# Programmation C++

## Héritage et Polymorphisme

ING2-GSI

CY Tech

2023-2024

# Héritage

# Ex1 : Héritage

## Introduction

- L'héritage permet d'ajouter des propriétes à une classe existante pour en obtenir une nouvelle plus précise : c'est la relation **"est un"** plus quelque chose.
- *Exemple* :
  Un étudiant est une personne, il a un nom et un prénom (personne). De plus, il a un numéro d'étudiant.
  `Etudiant` est dérivé de `Personne`.
- Principe :
  - Déclarer une classe de base, dite aussi classe parente (Personne) pour déclarer une classe dérivéee (Etudiant).
  - La classe dérivée hérite tous les membres (données et méthodes) de la classe de base.

# Ex2 : Héritage

## Introduction

```
class Personne {
private :
    string nom;
public :
    string getNom() const;
};

class Etudiant : public Personne {
private :
    string id;
public :
    string getId() const;
};
```

# Ex2 : Héritage
## Principe

- L'héritage permet :
  - la réutilisation du code déjà écrit
  - l'ajout de nouvelles fonctionnalités
  - la modification d'un comportement existant (redéfinition)

```
class Personne {
private :
  string nom;
  public:
  string getNom() const;
  void afficher() const {cout << nom << endl;}
};

class Etudiant : public Personne {
private :
  string id;
public :
  string getId() const; // ajout fonctionnalites
  void afficher() const {  // modif. de l'existant
    Personne::afficher(); // reutilisation du code
    cout << id << endl;
  }
};
```

# Ex3 : Héritage

## Constructeur et Destructeur

- Le constructeur de la classe de base est appelé **avant** le constructeur de la classe dérivée.
  - Soit on appelle explicitement un constructeur de la classe mère dans le constructeur de la classe fille
  - Soit le constructeur par défaut de la classe mère est appelé automatiquement (le compilateur envoie un message d'erreur si celui n'existe pas)
- Les destructeurs sont appelés dans l'**ordre inverse**.

# Ex3 : Héritage

## Constructeur et Destructeur

```
class Personne {
private :
    string nom;
public :
    Personne(string pNom) : nom{pNom}{}
};

class Etudiant : public Personne {
private :
    string id;
public :
    Etudiant(string pNom, string pId) :
        Personne(pNom),id{pId}{};
};
```

# Héritage : publique ou privé ?

```
class cBase {
    ...
};

class cDerivee : public|protected|private cBase {
    ...
};
```

# Héritage : publique ou privé ?

| Mode de dérivation | Statut dans la classe de base | Statut dans la classe dérivée |
|---|---|---|
| public | public<br>protected<br>private | public<br>protected<br>inaccessible |
| protected | public | protected |
| | protected | protected |
| | private | inaccessible |
| private | public | private |
| | protected | private |
| | private | inaccessible |

# Héritage : publique ou privé ?

- Héritage publique = "is-a"
- Héritage privé= "is-implemented-in-terms-of"
  - Pas un vrai héritage

```
class Etudiant : private Personne{...};
void danser(const Personne & p);
Etudiant e;
danser(e); //Erreur !
```

