

**Département Mathématique et Informatique**

**- Cycle Ingénieur -**

**POO**

**Cours - resume**

**Réaliser par :**

**KARROUM Houssam – BDCC 1**



## Table of Contents

Introduction .....	4
I. Résumé du cours ( séance 1 et 2 ) .....	5
1. Partie 1 : Généralité sur POO .....	5
2. C++ COMME UN LANGAGE C AVANCE.....	12
3. Conclusion.....	19

# Introduction

Le cours dispensé par M. Khalifa Mansouri sur la Programmation Orientée Objet (POO) offre une exploration approfondie des principes fondamentaux de la programmation orientée objet ainsi que de leur application spécifique en langage C++. Divisé en deux parties distinctes, ce cours vise à fournir à nous une compréhension complète des concepts de base de la POO ainsi que des compétences pratiques pour les mettre en œuvre dans le cadre du langage C++.

La première partie du cours se concentre sur l'acquisition des concepts généraux de la POO. Nous y découvrons les notions fondamentales telles que les classes, les objets, l'encapsulation, l'héritage et le polymorphisme. Ces concepts constituent les piliers de la programmation orientée objet et sont essentiels pour comprendre en profondeur la manière dont les objets interagissent dans un système logiciel.

En outre, des travaux dirigés (TD) sont proposés pour permettre aux étudiants de mettre en pratique les concepts abordés en cours.

Cette première partie du cours constitue ainsi une base solide pour la suite du programme, préparant les étudiants à aborder avec confiance des concepts plus avancés de la POO et leur application spécifique en langage C++.

# I. Résumé du cours ( séance 1 et 2 )

## 1. Partie 1 : Généralité sur POO

Dans le monde de développement des application informatiques on peut noter, en generale, deux types de programmation :

- Procédurale :
  - ✓ Dans la programmation procédurale, le programme est structuré autour de procédures ou de fonctions qui effectuent des actions sur des données.
  - ✓ Les données sont généralement des variables globales ou passées en paramètres aux fonctions.
  - ✓ L'accent est mis sur les étapes séquentielles d'exécution du programme.

Exemples de langages de programmation procéduraux : langage C, Pascal.

Exemple de code procédural en C :

```
#include <stdio.h>

// Fonction pour calculer la somme de deux nombres
int somme(int a, int b) {
    return a + b;
}

int main() {
    int x = 0;
    int y = 33;
    int resultat = somme(x, y);
    printf("La somme de %d et %d est : %d\n", x, y, resultat);
    return 0;
}
```

- Orientée objets :
  - ✓ Dans la programmation orientée objet, les données et les méthodes qui les manipulent sont encapsulées dans des entités appelées objets.
  - ✓ Les objets interagissent entre eux en envoyant des messages, généralement en appelant des méthodes.
  - ✓ Les concepts clés de la POO comprennent l'encapsulation, l'héritage et le polymorphisme.
  - ✓ Exemples de langages de programmation orientés objet : C++, Java, Python.

#### Exemple de code orienté objet en C++ :

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

// Définition de la classe Point
class Point {
private:
    int x, y;
public:
    Point(int x, int y) : x(x), y(y) {}
    void afficher() {
        cout << "Coordonnées : (" << x << ", " << y << ")" << endl;
    }
};

