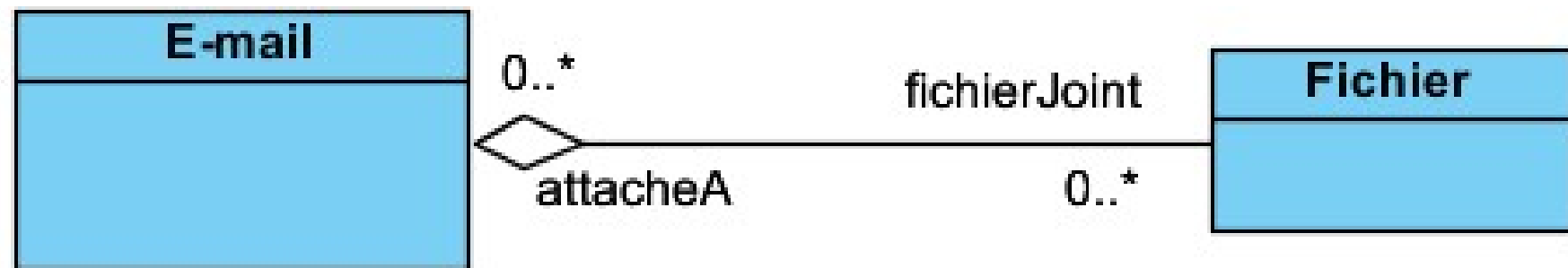

Module R3.04

Chapitre 7

**Composition, Agrégation
Héritage, Polymorphisme**

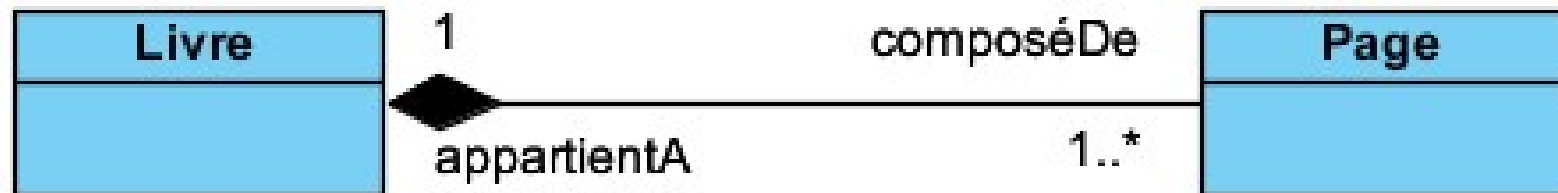
7. Agrégation - Rappel

- **L'agrégation** est une association non symétrique, qui exprime un couplage fort et une relation de subordination entre éléments, dans laquelle un élément peut continuer à exister même si l'autre disparaît.
- **Exemple** : Ici on exprime qu'un E-mail peut comporter plusieurs fichier joints (et qu'un fichier peut être joint à plusieurs E-mails...). Un E-mail **agrège** donc un ou plusieurs fichier. Si l'E-mail est supprimé le fichier existe toujours, et réciproquement



7. Composition - Rappel

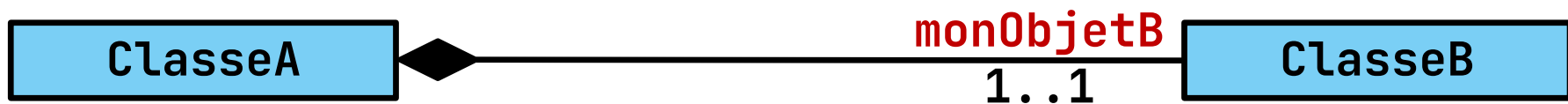
- La **composition** est une agrégation forte (on parle aussi d'**agrégation par valeur** ou d'**agrégation interne**).
C'est une relation entre éléments dans laquelle l'agrégat et ses éléments sont fortement liés : si l'agrégat disparaît, ses éléments disparaissent aussi.
- **Exemple** : on exprime ici qu'un livre est composé de 1 ou plusieurs pages et qu'une page appartient « physiquement » à un livre et un seul... Si le livre est détruit, ses pages aussi !



7. Composition et Agrégation en C++ (1)

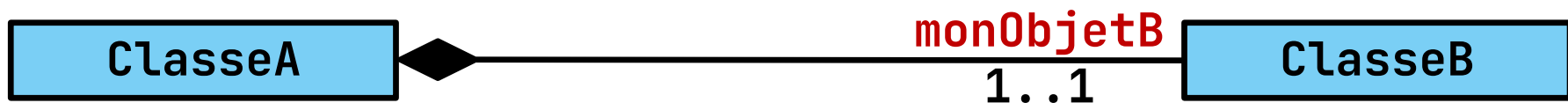
- **En java**, les implémentations d'une composition et d'une agrégation sont très proches. Dans un cas comme dans l'autre, l'agrégat contiendra un attribut **référence** à l'élément qu'il contient... puisqu'en java tous les objets sont manipulés par référence !
 - ❑ **Composition** : l'élément contenu sera instancié par l'agrégat (typiquement dans le constructeur de celui-ci). Seul l'agrégat connaîtra la référence de l'objet qu'il contient.
 - ❑ **Agrégation** : l'élément contenu sera instancié « **à l'extérieur** » de l'agrégat et sa référence sera transmise à l'agrégat (typiquement dans le constructeur de celui-ci)
- **En C++**, les implémentations d'une composition et d'une agrégations seront plus différenciées.
 - ❑ **Composition** : l'élément contenu sera un attribut qui fera partie intégrante de l'agrégat (agrégation **par valeur** ou **interne**).
Leurs cycle de vie sont liés
 - ❑ **Agrégation** : l'agrégat contiendra soit un attribut **référence** soit un **pointeur** vers l'élément associé
 - ❑ L'implémentation dépend donc de la **nature de la relation** (composition ou agrégation) et de sa **cardinalité**

