

Chapitre 12

Héritage

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 12 : Héritage

217

1. Généralités

L'héritage est le troisième des paradigmes de la programmation orientée objet (le 1^{er} étant l'encapsulation, le 2^e la structure de classe).

L'emploi de l'héritage conduit à un style de programmation par raffinements successifs et permet une programmation incrémentielle effective.

L'héritage peut être simple ou multiple.

Il représente la relation: EST-UN

Exemple:

Chat est un animal

Moto est un véhicule

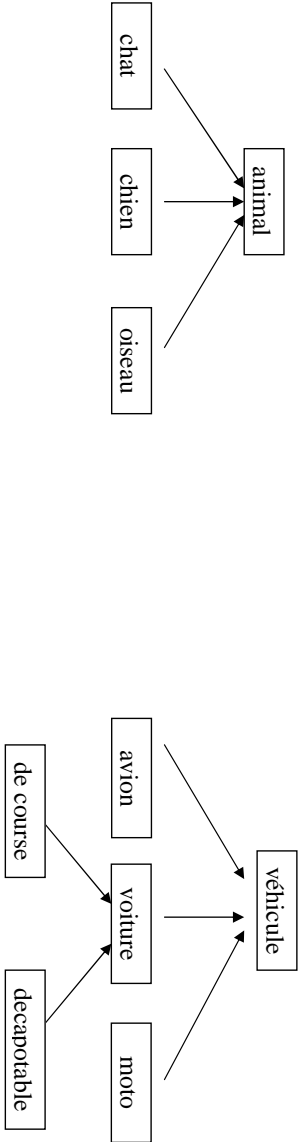
Cercle est une forme

Alors que l'objet membre représente la relation: A-UN

Une voiture a un moteur

L'héritage est mis en œuvre par la construction de classes dérivées.

Le graphe de l'héritage est comme suit:



La classe animal est la classe de base (classe supérieure),
Les classes chat, chien et oiseau sont des classes dérivées (sous-classes).

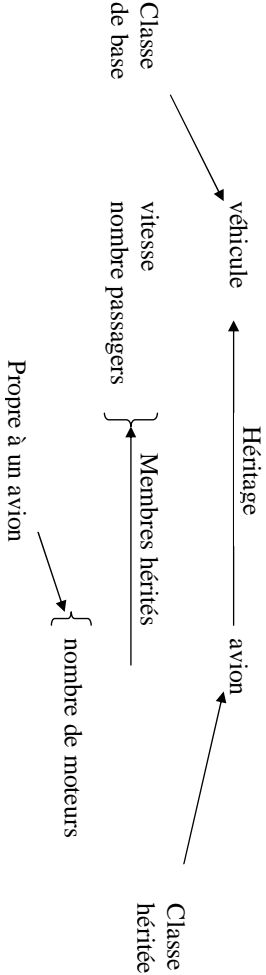
2. Une classe dérivée

Une classe dérivée modélise un cas particulier de la classe de base, et est donc enrichie d'informations supplémentaires.

La classe dérivée possède les propriétés suivantes:

- contient les données membres de la classe de base,
- peut en posséder de nouvelles,
- possède (à priori) les méthodes de sa classe de base,
- peut redéfinir (masquer) certaines méthodes,
- peut posséder de nouvelles méthodes.

La classe dérivée hérite des membres de la classe de base.



