Université Hassan II de Casablanca
ENSET de Mohammedia
Département Mathématiques et Informatique
Filière d'Ingénieurs, Génie du Logiciel et des Systèmes Informatiques Distribués - GLSID

COURS

PROGRAMMATION ORIENTEE OBJET EN LANGAGE C++

Par :
M. Khalifa MANSOURI

Université Hassan II de Casablanca
ENSET de Mohammedia
Département Mathématiques et Informatique

COURS

CONCEPTION ET PROGRAMMATION ORIENTEE OBJET EN LANGAGE C++

Par :

M. Khalifa MANSOURI

SOMMAIRE

CHAPITRE 1
C++ COMME UN LANGAGE C AVANCE
PRINCIPAUX APPORTS

2.1. LES COMMENTAIRES

En plus des symboles /* et */ utilisés en C, le langage C++ offre les symboles // qui permettent d'ignorer tout jusqu'à la fin de la ligne.

Exemples :

```
/* commentaire traditionnel sur plusieurs lignes valide en C et
C++  */
// commentaire de fin de ligne valide en C++
// Tout ce qui est situé entre ces deux symboles
// et la fin de ligne est un commentaire
```

Nota : Il est préférable d'utiliser les symboles // pour la plupart des commentaires et de n'utiliser les commentaires C (/* */) que pour isoler des blocs importants d'instructions.

2.2. EMPLACEMENT DES DECLARATIONS

En C ++ , il n'est plus obligatoire de regrouper au début toutes les déclarations effectuées au sein d'une fonction ou d'un bloc.

Les exemples suivants ne sont pas corrects en C, mais ils sont acceptés en C++.

Exemple 1 :

```
Void Main ()
{
   int n ;
   ..... ;
   n = n + 1 ;
   ..... ;
   int m = 2*n + 1 ;
   ..... ;
}
```

La déclaration tardive de m permet de l'initialiser avec une expression dont la valeur n'était pas connue lors de l'entrée dans la fonction main.

Exemple 2 :

```
Void main()
{
      ..... ;
      For(int i = 1 ; .... ; ....)
      {
      ..... ;
      }
      ..... ;
}
```

L'exemple peut être écrit autrement :

```
Void Main ()
{
      ..... ;
      int i ;
      For(i = 1 ; .... ; ....)
      {
      ..... ;
      }
      ..... ;
}
```

L'exemple n'est pas équivalent au programme suivant :

```
Void Main ()
{
      ..... ;
      int i ;
      For(i = 1 ; .... ; ....)
      {
            int j ;
            ..... ;
      }
      ..... ;
}
```

Dans lequel, on déclare une variable j dont la portée est limitée au bloc de l'instruction For.

2.3.LES NOUVELLES POSSIBILITES D'ENTREES/SORTIES CONVENTIONNELLES : CIN, COUT

2.3.1. Introduction

Les entrées / sorties en langage C s'effectuent par les fonctions *(printf, scanf, puts, gets, putc, getc, ...)* de la librairie standard stdio.h.

Il est possible d'utiliser ces fonctions pour effectuer les entrées / sorties dans les programmes, mais il est préférable en C++ d'utiliser les entrées/sorties par flux (ou flot ou stream).

Les flots prédéfinis lorsqu'on inclue le fichier d'en-tête « iostream.h » sont :

- ***cout* qui correspond à la sortie standard**
- ***cin* qui correspond à l'entrée standard**

Ces nouvelles possibilités ne nécessitent pas de FORMATAGE des données.

L'opérateur (surchargé) << permet d'envoyer des valeurs dans un flot de sortie, tandis que >> permet d'extraire des valeurs d'un flot d'entrée.

2.3.2. Ecriture de données sur la sortie standard

2.3.2.1. Quelques exemples

Exemple 1 :

```
# include <iostream.h>
void main( )
{
      cout << " Langage C++ " ;
}
```

Dans ce programme, << est un opérateur dont l'opérande de gauche (Cout) est un flot et l'opérande de droite est une expression de type quelconque. On peut interpréter cette instruction comme suit : le flot Cout reçoit la valeur " Langage C++ "

Exemple 2 :

```
# include <iostream.h>
void main( )
{
      float Pi = 3.14 ;
      cout << " La valeur de Pi est : " ;
      cout << Pi ;
}
```

Dans cet exemple :

- On a utilisé l'opérateur << pour envoyer deux valeurs de types différents sur le flot cout : une chaîne de caractère puis un réel ;
- Avec le deuxième Cout, on procéder à la conversion d'un réel en une suite de caractères ;
- Les deux Cout de l'exemple 2 peuvent se combiner en un seul :

```
cout << " La valeur de Pi est : " << Pi ;
```

  Dans un premier temps le flot cout reçoit la chaîne : " La valeur de Pi est : " ,
  puis dans un second cas, le flot : cout << " La valeur de Pi est : " reçoit la valeur Pi.
  On peut dire que le flot cout augmenté de la chaîne " La valeur de Pi est : " a reçu la valeur de Pi.
- On peut avoir plusieurs combinaisons à la fois : Cout << << << << << <<

2.3.3. Lecture de données sur l'entrée standard

2.3.3.2. Exemple

```
# include <iostream.h>
Void main( )
{
      int N ;
      cout << « Entrer un nombre entier : » ;
      Cin >> N ;
      Cout << « Le carré du nombre entré est : » << N*N;
}
```

Dans ce programme,

- On a inclus une directive appelée iostream.h ;
- Cin : désigne un flot d'entrée prédéfini ;
- >> est un opérateur permettant de recevoir la donnée en provenance d'un flot d'entrée.

2.3.3.3.Différentes possibilités de lecture sur Cin

On peut utiliser l'opérateur << pour accéder aux données de tous les types standards : caractère, entier, réel, chaîne de caractères, …

Mais on ne peut pas lire de pointeur sur Cin.

2.3.3.4.Remarques

- Tout comme pour la fonction *scanf*, les espaces sont considérés comme des séparateurs entre les données par le flux *cin*.
- On note l'absence de l'opérateur & dans la syntaxe du *cin*. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire.

2.4.LES CONVERSIONS EXPLICITES

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

Exemple 1 :

```
double d;
int i;

i = (int) d;
```

Exemple 2 :

```
#include <iostream.h>
#include <conio.h>
void main()
{
        char c='m',d=25,e;
        int i=42,j;
        float r=678.9,s;
        j = c;
        cout << j << "\n"; // j vaut 109
        j = r;
        cout << j<< "\n"; // j vaut 678
        s = d;
        cout << s<< "\n"; // s vaut 25.0
        e = i;
        cout << e<< "\n"; // e vaut *
        getch();
}
```

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

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

2.5.VISIBILITE DES VARIABLES

L'opérateur de résolution de portée « *::* » permet d'accéder aux variables globales plutôt qu'aux variables locales.

Exemple :

```
#include <iostream.h>
int i = 11;
void main()
{
   int i = 34;
   {
        int i = 23;
        ::i = ::i + 1;
        cout << ::i << "  " << i << endl;
   }
    cout << ::i << "  " << i << endl;
}
```

/*-- résultat de l'exécution ----------
12 23
12 34
----------*/

2.6.LES FONCTIONS

2.6.1. Déclaration des fonctions

Une fonction doit être déclarée avant son utilisation. Cette déclaration doit comporter le nombre et le type des arguments à utiliser.

La déclaration suivante ***int f1();*** dont laquelle la liste des arguments est vide est interprétée comme : ***int f1(void);***

Une fonction dont le type de la valeur retournée n'est pas `void`, doit **obligatoirement** retourner une valeur.

2.6.2. Valeur par défaut des paramètres

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 1 :

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

void f2(int n, float x = 2.35)
{
     ...;
}

void f3(char c, int n = 3, float x = 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 : 3
     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
}
```

Nota :

Les paramètres prenant des valeurs par défaut sont OBLIGATOIREMENT placés en fin de liste.

2.6.3. Surcharge (surdefinition) des fonctions ('overloading')

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.

Exemples :

```
int somme( int n1, int n2)
{
     return n1 + n2;
}
int somme( int n1, int n2, int n3)
{
     return n1 + n2 + n3;
}

double somme( double n1, double n2)
{
     return n1 + n2;
}
void main()
{
     cout << "1 + 2 = " << somme(1, 2) << endl;
     cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
     cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;
}
```

Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction. Ce choix se fait lors de la compilation.

Exercice :

Ecrire trois fonctions portant le même nom et permettant d'afficher respectivement : un entier, un réel, un complexe.

Prévoir une fonction principale permettant d'appeler ces fonctions.

```
struct complexe
{
        double reel,im ;
}
void affiche(int);
void affiche(double);
void affiche(complexe);

int main(void)
{
        int a = 5;
        double d = 0.0;
        complexe c = { 1.0,-1.0 };
        affiche(a);  // Appel la fonction (1)
        affiche(b);  // Appel la fonction (2)
        affiche(c);  // Appel la fonction (3)
}

void affiche(int i)
{
```



```
          cout << "TYPE de variable (int) :" << endl;
          cout << "Valeur : " << i << endl;
}

void affiche(double d)
{
        cout << "TYPE de variable (double) :" << endl;
        cout << "Valeur : " << d << endl;
}

void affiche(complexe c)
{
          cout << "TYPE de variable (complexe) :" << endl;
          cout << "Valeur : " << c.reel << endl;
          cout << "Valeur : " << c.im << endl;
}
```

2.7.ALLOCATION MEMOIRE

Le C++ met à la disposition du programmeur deux opérateurs *new* et *delete* pour remplacer respectivement les fonctions *malloc* et *free* (bien qu'il soit toujours possible de les utiliser).

2.7.1. L'opérateur *new*

L'opérateur **new** réserve l'espace mémoire qu'on lui demande et l'initialise. Il retourne l'adresse de début de la zone mémoire allouée.

Exemple :

```
int *pi;      // déclaration du pointeur
pi = new int; // allocation de la mémoire
```

On aurait pu écrire :

```
int *pi = new int;
```

Ceci peut se faire en langage C de la manière suivante :

```
pi = (int *)malloc(sizeof(int));
```

Autres exemples :

```
int *ptr1, *ptr2, *ptr3;

// allocation dynamique d'un entier
ptr1 = new int;

// allocation d'un tableau de 10 entiers
ptr2 = new int [10];

// allocation d'un entier avec initialisation
ptr3 = new int(10);

struct date {int jour, mois, an; };
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};

// allocation dynamique d'une structure
ptr4 = new date;

// allocation dynamique d'un tableau de structure
```

```
ptr5 = new date[10];

// allocation dynamique d'une structure avec initialisation
ptr6 = new date(d);
```

L'allocation des tableaux à plusieurs dimensions est possible :

```
typedef char TAB[80]; // TAB est un synonyme de : tableau de 80
                         caractères
TAB *ecran;

ecran = new TAB[25]; // ecran est un tableau de 25 fois 80
                        caractères
ecran[24][79]='$';
ou
char (*ecran)[80] = new char[25][80];

ecran[24][79]='$';
```

L'opérateur *delete*

Un objet créé par new ne décède pas à la sortie du bloc (ou de la fonction) ou il a été créé. L'opérateur **`delete`** libère l'espace mémoire alloué par *new* à un seul objet, tandis que l'opérateur **`delete[]`** libère l'espace mémoire alloué à un tableau d'objets.

Exemple 1:

```
delete pi;   // desalloue la zone adressée par pi
             // pi existe encore mais pas pi*
char *pc = new char[100];

delete pc;      // desalloue la zone de 100 caractères
delete [100]pc; // instruction équivalente
```

L'opérateur new retourne le pointeur NULL (0) en cas d'échec d'allocation, il est prudent de le tester.

