

Cours Langage C++ : Héritage et polymorphisme

Programmation Orienté Objet

Thierry Vaira

BTS IRIS Avignon

tvaira@free.fr © v0.1



Sommaire

- 1 Classes et objets
- 2 L'héritage
- 3 Polymorphisme
- 4 Classe abstraite
- 5 Transtypage

Classes et objets

- Les classes sont les éléments de base de la **programmation orientée objet (POO)** en C++. Dans une classe, on réunit :
 - des **données variables** : les données membres, ou les **attributs** de la classe.
 - des **fonctions** : les fonctions membres, ou les **méthodes** de la classe.
- Une classe A apporte un nouveau type **A** ajouté aux types (pré)définis de base par C++.
- Une variable a créée à partir du type de classe A est appelée **instance** (ou **objet**) de la classe A.

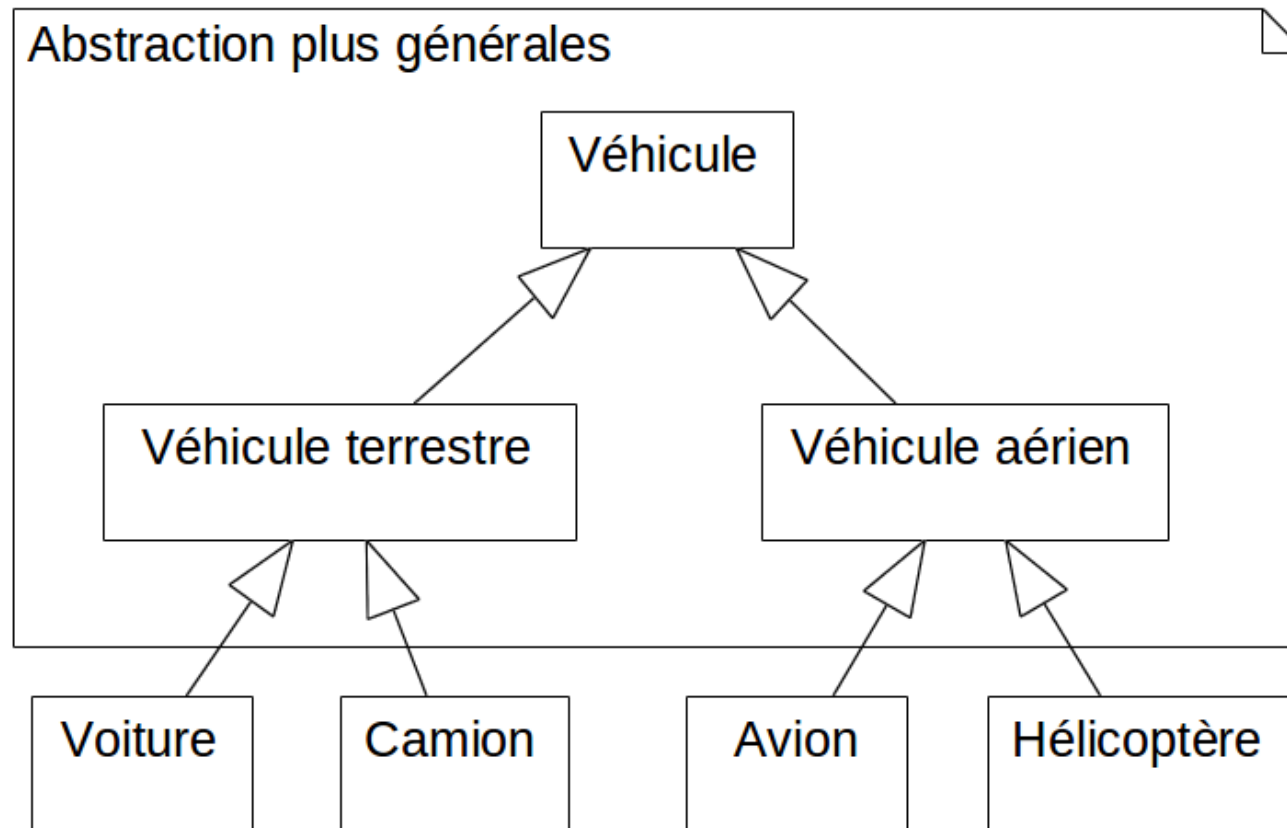
Exemple (non compilable*) :

```
class A; // déclare une classe A (* car elle n'est pas définie)

A a1; // instancie un objet a1 de type A
A a2; // crée une instance a2 de type A
```

