DAUPHINE UNIVERSITÉ PARIS | TUNIS

Cours C++

Master Actuariat

Fatma CHAKER
chakerfatma@yahoo.fr

Séance 10 : L'héritage simple

Année universitaire 2014/2015

Plan

- Héritage simple : définition et syntaxe
- Types d'héritage
- Constructeurs/destructeurs
- Redéfinition de méthodes
- Compatibilité entre classe de base et classe dérivée

Problématique

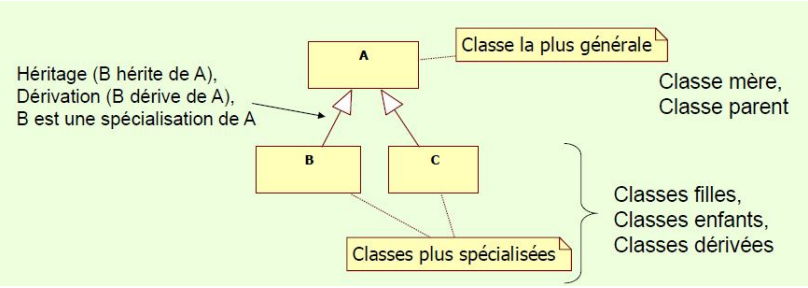
- Souvent, dans une application, des classes ont des membres similaires car elles sont sémantiquement proches:
  - Les classes Secrétaire, Technicien et Cadre peuvent avoir en commun les champs Nom, Age, Adresse et Salaire et les méthodes Evalue() et Augmente().
  - Les classes Carré, Triangle et Cercle peuvent avoir en commun les champs CouleurDuTrait et TailleDuTrait et les méthodes Dessine(), Translation() et Rotation().
- On comprend bien que c'est parce que:
  - Une secrétaire, un technicien ou un cadre est un **employé** (i.e une spécialisation du concept d'employé).
  - Un carré, un triangle ou un cercle est une **forme géométrique**.

Solution : L'héritage

- L'idée de la réutilisation est de cerner ces similitudes, et de les encapsuler dans une classe appelée ***classe de base*** (par exemple Employé ou FigureGéométrique).
- Les classes de base sont parfois appelées ***super classes***.
- Une ***classe dérivée*** est une classe qui ***hérite*** des membres de la classe de base. Concrètement si Technicien hérite de Employé, Technicien hérite des champs Nom, Age, Adresse et Salaire et des méthodes Evalue() et Augmente().
- On dit aussi que la classe dérivée est une **spécialisation** de la clase de base.

Héritage

L'idée est : "un B **est un A avec des choses en plus**".



Plutôt que de ré implémenter des fonctionnalités existantes déjà dans une classe A, une classe B peut intégrer des données et fonctions membres de la classe A. On dit que B hérite de A, que B spécialise A, que A généralise B. B est une sous-classe de A.

Héritage : Syntaxe

Syntaxe

```
class NomClasseFille:
  public NomClasseMere1, public NomClasseMere2 ...
{
  // Déclaration des nouveaux attributs
  // Déclaration des nouvelles méthodes
  // Redéfinition de méthodes
  ...
};
```

C++ permet aussi d'utiliser l'héritage **protected** et **private**, rarement utilisés.

Héritage

L'**héritage permet de former une nouvelle classe à partir de** classes existantes.

La nouvelle classe (**classe fille, sous-classe) hérite des** attributs et des méthodes des classes à partir desquelles elle a été formée (**classes mères, super-classes).**

De nouveaux attributs et de nouvelles méthodes peuvent être définis dans la classe fille.

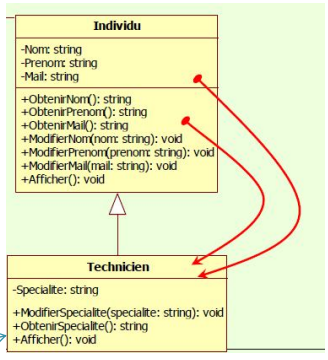
Des méthodes des classes mères peuvent être **redéfinies dans** la classe fille.

La **redéfinition de méthode consiste à (re)définir dans la** classe fille le comportement (code de la méthode) qui existait déjà dans une classe mère.

Si une méthode d'une classe mère n'est pas redéfinie dans la classe fille, alors le code défini dans la classe mère sera utilisé « tel quel » sur les instances de la classe fille.

