiche()
{cout<<"x = "<<x<<"  y = "<<y<<"\n";}
```

Exercice VII-4 :

Programme principal :

```
void main()
{vecteur3 v, *w;
w = new vecteur3;
v.initialise(5, 4, 3);v.affiche();
w->initialise(8,2);w->hauteur(7);
w->affiche();
cout<<"*******************************\n";
w->affiche();
w->homotethie(6);w->affiche();
delete w;}
```

Exercice VII-8 : Seuls la classe **chaine_T** et le programme principal sont listés

```
class chaine_T :public chaine
{int Type;
float Val ;
public :
chaine_T(); // constructeurs
chaine_T(char *);
void affiche();
};

chaine_T::chaine_T():chaine()      // constructeurs
{Type=0;Val=0;}        // dans les 2 cas le constructeur
                       // correspondant de chaine est appelé
chaine_T::chaine_T(char *texte):chaine(texte)
{Type=0;Val=0;}

void chaine_T::affiche()
{chaine::affiche();
cout<<"Type= "<<Type<<"  Val= "<<Val<<"\n";}

void main()
{chaine a("Bonjour ");
chaine_T b("Coucou "),c;
cout<<"a: ";a.affiche();
cout<<"b: ";b.affiche();
cout<<"c: ";c.affiche();
getch();}
```

Exercice VII-9 : Seules les modifications ont été listées

```
class chaine_T :public chaine
{int Type ;
float Val ;
public :
chaine_T(); // constructeurs
```

```
chaine_T(char *);
chaine_T(chaine_T &ch); // constructeur par recopie
void affiche();
};
```

puis :

```
chaine_T::chaine_T(chaine_T &ch):chaine(ch) //constructeur par recopie
{Type=0;Val=0;}       // il appelle le constructeur
                                    // par recopie de chaine

void main()
{chaine_T b("Coucou ");
chaine_T c = b;
cout<<"b: ";b.affiche();
cout<<"c: ";c.affiche();
getch();}
```

Exercice VII-10 : Seules les modifications ont été listées

```
void chaine_T::calcul()
{
Val = atof(adr);    // possible car donnée "protected"
if(Val!=0)Type = 1;
}
```

puis :

```
void main()
{chaine_T b("Coucou "),c("123"), d("45.9"), e("XFLR6");
b.calcul();c.calcul();d.calcul();e.calcul();
b.affiche();c.affiche();d.affiche();e.affiche();
getch();}
```

Exercice VII-11 : Seules les modifications ont été listées

```
void main()
{vecteur u;
vecteur3 v(7,8,9);
u.affiche();v.affiche();
u=v; u.affiche();
getch();}
```