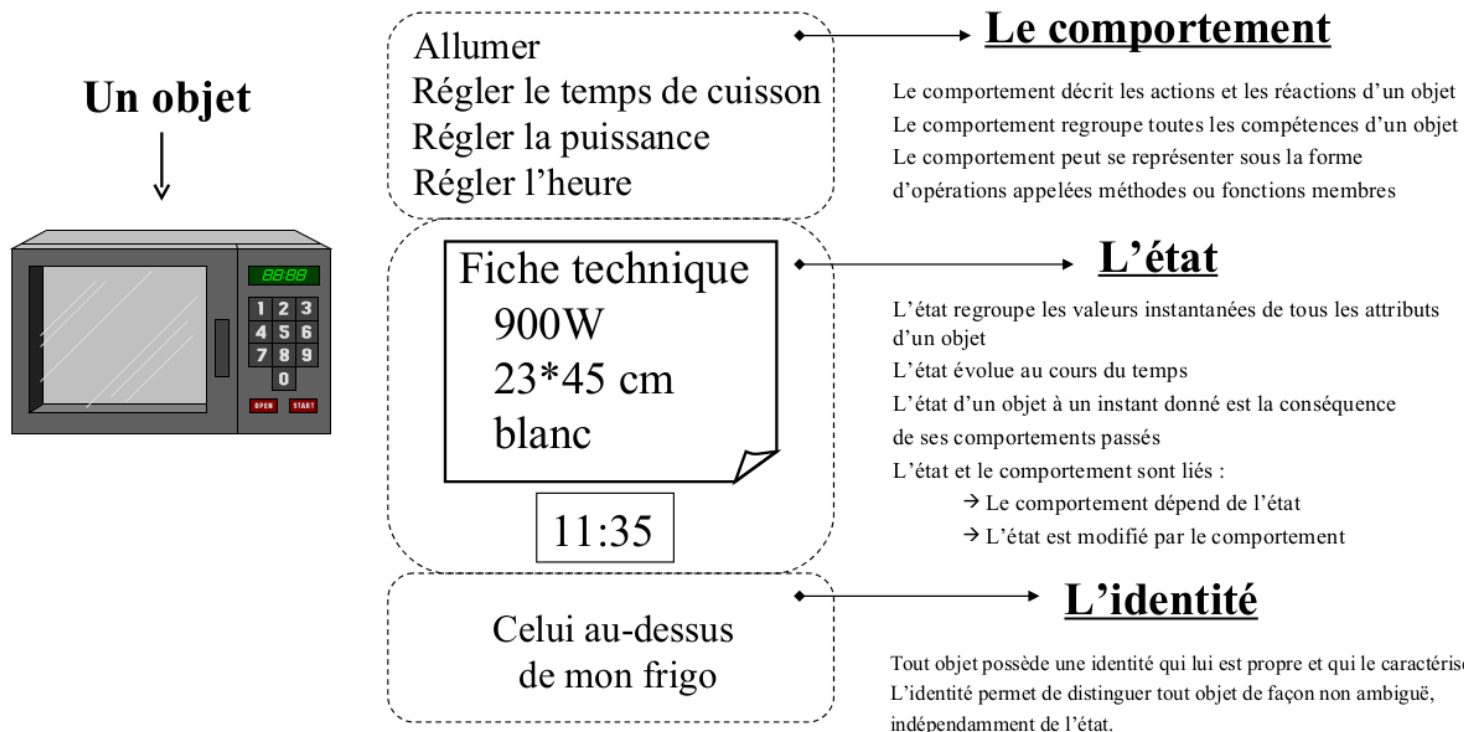
Classifier

- Regrouper des objets suivants des critères de ressemblance s'appelle **classer** (**classifier**) :
- La classe est une **description abstraite d'un ensemble d'objets**



État et comportement

- Un objet possède une **identité** qui permet de distinguer un objet d'un autre objet (son nom, une adresse mémoire).
- Un objet possède un **état** (les valeurs contenues dans les attributs propres à cet objet).
- Un objet possède un **comportement** (l'utilisation de ses méthodes lui fera changer d'état).