Héritage simple

- Lorsqu'une classe hérite d'une autre classe, elle possède tous les attributs et opérations de la classe mère



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

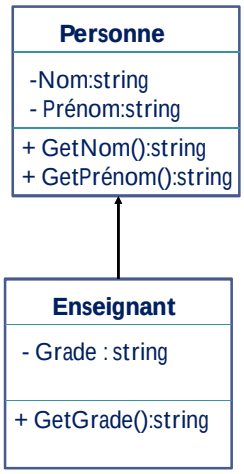
Hiérarchie d'héritage

Plusieurs classes peuvent intervenir et former ainsi ce qu'on appelle une hiérarchie d'héritage.

- Lorsque **chaque classe hérite d'une seule classe**, on parle **d'héritage simple**.
- Lorsqu'une classe **hérite de plusieurs classes**, on parle **d'héritage multiple,**

Certains langages comme JAVA limitent l'héritage multiple.

Héritage simple : Exemple



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

Héritage : Exemple

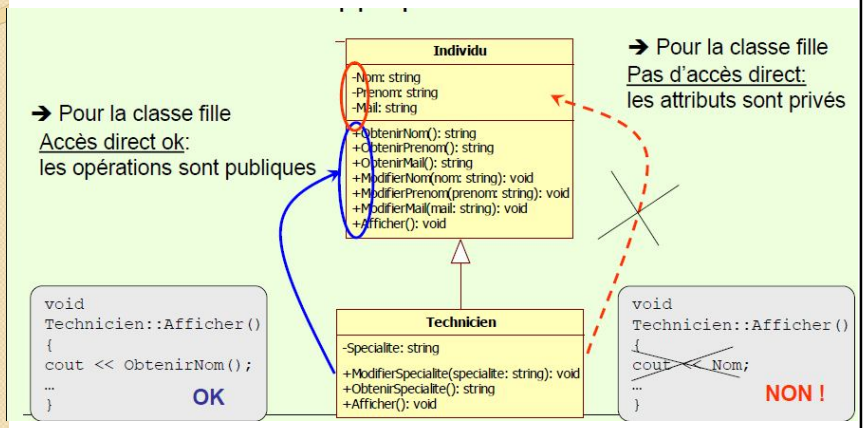
```
// Personne est la classe mère de Enseignant
class Personne {
        private : string nom,prenom;
        public :
        string GetNom() const {return nom;}
        string GetPrenom() const {return prenom;}
        };
// Enseignant est la classe fille ou dérivée de Personne : Enseignant hérite de Personne

class Enseignant : public Personne{
        private : string  Grade;
        public :
        string GetGrade() const {return Grade;}
        };
…
int main()
Personne unePersonne; Enseignant unEnseignant;
unePersonne.GetNom();  // légal : unePersonne est une Personne
unEnseignant.GetGarde();  // légal : unÊnseignant est un Enseignant
unEnseignant.GetNom();  // légal : unEnseignant est une Personne (par héritage)
unePeronne.GetGrade(); //illégal  : unePersonne n'est pas un Enseignant !Erreur
```


Droits d'accès aux membres

Rappel : En C++, on dispose des droits d'accès suivants :

- **Accès public : O**n peut utiliser le membre de n'importe où dans le programme.
- **Accès private : S**eule une fonction membre de la même classe A peut utiliser ce membre ; il est invisible de l'extérieur de A.
- **Accès protected : C**e 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.

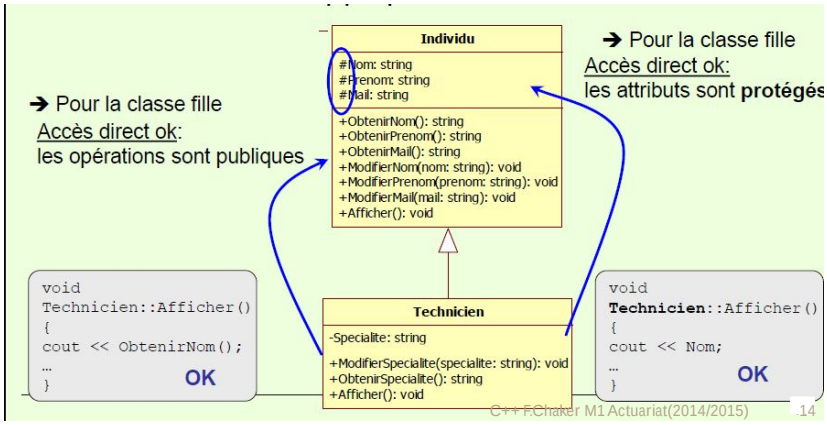
Droits d'accès aux membres



Droits d'accès aux membres

Membre protégé :