int main() {
    // Création d'un objet de la classe Point
    Point p(3, 4);
    // Appel de la méthode afficher() de l'objet p
    p.afficher();
    return 0;
}
```

**Il est important de noter que** de nombreux concepts fondamentaux et techniques utilisés en programmation procédurale restent applicables en programmation orientée objet. La maîtrise de ces concepts permet aux programmeurs de passer facilement des langages traditionnels aux langages orientés objet.

#### **a. Notion d'objets**

Dans un langage orienté objet, les programmes reposent sur la notion d'objets. Contrairement aux langages traditionnels où les variables sont limitées à un type donné et à une seule donnée, les objets permettent une abstraction plus puissante en regroupant plusieurs variables de types différents en une seule structure de données. Chaque variable représentant un objet est associée à un nom unique, et ces variables partagent des caractéristiques communes.

Les points clés à retenir sont les suivants :

##### **Structure d'un objet :**

Un objet est une super-variable regroupant plusieurs variables de types différents.

Ces variables, appelées propriétés de l'objet, définissent ses caractéristiques.

Les règles de déclaration, d'identification, de typage et de manipulation des variables s'appliquent également aux propriétés d'un objet.

##### **Méthodes :**

Pour compléter la notion d'objet, des procédures appelées méthodes sont intégrées à la structure de l'objet.

Les méthodes agissent sur les propriétés de l'objet et réalisent des traitements spécifiques.

Ainsi, un objet se compose de deux parties : une partie statique constituée des propriétés (données) et une partie dynamique constituée des méthodes (traitements).

## b. Notion de classe

La notion de classe est fondamentale en programmation orientée objet. Voici un résumé des points clés :

### Définition des classes :

**En programmation orientée objet**, les objets ayant des propriétés et des comportements similaires sont regroupés en une entité appelée classe.

Tous les objets instanciés à partir d'une même classe partagent la même structure de données (attributs) et les mêmes comportements (méthodes).

### Rôle des classes :

Les classes agissent comme des modèles ou des prototypes à partir desquels des objets peuvent être créés (instanciés).

Elles fournissent une abstraction pour représenter des concepts du monde réel ou des entités logiques dans un programme informatique.

### Avantages de l'utilisation des classes :

**Abstraction** : Les classes permettent de modéliser un problème de manière abstraite en regroupant les objets similaires.

**Réutilisation du code** : Les méthodes définies dans une classe peuvent être réutilisées par toutes les instances de cette classe.

**Stockage efficace** : Les informations communes telles que le nom de la classe et les noms d'attributs sont stockées une seule fois par classe, plutôt que pour chaque instance individuelle.

Exemple d'utilisation des classes :

**Par exemple**, dans une classe **Cercle**, on pourrait définir des attributs tels que le rayon et des méthodes telles que le calcul de l'aire ou du périmètre.

Tous les objets instanciés à partir de la classe Cercle partageraient ces mêmes caractéristiques et méthodes, permettant une gestion uniforme et cohérente des cercles dans le programme.



### c. Notion de l'encapsulation

**L'encapsulation de données** en programmation orientée objet (POO) consiste à regrouper les attributs (variables) et les méthodes (fonctions) au sein d'une classe. Imaginez une classe comme une boîte qui contient à la fois des données et les actions que l'on peut effectuer sur ces données.

L'objectif principal de l'encapsulation est de cacher les détails d'implémentation de cette boîte à l'utilisateur extérieur. Cela se fait en définissant certains membres de la classe comme étant privés, ce qui signifie qu'ils ne peuvent être manipulés qu'à l'intérieur de la classe elle-même. Ces membres privés sont accessibles uniquement par les méthodes de la classe.

En fournissant une interface publique (des méthodes publiques) pour accéder et modifier ces membres privés, l'encapsulation assure que les utilisateurs de la classe ne peuvent interagir qu'avec les fonctionnalités prévues, sans avoir à se soucier de la façon dont elles sont mises en œuvre en interne.

**Pour rendre cette idée plus concrète, imaginez une voiture.** Les détails internes du moteur, des roues ou du système de freinage sont cachés à l'utilisateur, qui n'a besoin que de connaître l'interface externe, comme le volant, les pédales et le levier de vitesse pour conduire la voiture en toute sécurité. C'est l'idée de l'encapsulation : cacher les détails internes et fournir une interface simple et sécurisée pour interagir avec l'objet.

**Les niveaux de protection couramment utilisés en C++ sont :**

**Public :** Les membres publics sont accessibles à partir de n'importe quelle partie du programme.

**Private :** Les membres privés ne sont accessibles que depuis la classe elle-même. Ils ne sont pas accessibles à partir d'autres parties du programme.

**Protected :** Les membres protégés sont similaires aux membres privés, sauf qu'ils sont également accessibles par les classes dérivées (héritées).

L'utilisation judicieuse de l'encapsulation garantit une conception de classe plus robuste et modulaire, favorisant la réutilisabilité du code et la maintenance facilitée.

#### **d. Notion d'héritage**

L'héritage est un principe fondamental de la programmation orientée objet (POO) où une classe peut hériter des propriétés et des comportements d'une autre classe appelée classe de base ou classe parent. La classe qui hérite est appelée classe dérivée ou classe enfant.

L'héritage permet la réutilisation du code et la création de hiérarchies de classes. La classe dérivée peut étendre ou spécialiser les fonctionnalités de la classe de base tout en ajoutant ses propres fonctionnalités uniques.

Par exemple, une classe "Voiture" peut hériter des caractéristiques d'une classe "Véhicule", tout en ajoutant des fonctionnalités spécifiques aux voitures comme le nombre de portes ou le type de carburant.

##### **Généralisation et Spécialisation :**

- La généralisation est le processus de création d'une classe plus générale à partir de classes plus spécifiques en identifiant les caractéristiques communes et en les regroupant dans une classe de base.
- La spécialisation est le processus inverse, où une classe plus spécifique est créée à partir d'une classe plus générale en ajoutant des attributs ou des méthodes supplémentaires pour répondre à des besoins particuliers.
- Par exemple, à partir d'une classe "Animal" (généralisation), on peut spécialiser des classes telles que "Chien" ou "Chat" avec des attributs et des méthodes spécifiques à chaque espèce.

##### **Héritage Multiple :**

- L'héritage multiple est la capacité d'une classe à hériter des attributs et des méthodes de plusieurs classes de base. Cela signifie qu'une classe peut avoir plusieurs ancêtres.
- Bien que certains langages de programmation le permettent, l'héritage multiple peut rendre la conception complexe et peut entraîner des ambiguïtés ou des conflits de noms.
- Il est souvent recommandé d'éviter l'héritage multiple au profit de la composition ou de l'utilisation d'interfaces pour éviter les problèmes potentiels de conception.

### e. Notion Polymorphisme

Le polymorphisme est un concept qui permet à un même nom de fonction d'avoir des comportements différents en fonction du contexte d'utilisation.

Il existe deux types de polymorphisme : le polymorphisme de compilation (ou statique) et le polymorphisme d'exécution (ou dynamique).

Le polymorphisme de compilation se produit lorsque le compilateur sélectionne la méthode à appeler en fonction du type statique du pointeur ou de la référence. Le polymorphisme d'exécution se produit lorsque la méthode appelée est déterminée au moment de l'exécution en fonction du type réel de l'objet.

Le polymorphisme permet une meilleure abstraction et une plus grande flexibilité dans la conception des systèmes logiciels, car il permet de traiter des objets de manière générale sans se soucier de leur type spécifique.

#### Exemple avec code :

