REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE D'ORAN
- USTO-MB-
FACULTE DES MATHEMATIQUES ET INFORMATIQUE
DEPARTEMENT D'INFORMATIQUE

Polycopié de cours

Introduction à la Programmation Orientée Objet

- Application en langage C++ -

Enseignante : Dr. H. YEDJOUR

2017 -2018

SOMMAIRE

Liste des exercices

A qui s'adresse ce polycopié de cours ?

Prérequis

Chapitre 1 Rappels élémentaires du langage C

1.1 La fonction principale main 3
1.2 Déclaration d'une variable 3
1.3 Fonctions et passage de paramètres........ 4
1.4 Définition et utilisation de tableaux........ 5
1.5 Définition et utilisation de structures........ 6

Chapitre 2 Introduction à la programmation orientée objet (POO)

2.1 Pourquoi utiliser la programmation orientée objet ?........ 9
2.2 Objectifs de la programmation orientée objet........ 9
2.3 Histoire de la programmation orientée objet........ 10
2.4 C++ et la programmation orientée objet 10

Chapitre 3 Principes fondamentaux de la POO

3.1 Notion d'objet et de classe........ 12
3.1.1 Notion d'objet 12
3.1.1.1 Les champs 12
3.1.1.2 Les méthodes 12
3.1.2 Notion de classe........ 12
3.1.3 Présentation de la notion d'objet/classe........ 13
3.1.4 Diagramme de classe........ 13
3.1.5 Utilisation d'une classe 14
3.1.6 Déclaration d'une fonction membre 15
3.2 Conception d'un programme Orienté Objet........ 15

3.2.1 Déclaration et instanciation.......... 17
3.2.1.1 Instanciation d'un objet.......... 17
3.2.1.2 Instanciation de plusieurs objets.......... 17

Chapitre 4 Notions de Constructeur, Destructeur
4.1 Les Constructeurs 23
4.1.1 Types de constructeurs 23
4.1.1.1 Constructeur par défaut 23
4.1.1.2 Constructeur d'initialisation 24
4.1.1.3 Surdéfinition ou surcharge de constructeurs.......... 25
4.1.1.4 Constructeur de copie.......... 29
4.2 Le destructeur 32
4.3 L'encapsulation 35
4.3.1 Les Différents niveaux d'accessibilité des propriétés et méthodes 35
4.3.1.1 Private 35
4.3.1.2 Public 35
4.3.1.3 Protected 35
4.3.2 Principe de l'encapsulation 35
4.3.3 Importance de l'encapsulation 36

Chapitre 5 Héritage et Polymorphisme
5.1 Héritage 40
5.1.1 Graphe de l'héritage.......... 40
5.1.2 Propriétés générales de l'héritage 41
5.1.3 Propriétés des classes dérivées et Syntaxe de l'héritage 41
5.2 Polymorphisme 51
5.2.1 Désignation d'un objet à l'aide d'une référence 54
5.2.2 Déclaration des méthodes virtuelles 55
Conclusion 59
Références 60

LISTE DES EXERCICES

Chapitre 3 Principes fondamentaux de la POO

Exercice 3.1 avec solution ………………………………………………………………….. 15
Exercice 3.2 avec solution ………………………………………………………………….. 17
Exercice 3.3 avec solution ………………………………………………………………….. 19

Chapitre 4 Notions de Constructeur, Destructeur

Exercice 4.1 avec solution ………………………………………………………………….. 26
Exercice 4.2 avec solution ………………………………………………………………….. 30
Exercice 4.3 avec solution ………………………………………………………………….. 32
Exercice 4.4 avec solution ………………………………………………………………….. 36

Chapitre 5 Héritage et Polymorphisme

Exercice 5.1 avec solution ………………………………………………………………….. 42
Exercice 5.2 sans solution ………………………………………………………………….. 45
Exercice 5.3 avec solution ………………………………………………………………….. 46

A qui s’adresse ce polycopié de cours ?

Ce polycopié est destiné aux étudiants du 1er cycle du système (LMD) du département Informatique, mais sa vocation première n’en reste pas moins une initiation à la programmation orientée objet.

Ce polycopié sera un compagnon d’étude enrichissant pour les étudiants qui comptent la programmation objet dans leur cursus d’étude spécialement en langage C++ orientée objet. Il devrait les aider, le cas échéant, à évoluer de la programmation procédurale à la programmation objet.

Les codes sources dans ce polycopié de cours sont exécutés sur l’environnement de Dev-C++. Il suffit de faire de copier/coller. Le résultat de l’exécution est affiché dans la plupart des exercices.

Ce polycopié est divisé en cinq parties :

- Rappels élémentaires du langage C,
- Introduction à la POO,
- Les principes fondamentaux de la POO,
- Les notions de Constructeur, Destructeur,
- L’Héritage et le polymorphisme.

Prérequis

La lecture de ce Polycopié de cours nécessite une connaissance des bases de la programmation en C :

- ✓ bonne connaissance en programmation C et en algorithmique ;
- ✓ notions de variable et de type ;
- ✓ structures conditionnelles et itératives (boucles) ;
- ✓ programmation modulaire (sous-programmes) et passage de paramètres ;
- ✓ tableaux et structures.

Chapitre 1

Rappels élémentaires du langage C

Nous allons commencer dans ce chapitre par donner quelques rappels sur des points les plus importants à l'aide des exemples de programmes en C, principalement :

- la fonction principale main,
- déclaration d'une variable,
- fonction et passage de paramètres,
- définition et utilisation de tableaux,
- définition et utilisation de structures.

1.1 La fonction principale main

La fonction main est le point d'entrée du programme, elle est indispensable, les instructions sont exécutées dans l'ordre d'apparition dans le programme.

Exemple :

```
#include <iostream>
#include <stdio.h>
int main(void)
{
    printf("Bienvenue tout le monde !\n");
    system ("pause");
    return 0;
}
```

Exécution :

```
Bienvenue tout le monde !
Appuyez sur une touche pour continuer...
```

1.2 déclaration d'une variable

En langage C, une variable est constituée d'un nom et d'une valeur.

Exemple :

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

int main(void)
{
    int a;        // Declaration de l'entier a
    int b = 7;  // Declaration et initialisation de l'entier b
    a = 8;       // Affectation de 8 a l'entier a
    printf("Ma valeur entiere a = %d\n",a);
    printf("Ma valeur entiere b = %d\n",b);
   return 0;
}
```

Exécution :

```
Ma valeur entiere a = 8
Ma valeur entiere b = 7

--------------------------------
Process exited after 0.1898 seconds with return value 0
Appuyez sur une touche pour continuer...
```

1.3 Fonctions et passage de paramètres

Le passage de paramètres pour les fonctions se fait soit par valeur, soit par adresse.

Exemple :

```
#include <stdio.h>

int addition1 (int a, int b) // c = a + b
{ int c; c = a + b; return c; }

void addition2 (int a, int b, int c) // c = a + b
{ c = a + b; }

void addition3 (int a, int b, int *c) // c = a + b
{ *c = a + b; }

int main( void )
{
  int x1 = 7, x2 = 8;
  int c_addition1 = addition1 (x1 ,x2); // x1 et x2 sont passées par valeurs
  int c_addition2 = 0; addition2 (x1 ,x2 ,c_addition2); //  c_addition2 est passée par valeur
  int c_addition3; addition3 (x1 ,x2 ,& c_addition3); //  c_addition3 est passée par adresse
  printf(" c_addition1 = %d\n c_addition2 = %d\n c_addition3 = %d\n",c_addition1 ,c_addition2
  ,c_addition3);
  return 0;
}
```

Exécution :

Remarque :

Une variable passée par valeur, n'est pas modifiée à l'extérieur de la fonction. Pour la modifier, il faut l'utiliser avec le passage par adresse (& c_addition3).

1.4 Définition et utilisation de tableaux

Le code suivant permet la recherche du minimum et du maximum dans un vecteur de dimension N.

```
#include <stdio.h>
#include <iostream>
main()
{         /* Déclarations */
    int A[50]; /* tableau donné  */
    int N;    /* dimension      */
    int I;    /* indice courant */
    int MIN;   /* position du minimum */
    int MAX;   /* position du maximum */
        /* Saisie des données */
    printf("Dimension du tableau (max.50): ");
    scanf("%d", &N );
    for (I=0; I<N; I++)
      {
       printf("A[%d]= ", I);
       scanf("%d", &A[I]);
      }
    /* Affichage du tableau */
    printf("\naffichage du tableau saisie :\n");
    for (I=0; I<N; I++)
       printf("%d ", A[I]);
       printf("\n");
    /* Recherche du maximum et du minimum */
    MIN=0;
    MAX=0;
    for (I=0; I<N; I++)
    {
        if(A[I]>A[MAX]) MAX=I;
        if(A[I]<A[MIN]) MIN=I;
    }
     /* Edition du résultat */
    printf("\nle minimum est: A[%d]=%d\n", MIN,A[MIN]);
    printf("le maximum est: A[%d]=%d\n", MAX,A[MAX]);
    system("pause");
    return  0;
    }
```

Exécution :

```
Dimension du tableau (max.50): 5
A[0]= 6
A[1]= -8
A[2]= 31
A[3]= 0
A[4]= -5

affichage du tableau saisie :
6 -8 31 0 -5

le minimum est: A[1]=-8
le maximum est: A[2]=31
Appuyez sur une touche pour continuer...
```

1.5 Définition et utilisation de structures

Les structures permettent de regrouper des objets (des variables) au sein d'une entité repérée par un seul nom de variable. Les objets contenus dans la structure sont appelés champs de la structure.

Exemple :

Dans cet exemple, nous définissons une structure point composée de 2 réels x et y. Dans la fonction main, on définit 3 points a,b et c.

```
#include<iostream>
using namespace std;

struct point
{
   double x,y;
};

int main()
{
    point a,b,c;

    // les coordonées de a sont fixées
        a.x=7;
        a.y=8;

    // saisie des coordonées de b
        cout << "Tapez l'abscisse de b : ";
        cin >> b.x;
        cout << "Tapez l'ordonnée de b : ";
        cin >> b.y;
    // calcul des coordonées de c
        c.x = (b.x + a.x) / 2;
```

```
        c.y = (b.y + a.y) / 2;
        cout << "Abscisse de c : " << c.x << endl;
        cout << "Ordonnée de c : " << c.y << endl;
return 0;
}
```

<u>Exécution :</u>

```
Tapez l'abscisse de b : 3
Tapez l'ordonnÚe de b : 2
Abscisse de c : 5
OrdonnÚe de c : 5

--------------------------------
Process exited after 12.76 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Chapitre 2

Introduction à la Programmation Orientée Objet (POO)

La POO nécessite une longue pratique, beaucoup d'empirisme. Dans ce chapitre, nous allons essayer de décrire brièvement ce qu'est la programmation orientée objet (abrégée souvent en POO), on essayera de répondre aux questions souvent posées sur l'utilisation de la POO, ses objectifs ainsi que son histoire.

