

# Modélisation objet & programmation objet en C++

TELLEZ Bruno, IUT Lyon1.

Département Informatique, Site de Bourg-en-Bresse

2023-2024

Séance 3

{ Héritage }

# Ca sert à quoi ?

## Etendre une classe Existante?

*Pourquoi ?*

Si on veut créer un **système d'information** du département

*Comment ?*

Identifier les propriétés spécifiques et partagées

*Quel objectif ?*

Distinguer des types de personnes :

*étudiant, professeur, administratif*

# Ajouter des attributs à la classe Personne ?

Problèmes :

Des attributs inutiles : numBadge pour un professeur

Les classes *Prof*, *Admin*, *Etudiant* disparaissent

```
class Personne {  
    private :  
        char* Nom;  
        char* Prenom;  
        int age;  
        //prof+admin  
        int numBureau;  
        int numTel;  
        //Etudiant  
        int numBadge;  
};
```

# Ajouter une classe/structure Personne ?

Problèmes :

Une indirection supplémentaire

*Etudiant* → *Personne* → *Nom*

Deux instances pour une même entité (Etudiant et Personne)

```
class Etudiant {  
    private :  
        Personne identité;  
        int numBadge;  
};
```

# La solution : l'héritage

On aimerait que la classe Etudiant hérite de la classe Personne de ces attributs et de ces méthodes

```
class Personne {  
    private :  
        char* nom;  
        char* prenom;  
};
```

```
class Etudiant : public Personne {  
    private :  
        int numBadge;  
};
```

Notez

On **NE** réécrit **PAS** les attributs de Personne dans Etudiant



# Un peu de vocabulaire

La classe Etudiant est une classe fille de la classe Personne  
une sous-classe  
une classe dérivée

La classe Personne est la classe mère de la classe Etudiant  
la super-classe

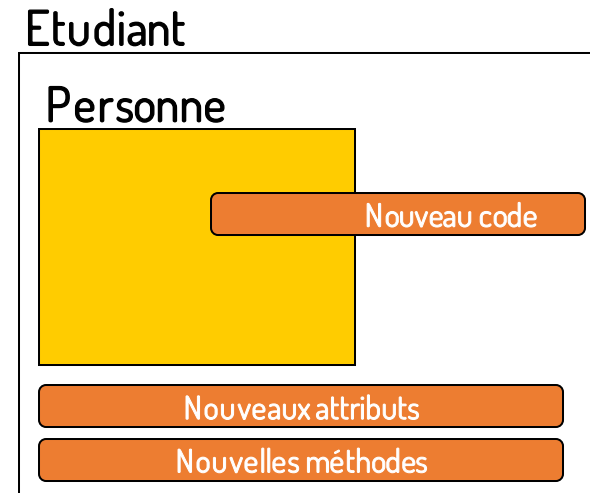
# Dans un classe héritée, on peut...

On peut

- ajouter des attributs
- ajouter des méthodes
- modifier le code d'une méthode héritée

On ne peut pas

- hériter des constructeurs, destructeurs, méthodes amies
- accéder aux attributs ou méthodes privées !





# Initialiser la classe fille

```
class Personne {  
    private :  
        char* nom;  
        char* prenom;  
    public:  
        setNom(char*)  
        setPrenom(char*)  
};
```

```
class Etudiant : public Personne {  
    private :  
        int numBadge;  
    public:  
        Etudiant(char*, char*, int);  
};  
  
Etudiant::Etudiant(char* n, char* p, int nb){  
    setNom(n);  
    setPrenom(p);  
    numBadge = nb;  
}
```

Ca marche mais...