```
struct complexe { double reel,im };
complexe *Z;

Z = new complexe[50];
...
delete Z; // ne libère que le premier élément
delete [50]Z;
 OU
delete []Z;
```

Remarques :

A chaque instruction `new` doit correspondre une instruction `delete`.
Il est important de libérer l'espace mémoire dès que celui ci n'est plus nécessaire.
La mémoire allouée en cours de programme sera libérée automatiquement à la fin du programme.

Tout ce qui est alloué avec **`new[]`**, doit être libéré avec **`delete[]`**

CHAPITRE 2

PROGRAMMATION ORIENTEE OBJET : NOTION DE CALSSE

3.1. INTRODUCTION

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*. Une 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 programmes.

3.2. RAPPEL SUR LA NOTION DE PROTOTYPE DE FONCTION

En C++, comme en C, on a fréquemment besoin de déclarer des *prototypes* de fonctions.
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»
```

3.3. LES STRUCTURES EN C++

3.3.1. Les structures en langage C

En langage C, la déclaration suivante :


```
Struct point
{
    int x ;
    int y ;
};
```

- Définit un type structuré nommé point,
- x et y sont les champs de la structure point,
- on déclare des variables de type point de la manière suivante : struct point a, b ;
- a.x désigne le champs y de la structure a.

3.3.1.1.Déclaration d'une structure comportant des fonctions membre

Nous souhaitons associer à la structure *point* trois fonctions.

Initialise : pour donner des valeurs aux coordonnées d'un point,
Deplace : pour modifier les coordonnées d'un point,
Affiche : pour afficher un point.

La structure *point* va se déclarer alors de la manière suivante :

```
Struct point
{
    int x ;
    int y ;
    Void initialise(int, int) ;
    void Deplace(int, int) ;
    void Affiche() ;
};
```

3.3.1.2. Définition d'une fonction membre

Une fonction membre est définie de la manière suivante :

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

Le symbole :: est l'opérateur de résolution de portée, il signifie que l'identificateur Initialise est celui définit dans point.

abs désigne la valeur reçue en premier argument. Mais x n'est ni un argument ni une variable locale. Elle désigne le membre x correspondant au type point (cet association étant réalisée par le symbole ::)

pour les autres fonctions :

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

```
Void point :: affiche()
{
    cout << « Je suis en « x «  » y « \n » ;
};
```

3.3.1.3.Utilisation d'une structure comportant des fonctions membres

Déclarations : point a, b ;

L'accès au champs pourrait se déroulé comme en langage C :

```
a.x = 5 ;
a.y = 2 ;
```

L'appel d'une fonction membre est fait d'une manière semblable :

```
a.initialise(5,2) ;
```

3.4. NOTION DE CLASSE

3.4.1. Définition d'un objet

Un ***objet*** est un ensemble de **données** et sur lesquelles des ***méthodes*** (procédures ou fonctions) peuvent être appliquées.

3.4.2. Définition d'une classe

Une *classe* constitue une sorte de ***type*** qui définit la structure des données et des méthodes d'un ensemble d'objets similaires.
Les objets d'une classe en sont des ***instances.***
A chaque instanciation, une allocation de mémoire est faite pour les données du nouvel objet créé. L'initialisation de l'objet nouvellement créé est faite par une méthode spéciale, le ***constructeur.*** Lorsque l'objet est détruit, une autre méthode est appelée : le ***destructeur.***

3.4.3. Déclaration d'une classe

La structure en C++ est un cas particulier de la classe.
Une classe sera une structure dans laquelle seulement certains membres et/ou fonctions seront « **publiques** », càd accessibles à l'extérieur.

Pour déclarer une classe, il suffit de replacer le mot réservé ***struct*** par ***class*** et procéder de la manière suivante :

```
class Nomclasse
{
   private    : // partie accessible uniquement aux fonctions
                // membres de la classe et aux fonctions amies

   protected : // partie accessible aux membres et amies de la
               // classe ainsi qu'aux classes dérivées

   public :    // accessible à tout utilisateur d'une instance
               // de la classe
} ;
```


Une classe est composée de trois niveaux de portée (encapsulation) : ***Private*** (option par défaut), ***protected*** ou ***public.***

Les fonctions membres d'une classe sont définies de la façon suivante :

```
TypeFonction NomCIasse :: NomFonction(ListParamètre)
{
    // Déclarations
    // Actions
}
```

Avec **::** est appelé l'opérateur de résolution de portée.

Exemple de déclaration d'une classe :

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

class point
{
    int x,y, couleur;
    public:
       void initialiser(int,int,int);
       void deplacer(int,int);
       void afficher();
       void effacer() ;
};

void point::initialiser(int abs,int ord, int c)
{
    x = abs;
    y = ord;
    couleur = c ;
}

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

void point::afficher()
{
    textcolor(couleur) ;
    gotoxy(x,y);
    cprintf<<"*";
}

void point::effacer()
{
    int aux = couleur ;
    couleur = black ;
    afficher() ;
    couleur = aux ;
}
```

3.4.4. Utilisation d'une classe

3.4.4.1.Instanciation

La déclaration d'une instance (objet) d 'une classe donnée se fait de la manière suivante :

```
NomClasse NomObjet ;        //Variable Objet
NomClasse *PointeurObjet    //Pointeur sur un objet
```

3.4.4.2.Accès aux membres d'une classe

Pour un instance x d'une classe X, l'appel d'une fonction f membre de la classe se fait de la façon suivante :

```
x.f ( … )
```

Si p est un pointeur sur une instance d'une classe X. l'appel d'une fonction f membre de la classe se fait de la façon suivante :

```
x->f ( … )
```

3.4.4.3.Exemple

```
main()
{
    point p1,*p2;

    p1.initialiser(30,15,MAGENTA);
    p1.afficher();
    getch();

    p1.effacer();
    p1.deplacer(10,5);
    getch();

    p1.effacer();
    p2=new point;
    p2->initialiser(15,20,WHITE);
    p2->afficher();
    delete p2;
    getch() ;
}
```


3.4.4.4.Commentaires

- «point» est une ***classe***,
- Elle est constituée des ***données*** x et y et des ***fonctions membres*** (***ou méthodes***) « initialiser », « deplacer », « afficher » et « effacer »,
- 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.

3.4.5. Notion de constructeur

Un constructeur est une fonction membre ***sans type*** (même le type void) qui porte *le même* nom que la classe, et qui ne retourne pas de valeur. Il est *systématiquement exécuté* lors de création d'une instance de la classe (déclaration d'un objet).

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

3.4.5.1.Exemple

```
#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;
}              // grâce au constructeur

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

void point::affiche()
{
```

```
    gotoxy(x,y);
    gotoxy(x,y);
    cprintf<<"*";
}

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

Exercice :

Reprendre l'exemple ci dessus en utilisant un constructeur à deux paramètres.

```
#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;
}             // grâce 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<<"*" ;
}

void main()
{
    point a(20,10),b(30,20);  // les deux points sont initialises
                              // a en 20,10  b en 30,20
    a.affiche();
    getch();
    a.deplace(17,10);
```



```
    a.affiche();
    getch();
    clrscr();
    b.affiche();
    getch();
}
```

3.4.6. Notion de destructeur

Le destructeur est une fonction membre *systématiquement exécutée* «à la fin de la vie » d'une instance (objet).

Un destructeur porte le même nom que la classe précédé d'un tilde (~), et n'a pas de paramètres ni de valeurs retournée.

Un destructeur permet de prévoir toute action devant s'exécuter avant la destruction d'une instance (Enregistrement de données, fermetures de fichiers, … etc.)

3.4.6.1.Exemple :

```
#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<<"*" ;
}

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

```
void test()
{
    point u(3,7);
    u.affiche();
    getch();
}

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

3.4.7. 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 alors que le destructeur est utilisé pour libérer la place.

Exercice 1 :

Ecrire une classe nommée ***SuiteAr***, dans laquelle le constructeur calcule les premiers (nb) termes d'une suite arithmétique de raison (Nul) qu'il range dans un tableau membre donné val. Le destructeur libère l'espace mémoire réservé pour le membre (Val).

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

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

SuiteAr:: SuiteAr (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;
}

SuiteAr::~ SuiteAr()
{
    delete val;
}  // abandon de la place réservée
```



```
void SuiteAr::affiche()
{
   int i;

   for(i=0;i<nbval;i++)
      cout<<val[i]<<" ";
   cout<<"\n";
}

void main()
{
   clrscr();
   SuiteAr suite1(10,4); //Calcul les 10 premiers termes de la
                         // suite arithmétique de raison 4
   suite1.affiche();
   getch();
   SuiteAr suite2(6,8); //Calcul les 6 premiers termes de la
                        // suite arithmétique de raison 8

   suite2.affiche();
   getch() ;
}
```

Exercice 2 :

Ecrire une classe nommée ***hasard***, dans laquelle le constructeur fabrique dix valeurs entières aléatoires qu'il range dans un tableau membre donné val. Ces valeurs sont prises entre zéro et la valeur qui lui est fourni en argument)

```
#include <iostream.h>        // destructeur
#include <conio.h>
#include <stdlib.h>
class hasard
{
    int val[10];
    public:
        hasard(int);
        void affiche();
};
hasard::hasard(int max)
{
    int i;
    for(i=0;i<10;i++)
       val[i] = double(rand())/RAND_MAX*max;
}
void hasard::affiche()
{
   int i;
   for(i=0;i<10;i++)
       cout<<val[i]<<"    ";
   cout<<"\n";
}
void main()
{
   hasard suite1(5);
   suite1.affiche();
   getch() ;
```

```
    hasard suite2(12);
    suite2.affiche();
    getch() ;
}
```

On désire disposer d'une classe nommée hasard, dans laquelle le nombre de valeurs peut être fourni en argument du constructeur.
Prévoir une allocation dynamique de val pour qu'il s'adapte automatiquement au nombre de valeurs voulu.
Naturellement cette allocation dynamique doit être faite par le constructeur lui même.

