

# Héritage multiple en C++

## Intérêt et ambiguïté

Guillaume Revy

`guillaume.revy@univ-perp.fr`

Université de Perpignan **Via Domitia**



# Récapitulatif des séances précédentes

- **Encapsulation**  $\rightsquigarrow$  rassembler les données et méthodes au sein d'une structure
  - ▶ en cachant l'implantation de l'objet (à l'utilisateur, au moins)
  - ▶ protection de l'accès aux données  $\rightsquigarrow$  mécanisme d'accès / modification
- **Héritage**  $\rightsquigarrow$  permettre la création d'une classe à partir d'une classe existante
  - ▶ la classe **dérivée** (*filles*) contient **sous certaines conditions** les attributs et les méthodes de la classe *mère* (*super-classe*)
  - ▶ permet ainsi la réutilisation de code
- **Polymorphisme**  $\rightsquigarrow$  permettre de redéfinir dans une classe dérivée les méthodes dont elle hérite de sa classe mère
  - ▶ une même méthode possède alors plusieurs formes

# Plan du cours

1. Qu'est ce que l'héritage multiple ?
2. Appel des constructeurs et destructeurs
3. Ambiguïté sur les attributs et les méthodes

# Plan du cours

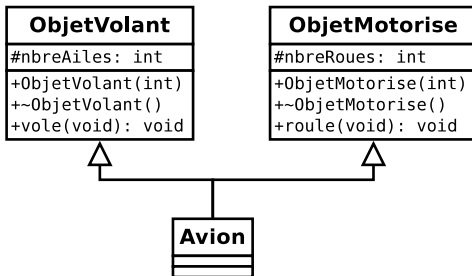
1. Qu'est ce que l'héritage multiple ?
2. Appel des constructeurs et destructeurs
3. Ambiguïté sur les attributs et les méthodes

# Qu'est ce que l'héritage multiple ?

- En C++, une classe peut hériter de plusieurs classes mères  $\rightsquigarrow$  on parle d'héritage multiple
  - ▶ la classe dérivée hérite des attributs et méthodes de **toutes** ses classes mères
  - ▶ **attention** : ce n'est pas possible dans tous les langages orientés objet

# Qu'est ce que l'héritage multiple ?

- En C++, une classe peut hériter de plusieurs classes mères  $\rightsquigarrow$  on parle d'**héritage multiple**
  - ▶ la classe dérivée hérite des attributs et méthodes de **toutes** ses classes mères
  - ▶ **attention** : ce n'est pas possible dans tous les langages orientés objet
- Par exemple, on considère une classe **Avion**



# Comment définir l'héritage multiple ?

- La définition de l'héritage multiple se fait de manière similaire à celle de l'héritage simple.

```
// Classe.hpp

#ifndef __CLASSE_HPP__  // <-- directives de precompilation
#define __CLASSE_HPP__  //

class Classe : <private|protected|public> ClasseMere1,
              <private|protected|public> ClasseMere2, ...
{
private:
    // ...
protected:
    // ...
public:
    // ...
};

#endif // __CLASSE_HPP__
```

## Exemple de la classe Avion

```
class ObjetVolant
{
protected:
    int nbreAiles;

public:
    ObjetVolant(int n)
    {
        nbreAiles = n;
        std::cout << "Demarrage d'un objet volant a " << nbreAiles << " ailes" << std
            ::endl;
    }
    virtual ~ObjetVolant()
    {
        std::cout << "Arret de l'objet volant a " << nbreAiles << " ailes" << std::
            endl;
    }
    void vole(void){ std::cout << "ca vole..." << std::endl; }
};
```



## Exemple de la classe Avion

```
class ObjetMotorise
{
protected:
    int nbreRoues;

public:
    ObjetMotorise(int n)
    {
        nbreRoues = n;
        std::cout << "Demarrage d'un objet motorise a " << nbreRoues << " ailes" <<
            std::endl;
    }
    virtual ~ObjetMotorise()
    {
        std::cout << "Arret de l'objet motorise a " << nbreRoues << " ailes" << std::
            endl;
    }
    void vole(void){ std::cout << "ca roule..." << std::endl; }
};
```