7.1. Composition – Cardinalité 1 (1)



- Un objet **ClasseA** possède sa propre instance d'un objet **ClasseB**
- Le rôle **monObjetB** se traduit dans **ClasseA** par un attribut de type **ClasseB**
- **monObjetB** devra être construit dans le constructeur de **ClasseA**
- **monObjetB** sera automatiquement libéré, et donc son constructeur sera automatiquement appelé, lorsque l'objet **ClasseA** est libéré

7.1. Composition – Cardinalité 1 (2)



```
#include "ClasseB.h"
```

```
class ClasseA {
public:
    ClasseA(...)
    : monObjetB(...) {...}

```

Le constructeur de **ClasseB** est explicitement appelé sur l'attribut (instance) **monObjetB**, dans la **liste d'initialisation**. Le constructeur de **monObjetB** sera donc exécuté **avant** que le corps du constructeur de **ClasseA** ne le soit.

```
~ClasseA() {...}
```

Le destructeur de **ClasseB** sera automatique exécuté sur l'attribut (instance) **monObjetB**, **après** que le destructeur de **ClasseA** ait été exécuté.

```
private:
    ClasseB monObjetB;
};
```


7.1. Composition – Cardinalité 0..1 (1)



- Un objet **ClasseA** possède **éventuellement** un objet **ClasseB**
- Le rôle **monObjetB** se traduit dans **classeA** par un attribut de type **ClasseB *** (**pointeur**)
- Si un objet **ClasseA** ne possède pas d'objet **ClasseB**, son attribut **monObjetB** vaudra **nullptr**
- Si un objet **classeA** possède un objet **ClasseB**, ce dernier devra être alloué **dynamiquement** dans le constructeur de **ClasseA** (ou plus tard dans une autre méthode) et son adresse sera stockée dans l'attribut **monObjetB**
- L'objet **ClasseB** pointé par **monObjetB**, s'il existe, doit être explicitement libéré dans le destructeur de **ClasseA**
- **ClasseA** doit être sous forme canonique de Coplien

7.1. Composition – Cardinalité 0..1 (2)

```
#include "ClasseB.h"
class ClasseA {
public:
    ClasseA(...)
    : monObjetB(nullptr) {
        // Si objetB doit être créé, on l'alloue dynamiquement (dans le constr. ou ailleurs)
        if (...) this->monObjetB = new ClasseB(...);
    }
    ClasseA(const ClasseA & unobjetA) {
        // on clone l'objetB de unobjetA s'il existe
        if (unobjetA.monObjetB == nullptr) this->monObjetB = nullptr;
        else this->monObjetB = new ClasseB(*(unobjetA.monObjetB));
    }
    ClasseA & operator = (const ClasseA & unobjetA) {
        // on supprime l'objetB actuel, s'il existe
        if (this->monObjetB != nullptr) delete this->monObjetB;
        // on clone l'objetB de unobjetA s'il existe
        if (unobjetA.monObjetB == nullptr) this->monObjetB = nullptr;
        else this->monObjetB = new ClasseB(*unobjetA.monObjetB);
        return *this;
    }
    ~ClasseA() {
        // si objetB présent, on le libère
        if (this->monObjetB != nullptr) delete this->monObjetB;
    }
private:
    ClasseB * monObjetB;
};
```



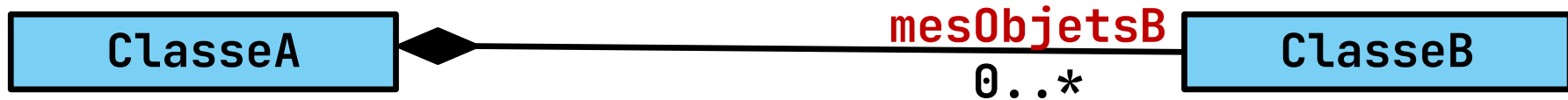
```
classDiagram
    ClasseA "1" *-- "0..1" ClasseB : monObjetB
```


7.1. Composition – Cardinalité 0..* (1)



- Un objet **ClasseA** possède **zéro, un ou plusieurs** objets **ClasseB**
- Le rôle **mesObjetsB** se traduit dans **ClasseA** par un attribut de type **vector<ClasseB>** (**vecteur d'objets**)
- Lorsqu'un objet **ClasseA** est libéré, son attribut **mesObjetsB** (objet **vector**) sera automatiquement libéré et tous les objets **ClasseB** qu'il contient aussi
- ***Autre Solution** : déclarer l'attribut **mesObjetsB** comme un vecteur de pointeurs sur des objets de **ClasseB** : **vector<ClasseB*>**
Dans ce cas il faudrait :*
 - ❑ *Gérer l'allocation dynamique des objets **ClasseB***
 - ❑ ***ClasseA** devra impérativement être sous forme canonique de Coplien*

7.1. Composition – Cardinalité 0..* (2)



```
#include <vector>
#include "ClasseB.h"

class ClasseA {
public:

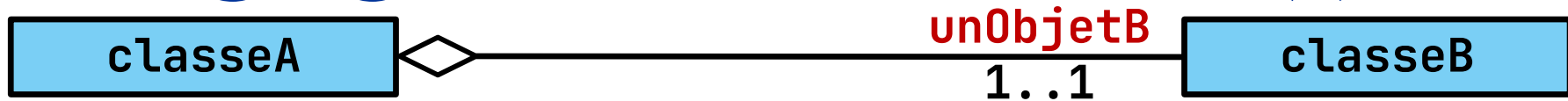
    ClasseA(...)
    : mesObjetsB() {} // constr. de mesObjetsB (vecteur vide) facultative
                      // constr. par défaut appelé automatiquement

    void addObjetB(...) {
        mesObjetsB.push_back(ClasseB(...)); // on crée un objet anonyme ClasseB
                                             // qui est ajouté au vecteur
    }

    ~ClasseA() {} // rien de particulier à faire

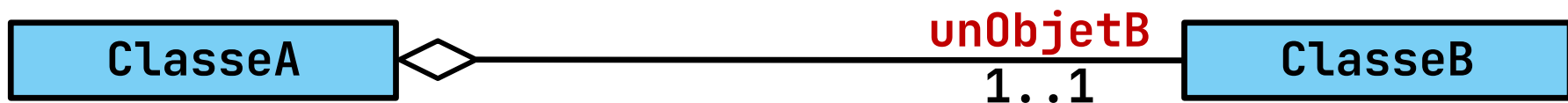
private:
    std::vector<ClasseB> mesObjetsB;
};
```