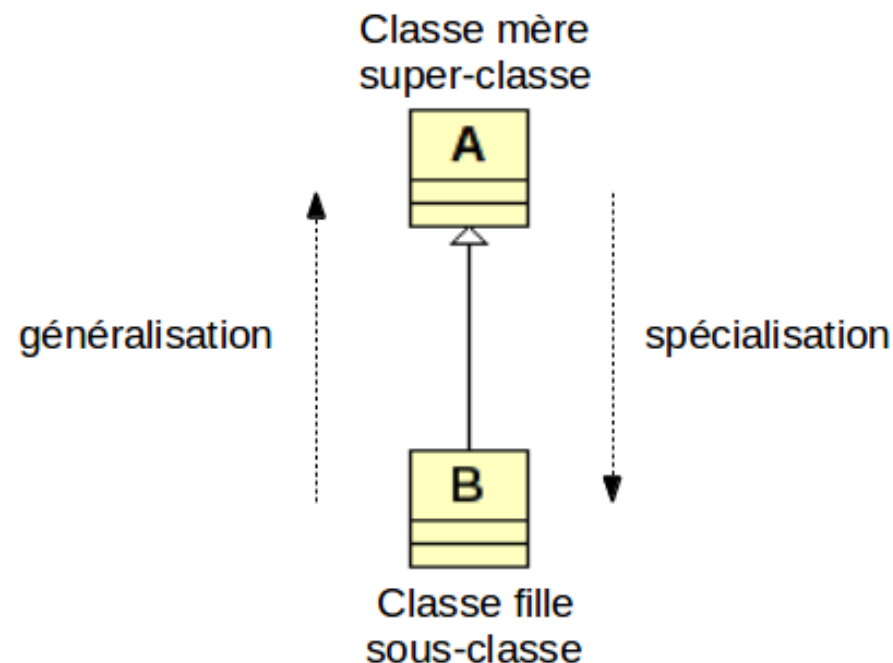


Droits d'accès aux membres

- Les droits d'accès aux membres d'une classe concernent aussi bien les méthodes que les attributs.
- En C++, on dispose des droits d'accès suivants :
 - **Accès public** : on peut utiliser le membre de n'importe où dans le programme.
 - **Accès private** : seule une fonction membre de la même classe A peut utiliser ce membre ; il est invisible de l'extérieur de A.
 - **Accès protected** : ce membre peut être utilisé par une fonction de cette même classe A, et pas ailleurs dans le programme (ressemble donc à `private`), mais il peut en plus être utilisé par une classe B qui hérite de A.

Définition

- L'**héritage** (ou **spécialisation**, ou **dérivation**) permet d'**ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise**.
- L'idée est : "un B **est un** A avec des choses en plus".



Exemple : Un étudiant **est** une personne, et a donc un nom (et un prénom, ...). De plus, il a un numéro INE.

```
// Personne est la classe parente ou mère de Etudiant (en anglais superclass)
class Personne {
    private: string nom;
    public: string getNom() const { return nom; }
};

// Etudiant est la classe fille ou dérivée de Personne : Etudiant hérite ou
// descend de Personne
class Etudiant : public Personne {
    private: string ine;
    public: string getINE() const { return ine; }
};

Personne unePersonne; Etudiant unEtudiant;

unePersonne.getNom(); // légal : unePersonne est une Personne
unEtudiant.getINE(); // légal : unEtudiant est un Etudiant
unEtudiant.getNom(); // légal : unEtudiant est une Personne (par héritage)
unePersonne.getINE(); // illégal : unePersonne n'est pas un Etudiant (erreur: '
    class Personne' has no member named 'getINE')
```


Types d'héritage

| mode de dérivation | Statut dans la classe de base | Statut dans la classe dérivée |
|--------------------|-------------------------------|-------------------------------|
| public | public | public |
| | protected | protected |
| | private | inaccessible |
| protected | public | protected |
| | protected | protected |
| | private | inaccessible |
| private | public | private |
| | protected | private |
| | private | inaccessible |

Remarque : Les constructeurs, le destructeur, de même que l'opérateur = de la classe de base ne sont pas hérités dans la classe dérivée.

Conversion automatique (1/2)

Si B hérite de A, alors toutes les instances de B sont aussi des instances de A, et il est donc possible de faire :

```
Personne unePersonne; Etudiant unEtudiant;

unePersonne = unEtudiant; // Ok !

// Propriété conservée lorsqu'on utilise des pointeurs :
Personne *pUnePersonne; Etudiant *pUnEtudiant = &unEtudiant;
pUnePersonne = pUnEtudiant; // pointer sur un B c'est avant tout pointer sur un A

// Évidemment, l'inverse n'est pas vrai :
unEtudiant = unePersonne; // ERREUR !
// Pareil pour les pointeurs :
pUnePersonne = &unePersonne; pUnEtudiant = pUnePersonne; // ERREUR !
// pointer sur un A n'est pas pointer sur un B
```

Conversion automatique (2/2)

Conclusion : Traiter un type dérivé comme s'il était son type de base est appelé **transtypage ascendant** ou surtypage (*upcasting*).

A l'opposé, le **transtypage descendant** (*downcast*) pose un problème particulier car leur vérification n'est possible qu'à l'exécution. Il nécessite l'utilisation d'opérateur de cast : `dynamic_cast` (vu plus tard).

Principes (1/2)

L'héritage permet :

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

Remarque : la suppression de membres en utilisation l'héritage privé (`private`).