```
#include <iostream>

// Classe de base (parent)
class Forme {
public:
    // Méthode virtuelle pure (abstraite)
    virtual void afficher() const = 0;
};

// Classe dérivée (enfant) : Rectangle
class Rectangle : public Forme {
public:
    void afficher() const override {
        std::cout << "Je suis un rectangle." << std::endl;
    }
};

// Classe dérivée (enfant) : Cercle
class Cercle : public Forme {
public:
    void afficher() const override {
        std::cout << "Je suis un cercle." << std::endl;
    }
};
```

## 2. C++ COMME UN LANGAGE C AVANCE

### a. LES COMMENTAIRES

En C++, en plus des symboles `/*` et `*/` utilisés pour les commentaires sur plusieurs lignes, le langage offre également le symbole `//` pour les commentaires en fin de ligne. Voici des exemples d'utilisation :

```
/* commentaire traditionnel sur plusieurs lignes valide en C et
C++ */
// commentaire de fin de ligne valide en C++
// Définition de la classe Point
```

### b. EMBLACEMENT DES DECLARATIONS

En C++, il n'est plus obligatoire de regrouper toutes les déclarations au début d'une fonction ou d'un bloc. Les exemples suivants illustrent cette flexibilité, **qui est acceptée en C++ mais non en C** :

```
9
10 void main() {
11     // ...
12     for(int i = 1; ...; ...) {
13         // ...
14     }
15     // ...
16 }
```