7.1. Agrégation – Cardinalité 1 (1)



- Un objet **ClasseA** est associé à un objet **ClasseB** qui ne lui appartient pas
- Le rôle **unObjetB** se traduit par la présence dans **ClasseA** d'un attribut de type **ClasseB & (référence)**
- La référence sera constante ou pas selon que l'objet **ClasseA** a le droit ou pas de modifier l'objet **ClasseB** auquel il est associé
- **La référence unObjetB** doit impérativement être initialisée dans le(s) constructeur(s) de **ClasseA** qui doivent donc tous recevoir en paramètre un objet **ClasseB**
- Les objets **ClasseA** ne s'occupent évidemment pas de la libération des objets **ClasseB** !
- **Autre Solution** : déclarer l'attribut **unObjetB** comme un pointeur sur un objet de **ClasseB** : **ClasseB***. Cela permettra de changer l'**ObjetB** associé, ce qui n'est pas possible avec une référence !

7.1. Agrégation – Cardinalité 1 (2)



```
#include "ClasseB.h"

class ClasseA {
public:

    ClasseA(const ClasseB & objetB)
    : unObjetB(objetB) { // La référence unObjetB doit être initialisée
                        // dans la liste d'initialisation
    }

    ~ClasseA() {} // rien à faire

private:
    const ClasseB & unObjetB; // Agrégation vers un objet ClasseB
                             // qu'on ne peut modifier (const)
};                          // Mettre const ou pas selon le besoin
```

7.1. Agrégation – Cardinalité 0..1 (1)



- Un objet **ClasseA** est **éventuellement** associé à un objet **ClasseB**
- Le rôle **monObjetB** se traduit par un attribut de type **ClasseB *** (**pointeur**) dans **ClasseA**
- On ne peut pas utiliser une référence comme précédemment car une référence doit impérativement référencer une valeur
- Si un objet **ClasseA** n'est pas associé à un objet **ClasseB**, son attribut **unObjetB** vaudra **nullptr**
- Dans son destructeur, un objet **ClasseA** ne devra pas tenter de libérer l'objet **ClasseB** qui ne lui appartient pas

7.1. Agrégation – Cardinalité 0..1 (2)



```
class ClasseA {
public:

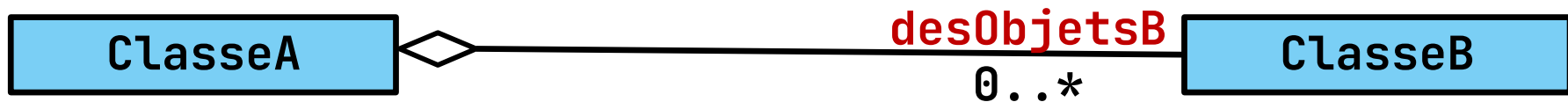
    ClasseA()
    : unObjetB(nullptr) { // Le pointeur unObjetB doit être initialisé
    }

    void associeObjetB(const ClasseB & objetB) {
        this->unObjetB = & objetB; // on stocke l'adresse de l'objet associé
    }

    ~ClasseA() {} // rien de particulier à faire
                  // et surtout pas delete unObjetB !!!

private:
    const ClasseB * unObjetB; // Agrégation optionnelle vers un objet
                              // ClasseB qu'on ne peut modifier (const)
};                          // Mettre const ou pas selon le besoin
```

7.1. Agrégation – Cardinalité 0..* (1)



- Chaque objet **ClasseA** est associé à **zéro, un ou plusieurs** objets **ClasseB**
- Le rôle **desObjetsB** se traduit dans **ClasseA** par un attribut de type **vector<ClasseB *>** (**pointeurs**)
- On peut pas utiliser de références ici car le *template* **vector** ne peut être instancié avec un type référence (vector<T&> ne compile pas)
- Un objet **ClasseA** ne doit pas tenter de libérer les objets **ClasseB** auquel il est associé car ils ne lui appartiennent pas

7.1. Agrégation – Cardinalité 0..* (2)

```
#include <vector>
#include "ClasseB.h"
```

```
class ClasseA {
public:
```

```
    ClasseA()
```

```
    : desObjetsB() { // L'appel explicite du constructeur par défaut
                      // de vector est facultative
    }
```

```
    void associeUnObjetB(const ClasseB & objetB) {
        desObjetsB.push_back(& objetB); // on ajoute l'adresse de l'objetB
                                          // au vecteur
    }
```

```
    ~ClasseA() {} // rien de particulier à faire,
                  // et surtout pas for(auto p : desObjetsB) delete p; !!!
```

```
private:
```

```
    std::vector<const ClasseB *> desObjetsB; // Ici agrégation vers des objets
                                              // qu'on ne peut modifier (const)
                                              // Mettre const selon le besoin
};
```



7.4. Donner accès à un composant ou à un objet associé

- Que ce soit dans le cadre d'une agrégation ou d'une composition, si l'on souhaite donné accès à un objet associé, il faut le faire par référence constante
- Voici les « getters » que l'on écrirait dans 3 des cas vus précédemment :

private: ClasseB monObjetB;	<code>const ClasseB & ClasseA::getObjetB () const { return monObjetB; }</code>
private: ClasseB & unObjetB;	<code>const ClasseB & ClasseA::getObjetB () const { return unObjetB; }</code>
private: ClasseB * monObjetB;	<code>const ClasseB & ClasseA::getObjetB () const { return *monObjetB; }</code>

- Si **exceptionnellement** vous souhaitez donner un accès à l'attribut d'un objet pour que cet attribut puisse être modifié par celui y accède :

private: ClasseB monObjetB;	<code>const ClasseB & ClasseA::getRefObjetB () const { return monObjetB; }</code>
--------------------------------	--------------------------------------------------------------------------------------------------------------