```
class hasard
{
    int nbval;
    int *val;
    public:
        hasard(int);
        void affiche();
};
```

L'espace qui a été alloué dynamiquement, doit être libéré lorsqu'il sera devenu inutile.

Ecrire un destructeur permettant de libérer l'espace alloué.

```
#include <iostream.h>        // destructeur
#include <conio.h>
#include <stdlib.h>
class hasard
{
    int nbval;
    int *val;
    public:
        hasard(int, int);
        ~hasard();
        void affiche();
};

hasard::hasard(int nb,int max)
{
    int i;
    val = new int[nbval = nb];
    for(i=0;i<nb;i++)
       val[i] = double(rand())/RAND_MAX*max;

}

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



```
void main()
{
   hasard suite1(15,5);
   suite1.affiche();
   getch() ;
   hasard suite2(6,12);
   suite2.affiche();
   getch() ;
}
```

3.4.8. Membres statiques

Lorsqu'on déclare deux objets différents d'une même classe dans un même programme, chacun des deux possède ses propres membres données.

Exemple :

```
class Exempe1
{
    int x;
    float y ;
};
```

Déclarons deux objets a et b de la classe exemple1 :

```
Exempe1 a, b;
```

Suite à cette déclaration :

l'emplacement réservé pour le membre x de l'objet a est différent de l'emplacement réservé pour le membre x de l'objet b,

de même, l'emplacement réservé pour le membre y de l'objet a est différent de l'emplacement réservé pour le membre y de l'objet b.

Pour permettre à tous les objets d'une classe de partager les mêmes données, il suffit de déclarer avec le qualificatif ***static*** tous les membres donnée que l'on désire voir en un seul exemplaire pour tous les objets de la classe.

Exemple :

```
class Exempe2
{
    static int x;
    float y ;
};
```

Déclarons deux objets a et b de la classe exemple2 :

```
Exempe2 a, b;
```

Suite à cette déclaration :

l'emplacement réservé pour le membre x de la classe est le même pour tous les objets de la classe. Donc a.x = b.x

mais, l'emplacement réservé pour le membre y de l'objet a est différent de l'emplacement réservé pour le membre y de l'objet b.

Exercice :

Créer une classe *compteur* permettant à tous moment de connaître le nombre d'objets existants. Pour se faire, nous allons déclarer statique un membre de la classe appelé *ctr*. Sa valeur est incrémentée de 1 à chaque appel du constructeur et décrémentée de 1 à chaque appel du destructeur.

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

class Compteur
{
    static int ctr = 0;
    public :
       Compteur();
       ~Compteur();
};

Compteur :: Compteur()
{
    cout<< «Un nouvel objet vient de se créer : » << « \n » ;
    cout<< «Il y a maintenant : » << ++ctr << « Objets » ;
    getch() ;
}

Compteur :: ~Compteur()
{
    cout<< «Un objet vient de se détruire : » << « \n » ;
    cout<< «Il reste maintenant : » << --ctr << « Objets » ;
    getch() ;
}

main()
{
    void Essai() ;

    Compteur a ;
    Essai() ;
    Compteur b ;
}

void Essai()
{
    Compteur u,v;
}
```


L'exécution du programme donne :

```
Un nouvel objet vient de se créer :
Il y a maintenant : 1 Objets

Un nouvel objet vient de se créer :
Il y a maintenant : 2 Objets

Un nouvel objet vient de se créer :
Il y a maintenant : 3 Objets

Un nouvel objet vient de se détruire :
Il y a maintenant : 2 Objets

Un nouvel objet vient de se détruire :
Il y a maintenant : 1 Objets

Un nouvel objet vient de se détruire :
Il y a maintenant : 2 Objets

Un nouvel objet vient de se détruire :
Il y a maintenant : 1 Objets

Un nouvel objet vient de se détruire :
Il y a maintenant : 0 Objets
```

CHAPITRE 3 :

PROPRIETES DES FONCTIONS MEMBRES

4.1. 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 : 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();
       void affiche(char *);
};

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<<"*" ;
}
```



```
void point::affiche(char *message)
{
    cout<< message ;
    affiche() ;
}

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

4.2.FONCTIONS MEMBRES EN LIGNE

En langage C++, on peut définir les fonctions membres lors de leurs déclarations dans la classe. On dit que l'on écrit une fonction « ***inline*** ». Celle-ci se présente comme une « macrofonction » c'est à dire à chaque appel, il y a génération du code de la fonction et non pas un appel à un sous-programme.

Les appels sont donc plus rapides mais cette méthode génère plus de code dans le programme.

Exemple :

```
#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<<"*"<<"\n";
}

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

NOTA : Comparer la taille du fichier objet « *.obj* » de cet exemple et celui de l'exemple précédent. Conclure ?

4.3.INITIALISATION DES PARAMETRES PAR DEFAUT

Exemple :

```
#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
défaut
};

void point::affiche(char *message)
{
   gotoxy(x,y-1);
   cout<<message;
   gotoxy(x,y);
   cout<<"le point est 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() ;
}
```


4.4.OBJETS TRANSMIS EN ARGUMENT D'UNE FONCTION MEMBRE

Une fonction membre peut recevoir un ou plusieurs arguments du type de sa classe.

Exemple :

Reprenons la classe point dans laquelle nous allons introduire une fonction membre nommée « *coïncidence* ». Cette fonction permet de détecter la coïncidence éventuelle entre deux points.

```
#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 dissymétrie 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() ;
}
```

L'exécution donne :

```
a et b : 0 ou 0
a et c : 1 ou 1
```

NOTA :

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

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.

Exercice 1 : Passage de paramètres par adresse :

a- Modifier la fonction membre « coïncidence » de l'exemple précédent de sorte que son prototype devienne int point:: coïncidence (point *adpt),
b- Ré-écrire le programme principal en conséquence.

Exercice 2 : Passage de paramètres par référence :

a- Modifier la fonction membre « ***coïncidence*** » de l'exemple précédent de sorte que son prototype devienne ***int point:: coïncidence (point &adpt),***
b- Ré-écrire le programme principal en conséquence.

Exercice 3 : Classe vecteur :

Soit une classe vecteur définie de la manière suivante :

```
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";
}
```

a- Mettre cette classe en en oeuvre dans un programme principal ***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),
b- Modifier la fonction déterminant de sorte de passer le paramètre par adresse.
c- Modifier la fonction **déterminant** de sorte de passer le paramètre par référence.

4.5.OBJETS RETOURNE PAR UNE FONCTION MEMBRE

On va voir ce qui se passe lorsqu'une fonction membre retourne elle-même un objet.

4.5.1. Retour par valeur :

Exemple : 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<<"Le point est en "<<x<<" et "<<" "<<y<<"\n";
}

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

L'exécution donne :

```
Le point est en -1  et  -6
Le point est en 1  et  6
```

4.5.2. Retour par adresse :

Exemple :

```
#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<<"Le point est en "<<x<<" et "<<y<<"\n";
}

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

L'exécution donne :

Le point est en -1 et -6
Le point est en 1 et 6

4.5.3. Retour par référence :

Exemple :

```
#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<<"Le point est en "<<x<<" et "<<y<<"\n";
}

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

L'exécution donne :

Le point est en -1 et -6
Le point est en 1 et 6

Remarque:

L'objet « ***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.

Exercice : Classe vecteur :

Reprenons la classe *vecteur* définie ci dessus.

a- Modifier la fonction « homothétie» qui retourne le vecteur modifié par valeur.
(Prototype: **vecteur vecteur::homotethie(float val)**).
b- Modifier la fonction « homothétie» qui retourne le vecteur modifié par adresse.
c- Modifier la fonction « homothétie» qui retourne le vecteur modifié par reference.

4.6.FONCTIONS MEMBRES STATIQUES :

De la même manière que le langage C++ permet d'utiliser des données membres statiques, il permet d'utiliser des fonctions membres statiques.

Exemple :

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

class compte_objet
{
   static int ctr ;
   public:
     compte_objet() ;
     ~compte_objet() ;
     static void compte() ;
};

compte_objet :: compte_objet()
{
    cout << « ++construction : il y a maintenant » << ++ctr
         << « objets\n » ;
}

compte_objet :: ~compte_objet()
{
    cout << « --desruction : il y a maintenant » << --ctr
         << « objets\n » ;
}

compte_objet :: compte()
{
    cout << « Appel compte : il y a » << ctr << « objets\n » ;
}

void fonction()
{
    compte_objet u,v ;
}

void main()
{
    void fonction()
    compte_objet :: compte() ;
    compte_objet a ;
    compte_objet :: compte() ;
    fonction() ;
    compte_objet :: compte() ;
    compte_objet b ;
    compte_objet :: compte() ;
}
```

4.7.LE MOT CLE « THIS »

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

A chaque appel d'une fonction membre , le compilateur passe implicitement un pointeur sur les données de l'objet en paramètre. Ce pointeur sur l'objet accessible à l'intérieur de la fonction membre porte le nom **this**. This est un pointeur constant, c'est à dire qu'on ne peut pas le modifier.

Exemple :

```
class point
{
    ...
    void affiche()
    {
       textcolor(couleur) ;
       gotoxy(x,y);
       cprintf(« * »);
    }
    ...
};
```

Dans la fonction « **affiche** », les membres x et y désignent ceux de l'objet à travers lequel la fonction est appelée. Cette fonction peut donc s'écrire de manière équivalente :

```
void affiche()
{
    textcolor(this -> couleur) ;
    gotoxy(this ->x, this ->y);
    cprintf(« * »);
}
```

Cette utilisation du pointeur this set inutile. Par contre, il arrive que dans une fonction membre d'un objet on doive faire référence à l'objet tout entier à travers lequel la fonction est appelée :

```
class point
{
    ...
       bool memePoint(point *pt)
       {
          return pt ==this
       }
    ...
};
```

Il est possible de transformer le pointeur constant **this** en un pointeur sur **les données constantes** pour chaque fonction membre (la fonction membre s'interdit la modification des données de l'objet). Ceci s'obtient en ajoutant le mot clé **const** à la suite de l'entête de la fonction membre.

Exemple :

```
class point
{
    float x, y;
    int couleur ;
    public :
      vois affiche(void) const;
};
vois affiche() const
{
    textcolor(couleur);
    gotoxy(x,y);
    cprintf(« * ») ;
}
```

Exercice 1 :

Analyser le programme suivant :

```
#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 2 :

Dans l'exercice qu'on a traité auparavant et utilisant la fonction « **coincide** », remplacer cette fonction par la fonction suivante:


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

Exercice 3 :

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.

