ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ
2018-2019

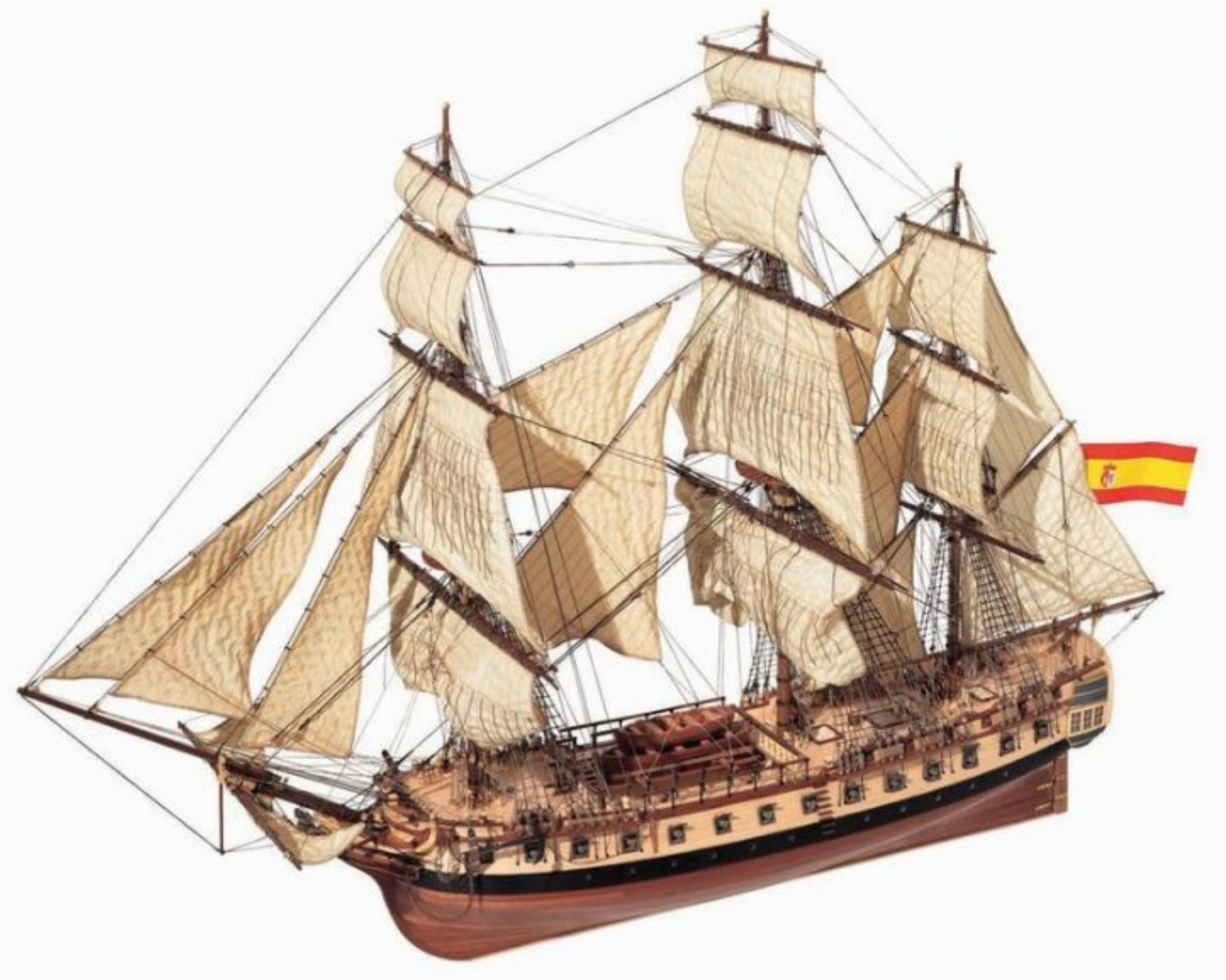
Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. *C++ pratique 1*		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. Classes & C++ : compléments		6. *UML et C++, associations*
7. *Conteneurs & C++ : la STL*		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. *Modèles objets avancés*		9. *Modèle composite et graphismes*
10. *Exceptions, flots, fichiers ...*		10. *Persistance / fichiers / except.*
11. *Templates côté développeur*		11. *Développement de templates*
12. *Gestion mémoire / smarts ptrs*		12. *Soutenance de **projet** ...*