Module R3.04

Chapitre 8

Héritage - Polymorphisme

8. Héritage et Polymorphisme en C++

- Le C++ propose une implémentation très complète de l'héritage car il propose aussi bien l'héritage simple que multiple, ainsi que des options de modification de la visibilité des membres hérités.
- C++ propose également le chaînage automatique des constructeurs et destructeurs.
- Le seul manque : l'absence d'interfaces au sens Java ou Objective C du terme, notion qu'il est toutefois possible de simuler en utilisant l'héritage multiple avec des **classes abstraites** (composées uniquement des méthodes virtuelles pures, évoquées plus tard)

8. Héritage : rappels

Les hiérarchies de classes permettent de gérer la complexité, en ordonnant les objets au sein d'arborescences de classes

■ Spécialisation

- ❑ Démarche **descendante**, qui consiste à capturer les particularités d'un ensemble d'objets, non discriminés par les classes déjà identifiées.
- ❑ Consiste à étendre les propriétés d'une classe, sous forme de sous-classes, plus spécifiques (permet l'extension du modèle par réutilisation).

■ Généralisation

- ❑ Démarche **ascendante**, qui consiste à capturer les particularités communes d'un ensemble d'objets, issus de classes différentes.
- ❑ Consiste à factoriser les propriétés d'un ensemble de classes, sous forme d'une super-classe, plus abstraite (permet de gagner en généricité).

■ Classification

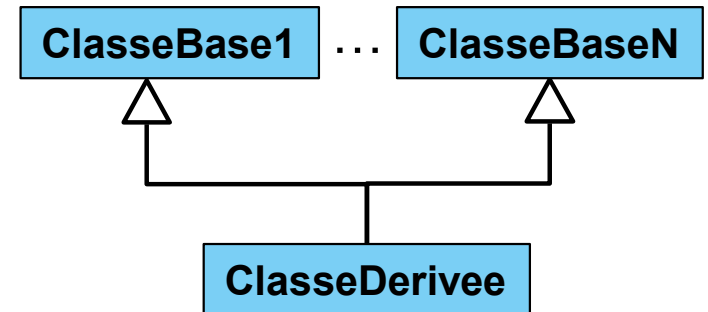
- ❑ L'héritage (spécialisation et généralisation) permet la classification des objets
- ❑ Une bonne classification est stable et extensible
- ❑ Les critères de classification sont subjectifs.
- ❑ **Le principe de substitution de Liskov** permet de déterminer si une relation d'héritage est bien employée pour la classification : Si Y hérite de X, cela signifie que "Y est une sorte de X" et l'on doit pouvoir utiliser un Y à la place d'un X sans modifier la sémantique du programme

Source: <http://uml.free.fr/>

8.1. Syntaxe générale

Modificateur d'héritage

```
■ class ClasseDerivee
  : [public|protected|private] ClasseBase1,
    [public|protected|private] ClasseBase2,
    ...
    [public|protected|private] ClasseBaseN
{
  // déclaration de classe
};
```



Déclaration	Commentaire
<code>class Cercle : public ObjetGraphique</code>	Héritage simple : Cercle hérite d'ObjetGraphique en mode public
<code>class TexteGraphique : public ObjetGraphique, public Chaine</code>	Héritage multiple, (double en fait) : TexteGraphique hérite à la fois d'ObjetGraphique et de Chaine (à chaque fois en public)
<code>class Pile : private Vecteur</code>	Pile hérite uniquement de la classe Vecteur mais en mode privé (pas bien)

8.2. Modificateur d'accès **private**, **protected**, **public**

- Un **membre protected** n'est pas visible à l'extérieur de sa classe mais est visible dans les classes dérivées.
- **protected** respecte le principe d'encapsulation tout en autorisant la transmission des membres entre classe de base et classes dérivées.
- Le tableau suivant récapitule les accès fournis par ces trois modificateurs :

Membre dans la classe de Base	Visibilité dans les Classes Dérivées	Visibilité à l'extérieur de la Classe de Base
private	Non	Non
protected	Oui	Non
public	Oui	Oui

8.2 Modificateur d'héritage **private**, **protected**, **public**

- Le **modificateur d'héritage** peut être **public**, **protected** ou **private**
- Selon sa valeur, il modifie, dans la classe dérivée, la visibilité des membres hérités de la classe de base
- Le tableau suivant précise les changements en fonction du modificateur d'héritage :

Membre dans la classe de base	Visibilité du membre hérité dans une classe dérivée		
	Héritage public	Héritage protected	Héritage private
private	<i>présent mais non visible</i>	<i>présent mais non visible</i>	<i>présent mais non visible</i>
protected	protected	protected	private
public	public	protected	private

8.2.1. Utilité de l'héritage en private (1)

- L'héritage en public et l'utilisation du modificateur d'accès **protected** traduit la notion classique d'héritage.
- Quelle est donc l'utilité du modificateur d'héritage **private** qui « cache » à l'utilisateur de la classe dérivée tous les membres de la classe mère ?
- Considérons la relation “Est implémenté sous forme de” au travers d'un exemple classique :
 - ❑ Soit la classe Pile. On peut l'implémenter à l'aide d'un tableau, d'une liste chaînée ou de toute autre classe de stockage (conteneur). Mais les utilisateurs de la classe pile ne doivent pas avoir accès aux membres de la classe de stockage.
 - ❑ On pourrait donc proposer en C++ la solution suivante :

```
class Pile : private ClasseDeStockage {  
    // membres de la classe Pile  
};
```


8.2.1. Utilité de l'héritage en private (2)

■ Avantages :

- ❑ L'interface de cette classe est uniquement constituée des membres publics de **Pile**, ce qui cache bien ceux de la **ClasseDeStockage** qui, du fait de l'héritage en **private**, sont **private** dans **Pile**
- ❑ Les méthodes de la classe **Pile** peuvent utiliser directement les attributs **protected** et **public** de la **ClasseDeStockage**, permettant une certaine efficacité dans le code.