3. Syntaxe de l'héritage

```
class classe_dérivée:protection classe_de_base { /* etc. */ }
```

Les types de protections sont :

public, protected et private

4. Déclarations

```
#include <iostream>
using namespace std;
class vehicule {

    double vitesse;
    int nbre_passagers;

public:

    // Fonction remplace le constructeur.
    void init_vehicule(double v, int np) {
        vitesse = v;
        nbre_passagers = np;
    }
}
```

```
// Pour afficher les membres private.
void affiche() {
    cout << "vitesse: " << vitesse << " ; nbre_passagers: " \
        << nbre_passagers << endl;
}

// avion hérite publiquement de véhicule.
class avion:public vehicule {
    int nbre_moteurs;

public:
    // fonction remplace le constructeur
    void init_avion(int nm) {
        nbre_moteurs = nm;
    }
};

// moto hérite publiquement de véhicule.
class moto:public vehicule {
    double cylindre;

public:
    // Fonction remplace le constructeur
    void init_moto(double cy) {
        cylindre = cy;
    }
};

int main() {
    vehicule v;
    avion boeing;
    moto suzuki;
    return 0;
}
```

5. Accès aux membres hérités

Si la classe dérivée hérite publiquement de la classe de base,

- les membres de la classe dérivée auront accès aux membres publics (champs et méthodes) de la classe de base,
- par contre ils n'auront pas accès aux membres privés de la classe de base.

Pour l'exemple du paragraphe 4, nous pouvons écrire la fonction `main` suivante:

```
int main() {  
    avion boeing767;  
    boeing767.init_avion(2);  
  
    // Fonctions héritées de la classe de base.  
    boeing767.init_vehicule(950, 200);  
    boeing767.affiche(); // affiche: 950, 200 mais pas 2.  
    // Erreur: accès à des données privées de la classe de base.  
    boeing767.nbre_moteurs=4;  
    boeing767.vitesse=800;  
  
    // Erreur: accès à des données privées de la classe de dérivée.  
    boeing767.nbre_passagers=188;  
    return 0;  
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

6. Redéfinition des fonctions membres

La fonction `affiche` de la page précédente est membre de la classe de base `vehicule`. Elle n'affiche que les membres privés de cette classe. On ne peut donc pas afficher le nombre de moteurs.

Pour faire cela, nous allons définir dans la classe dérivée une fonction portant le même nom, et qui aura pour rôle d'afficher les données privées de la classe dérivée. On parle alors de redéfinition (ou surcharge) d'une fonction de la classe de base.

```
class avion:public vehicule {  
    int nbre_moteurs;  
public:  
    // Fonction remplace le constructeur  
    void init_avion(int nm) {  
        nbre_moteurs = nm;  
    }  
    // Pour afficher les membres private  
    void affiche() {  
        cout << "vitesse: " << vitesse << " ";  
        cout << "nbre_passagers: " << nbre_passagers << endl;  
        cout << "nbre_moteurs: " << nbre_moteurs << endl;  
    }  
};
```

La fonction `affiche` dans ce cas-là ne va pas fonctionner, il y aura erreur de compilation.

En effet, la classe `avion` n'a pas le droit d'accéder aux membres privés de la classe de base.

Comment afficher alors les données privées de la classe de base et celles de la classe dérivée et cela par l'utilisation d'une fonction dans la classe dérivée?

```
class avion:public vehicule {
    int nbre_moteurs;

public:
    // Fonction remplace le constructeur.
    void init_avion(int nm) {
        nbre_moteurs = nm;
    }

    // Pour afficher les membres private.
    void affiche() {
        vehicule::affiche(); // appel de la fonction affiche de la classe de base.
        cout << "nbre_moteurs: " << nbre_moteurs << endl;
    }
};
```

Puisque `affiche` de la classe de base (ici `vehicule`) est accessible, on fera donc appel à elle à partir de la fonction `affiche` de la classe dérivée.

La nouvelle définition de `avion::affiche` cache l'ancienne définition (celle de `vehicule::affiche`) accessible via héritage.

```
int main() {
    avion boeing767;
    boeing767.init_avion(2);

    // Fonctions héritées de la classe de base
    boeing767.init_vehicule(950,200);

    // Fonction de la classe dérivée
    // Appel de avion::affiche et non pas vehicule::affiche
    boeing767.affiche();

    return 0;
}
```

7. Constructeurs et destructeurs

Pour construire un `avion`, il faut construire d'abord un `vehicule`;

Le constructeur de la classe de base (`vehicule`) est donc appelé **avant** le constructeur de la classe dérivée (`avion`).

De façon symétrique, le destructeur de la classe de base (`vehicule`) est appelé **après** le destructeur de la classe dérivée (`avion`).

