Université de Monastir

Cours: Programmation C++

Chapitre 5 : L'héritage en C++

Réalisé par:
Dr. Sakka Rouis Taoufik
https://github.com/srtaoufik/coursCpp

I. L'héritage: Définition

L'héritage (appelé dérivation de classes) est le fait de définir une nouvelle classe (appelée classe fille) en se basant sur la définition d'une classe déjà existante.

La nouvelle classe hérite alors les attributs et les méthodes de la classe de base (ou mère) et ce en plus de ses propres membres (ses attributs et/ou méthodes spécifiques).

Chapitre 5 : L'héritage en C++

I. L'héritage : Syntaxe de dérivation en C++

class NomClasseDérivée : [Mode_Dérivation] NomClasseDeBase {
// Liste des nouveaux membres de la classe dérivée
};

• ***NomClasseDérivée*** : désigne la nouvelle classe créée par dérivation.

• ***NomClasseDeBase*** : désigne le nom de la classe à partir de laquelle se fait la dérivation.

• ***Mode_Dérivation*** : sert à indiquer le mode avec lequel sera faite la dérivation. Ce mode détermine les droits d'accès qui seront attribués aux membres de la classe de base dans la classe dérivée. Les modes qui sont définis en C++ sont : *public*, *protected* et *private*.

RQ. L'accessibilité des membres d'une classe de base depuis une classe dérivée dépend :

• des droits d'accès de ces membres dans la classe de base,

• du mode de dérivation utilisé.

Le tableau suivant dresse tous les cas de figure possibles concernant cette accessibilité :

3

Chapitre 5 : L'héritage en C++

I. L'héritage : Syntaxe de dérivation en C++

Mode de dérivation	Statut du membre dans la classe de base	Statut du membre dans la classe dérivée	Accessibilité du membre dans la classe dérivée	Accessibilité du membre par les utilisateurs de la classe dérivée
public	public	public	Accessible	Accessible
	protected	protected	Accessible	Inaccessible
	private	private	Inaccessible	Inaccessible
protected	public	protected	Accessible	Inaccessible
	protected	protected	Accessible	Inaccessible
	private	private	Inaccessible	Inaccessible
private	public	private	Accessible	Inaccessible
	protected	private	Accessible	Inaccessible
	private	private	Inaccessible	Inaccessible

RQ. Le mode de dérivation par défaut (si rien n'est mentionné) est le mode "*private*".

RQ. Les membres d'une classe de base qui sont accessibles dans la classe dérivée sont manipulés dans cette dernière directement à travers leurs noms à la manière des membresde la classe dérivée.

4

I. L'héritage : Syntaxe de dérivation en C++

Exemple

```
class Document {
        char Reference[10];
        char Titre[100];
        void Afficher();
        char* GetReference();
        char* GetTitre();
};
class Livre : public Document {
        char Auteur[10];
        char Editeur[100];
        char* GetAuteur();
        char* GetEditeur();
};
```


I. L'héritage : Syntaxe de dérivation en C++

```
class A {
        int a1;
protected :
        int a2;
public :
        int a3;
};
class B : private A {
private
        int b1, b2;
protected :
        void f() {
                b1 = a1*2;
        //Erreur car a1 est inaccessible dans B
        }
        void g(){
                b2 = a2*a3;
                //OK car a2 et a3 sont accessibles dans B
        }
};
```

```
void main() {
        A Obj1;
        B Obj2;
        Obj1.a2 = 6;
 //Erreur car a2 est protégé dans A
        Obj2.a3 = 8;
//Erreur car a3 est privé dans B
        Obj1.a3 = 8;
//ok pas de prob

}
```

II. Redéfinition des méthodes

La redéfinition des méthodes est le fait de proposer **une nouvelle définition** dans la classe dérivée **d'une méthode déjà existante** dans la classe de base.

• Généralement la méthode de base et celle redéfinie assurent la même fonctionnalité.

Toutefois la redéfinition est souvent faite dans un souci de prise en compte des spécificités de la classe dérivée.