■ Inconvénients

- ❑ Les méthodes de la classe **Pile** peuvent utiliser directement les attributs **protected** de la **ClasseDeStockage** ce qui peut impliquer des manipulations dangereuses vis à vis de l'intégrité des données dans la **ClasseDeStockage**.
- ❑ **Il est toujours dangereux d'utiliser de l'héritage pour une relation qui ne soit pas de la Généralisation / spécialisation : cela viole le principe de substitution**
Une **Pile** n'est pas une spécialisation d'un vecteur ou d'une liste chaînée. La notion de Pile traduit une forme d'accès aux éléments alors que la **ClasseDeStockage** propose des notions différentes.

8.2.1. Utilité de l'héritage en private (3)

- **Conclusion : La rigueur conceptuelle doit toujours primer sur l'efficacité du code.** L'héritage, qu'il soit **public** ou **private** doit toujours rendre compte d'une relation de généralisation / spécialisation. Il ne faut donc pas utiliser l'héritage en **private** pour traduire la relation "*Est implémenté sous forme de*". Il faut utiliser pour cela la **composition** :

```
class Pile { private: ClasseDeStockage elements; }
```

- Il n'est alors plus possible d'utiliser les attributs non **public** de **ClasseDeStockage** dans les méthodes de **Pile**, mais si la classe **ClasseDeStockage** est bien conçue, cela ne posera pas de problème.
- Et il n'est pas incohérent d'un point de vue conception de dire :
"*Une pile contient un objet de ClasseDeStockage pour stocker ces éléments*"
- La Librairie standard du C++ prône elle aussi l'utilisation de la **composition** pour la relation "*Est implémenté sous forme de*".

8.2.2. Faut-il déclarer **private** ou **protected** les attributs dans une classe ?

- **Rappel** : il est fortement déconseillé de déclarer **public** un attribut, pour respecter le principe d'encapsulation des objets.
Restent 2 possibilités : les déclarer **protected** ou **private**.
- Certains préconisent, au nom de la simplicité, de systématiquement déclarer les attributs **protected**...
- ... Mais certains attributs doivent absolument rester **private** afin de garantir le respect des contraintes d'intégrité
(cf TPs : $\text{min} \leq \text{valeur} \leq \text{max}$ dans `NombreContraint`)
- Conduite à adopter :
 - Ne déclarer **protected** que les attributs qui pourront être modifiés librement dans les classes dérivées, sans danger pour l'intégrité totale de l'objet
 - Laisser les autres attributs en **private**, en proposant des getter/setter (**inline**) garantissant l'intégrité de l'objet, si nécessaire

8.3. Construction/destruction des objets « dérivés » (1)

- Pour les **constructeurs** :

*Le constructeur d'une classe dérivée appelle toujours les constructeurs de ses classes de base **avant** de construire ses propres attributs.*

- Pour les **destructeurs** :

*Les destructeurs des classes de base sont automatiquement appelés **après** l'exécution du destructeur de la classe dérivée.*

- Deux possibilités :

- ❑ Vous spécifiez vous même l'appel au constructeur de la classe de base
- ❑ Vous ne le spécifiez pas, et il y a appel automatique du **constructeur par défaut** de la classe de base. Si celui-ci n'existe pas, le compilateur envoie un message d'erreur

8.3. Construction/destruction des objets « dérivés » (2)

- Exemple : classe LigneHorizontale dérivée de ObjetGraphique
- L'appel au constructeur de la classe de base se fait **en première position** dans **la liste d'initialisation**, avant l'initialisation des attributs.

```
class ObjetGraphique {
public:
    ObjetGraphique(int x, int y, int couleur = 0) :
        m_pointBase(x, y),
        m_couleur_couleur) {}
protected:
    Point m_pointBase;
    int m_couleur;
};

class LigneHorizontale : public ObjetGraphique {
public:
    LigneHorizontale(int x, int y, int longueur, int couleur = 0)
    : ObjetGraphique(x, y, couleur),
      m_longueur(longueur) {
}
private:
    int m_longueur;
};
```

Appel du constructeur
de la classe de base
dans la liste d'initialisation

8.3. Construction/Destruction des objets « dérivés » (2)

- Dans le cas de l'héritage multiple, on doit appeler, dans la **liste d'initialisation**, les constructeurs en respectant l'ordre de dérivation.
- Par exemple, supposons que l'on souhaite créer une classe **TexteGraphique** héritant à la fois des classes **ObjetGraphique** et **Chaine** :

```
class TexteGraphique : public ObjetGraphique, public Chaine {
public:
    TexteGraphique(int x, int y, int couleur, const std::string & texte,
                   const std::string & nomPolice = "Verdana",
                   unsigned short taillePolice = 12)
    : ObjetGraphique(x, y, couleur),
      Chaine(texte),
      m_police(System::GetFont(nomPolice))
    {
        this->m_police->setSize(taillePolice);
    }
    ~TexteGraphique() {
        delete m_police;
    }
private:
    Font * m_police;
};
```

Liste d'initialisation : appel des constructeurs des classes de base (**ObjetGraphique & Chaine**) puis construction des **attributs** spécifiques de la classe dérivée (**m_police**)

8.4. Polymorphisme : Méthodes Virtuelles (1)

- Considérons les classes :

```
Personne      // Définit une méthode afficher()  
Etudiant      // Dérive de Personne, redéfinit afficher()  
Salarie       // Dérive de Personne, redéfinit afficher()
```

- Le langage C++ permet d'écrire les déclarations suivantes :

```
Personne * p1 =  
    new Personne("dupond", "10 rue de l'isère");  
Personne * p2 =  
    new Etudiant("durand", "15 rue de l'isere");  
Personne * p3 =  
    new Salarie("dumoulin", "21 rue jean jaures",  
                "schneider", 4000.00);
```

- Dans **p1**, **p2**, **p3**, déclarés pointeurs sur **Personne**, on peut mettre l'adresse d'un objet de type **Personne**, **Salarie** ou **Etudiant**
- Ceci est possible car les classes **Salarie** et **Etudiant** dérivent (héritent) de la classe **Personne**

8.4. Polymorphisme : Méthodes Virtuelles (2)

- Considérons le code suivant :

```
int main() {  
    Personne * p1 =  
        new Personne("dupond", "10 rue de l'isere");  
    Personne * p2 =  
        new Salarie("dumoulin", "21 rue jean jaures",  
                    "schneider", 4000.00);  
    Personne * p3 =  
        new Etudiant("durand", "15 rue de l'isere");  
  
    p1->afficher(); // appel de méthode (1)  
    p2->afficher(); // appel de méthode (2)  
    p3->afficher(); // appel de méthode (3)  
  
    return 0;  
}
```

Dans un langage offrant le polymorphisme, ces trois instructions **identiques** devraient exécuter des codes **différents** (polymorphisme !)