Principes (2/2)

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

// Classe dérivée
class Etudiant : public Personne {
    private: string ine;
    public:
        string getINE() const; // ajout de nouvelles fonctionnalités
        void afficher() const // modification d'un comportement existant
        {
            Personne::afficher(); // réutilisation du code déjà écrit
            cout << ine << endl;
        }
};
```

Notion de redéfinition

Il ne faut pas mélanger la redéfinition et la surdéfinition :

- Une **surdéfinition ou surcharge (*overloading*)** permet **d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe avec une signature différente.**
- Une **redéfinition (*overriding*)** permet **de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer.** Elle doit avoir une signature rigoureusement identique à la méthode parente.

Un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire ou de la compléter.

Appel de constructeurs (1/2)

- Comme une instance de Etudiant (classe dérivée) est avant tout une instance de Personne (classe de base), dans le constructeur de Etudiant, on explicite la façon de créer l'instance Personne dans la **liste d'initialisation**.
- Le constructeur de Personne (classe de base) est appelé **avant** le constructeur de Etudiant (classe dérivée).

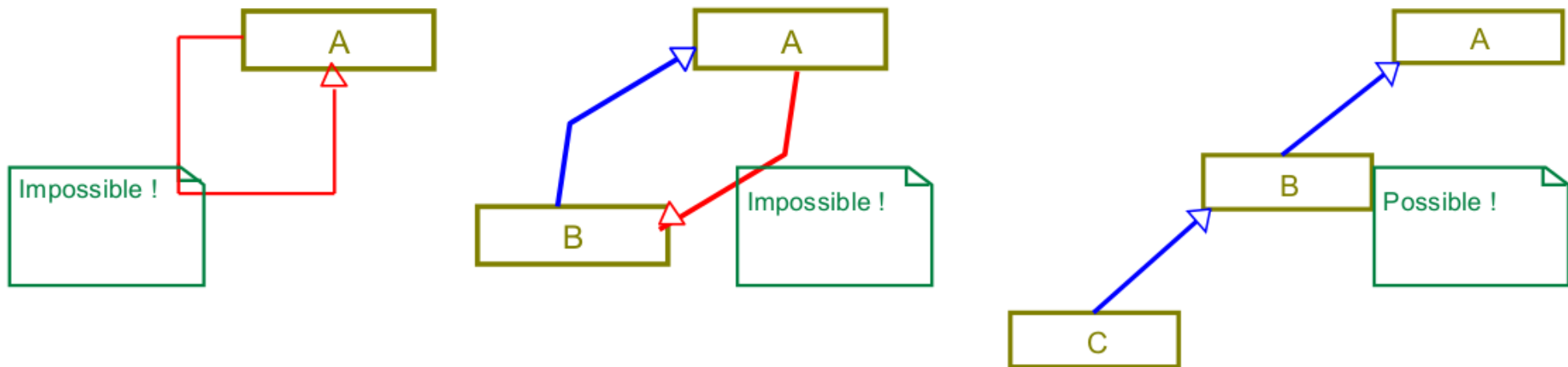
Remarque : les destructeurs sont appelés dans l'ordre inverse.

Appel de constructeurs (2/2)

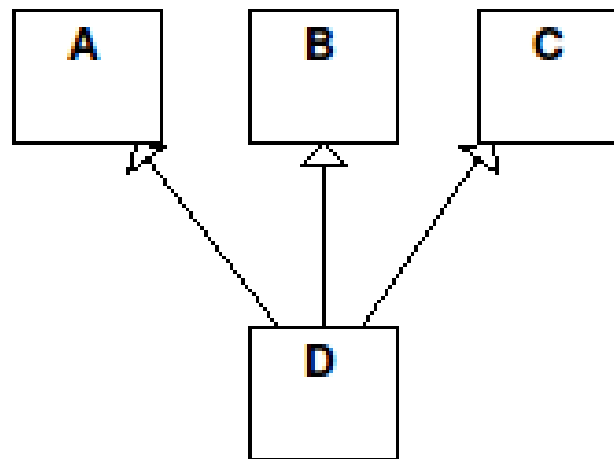
```
// Classe de base
class Personne {
    private: string nom;
    public:
        Personne(string n) { nom = n }
};

