



جامعة عبد المالك السعدي
Université Abdelmalek Essadi



Programmation Orientée Objet C++

2023-2024

1ère Année Génie Informatique

Prof : Anouar RAGRAGUI

Plan du cours

- ❑ Chapitre 1: Introduction à l'orienté objet
- ❑ Chapitre 2: Spécificité de C++
- ❑ Chapitre 3: Classes et objets
- ❑ Chapitre 4: Fonctions et classes amies && Surdéfinition des opérateurs
- ❑ **Chapitre 5: Héritage && Polymorphisme**
- ❑ Chapitre 6: Les Templates
- ❑ Chapitre 7: Notion de la bibliothèque STL
- ❑ Chapitre 8: Gestion des Exceptions
- ❑ Chapitre 9: Gestion des bases de données
- ❑ Chapitre 10: Introduction aux interfaces graphiques avec C++ (QT)

- ❑ Notion d'héritage
- ❑ Héritage simple
- ❑ Héritage multiple
- ❑ Polymorphisme
- ❑ Classes abstraites

- Permet de définir une nouvelle classe : **classe dérivée**
 - ▣ A partir d'une ou plusieurs classes existantes : **classes de base**
 - ▣ Enrichie par de nouveaux attributs et de nouvelles méthodes
- Représente la relation **EST UN(E)**
 - ▣ Un chien est un animal
 - ▣ Une vache est un animal
 - ▣ Une voiture est un véhicule

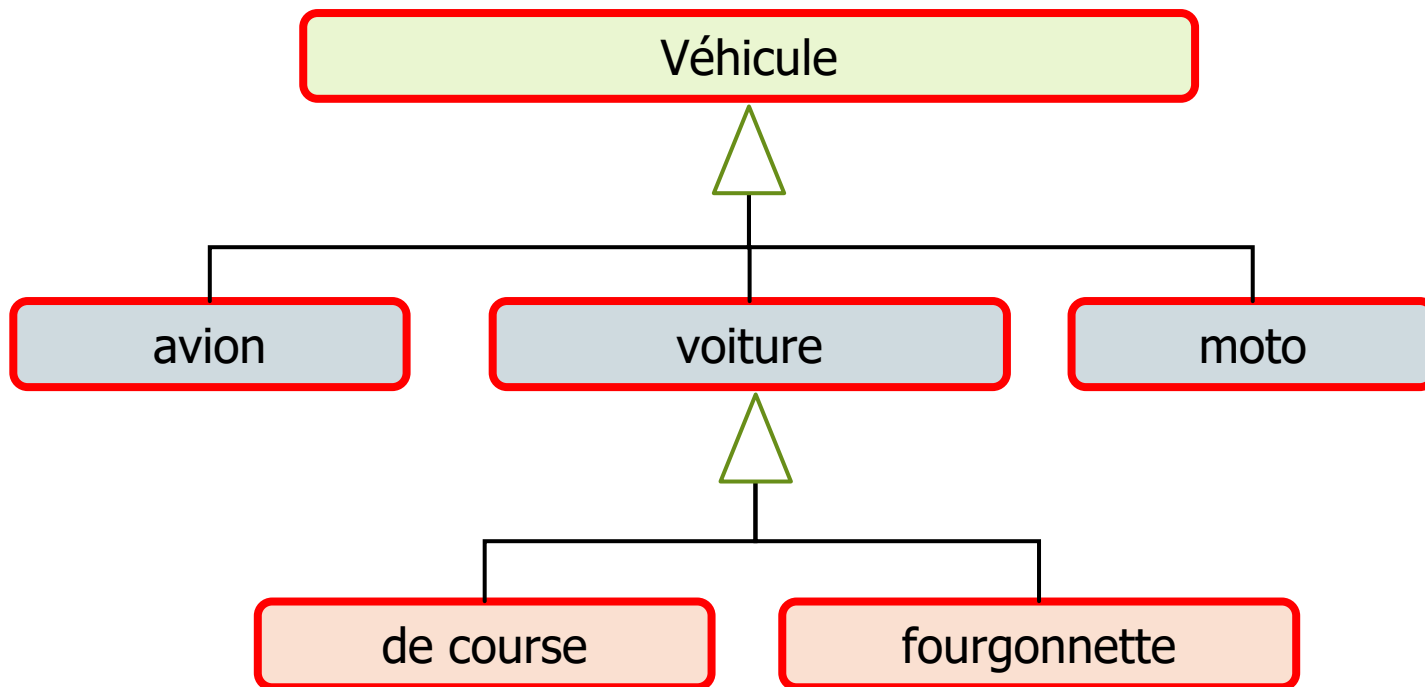
□ Type

- ▣ **Simple** — Une seule classe de base
- ▣ **Multiple** — Plusieurs classes de base