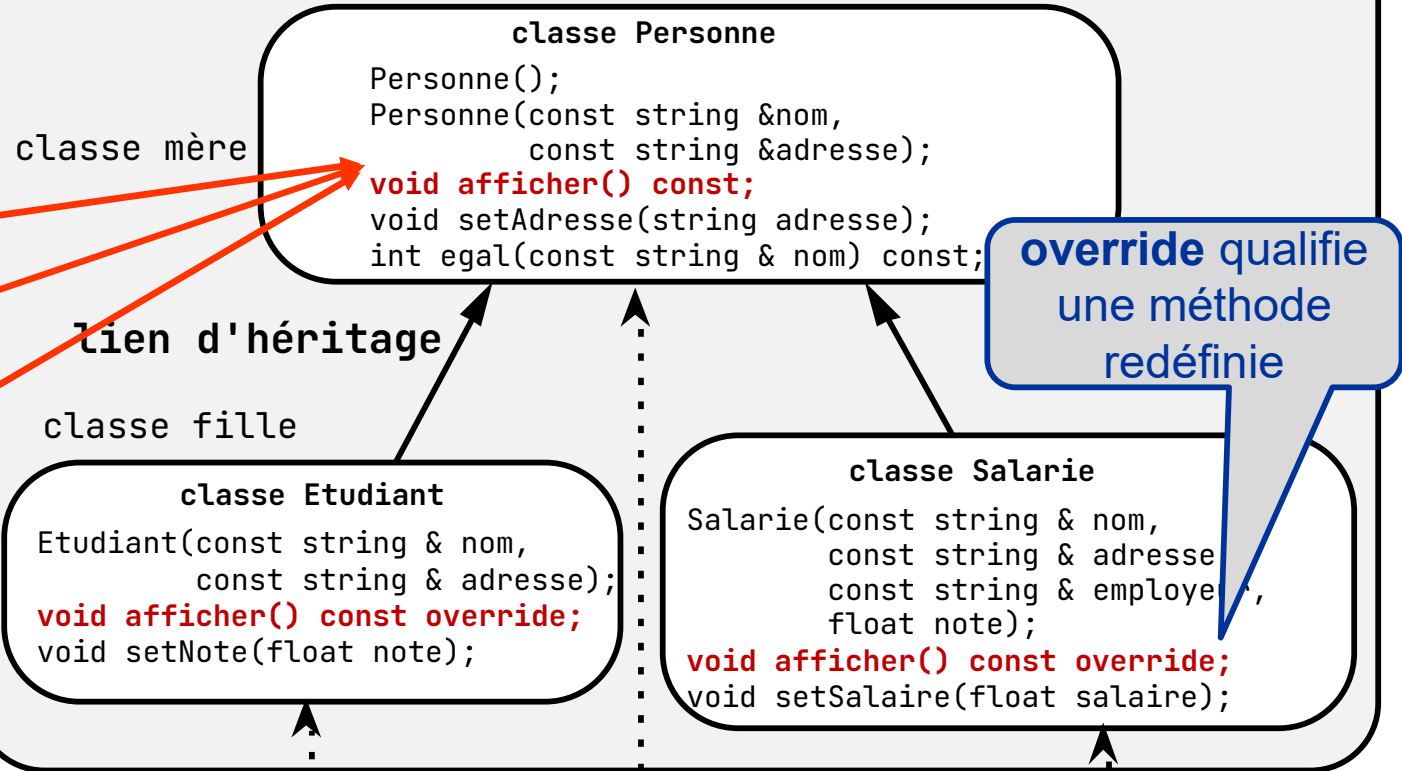
Par défaut : pas de polymorphisme en C++
« Lien Statique »

p1->afficher();

p2->afficher();

p3->afficher();

Arbre d'héritage des classes

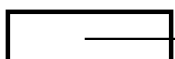


override qualifie une méthode redéfinie

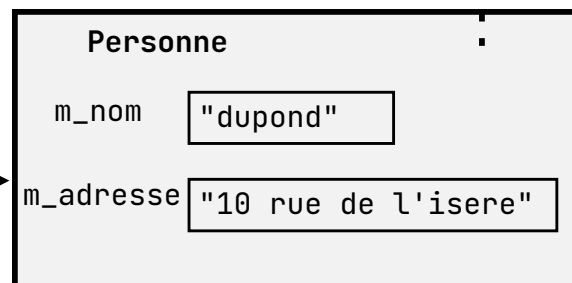
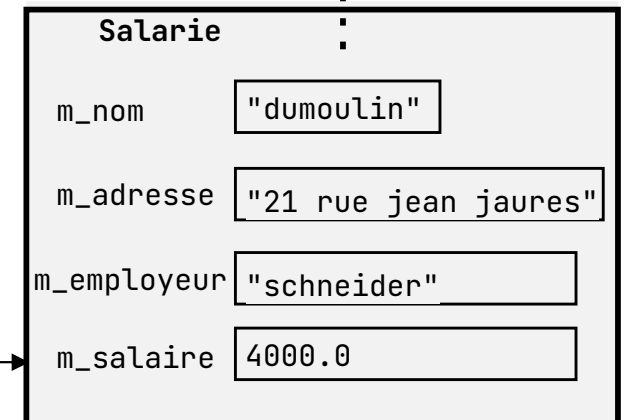
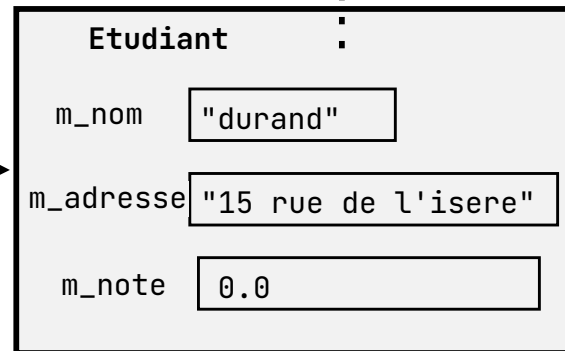
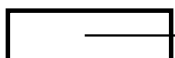
Personne * p3