- Equivalent à un membre privé pour les utilisateurs de la classe
- Comparable à un membre public pour le concepteur d'une classe dérivée
- Mais comparable à un membre privé pour les utilisateurs de la classe dérivée



Types d'héritage (Mode de dérivation)

La classe B peut hériter de la classe A de trois façons différentes :

class B : *<contrôle d'accès> A*

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

Par défaut

Ne pas confondre le mode de dérivation et le statut des membres d'une classe

Types d'héritage : Héritage public

- La classe dérivée possède tous les membres (attributs et méthodes) de la classe mère, avec les même restrictions d'accès que celles définies dans la classe mère

Héritage public

visibilité dans la classe mère	visibilité dans la classe fille
public	public
protected	protected
private	*« private »* inaccessible

Il est conseillé généralement d'utiliser l'héritage public, car dans le cas contraire on se prive de pouvoir créer de nouvelles classes elles-mêmes dérivées de la classe dérivée.

Types d'héritage : Héritage private

Dérivation privée :

- Transformation du statut des membres publics et protégés de la classe de base en statut privé dans la classe dérivée
  - Pour ne pas accéder aux anciens membres de la classe de base lorsqu'ils ont été redéfinis dans la classe dérivée
  - Pour adapter l'interface d'une classe, la classe dérivée n'apportant rien de plus que la classe de base (pas de nouvelles propriétés) mais offrant une utilisation différente des membres

Types d'héritage : Héritage protected

Dérivation protégée :

Transformation du statut des membres publics et protégés de la classe de base en statut protégé dans la classe dérivée

Statut dans la classe de base	Statut dans la classe dérivée
Public	Protected
Protected	Protected
Private	inaccessible

Héritage et Constructeurs/Destructeurs

- **Pas d'héritage** des constructeurs et destructeurs ➔ il faut les **redéfinir**
- **Appel implicite** des **constructeurs par défaut** des classes de base (super-classe) avant le constructeur de la classe dérivée (sous-classe)
- Possibilité de passage de paramètres aux constructeurs de la classe de base dans le constructeur de la classe dérivée par **appel explicite**
- **Appel automatique** des destructeurs dans **l'ordre inverse** des constructeurs
- **Pas d'héritage des constructeurs de copie et des opérateurs d'affectation**

Héritage et Constructeur par défaut

Ordre des constructeurs

Lors de la création d'un objet d'une classe dérivée, le constructeur de la classe de base est **toujours appelé d'abord**.

- S'il existe dans les deux classes des constructeurs **sans argument**, c'est simple :
  - le constructeur de la classe de base est exécuté en premier,
  - puis le constructeur de la classe dérivée.

```
class Personne {
private : string nom,prenom;
public :
Personne()
  { nom="Ben Foulen "; prenom="Foulen ";}

          ....
};
```

```
class Enseignant : public Personne {
private : string Grade;
public :
Enseignant( ) // appel implicite de Personne( )
  { Grade="Assistant";}

};
```

```
int main()
{
Enseignant unEnseignant ;
```

Héritage et constructeurs : transmission d'arguments

- Si les constructeurs réclament des arguments :

On va créer un objet de type Enseignant avec la déclaration suivante :

Enseignant unEnseignant("Ben Mohamed", "Mohamed" , "MC")

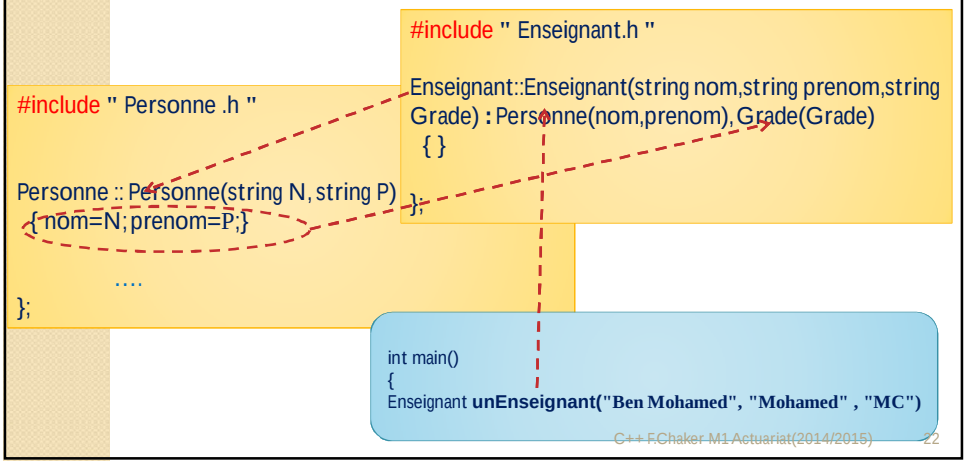
Comment le constructeur de Enseignant va-t-il transmettre les valeurs de nom et prenom ?