Classes & C++ : compléments



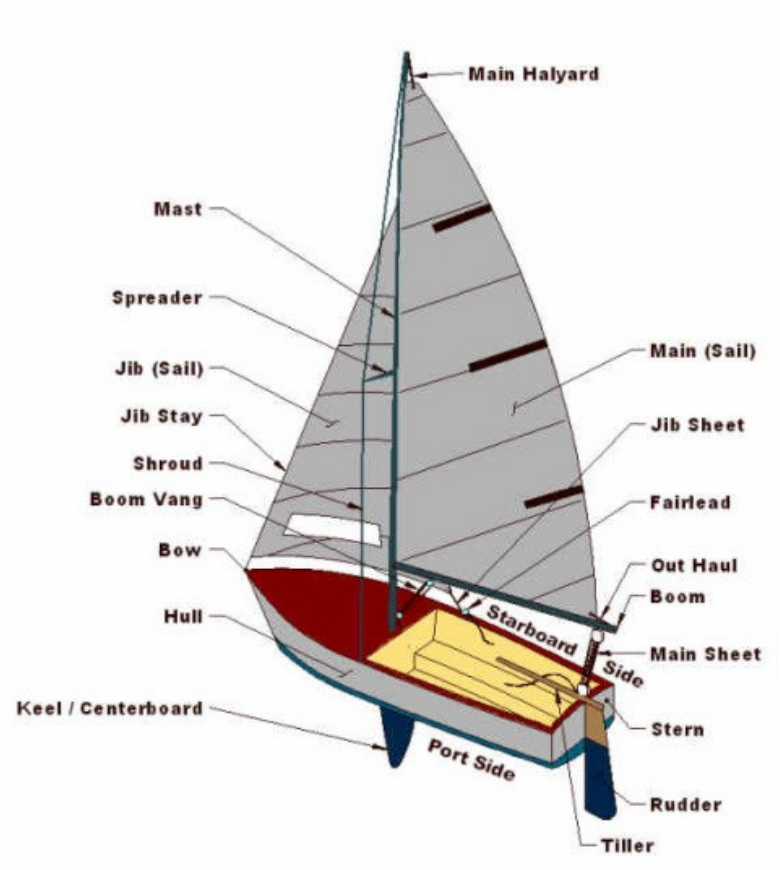
COURS 6

A) Du modèle objet au C++
B) Types valeur / types entité
C) Copiabilité en C++
D) Composition en C++
E) Associations à sens unique
F) Associations à double sens

COURS 6

A) Du modèle objet au C++
B) Types valeur / types entité
C) Copiabilité en C++
D) Composition en C++
E) Associations à sens unique
F) Associations à double sens

Du modèle objet au C++



Du modèle objet au C++

- *Le modèle objet exprimé en UML est une abstraction indépendante d'un langage de programmation spécifique*
- *Différents langages de programmation orienté objet ont différentes syntaxes pour exprimer les mêmes concepts mais aussi des mécanismes différents*
- *Le C++ est un « langage système » qui permet d'implémenter un modèle objet au plus près des ressources matérielles (close to metal)*
- *Les performances optimales réalisables se payent au prix d'une plus grande complexité...*

Du modèle objet au C++

- *Classes → classes Méthodes → méthodes(val. / adr. / ref.)*
- *Instances → objets alloués à certaines adresses RAM*
- *Attributs → attributs « types valeurs »*
- *Compositions → attributs « types valeurs » (souvent)*
- *Compositions facultatives ou lourdes*
  → attributs pointeurs sur ressources exclusives
  → attributs vecteurs de ressources exclusives
  (vecteur = valeur qui cache un pointeur → zone allouée)
- *Associations sens unique → attributs pointeurs*
- *Associations double sens → pointeurs réciproques*
- *Multiplicité 0..N → vecteurs de valeurs ou de pointeurs*