Personne *p2



Personne *p1



8.4. Polymorphisme : Méthodes Virtuelles (3)

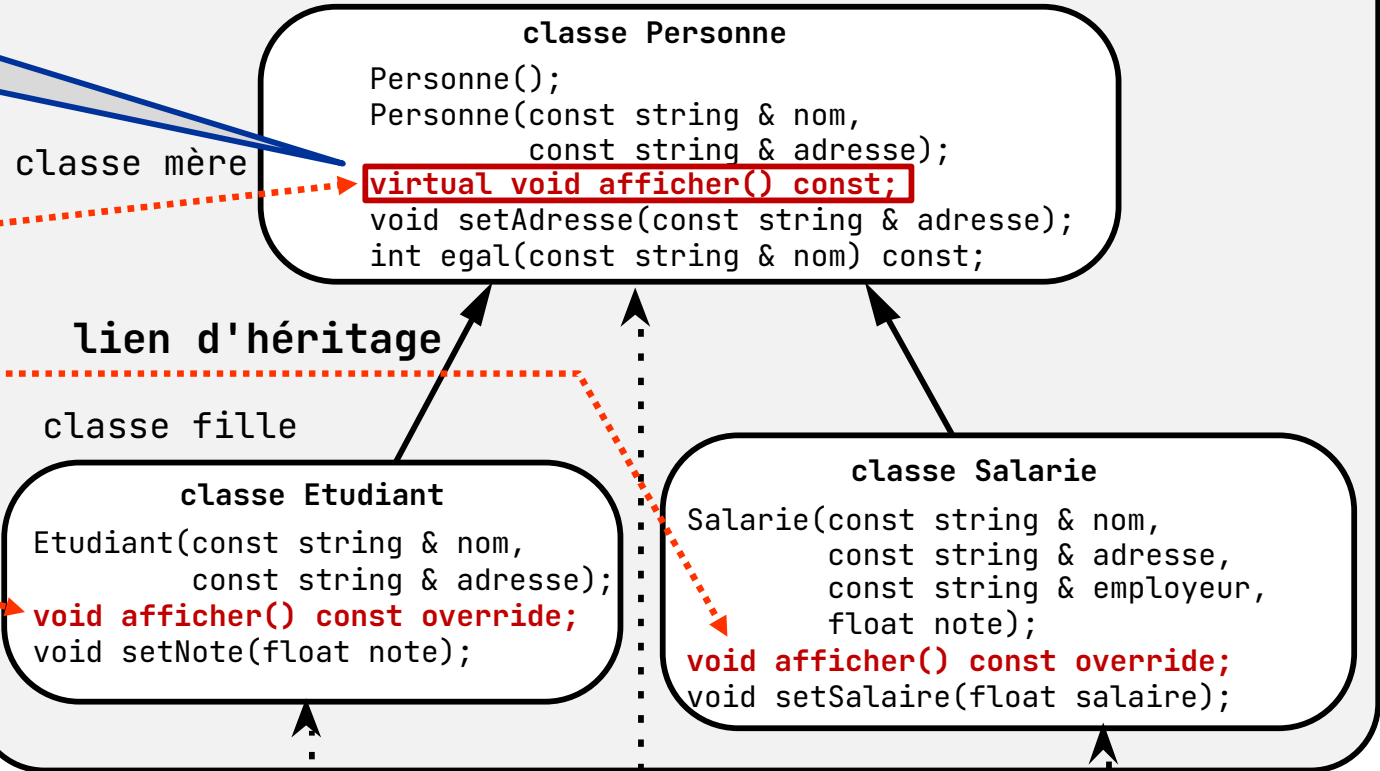
- L'appel de méthode (1) appellera **Personne::afficher** car :
 - ❑ p1 est déclaré pointeur sur un objet de classe **Personne**
 - ❑ le compilateur sait donc qu'il faut appeler cette méthode.
- Pour la même raison (p2 et p3 de type pointeur sur **Personne**) :
 - ❑ (2) et (3) appelleront **Personne::afficher**.
- Or, si l'on souhaite du **polymorphisme**, la méthode appelée devrait être :
 - ❑ **Salarie::afficher** pour l'appel de méthode (2)
 - ❑ **Etudiant::afficher** pour l'appel de méthode (3)

8.4. Polymorphisme : Méthodes Virtuelles (4)

- **Au moment de la compilation**, le compilateur :
 - ❑ ne connaît pas la classe de l'objet qui, **lors de l'exécution**, sera pointé par **p2**
 - ❑ ne peut donc pas appeler la méthode qui convient pour afficher cet objet.
- On parle de **lien statique** lorsque :
 - ❑ le compilateur appelle la méthode de la classe du pointeur
 - ❑ C'est un lien qui appelle toujours la même méthode
- Pour avoir du **polymorphisme**, le compilateur doit mettre en place un mécanisme qui :
 - ❑ **au moment** de l'exécution (dynamiquement)
 - ❑ appelle la méthode de la classe de l'objet pointé par le pointeur
- Le Mécanisme appelé **lien dynamique** est un lien :
 - ❑ qui détermine la méthode appelée à l'exécution du programme
 - ❑ qui n'appelle pas toujours la même méthode (pas systématiquement celle de la classe du pointeur)

Virtual nécessaire
pour obtenir un
« Lien Dynamique »