- Le C++ permet de répondre à ce problème en permettant l'appel explicite d'un constructeur d'une classe de base par le constructeur d'une classe dérivée.

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

Héritage et constructeurs : transmission d'arguments

Le constructeur de Personne (classe de base) est appelé **avant le** constructeur de Enseignant (classe dérivée).



Héritage et constructeur par recopie

Soit une classe B, dérivant d'une classe A :

- Si aucun constructeur par copie n'est défini dans B : appel du constructeur par copie par défaut de B qui procédera comme suit :
  - Appel du constructeur par copie (par défaut ou celui qui a été défini) de la classe de base A faisant une copie membre à membre
  - Initialisation des membres données de B qui ne sont pas héritées de A.
- Si un constructeur par copie défini dans B : Il faut tenir compte de ce que l'appel de ce constructeur par recopie entraînera l'appel
  - Du constructeur de la classe de base mentionné dans son entête :

B ([const] B& x) : A (x) // transmission de la partie //de B héritée de A

  - d'un constructeur sans argument si aucun constructeur de la classe de base n'est mentionné dans l'entête

B ([const] B&)

Note : Dans ce cas il est indispensable que la classe mère dispose d'un constructeur sans argument, sinon Erreur de compilation !!!

Héritage et constructeur de recopie

Rappel : appel du constructeur par copie lors :

- **de l'initialisation d'un objet par un objet de même type**
- **de la transmission de la valeur d'un objet en argument ou en retour de fonction**

```
#include " Personne .h "

…
Personne :: Personne(Personne & P)
  { nom=P.nom;
    prenom=P.prenom;}
            ….
};
```

```
#include " Enseignant.h "
// constructeur de recopie
// il y aura conversion implicite de E dans le type
//Personne
Enseignant::Enseignant(Enseignant &E) : Personne(E)
  {Grade=E.Grade; }
};
```

Si pas de constructeur par copie défini dans la sous-classe ➔ Appel du constructeur par copie par défaut de la sous-classe et donc du constructeur par copie de la super-classe

Héritage et Destructeur

Ordre des destructeurs

- Le destructeur des classes mères est **toujours appelé implicitement après l'exécution du destructeur de la classe** fille.
- On n'appelle **jamais explicitement le destructeur des classes** mères dans le destructeur de la classe fille.

Héritage : Principe (1/2)

L'héritage permet :

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

Principe (2/2)

```
            // Classe de base Personne
class Personne {
            private : string nom,prenom;
            public :
            string GetNom() const {return nom;}
            string GetPrenom() const {return prenom;}
            void afficher( ) const { cout<<nom<<   <<prenom<<endl;}
            };
            // Classe dérivée Enseignant
class Enseignant : public Personne{
            private : string  Grade;
            public :
            string GetGrade() const {return Grade;} // ajout de nouvelles fonctionnalités
            void afficher( ) const          // modification d'un comportement existant
            { Personne::afficher();       // réutilisation du code déjà écrit
              cout<<Grade<<endl;
            }
};
```

deux fonctions avec le même nom et les mêmes arguments dans la classe !!

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.