CHAPITRE 5 :
CONSTRUCTION ET DESTRUCTION ET INITIALISATION DES OBJETS

5.1. CONSTRUCTION ET DESTRUCTION DES OBJETS AUTOMATIQUES

Une variable locale est aussi appelée « 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.

Les objets automatiques sont ceux crées par une déclaration.

- Dans une fonction, il est détruit à la fin de l'execution de la fonction,
- Dans un bloc, il est détruit lors de la sortie du bloc.

En ce qui concerne la chronologie :

- Le constructeur est appelé après la création de l'objet,
- Le destructeur est appelé avant la destruction de l'objet.

Exemple :

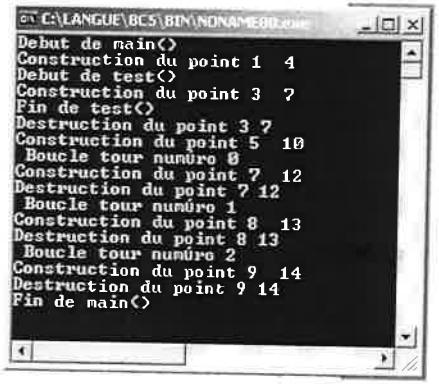
L'objectif de cet exemple et d'étudier soigneusement à quel moment sont créés puis détruits les objets déclarés.

```
#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++)
    {
       cout << " Boucle tour numéro " << I << "\n";
       point(7+i,12+i);
    }
    cout<<"Fin de main()\n";
    getch() ;
}
```

L'exécution donne :



5.2.CONSTRUCTION ET DESTRUCTION DES OBJETS STATIQUES

Les objets statiques sont ceux crées par une déclaration située.

- En dehors de toute fonction,
- Dans une fonction, mais assortie du qualificatif ***sratic***.

Remarque :

Les objets statiques sont crées avant le début de l'exécution de la fonction main() et ils sont détruits après la fin de son exécution.

Exemple :

L'objectif de cet exemple et d'étudier soigneusement à quel moment sont créés puis détruits les objets déclarés.

```
#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() ;
}
```

5.3.CONSTRUCTION ET DESTRUCTION DES OBJETS GLOBAUX

Exemple :

L'objectif de cet exemple et d'étudier soigneusement à quel moment sont créés puis détruits les objets déclarés.

```
#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() ;
}
```

5.4.CONSTRUCTION ET DESTRUCTION DES OBJETS TEMPORAIRES

Si **a** est un objet de type **point**, on peut écrire l'affectation : ***a = point(1,2) ;*** dont laquelle l'évaluation de l'expression ***point(1,2)*** conduit à :

- La déclaration d'un objet temporaire de type **point**,
- L'appel du constructeur point, pour cet objet temporaire, avec transmission des arguments spécifiés,
- La recopie de cet objet temporaire dans a.

Exemple :

```
#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
       << «  à l'adresse : «  << this <<"\n";
}


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

```
void main()
{
   cout<<"Debut de main()\n";
   point a(0,0) ;
   a = point(1,2) ;
   a =point(3,5) ;
   cout<<"Fin de main()\n";
   getch() ;
}
```

5.5.CONSTRUCTION ET DESTRUCTION DES OBJETS DYNAMIQUES

La déclaration d'un objet dynamique se fait de la manière suivante :

```
Point *adr ;
```

A partir de là, nous pouvons créer dynamiquement un emplacement de type ***point*** et affecter son adresse à ***adr*** par :

```
adr = new point;
```

L'accès aux fonctions membres de l'objet pointé par ***adr*** se fera des appels de la forme suivante :

```
adr -> initialise(1,3);
adr -> affiche();
```

L'objet peut être supprimé de la manière suivante :

```
Delette adr ;
```

Remarque :

Si la classe possède un constructeur, l'opérateur new appellera un constructeur de l'objet. Ce constructeur sera déterminé par la nature des arguments qui figurent comme paramètres :

```
adr = new point(2, 5);
```

Exemple :

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

void fct(point *adp)
{
    cout<<"Debut de la fonction\n";
    delete adp;  // liberation de la place
    cout<<"Fin de la fonction\n";
}
```

- Exécutons ce programme une première fois,
- Reéxecutons à nouveau ce programme en mettant en commentaire l'instruction « delete adp »,
- Conclure.

5.6.INITIALISATION DES OBJETS

En langage C, on peut initialiser une variable lors de sa déclaration :

```
Int i = 1;
```

Que se passe-t-il alors à la création du point b ? En particulier, quel constructeur est-il exécuté?

```
Point a(5, 6); // constructeur avec arguments par défaut
```

On peut aussi procéder ainsi :

```
Point b = a;
```

Que se passe-t-il alors à la création du point b ? En particulier, quel constructeur est-il exécuté?

Exemple :

```
#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() ;
}
```

Dans ce programme, le constructeur est exécuté pour a uniquement, alors que le destructeur est exécuté pour a et b.

Le compilateur affecte correctement des emplacements mémoire différents de a et b.

Exemple 1 :

On considère une classe ***liste*** contenant un membre privé de type pointeur. Le constructeur doit lui allouer dynamiquement de la place mémoire. On va chercher ce qui se passe lors des deux initialisations suivantes :

```
Liste a(3);
Liste b = a;
```


Programme complet :

```
#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, le constructeur est exécuté pour a uniquement, alors que le destructeur est exécuté pour a et 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:

Exemple 2 :

Dans cet exemple, on va ajouter un constructeur de prototype ***liste(liste &)*** appelé aussi « **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() ;
}
```

Dans ce cas, toutes les réservations de place en mémoire ont été correctement réalisées.

Entre les deux exemples précédents, on remarque qu'avec le « constructeur par recopie », on a arrive à résoudre le problème de la même adresse pour les deux pointeurs.

En cas général, 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.

5.7.ROLE DU CONSTRUCTEUR LORSQU'UNE FONCTION RETOURNE UN OBJET

Exemple 1 :

Reprenons la fonction membre ***point symetrique()*** déjà étudiée dans chapitre 4. Cette fonction retourne un objet.

On va manipuler 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.

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.

Exemple 2 :

Reprenons la classe ***liste*** étudiée précédemment dans laquelle on va écrire une fonction membre de prototype ***liste oppose()*** qui retourne la liste de coordonnées opposées.

```
#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() ;
}
```

5.8.LES TABLEAUX D'OBJETS

Les tableaux d'objets se manipulent comme les tableaux classiques du langage C

Reprenons la classe ***point*** déjà étudiée, nous pouvons déclarer un tableau de 50 objets de type point par :

```
point courbe[50]; // déclaration d'un tableau de 50 points
```

Si i est un entier, la notation « `courbe[i]` », désignera un objet de type point.

L'instruction « `courbe[i].affiche()` ; », appellera la fonction membre ***affiche*** pour le point ***courbe[i]*** .

Le programme suivant affiche tous les points de la courbe :

```
For(i=0 ; i<50 ; i++)
    Courbe[i].affiche() ;
```

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};
}
```

Ce programme conduit aux 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[50];
```

Pour détruire ce tableau, on écrira : `delete []adcourbe;`

Le destructeur sera alors exécuté pour chaque élément du tableau.

5.9.OBJETS MEMBRES OU OBJETS D'OBJETS

5.9.1. Introduction :

Un membre d'une classe peut tout à fait être lui même de type classe.

Exemple :

On définit une classe ***point*** :


```
class point
{
   int x,y;
   public:
      point init(int ,int ) ;
      void affiche() ;
};
```

Nous pouvons définir ensuite une autre classe ***poincol*** de la manière suivante :

```
class poincol
{
   point p ;
   int couleur ;
   public:
      void affcoul() ;
};
```

Et on déclare un nouveau objet a :

```
poincol a;
```

Dans ce cas, l'objet a possède un membre donné p, de type point.

L'accès à la méthode ***affcol*** se fait par ***a.affcol( )***.

L'accès aux méthodes de la classe ***point*** se fait par ***a.p.init( )*** ou ***a.p.affiche( )***.

5.9.2. Mise en œuvre des constructeurs et de destructeurs :

On considère que la classe ***point*** est définie avec un constructeur :

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

Il faut donc :

- d'une part, définir un constructeur pour la classe ***poincol***.
- d'autre part, spécifier les arguments à fournir au constructeur de ***point*** : ceux-ci doivent être choisis obligatoirement parmi ceux fournis à ***poincol***.

La calsse ***poincol*** et son constructeur vont être définis comme suit :

```
class poincol
{
   point p ;
   int couleur ;
   public:
      poincol(int, int, int) ;
};

poincol ::poincol(int abs, int ord, int coul) :p(abs,ord)
{
}
```

On constate que l'en-tête de ***poincol*** spécifie, après les deux points, la liste des arguments qui seront transmis à ***point***.

Les constructeurs seront appelés dans l'ordre suivant : ***point***, ***poincol***.

S'il existe des destructeurs, ils seront appelés dans l'ordre inverse.