  - Réutilisation du code

# Héritage multiple

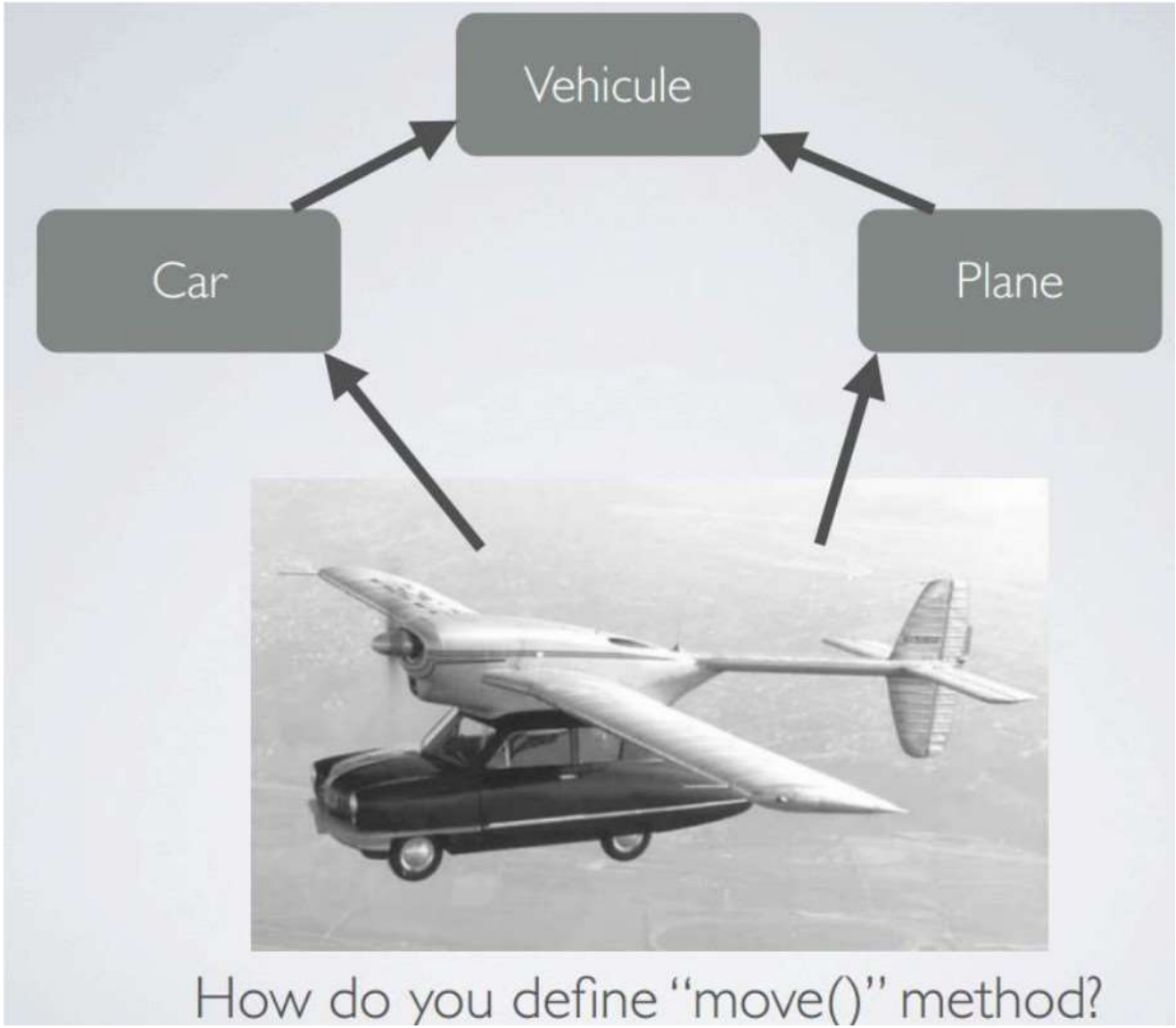
```
class A { ...
};

class B { ...
};

class C : public A, public B { ...
};
```

- La classe C hérite de A et B : une instance de C possède à la fois les données et les fonctions membres de A et B.
- Quand un objet C est créé le constructeur de A est appelé en premier, en suite celui de B (dans l'ordre).
- Quand un objet C est détruit, le destructeur de B est appelé en premier, après celui de A (sens inverse).

# Héritage multiple



# Polymorphisme

# Ex3 : Polymorphisme

- Un objet *polymorphe* est un objet susceptible de prendre plusieurs formes pendant l'exécution.
- Le *polymorphisme* représente la capacité du système à choisir **dynamiquement** la méthode qui correspond au type de l'objet en cours de manipulation.

```
Personne *p = new Etudiant("Bob","e31415");
p->afficher(); // methode de Personne ?
               // de Etudiant ?
```

- Le polymorphisme est implémenté en C++ avec les fonctions virtuelles (**virtual**) et l'héritage.