□ Apports

- ▣ **Modélisation explicite** des relations structurelles et sémantiques entre les classes
- ▣ **Réduction** des redondances de description et de stockage des propriétés
- ▣ **Réutilisation** des classes existantes

□ Exemple:



- Modélise un **cas particulier** de la classe de base
- Si une classe **D** dérive de la classe **B**, alors
 - ▣ **D** va **hériter implicitement** de l'ensemble:
 - Des attributs de **B**
 - De toutes les méthodes de **B**
 - ▣ **D** peut définir des attributs et des méthodes supplémentaires : **enrichissement**
 - ▣ **D** Peut redéfinir (**masquer**) certaines méthodes

Syntaxe:

```
class class_derivee : Mode class_de_base {  
    /*  
        définitions des méthodes et des attributs  
        spécifiques à la classe dérivée  
    */  
    ...  
}
```


- **Exemple :** Réaliser une classe personne (les données membres : nom, prénom, CIN et les fonctions membres : saisir, afficher) et une classe Étudiante qui hérite de la classe personne (les données membres: CNE les fonctions membres :saisir_cne et afficher_cne)

```
class personne {
    char nom[20];
    char prenom[20];
    int CIN;
public:
    void saisir(char*, char*, int);
    void afficher();
};

class etudiant : public personne {
    int CNE;
public:
    void saisir_cne(int);
    void afficher_cne();
};
```

- Le droit d'accès aux membres d'une classe peut être
 - ▣ **Accordé** au « grand public »
 - A partir de n'importe quel point du programme
 - Concerne l'Interface publique d'une classe
 - ▣ **Restreint** aux fonctions membres et aux fonctions amies
 - Représente le principe d'encapsulation
 - Concerne les membres protégés de la classe

- Trois modes d'accès aux membres d'une classe:
 - ▣ **public** : Accès permis à toute fonction à partir de n'importe quel point du programme
 - ▣ **private** : *Mode par défaut* — Accès permis aux
 - Fonctions membres de la classe
 - Fonctions amies
 - ▣ **protected** : *sémantique liée à l'héritage* — Accès permis aux:
 - Fonctions membres de la classe
 - Fonctions **des classes dérivées**
 - Fonctions amies

□ Exemple

```
class A {  
public:  
    int pub;  
  
protected:  
    int pro;  
  
private:  
    int priv;  
};  
  
class B : public A {  
public:  
    void f(int x);  
};
```

```
void B::f(int x)  
{  
    pub = x    // ok accès possible  
    pro = x;   // ok accès possible  
    priv = x;  // erreur accès interdit  
}  
  
void fexterne(A oa, int x)  
{  
    oa.pro = x; // erreur  
    oa.priv = x; // erreur  
    oa.pub = x; // ok  
}
```

□ Mode de dérivation

Syntaxe

```
class class_derivee : mode class_de_base
{
    ...
}
```

▣ **mode** : désigne le mode de **protection** des membres des classes

▣ Trois mode de dérivation

■ Public

■ private — *mode par défaut*

■ protected

□ Mode de dérivation **public**

| Droits d'accès aux membres dans la classe de base | Droits d'accès aux membres dans la classe dérivée |
|---|---|
| public | public |
| protected | protected |
| private | inaccessible |

□ Exemple:

```
class A {  
public:  
    int pub;  
  
protected:  
    int pro;  
  
private:  
    int priv;  
};  
  
class B : public A {  
public:  
    void f(int x);  
};
```

```
void B::f(int x)  
{  
    pub = x    // accès possible -public  
    pro = x;   // accès possible -protected  
    priv = x;  // accès interdit -private  
}  
  
void fexterne(B& b, int x)  
{  
    b.pub = x; // accès possible -public  
    b.pro = x; // accès interdit-protected  
    b.priv = x; //accès interdit-inaccessible  
}
```

□ Mode de dérivation **protected**

| Droits d'accès aux membres dans la classe de base | Droits d'accès aux membres dans la classe dérivée |
|---|---|
| public | protected |
| protected | protected |
| private | inaccessible |

□ Exemple:

```
class A {  
public:  
    int pub;  
  
protected:  
    int pro;  
  
private:  
    int priv;  
};  
  
class B : protected A {  
public:  
    void f(int x);  
};
```

```
void B::f(int x)  
{  
    pub = x;    // accès possible -protected  
    pro = x;    // accès possible -protected  
    priv = x;   // accès interdit -private  
}  
  
void fexterne(B& b, int x)  
{  
    b.pub = x;  // accès interdit-protected  
    b.pro = x;  // accès interdit -protected  
    b.priv = x; //accès interdit-inaccessible  
}
```

□ Mode de dérivation **private**

| Droits d'accès aux membres dans la classe de base | Droits d'accès aux membres dans la classe dérivée |
|---|---|
| public | private |
| protected | private |
| private | inaccessible |