Exemple :

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

class point
{
    int x,y;
    public:
      point(int abs =0,int ord = 0);
      {
          x = abs ;
          y = ord ;
          cout<<"Constructeur point "<<x<<" "<<y<<"\n";
      }
};

class poincol
{
   point p ;
   int couleur ;
   public:
      poincol(int, int, int) ;
};

poincol ::poincol(int abs, int ord, int coul) :p(abs,ord)
{
   couleur = coul ;
   cout<<"Constructeur poincol "<<couleur<<"\n";
}

void main()
{
    poincol  a(1, 3, 9)
}
```

CHAPITRE 6 :

SURDEFINITION DES OPERTATEURS

6.1. DEFINITION

La surdefinition des opérateurs est une technique qui nous permet de créer par le biais des classes, des types à part entière, c'est à dire des types munis, comme les types de base, d'opérateurs parfaitement intégrés.

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

6.2.LE MECANIQME DE SURDEFINITION D'OPERATEURS

Considérons la classe point vu précédemment :

```
class point
{
   int x, y ;
   . . .
   . . .
   . . .
};
```

Soit a et b deux objets de calsse ***point***, nous souhaitons définir un opérateur + pour donner une signification à l'expression ***a + b***. la convention adoptée par le langage C++ pour surdefinir cet opérateur + consiste à définir une fonction de nom : ***operator+***.

La fonction ***operator+*** doit disposer de deux arguments de type ***point*** et fournir une valeur résultat de même type.

Cette fonction peut être une fonction membre ou une fonction indépendante.

Exemple :

Reprenons 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;
```

Etudions 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 1 :

1- 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;
```

2- 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.

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

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

Cet 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 », que nous allons étudier par la suite, pour le rendre symétrique.

6.3.APPLICATION : UTILISATION DANS 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
```

6.4.LES POSSIBILITES ET LIMITES DE LA SURDEFINITION D'OPERATEURS EN C++

6.4.1. Il faut se limiter aux opérateurs existants :

Le symbole suivant le mot clé ***operator*** doit obligatoirement être un opérateur déjà défini pour les types de base. Il n'est donc pas possible de créer de nouveaux symboles.

Lorsque plusieurs opérateurs sont combinés au sein d'une même expression, ils conservent leur priorité relative et leur associativité.

6.4.2. Il faut se limiter au contexte de la classe :

On peut surdéfinir un opérateur que s'il comporte au moins un argument de type classe. Autrement dit, il doit s'agir :

- Soit d'une fonction membre, dans ce cas, elle comporte un argument de type classe, à savoir l'objet l'ayant appelé,
- Soit d'une fonction indépendante ayant au moins un argument de type classe. En général il s'agira d'une fonction amie.

6.4.3. Remarques générales :

- 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.
- L'opérateur = peut-être redéfini. S'il ne l'est pas, une recopie est exécutée.

Un risque de dysfonctionnement existe si la classe contient des données dynamiques.

Exemple :

Dans le programme ci-dessous, on surdéfinit l'opérateur =.
On va étudier soigneusement la syntaxe, et tester avec et sans la surdéfinition de l'opérateur = pour 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 même
    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'avoir a.adr et b.adr deux champs occupant deus endroits dans la mémoire au lieu d'un seul.

Exercice 1 :

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 2 :

Définir une classe ***chaîne*** permettant de créer et de manipuler une chaîne de caractères:

Données:

- longueur de la chaîne (entier)
- adresse d'une zone 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 ([]).

CHAPITRE 7 :
FONCTIONS AMIES

7.1. DEFINITION

Une fonction amie est une fonction avec laquelle on pourra accéder aux membres privés d'une classe, d'une autre manière que l'accès à l'aide des fonctions membres.

On peut citer plusieurs situations d'amitié :

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

7.2.FONCTION INDEPENDANTE AMIE D'UNE CLASSE

Nous avons écrit précédemment une fonction « ***coincide*** » qui permettait de déterminer la coïncidence entre deux objets de type ***point***. C'était une fonction membre de la classe. Nous pouvons résoudre le même problème en faisant de « ***coincide*** » une fonction indépendante **amie** de la classe **point**. Elle sera une fonction ordinaire qui peut manipuler les membres privés de la classe **point**. Il faut introduire dans point, la déclaration d'amitié approprié :

```
friend int coincide(point,point);
```

Exemple :

```
#include <iostream.h>   //fonction indépendante, 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);
      //déclaration 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() ;
}
```

7.3.LES AUTRES SITUATIONS D'AMITIE

1ère situation :

Dans cette situation, la fonction « ***FM_de_A*** », fonction membre de la classe **A**, a accès aux membres privés de la classe **B**:

```
class B
{
    // partie privée
    ........
    // partie publique
    friend int A::FM_de_A(char, B);
};

class A
{
   ........
   int FM_de_A(char, B);
};

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

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

2ème situation :

Dans cette situation, la fonction « ***f_anonyme*** » a accès aux membres privés des classes B et A :

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

```
class A
{
   // partie privée
   ........
   // partie publique
   friend void f_anonyme(B, A);
};
void f_anonyme(B to, A ti)
{
   ...
} // on pourra trouver ici une invocation des membres privés
     des objets to et ti.
```

7.4.APPLICATION A LA SURDEFINITION DES OPERATEURS

Exemple :

Reprenons l'exemple étudié précédemment et 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() ;
}
```

CHAPITRE 8 :
LA TECHNIQUE D'HERITAGE

Le concept de l'héritage constitue l'un des fondements de la programmation orientée objets. Il est la base des possibilités de réutilisation des composants logiciels.

Cette technique 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.

8.1.MISE EN ŒUVRE DE LA TECHNIQUE DE L'HERITAGE EN C++

Soit la classe point (sans constructeur ni destructeur)

```
#include <iostream.h>
#include <conio.h>
class point
{
   int x,y;
   public:
       void intialise(int abs, int ord)
       {
           x=abs;
           y=ord;
       }

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

       void affiche()
       {
           cout<< « le point est en » <<x<< «    »<<y<< « \n » ;
       }
} ;
```

Nous avons besoin de définir un nouveau type classe nommé ***pointcol***, destiné à manipuler des points colorés. Un tel point coloré peut être défini par ses coordonnées (comme un objet de type point), auxquelles on adjoint une information de couleur. Nous pouvons être tentés de définir poincol comme une classe dérivée de point.


```
class pointcol : public point
{
   short couleur;
   public:
       void colore(short c)
       {
           couleur = c ;
       }
} ;
```

La déclaration : « `class pointcol : public point ;` » spécifie que ***pointcol*** est une classe dérivée de la classe de base ***point***. Le mot **public** signifie que les membres publics de la classe de base ***point*** seront des membres publics de la classe dérivée.

On peut déclarer des objets de type ***pointcol*** par : `pointcol p, q ;`
Chaque objet de type ***pointcol*** peut alors faire appel :

a- Aux méthodes publiques de ***pointcol*** (ici colore)
b- Aux méthodes publiques de la classe de base *point* (ici init, déplace, affiche).

Exemple :

```
#include <iostream.h>
#include <point.h>
class pointcol : public point
{
   short couleur;
   public:
       void colore(short c)
       {
           couleur = c ;
       }
} ;

main()
{
   pointcol p;
   p.initialise(10,20);
   p.colore(5);
   p.affiche();
   p.deplace(2,4) ;
   p.affich() ;
}
```

8.2.UTILISATION, DANS UNE CLASSE DERIVEE, DES MEMBRES DE LA CLASSE DE BASE

Grâce à l'emploi du mot Public, les membres publics de ***point*** étaient également membres publics de ***pointcol*** ; c'est ce qui nous a permis d'y faire appel, au sein de la fonction **main( )**.

Or si on appel **affiche( )** pour un objet de type ***pointcol***, nous n'obtenons aucune information sur sa couleur.

Solution 1 :

Une première solution consiste à écrire une nouvelle fonction membre publique de ***poinlcol***, censée afficher à la fois les coordonnées et la couleur.

```
void affichec()
{
   cout<< x << «    » <<y<< « \n » ;
   cout<< « couleur : » << couleur << « \n » ;
}
```

Cela signifierait que la fonction **affichec( )**, membre de ***pointcol***, aurait accès aux membres privés de ***point***. Ceci serait contraire au principe d'encapsulation : d'où la règle adoptée par C++.

Une classe dérivée n'a pas accès aux membres privés de sa classe de base.

Par contre affichec peut accéder aux membres public de ***point***. Affichec ne peut pas accéder directement aux données privés x et y de la classe ***point***, elle peut néanmoins faire appel à la fonction **affiche( )** de cette même classe.

```
void pointcol ::affichec()
{
   affiche() ;
   cout<< « couleur : » << couleur << « \n » ;
}
```

D'une manière analogue, nous pouvons définir dans ***pointcol***, une nouvelle fonction d'initialisation nommée **initialisec**, chargée d'attribuer des valeurs aux données x et y et couleur.

```
void pointcol ::ainitialisec(int abs, int ord, int cl)
{
   initialise(abs, ord);
   couleur = cl;
}
```

Exemple :

```
#include <iostream.h>
#include <point.h>
class pointcol : public point
{
   short couleur;
   public:
       void colore(short cl)
       {
           couleur = cl ;
       }
       void affichec() ;
       void initialisec(int, int, short) ;
```



```
} ;

void pointcol ::affichec()
{
   affiche() ;
   cout<< « couleur : » << couleur << « \n » ;
}

void pointcol ::ainitialisec(int abs, int ord, int cl)
{
   initialise(abs, ord);
   couleur = cl;
}
```

8.3.REDEFINITION DES FONCTIONS MEMBRE

Dans l'exemple précédent de la classe ***pointcol***, nous disposons de deux fonctions membre (**affiche** et **affichc**).

Il est possible en C++ de leur donner le même nom, moyennant une petite précaution. Il n'est plus possible au sein de la fonction affiche de ***pointcol***, d'appeler la fonction **affiche** de ***point*** cela provoquerait un appel récursif. Il faut faire appel à l'opérateur de portée (**::**)

```
#include <iostream.h>
#include <point.h>
class pointcol : public point
{
   short couleur;
   public:
       void colore(short cl)
       {
           couleur = cl ;
       }
       void affiche() ;
       void initialise(int, int, short) ;
} ;
void pointcol ::affiche()
{
   point ::affich();
   cout<< « couleur : » << couleur << « \n » ;
}
void pointcol ::ainitialise(int abs, int ord, int cl)
{
   point :: initialise(abs, ord);
   couleur = cl;
}
 main()
{
   pointcol p;
   p.initialise(10,20, 5);
   p.affiche()
   p.point ::affiche();
   p.deplace(2,4) ;
   p.affich() ;
   p.colore(2) ;
   p.affiche() ;
}
```

8.4.APPEL DES CONSTRUCTEURS ET DES DESTRUCTEURS

8.4.1. Hiérarchisation des appels :

```
Class A
{
   ... ;
   public :
       A(...)
       ~A()
       ...
};

Class B
{
   ... ;
   public :
       B(...)
       ~B()
       ...
};
```

Si vous créer un objet de type B, C++ se charge automatiquement de faire appel au constructeur de A. La même démarche s'applique au destructeur.