2.1 Pourquoi utiliser la programmation orientée objet ?

La programmation de logiciels a connu il y a quelques années le passage d'une ère artisanale à une ère industrielle. Des logiciels de plus en plus complexes doivent être réalisés dans des délais de plus en plus courts, tout en maintenant le meilleur niveau de qualité possible.

Le coût des dépenses informatiques principalement coût des logiciels est de plus en plus élevé. L'emploi de techniques de développement adaptées ainsi que la création de nouvelles applications à partir de composants existants, déjà testés, a toutes chances de produire un code plus fiable. De plus, elle peut se révéler nettement plus rapide et plus économique.

2.2 Objectifs de la programmation orientée objet

La programmation orientée objet, souvent abrégée POO, permet de concevoir une application sous la forme d'un ensemble de briques logicielles appelées des objets. Chaque objet joue un rôle précis et peut communiquer avec les autres objets. Les interactions entre les différents objets vont permettre à l'application de réaliser les fonctionnalités attendues.

Un programme objet est une production d'un ensemble d'objets. L'accès à un objet se fait uniquement via l'interface, il permet en effet, la séparation de l'interface et de l'implantation de ce que fait l'objet.

Finalement, la POO facilite la conception de programmes par réutilisation de composants existants, avec tous les avantages évoqués plus haut. Elle constitue le standard actuel (on parle de paradigme) en matière de développement de logiciels.

Les objectifs de la POO peuvent être classés selon les points suivants :

- ✓ Diminuer le coût du logiciel ;
- ✓ Augmenter sa durée de vie, sa réutilisabilité et sa facilité de maintenance ;
- ✓ Concevoir des logiciels avec des exigences de qualité.

2.3 Histoire de la programmation orientée objet

Les concepts de la POO naissent au cours des années 1970 dans des laboratoires de recherche en informatique. Les premiers langages de programmation véritablement orientés objet ont été Simula, puis Smalltalk.

À partir des années 1980, les principes de la POO sont appliqués dans de nombreux langages comme Eiffel (créé par le Français Bertrand Meyer), C++ (une extension

objet du langage C créé par le Danois Bjarne Stroustrup) ou encore Objective C (une autre extension objet du C utilisé, entre autres, par l'IOS d'Apple).

Les années 1990 ont vu l'avènement des langages orientés objet dans de nombreux secteurs du développement logiciel, et la création du langage Java par la société Sun Microsystems. Le succès de ce langage, plus simple à utiliser que ses prédécesseurs, a conduit Microsoft à riposter en créant au début des années 2000 la plate-forme .NET et le langage C#, cousin de Java.

De nos jours, de très nombreux langages permettent d'utiliser les principes de la POO dans des domaines variés : Java et C# bien sûr, mais aussi PHP (à partir de la version 5), VB.NET, PowerShell, Python, etc. Une connaissance minimale des principes de la POO est donc indispensable à tout informaticien, qu'il soit développeur ou non.

2.4 C++ et la programmation orientée objet

C++ est un langage de programmation compilé permettant la programmation sous de multiples paradigmes (comme la programmation procédurale, orientée objet ou générique). Ses bonnes performances, et sa compatibilité avec le C en font un des langages de programmation les plus utilisés dans les applications où la performance est critique. Créé initialement par Bjarne Stroustrup dans les années 1980, le langage C++ est aujourd'hui normalisé par l'ISO. Les langages C/C++ bénéficient d’un large plébiscite. Ensemble, ils sont plus utilisés que Java et presque autant que Python. Parmi les environnements de développement intégrés (EDI) C/C++ les plus utilisés en 2018, on trouve le Visual Studio de Microsoft en tête. De nombreux programmeurs utilisent le Dev-C++.

Le C++ utilise les concepts de la programmation orientée objet et permet entre autres la classification (classes), l'encapsulation, des relations entre les classes tels que l'héritage simple et multiple ainsi que le polymorphisme. On en reviendra sur ces notions dans les prochains chapitres.

Chapitre 3

Principes fondamentaux de la POO

Dans ce chapitre, on travaillera avec les bases de la programmation orientée objets. Des exemples de programmes seront modularisés sous forme de méthodes auxiliaires ainsi que sous forme de classes comprenant des variables d'instance et des méthodes d'instance. Ce qui est important ici, c'est de comprendre les notions de base de la POO et de pouvoir les utiliser dans de petits programmes.

3.1 Notion d’objet et de classe

3.1.1 Notion d’objet

Quand on utilise la POO, on cherche à représenter le domaine étudié sous la forme d'objets. C'est la phase de modélisation orientée objet.

Un objet est une entité qui représente (modélise) un élément du domaine étudié. Il est caractérisé par :

- ✓ un état : ensemble de valeurs, les données.
- ✓ un comportement : ensemble d’opérations réalisables.

Exemple : une voiture, un compte bancaire, un nombre complexe, une facture, etc.

Le fait de concevoir une application comme un système d’objets interagissant entre eux apporte une certaine souplesse et une forte abstraction. Un objet rassemble donc de fait deux éléments de la programmation procédurale : les champs et les méthodes.

3.1.1.1 Les champs

Les champs sont à l'objet ce que les variables sont à un programme : ce sont eux qui ont en charge les données à gérer. Tout comme n'importe quelle autre variable, un champ peut posséder un type quelconque défini au préalable : nombre, caractère... ou même un type objet.

3.1.1.2 Les méthodes

Les méthodes sont les éléments d'un objet qui servent d'interface entre les données et le programme. Sous ce nom se cachent simplement des procédures ou fonctions destinées à traiter les données.

Les champs et les méthodes d'un objet sont ses membres. Si nous résumons : un objet est donc un type servant à stocker des données dans des champs et à les gérer au travers des méthodes. Si on se rapproche du C, un objet n'est donc qu'une extension évoluée des *enregistrements* (type **struct**) disposant de procédures et fonctions pour gérer les champs qu'il contient.

3.1.2 Notion de classe

On appelle classe la structure d’un objet, c’est à dire la déclaration de l’ensemble des entités qui composeront un objet. Un objet est donc « issu » d’une classe. En réalité, on dit qu’un objet est une instanciation d’une classe, c’est la raison pour laquelle on pourra parler indifféremment d’objet ou d’instance (éventuellement d’occurrence).

Une classe est composée de deux parties :

- Les attributs (parfois appelés champs, ou variables d'instances ou données membres) : il s'agit des données représentant l'état de l'objet
- Les méthodes (parfois appelées fonctions membres) : il s'agit des opérations applicables aux objets.

3.1.3 Présentation de la notion d'objet/classe

Rappelons qu'il faut bien faire la distinction entre la notion de classe et d'objet. La classe n'est que la définition d'un type d'objet mais n'est pas un objet en elle-même. Il faut ensuite créer des objets à partir de cette classe.

Une classe est un type abstrait (exemple : un compte bancaire en général). Un objet est un exemplaire concret d'une classe (exemple : le compte bancaire de XXX). Donc on peut conclure qu'un objet est une variable particulière dont le type est une classe.

En fait, on considère plus souvent que les classes sont les descriptions des objets lesquels sont des instances de leur classe. Une classe décrit la structure interne d'un objet : les données qu'il regroupe, les actions qu'il est capable d'assurer sur ses données. Un objet est un état de sa classe.

3.1.4 Diagramme de classe

Afin de faciliter la communication entre les programmeurs n'utilisant pas le même langage de programmation objet, il existe un standard de représentation graphique d'une classe. Ce standard fait partie de la norme UML (Unified Modeling Language).

Considérons par exemple la modélisation d'un véhicule telle que présentée par la figure suivante :

Véhicule	*Nom de la classe*
#Marque : Chaîne #Puissance fiscale : Entier #VitesseMaximale : Entier #VitesseCourante : Entier	*Description des attributs ou données membres*
+Créer un véhicule() +Détruire un véhicule() +Démarrer() +Accélerer(Taux : Entier) +Avancer() +Reculer()	*Description des méthodes ou fonctions membres*

Représentation de la classe Véhicule sous forme de diagramme

Quel que soit l'outil utilisé, on observe que la classe est décomposée en deux parties :

1. les champs, ou attributs qui correspondent aux informations de la classe : Marque, Puissance, ….VitesseCourante.
2. les méthodes, qui correspondent aux actions réalisables : Créer, Détruire, Démarrer….,Reculer.

Dans ce modèle, un véhicule est représenté par une chaîne de caractères (sa marque) et trois entiers : la puissance fiscale, la vitesse maximale et la vitesse courante. Toutes ces données sont représentatives d'un véhicule particulier, autrement dit, chaque objet véhicule aura sa propre copie de ses données : on parle alors d'attribut d'instance. L'opération d'instanciation qui permet de créer un objet à partir d'une classe consiste précisément à fournir des valeurs particulières pour chacun des attributs d'instance.

3.1.5 Utilisation d'une classe

On rappelle qu'une classe contient donc :

- des données-membres ou attributs.
- des fonctions-membres ou méthodes.

Le squelette d'une classe est donné par le modèle suivant :

```
class Nom_class
{
private:
     // Déclaration des attributs
    // et méthodes privés
public:
   // Déclaration des attributs
   // et méthodes publics
};
```

L'exemple suivant permet de définir la classe point aux coordonnées *x* et *y* et une fonction membre *Affiche()* qui permet de les afficher.

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

En utilisant le modèle fourni par la classe ci-dessus, il est possible de créer autant d'objets ou de points que nécessaire. Différents objets d'une même classe disposent des mêmes attributs et des mêmes méthodes, mais les valeurs des attributs sont différentes pour chaque objet.