□ Exemple:

```
class A {  
public:  
    int pub;  
  
protected:  
    int pro;  
  
private:  
    int priv;  
};  
  
class B : private A {  
public:  
    void f(int x);  
};
```

```
void B::f(int x)  
{  
    pub = x    // accès interdit -private  
    pro = x;   // accès interdit -private  
    priv = x;  // accès interdit -private  
}  
  
void fexterne(B& b, int x)  
{  
    b.pub = x; // accès interdit -private  
    b.pro = x; // accès interdit -private  
    b.priv = x; //accès interdit-inaccessible  
}
```

□ Exemple:

```
class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
};
class Derived_Public : public Base {
    // Ici, les membres publics et protégés de Base restent publics et
    protégés dans Derived_Public.
};
class Derived_Protected : protected Base {
    // Ici, les membres publics et protégés de Base deviennent protégés
    dans Derived_Protected.
};
class Derived_Private : private Base {
    // Ici, tous les membres de Base deviennent privés dans
    Derived_Private.
};
```

- Un objet d'une classe dérivée peut être affecté à un objet de la classe de base
 - ▣ Un objet d'une classe dérivée est un cas particulier de la classe de base
 - ▣ Un étudiant est « toujours une personne »
- Un objet d'une classe de base ne peut pas être affecté à un objet de l'une des classes dérivées
 - ▣ Un objet de la classe de base n'est pas forcément cohérent avec la classe dérivée
 - ▣ Une personne n'est pas forcément étudiant

□ Exemple

```
class personne {
    char nom[20];
    int  CIN;
public:
    void saisir(char*, int);
    void afficher();
};
class etudiant : public personne {
    int CNE;
public:
    etudiant(char*, int, int);
    void saisir_cne(int);
    void afficher_cne();
};
int main() {
    char nom[] = "Ahmed";
    personne p(nom, 12547);
    etudiant e(nom, 12547, 151256);
    p = e; // affectation permise
    e = p; // erreur
}
```

- Besoin d'adapter certaines fonctions de la classe de base à la classe dérivée
- La nouvelle fonction doit garder le *même nom* que celui de la classe de base
- Redéfinition \neq surdéfinition
- La nouvelle définition (*celle de la classe dérivée*) va *masquer* l'ancienne (*celle de la classe de base*)

□ Exemple:

```
class Rectangle {  
protected:  
    float largeur;  
    float hauteur;  
public:  
    float surface();  
};  
float Rectangle::surface() {  
    return largeur * hauteur;  
}  
class Rectangle3D :public Rectangle {  
protected:  
    float profondeur;  
public:  
    float surface();  
};
```

```
float Rectangle3D::surface()  
{  
    return(2 * largeur * hauteur + 2 *  
           largeur * profondeur + 2 * hauteur *  
           profondeur);  
}  
void main()  
{  
    Rectangle3D r;  
    //celle de la classe Rectangle3D  
    r.surface();  
}
```


□ Parfois il est souhaitable d'accéder aux fonctions de la classe de base qui sont cachées lors de la redéfinition par la classe dérivée

□ Pour y accéder on utilise l'opérateur `::`

□ **Syntaxe**

```
//définition de la méthode dans la classe de base
```

```
class_base::methode() {  
    ...  
}
```

```
// redéfinition de la méthode dans la classe dérivée
```

```
class_derivee::methode() {  
    ...  
}
```

```
// accès à la fonction de la classe de base
```

```
void class_derivee::f() {  
    class_base::methode();  
}
```

□ **Exemple:** redéfinir la fonction surface de la classe Rectangle3D

```
class Rectangle3D :public Rectangle {
protected:
    float profondeur;
public:
    float surface();
};
float Rectangle3D::surface() {
    if (profondeur == 0)
        return Rectangle::surface();
    else
        return (2 * largeur * hauteur + 2 * largeur * profondeur + 2 *
            hauteur * profondeur);
}
```

□ Exemple: *Utilisation non fréquente*

```
class A {  
protected:  
    int a;  
public:  
    A() { a = 0; }  
};  
class B : public A {  
    int a;  
public:  
    B() { a = 10; };  
    void f() {  
        cout << "A::" << A::a; //affiche 0  
        cout << "B::" << a; // affiche 10  
    }  
};
```

- ❑ Pas d'héritage des constructeurs et destructeurs : il faut les redéfinir
- ❑ Appel implicite des constructeurs par défaut des classes de base (super-classe) avant le constructeur de la classe dérivée (sous-classe)
- ❑ Possibilité de passage de paramètres aux constructeurs de la classe de base dans le constructeur de la classe dérivée par appel explicite
- ❑ Appel automatique des destructeurs dans l'ordre inverse des constructeurs
- ❑ Pas d'héritage des constructeurs de copie et des opérateurs d'affectation

□ Ordre d'appel des constructeurs/destructeur

- Les destructeurs sont toujours appelés dans l'ordre inverse /symétrique) des constructeurs.