```
#include <iostream>
using namespace std;
class vehicule {
    double vitesse;
    int nbre_passagers;
public:
    // Constructeur
    vehicule(double v,int np) {
        vitesse = v;
        nbre_passagers = np;
    }
    ~vehicule() {}
    // Pour afficher les membres private.
    void affiche() {
        cout << "vitesse: " << vitesse << " ; nbre_passagers: " << nbre_passagers \
        << endl;
    }
};

class avion:public vehicule {
    int nbre_moteurs;
public:
    // Constructeur de la classe dérivée
    avion(int nm,double v,int np):vehicule(v,np) {
        nbre_moteurs = nm;
    }
    void affiche() {
        vehicule::affiche(); // appel de la fonction affiche de la classe de base.
        cout << "nbre_moteurs: " << nbre_moteurs << endl;
    }
};
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 12 : Héritage

227

```
int main() {
    avion boeing767(2,950,200);
    boeing767.affiche();
    return 0;
}
```

Si la classe de base a un constructeur autre que celui par défaut, la classe dérivée doit avoir un constructeur, sinon il est impossible de créer un objet.

Si dans l'appel du constructeur de la classe dérivée, le nom du constructeur de la classe de base n'est pas mentionné explicitement, le constructeur par défaut de la classe de base sera pris en compte. Si la classe de base ne possède pas ce constructeur, il y aura alors une erreur de compilation.

Question : Citez dans quels cas la classe de base ne possèdera pas de constructeur par défaut?

© Mohamed N. Lokbani

v3.00

POO avec C++

8. Contrôle des accès

Les droits d'accès protègent les données et les méthodes, et réalisent aussi l'encapsulation.

Les droits d'accès sont accordés aux fonctions membres, ou aux fonctions globales.

L'unité de protection est la classe: tous les objets de la classe bénéficient de la même protection.

Il y a 3 catégories de protection:

- un membre **public** est accessible à toute fonction,
- un membre **private** n'est accessible qu'aux fonctions membre de la classe ou aux fonctions amies,
- un membre **protected** n'est accessible qu'aux fonctions membres de la classe de base ou des classes dérivées ou aux fonctions amies.

```
#include <iostream>
using namespace std;
class simule {
    int a;
    void fa();
protected:
    int b;
    void fb();
public:
    int c;
    void fc();
};
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 12 : Héritage

```
class vide:public simule{
public:
    void test() {
        cout << a << endl; // Erreur car a est déclarée private.
        fa(); // Erreur car fa est déclarée private.

        // La fonction test accède aux données public et protected.

        cout << b << endl; // ok car b est déclarée protected.
        fb(); // ok car fb est déclarée protected.
        cout << c << endl; // ok car c est déclarée public.
        fc(); // ok car fc est déclarée public.
    }
};

int main() {
    simule rien;

    cout << rien.a << endl; // Erreur car a est déclarée private.
    rien.fa(); // Erreur car fa est déclarée private.
    cout << rien.b << endl; // Erreur car b est déclarée protected.
    rien.fb(); // Erreur car fb est déclarée protected.
    cout << rien.c << endl; // ok car c est déclarée public.
    rien.fc(); // ok car fc est déclarée public.
    return 0;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

		Mode de dérivation		
		public	protected	private
statut dans la classe de base	public protected private	public protected private	protected protected private	private protected private

Mode de dérivation:

- public: le plus utilisé
- protected: rarement utilisé
- private: pour complètement réécrire l'interface. De ce fait la notion EST-UN disparaît, on ne parle plus donc d'héritage.