8.4.2. Transmission d'informations entre constructeurs :

Un problème se pose dans le cas où le constructeur de A nécessite des arguments. C++ a prévu la possibilité de spécifier, dans la définition d'un constructeur d'une classe dérivée, les informations que l'on souhaite transmettre à un constructeur de la classe de base.

```
Class point
{
   ... ;
   public :
       point(int, int)
       ...
};

Class pointcol : public point
{
   ... ;
   public :
       pointcol(int, int, char)
       ...
};
```

Si l'on souhaite que ***pointcol*** retransmette à ***point*** les deux premières informations, on écrira :

```
Pointcol(int abs, int ord, short cl) : point (abs, ord) ;
```


8.4.3. Exemple

```
#include <iostream.h>
#include <conio.h>
class point
{
   int x,y;
   public:
       point(int abs = 0, int org = 0)
       {
           cout<<"++Const.point:"<< abs << "  " << ord << "\n";
           x=abs;
           y=ord;
       }
       ~point()
       {
           cout<<"--Destr.point:"<< x << "  " << y << "\n";
       }
} ;
class pointcol : public point
{
   short couleur;
   public:
       pointcol(int, int, short);
       ~point() ;
       {
           cout<<"--destr.pointcol couleur:"<< couleur << "\n";
       }
} ;

pointcol ::pointcol(int abs = 0, int org = 0, short cl = 1) :
          point(abs, ord)
{
   cout<<"++Const.point:"<< abs << "  " << ord << «   «  <<cl <<
         "\n";
   couleur = cl ;
}
main()
{
   pointcol a(10, 15, 3) ;
   poincol(2, 3);
   pointcol c(12);
   Pointcol d;
   pointcol * adr;
   adr = new pointcol(12, 25);
   delete adr;
}
```

8.5.CONTROLE DES ACCES

Jusqu'à maintenant nous n'avons examiné que le cas d'héritage le plus naturel càd :

- La classe dérivée a accès aux membres publics de la classe de base,
- Les utilisateurs de la classe dérivée ont accès à ses membres publics, ainsi qu'aux membres publics de sa classe de base.

8.5.1. Action sur le statut des membres d'une classe dérivée :

Pour que l'utilisateur d'une classe dérivée n'ait pas accès aux membres publics de sa classe de basse, il suffit de remplacer le mot **public** par **private**.

```
class point
{
   ...;
   public:
        point ( ...) ;
        void deplace( ...) ;
        void affiche() ;
        ...
} ;


class pointcol:private point
{
   ...;
   public:
        pointcol ( ...) ;
        void colore( ...) ;
        ...
} ;
```

Si p est de type ***pointcol***, les appels suivants seront rejetés par le compilateur :

```
p.affiche()   ->   p.point :: affiche()
p.deplace()   ->   p.point :: deplace()
```

`p.colore( )` est accepté.

Conclusion :

Le concepteur de la classe dérivée peut utiliser librement les membres publics de la classe de base, en revanche, il décide de fermer totalement cet accès à l'utilisateur de la classe dérivée. Cela ne sera employé que dans des cas bien précis, par exemple :

- Lorsque toutes les fonctions utiles de la classe de base sont redéfinies dans la classe dérivée et qu'il n'y a aucune raison de laisser l'utilisateur accéder aux anciennes.
- Lorsque l'on souhaite adapter l'interface d'une classe, de manière à répondre à certaines exigences dans ce cas la classe dérivée n'apporte rien de plus.

8.5.2. Les membres protégés :

Il existe un troisième statut protégé, il est défini par le mot clé **protected**.

```
Class X
{
   ... ;
   protected :
        ...
   public :
        ...
};
```


Les membres protégés restent inaccessibles à l'utilisateur de la classe, pour qui ils apparaissent analogues à des membres privés. Mais ils seront accessibles aux membres d'une éventuelle classe dérivée, tout en restant inaccessibles aux utilisateurs de cette classe.

8.5.3. Exemple :

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

class pointcol : public point
{
   short couleur;
   public:
       void affiche()
       {
           cout << « x: » << x « y : » << y << « \n » ;
           cout<< « couleur : » << couleur << « \n » ;
       }
} ;
```

8.5.4. Intérêt du statut protégé :

Les membres protégés restent comparables à des membres privés pour l'utilisateur de la classe, mais ils sont comparables à des membres publics pour le concepteur d'une classe dérivée.

Résumé :

- Protected, private et public contrôlent les droits que la classe de base accorde à une classe dérivée,
- Public et private contrôlent les droits que la classe dérivée peut s'octroyer,
  C'est la combinaison des deux qui va déterminer les droits obtenus. Dans l'exemple et le tableau suivant :
- XXXX désigne soit protecled, private ou public, et s'applique à la classe de base,
- YYYY désigne soit public, soit private et s'applique à la classe dérivée.

```
class A
{
   XXXX :
        int x ;
} ;

class B : YYYY A
{
   ...;
   ...;
} ;
```

XXXX	YYYY	Résultat
Public	Public	Accès sans restriction à x de A
Private	Public	Aucun accès à x de A.
Protected	Public	B peut accéder à x de A, l'accès est également! accordé aux classes dérivées à partir de B.
Public	Private	B peut accéder à x de A. Cet accès n'est cependant pas accordé aux classes dérivées à partir de B.
Private	Private	Aucun accès à x de A
Protected	Private	B peut accéder à x de A. Cet accès n'est cependant pas accordé aux classes dérivées à partir de B.

CHAPITRE 9 :
L'HERITAGE MULTIPLE

Au cours de ce chapitre on va répondre aux questions suivantes :

- ✓ Comment exprimer la dépendance « **Multiple** ». au sein d'une classe dérivée ?
- ✓ Comment seront appelés les constructeurs et destructeurs concernés : ordre, transmission d'informations ?
- ✓ Comment régler les conflits qui risquent d'apparaître dans des situations telles que celle-ci : les classes B et C sont filles de la classe A et aux même temps elles sont mères d'une classe D.

9.1. MISE EN ŒUVRE DE L'HERITAGE MULTIPLE

Soit la situation suivante : une classe ***pointcoul*** hérite au même temps des deux classes ***point*** et ***coul***.

```
class point
{
   int x, y;
   public:
       point ( ...)
      {
        ... ;
      }
      ~point(...)
      {
        ... ;
      }
       void affiche() ;
      {
        ... ;
      }
} ;
class coul
{
   short couleur;
   public:
       coul ( ...)
      {
        ... ;
      }
      ~point(...)
      {
        ... ;
      }
```

```
        void affiche() ;
        {
          ... ;
        }
} ;
```

9.2. DECLARATION DE POINTCOUL

```
Class pointcoul : public point, public coul
```

Dans le cas de l'héritage simple, le constructeur devait pouvoir retransmettre des informations au constructeur de la classe de base. Il en va de même ici, avec cette différence qu'il y a deux classes de base. L'en-tête du constructeur se présentera ainsi :

```
Pointcoul(Arguments de pointcoul) : Point (informations à transmettre à
point),  Coul(informations à transmettre à coul)
```

L'ordre d'appel des constructeurs est le suivant :

Constructeurs des classes de base, dans l'ordre où les classes de base sont déclarées dans la classe dérivée (ici, point puis coul),

Constructeur de la classe dérivée (ici, pointcoul).

Les destructeurs éventuels seront, là encore, dans l'ordre inverse lors de la destruction d'un objet de type ***pointcoul***.

Comme dans le cas de l'héritage simple, on peut, dans une fonction membre de la classe dérivée, utiliser toute fonction membre publique (ou protégé) d'une classe de base. Lorsque plusieurs fonctions membre portent le même nom dans différentes classes, on peut lever l'ambiguïté en employant l'opérateur de résolution de portée.

9.3. AFFICHAGE DE POINTCOUL

```
Void affiche()
{
   point :: affiche();
   coul :: affiche();
}
```

```
#include <iostream.h>
#include <conio.h>
class point
{
   int x,y;
   public:
       point(int abs , int org )
       {
           cout<<"++Const.point:" << "\n";
           x=abs;
           y=ord;
       }
       ~point()
       {
           cout<<"--Destr.point:" << "\n";
       }
```



```
       Void affiche()
       {
          cout<<"Coordonnées :" << x << «  » << y << "\n";
       }
} ;

class coul
{
   short couleur;
   public:
       coul(int cl)
       {
           cout<<"++Const.coul:" << "\n";
           couleur =cl;
       }
       ~point()
       {
           cout<<"--Destr.coul:" << "\n";
       }
       Void affiche()
       {
          cout<<"Couleur :" << couleur << "\n";
       }
} ;

class pointcol : public point, point coul
{
   public:
       pointcol(int, int, int);
       ~pointoul() ;
       {
           cout<<"--destr.pointcol"<< "\n";
       }
       Void affiche()
       {
          point ::affiche() ;
          coul :: affiche()
       }
} ;

pointcol ::pointcol(int abs, int org, int cl) : point(abs, ord),
           coul(cl)
{
   cout<<"++Const.pointcoul <<  "\n";
}

main()
{
   pointcol p(3, 9, 2) ;
   cout << "------------------\n";
   p.affiche();
   cout << "------------------\n";
   p.point :: affiche();
   cout << "------------------\n";
   p.coul ::affiche();
   cout << "------------------\n";
}
```

9.4. LES CLASSES VIRTUELLES :

Soit une situation dans laquelle, deux classes B et C sont filles de la classe A et aux même temps elles sont mères d'une classe D.