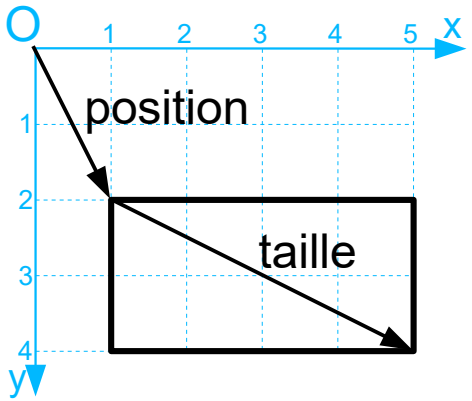
Du modèle objet au C++

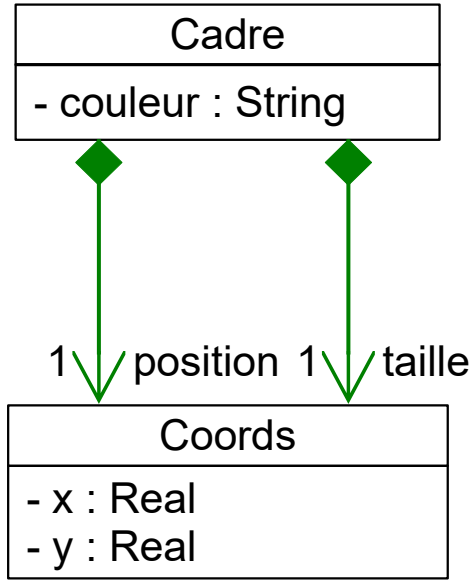
- *Classes → classes Méthodes → méthodes(val. / adr. / ref.)*
- *Instances → objets alloués à certaines adresses RAM*
- *Attributs → attributs « types valeurs »*
- *Compositions → attributs « types valeurs » (souvent)*
- *Compositions facultatives ou lourdes*
  → attributs pointeurs sur ressources exclusives
  → attributs vecteurs de ressources exclusives
  (vecteur = valeur qui cache un pointeur → zone allouée)
- *Associations sens unique → attributs pointeurs*
- *Associations double sens → pointeurs réciproques*
- *Multiplicité 0..N → vecteurs de valeurs ou de pointeurs*

Du modèle objet au C++

- *Dans un modèle UML il y a beaucoup de **flèches***
- *Dans un code C++ il y a beaucoup de **pointeurs** et d'histoires d'**adresses mémoire***

Ces flèches de navigation UML...





Ne correspondent à aucun pointeur ici !

```
class Cadre
{
    private :
        Coords m_position;
        Coords m_taille;
        std::string m_couleur;

    public :
        ...
};
```

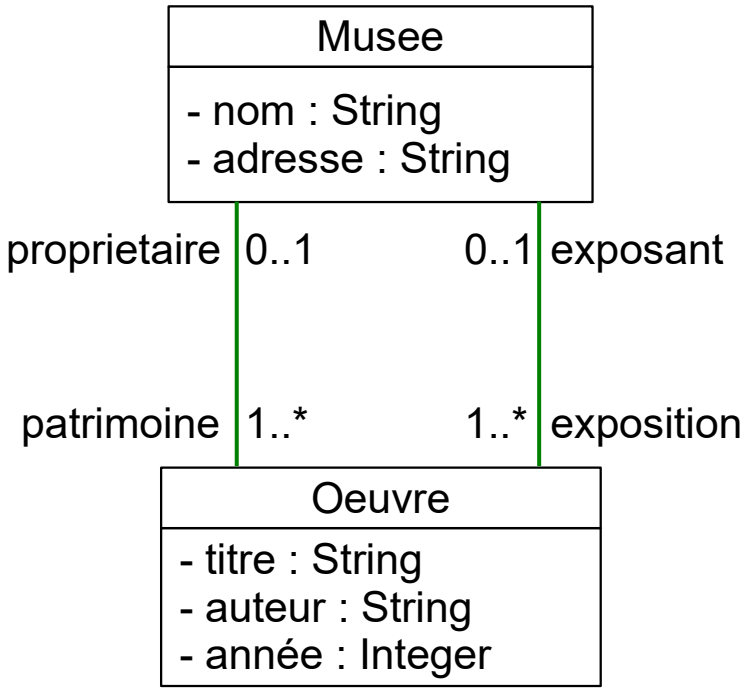
```
class Coords
{
    private :
        double m_x;
        double m_y;

    public :
        ...
};
```

Du modèle objet au C++

- *Dans un modèle UML il y a beaucoup de **flèches***
- *Dans un code C++ il y a beaucoup de **pointeurs** et d'histoires d'**adresses mémoire***

Double navigation UML
pas de flèche...