Redéfinition : Explication par l'exemple

- Supposons que la classe Personne contient une fonction qui permet à toute personne de se présenter :

Prototype : void afficher ( ) const;

Définition : void Personne :: afficher () const

{ cout<< "Bonjour, je m'appelle " <<nom << " " <<prenom <<endl; }

Dans le main() :

```
Personne unePersonne(" Ben Mohamed " , " Mohamed " );
unePersonne.afficher();
return 0;
```

Exécution :
Bonjour, je m'appelle Ben Mohamed Mohamed

La fonction afficher( )est héritée dans la classe fille Enseignant
Un enseignant est une Personne et peut donc se présenter

```
int main(){
Enseignant unEnseignant("Ayari", "Fatma", "MC");
unEnseignant.afficher();
return 0;
}
```

Exécution :
Bonjour, je m'appelle Ayari Fatma

Redéfinition : Explication par l'exemple

- Imaginons maintenant que les Enseignants ont une manière différente de se présenter. Ils doivent en plus préciser qu'ils sont enseignants et afficher leur grade. Nous allons donc écrire une version différente de la fonction afficher(), spécialement pour eux :

Code C++

```
void Enseignant::afficher() const
{
cout << "Bonjour, je m'appelle " << nom << " " <<prenom<< endl;
cout << "Je suis un enseignant " << endl;
cout << " et mon grade est :" << grade<<endl;
}
```

```
int main(){
Enseignant unEnseignant("Ayari", "Fatma", "MC");
unEnseignant.afficher();
return 0;
}
```

Exécution :
Bonjour, je m'appelle Ayari Fatma
Je suis un enseignant
et mon grade est : MC

Quand on écrit une fonction qui a le même nom que celle héritée de la classe mère, on parle de **masquage. La fonction afficher héritée de** Personne est masquée cachée.

Redéfinition : Explication par l'exemple

Mais on peut faire encore mieux ☺

La fonction **afficher()** de la classe Enseignant a une ligne identique à ce qu'il y a dans la même fonction de la classe Personne. On pourrait donc économiser des lignes de code en appelant la fonction masquée ➔ On parle de **démasquage**.

Code C++

```
void Enseignant::afficher() const
{
Personne::afficher();
cout << "Je suis un enseignant " << endl;
cout << " et mon grade est :" << grade<<endl;
}
```

Exécution :
Bonjour, je m'appelle Ayari Fatma
Je suis un enseignant
et mon grade est : MC

L'opérateur de résolution de portée :: sert à déterminer quelle fonction (ou variable) utiliser quand il y a ambiguïté ou si il y a plusieurs possibilités.

Compatibilité entre classe de base et classe dérivée (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;
Enseignant unEnseignant;
unePersonne=unEnseignant; // OK ! Tout enseignant est une
// personne. Le compilateur extrait, dans l'objet-source
//unEnseignant, les données membres héritées de Personne pour
//les copier dans l'objet-cible unePersonne

/ l'inverse n'est pas vrai :
unEnseignant=unePersonne; // ERREUR  toute personne n'est pas
// forcément un enseignant !! Le compilateur ne peut générer les
// données manquantes pour remplir la totalité des données //
membres de unEnseignant.
```

Compatibilité entre classe de base et classe dérivée (2/2)

- Il est possible aussi de convertir un pointeur sur une instance de la classe dérivée en un pointeur sur une instance de la classe de base, si l'héritage est public

```
Personne  P;
Enseignant E;
Personne *pP=&P;
Enseignant *pE=&E;
```

Alors :

*- l'affectation pP = pE est **correcte***

- l'affectation inverse pE= pP est **illégale** ; on peut cependant la forcer par l'*opérateur de conversion de type (), en écrivant :*

pE = (Enseignant) pP (cf. plus loin)*

En résumé

- L'héritage permet de spécialiser une classe.
- Lorsqu'une classe **hérite** d'une autre classe, elle **récupère toutes ses propriétés et ses méthodes**.
- Faire un héritage a du sens si on peut dire que l'objet A « **est un** » objet B. Par exemple, une Voiture « est un » Vehicule.
- La classe de base est appelée **classe mère** et la classe qui en hérite est appelée **classe fille**.
- Les constructeurs sont appelés dans un ordre bien précis : **classe mère, puis classe fille**.
- En plus de **public et private, il existe une portée protected. Elle est équivalente à private mais elle est un peu** plus ouverte : les classes filles peuvent elles aussi accéder aux éléments.
- Une fonction membre de la classe dérivée peut avoir **la même signature** qu'une fonction de la classe de base. On dit alors que la fonction membre de la classe de base est « **Redéfinie** » dans la classe dérivée.
- Quand on est dans la classe dérivée, on appelle la fonction de la classe dérivée de façon habituelle. Si on veut appeler la fonction membre de la classe de base, il faut **préciser le nom de la classe de base :: en préfixe.**