# Mais...une initialisation peu efficace

L'initialisation de la partie Personne est

- Tardive : initialisation par défaut de Personne, puis accesseurs
- Inexploitée : des accesseurs et pas de constructeur de Personne

Une erreur à ne pas commettre :

```
Etudiant::Etudiant(char* n, char* p, int nb){  
    Personne(n,p);  
    numBadge = nb;  
}
```

# Pour une meilleure initialisation d'une classe fille

Une initialisation de la partie Personne

- Synchrone et via son propre constructeur

La liste d'initialisation:

```
Etudiant::Etudiant(char* n, char* p, int nb) : Personne(n,p) {  
    numBadge = nb;  
}
```

```
Etudiant::Etudiant(char* n, char* p, int nb) : Personne(n,p), numBadge(nb){  
}
```

# Concernant l'affichage de la classe Fille

Deux solutions :

- une méthode Affiche()
- Surcharge de l'opérateur <<

```
void Etudiant::Affiche() {  
    cout << getPrenom() << getNom() << noBadge << endl;  
}
```

```
ostream& operator << (ostream&o, const Etudiant& e){  
    o << e.getPrenom() << e.getNom() << e.get_noBadge() << endl;  
    return o;  
}
```

Ca marche mais...

# C'est inefficace...

Parce que dans les deux cas, on réécrit (inutilement)  
l'affichage des champs de la classe mère

Risques :

- D'erreur à la recopie (et c'est long!)

- D'oublier de modifier cet affichage si la classe mère change

# Quel affichage ... avec héritage

```
void Etudiant::Affiche() {  
    Personne::Affiche();  
    cout << noBadge << endl;  
}
```

Utilisation de l'opérateur de portée



```
ostream& operator << (ostream&o, const Etudiant& e){  
    o << (Personne) e;  
    o << e.get_noBadge() << endl;  
    return o;  
}
```

Conversion entre classe (on va y revenir)



# A propos des conversions

Il est possible de convertir de la classe dérivée vers la classe de base

Etudiant e; Personne p = e;

Etudiant\* pe; Personne\* pp = pe;

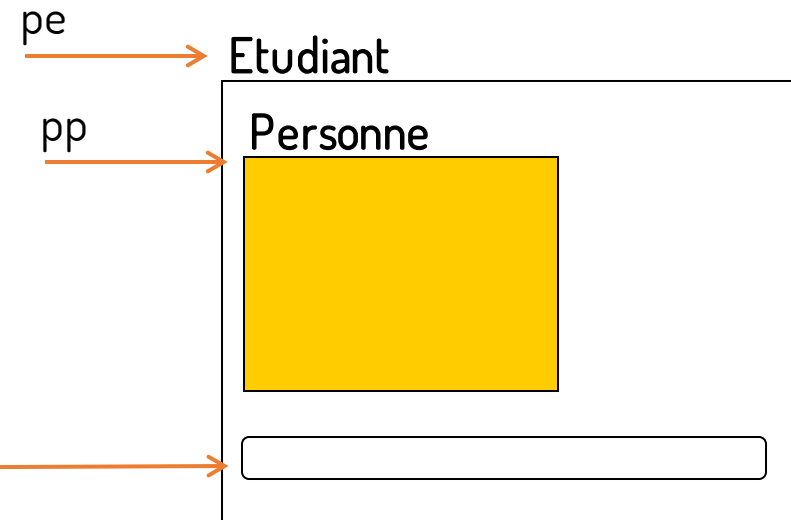
Etudiant e; Personne& p = e;

A priori, il est impossible de retrouver les attributs de la classe dérivée  
sauf...

# Conversion entre pointeurs

Etudiant\* pe; Personne \* pp = pe;  
On ne fait que déplacer un pointeur

Ces attributs ne sont pas perdus



A quoi ca pourrait servir ?



# Imaginons...

Comment stocker toutes les informations de votre SI pour l'IUT (cf début du cours)

Vous devez créer un tableau pour chaque sous-classe : LOURD !

Vous créez un tableau de personnes et vous espérez retrouver les sous-classes d'origine

Un début de solution : le tableau de pointeurs sur Personne

`Personne* DepartementInfo_SI[200];`

Ok, maintenant affichons les éléments de ce tableau :

```
for (int i=0;i<200;i++){  
    if (DepartementInfo_SI[i] != NULL)  
        DepartementInfo_SI[i]->Affiche();  
}
```

J'affiche quoi ?

# Un tableau de personnes mais...

...pas de professeur, ni d'étudiant, ni d'administratif

La solution !

Remettre le pointeur à la « bonne » place

Comment ?