## Exemple de la classe Avion

```
class ObjetVolant
{
    /* ... */
};

class ObjetMotorise
{
    /* ... */
};

class Avion: public ObjetVolant, public ObjetMotorise
{
    // ...
};
```

### ■ Remarques :

- ▶ la classe Avion hérite des attributs (`nbreAiles` et `nbreRoues`) et des méthodes (`vole` et `roule`) des classes `ObjetVolant` et `ObjetRoulant`
- ▶ pour une classe dérivée donnée, il n'y a pas de restrictions sur le nombre de classes mères
- ▶ l'ordre des déclarations des classes mères a une incidence sur les appels des constructeurs et destructeurs

# Plan du cours

1. Qu'est ce que l'héritage multiple ?
2. Appel des constructeurs et destructeurs
3. Ambiguïté sur les attributs et les méthodes

# Appel aux constructeurs et destructeurs

- De la même manière que pour l'héritage simple, pour l'héritage multiple, l'initialisation des instances des classes de bases se fait sur le même modèle que l'initialisation des membres constants  $\rightsquigarrow$  avant l'appel du constructeur

```
Classe::Classe( <liste des parametres> ): ClasseMere1(...),  
                                         ClasseMere2(...), ...  
{  
    // ...  
}
```

# Appel aux constructeurs et destructeurs

- De la même manière que pour l'héritage simple, pour l'héritage multiple, l'initialisation des instances des classes de bases se fait sur le même modèle que l'initialisation des membres constants  $\rightsquigarrow$  avant l'appel du constructeur

```
Classe::Classe( <liste des parametres> ): ClasseMere1(...),  
                                         ClasseMere2(...), ...  
{  
    // ...  
}
```

- Lorsqu'une classe admet un constructeur par défaut, il n'est pas forcément de l'invoquer explicitement

# Appel aux constructeurs et destructeurs

- De la même manière que pour l'héritage simple, pour l'héritage multiple, l'initialisation des instances des classes de bases se fait sur le même modèle que l'initialisation des membres constants  $\rightsquigarrow$  avant l'appel du constructeur

```
Classe::Classe( <liste des parametres> ): ClasseMere1(...),  
                                       ClasseMere2(...), ...  
{  
    // ...  
}
```

- Lorsqu'une classe admet un constructeur par défaut, il n'est pas forcément de l'invoquer explicitement
- Si le destructeur de la classe de base a été déclaré virtuel, la destruction de tous les objets de la chaînes d'héritage se fait automatiquement

# Retour à la classe Avion

```
class Avion: public ObjetVolant, public ObjetMotorise
{
public:
    Avion():ObjetMotorise(2), ObjetVolant(2)
    {
        std::cout << "Demarrage d'un avion" << std::endl;
    }
    ~Avion()
    {
        std::cout << "Arret d'un avion" << std::endl;
    }
};
```

# Retour à la classe Avion

```
class Avion: public ObjetVolant, public ObjetMotorise
{
public:
    Avion():ObjetMotorise(2), ObjetVolant(2)
    {
        std::cout << "Demarrage d'un avion" << std::endl;
    }
    ~Avion()
    {
        std::cout << "Arret d'un avion" << std::endl;
    }
};
```

```
Demarrage d'un objet volant a 2 ailes
Demarrage d'un objet motorise a 2 ailes
Demarrage d'un avion
Arret d'un avion
Arret de l'objet motorise a 2 ailes
Arret de l'objet volant a 2 ailes
```

- Les constructeurs sont appelés dans l'ordre d'apparition dans la déclaration de l'héritage, et non dans l'ordre des appels aux constructeurs