```
class A
{
   ...;
   int x, y ;
} ;

class B : public A
{
   ...;
} ;

class C : public A
{
   ...;
} ;

class D : public B, pubic C
{
   ...;
} ;
```

On peut dire que D hérite deux fois de A ! Dans ces conditions les membres de A vont apparaître deux fois dans D.

Pour éviter cela, on peut préciser dans la déclaration des classes B et C que la classe A est ***virtuelle.***

```
class B : public virtual A
{
   ...;
} ;

class C : public virtual A
{
   ...;
} ;

class D : public B, pubic C
{
   ...;
} ;
```

CHAPITRE 10 :
LE POLYMORPHISME EN C++

10.1. Introduction au concept de polymorphisme

Le polymorphisme est un concept des langages objet qui découle directement de l'héritage. Ce concept s'applique uniquement aux fonctions membres de classes dérivées. Il consiste à redéfinir une fonction pour une classe. Ainsi, une même fonction aura un traitement différent pour deux objets différents.

10.1.1. Le polymorphisme en C++

En C++, il faut indiquer au compilateur qu'il a affaire à une fonction polymorphe, sinon il serait tenté d'utiliser la fonction de la classe de base.

En fait, s'il y a un accès par référence, alors la fonction utilisée sera la fonction "terminal", alors que si l'accès se fait par adresse alors il utilisera la fonction du type de l'objet.

Il ne faut pas oublier de rappeler qu'il est possible de pointer un objet dérivé par un pointeur vers un objet de la classe de base.

Pour permettre d'utiliser la fonction la plus "évoluée" dans l'architecture de classes, il faudra utiliser le mot clé virtual du C++ devant le prototype de la fonction qui doit être polymorphe (et devant le prototype seulement). Il faut le spécifier devant la fonction de base uniquement pour qu'elle puisse être "polymorphée". Mais en général, on le met devant tous les prototypes de cette fonction. Ainsi, on permet une future dérivation.

Il est à noter que si une fonction virtuelle n'est pas redéfinie, le programme utilisera la dernière fonction redéfinie. Il se peut que la fonction polymorphe consiste à faire la même chose que la fonction de base, plus un traitement. A ce moment là on peut appeler la fonction d'une classe précédente en faisant: <classe_de_base>::<fonction> (<parametre>);

10.1.2. Polymorphisme et constructeur/destructeur

Si on défini un destructeur pour la classe, on est obligé de le définir virtuellement. Evidement, il faut que l'ensemble de l'objet soit détruit. En ce qui concerne les constructeurs, il est à noter qu'ils ne peuvent appeler de fonction polymorphe. De plus, ni les constructeurs ni les destructeurs ne peuvent être polymorphes.

Une classe abstraite est une classe qui contient au moins une fonction virtuelle pure, ou qui n'a pas redéfinit une fonction virtuelle pure. Une fonction virtuelle pure est une fonction

virtuelle dont le corps est explicitement non donné, on précise un =0 à la fin prototype d'une telle fonction.

Une classe abstraite est une classe dont aucun objet de ce type ne peut être créé. Ainsi l'abstraction de la classe se propagera dans les classes dérivées tant que cette fonction n'aura pas été redéfinie. Cela est relativement pratique dans le cas d'une classe définissant un concept général, et non une classe en elle même.

On pourrait par exemple imaginer une classe arme se dérivent en arme_blanche et arme_feu. Il est évident qu'ici aucun objet de type arme ne sera créé car arme est un concept général. On pourra donc spécifier une fonction virtuelle pure dans la classe arme.

10.1.3. Origine du mot polymorphisme

En grec, « Poly » signifie « plusieurs », comme dans *polygone* ou *polytechnique*, et « morphe » signifie «forme» comme dans… euh… *amorphe* ou *zoomorphe*.

Nous allons donc parler de choses ayant plusieurs formes. Ou, pour utiliser des termes informatiques, nous allons créer du code fonctionnant de différentes manières selon le type qui l'utilise.

10.2. La résolution des liens

Commençons avec un peu d'héritage simple. Prenons un autre exemple pour varier un peu. Il s'agit de la création d'un programme de gestion d'un garage et des véhicules qui y sont stationnés. Imaginons que notre garagiste sache réparer à la fois des voitures et des motos.

Dans son programme, il aurait les classes suivantes : `Vehicule`, `Voiture` et `Moto`.

```
class Vehicule
{
public:
void affiche() const;//Affiche une description du Vehicule

protected:
int m_prix;//Chaque véhicule a un prix
};

class Voiture :public Vehicule //Une Voiture EST UN Vehicule
{
public:
void affiche() const;

private:
int m_portes;//Le nombre de portes de la voiture
};

class Moto :public Vehicule  //Une Moto EST UN Vehicule
{
public:
void affiche() const;

private:
```

```
    double m_vitesse;//La vitesse maximale de la moto
};
```

Le corps des fonctions `affiche()` est le suivant :

```
void Vehicule::affiche() const
{
    cout <<"Ceci est un vehicule."<< endl;
}

void Voiture::affiche() const
{
    cout <<"Ceci est une voiture."<< endl;
}

void Moto::affiche() const
{
    cout <<"Ceci est une moto."<< endl;
}
```

Chaque classe affiche donc un message différent. On utilise ici le masquage pour redéfinir la fonction `affiche()` de `Vehicule` dans les deux classes filles.

Essayons donc ces fonctions avec un `main()` :

```
int main()
{
    Vehicule v;
    v.affiche();//Affiche "Ceci est un vehicule."

    Moto m;
    m.affiche();//Affiche "Ceci est une moto."

return0;
}
```

En testant ce programme, nous devrions rien observer de particulier. Mais cela va venir.

10.2.1. La résolution statique des liens

Créons une fonction supplémentaire qui reçoit en paramètre un `Vehicule` et modifions le `main()` afin d'utiliser cette fonction :

```
void presenter(Vehicule v)  //Présente le véhicule passé en
argument
{
    v.affiche();
}

int main()
{
    Vehicule v;
    presenter(v);

    Moto m;
```



```
    presenter(m);

    return0;
}
```

A priori, rien n'a changé. Les messages affichés devraient être les mêmes. Voyons cela :

```
Ceci est un vehicule.
Ceci est un vehicule.
```

Le message n'est pas correct pour la moto ! C'est comme si, lors du passage dans la fonction, la vraie nature de la moto s'était perdue et qu'elle était redevenue un simple véhicule.

Comment est-ce possible ?

Comme il y a une relation d'héritage, nous savons qu'une moto *est un* véhicule, un véhicule amélioré en quelque sorte puisqu'il possède un attribut supplémentaire. La fonction `presenter()` reçoit en argument un `Vehicule`. Ce peut être un objet réellement de type `Vehicule` mais aussi une `Voiture` ou, comme dans l'exemple, une `Moto`.

Ce qui est important c'est que, pour le compilateur, à l'intérieur de la fonction, on manipule un `Vehicule`. Peu importe sa vraie nature. Il va donc appeler la « version `Vehicule` » de la méthode `afficher()` et pas la « version `Moto` » comme on aurait pu l'espérer. Dans la fonction`presenter()`, pas moyen de savoir ce que sont réellement les véhicules reçus en argument.

En termes techniques, on parle de **résolution statique des liens**. La fonction reçoit un `Vehicule`, c'est donc toujours la « version `Vehicule` » des méthodes qui sera utilisée.

C'est le type de la variable qui détermine quelle fonction membre appeler et non sa vraie nature.

Pour résoudre le problème, Il y a un moyen de changer ce comportement.

10.2.2. La résolution dynamique des liens

Ce qu'on souhaiterait, c'est que la fonction `presenter()` appelle la bonne version de la méthode. C'est-à-dire qu'il faut que la fonction connaisse la vraie nature du `Vehicule`. C'est ce qu'on appelle la **résolution dynamique des liens**. Lors de l'exécution, le programme utilise la bonne version des méthodes car il sait si l'objet est de type mère ou de type fille.

Pour faire cela, il faut deux *ingrédients* :

- utiliser un pointeur ou une référence ;
- utiliser des méthodes virtuelles.

Si ces deux ingrédients ne sont pas réunis, alors on retombe dans le premier cas et l'ordinateur n'a aucun moyen d'appeler la bonne méthode.

10.3. Les fonctions virtuelles

Je vous ai donné la liste des ingrédients, allons-y pour la préparation du menu. Commençons par les méthodes virtuelles.

10.3.1. Déclarer une méthode virtuelle

Il suffit d'ajouter le mot-clé `virtual` dans le prototype de la classe (dans le fichier `.h` donc). Pour notre garage, cela donne :

```
class Vehicule
{
public:
virtual void affiche() const;//Affiche une description du
Vehicule

protected:
int m_prix;//Chaque véhicule a un prix
};

class Voiture:public Vehicule  //Une Voiture EST UN Vehicule
{
public:
virtual void affiche() const;

private:
int m_portes;//Le nombre de portes de la voiture
};

class Moto :public Vehicule  //Une Moto EST UN Vehicule
{
public:
virtual void affiche() const;

private:
double m_vitesse;//La vitesse maximale de la moto
};
```

Il n'est pas nécessaire de mettre « `virtual` » devant les méthodes des classes filles. Elles sont automatiquement virtuelles par héritage.
Il est préférable de le mettre pour se souvenir de leur particularité.
Notez bien qu'il n'est pas nécessaire que toutes les méthodes soient virtuelles. Une classe peut très bien proposer des fonctions « normales » et d'autres virtuelles.

Il ne faut pas mettre `virtual` dans le fichier `.cpp` mais uniquement dans le `.h`

10.3.2. Et utiliser une référence

Le deuxième ingrédient est un pointeur ou une référence. On peut aussi utiliser des pointeurs.

Réécrivons donc la fonction `presenter()` avec comme argument une référence.

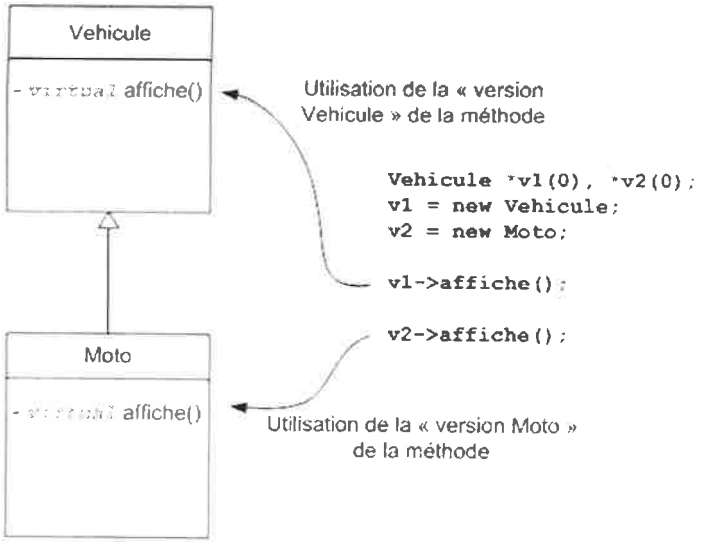
```
void presenter(Vehicule const& v)  //Présente le véhicule passé
en argument
{
    v.affiche();
}

int main()  //Rien n'a changé dans le main()
{
    Vehicule v;
    presenter(v);