Employer des fonctions virtuelles (déclarées dans la classe mère)

```
class Personne {  
    public:  
        virtual void Affiche ();  
};
```

# Les fonctions virtuelles...ça marche comment



On peut avoir :

- Plusieurs fonctions virtuelles dans une classe
- La classe fille peut surcharger le comportement de cette fonction  
C'est cette version qui sera appelée si la classe origine du pointeur était cette classe fille
- Le destructeur de la classe mère sera souvent lui-même virtuel

```
class Personne {  
    public:  
        virtual ~Personne();  
};
```

```
for (int i=0;i<200;i++){  
    if (DepartementInfo_SI[i] != NULL)  
        delete DepartementInfo_SI[i];  
        //appel aux constructeurs des classes filles  
}  
delete [] DepartementInfo_SI;
```

# Les classes abstraites

Prenons l'exemple des figures géométriques

Ecrivez un modèle de hiérarchie pour les figures géométriques habituelles

Pourquoi la classe FigureGeometrique a un statut particulier ?

Réponse :

# Les classes abstraites : définition

Comment définir une classe abstraite ?

Elle doit avoir au moins une méthode virtuelle pure

```
virtual float Aire() = 0;
```

Le =0 sur une fonction virtuelle indique

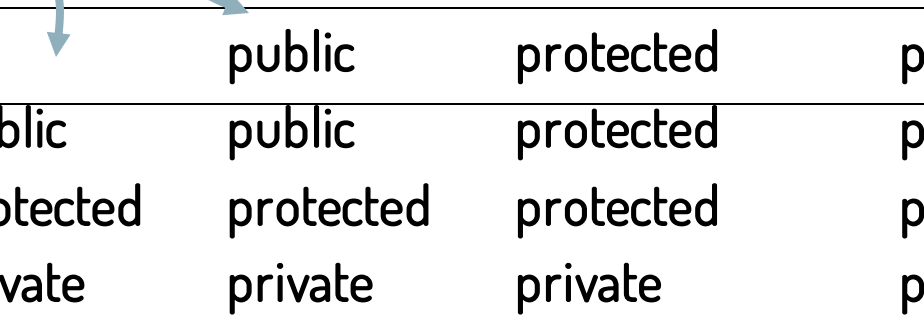
- Qu'elle devient virtuelle pure
- Qu'il n'y aura pas de code
- Qu'on oblige les classes filles à implémenter cette fonction

# Droits d'accès

- Une classe dérivée
  - ne peut pas accéder aux membres privés de sa classe mère
  - est à l'extérieur donc passage par les méthodes publiques
    - Une solution : déclarer des attributs protected
- Protected
  - équivalent à public pour les classes filles
  - équivalent à private pour les autres classes

Membres dans  
classe de Base

Types d'héritage



	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Membres dans  
classes filles

# Données et méthodes statiques

Mot clé : static

Donnée statique appartient à toute la classe, partagée par toutes les instances

Donnée statique de méthode locale à une fonction

Méthode statique appartient à la classe (pas à l'instance), accès aux variables statiques

# Donnée statique

- Donnée statique de classe
  - `class A {`
  - `static int i;` ← Déclaration dans la classe.
  - `};`
  - `int A::i=3;` ← Initialisation en dehors de la classe
- Appartient à la classe
- Toutes les instances partagent la même et unique variable
- Ne doit être définie qu'une seule fois  
(en dehors des classes et des fonctions et donc du main)



# Donnée statique de fonction

- Donnée statique locale à une fonction

- class A {
- public:
- int f(void) ;
- };
- int A::f(void)
- {
- static int val=0 ;
- return val++ ;
- }

- Portée réduite au bloc

```
void main(void)
{
    A a1, a2 ;
    printf("%d ", a1.f()) ; → Affiche 0
    printf("%d\n", a2.f()) ; → Affiche 1
}
```

# Méthode statique

- Méthode statique de classe

- `class A {`
- `public :`
- `int val;`
- `static int valstat;`
- `public :`
- `static int f(void);`
- `};`
- `int A::f(void)`
- `{`
- `val++ // interdit`
- `return valstat++;`
- `}`

```
void main(void)  
{  
    int a;  
    a=A::f();  
}
```

- Utile pour des méthodes relatives à la classe