```
18 void main() {
19     // ...
20     int i;
21     for(i = 1; ...; ...) {
22         // ...
23     }
24     // ...
25 }
26
```

Il est important de noter qu'il existe une différence subtile entre ces deux approches. Dans le premier exemple, `i` serait accessible en dehors de la boucle `for`, tandis que dans le second exemple, sa portée serait limitée au bloc de l'instruction `for`.

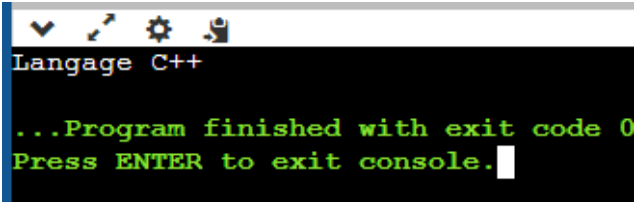
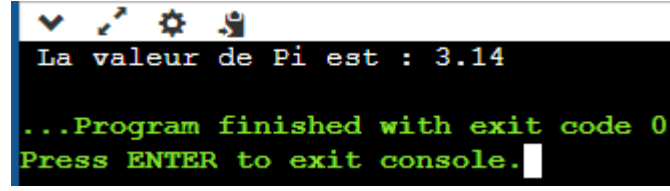
### c. LES NOUVELLES POSSIBILITES I/O CONVENTIONNELLES : CIN, COUT

En C++, lorsqu'on inclut le fichier d'en-tête `<iostream>`, les flux prédéfinis suivants sont disponibles :

**std::cout** : Ce flux est utilisé pour l'affichage des données vers la sortie standard (généralement la console).

**std::cin** : Ce flux est utilisé pour la lecture des données depuis l'entrée standard (généralement le clavier).

Il est important de noter que l'inclusion du fichier d'en-tête `<iostream>` en C++ standard ne nécessite pas l'extension `.h` et qu'il est inclus dans l'espace de noms `std`, donc les flux sont généralement référencés comme `std::cout` et `std::cin`.

Exemple	execution
<pre># include &lt;iostream.h&gt;  void main( ) { cout &lt;&lt; " Langage C++ " ; }</pre>	 A screenshot of a C++ program execution. The window title is 'Langage C++'. The output shows '...Program finished with exit code 0' and 'Press ENTER to exit console.'.
<pre># include &lt;iostream.h&gt;  void main( ) { float Pi = 3.14 ; cout &lt;&lt; " La valeur de Pi est : " ; cout &lt;&lt; Pi ; }</pre>	 A screenshot of a C++ program execution. The window title is 'La valeur de Pi est : 3.14'. The output shows '...Program finished with exit code 0' and 'Press ENTER to exit console.'.

### 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.

#### d. LES CONVERSIONS EXPLICITES

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

ex1:

```
double d;  
int i = (int) d;
```

ex2:

```
#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();  
}
```

**Remarque :**

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

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

### e. VISIBILITE DES VARIABLES

```
#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;
}
```

Ce code illustre l'utilisation de l'opérateur de résolution de portée (::) pour accéder aux variables globales plutôt qu'aux variables locales. Voici un résumé de cette partie :

- La variable **i** est déclarée globalement avec une valeur de 11.
- À l'intérieur de la fonction **main()**, une nouvelle variable **i** est déclarée localement avec une valeur de 34.
- À l'intérieur d'un bloc de code imbriqué, une autre variable **i** est déclarée localement avec une valeur de 23.
- En utilisant **::i**, l'opérateur de résolution de portée, la variable globale **i** est modifiée en ajoutant 1 à sa valeur.
- Le premier **cout** affiche la valeur de la variable globale **i** (12) et la valeur de la variable locale la plus interne **i** (23) à l'intérieur du bloc de code imbriqué.
- Le deuxième **cout** affiche à nouveau la valeur de la variable globale **i** (12) et la valeur de la variable locale **i** (34) à l'extérieur du bloc de code imbriqué.

En résumé, l'opérateur de résolution de portée permet d'accéder aux variables globales même si des variables locales portent le même nom.

## f. LES FONCTIONS

Les fonctions doivent être déclarées avant d'être utilisées, avec le nombre et le type d'arguments spécifiés. Par exemple, la déclaration `"int f1();"` sans arguments est équivalente à `"int f1(void);"`. Si une fonction retourne une valeur autre que void, elle doit obligatoirement retourner cette valeur.

### ➤ Valeur par défaut des paramètres:

En C++, vous pouvez spécifier une valeur par défaut pour un argument de fonction. Lorsque vous appelez cette fonction, si vous omettez cet argument, il prendra la valeur par défaut spécifiée. Sinon, si vous fournissez une valeur pour cet argument, la valeur par défaut sera ignorée.

Par exemple, dans la fonction `f1(int n = 3)`, le paramètre `n` a une valeur par défaut de 3. Si vous appelez `f1(i)`, où `i` vaut 2, `n` prendra la valeur 2, ignorant ainsi la valeur par défaut. Mais si vous appelez simplement `f1()`, sans fournir d'argument, `n` prendra la valeur par défaut, soit 3.

**Il est important de noter que** les paramètres avec des valeurs par défaut doivent être placés en fin de liste des arguments de la fonction.

### ➤ Surcharge (surdefinition) des fonctions ('overloading'):

En C++, il est possible de définir plusieurs fonctions avec le même nom, mais elles doivent être différenciées par le type et/ou le nombre des arguments. Par exemple, vous pouvez avoir une fonction "somme" prenant deux entiers comme arguments, une autre prenant trois entiers, et une autre encore prenant deux doubles.

Lors de l'appel d'une de ces fonctions, le compilateur sélectionnera la fonction à utiliser en se basant sur le type et le nombre d'arguments fournis dans l'appel. Cette sélection est effectuée au moment de la compilation du programme.



Dans l'exemple donné, les appels `somme(1, 2)`, `somme(1, 2, 3)` et `somme(1.2, 2.3)` sont correctement dirigés vers les fonctions appropriées en fonction du nombre et du type des arguments fournis.

#### **g. ALLOCATION MEMOIRE**

En C++, le programmeur a à sa disposition deux opérateurs, `new` et `delete`, pour gérer dynamiquement la mémoire, remplaçant ainsi les fonctions `malloc` et `free` du langage C (bien qu'il soit toujours possible d'utiliser ces dernières).

L'opérateur `new` est utilisé pour allouer dynamiquement de la mémoire et l'initialiser. Il renvoie l'adresse de début de la zone mémoire allouée. Par exemple :