// Classe dérivée
class Etudiant : public Personne {
    private: string ine;
    public:
        Etudiant(string nom, string ine) : Personne(nom), ine(ine) { }
};
```

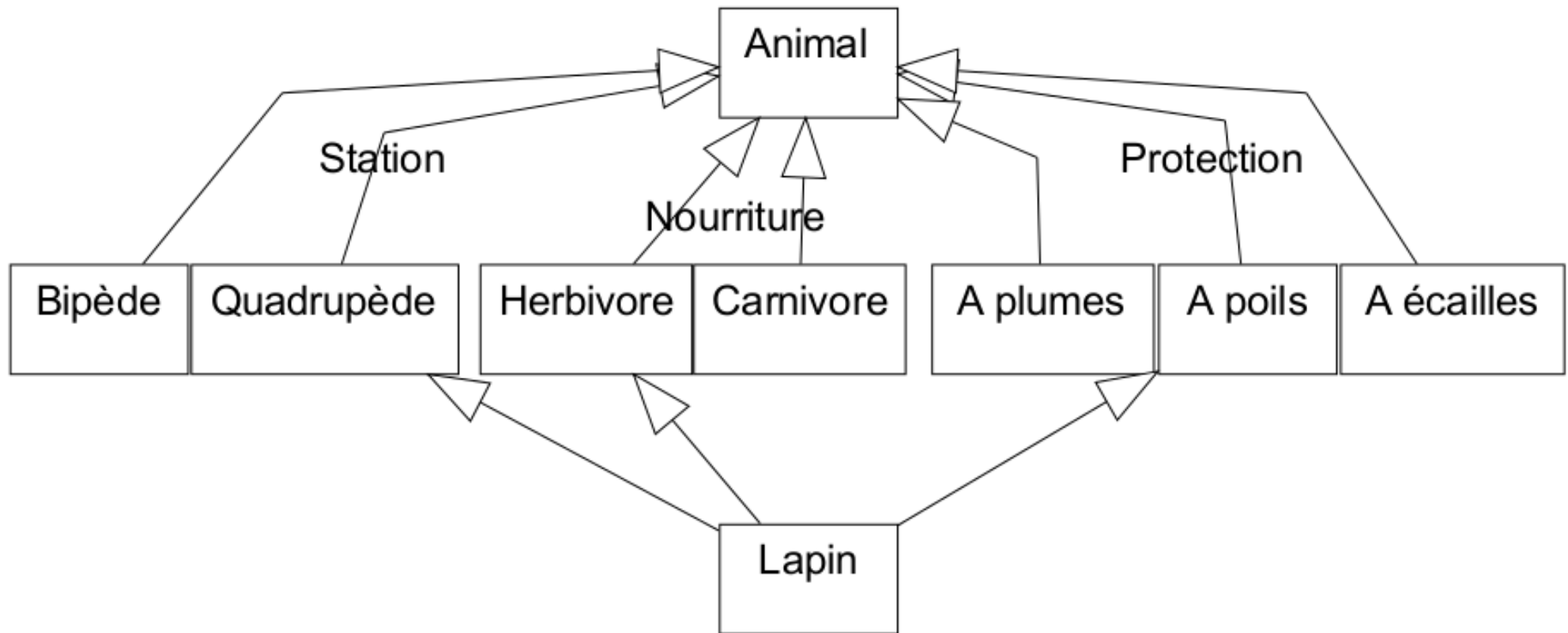

Propriétés de la généralisation



Remarque : en C++, une classe peut être dérivée à partir de plusieurs classes de bases (héritage multiple)



Héritage multiple



Remarque : en C++, une classe peut être dérivée à partir de plusieurs classes de bases mais elle peut aussi être dérivée plusieurs fois de la même classe de base d'une manière indirecte.

Lire : l'héritage en diamant sur

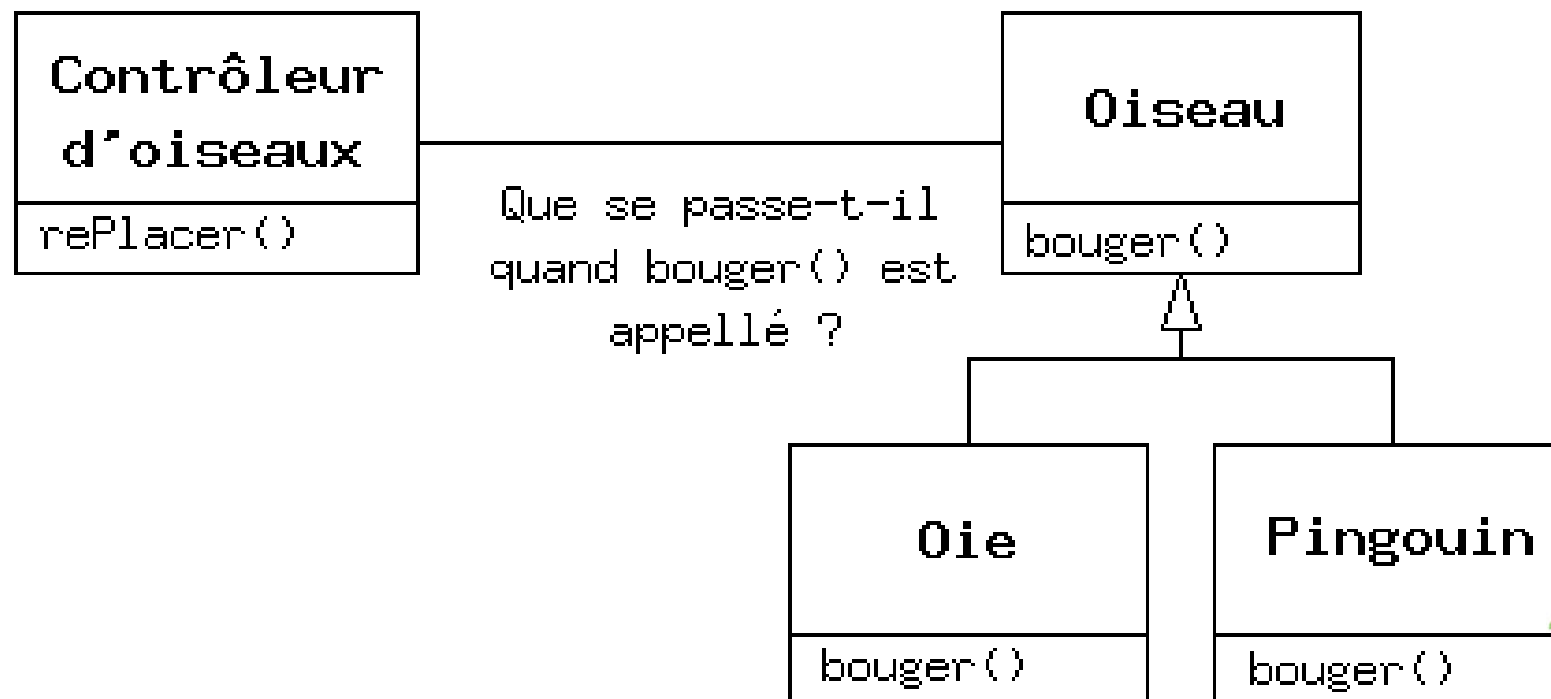
loic-joly.developpez.com/articles/heritage-multiple/.

Définitions

- Le **polymorphisme** est un moyen de manipuler des objets hétéroclites de la même manière, pourvu qu'ils disposent d'une interface commune.
- 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.
- Le **polymorphisme** est implémenté en **C++** avec **les fonctions virtuelles (virtual) et l'héritage**.

Comportement Polymorphe (1/4)

- Par exemple, dans le diagramme suivant, l'objet Contrôleur d'oiseaux travaille seulement avec des objets Oiseaux génériques, et ne sait pas de quel type ils sont. C'est pratique du point de vue de Contrôleur d'oiseaux, car il n'a pas besoin d'écrire du code spécifique pour déterminer le type exact d'Oiseau avec lequel il travaille, ou le comportement de cet Oiseau.



Comportement Polymorphe (2/4)

- Comment se fait-il donc que, lorsque `bouger()` est appelé tout en ignorant le type spécifique de l'Oiseau, on obtienne le bon comportement (une Oie court, vole ou nage, et un Pingouin court ou nage) ?
- La réponse constitue l'astuce fondamentale de la programmation orientée objet : le compilateur ne peut faire un appel de fonction au sens traditionnel du terme. Un appel de fonction généré par un compilateur non orienté objet crée ce qu'on appelle une association prédéfinie : le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter.