3.1.6 Déclaration d'une méthode ou fonction membre

Une méthode représente une action ou un comportement réalisable sur un objet de la classe dans laquelle cette méthode est définie. Comme pour la classe point définie en haut, la fonction membre *Affiche()* par permet d'afficher les positions *(x,y)* de chaque point. La déclaration d'une fonction membre se réalise en deux étapes :

1. **Dans la classe elle-même :**

```
class point
{
    private :
          int x,y;
    public :
         void initialise(int a,int b);
         void deplacedroite(int dx,int dy);
         void affiche();
};
```

2. **A l'extérieur de la classe :**

```
void point::initialise(int a, int b)
{   x=a;
    y=b;
}
void point::deplacedroite(int dx,int dy)
{   x=x+dx;
    y=y+dy;
}
 void point::affiche()
{  cout<<"Le point est en position: "<<x<<","<<y<<endl;
}
```

(void) signifie que la fonction ne retourne pas et les (**::**) désigne l'opérateur de portée.

3.2 Conception d'un programme Orienté Objet

Dans cette partie, on va pouvoir écrire un programme en C++, en se basant sur la notion d'objet et de classe. L'exercice suivant utilise la classe *point*, tous les points auront des coordonnées en x et en y, mais cet emplacement sera différent pour chaque point.

Exercice 3.1:

Le but de cet exercice est d'illustrer les notions d'objet et de classe au moyen d'un exemple simple. Il s'agit ici de réaliser une classe ***point*** permettant de manipuler un point dans un plan. Cette classe comporte deux membres données privées : ***x*** et ***y*** et trois fonctions membres (ou méthodes) publiques : ***initialise, deplacedroite, affiche***.

On déclare la classe au début du programme après le mot clé **class**, puis on définit le contenu des méthodes membres.

p1 , p2 et p3 sont des objets de classe *point*, c'est-à-dire des variables de type *point*. Commencez par ouvrir un nouveau fichier source dans l'environnement de développement de C++, Entrez directement le code suivant :

```
#include <iostream>     //à ajouter pour le cin et cout
#include <conio.h>      // à ajouter pour le getch()

using namespace std ;

/*création de la classe point*/
class point
{
     private :
            int x,y;
     public :
           void initialise(int a,int b);
           void deplacedroite(int dx,int dy);
           void affiche();
};
void point::initialise(int a, int b)
{    x=a;
     y=b;
}
void point::deplacedroite(int dx,int dy)
{    x=x+dx;
     y=y+dy;
}
 void point::affiche()
{  cout<<"Le point est en position: "<<x<<","<<y<<endl;
}
int main(int argc, char* argv[])
{    point p1;                    //construction d'un point p1
     p1.initialise(10,20);        //initalisation de ce point de
                                  //coordonnées x=10, y=20
     p1.affiche();                //affichage de ce point
     p1.deplacedroite(2,3);  //déplacement de ce point à droite
     p1.affiche();                //nouvel affichage
     getch();
     return 0 ;
}
```

La définition d'une classe commence par le mot-clé « **class** ». On retrouve ensuite la définition des champs (attributs) et des méthodes de la classe. On remarque que les méthodes utilisent (et modifient) les valeurs des attributs.

3.2.1 Déclaration et instanciation

Tout objet est instance d'une classe. Dans la classe point, on remarque que la création de l'objet p1 se fait en deux étapes :

- Déclaration de l'objet ;
- Instanciation de l'objet.

La déclaration permet de créer une nouvelle variable. À ce stade, aucune réservation de mémoire n'a eu lieu pour cet objet. Il est donc inutilisable.

L'instanciation est l'opération qui consiste à créer un nouvel objet à partir d'une classe. Elle permet de réserver une zone mémoire spécifique pour l'objet. On dit que l'objet est instancié. Sans cette étape indispensable, l'objet déclaré ne peut pas être utilisé.

3.2.1.1 Instanciation d'un objet

Pour instancier un objet, on doit le déclarer d'abord, pour le déclarer il faut donner le type de l'objet. Le type d'un objet est sa classe. Etant donné une instance d'un objet, on accède à ses attributs et à ses méthodes grâce à la notation pointée ''.''.

Dans l'exercice 3.1 sur la classe *point*. On a d'abord déclaré un objet appelée p1 de type point. Puis on l'a initialisé à la position (x,y)=(10,20).

```
point p1;                //déclaration d'un nouvel objet p1
p1.initialise(10,20);    //initalisation de ce point de point à x=10 et y=20
```

L'exécution du code de l'exercice 3.1 donne les résultats suivants pour l'instanciation de l'objet p1 avant et après le déplacement à droite :

```
Le point est en position: 10,20
Le point est en position: 12,23
```

3.2.1.2 Instanciation de plusieurs objets

On peut instancier plusieurs d'objet d'une même classe Une fois les objets déclarés, autrement dit le type défini.

<u>Exercice 3.2:</u>

A travers le code fourni ci-dessus sur la conception de la classe *point*, on va pouvoir modifier le programme principal pour instancier plusieurs objets de type point :

1. Créez un objet p2 de type *point* avec les valeurs suivantes : (30,40).
2. Faites un déplacement de p2 à droite de (6,4).

3. Ajoutez une méthode *deplacegauche* qui déplace le point à gauche.
4. Faites un déplacement de p2 à gauche de (6,4).
5. Créez un objet p3 de type *point* avec les valeurs suivantes : (15,15).
6. Faites un déplacement de p3 à gauche de (5,5) et à droite de (15,15).

Solution 3.2:

```
#include <iostream>
#include <conio.h>
//------------------------------------------------------------------------
using namespace std ;

/*création de la classe point*/
class point
{
private :
   int x,y;
public :
     void initialise(int a,int b);
     void deplacedroite(int dx,int dy);
     void deplacegauche(int dx,int dy);
     void affiche(int n);
};
void point::initialise(int a, int b)
{
    x=a;
    y=b;
}
void point::deplacedroite(int dx,int dy)
{
    x=x+dx;
    y=y+dy;
}
void point::deplacegauche(int dx,int dy)
{
    x=x-dx;
    y=y-dy;
}
void point::affiche(int n)
{
    cout<<"Le point p"<<n<<" est en position: "<<x<<","<<y<<endl;
}
int main(int argc, char* argv[])
{
    point p1,p2,p3;        //déclaration d'un point p1
    p1.initialise(10,20); //initalisation de ce point de coordonnées x=10, y=20
    p1.affiche(1);
```

```
    p2.initialise(30,40);
    p2.affiche(2);          //affichage du point p2
    p3.initialise(15,15);
    p3.affiche(3);          //affichage du point p3
    p1.deplacedroite(2,3); //déplacement de p1 à droite
    p1.affiche(1);
    p2.deplacedroite(6,4);
    p2.affiche(2);              //nouvel affichage de p2
    p2.deplacegauche(6,4);
    p2.affiche(2);
    p3.deplacegauche(5,5);
    p3.affiche(3);
    p3.deplacedroite(15,15);
    p3.affiche(3);
    getch();
    return 0;
}
//-------------------------------------------------------------------------
```

Le code ci-dessus montre l'opération d'instanciation de la classe *point* en 3 objets différents : p1, p2 et p3. L'exécution sur l'environnement Dev-C++ donne les résultats suivants :

```
Le point p1 est en position: 10,20
Le point p2 est en position: 30,40
Le point p3 est en position: 15,15
Le point p1 est en position: 12,23
Le point p2 est en position: 36,44
Le point p2 est en position: 30,40
Le point p3 est en position: 10,10
Le point p3 est en position: 25,25
```

Exercice 3.3:

Il s'agit ici de définir une classe ***rectangle*** représentant une abstraction (informatique) de ce qu'est un rectangle : une largeur, une longueur, sa surface et son périmétre. Cette classe contient un emplacement ***x*** et ***y*** une ***longueur*** et une ***largeur*** comme variables, les fonctions ***initialise***, **deplace**, ***surface***, ***perimetre*** et ***affiche*** sont leurs méthodes.

- La fonction ***initialise*** fixe les valeurs initiales de ***x***, ***y***, ***largeur*** et ***longueur***.
- La fonction ***deplace*** déplace le rectangle par un décalage à droite de ***dx*** et ***dy***.
- La fonction ***surface*** renvoie la valeur la surface du rectangle.
- La fonction ***perimetre*** renvoie la valeur du périmètre du rectangle.
- La fonction ***affiche,*** affiche la position, la longueur, la largeur, la surface ainsi que le périmètre du rectangle.

En se servant du code programme de l'exercice 3.1, définissez une classe *rectangle* permettant de manipuler les fonctions définies ci-dessus. Utilisez cette classe pour instancier un objet *rec1* :

a) Crée un objet rec1 de type rectangle.
b) Initialise le rec1 par les valeurs (10,2,5,4).
c) Fait un déplacement de rec1 de (2,4).
d) Calcule la surface et le périmètre de rec1.
e) Affiche les résultats.

Solution 3.3:

```
//-------------------------------------------------------------------------
#include <iostream>
#include <conio.h>
using namespace std;

class rectangle
{
    private :
        int x,y;
        float lon,lar;
    public :
        void initialise(int a,int b, int c, int d);
        void deplace(int dx, int dy);
        float surface();
        float perimetre();
        void affiche();
};
void rectangle::initialise(int a, int b, int c, int d)
{
    x=a;
    y=b;
    lon=c;
    lar=d;
}
void rectangle::deplace(int dx,int dy)
{
    x=x+dx;
    y=y+dy;
}
float rectangle::surface()
{
    return lon*lar;
}
float rectangle::perimetre()
{
    return 2*(lon+lar);
}
```

```
void rectangle::affiche()
{
    cout<<"Le rectangle est en position: "<<x<<","<<y<<"\n"
    <<"de longueur: "<<lon<<","<<lar<<"\n"
    <<"de surface: "<<surface()<<"\n"
    <<"de perimetre: "<<perimetre()<<endl;
}
int main(int argc, char* argv[])
{
    rectangle rec1;
    rec1.initialise(10,2,5,4);
    rec1.deplace(2,4);
    rec1.surface();
    rec1.perimetre();
    rec1.affiche();
    getch();
    return 0;
}
//--------------------------------------------------------------------------
```

Après l’exécution on obtient :

```
Le rectangle est en position: 12,6
de longueur: 5,4
de surface: 20
de perimetre: 18
```

Chapitre 4

Notions de Constructeur, Destructeur

Toute classe nécessite d'être crée puis détruite (ordre logique), pour cela il faut un constructeur et un destructeur, ce sont des procédures très spéciales. Le constructeur alloue implicitement de la mémoire et permet d'initialiser les variables internes. Le destructeur quant à lui est appelé lorsque la classe va être détruite et libère ainsi la mémoire.

4.1 Les Constructeurs

Avant de pouvoir être utilisé, un objet doit être construit. Cette opération est effectuée par des méthodes spécifiques : les constructeurs. On peut en avoir plusieurs pour une même classe. C'est typiquement dans cette fonction que l'on pourra faire l'initialisation des différentes variables membres.