- Par exemple : **C** **c**, lors de la destruction d'un **c**, on aura appel et exécution de :

C::~C()

B::~B()

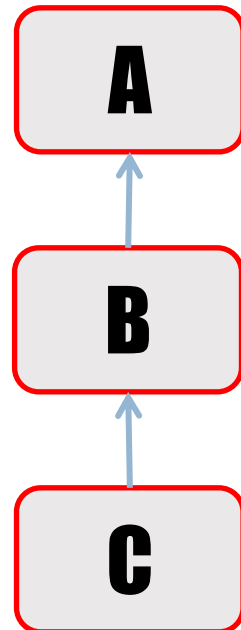
A::~A()

- car dans cet ordre les constructeurs avaient été appelés dans l'ordre

A::A()

B::B()

C::C()



- **constructeurs de la classe de base**
- Trois cas possibles pour le constructeur de la classe de base
 - ▣ Constructeur par défaut : *Constructeur n'est pas défini*
 - ▣ Constructeur sans paramètres : *Initialisation des membres données par des valeurs aléatoires ou constantes*
 - ▣ Constructeur avec paramètres
- Dans les deux premiers cas, le constructeur de la classe de base peut être invoqué **implicitement** sans problème
- Dans le troisième cas le constructeur de la classe de base requière des paramètres !!

Constructeurs de la classe de base

Constructeur de la classe de base avec paramètres

Comment lui passer les paramètres ?



- ❑ Trois cas pour Le constructeur de la classe dérivée
 - Non défini
 - Sans paramètres
 - Avec paramètres

Constructeur de la classe de base avec paramètres

- Comment passer les paramètres au constructeur de la classe de base?
- Constructeur de la classe dérivée n'est pas défini
- **Erreur sauf** Si les arguments du constructeur de la classe de base sont déclarés par défaut

```
class A {  
protected:  
    int a;  
public:  
    A(int a) { this->a = a };  
};  
class B : public A {  
    int a;  
};
```

ERREUR

```
class A {  
protected:  
    int a;  
public:  
    A(int a = 10) { this->a = a };  
};  
class B : public A {  
    int a;  
};
```

Juste

- ❑ *Constructeur de la classe de base avec paramètres*
- ❑ *Comment passer les paramètres au constructeur de la classe de base?*
- ❑ *Constructeur de la classe dérivée sans paramètres*
- ❑ *Erreur sauf :*
 - ▣ Les arguments du constructeur de la classe de base sont déclarés par défaut
 - ▣ On improvise des valeurs pour les passer au constructeur de la classe de base

```
class A {  
protected:  
    int a;  
public:  
    A(int a) { this->a = a };  
};  
class B : public A {  
    int a;  
    B() {};  
};
```

ERREUR

```
class A {  
    protected:  
        int a;  
public:  
    A(int a) { this->a = a; };  
};  
class B : public A {  
    int a;  
    B() :A(10) {};  
};
```

A(int a = 10){this->a=a};

- ❑ Constructeur de la classe de base avec paramètres
- ❑ Comment passer les paramètres au constructeur de la classe de base?
- ❑ Constructeur de la classe dérivée avec paramètres

Un sous ensemble des paramètres est passé **explicitement** au constructeur de la classe de base lors de la déclaration du constructeur de la classe dérivée

```
class A {  
protected:  
    int a;  
    float af;  
    char ac;  
public:  
    A(int a, float af, char ac) {  
        this->a = a;  
        this->af = af;  
        this->ac = ac;  
    };  
};
```

```
class B : public A {  
    int b;  
public:  
    int a, float af, char ac,  
    B(int a, float af, char ac, int b)  
    :A(a, af, ac)  
    {  
        this->b = b;  
    }  
};
```

Paramètres de la classe de base

Récapitulatif sur les constructeurs

Classe de base

Classe dérivée

| | Constructeur par défaut | Constructeur sans paramètres | Constructeur avec paramètres |
|------------------------------|--|--|--|
| Constructeur par défaut | Appel implicite du constructeur par défaut | Appel implicite du constructeur de la classe de base | Erreur |
| Constructeur sans paramètres | Appel implicite du constructeur par défaut | Appel explicite du constructeur de la classe de base | Erreur (sauf valeurs par défaut ou improvisées) |
| Constructeur avec paramètres | Appel implicite du constructeur par défaut | Appel implicite du constructeur de la classe de base | Transmission des paramètres au constructeur de la classe de base |

- Recopie des attributs de la classe de base et ceux de la classe dérivée
- Deux cas se présentent :
 - ▣ La classe dérivée n'a pas de constructeur de copie
 - ▣ La classe dérivée possède un constructeur de copie

□ Classe dérivée n'a pas de constructeur de copie

□ Appel des constructeurs de copie dans l'ordre suivant:

- ▣ Constructeur de copie explicite ou par défaut de la classe de base
- ▣ Constructeur de copie par défaut de la classe dérivée

```
class A {  
protected:  
    int a;  
public:  
    A(A& oa) {  
        this->a = oa.a;  
    };  
};
```

```
class B : public A {  
    int b;  
};  
main() {  
    B oa1;  
    B oa2 = oa1; //appel de  
                  copie  
}
```

- Classe dérivée possède un constructeur de copie
- Pas d'appel automatique du constructeur de copie de la classe de base
- La classe dérivée
 - Doit prendre en charge l'intégralité de la copie de l'objet et non seulement sa partie héritée
 - Utilise le mécanisme de transmission entre constructeurs

```
class A {  
protected:  
    int a;  
public:  
    A(A& oa) {  
        this->a = oa, a;  
    };  
};
```

```
class B : public A {  
    int b;  
public:  
    B(B& ob) ):A(ob) {  
        this->b = ob.b;  
    };  
};
```

□ L'opérateur = n'est pas défini dans la classe dérivée

□ *Affectation membre à membre*

□ *Appel implicite de l'opérateur par défaut ou surchargé de la classe de base pour copier la partie héritée*

□ *Appel ensuite de l'opérateur de la classe dérivée*

```
class A {  
protected:  
    int a;  
public:  
    A& operator=(const A& oa)  
    {  
        this->a = oa.a;  
        return *this;  
    };  
};
```

```
class B : public A {  
    int b;  
  
};  
main() {  
    B oa1;  
    B oa2;  
    oa1 = oa2; // appel de  
                //l'opérateur d'affectation  
}
```

□ L'opérateur = est défini dans la classe dérivée

- ▣ Appel de l'opérateur de la classe dérivée surchargé
- ▣ Pas d'appel de l'opérateur de la classe de base même s'il est surchargé
- ▣ L'opérateur de la classe dérivée doit prendre en charge l'affectation de tous l'objet

```
class A {  
protected:  
    int a;  
public:  
    A& operator=(const A& oa) {  
        this->a = oa, a;  
        return *this;  
    };  
};
```

```
class B : public A {  
    int b;  
public:  
    B& operator=(const B& ob) :  
        operator=(ob) {  
            this->b = ob.b;  
            return *this;  
        };  
};
```


- L'amitié accorde aux fonctions de cette classe les mêmes droits d'accès que ceux des fonctions membres
- L'amitié ne se propage pas aux classes dérivées
- **L'amitié pour une fonction n'est pas héritée.** A chaque dérivation il faut définir l'amitié pour cette fonction

- Possibilité de créer des classes dérivées à partir de plusieurs classes de base
- Pour chaque classe de base il y a la possibilité de définir le mode de dérivation
- Il n'y a pas de restriction sur le nombre de classes de base
- L'ordre des classes de base est pris en compte lors de l'invocation des constructeurs et destructeurs
 - ▣ Appel des constructeurs dans l'ordre de déclaration de l'héritage
 - ▣ Appel des destructeurs dans l'ordre inverse de celui des constructeurs

Syntaxe:

```
class D : mode_B1 B1, mode_B2 B2, ... mode_Bn Bn {  
    ...  
}
```

- Chaque classe de base définit son mode de dérivation
- Mode = {public, protected, private}

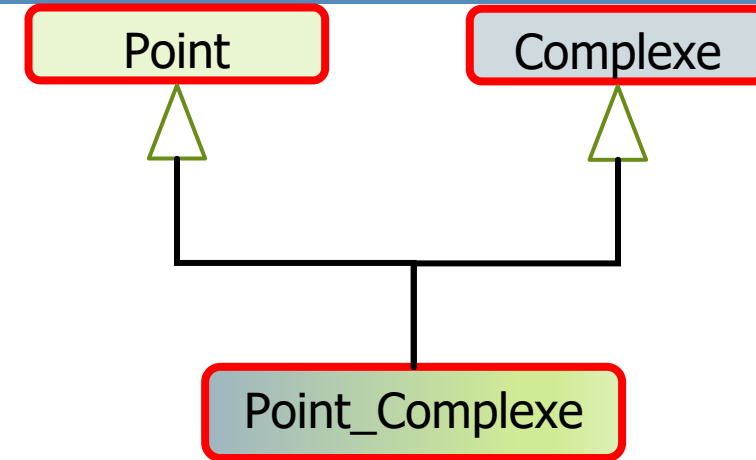
□ Exemple :

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

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

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

void point::affiche_point() {
    cout << "position :" << x << "," << y << "\n";
}
```