II. Redéfinition des méthodes

```
class A {
public :
        int att_A;
        A(int p) { att_A = p;}
        void Afficher() { cout<<"La valeur de att_A est : "<<att_A<<endl; }
};
class B : public A {
public :
        int att_B;
        B(int p1, int p2) : A(p1) {  att_B = p2;}
};
void main() {
        B b(5,9);
        b.Afficher();
}
```

La méthode *Afficher* appelée à partir de *l'objet b* va engendrer la sortie suivante :
La valeur de att_A est : 5

II. Redéfinition des méthodes

Nouvelle définition de *B* :

```
class B : public A {
public :
        int att_B;
        B(int p1, int p2) : A(p1) { att_B = p2;}
        void Afficher() { //rédéficition de la fonction Afficher
                cout<<"La valeur de att_A est : "<<att_A<<endl;
                cout<<"La valeur de att_B est : "<<att_B<<endl;
        }
};
```

Avec cette définition de *B* le code suivant :

```
B b(5,9);
b.Afficher();
```

Dans *ce cas, l*a méthode *Afficher* appelée à partir de *l'objet b* va engendrer la sortie suivante :

La valeur de att_A est : 5
La valeur de att_B est : 9

II. Redéfinition des méthodes

Appel des membres d'une classe de base redéfinis dans la classe dérivée

• Les membres d'une classe de base qui sont accessibles dans la classe dérivée sont manipulés dans cette dernière directement à travers leurs noms à la manière des membres de la classe dérivée. Toutefois dans le cas où un membre (attribut ou méthode) est redéfini, ce dernier sera caché dans la classe dérivée par la redéfinition. Pour pouvoir l'appeler quand même dans la classe dérivée, le nom de ce membre doit être associé à celui de la classe de base et ce à l'aide de l'opérateur de résolution de portée selon la syntaxe suivante :

NomClasseBase::NomMembre;

RQ. Si le membre en question est public dans la classe dérivée alors il peut être manipulé à partir des objets de cette dernière selon la syntaxe suivante :

NomObjet.NomClasseBase::NomMembre;

II. Redéfinition des méthodes

Remarque :
Dans l'exemple précédent, il est plus intéressant d'appeler la méthode *Afficher* de la classe *A* dans la définition de *Afficher* de *B* pour assurer l'affichage de *att_A* (afin d'éviter de réécrire un code déjà écrit). Toutefois la définition suivante de la méthode *Afficher* de *B* :

```
void Afficher() {
        Afficher();
        cout<<"La valeur de att_B est : "<<att_B<<endl;
}
```

sera interprétée comme un appel récursif de *Afficher* de *B*. Pour obtenir l'effet souhaité la redéfinition de *Afficher* doit se faire comme suit :

```
void Afficher() {
        A::Afficher ();
        cout<<"La valeur de att_B est : "<<att_B<<endl;
}
```


II. Redéfinition des méthodes

***RQ1.* Différence entre redéfinition et surdéfinition:**
Il ne faut pas confondre surdéfinition et redéfinition. Dans la redéfinition, la méthode de base et celle redéfinie doivent avoir exactement la même signature. Ce n'est pas le cas dans la surdéfinition.

RQ2. Compatibilité entre classe de base et classe dérivée:
D'une manière générale en POO, il y a une certaine compatibilité entre un objet d'une classe de base et un objet d'une classe dérivée. Cette compatibilité va dans le sens suivant (Objet dérivée → Objet de base). Elle est interprétée comme suit : *tout objet d'une classe dérivée peut être considéré comme un objet de la classe de base*.
➔ Tout objet d'une classe dérivée comporte tous les membres d'un objet de la classe de base. Il peut par conséquent le remplacer là ou l'objet de la classe de base est demandé (le remplacer syntaxiquement mais cela ne garantit pas l'obtention d'un résultat toujours satisfaisant).
➔ L'opération inverse n'est pas vraie. En d'autres mots, un objet d'une classe de base ne peut pas remplacer un objet d'une classe dérivée.

II. Redéfinition des méthodes

Exemple : Etant données les deux classes *A* et *B* définies comme suit :

```
class A {
public :
        int att_A;
        void Afficher(){cout<<" att_A :"<<att_A<<endl}
};
class B : public A {
public :
        int att_B;
        void Afficher() {
                cout<<" att_A :"<<att_A<<endl;
                cout<<" att_B :"<<att_B<<endl;
        }
};
…
A a , *pa;
B b , *pb;
…
```

```
a=b ;      // OK
b=a ;      // Erreur

pa = &b; // OK
pb = &a; // Erreur

pb = (A*) &a; // OK, casting
//Conversion explicite
pb->att_B; // Erreur

pa->att_A; // OK
pa->att_B; // Erreur

pa->Afficher();
/*Cette instruction va
engendrer l'appel de la
fonction Afficher définie
dans A.  */
```


III. Construction d'objets dérivés

• L'instanciation d'un objet d'une classe dérivée engendre nécessairement l'instanciation de la partie de ses attributs provenant de la classe de base. Ce processus d'instanciation s'effectue

En C++ en quatre étapes :

o Allocation de l'espace mémoire des attributs,

o Appel du constructeur de la classe dérivée.

o Appel et exécution du constructeur de la classe de base (initialisation des attributs issus de la classe de base).

o Exécution du constructeur de la classe dérivée (initialisation des attributs dérivés).