Tout constructeur doit répondre aux caractéristiques suivantes :

- est une fonction membre qui porte le même nom que sa classe,
- est appelé après l'allocation de l'espace mémoire destiné à l'objet,
- ne renvoie pas de valeur (pas même *void* ne doit figurer devant sa déclaration ou sa définition).
- Si aucun constructeur n'est déclaré, un constructeur par défaut, sans paramètres, est automatiquement créé.

4.1.1 Types de constructeurs

Il existe trois types de constructeurs :

1- Constructeur par défaut (sans paramètres)
2- Constructeur d'initialisation ou paramétré (avec paramètres)
3- Constructeur par copie (reçoit en paramètre un objet)

4.1.1.1 Constructeur par défaut

Appelé aussi le constructeur sans paramètres. Une classe qui ne déclare aucun constructeur explicitement en possède en fait toujours un : le constructeur vide par défaut, qui ne prend aucun paramètre, il définira les variables à 0 et les chaînes à vide. Prenons l'exemple de la classe *point* ci-dessous.

```
class point
{    private :
        int x,y;
     public :
        point();
        void affiche();
};
void point::point()
{
    x=0;
    y=0;
}
void point::affiche()
{
    cout<<"Le point est en position: "<<x<<","<<y<<endl;}
```

Un constructeur vide par défaut est créé dans cette classe, on peut donc l'instancier de la façon suivante :

```
point p1 ;
p1.affiche();
```

Vous obtenez comme résultat la fenêtre suivante :

```
Le point est en position: 0,0
```

4.1.1.2 Constructeur d'initialisation

Appelé aussi le constructeur paramétré. Si l'on définit un constructeur explicite dans cette classe (i.e. la classe *point*), alors ce constructeur vide par défaut n'existe plus (il n'est plus crée) ; on doit le déclarer explicitement si l'on veut encore l'utiliser. Si l'on reprend notre exemple :

```
class point
{    private :
        int x,y;
     public :
        point(int a, int b);
        void affiche();
};
void point::point(int a, int b)
{
    x=a;
    y=b;
}
void point::affiche()
{
    cout<<"Le point est en position: "<<x<<","<<y<<endl;}
```

On ne peut plus instancier cette classe comme précédemment. On ne peut l'instancier que de la façon suivante :

```
point p1(10,15) ;
p1.affiche();
```

Vous obtenez comme résultat après exécution la fenêtre suivante :

```
Le point est en position: 10,15
```

4.1.1.3 Surdéfinition ou surcharge de constructeurs

Les constructeurs peuvent être surchargés si la classe contient plus d'un constructeur. Si on reprend toujours notre exemple :

```
/*création de la classe point*/
class point
{
private :
   int x,y;
public :
     point();                  // constructeur par défaut
     point(int a);              // constructeur d'initialisation avec un seul paramétre
     point(int a, int b);       // constructeur d'initialisation avec deux paramétres
     void affiche(int n);
};
point::point()  //initialise x à la valeur 0 et y à la valeur 0
{
    x=0;
    y=0;
}
point:: point(int a) //initialise x à la valeur a et y à la valeur a
{
    x=a;
    y=a;
}
point:: point(int a, int b) //initialise x à la valeur a et y à la valeur b
{
    x=a;
    y=b;
}

void point::affiche(int n)
{
    cout<<"Le point p"<<n<<" est en position: "<<x<<","<<y<<endl;
}
```

On peut instancier plusieurs objets de la même classe par différents types de constructeurs de la manière suivante :

```
point p1,p2;          // appel du constructeur par défaut par les objets p1 et p2
point p3(7) ;         // appel du constructeur d'initialisation avec un seul paramétre par l'objet p3
point p4(10,15) ;   // appel du constructeur d'initialisation avec deux paramétres par l'objet p4
p1.affiche(1);
p2.affiche(2);
p3.affiche(3);
p4.affiche(4);
```

L'exécution du code source donne comme résultats :

```
Le point p1 est en position: 0,0
Le point p2 est en position: 0,0
Le point p3 est en position: 7,7
Le point p4 est en position: 10,15
```

On peut modifier le code précédent on utilisant un tableau d'une dimension. L'instanciation devient alors :

```
point p[3] = { , , point(7),point(10,15)} ;   // appel du constructeur par défaut par les deux
                                              //premiers éléments du tableau et un constructeur
                                             // d'initialisation avec un seul paramétre pour le 3éme élément
                                            // d'initialisation avec deux paramétres pour le 4éme élément
for (i=0 ;i<4 ;i++)
{p[i].affiche();}
```

Exercice 4.1:

Utilisez la classe *rectangle* de l'exercice 3.3., on utilisera cinq (5) constructeurs cette fois-ci, qui sont définis comme suit :

1. Un constructeur par défaut : *rectangle()* ;
2. Un constructeur d'initialisation ou paramétré avec la longueur donné : *rectangle(lon1)* ;
3. Un constructeur d'initialisation ou paramétré avec la largeur donné : *rectangle(lar1)* ;
4. Un constructeur d'initialisation avec deux paramètres qui sont la longueur et la largeur : *rectangle(lon1, lar1)* ;
5. Un constructeur d'initialisation avec les position x et y initialisé ainsi que les dimensions x et y : *rectangle(x1,y1,lon1, lar1)*;

Solution 4.1:

```
//--------------------------------------------------------------------------
#include <iostream>
#include <conio.h>
using namespace std;

class rectangle
{
   private :
      int x,y;
      float lon,lar;
   public :
       rectangle();
       rectangle(int);
       rectangle(int,int);
```

```
        rectangle(int,int, int);
        rectangle(int,int, int, int);
        void deplace(int dx, int dy);
        int surface();
        int perimetre();
        void affiche(int n);
};

rectangle::rectangle()
{
    x=0;
    y=0;
   lon=0;
   lar=0;
}

rectangle::rectangle(int lon1)
{
    x=8;
    y=5;
    lon=lon1;
    lar=6;
}

rectangle::rectangle(int lon1, int lar1)
{
    x=8;
    y=0;
    lon=lon1;
    lar=lar1;
}

rectangle::rectangle(int x1, int y1,int lar1)
{
     x=x1;
     y=y1;
    lon=lar1;
    lar=lar1;
}

rectangle::rectangle(int x1, int y1, int lon1, int lar1)
{
    x=x1;
    y=y1;
    lon=lon1;
    lar=lar1;
}
void rectangle::deplace(int dx,int dy)
{
     x=x+dx;
     y=y+dy;}
```

```
int rectangle::surface()
{
     return lon*lar;
}
int rectangle::perimetre()
{
     return 2*(lon+lar);
}
void rectangle::affiche(int n)
{
     cout<<"Le rectangle "<<n<<" est en position: "<<x<<","<<y<<"\n"
     <<"de longueur: "<<lon<<","<<lar<<"\n"
     <<"de surface: "<<surface()<<"\n"
     <<"de perimetre: "<<perimetre()<<endl;
}
int main(int argc, char* argv[])
{
     rectangle rec1;
     rectangle rec2(3);
     rectangle rec3(8,2);
     rectangle rec4(3,6,4);
     rectangle rec5(10,2,5,4);
     rec1.surface();
     rec1.surface();
     rec1.perimetre();
    rec1.affiche(1);
    rec2.surface();
    rec2.perimetre();
    rec2.affiche(2);
    rec3.surface();
    rec3.perimetre();
    rec3.affiche(3);
    rec4.surface();
    rec4.perimetre();
    rec4.affiche(4);
    rec5.deplace(2,4);
    rec5.surface();
    rec5.perimetre();
    rec5.affiche(5);
   getch();
   return 0;
}
//-----------------------------------------------------------------------
```

Exécution 4.1

```
Le rectangle 1 est en position: 0,0
de longueur: 0,0
de surface: 0
de perimetre: 0
Le rectangle 2 est en position: 8,5
de longueur: 3,6
de surface: 18
de perimetre: 18
Le rectangle 3 est en position: 8,0
de longueur: 8,2
de surface: 16
de perimetre: 20
Le rectangle 4 est en position: 3,6
de longueur: 4,4
de surface: 16
de perimetre: 16
Le rectangle 5 est en position: 12,6
de longueur: 5,4
de surface: 20
de perimetre: 18
```

4.1.1.4 Constructeur de copie :

Il existe un constructeur particulier, appelé constructeur de copie. Il doit effectuer une copie d'un objet du même type.

- il sert à créer une copie explicitement,
- il est aussi utilisé implicitement par le compilateur,
- comme le constructeur sans arguments, toutes les classes en ont un par défaut.

Ce constructeur par défaut recopie simplement toutes les données membres de l'objet initial transmises en arguments comme données membres du nouvel objet.

Dans l'exemple suivant, la classe point définit un constructeur de copie qui prend comme argument une instance de point. Les valeurs des propriétés de l'argument sont assignées aux propriétés de la nouvelle instance de **point**. Le code contient un autre constructeur de copie qui envoie les propriétés x et y de l'instance que vous voulez copier au constructeur d'instance de la classe.

```
class point
{
    public :
        point(int a, int b);  // constructeur d'initialisation
        void affiche();
        int x,y;
};
point:: point(int a, int b) //initialise x à la valeur a et y à la valeur b
{   x=a;
    y=b;}
```

```
void point::affiche()
{cout<<"Le point est en position: "<<x<<","<<y<<endl;}
```

L'instanciation de l'objet p2 se fait par un constructeur de copie de la façon suivante :

```
point p1(10,15) ;// création d'un objet p1 par le constructeur d'initialisation
point p2=point(p1) ;// création d'un objet p2 par le constructeur de copie

// changement des coordonnées  x et y pour chaque objet
p1.x=12 ;
p1.y=23 ;
p2.x=12 ;
p2.y=23 ;

//affichage des résultats
p1.affiche();
p2.affiche();
```

On remarque dans l'exemple en haut que les coordonnées du point *x* et *y* ne sont plus *private*, puisqu'on les modifie après les avoir construit. Le résultat de l'exécution donne :

```
Le point 1 est en position: 10,15
Le point 2 est en position: 10,15
Le point 1 est en position: 12,15
Le point 2 est en position: 10,23
```

Exercice 4.2:

Définir un constructeur de copie pour la classe « ***personne*** » qui prend comme argument une instance de personne. Les valeurs des propriétés de l'argument sont assignées aux propriétés de la nouvelle instance de personne. Cette classe comporte deux membres données privées : ***nom*** et ***age.*** Et une fonction membre ***affiche***.