```
class complexe {
    float reel, img;
public:
    void complexe(int, int);
    void affiche_complexe();
};

void complexe::complexe(int r, int i) {
    reel = r;
    img = i;
}

void complexe::affiche_complexe() {
    cout << "abscisse :" << x << ",
    ordonnee:" << y << "\n";
}
```

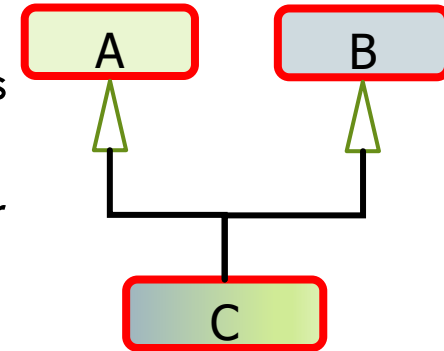
□ Exemple :

```
class point_complexe : public point, public complexe {
public:
    point_complex(int, int, float, float);
    void affiche_pc();
};
point_complexe::point_complexe(int n, int m, float x, float y) : point(n,
m), complexe(x, y)
{
}

void point_complexe::affiche_pc()
{
    affiche_point();
    affiche_complexe();
}
```

Conflit 1 :

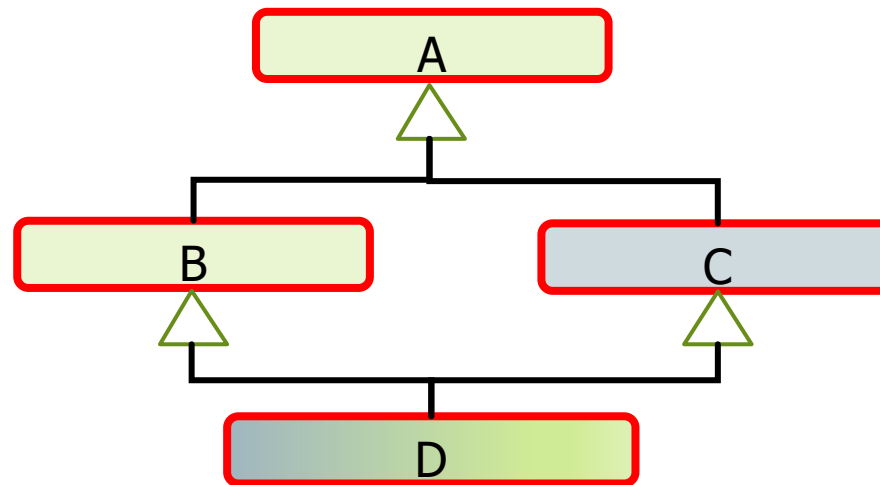
- ❑ **Problème** : Les classes A et B possèdent des attributs ou des fonctions portant le même nom
- ❑ **Solution** : Possibilité de les distinguer en utilisant l'opérateur de résolution ::
- ❑ **Exemple**



```
class A {  
protected: int a;  
public:  
    A() { a = 1; }  
};  
  
class B {  
protected: int a;  
public:  
    B() { a = 2; }  
};
```

```
class C :public A, public B {  
public:  
    void affiche() {  
        cout << "attribut herite de A  
est :« << A::a;  
        cout << "attribut herite de B  
est :\" << B::a;  
    }  
};
```

Conflit 2 :



- ❑ **Problème :** Duplication des membres fonctions ou attributs dans tous les objets de D
- ❑ **Solution :** **Dérivation virtuelle** le membre sera présent en un seule exemplaire

- Possibilité de déclarer une classe « virtuelle » pour indiquer au compilateur les classes à ne pas dupliquer
- Placement du mot clé « **virtual** » devant le mode de dérivation de la classe
- **Syntaxe:**

```
class A
{...}
// les descendants de B hériteront une seule fois les membres de A
class B : public virtual A
{...}
class C : public virtual A
{...}
class D : public B, public C
{...}
```
- Constructeurs : Cas où la classe A possède un constructeur avec paramètres
 - ▣ Pas de transmission d'arguments au constructeur de la classe A dans les classes B et C qui sont « **virtual** »
 - ▣ Possibilité de transmettre les arguments nécessaires au constructeur de la classe A
 - ▣ Mais nécessité d'avoir un constructeur sans paramètres pour la classe A

□ Exemple:

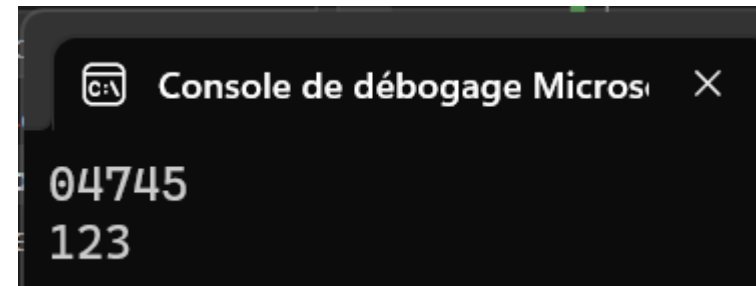
```
class A {
protected: int a;
public:
    A(int a) {
        this->a = a;
    }
    A() {} // nécessaire
};

class B : public virtual A {
protected: int b;
public:
    B(int b) {
        this->b = b;
    }
};

class C : public virtual A {
protected: int c;
public:
    C(int c) { this->c = c; }
};
```