- En POO, le programme ne peut déterminer l'adresse du code avant la phase d'exécution, un autre mécanisme est donc nécessaire quand un message est envoyé à un objet générique.

Comportement Polymorphe (3/4)

- Pour résoudre ce problème, les langages orientés objet utilisent le concept d'**association tardive**. Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution. Le compilateur s'assure que la fonction existe et vérifie le type des arguments et de la valeur de retour, mais il ne sait pas exactement quel est le code à exécuter.
- Pour créer une association tardive, le compilateur C++ insère une portion spéciale de code en lieu et place de l'appel absolu. Ce code calcule l'adresse du corps de la fonction, en utilisant des informations stockées dans l'objet. Ainsi, chaque objet peut se comporter différemment suivant le contenu de cette portion spéciale de code. Quand un objet reçoit un message, l'objet sait quoi faire de ce message.

Comportement Polymorphe (4/4)

- On déclare qu'on veut une fonction qui ait la flexibilité des propriétés de l'association tardive en utilisant le mot-clé `virtual`.
- On n'a pas besoin de comprendre les mécanismes de `virtual` pour l'utiliser, mais sans lui on ne peut pas faire de la programmation orientée objet en C++.
- En C++, on doit se souvenir d'ajouter le mot-clé `virtual` (devant une méthode) parce que, par défaut, les fonctions membres ne sont pas liées dynamiquement. Les fonctions virtuelles permettent d'exprimer des différences de comportement entre des classes de la même famille.
- Ces différences sont ce qui engendre un **comportement polymorphe**.

Exemple : comportement non polymorphe (1/3)

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

class Forme {
public:
    Forme() { cout << "constructeur Forme <|- "; }
    void dessiner() { cout << "je dessine ... une forme ?\n"; }
};

class Cercle : public Forme {
public:
    Cercle() { cout << "Cercle\n"; }
    void dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
public:
    Triangle() { cout << "Triangle\n"; }
    void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```


Exemple : comportement non polymorphe (2/3)