- Créer une personne p1 par le constructeur d'initalisation .
- Créer deux personnes p2 et p3 par le constructeur de copie.
- Modifier l'âge pour les deux personnes p1, p2 et p3.
- Afficher les résultats.

Solution 4.2

```
#include <iostream>      //à ajouter pour le cin et cout
#include <conio.h>       // à ajouter pour le getch()

using namespace std ;

/*création de la classe personne*/
class personne
{
private :
    string nom;

public :
     personne(string);  // constructeur d'initialisation
     void affiche(int);
     int age ;
};
personne::personne(string nom1) //initialise nom à la valeur nom1
{
nom=nom1;
}
void personne::affiche(int n)
{
cout<<nom<<n<<" est âgé de "<<age<<" ans"<<endl;
}

int main(int argc, char* argv[])
{
personne p1("Nabil") ;
personne p2=personne(p1);
personne p3=personne(p1);

// changement de l'âge pour chaque personne crée
// On suppose qu'il porte tous le nom Nabil mais l'âge peut être différent
p1.age=18 ;
p2.age=17 ;
p3.age=15 ;

//affichage des résultats
p1.affiche(1);
p2.affiche(2);
p3.affiche(3);
getch();
return 0 ;
}
```

Exécution 4.2

Le code contient un autre constructeur de copie qui envoie la propriété nom de l'instance qu'on veut copier au constructeur d'instance de la classe.

Après l'instanciation, selon le code fournit, on obtient :

```
Nabil1 est ôgú de 18 ans
Nabil2 est ôgú de 17 ans
Nabil3 est ôgú de 15 ans
```

4.2 Le destructeur

Lorsqu'un objet devient inutile, il doit être détruit. Cette opération est réalisée par le destructeur. Les caractéristiques d'un destructeur sont les suivants :

- Il est unique.
- est une fonction membre portant le même nom que sa classe, précédé du symbole (~),
- est appelé avant la libération de l'espace mémoire associé à l'objet
- ne peut pas comporter d'arguments et il ne renvoie pas de valeur (aucune indication de type ne doit être prévue).
- Il doit restituer la place mémoire allouée dynamiquement par l'objet.

Le destructeur est une méthode particulière qui est définie implicitement pour tous les objets. Par défaut il ne fait rien.

Exercice 4.3

Transformez le code précédent en utilisant un constructeur pour initialiser et un destructeur pour détruire chaque objet crée. Utilisez les deux méthodes ci-dessous :

rectangle::rectangle(int a, int b, float c, float d)
//constructeur
{ x=a;
y=b;
lon=c;
lar=d;
ctr++;
}
	rectangle::~rectangle() // destructeur
	{
	ctr-- ;
	cout << "Nombre d'objets rectangle restants : " << ctr << endl ;
	}

Après l'implémentation de ces deux méthodes, utilisez le constructeur pour instancier un objet *rec2* de type *rectangle* :

a) Ajoutez un rec2(20,4,10,8).
b) Faites un deplacement de rec2(6,3).
c) Calculez la surface et le périmètre de rec2.
d) Affichez les résultats.
e) Détruisez seulement le rec2.

Solution 4.3

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

using namespace std ;

class rectangle
{
    private :
        int x,y;
        float lon,lar;
        static int ctr ; // compteur d'objets
    public :
         rectangle (int,int,float,float); // Constructeur
         void deplace(int dx, int dy);
         float surface();
         float perimetre();
         void affiche(int);
         ~rectangle(); // Destructeur
};

int rectangle::ctr = 0; // init. A 0 du nb d'objets rectangle

rectangle::rectangle(int a, int b, float c, float d) //constructeur
{
      x=a;
      y=b;
      lon=c;
      lar=d;
     ctr++;
}
void rectangle::deplace(int dx,int dy)
{
    x=x+dx;
    y=y+dy;
}

float rectangle::surface()
{
    return lon*lar;
}

float rectangle::perimetre()
{
```

```
    return 2*(lon+lar);
}

void rectangle::affiche(int n)
{
     cout<<"Le rectangle "<<n<<" est en position: "<<x<<","<<y<<"\n"
    <<"de longueur: "<<lon<<","<<lar<<"\n"
    <<"de surface: "<<surface()<<"\n"
    <<"de perimetre: "<<perimetre()<<endl;
}

rectangle::~rectangle() // destructeur
{
    ctr-- ;
    cout << "Nombre d'objets rectangle restants : " << ctr << endl ;
}

int main(int argc, char* argv[])
{
    rectangle rec1(10,2,5,4);   //construction d'un rectangle rec1
    rectangle rec2(20,4,10,8); //construction d'un rectangle rec2
    rec1.deplace(2,4);
    rec2.deplace(6,3);
    rec1.surface();
    rec2.surface();
    rec1.perimetre();
    rec2.perimetre();
    rec1.affiche(1);
    rec2.affiche(2);
    rec2.~rectangle();       //destruction du rectangle rec1
    getch();
    return 0;
}
```

Remarque : l'attribut statique *(ctr)* dans le code fourni ci-dessus est un membre de la classe *rectangle* qui est commun (c'est à dire partagé) à toutes les instances d'une classe.

Après l'exécution du code on obtient :

```
Le rectangle 1 est en position: 12,6
de longueur: 5,4
de surface: 20
de perimetre: 18
Le rectangle 2 est en position: 26,7
de longueur: 10,8
de surface: 80
de perimetre: 36
Nombre d'objets rectangle restants : 1
```

4.3 L'encapsulation

En P.O.O les données sont encapsulées, ce qui signifie que leurs accès ne peuvent se faire que par le biais des méthodes. La déclaration d'une classe précise quels sont les membres (données ou fonctions) publics (c'est-à-dire accessibles à l'utilisateur de classe), on utilise dans ce cas le mot *public* et quels sont les membres privés (inaccessibles par l'utilisateur de la classe), on utilise dans ce cas le mot *private*.

4.3.1 Différents niveaux d'accessibilité des propriétés et méthodes

Il existe trois catégories définissant les restrictions d'accès aux fonctions et variables, ceci renforce l'encapsulation ou le masquage des données. Ces catégories sont définies par les mots-clés *public, private* ou *protected.*

4.3.1.1 Private

On ne peut accéder aux données et fonctions private que de l'intérieur de l'objet. Cela implique qu'une fonction *private* ne peut être appelée qu'à partir des fonctions membres de la classe, et qu'une donnée membre *private* ne peut être lue ou modifiée que par des fonctions membres de la classe.

4.3.1.2 Public

Pour les données et variables membres *public*, il n'y a pas de restriction et on peut y accéder à l'extérieur comme à l'intérieur de l'objet.

4.3.1.3 Protected

Il s'agit d'un cas intermédiaire entre *private* et *public*. Les fonctions et variables membres de type *protected* sont accessibles par toute fonction membre de l'objet ou d'une des classes dérivées de l'objet. La notion de classe dérivée se reportera au chapitre 5 traitant des notions d'héritage et de polymorphisme.

4.3.2 Principe de l'encapsulation

Chaque membre se voit définir un droit d'accès par l'un des mots-clés *public, private* ou *protected*. Seuls les membres *publics* sont accessibles de l'extérieur de la classe. Il est à noter que si une catégorie ne comporte ni variable, ni fonction, il n'est pas indispensable de faire figurer le mot-clé correspondant, comme dans l'exemple suivant :

class point // **class** : mot réservé et **point** : nom de la classe

```
{
    private :
        int x ;     // champ membre privé de la classe
        float y ; // champ membre privé de la classe
        void g( ) ; // fonction membre privée
    public :
        void f( ) ; // fonction membre public } ;
}
```

Ainsi dans l'exemple, la catégorie *protected* ne figure pas, on a supprimé le mot-clé *protected*.

4.3.3 Importance de l'encapsulation

Lorsque l'on crée une nouvelle classe, il faut toujours se demander quelles propriétés et méthodes vont devoir ou pouvoir être accessibles depuis l'extérieur de la classe.

En effet, parfois, nous aurons besoin d'avoir accès à des méthodes ou propriétés depuis l'extérieur d'une classe afin qu'un script ou programme fonctionne correctement.

A l'inverse, il nous faudra dans d'autre cas interdire l'accès à certaines propriétés ou méthodes depuis l'extérieur de la classe sous peine d'ouvrir notre script à de sérieux problèmes de vulnérabilité ou failles.

Par exemple, si vous créez un logiciel et que vous laissez la possibilité à des développeurs externes d'étendre vos classes, il faudra vous demander à chaque fois si ces développeurs vont avoir besoin d'accéder à telle ou telle propriété ou méthode et jauger les problèmes potentiels que ça peut poser.

Exercice 4.4

Réalisez une classe « ***carre*** » analogue à la classe « *rectangle* », mais ne comportant pas de fonction *affiche*. Pour respecter le principe **d'encapsulation de données,** prévoyez trois fonctions membres *publiques* (nommées *positionx, positiony, et cote*) fournissant en retour respectivement la position en x, la position en y et le côté du carré. Adaptez le programme de la solution de l'exercice 4.3 pour qu'il fonctionne avec cette nouvelle classe. Utilisez un constructeur et un destructeur pour créer et détruire les objets carré.

Solution 4.4

```
#include <iostream>
#include <conio.h>
using namespace std;
class carre
{
    private :
        int x,y;
        float cot;
        static int ctr ; // compteur d'objets
    public :
        carre (int,int,float); // Constructeur
        void deplace(int dx, int dy);
        float surface();
        float perimetre();
        float positionx(); //position en x de l'objet
        float positiony(); //position en y de l'objet
        float cote();        // le côté du carré
        ~carre();            // Destructeur
};
int carre::ctr = 0; // init. A 0 du nb d'objets carre
carre::carre(int a, int b, float c) //Constructeur
{
    x=a;
    y=b;
    cot=c;
    ctr++;
}
void carre::deplace(int dx,int dy)
{
    x=x+dx;
    y=y+dy;
}
float carre::surface()
{
    return cot*cot;
}
float carre::perimetre()
{
    return 4*cot;
}
float carre::positionx()
{
    return x;
}
float carre::positiony()
{
    return y;
}
float carre::cote()
```