```
class D : public B, public C {
public:
    D(int x, int y, int z) : A(x),
        B(y), C(z) {}
    D() : A(0), C(45), B(47) {}
    void afficher() {
        cout << a << b << c << endl;
    }
};

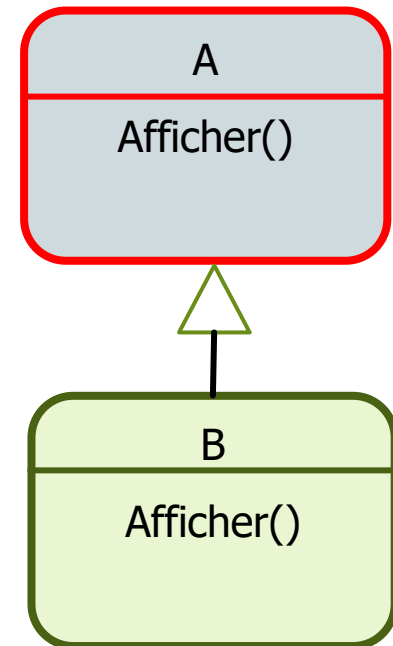
int main() {
    D ex1, ex2(1, 2, 3);
    ex1.afficher();
    ex2.afficher();
}
```



- Ce terme désigne la capacité qu'une méthode de s'adapter automatiquement à l'objet manipulé.
- « **Poly** » signifie « **plusieurs** », et « **morphe** » signifie « **forme** »
- N'a de sens que dans un contexte « **Héritage** ».
- Principale application : regroupement d'objets dans des listes hétérogènes.
- Le polymorphisme en C++ signifie **qu'un appel à une fonction membre entraînera l'exécution d'une fonction différente en fonction du type d'objet qui appelle la fonction.**
- Comment C++ implémente-t-il le polymorphisme ?
 - ▣ **Typage dynamique**
 - ▣ **Fonctions virtuelles**

- Concerne les objets dynamiques et l'héritage

```
A* pp; // pp est un pointeur de type A  
B b; // objet de type B  
pp = &b; // pp pointe sur un objet de type B
```



- Problème : Comment choisir la fonction adéquate à exécuter dans l'instruction ?

`pp->Afficher()`

□ Typage statique:

- ▣ Le type de l'objet pointé est déterminé au moment de la **compilation**
- ▣ Le choix de la fonction est déterminé par le **type du pointeur** et non par celui de l'objet pointé

➡ L'instruction **pp->Afficher()** ; correspond à la fonction de la classe **A**

□ Typage dynamique:

- ▣ Le type de l'objet pointé est déterminé au moment de **l'exécution** et non de la compilation
- ▣ Le choix de la fonction est déterminé par le **type de l'objet pointé** et non par le type du pointeur

➡ L'instruction **pp->Afficher()** ; correspond à la fonction de la classe **B**

- ▣ Obtenu par **les fonctions virtuelles**

□ Fonctions virtuelles

- ▣ Ne pas confondre l'héritage virtuel et le statut virtuel des fonctions membres
- ▣ Faire précéder la fonction en question par le mot clé **virtual**
- ▣ Si une méthode d'une classe de base est virtuelle, alors, en cas d'appel de cette méthode sur un pointeur pointant sur un objet d'une classe dérivée,
 - Si la classe dérivée a implémenté la méthode : c'est celle-ci qui sera exécutée.
 - Si la classe dérivée n'a pas implémenté la méthode : c'est la méthode de la classe de base qui sera exécutée

□ Fonctions virtuelles pures

- ▣ Fonctions déclarées
 - Sans définition dans une classe
 - Doivent être redéfinies dans les classes dérivées

Syntaxe:

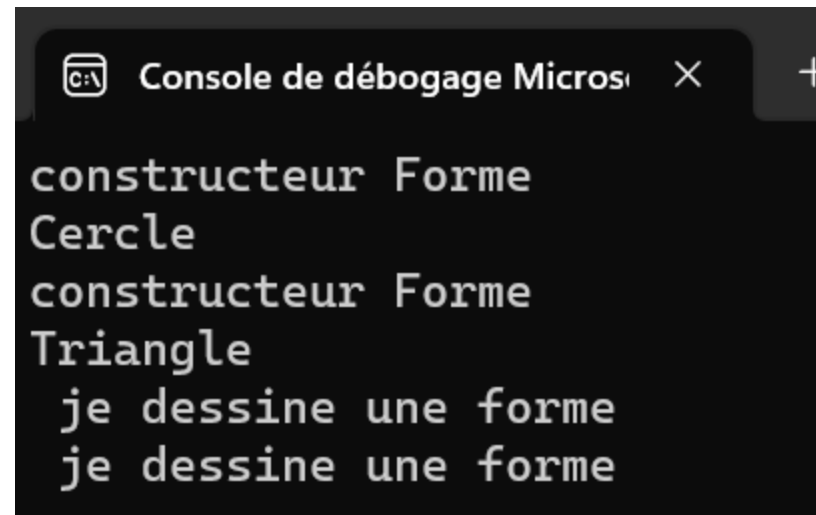
```
virtual type_retour nom_fonc(param) = 0 ;
```

□ Exemple : comportement non polymorphe