9. Constructeur de recopie

On doit recopier les champs de la classe dérivée et ceux de la classe de base.

```
class B {  
    /* classe de base */  
};  
  
class D:public class B {  
    /* classe dérivée */  
};
```

Deux cas peuvent se présenter:

1/ la classe D n'a pas de constructeur de recopie:

Les appels des constructeurs se feront comme suit:

- constructeur de recopie par défaut de la classe D,
- constructeur de recopie explicite ou par défaut de la classe B.

2/ la classe D a un constructeur de recopie :

Les appels des constructeurs se feront comme suit:

- le compilateur appelle ce constructeur de recopie, c'est à ce constructeur de recopie d'appeler celui de la classe de base (s'il veut, et habituellement on veut qu'il le fasse). Si on ne fait pas cet appel, et si la classe de base n'a pas de constructeur de recopie explicite, alors c'est le constructeur par défaut qui est appelé. Si la classe de base n'en possède pas un, c'est le constructeur avec arguments par défaut qui est appelé, s'il n'en existe pas un, il y aura erreur de compilation.

```
#include <iostream>
using namespace std;
class vehicule {
    double vitesse;
    int nbre_passagers;
public:
    vehicule(double,int); // constructeur.
    vehicule(const vehicule& v2); // constructeur de recopie.
    ~vehicule(); // destructeur.
    // Pour afficher les membres private.
    void affiche();
};
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 12 : Héritage

```
// avion hérite publiquement de véhicule.
class avion:public vehicule {
    int nbre_moteurs;
public:
    avion(int,double,int); // constructeur.
    avion(const avion&); // constructeur de recopie.
    ~avion(); // destructeur.
    // Pour afficher les membres private.
    void affiche();
};

// Constructeur de véhicule.
vehicule::vehicule(double v,int np) {
    vitesse = v;
    nbre_passagers = np;
    cout << "C.Veh.: " << this << endl;
}

// Destructeur de véhicule.
vehicule::~vehicule() {
    cout << "D.Veh.: " << this << endl;
}

// Constructeur de recopie.
vehicule::vehicule(const vehicule& v2) {
    vitesse = v2.vitesse;
    nbre_passagers = v2.nbre_passagers;
    cout << "R.Veh.: " << this << endl;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

```
// Constructeur de avion, qui appelle le constructeur de véhicule.
avion::avion(int nm,double v,int np):vehicule(v,np) {
    nbre_moteurs = nm;
    cout << "C.Av.: " << this << endl;
}

// Constructeur de recopie de avion, qui appelle explicitement le constructeur de recopie de véhicule.
// a2 est converti en objet du type véhicule.
avion::avion(const avion& a2):vehicule(a2) {
    nbre_moteurs = a2.nbre_moteurs;
    cout << "R.Av.: " << this << endl;
}

// Destructeur de avion.
avion::~avion() {
    cout << "D.Av.: " << this << endl;
}

// Fonction pour afficher les données private de la classe de véhicule.
void vehicule::affiche() {
    cout << "vitesse: " << vitesse << " ; nbre_passagers: " << nbre_passagers << endl;
}

// Fonction pour afficher les données private de des classes base et dérivée.
void avion::affiche() {
    vehicule::affiche(); // appel de la fonction affiche de la classe de base.
    cout << "nbre_moteurs: " << nbre_moteurs << endl;
}
```

```
int main() {
    vehicule va(110,4); // création de l'objet va
    vehicule vcopie(va); // création par recopie de l'objet vcopie

    // Création de l'objet aa du type avion, appelle aussi le constructeur de véhicule.
    avion aa(2,200,180);

    // Création de l'objet bcopie du type avion, appelle aussi le constructeur de recopie de véhicule.
    avion bcopie(aa);

    // Affichage en sortie.
    vcopie.affiche();
    bcopie.affiche();

    return 0;
}
```

Sortie:

C.Veh.: 0xbffff7ac ➡ constructeur objet va.
R.Veh.: 0xbffff7a0 ➡ constructeur de recopie pour vcopie.
C.Veh.: 0xbffff790 ➡ constructeur objet aa: appelle d'abord constructeur véhicule,
C.Av.: 0xbffff790 ➡ puis le constructeur avion.

```
R.Veh. : 0xbffff780 ← constructeur de copie pour bcopie; appelle d'abord
                    constructeur de copie de véhicule,
R.Av. : 0xbffff780 ← puis constructeur de copie pour l'objet bcopie.
vitesse: 110 ; nbre_passagers : 4 ← fonction affiche de véhicule.
vitesse: 200 ; nbre_passagers : 180 ← fonction affiche de avion.
nbre_moteurs : 2

D.Av. : 0xbffff780 ← destructeur de bcopie, puis
D.Veh. : 0xbffff780 ← celui de sa classe de base.

D.Av. : 0xbffff790 ← destructeur de aa, puis
D.Veh. : 0xbffff790 ← celui de sa classe de base.

D.Veh. : 0xbffff7a0 ← destructeur de vcopie.

D.Veh. : 0xbffff7ac ← destructeur de va.
```

10. Opérateur d'affectation

2 cas peuvent se présenter:

1/ classe dérivée n'a pas surdéfini l'opérateur d'affectation =

Dans ce cas, le compilateur appelle:

-a- l'opérateur = de la classe de base (par défaut ou surdéfini).

Sinon:

-b- l'opérateur = par défaut de la classe dérivée.

2/ classe dérivée a surdéfini l'opérateur d'affectation =

Dans ce cas, le compilateur appelle seulement cet opérateur, à lui d'appeler l'opérateur = de la classe de base, s'il veut (habituellement, oui!).

Pour l'exemple du paragraphe 9, nous obtenons ce qui suit:

```
class vehicule {
    double vitesse;
    int nbre_passagers;
public:
    // etc.
    vehicule& operator=(const vehicule&);
};

// avion hérite publiquement de vehicule.
class avion:public vehicule {
    int nbre_moteurs;
public:
    // etc.
    avion& operator=(const avion&);
};
```

// Opérateur d'affectation de la classe de base.

```
vehicule& vehicule::operator=(const vehicule& v2) {
    if (this != &v2) {
        vitesse = v2.vitesse;
        nbre_passagers = v2.nbre_passagers;
    }
    cout << "A.Veh.: " << this << endl;
    return *this;
}
```

// Opérateur d'affectation de la classe dérivée.

```
avion& avion::operator=(const avion& a2) {
    if (this != &a2) {
        vehicule::operator=(a2);
        nbre_moteurs = a2.nbre_moteurs;
    }
    cout << "A.av.: " << this << endl;
    return *this;
}
```

```
int main() {
    vehicule v1(110,4);
    vehicule v2(240,2);

    v2 = v1; ← opérateur d'affectation de la classe véhicule.
    avion ava(2,200,180);
    avion avb(4,300,250);

    avb = ava; ← opérateur d'affectation de la classe de base, puis de la classe dérivée.
    V2.affiche();
    avb.affiche();

    return 0;
}
```

Une autre manière décrit l'opérateur d'affectation de la classe dérivée est comme suit:

```
avion& avion::operator=(avion& a2) {
    if (this != &a2) {
        vehicule* ptrv = this;
        vehicule* ptra = &a2;
        // utilise l'opérateur d'affectation de véhicule.
        *ptrv = *ptra;
        nbre_moteurs = a2.nbre_moteurs;
    }
    cout << "A.av.: " << this << endl;
    return *this;
}
```

Nous avons modifié l'entête de la fonction `operator=` pour passer l'argument comme une variable non constante, à cause de cette instruction:

```
vehicule* ptra = &a2; // a2 ne peut pas être constante.
```

Question : Comment faudrait-il réécrire ce code, pour que l'argument `a2` reste constant?