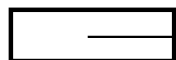
Arbre d'héritage des classes



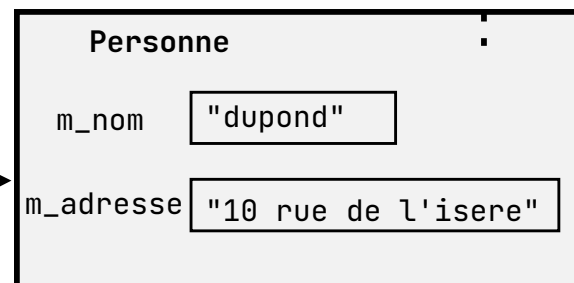
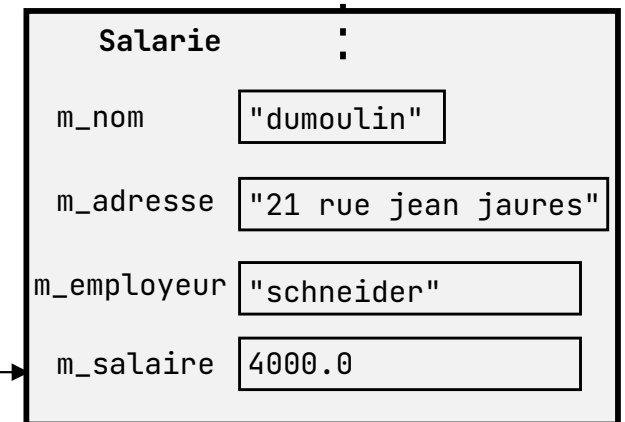
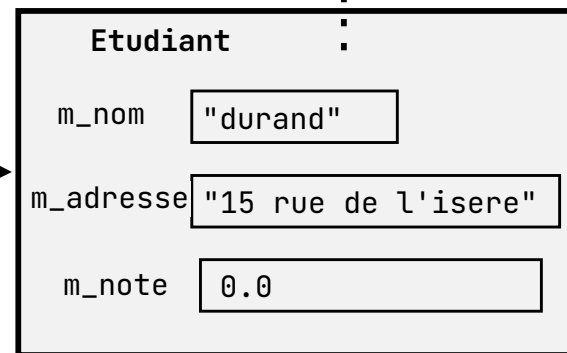
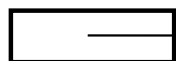
Personne * p3



Personne * p2



Personne * p1



8.4. Polymorphisme : Méthodes Virtuelles (5)

- Considérons l'appel de méthode :

```
Personne * p;
```

```
p = new Etudiant("durand", "15 rue de l'isere");
```

```
p->afficher(); // Doit appeler Etudiant::afficher  
               // lien dynamique
```

- Pour que le lien `->afficher()` soit dynamique, Il faut que dans la classe `Personne`, la méthode `afficher` ait été déclarée **virtuelle** (**virtual**) dans la classe **Personne**

8.4. Polymorphisme : Méthodes Virtuelles (6)

```
class Personne {  
  
protected:  
    std::string m_nom;  
    std::string m_adresse;  
public:  
    Personne();  
    Personne(const std::string & nom,  
              const std::string & adresse);  
    virtual void afficher() const;  
    void setAdresse(const std::string & adresse);  
    int egal(const std::string & nom) const;  
};
```

Avec **virtual**,
l'appel à la méthode **afficher**
via un pointeur de type
Personne* ou via une
référence de type **Personne&**
sera dynamique :
il y aura bien polymorphisme !

8.4. Polymorphisme : Méthodes Virtuelles (7)

- **Personne::afficher** étant déclarée **virtual**, le compilateur met en place un lien dynamique lors de l'invocation de la méthode **afficher** via un **pointeur** ou une **référence** à une **Personne** :

```
Personne * p;  
p=new Etudiant("durand", "15 rue de l'isere");  
p->afficher(); // Etudiant::afficher() est appelée
```

- La propriété « virtual » est propagée dans l'arbre d'héritage
 - Soit une classe **SalarieAuMois**, dérivée de **Salarie** et qui redéfinit elle aussi la méthode **Salarie::afficher**
 - Dès lors que **Personne::afficher** a bien été déclarée **virtual**, le compilateur mettra **implicitement** en place un lien dynamique, même si **Salarie::afficher** n'a pas été déclarée **virtual** :

```
Salarie * p; // p pointeur sur un Salarie  
p=new SalarieAuMois("durand", "15 rue de l'isere", 4000.00);  
p->afficher(); // SalarieAumois::afficher() est appelée
```

8.4. Polymorphisme : Méthodes Virtuelles (8)

- Une méthode d'une classe de base doit être déclarée **virtual** :
 - ❑ si on sait que la méthode sera redéfinie (**override**) dans une classe dérivée
 - ❑ et si on va manipuler des objets de la classe de base ou de ses classes dérivées *via* des **pointeurs** ou des **références**

```
// Classe de Base définissant une méthode
class ClasseDeBase { virtual void uneMethode(...) const {...}};

// Classe Dérivée redéfinissant cette méthode
class ClasseDerivee : public ClasseDeBase { void uneMethode(...) const override {...}};

// Fonction avec passage d'un objet par référence
void uneFonction(const ClasseDeBase & unObjet) {
    unObjet.uneMethode(...); // Polymorphisme si uneMethode est virtual dans ClasseDeBase
}

int main() {
    ClasseDerivee objetDerive;
    uneFonction(objetDerive); // Passage d'un objet par référence
    return 0;
}
```


8.4. Polymorphisme : Méthodes Virtuelles (9)

- Les méthodes **virtuelles** sont indispensables pour pouvoir mettre en œuvre le **polymorphisme**
- Dans la plupart des langages orientés objets, les méthodes sont implicitement virtuelles
- C'est pour des raisons d'efficacité que C++ oblige le programmeur à spécifier explicitement les méthodes virtuelles qui seront appelées avec un lien dynamique
- En effet un lien statique est plus efficace (en temps d'exécution) qu'un lien dynamique