```
{
    return cot;
}
carre::~carre() // destructeur
{
    ctr-- ;
    cout << "Nombre d'objets carre restants : " << ctr << endl ;
}
int main(int argc, char* argv[])
{
    carre carre1(10,2,5); //construction d'un carre carre1
    carre carre2(20,4,10); //construction d'un carre carre2
    carre1.deplace(2,4);
    carre2.deplace(6,3);
    carre1.surface();
    carre2.surface();
    carre1.perimetre();
    carre2.perimetre();
   // affichage de carre1
    cout<<"Le carre1 est en position: "<<carre1.positionx()<<","<<carre1.positiony()<<"\n"
    <<"de cote: "<<carre1.cote()<<"\n"
    <<"de surface: "<<carre1.surface()<<"\n"
    <<"de perimetre: "<<carre1.perimetre()<<"\n";
  // affichage de carre 2
    cout<<"Le carre2 est en position: "<<carre2.positionx()<<","<<carre2.positiony()<<"\n"
    <<"de cote: "<<carre2.cote()<<"\n"
    <<"de surface: "<<carre2.surface()<<"\n"
    <<"de perimetre: "<<carre2.perimetre()<<"\n";
    carre1.~carre(); //destruction carre1
    carre2.~carre(); //destruction carre2
    getch();
    return 0;
}
```

Les résultats après l'exécution du code de l'exercice 4.4 est le suivant :

```
Le carre1 est en position: 12,6
de cote: 5
de surface: 25
de perimetre: 20
Le carre2 est en position: 26,7
de cote: 10
de surface: 100
de perimetre: 40
Nombre d'objets carre restants : 1
Nombre d'objets carre restants : 0
```

Chapitre 5

Héritage et Polymorphisme

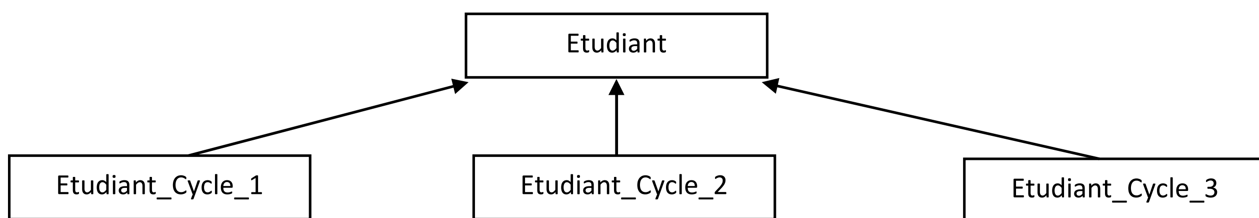
En POO, il existe une particularité dans la façon d'organiser ses classes : l'héritage de propriétés. L'objectif est de construire de nouvelles classes en réutilisant des attributs et des méthodes de classes déjà existantes. Afin de gagner en généricité, on peut appliquer un même code à des objets de types différents, lorsque les classes de ces objets sont liées par héritage. C'est le principe du polymorphisme.

5.1 Héritage

L'héritage est l'outil qui permet d'exprimer entre deux classes A et B une relation de type "est un". Plutôt que de réimplémenter des fonctionnalités existantes déjà dans une classe A. Une classe B peut intégrer des données et fonctions membres de la classe A. On dit que B hérite de A ou B spécialise A, et A généralise B.

5.1.1 Graphe de l'héritage

Pour bien comprendre le concept de l'héritage, considérons le graphe de l'héritage dans l'exemple suivant :



Graphe de l'héritage

Dans ce type de graphique :

- La classe *Etudiant* est la classe de base (classe supérieure),
- Les classes *Etudiant_Cycle_1*, *Etudiant_Cycle_2* et *Etudiant_Cycle_3* sont ses classes dérivées (ou sous classes).

Voici quelques exemples d'héritage :

- Une fleur est une plante (une fleur hérite de plante) ;
- un tram est un véhicule (Tram hérite de véhicule) ;
- un chat est un animal (chat hérite d'animal);
- une fourmi est un insecte (fourmi hérite d'insecte) ;
- etc.

L'héritage peut être simple (une seule classe de base directe) ou multiple (héritage de plusieurs classes), par exemple, on peut dire qu'une rose est une fleur, et qu'une fleur est une plante.

L'héritage multiple doit être utilisé à bon escient. En effet, le programme se complique à mesure du développement de la hiérarchie des classes et le débogage des méthodes d'objets dérivés peut être difficile. L'héritage multiple est rarement supporté. Dans tout ce qui suit, on prend en considération l'héritage simple.

5.1.2 Propriétés générales de l'héritage

L'héritage établit une relation de généralisation-spécialisation qui permet d'hériter dans la déclaration d'une nouvelle classe (appelée classe dérivée, classe fille, classe enfant ou sous-classe) des caractéristiques (propriétés et méthodes) de la déclaration d'une autre classe (appelée classe de base, classe mère, classe parent ou super-classe).

- Une classe B qui hérite d'une classe A dispose implicitement de tous les attributs et de toutes les méthodes définis dans A.

- Les attributs et les méthodes définis dans B sont prioritaires par rapport aux attributs et aux méthodes de même nom définis dans A.

On peut dire qu'une classe dérivée modélise un cas particulier de la classe de base et, est donc enrichie d'informations supplémentaires.

5.1.3 Propriétés des classes dérivées et Syntaxe de l'héritage

La classe dérivée possède les propriétés suivantes :

- Les attributs et les méthodes peuvent être modifiés au niveau de la sous-classe.
- Il peut y avoir des attributs et/ou des méthodes supplémentaires dans une sous-classe.

On rappelle que nous ne traitons ici que l'héritage simple. En langage C++, l'héritage est donné par la syntaxe suivante :

```
class classe_dérivée:protection classe_de_base {/* etc. */}
```

Les types de protections sont : « *public* », « *protected* », ou « *private* ». L'héritage public en C++ est le plus utilisé.

Si on prend l'exemple précédent de la classe Etudiant. On dit qu'un *Etudiant_Cycle_1* est un *Etudiant*. Cela signifie que *Etudiant_Cycle_1* hérite de *Etudiant*. On écrit la syntaxe comme suit :

```
class  Etudiant_Cycle_1: public Etudiant
```

Pour qu'une classe fille puisse hériter des propriétés de la classe mère, il faut que les propriétés de la classe mère possèdent des attributs de visibilité compatibles. Les notions du concept de l'héritage sont résumées à travers l'exemple général suivant :

```
// La classe A est la classe de base
class A
{
   public:
       A();
       ~A();
       void f();
       void g();
       int n;
   private:
       int p;
};
```

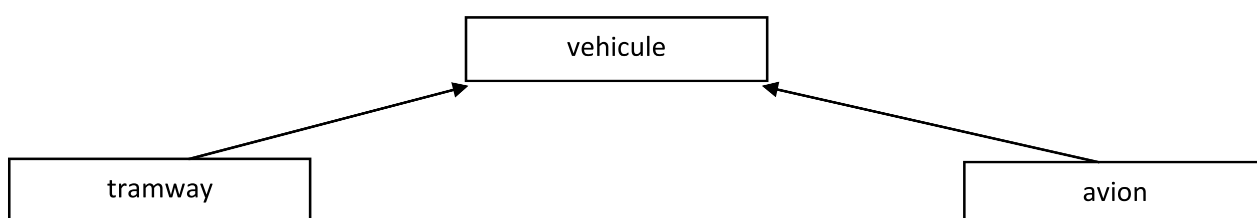
```
// La classe B hérite de la classe A
class B : public A //// heritage public
{
       public:
           B();
           ~B();
           void h();
       private:
           int q;
};
```

La classe fille B peut être instanciée :

```
int main()
{
   B b;        //instaciation de l'objet b de la classe B
   b.h();     // OK, fonction membre de B
   b.f();     // OK, fonction membre (public) héritée de A
   b.n = 3;  // OK, donnée membre (public) héritée de A
   b.p = 5;  // ERREUR, donnée membre (private) héritée de A
             // interdit aussi dans une fonction membre de B.
}
```

Exercice 5.1

Soit le graphe de l'héritage suivant. Les classes dérivées *tramway* et *avion* modélisent un cas particulier de la classe de base qui est *vehicule*.



Graphe de l'héritage

En se basant sur ce graphe, écrire un programme C, permettant de modéliser la gestion des véhicules en introduisant le concept de l'héritage. Un *véhicule* est caractérisé par ses données membres privés qui sont la vitesse du véhicule (*vitesse*) et le nombre de passagers (*nbr_passager*), ainsi qu'une fonction membre (*affiche*) qui permet d'afficher ses données privés. La classe dérivée a*vion* est caractérisée par sa donnée privée qui est le nombre de moteurs (*nbr_moteur*), et la classe dérivée *tramway* est caractérisée par sa donnée privée qui est le nombre de portes (*nbr_porte*).

Solution 5.1

On déclare d'abord la classe de base *Vehicule* puis, à partir de celle-ci, on déclare les classes dérivées *Tramway* et *Avion* qui héritent donc des caractéristiques de la classe de base *Vehicule*.

```
#include <iostream>      //à ajouter pour le cin et cout
#include <conio.h>       // à ajouter pour le getch()

using namespace std ;

/*création de la classe vehicule*/

class vehicule    // classe de base
{
   private :
      double vitesse ;
      int nbr_passager;

   public :
      vehicule (double v, int p) ;
      void affiche(string) ;

};
vehicule::vehicule(double v, int p)        //constructeur d'initialisation
{
   vitesse=v;
   nbr_passager=p;
}

void vehicule::affiche(string a)            // affichage des membres private
{
    cout<<"vitesse de "+a<<" est de "<<vitesse<<" Km/h et le nombre passagers est de:
"<<nbr_passager<<endl;
}

//tramway hérite publiquement de vehicule
class tramway:public vehicule
{
    private:
        int nbr_porte;
    public:
        tramway(int np, double v, int nbr_passager);
};

//définition du constructeur de la classe fille tramway
//avec appel du constructeur de la classe mère vehicule

tramway::tramway(int np, double v, int nbr_passager):vehicule(v,nbr_passager)
{
```

```
  nbr_porte=np;
}
//moto hérite publiquement de vehicule
class avion:public vehicule
{
          private:
                    int nb_moteur;
   public:
               avion(int nm,double v, int np);
};

//définition du constructeur de la classe fille avion
//avec appel du constructeur de la classe mère vehicule

avion::avion(int nm,double v, int np):vehicule(v,np)
{
  nb_moteur=nm;
}

int main()
{
  tramway T4000(1,70,253);
  avion AirbusA320(2,850,174);
  T4000.affiche("T4000");
  AirbusA320.affiche("AirbusA320");
  getch();
  return 0;
}
```

Remarque importante : La définition du constructeur de la classe fille se fait avec appel du constructeur de la classe mère, mais la définition du destructeur de la classe fille se fait avec appel automatique du destructeur de la classe mère.

L'exécution du code C donne les résultats suivants :