    Moto m;
    presenter(m);

return0;
}
```

On aussi ajouté un `const`. Comme on ne modifie pas l'objet dans la fonction, autant le faire savoir au compilateur et au programmeur en déclarant la référence constante.
Voilà. Il ne nous reste plus qu'à tester :

```
Ceci est un vehicule.
Ceci est une moto.
```

Cela marche, la fonction `presenter()` a bien appelé la bonne version de la méthode. En utilisant des fonctions virtuelles ainsi qu'une référence sur l'objet, la fonction `presenter()` a pu correctement choisir la méthode à appeler.

On aurait obtenu le même comportement avec des pointeurs à la place des références, comme sur le schéma suivante.



Un même morceau de code a eu deux comportements différents suivant le type passé en argument. C'est donc du polymorphisme. On dit aussi que les méthodes `affiche()` ont un *comportement polymorphique*.

10.4. Les méthodes spéciales

Les méthodes d'une classe qui ne sont jamais héritées sont :

- tous les constructeurs ;
- le destructeur.

Toutes les autres méthodes peuvent être héritées et peuvent avoir un comportement polymorphique si on le souhaite. Mais qu'en est-il pour ces méthodes spéciales ?

10.4.1. Le cas des constructeurs

Un constructeur virtuel a-t-il du sens ? Non ! Quand on veut construire un véhicule quelconque, on ne sais lequel on veut veut construire. On peut donc à la compilation déjà savoir quel véhicule construire. On n'a pas besoin de résolution dynamique des liens et, par conséquent, pas besoin de virtualité. *Un constructeur ne peut pas être virtuel.*

Et cela va même plus loin. Quand nous sommes dans le constructeur, on sait quel type nous construisons, on n'a donc à nouveau pas besoin de résolution dynamique des liens. D'où la règle suivante : *on ne peut pas appeler de méthode virtuelle dans un constructeur. Si on essaye quand même, la résolution dynamique des liens ne se fait pas.*

10.4.2. Le cas du destructeur

Créons un petit programme utilisant nos véhicules et des pointeurs, puisque c'est un des ingrédients du polymorphisme.

```
int main()
{
    Vehicule *v(0);
    v =new Voiture;
//On crée une Voiture et on met son adresse dans un pointeur de
Vehicule

    v->affiche();//On affiche "Ceci est une voiture."

    delete v;//Et on détruit la voiture

    return0;
}
```

Nous avons un pointeur et une méthode virtuelle. La ligne `v->affiche()` affiche donc le message que l'on souhaitait. Le problème de ce programme se situe au moment du `delete`. Nous avons un pointeur mais la méthode appelée n'est pas virtuelle. C'est donc le destructeur de `Vehicule` qui est appelé et pas celui de`Voiture` !

Dans ce cas, cela ne porte pas vraiment à conséquence, le programme ne plante pas. Mais imaginez que vous deviez écrire une classe pour le maniement des moteurs électriques d'un robot. Si c'est le mauvais destructeur qui est appelé, vos moteurs ne s'arrêteront peut-être pas. Cela peut vite devenir dramatique.

Il faut donc impérativement appeler le bon destructeur. Et pour ce faire, une seule solution : rendre le destructeur virtuel ! Cela nous permet de formuler une nouvelle règle importante : *un destructeur doit toujours être virtuel si on utilise le polymorphisme.*

10.4.3. Le code amélioré

Ajoutons donc des constructeurs et des destructeurs à nos classes. Tout sera alors correct.

```
class Vehicule
{
public:
    Vehicule(int prix);//Construit un véhicule d'un certain prix
    virtualvoid affiche() const;
    virtual~Vehicule();//Remarquez le 'virtual' ici

protected:
    int m_prix;
};

class Voiture:public Vehicule
{
```

```
public:
    Voiture(int prix,int portes);
   //Construit une voiture dont on fournit le prix et le nombre
     de portes
    virtualvoid affiche() const;
    virtual~Voiture();

private:
    int m_portes;
};

class Moto :public Vehicule
{
public:
    Moto(int prix,double vitesseMax);
   //Construit une moto d'un prix donné et ayant une certaine
     vitesse maximale
     virtualvoid affiche() const;
     virtual~Moto();

private:
     double m_vitesse;
};
```

Il faut bien sûr également compléter le fichier source :

```
Vehicule::Vehicule(int prix):m_prix(prix)
{}

void Vehicule::affiche() const
//On va profiter pour modifier un peu les fonctions d'affichage
{
    cout <<"Ceci est un vehicule coutant "<< m_prix <<"Dhs."<<
       endl;
}

Vehicule::~Vehicule() //Même si le destructeur ne fait rien, on
doit le mettre !
{}

Voiture::Voiture(int prix,int portes):Vehicule(prix),
m_portes(portes)
{}

void Voiture::affiche() const
{
    cout <<"Ceci est une voiture avec "<< m_portes <<" portes et
    coutant "<< m_prix <<" Dhs."<< endl;
}

Voiture::~Voiture()
{}

Moto::Moto(int prix,double vitesseMax):Vehicule(prix),
m_vitesse(vitesseMax)
{}

void Moto::affiche() const
{
    cout <<"Ceci est une moto allant a "<< m_vitesse <<" km/h et
```



```
coutant "<< m_prix <<" Dhs."<< endl;
}

Moto::~Moto()
{}
```

Nous sommes donc prêts à aborder un exemple concret d'utilisation du polymorphisme.

10.5. Les fonctions virtuelles pures

10.5.1. Le problème des roues

Voici une version de la fonction `nbrRoues()` pour les classes `Vehicule` et `Voiture` uniquement.

```
class Vehicule
{
public:
    Vehicule(int prix);
    Virtual void affiche() const;
    virtualint nbrRoues() const;//Affiche le nombre de roues du
                                  véhicule
    virtual~Vehicule();

    protected:
       int m_prix;
};

class Voiture :public Vehicule
{
public:
    Voiture(int prix,int portes);
    Virtual void affiche() const;
    Virtual int nbrRoues() const;//Affiche le nombre de roues de
                                   la voiture
    virtual~Voiture();

private:
    int m_portes;
};
```

```
int Vehicule::nbrRoues() const
{
   //Que mettre ici ????
}

int Voiture::nbrRoues() const
{
    return4;
}
```

Vous l'aurez compris, on ne sait pas vraiment quoi mettre dans la « version `Vehicule` » de la méthode. Les voitures ont 4 roues et les motos 2 mais, pour un véhicule en général, on ne

peut rien dire ! On aimerait bien ne rien mettre ici ou carrément supprimer la fonction puisqu'elle n'a pas de sens.

Mais si on ne déclare pas la fonction dans la classe mère, alors on ne pourra pas l'utiliser depuis notre collection hétérogène. Il nous faut donc la garder ou au minimum dire qu'elle existe mais qu'on n'a pas le droit de l'utiliser. On souhaiterait ainsi dire au compilateur : «Dans toutes les **classes filles** de `Vehicule`, il y a une fonction nommée `nbrRoues()` qui renvoie un `int` et qui ne prend aucun argument mais, dans la classe `Vehicule`, cette fonction n'existe pas. »

C'est ce qu'on appelle une *méthode virtuelle pure*.

Pour déclarer une telle méthode, rien de plus simple. Il suffit d'ajouter « = 0 » à la fin du prototype.

```
class Vehicule
{
public:
    Vehicule(int prix);
    Virtual void affiche() const;
    Virtual int nbrRoues() const=0;//Affiche le nombre de roues
                                   du véhicule
    virtual~Vehicule();

protected:
    int m_prix;
};
```

Et évidemment, on n'a rien à écrire dans le `.cpp` puisque, justement, on ne sait pas quoi y mettre. On peut carrément supprimer complètement la méthode. L'important étant que son prototype soit présent.

10.5.2. Les classes abstraites

Une classe qui possède au moins une méthode virtuelle pure est une **classe abstraite**. Notre classe `Vehicule` est donc une classe abstraite.

Pourquoi donner un nom spécial à ces classes ? Eh bien parce qu'elles ont une règle bien particulière : *on ne peut pas créer d'objet à partir d'une classe abstraite*. La ligne suivante ne compilera pas.

```
Vehicule v(1000000);//Création d'un véhicule valant 100 000 Dhs.
```

Dans le jargon des programmeurs, on dit qu'on ne peut pas créer d'instance d'une classe abstraite.
La raison en est simple : si nous pouvons créer un `Vehicule`, alors nous pourrions essayer

d'appeler la fonction `nbrRoues()` qui n'a pas de corps et ceci n'est pas possible. Par contre, nous pouvons tout à fait écrire le code suivant :

```
int main()
{
    Vehicule* ptr(0);//Un pointeur sur un véhicule

    Voiture caisse(20000,5);
    //On crée une voiture
    //Ceci est autorisé puisque toutes les fonctions ont un
      corps

    ptr = &caisse;//On fait pointer le pointeur sur la voiture

    cout << ptr->nbrRoues() << endl;
    //Dans la classe fille, nbrRoues() existe, ceci est donc
      autorisé

return0;
}
```

Ici, l'appel à la méthode `nbrRoues()` est polymorphique, puisque nous avons un pointeur et que notre méthode est virtuelle. C'est donc la « version `Voiture` » qui est appelée. Donc même si la « version`Vehicule` » n'existe pas, il n'y a pas de problèmes.

Si l'on veut créer une nouvelle sorte de `Vehicule` (`Camion` par exemple), on sera obligé de redéfinir la fonction `nbrRoues()`, sinon cette dernière sera virtuelle pure par héritage et, par conséquent, la classe sera abstraite elle aussi.

On peut résumer les fonctions virtuelles de la manière suivante :

- une méthode virtuelle *peut* être redéfinie dans une classe fille ;
- une méthode virtuelle pure *doit* être redéfinie dans une classe fille.

En résumé

- Le polymorphisme permet de manipuler des objets d'une classe fille *via* des pointeurs ou des références sur une classe mère.
- Deux ingrédients sont nécessaires : des fonctions virtuelles et des pointeurs ou références sur l'objet.
- Si l'on ne sait pas quoi mettre dans le corps d'une méthode de la classe mère, on peut la déclarer virtuelle pure.
- Une classe avec des méthodes virtuelles pures est dite abstraite. On ne peut pas créer d'objet à partir d'une telle classe.