III. Construction d'objets dérivés

A Cas où la classe dérivée dispose d'un constructeur paramétré :

A.1. Cas ou la classe de base possède un constructeur paramétré :

ClasseDérivée(Arguments):ClasseBase(Arguments) {… … …}

Exemple :

```
class A {
public :
int att_A;
A(int p) : att_A(p){
cout<<"Constructeur de A" ;
Cout <<endl;
}
void Afficher() {
cout<<" att_A :"<<att_A;
cout <<endl;
}
};
```

```
class B : public A {
public :
int att_B;
B(int p1,int p2) : A(p1), att_B(p2){
cout<<"Constructeur de B"<<endl;
}
void Afficher(){
cout<<" att_A :"<<att_A;
cout<<" et att_B :"<<att_B;
cout <<endl;
}
};
...
B b(5,7); // OK
```


III. Construction d'objets dérivés

A Cas où la classe dérivée dispose d'un constructeur paramétré :

A.2. Cas ou la classe de base possède un constructeur non paramétré :

Ce constructeur peut être implicite ou explicite. Dans ce cas, l'appel explicite du constructeur de la classe de base dans la classe dérivée **n'est pas nécessaire**.

Exemple :

```
class A {
public :
int att_A;
A(){
cout<<"Constructeur de A" ;
Cout <<endl;
}
void Afficher() {
cout<<" att_A :"<<att_A;
cout <<endl;
}
};
```

```
class B : public A {
public :
int att_B;
B(int p1,int p2) : Att_B(p2){
cout<<"Constructeur de B"<<endl;
}
void Afficher(){
cout<<" att_A :"<<att_A;
cout<<" et att_B :"<<att_B;
cout <<endl;
}
};

... ...
B b(7); // OK
```

III. Construction d'objets dérivés

B. Cas où la classe dérivée ne dispose pas de constructeur paramétré

B.1. Si la classe de base dispose de constructeurs paramétrés

Dans ce cas **l'instanciation de l'objet dérivé ne sera pas possible**. En effet, l'instanciation de cet objet se fera avec le constructeur par défaut généré par le compilateur. Or ce dernier ne peut pas faire un appel automatique au constructeur paramétré de la classe de base car il ne saura pas quels arguments faut-il lui passer. Cette situation engendre une erreur de compilation.

```
class A {
public :
int att_A;
A(int p) : att_A(p) {
cout<<"Constructeur de A"<<endl;
}
};
```

```
class B : public A {
public :
int att_B;

};
… …
B b; // Erreur
```


III. Construction d'objets dérivés

B. Cas où la classe dérivée ne dispose pas de constructeur paramétré

B.2. Si la classe de base dispose d'un constructeur par défaut

Dans ce cas c'est ce dernier qui sera automatiquement appelé par le constructeur par défaut de la classe dérivée.

```
class A {
public :
        int att_A;


};
```

```
class B : public A {
public :
int att_B;

};
… …
B b; // OK: appel du constructeur par
//défaut de B qui fait appel à son
// tour au constructeur par défaut de A.
```

IV. Destruction d'objets dérivés

• L'appel du destructeur d'un objet d'une classe dérivée engendre l'appel automatique du destructeur de la classe de base.

• L'ordre d'exécution de ces destructeurs est inverse à l'ordre d'exécution des constructeurs : Exécution du destructeur de la classe dérivée puis exécution du destructeur de la classe de base

• D'une manière plus détaillée, la libération se fait comme suit :

o Appel et exécution du destructeur de la classe dérivée.

o Appel et exécution du destructeur de la classe de base.

o Libération de la mémoire utilisé par les attributs (Il ne s'agit pas des espaces mémoires dynamiques pointés par les éventuels attributs de type pointeurs mais des espaces occupés par les attributs eux mêmes).

IV. Destruction d'objets dérivés

Exemple :

```
class A {
public :
… … …
~A(){ cout<<"Destructeur de A"<<endl;}
… … …
};
class B : public A {
public :
… … …
~B(){ cout<<"Destructeur de B"<<endl;}
… … …
};
```

La destruction d'un objet déclaré comme suit :

```
B obj(5,7);
```

va engendrer la sortie suivante

```
… … …
Destructeur de B
Destructeur de A
```

V. Polymorphisme

• Le mot polymorphisme vient du grec et désigne le fait de pouvoir prendre plusieurs formes.

• En programmation orientée objet le polymorphisme est un concept qui se manifeste par la capacité d'une méthode à s'exécuter de différentes manières selon la nature de l'objet qui l'appelle (objet de base ou objet dérivé).

• Le polymorphisme concerne essentiellement des objets liés par une relation d'héritage.