# Retour à la classe Avion

```
class Avion: public ObjetVolant, public ObjetMotorise
{
public:
    Avion():ObjetMotorise(2), ObjetVolant(2)
    {
        std::cout << "Demarrage d'un avion" << std::endl;
    }
    ~Avion()
    {
        std::cout << "Arret d'un avion" << std::endl;
    }
};
```

```
Demarrage d'un objet volant a 2 ailes
Demarrage d'un objet motorise a 2 ailes
Demarrage d'un avion
Arret d'un avion
Arret de l'objet motorise a 2 ailes
Arret de l'objet volant a 2 ailes
```

- Les constructeurs sont appelés dans l'ordre d'apparition dans la déclaration de l'héritage, et non dans l'ordre des appels aux constructeurs
- Les destructeurs sont appelés dans l'ordre inverse de celui des constructeurs

# Plan du cours

1. Qu'est ce que l'héritage multiple ?
2. Appel des constructeurs et destructeurs
3. Ambiguïté sur les attributs et les méthodes

## Exemple d'ambiguïté sur la classe Avion

- Une ambiguïté peut se produire lorsque le nom d'un attribut ou d'une méthode est le même dans deux classes mères différentes
- **Par exemple** : on ajoute la méthode `roule` à la classe `ObjetVolant`.

```
class ObjetVolant
{
    // ...
    void roule(void){ "ca roule..."; }
    void vole(void){ "ca vole..."; }
};

class ObjetMotorise
{
    // ...
    void roule(void){ "ca roule..."; }
};

// ...
int
main(void)
{
    Avion a;  a.roule();    // ERREUR : erreur de compilation, meme si les deux
                        // methodes roule() n'avaient pas eu la meme signature

    return 0;
}
```

# Solution 1

⇒ utilisation de l'opérateur de résolution de portée

```
class Avion: public ObjetVolant, public ObjetMotorise
{
public:
    Avion():ObjetMotorise(2), ObjetVolant(2)
    {
        std::cout << "Demarrage d'un avion" << std::endl;
    }
    ~Avion()
    {
        std::cout << "Arret d'un avion" << std::endl;
    }
};

int
main(void)
{
    Avion a;
    a.ObjetVolant::roule();
}
```

# Solution 1

↪ utilisation de l'opérateur de résolution de portée

```
class Avion: public ObjetVolant, public ObjetMotorise
{
public:
    Avion():ObjetMotorise(2), ObjetVolant(2)
    {
        std::cout << "Demarrage d'un avion" << std::endl;
    }
    ~Avion()
    {
        std::cout << "Arret d'un avion" << std::endl;
    }
};

int
main(void)
{
    Avion a;
    a.ObjetVolant::roule();
}
```

- ⊖ C'est le développeur, et non l'utilisateur, de la classe Avion qui doit décider quelle méthode appeler

# Solution 2

↪ redéfinition de la méthode dans la classe dérivée

```
class Avion: public ObjetVolant, public ObjetMotorise
{
public:
    Avion():ObjetMotorise(2), ObjetVolant(2)
    {
        std::cout << "Demarrage d'un avion" << std::endl;
    }
    ~Avion()
    {
        std::cout << "Arret d'un avion" << std::endl;
    }

    void roule(void){ ObjetVolant::roule(); }
};

int
main(void)
{
    Avion a;
    a.roule();
}
```

# Solution 3

↪ indication explicite de quelle méthode appelée

```
class Avion: public ObjetVolant, public ObjetMotorise
{
public:
    Avion():ObjetMotorise(2), ObjetVolant(2)
    {
        std::cout << "Demarrage d'un avion" << std::endl;
    }
    ~Avion()
    {
        std::cout << "Arret d'un avion" << std::endl;
    }

    using ObjetVolant::roule;                // ATTENTION : pas de parenthese !!
};

int
main(void)
{
    Avion a;
    a.roule();
}
```

# Questions ?