```
#include<iostream>
using namespace std;
class Forme {
public :
    Forme() {
        cout << "constructeur Forme"
        << endl;
    }
    void dessiner() {
        cout<<" je dessine une forme"<<
        endl;
    }
};
class Cercle : public Forme {
public:
    Cercle() {
        cout << "Cercle" << endl;
    }
    void dessiner() {
        cout << " je dessine un Cercle" <<
        endl;
    }
};
```

```
class Triangle : public Forme {
public:
    Triangle() {
        cout << "Triangle" << endl;
    }
    void dessiner() {
        cout << " je dessine un Triangle"
        << endl;
    }
};
void faireQuelqueChose(Forme& f) {
    f.dessiner();
}
int main() {
    Cercle c;
    Triangle t;
    faireQuelqueChose(c);
    Forme *f =& t;
    f->dessiner();
    return 0;
}
```

□ Affichage :



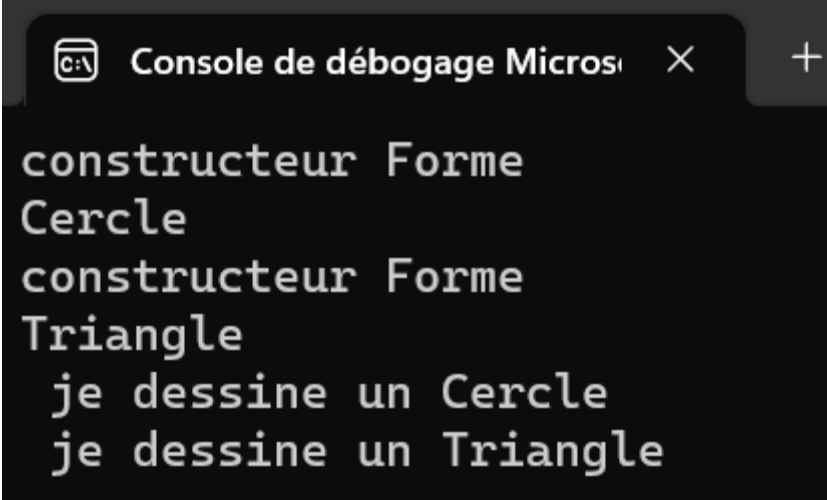
```
constructeur Forme
Cercle
constructeur Forme
Triangle
je dessine une forme
je dessine une forme
```

□ Exemple : comportement polymorphe

```
#include<iostream>
using namespace std;
class Forme {
public :
    Forme() {
        cout << "constructeur Forme" <<
        endl;
    }
    virtual void dessiner() {
        cout<<" je dessine une
        forme"<<endl;
    }
};
class Cercle : public Forme {
public:
    Cercle() {
        cout << "Cercle" << endl;
    }
    void dessiner() {
        cout << " je dessine un Cercle" <<
        endl;
    }
};
```

```
class Triangle : public Forme {
public:
    Triangle() {
        cout << "Triangle" << endl;
    }
    void dessiner() {
        cout << " je dessine un Triangle"
        << endl;
    }
};
void faireQuelqueChose(Forme& f) {
    f.dessiner();
}
int main() {
    Cercle c;
    Triangle t;
    faireQuelqueChose(c);
    Forme *f =& t;
    f->dessiner();
    return 0;
}
```


□ Affichage :



```
constructeur Forme
Cercle
constructeur Forme
Triangle
je dessine un Cercle
je dessine un Triangle
```

- ❑ Classe destinée uniquement à la dérivation
- ❑ Pas de possibilité de définir une instance d'une classe abstraite
- ❑ Toute classe comportant une fonction virtuelle pure est abstraite
- ❑ Possibilité de définir des pointeurs et des références sur une classe abstraite
- ❑ Il est obligatoire d'avoir une définition pour les fonctions virtuelles pures au niveau des classes dérivées

□ Exemple

```
#ifndef FORME
#define FORME
class Forme {
public:
    virtual float Perimetre() = 0;
    virtual float Aire() = 0;
    virtual float Volume() = 0;
    virtual void PrintName() = 0;
    virtual void Print() = 0;
};
#endif
```