```
vitesse de T4000 est de 70 Km/h et le nombre passagers est de: 253
vitesse de AirbusA320 est de 850 Km/h et le nombre passagers est de: 174
```

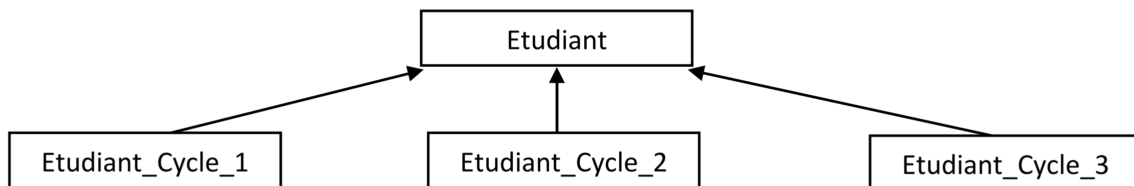
On remarque aussi, que la méthode *affiche* de la classe de base *vehicule* n'affiche que les membres privés de sa classe. Donc le nombre de moteur ne peut être affiché par celle-ci. Pour faire usage à ceci, on doit définir dans chaque classe dérivée une méthode *affiche* dont le rôle est d'afficher les membres privés des classes dérivées. On parle alors de surcharge ou surdéfinition des méthodes ou des fonctions membres.

Exercice 5.2

Objectifs de l'exercice :

1. Définir les propriétés et méthodes d'une classe
2. Définir des constructeurs et destructeurs.
3. Créer une instance de classe
4. Appliquer des méthodes
5. Procéder à l'héritage.

Le graphe de l'héritage suivant montre la classe de base *Etudiant* avec ses trois classes dérivées : *Etudiant_Cycle1*, *Etudiant_Cycle2* et *Etudiant_Cycle3*.



Graphe de l'héritage de la classe *Etudiant*

On veut mettre en œuvre la notion de l'héritage pour la classe *Etudiant* par la construction de la classe dérivée *Etudiant_Cycle1* en programmation orientée objet. Le diagramme de classe suivant, montre les données et les fonctions membres des deux classes.

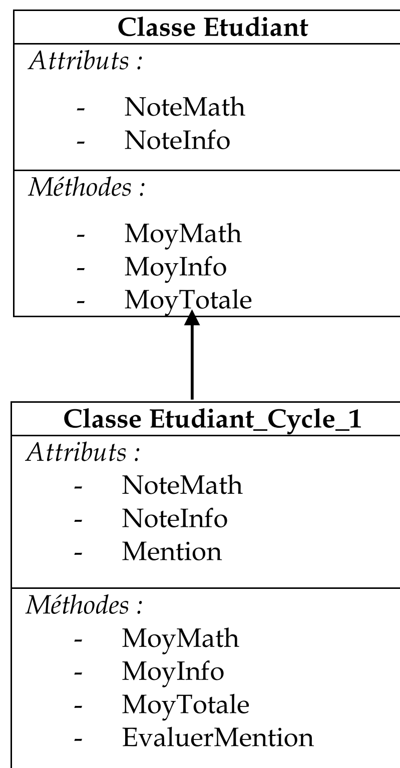


Diagramme de classe pour la classe mère *Etudiant*
et la classe fille *Etudiant_cycle_1*

Etudiant_Cycle_1 hérite de *Etudiant*. Tout se passe comme si toute la classe *Etudiant* était recopiée dans la sous-classe *Etudiant_cycle_1*. La nouvelle classe dispose d'un attribut supplémentaire (*Mention*) et d'une méthode supplémentaire (*EvaluerMention*) définie comme suit :

MoyTotale=(NoteMath+ NoteInfo)/2
EvaluerMention = ''Bien'' si MoyTotale >=14
EvaluerMention = ''ABien'' si 11< MoyTotale <14
EvaluerMention = ''Passable'' si MoyTotale <=14

En utilisant les concepts d'héritages, implémentez un code en C++ permettant la modélisation de la classe *Etudiant,* qui répond à chaque question demandée.

Partie A :

- Créer une classe *Etudiant* avec les attributs (*NoteMath*, *NoteInfo*).
- Ajouter un constructeur et un destructeur de cette classe.
- Définir les deux méthodes *MoyTotale,* et *affiche* qui affiches les attributs *NoteMath, NoteInfo* ainsi que la moyenne totale *MoyTotale*.
- Instancier cette classe Etudiant pour créer un objet étudiant (*E1_1*) ayant les notes 10 et 13 affectées respectivement aux attributs *NoteMath* et *NoteInfo*.
- Calculer sa moyenne totale *MoyTotale*.
- Afficher (*E1_1*) avec tous ses attributs. Ensuite le détruire.

Partie B :

- Créer une classe *Etudiant_Cycle_1* qui hérite de la classe *Etudiant* avec un nouvel attribut (*Mention : string*).
- Ajouter la méthode *EvaluerMention*.
- Créer un objet Etudiant_Cycle_1 (*EC1_1*) et refaire les questions 1 et 2 de la partie A.
- Evaluer sa mention.
- Afficher (*EC1_1*) avec tous ses attributs.

Exercice 5.3

On souhaite modéliser la gestion des comptes bancaires d'un établissement financier grâce à la programmation orientée objet.

Un compte bancaire est caractérisé par son numéro, sa date d'ouverture, son solde. Outre la création et la fermeture d'un compte, on peut effectuer des versements et des retraits.

Le versement consiste à ajouter un montant positif au solde du compte. Le retrait consiste à retirer un montant positif au solde du compte. Le solde résultant ne doit en aucun cas être inférieur à zéro.

Un compte d'épargne est un compte bancaire disposant d'une rémunération mensuelle fixée par un taux d'intérêts et ayant notamment une méthode permettant d'ajouter les intérêts au compte d'épargne à chaque fin de mois.

Questions :

1. Créer une classe *CompteBancaire* avec les attributs (*numéro : entier, date d'ouverture : date, solde : réel)*.
2. Ajouter un constructeur de cette classe et l'initialiser (*numéro, date_d'ouverture , solde*). La date d'ouverture est initialisée à la date du système et le solde à 0.
3. Ajouter un destructeur.
4. Ajouter la méthode versement (*versement (float montant)*).
5. Ajouter la méthode retrait *(retrait(float montant)),* le montant doit être inférieur au solde.
6. Ajouter la méthode *affiche()* qui affiche l'état du compte bancaire après chaque opération effectuée sur celui-ci.
7. Créer une classe *CompteEpargne* qui hérite de la classe *CompteBancaire,* qui définit un nouvel attribut (*taux : real*).
8. Ajouter un constructeur de cette classe et l'initialiser : *CompteEpargne(int num,float t):CompteBancaire(num).*
9. Ajouter la méthode *InteretsMensuels()* qui calcule les intérêts comme suit : (*solde*taux*) et les ajouter au *solde* de ce compte.
10. Dans le programme principal, créer l'objet *CompteBancaire1* qui appartient à la classe *CompteBancaire* et l'objet *compteEpargne1* qui appartient à la classe *CompteEpargne*.
11. Initialiser l'objet *CompteBancaire1* avec la valeur *12* affectée à l'attribut *numero.*
12. Verser la somme *800 Da* au solde de l'objet *CompteBancaire1*.
13. Retirer la somme *100 Da* du solde de l'objet *CompteBancaire1*.
14. Initialiser l'objet *CompteEpargne1* avec la valeur *10* affectée à l'attribut *numero* et la valeur *0.05* à l'attribut *taux*.
15. Calculer le nouveau solde de l'objet après ajout des intérêts *CompteEpargne1*.
16. Détruire les deux objets.

N'oublier pas d'afficher l'état du compte après chaque opération effectuée.

Solution 5.3

```
#include <iostream>
#include <conio.h>
#include <time.h>
using namespace std;

class CompteBancaire
```

```
{
    private :
        float solde;
        int numero ;
        time_t date_ouverture;
    public :
        CompteBancaire(int num); // Constructeur
        float versement(float montant); // ramène le montant du solde après le versement d'une
                                                    //somme
        float retrait(float montant); // ramène le montant de ce qui a été retiré
        void affiche();
        float Lesolde();
        ~CompteBancaire(); // Destructeur
};
CompteBancaire::CompteBancaire(int num)
{
    solde=0;
    numero=num;
    date_ouverture= time(NULL);
}
float CompteBancaire::versement(float montant)
{
    if (montant>0) {solde=solde+montant;}
    else cout<<"Le montant doit être positif"<<endl;
    return solde;
}
float CompteBancaire::retrait(float montant)
{
    if (montant<=0 ) {cout <<"le montant doit etre positif"<<endl;}
    else if (montant > solde)
    {cout<<"Vous n avez que "<<solde<<" Dinars sur votre compte. Seule cette somme vous est
versée" <<endl;  solde=0; }
    else  {solde=solde-montant;}
   return solde;
}
void CompteBancaire::affiche()
{
    cout<<"numero de compte bancaire= "<<numero<<" date de creation
"<<ctime(&date_ouverture)<<endl;
    cout<<" le solde est = "<<solde<<endl;
}
float CompteBancaire::Lesolde()
{
    return solde;
}
CompteBancaire::~CompteBancaire()
{
    cout<<"Le compte est cloturé. Il vous est restitué "<<solde<<" Dinars"<<endl;
    cout<<" et le solde est à 0 Dinar"<<endl;
}
// Compte d'épargne
```