# Ex4 : Polymorphisme

```
class Polygon {
protected :
    int width, height;
public:
    virtual int getArea() {return 0;}
};

class Rectangle : public Polygon {
public:
    int getArea() override { return width*height; }
};

class Triangle : public Polygon {
public:
    int getArea() override { return width*height/2; }
};
```

# Ex5 : Polymorphisme

```
vector<const Polygon *> vp;
Polygon * p1 = new Rectangle(4,5);
Polygon * p2 = new Triangle(4,5);
vp.push_back(p1);
vp.push_back(p2);

for (auto p : vp) {
    cout << p->getArea() << endl;
}

// n'oubliez pas delete !
```

# Ex5 : Polymorphisme

## Destructeur virtuel

```
class Base {
    // des methodes virtuelles
};

class Derivee : public Base {
    ~Derivee(){
        // "supprimer" ce qu'il faut
    }
};
Base *b = new Derivee();
// utilisation de b
delete b; // Warning : deleting object of polymorphic
// class type 'Base' which has non-virtual destructor
// might cause undefined behavior
```

# Ex5 : Polymorphisme

## Destructeur virtuel

```
Base *b = new Derivee();
delete b; // Warning !
```

- Le destructeur de `Base` n'étant pas virtuel et b étant une `Base*` pointant sur un objet dérivé, `delete b` a un comportement indéfini.
- Dans la plupart des implémentations, l'appel du destructeur sera résolu comme n'importe quel code non virtuel
  - le destructeur de la classe `Base` sera appelé,
  - pas celui de la classe `Derivee`, ce qui entrainera une fuite de mémoire.
- Toujours rendre les **destructeurs** des classes mères **virtuels** lorsqu'ils doivent être manipulés de manière polymorphe.

```
class Base {
    virtual ~Base();
};
```

# Classe abstraite

# Classe abstraite

- Une classe est dite **abstraite** si elle contient au moins une fonction virtuelle pure.
- Une fonction membre est dite virtuelle **pure lorsqu'elle** est déclarée de la façon suivante :

```
virtual type nomMethode(parametres) = 0;
```

- Une classe abstraite ne peut pas être instanciée : on ne peut pas créer d'objet à partir d'une classe abstraite.
- Il est *obligatoire* d'avoir une définition pour les fonctions virtuelles pures au niveau des classes dérivées.

# Classe abstraite

```
class Polygon {
protected :
    int width, height;
public:
    virtual int getArea() = 0;
};
```

- La classe `Polygon` ne peut pas être instanciée : elle est dite abstraite
- La méthode `getArea()` est virtuelle pure et ne posséde aucune définition : toutes les classes dérivées doivent contenir une méthode `getArea()`
- Une classe dans laquelle il n'y a plus une seule fonction virtuelle pure est dite concrète et devient instanciable.

# Pour résumer

```
class Shape {
public :
    // fonction virtuelle pure
    virtual void draw() const = 0;
    // fonction virtuelle
    virtual void error(const string & msg);
    // fonction non-virtuelle
    int objectID() const;
    ...
};

class Rectangle : public Shape { ... };

class Ellipse : public Shape {...};
```

# Pour résumer

- Fonction virtuelle pure : héritage d'interface
  - classes dérivées doivent hériter l'interface de la fonction
  - classes dérivées concrètes doivent fournir l'implémentation de la fonction
- Fonction virtuelle : héritage d'interface + héritage d'implémentation par défaut
  - classes dérivées doivent hériter l'interface de la fonction
  - elles peuvent hériter l'implémentation par défaut si elles veulent, ou elles peuvent la redéfinir.
- Fonction non-virtuelle : héritage d'interface + héritage d'implémentation obligatoire
  - classes dérivées doivent hériter l'interface et l'implémentation de la fonction (invariance par rapport aux spécialisations)