```
void faireQuelqueChose(Forme &f)
{
    f.dessiner(); // dessine une Forme
}

int main()
{
    Cercle c;
    Triangle t;

    faireQuelqueChose(c); // avec un cercle
    faireQuelqueChose(t); // avec un triangle

    return 0;
}
```

L'exécution du programme d'essai nous montre que nous n'obtenons pas un comportement polymorphe puisque c'est la méthode dessiner() de la classe Forme qui est appelée :

Exemple : comportement non polymorphe (3/3)

```
constructeur Forme <|- Cercle  
constructeur Forme <|- Triangle  
je dessine ... une forme ?  
je dessine ... une forme ?
```

Exemple : comportement polymorphe (1/3)

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

class Forme {
public:
    Forme() { cout << "constructeur Forme <- "; }
    // la méthode dessiner sera virtuelle et fournira un comportement polymorphe
    virtual void dessiner() { cout << "je dessine ... une forme ?\n"; }
};

class Cercle : public Forme {
public:
    Cercle() { cout << "Cercle\n"; }
    void dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
public:
    Triangle() { cout << "Triangle\n"; }
    void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

Exemple : comportement polymorphe (2/3)

```
void faireQuelqueChose(Forme &f)
{
    f.dessiner(); // dessine une Forme
}

int main()
{
    Cercle c;
    Triangle t;

    faireQuelqueChose(c); // avec un cercle
    faireQuelqueChose(t); // avec un triangle

    return 0;
}
```

L'exécution du programme d'essai nous montre maintenant que nous obtenons un comportement polymorphe puisque c'est la "bonne" méthode `dessiner()` qui est appelée :

Exemple : comportement polymorphe (3/3)

```
constructeur Forme <|- Cercle  
constructeur Forme <|- Triangle  
je dessine un Cercle !  
je dessine un Triangle !
```


Conclusion : Le C++ permet, par le polymorphisme, que des objets de types différents (`Cercle`, `Triangle`, ...) répondent différemment à un même appel de fonction (`dessiner()`).

Appel de destructeurs : problème

```

class A {
    private:    int *p;
    public:
        A() {p = new int[4]; cout << "A()";}
        ~A() {delete [] p; cout<< " ~A()" << endl;}
};
class B : public A {
    private:    int *q;
    public:
        B() {q = new int[64]; cout<< "B() et q=" << q;}
        ~B() {delete [] q; cout<< " ~B()";}
};
int main() {
    A *pA = new B(); delete pA; // A()B() et q=0x100170 ~A()
}

```

L'affichage met en évidence un problème de "fuite" de mémoire. Le destructeur de B n'est jamais appelé ! 

Appel de destructeurs : solution