11. Typage statique vs. Typage dynamique

Type statique d'une variable: type à la compilation, type déclaré.
Type dynamique d'une variable: type à l'exécution, type en mémoire.

Par défaut, nous utilisons le typage statique, car il est moins coûteux (espace/temps). Nous pouvons utiliser aussi le typage dynamique, en employant le mot clé réservé: `dynamic_cast` (pour plus de détails, voir chapitre suivant).

12. Compatibilité entre objets d'une classe de base et objets d'une classe dérivée

Un objet d'une classe dérivée peut toujours être utilisé au lieu d'un objet de sa classe de base. (Applicable que dans le cas de la dérivation `public`).

Par exemple, un avion est un véhicule. Mais l'inverse n'est pas vrai, un véhicule n'est pas nécessairement un avion.

Soit l'exemple suivant:

```
vehicule v(300,4);
avion a(800,350,3);

vehicule* ptrv;
avion* ptra;

ptrv = &v;
ptra = &a;
```

1^{er} cas:

```
v = a
```

Conversion implicite de tout avion EST-UN véhicule. Le compilateur fait une copie en ignorant les membres excédentaires (`nbre_moteurs`).

Supposez que tous les membres des classes base et dérivée ont été déclarés `public`, nous aurons ce qui suit:

```
cout << v.vitesse << endl; // ok.
cout << v.nbre_moteurs << endl; // erreur car véhicule n'a pas d'information sur
// le nombre de moteurs.
```

2^e cas:

```
a = v // erreur
```

Un `véhicule` n'est pas forcément un `avion`. On ne peut pas deviner quelles seront les valeurs manquantes (Dans cet exemple: `nbre_moteurs`). Un `véhicule` n'a pas toutes les données d'un `avion`.

3^e cas:

```
ptrv = ptra // ok,
```

Mais ...

```
cout << ptrv->vitesse << endl; // ok
cout << ptrv->nbre_moteurs << endl; // erreur
```

À la compilation, `ptrv` pointe sur un `véhicule`, par la suite même si `ptrv` prend le pointeur de `avion`, il ne reconnaît pas les autres éléments (ne seront pas accessibles, ne les reconnaît pas).

4^e cas:

```
ptra = ptrv // erreur
```

Pour que ça marche, il faut "caster" (forcer le changement de type) le pointeur `ptrv`, c.-à-d.:

```
ptra = (avion *) ptrv // ok
```

```
cout << ptra.nbre_moteurs << endl;
```

Il affiche n'importe quelle valeur (par exemple: -1073743912) car `nbre_moteurs` n'a pas été initialisée (ne contient aucune donnée).

Récapitulatif

- Conversion de classe dérivée à classe de base // ok
- Conversion de classe de base à classe dérivée // Erreur, conversion interdite
- Conversion de (classe dérivée) * à (classe de base) * // ok
- Conversion de (classe de base) * à (classe dérivée) * // accepté avec un cast explicite.

Réponse à la question posée au paragraphe 7, page 227 : dans le cas où nous avons défini explicitement un constructeur (sans valeurs par défaut) dans cette classe de base (masque la présence du constructeur par défaut) ET nous n'avons pas défini, en plus, un constructeur sans arguments (pour recréer le constructeur par défaut).

Réponse à la question posée au paragraphe 10, page 240 : Il faut réécrire le code de l'opérateur d'affectation comme suit:

```
avion& avion::operator=(const avion& a2) {  
    if (this != &a2) {  
        vehicule* ptrv = this;  
        vehicule* ptra = const_cast <avion*> (&a2);  
        // Utilise l'opérateur d'affectation de véhicule.  
        *ptrv = *ptra;  
        nbre_moteurs = a2.nbre_moteurs;  
    }  
    cout << "A.av.: " << this << endl;  
    return *this;  
}
```