```
int *pi; // Déclaration du pointeur
pi = new int; // Allocation de la mémoire
```

On aurait également pu écrire cette allocation en une seule ligne :

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

En langage C, l'allocation de mémoire équivalente avec `malloc` serait :

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

Ces deux codes accomplissent essentiellement la même tâche, mais la syntaxe C++ utilisant `new` est souvent considérée comme plus sûre et plus pratique car elle prend en charge la détermination de la taille de l'allocation.

## h. L'opérateur delete

En C++, l'opérateur delete est utilisé pour libérer l'espace mémoire alloué dynamiquement par l'opérateur new pour un seul objet, tandis que delete[] est utilisé pour libérer l'espace mémoire alloué à un tableau d'objets.

Exemple 1 :

```
delete pi; // désalloue la zone pointée par pi
// pi existe encore mais pas pi*
char *pc = new char[100];
delete pc; // désalloue la zone de 100 caractères
delete [100]pc; // instruction équivalente
```

L'opérateur new renvoie un pointeur NULL (0) en cas d'échec d'allocation, donc 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;
```

**Il est important de noter que pour chaque utilisation de new, il doit y avoir une correspondance avec delete pour libérer la mémoire allouée. Il est également crucial de libérer la mémoire dès qu'elle n'est plus nécessaire, car la mémoire allouée dynamiquement n'est pas automatiquement libérée à la fin du programme. De plus, toute allocation effectuée avec new[] doit être libérée avec delete[].**

### **3. Conclusion**

En conclusion, les concepts de base de la programmation orientée objet (POO) introduisent un paradigme de programmation où les entités du monde réel sont modélisées sous forme d'objets interagissant entre eux. Les principaux concepts de la POO sont l'encapsulation, l'héritage et le polymorphisme.

En parallèle, les bases du langage C++ offrent un ensemble de fonctionnalités puissantes pour développer des logiciels robustes et efficaces. Avec C++, les programmeurs ont accès à des fonctionnalités telles que la gestion dynamique de la mémoire, les classes et les objets, ainsi que les opérateurs de surcharge.

En combinant ces deux aspects, les programmeurs peuvent créer des applications sophistiquées et flexibles, en exploitant la richesse de la POO pour concevoir des architectures modulaires et en utilisant les fonctionnalités du langage C++ pour une implémentation efficace et précise. En somme, la maîtrise de ces concepts et de ce langage ouvre la voie à des développements logiciels de qualité, tant en termes de structure que de performance.