```

class A {
    private:    int *p;
    public:
        A() {p = new int[4]; cout << "A()";}
        virtual ~A() {delete [] p; cout<< " ~A()" << endl;}
};
class B : public A {
    private:    int *q;
    public:
        B() {q = new int[64]; cout<< "B() et q=" << q;}
        ~B() {delete [] q; cout<< " ~B()";}
};
int main() {
    A *pA = new B(); delete pA; // A()B() et q=0x100170 ~B() ~A()
}

```

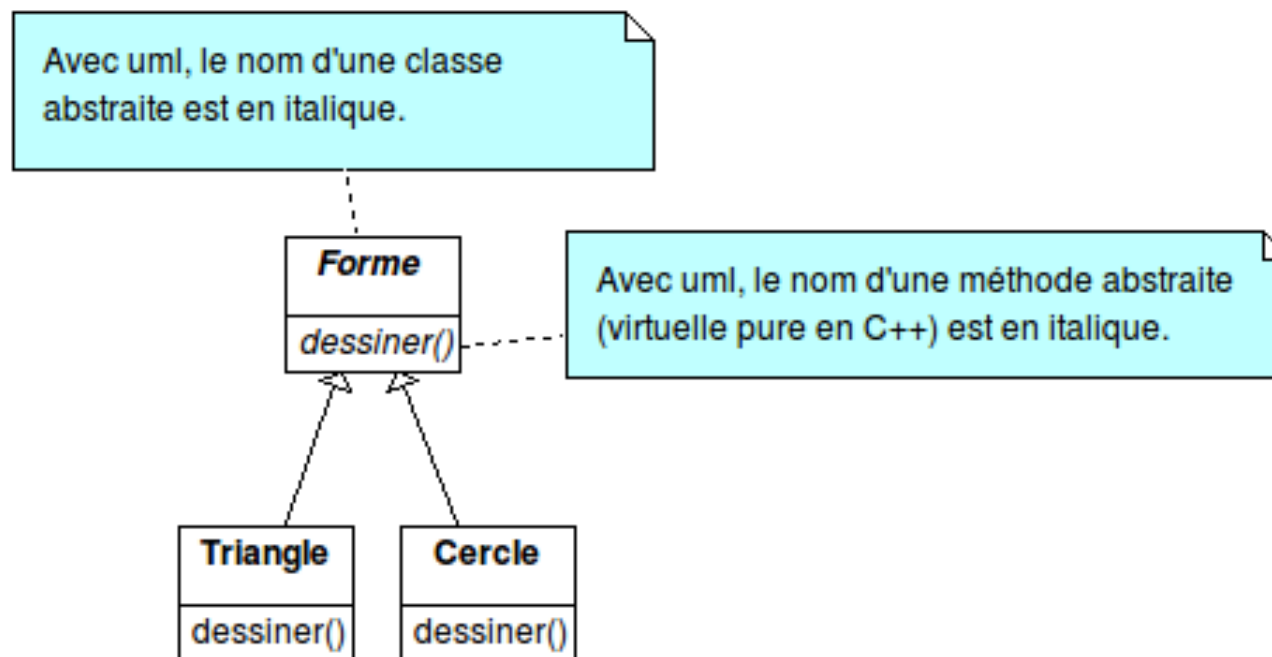
Avec la déclaration du destructeur de A en virtuel (`virtual`), le destructeur de B est correctement appelé.

Classe abstraite en C++

- 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(paramètres) = 0;`
- Une classe abstraite ne peut pas être instanciée : on ne peut 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 en UML

- Une classe abstraite permet d'introduire certaines fonctions virtuelles dont on ne peut encore donner aucune définition.
- Dans cet exemple, on ne sait pas programmer ce que doit faire la fonction `dessiner()` dans le contexte de `Forme`. On veut juste s'assurer de sa présence dans toutes les classes filles, sans devoir la définir dans la classe parente.



Exemple : classe abstraite

```
// La classe Forme ne peut pas être instanciée : elle est dite abstraite
class Forme {
    public:
        Forme() {}
        // la méthode dessiner est virtuelle pure et ne possède aucune définition
        virtual void dessiner() = 0; // cela oblige tous les descendants à contenir
            une méthode dessiner()
};

// Une classe dans laquelle il n'y a plus une seule fonction virtuelle pure est
// dite concrète et devient instanciable
class Cercle : public Forme {
    public:
        Cercle() {}
        void dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
    public:
        Triangle() {}
        void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

Le transtypage en C++

- Le transtypage (*cast* ou conversion de type) en C++ permet la conversion d'un type vers un autre.
- Nouvelle syntaxe de l'opérateur traditionnel :
 - En C : (nouveau type)(expression à transtyper);
 - En C++ : type(expression à transtyper);
- Nouvel opérateur de transtypage :
 - `static_cast` : Opérateur de transtypage à tout faire. Ne permet pas de supprimer le caractère `const` ou `volatile`.
 - `const_cast` : Opérateur spécialisé et limité au traitement des `const` et `volatile`
 - `dynamic_cast` : Opérateur spécialisé et limité au traitement des *downcast*.
 - `reinterpret_cast` : Opérateur spécialisé dans le traitement des conversions de pointeurs peu portables.

- La syntaxe est la suivante :

`op_cast<expression type>(expression à transtyper);` où `op` prend l'une des valeurs (`static`, `const`, `dynamic` ou `reinterpret`)

Transtypage « ascendant »

Traiter un type dérivé comme s'il était son type de base est appelé transtypage ascendant, surtypage ou généralisation (*upcasting*). Cela ne pose donc aucun problème.

Exemple : transtypage « ascendant »

```
class Forme {};  
class Cercle : public Forme {};  
class Triangle : public Forme {} ;  
  
void faireQuelqueChose(Forme &f) { f.dessiner(); }  
  
...  
  
Cercle c;  
  
// Un Cercle est ici passé à une fonction qui attend une Forme.  
// Comme un Cercle est une Forme, il peut être traité comme tel par  
    faireQuelqueChose()  
faireQuelqueChose(c);
```

Transtypage « descendant »

Conversion d'un pointeur sur un objet d'une classe générale vers un objet d'une classe spécialisée.

Exemple : transtypage « ascendant »

```
class Animal { public: virtual ~Animal() {} };
class Chien : public Animal {};
class Chat : public Animal {};

int main() {
    Animal* a = new Chat; // Transtypage ascendant

    // On essaye de le transtyper en Chat* :
    Chat* c1 = a; // Erreur : invalid conversion from 'Animal*' to 'Chat*'

    Chat* c2 = dynamic_cast<Chat *>(a); // Valide : Transtypage descendant
}
```