```
class CompteEpargne :public CompteBancaire
{    private :
        float taux;
     public :
        CompteEpargne(int,float);
        float interetMensuel();
        ~CompteEpargne();
};
CompteEpargne::CompteEpargne(int num,float t):CompteBancaire(num)
{
    taux=t;
}
float CompteEpargne::interetMensuel()
{
    return CompteBancaire::versement(taux*Lesolde());
}
CompteEpargne::~CompteEpargne()
{    cout<<"Le compte est cloturé. Il vous est restitué "<<Lesolde()<<" Dinars"<<endl;
     cout<<" et le solde est à 0 Dinar"<<endl;}
// programme principal
int main()
{    CompteBancaire CompteBancaire1(12); // création de l'objet CompteBancaire1
     CompteBancaire1.affiche();
// versement dans votre compte bancaire
     CompteBancaire1.versement(800);
     cout<<" l'etat du compte bancaire apres versement : "<<endl;
     CompteBancaire1.affiche();
// retrait d'une somme de votre compte bancaire
     CompteBancaire1.retrait(100);
     cout<<" l'etat du compte bancaire apres retrait : "<<endl;
     CompteBancaire1.affiche();
//Gestion du compte d'epargne
// création de l'objet CompteEpargne1
     CompteEpargne CompteEpargne1(10,0.05);
// versement dans votre compte d'epargne
     CompteEpargne1.versement(450);
// Afficher l'etat du compte d'epargne
     cout<<" l'etat du compte d'epargne apres versement : "<<endl;
     CompteEpargne1.affiche();
// calcul des interets
     CompteEpargne1.interetMensuel();
     cout<<" l'etat du compte apres l'ajout des interets : "<<endl;
     CompteEpargne1.affiche();
//CompteEpargne1.~CompteEpargne();
     cout<<" Destruction du compte bancaire : "<<endl;
     CompteBancaire1.~CompteBancaire();
     cout<<" Destruction du compte d'epargne : "<<endl;
     CompteEpargne1.~CompteEpargne();
     getchar();
     return 0;
}
```

L'exécution du code donne les résultats finals suivants :

```
 le solde est = 0
 l'etat du compte bancaire apres versement :
numero de compte bancaire= 12 date de creation Mon Apr 02 19:17:39 2018

 le solde est = 800
 l'etat du compte bancaire apres retrait :
numero de compte bancaire= 12 date de creation Mon Apr 02 19:17:39 2018

 le solde est = 700
 l'etat du compte d'epargne apres versement :
numero de compte bancaire= 10 date de creation Mon Apr 02 19:17:39 2018

 le solde est = 450
 l'etat du compte apres l'ajout des interets :
numero de compte bancaire= 10 date de creation Mon Apr 02 19:17:39 2018

 le solde est = 472.5
 Destruction du compte bancaire :
Le compte est cloturú. Il vous est restituú 700 Dinars
 et le solde est ó 0 Dinar
 Destruction du compte d'epargne :
Le compte est cloturú. Il vous est restituú 472.5 Dinars
 et le solde est ó 0 Dinar
Le compte est cloturú. Il vous est restituú 472.5 Dinars
 et le solde est ó 0 Dinar
```

5.2 Polymorphisme

Le terme polymorphisme vient du grec, est composé de deux mots : *poly* signifie *plusieurs* et *morphisme* signifie *forme*. Le polymorphisme traite de la capacité de l'objet à posséder *plusieurs formes*. Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, comme on le sait déjà, un objet va hériter des champs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire, ou de la compléter. Autrement dit, le polymorphisme permet à une méthode d'adopter plusieurs formes sur des classes différentes.

Pour bien comprendre la puissance du polymorphisme, nous allons traiter un exemple sur la gestion d'un garage de véhicules. Ce garage comporte des voitures et des motos. On doit déclarer ainsi la classe de base qui est *Vehicule* et ses deux classes dérivées qui sont : *Voiture* et *Moto*.

```
#include <iostream>     //à ajouter pour le cin et cout
#include <conio.h>      // à ajouter pour le getch()

using namespace std ;

/*création de la classe vehicule*/
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
};
```

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

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

   return 0;
}
```

Dans ce code, chaque classe affiche un message différent, l'exécution de ce code donne :

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

Jusqu'ici on n'a utilisé que le principe de l'héritage vu précédemment. L'héritage nous permet de réutiliser le code écrit pour la classe de base dans les autres classes de la hiérarchie des classes de notre application.

Le polymorphisme rendra possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérachie des classes. Pour voir cette différence. Créons une fonction supplémentaire qui reçoit en paramètre un *Vehicule* et modifions le main() afin d'utiliser cette fonction. Le code devient alors :

```
#include <iostream>     //à ajouter pour le cin et cout
#include <conio.h>      // à ajouter pour le getch()

using namespace std ;

/*création de la classe vehicule*/
```

```
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
};

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;
}
void presenter(Vehicule v)  //Présente le véhicule passé en argument
{
   v.affiche();
}
int main()
{
   Vehicule v;
   presenter(v);
   Moto m;
   presenter(m);
   return 0;}
```

Les messages affichés après exécution deviennent :

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

En C++, on peut manipuler tout objet de type B soit comme un objet de type B, soit comme un objet de type A où A est une classe mère de B. Si on dispose d'une collection d'objets dont les types d'instanciation sont des sous-classes d'une classe A, on peut manipuler tous ces objets de façon uniforme en les considérant comme des objets de type A.

Pour ce dernier code fournit comme exemple, 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. Comme il y a une relation d'héritage. La fonction *presenter()* reçoit en argument un *Vehicule*. Donc, à l'intérieur de la fonction, on manipule un Vehicule. Peu importe sa vraie nature. Le compilateur va donc appeler la méthode *affiche()* de la classe mère «*Vehicule*» et pas la classe fille «*Moto*». C'est le type de la variable qui détermine quelle fonction membre appeler.

Pour l'exemple de la gestion du garage, afin de changer le comportement de la fonction *presenter()* lors de l'exécution, c'est-à-dire, le programme doit utiliser la bonne version de la méthode *affiche()* et différencie entre l'objet de type mère et l'objet de type fille. On peut alors spécialiser certains comportements suivant le type d'instanciation de chaque objet. On parle alors de *polymorphisme*.

En C++, le polymorphisme est mis en œuvre par l'utilisation de deux moyens différents :

- références ou pointeurs ;
- méthodes ou fonctions virtuelles.

5.2.1 Désignation d'un objet à l'aide d'une référence

Tout objet d'une classe dérivée peut être traité et utilisé comme un objet de sa classe de base ou sa classe mère. Si B hérite de A, on peut ainsi affecter un objet de type B à un objet de type A, passer un objet de type B par valeur ou référence dans une fonction ou méthode qui attend un argument de type A. On peut aussi utiliser un pointeur vers A pour pointer vers un objet instancié en tant que B.

Pour fixer les idées, reprenons le même code pour la gestion du garage en réécrivant la fonction *presenter()* avec comme argument une référence :

```
void presenter(Vehicules const& v)  //le véhicule passé comme argument une référence
{
   v.affiche();
}

int main()  //on a gardé le même main()
{
   Vehicules v;
   presenter(v);

   Motos m;
   presenter(m);

   return 0;
}
```

Après l'exécution on obtient.

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

Du fait de son caractère polymorphe, la bonne version de la méthode *affiche()* est invoquée au moment de l'exécution. On peut avoir le même comportement en utilisant des pointeurs.

5.2.2 Déclaration de méthodes virtuelles

Une opération substituée (surchargée) peut donc prendre plusieurs formes ou implémentation en fonction du type d'objet auquel elle s'applique. L'opération est dite alors polymorphe. En C++, on parle alors de méthode virtuelle (méthode à laquelle on rajoute le mot-clé *virtual* dans la définition).

Considérons toujours l'exemple de la gestion du garage. Il suffit d'ajouter le mot-clé *virtual* dans le prototype de la classe (dans le *Vehicule.h*). Cela donne :

```
// le fichier Vehicule.h qui est sauvegardé à part

#include <iostream>
#include <conio.h>
using namespace std ;
class Vehicule
{
   public:
   Vehicule(int prix);          //Construit un véhicule d'un certain prix
   virtual void 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
   virtual void 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
   virtual void affiche() const;
   virtual ~Moto();

   private:
   double m_vitesse;
};
Vehicule::Vehicule(int prix):m_prix(prix)
{}

void Vehicule::affiche() const
{
   cout << "Ceci est un vehicule coutant " << m_prix << " DA." << 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 vaut " << m_prix << " DA." << 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 << " DA." <<
endl;
}

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

Attention ! une particularité du C++ impose que toute classe possédant au **moins une méthode virtuelle** doit également avoir un **destructeur virtuel**.

Il n'est pas nécessaire de mettre « «*virtual* » devant les méthodes des classes filles. Elles sont automatiquement virtuelles par héritage. On remarque que les constructeurs et les destructeurs sont à vide, il n'y a pas de soucis.

Rien n'a changé dans le main, on l'a laissé tel qu'il était dans l'exemple du code précédent pour mieux voir la différence dans l'exécution :

```
#include <iostream>          //à ajouter pour le cin et cout
#include <conio.h>           // à ajouter pour le getch()
#include <c:Vehicule.h>    // on a séparé l'interface de la classe Vehicule et son corps

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

int main()
{
   Vehicule v1(1000000);
   presenter(v1);

   Moto m1(400000,70);
   presenter(m1);

   return 0;
}
```

L'exécution cette fois-ci donne :

```
Ceci est un vehicule coutant 1000000 DA.
Ceci est une moto allant a 70 km/h et coutant 400000 DA.
```

On peut remarquer que l'utilisation des références est plus simple que les méthodes virtuelles. La fonction *presenter()* dans l'exemple donné a bien appelé la bonne

version de la méthode *affiche*. En utilisant des fonctions virtuelles ainsi qu'une référence sur l'objet, la fonction *presenter()* a pu correctement choisir la méthode à appeler. On dit aussi que les méthodes *affiche()* ont un *comportement polymorphique*.

Conclusion

L'objectif principal de ce polycopié est de vous faire découvrir les bases de la programmation orientée objet et vous donner envie de s'intéresser et de découvrir plus par vous-même. Il vous permet à présent à même de construire vous-même vos propres programmes objets. Nous nous sommes basés sur des exemples simples et clairs pour créer des classes et manipuler ainsi des objets, en se basant surtout sur la pratique avec le C++, car seule la pratique permet de faire des progrès.

REFERENCES

1. Eric Sigoillot . Introduction à la Programmation Orientée Objet. Par Date de publication : 25 juillet 2004

2. B. W. Kernighan et D. M. Ritchie - “Le langage C : Norme ANSI”. Editions Dunod, 2004.

3. C. Delannoy. – S'initier à la programmation et à l'orienté objet. Avec des exemples en C, C++, C#, Python, Java et PHP. N°14011, 2e édition, juillet 2016, 360 pages.

4. B. Stroustrup - “Le langage C++”. Editions Pearson Education, 2004.

5. C. Delannoy - “Programmer en langage C++”. N°14008, 8e édition, 2011-2014, 820 pages.

6. Baptiste Pesquet . Introduction à la programmation orientée objet (POO) en C# Par Date de publication : 26 juin 2017.

7. https://openclassrooms.com/courses/programmez-avec-le-langage-c++

8. Hugues Bersini. Cours et exercices en UML2, Python, PHP, C#, C++ et Java (y compris Android). 7éme Edition.