Plein de pointeurs !

```
class Oeuvre
{
    private :
        /// Attributs valeurs simples
        std::string m_titre;
        std::string m_auteur;
        int m_annee;

        /// Attribut références à des entités
        Musee* m_proprietaire;
        Musee* m_exposant;

    public :
        Oeuvre(std::string titre,std::string auteur);

        void setProprietaire(Musee* m);
        Musee* getProprietaire() const;

        void setExposant(Musee* m);
        Musee* getExposant() const;

};
```

Du modèle objet au C++

- *Dans un modèle UML il y a beaucoup de **flèches***
- *Dans un code C++ il y a beaucoup de **pointeurs** et d'histoires d'**adresses mémoire***

Flèche UML ≠ pointeur

Du modèle objet au C++

- *De même attention à la polysémie (sens multiple)*

  *Une **référence** au sens **conception objet** est quelque chose qui permet de **désigner** un objet*
  Ça peut s'implémenter de différentes façons :
- ***Référence** au sens mécanisme **C++** :* `Type&`
- ***Pointeur** au sens mécanisme C++ :* `Type*`
- ***Index** entier sur un élément dans vecteur*
- ***Nom textuel** permettant de retrouver l'élément*
- ***Clé** dans une base de donnée*

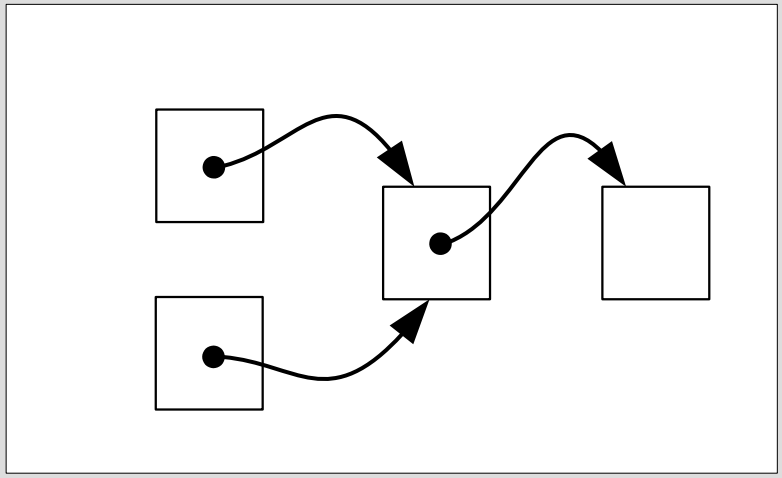
Référence POO ≠ Référence C++

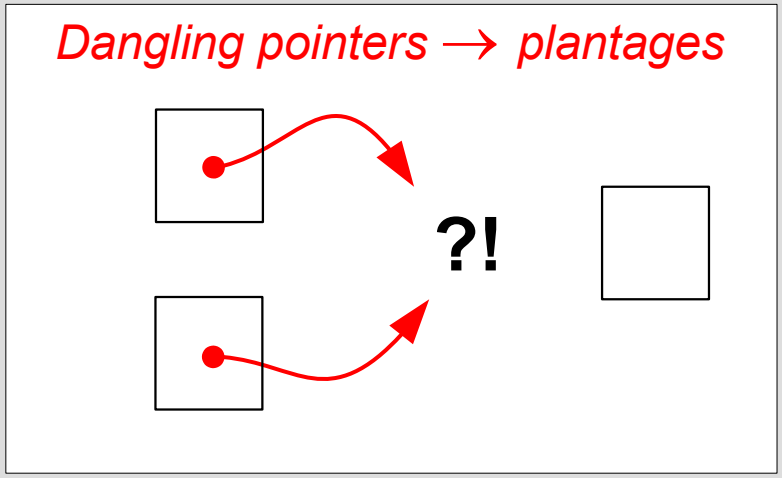
Du modèle objet au C++

- *Les références au sens technique C++* `Type&` *sont en fait des « pointeurs cachés »*
- *Qu'on utilise des pointeurs ou des références :*

  Un objet sait qui il pointe mais il ne sait pas qui le pointe...

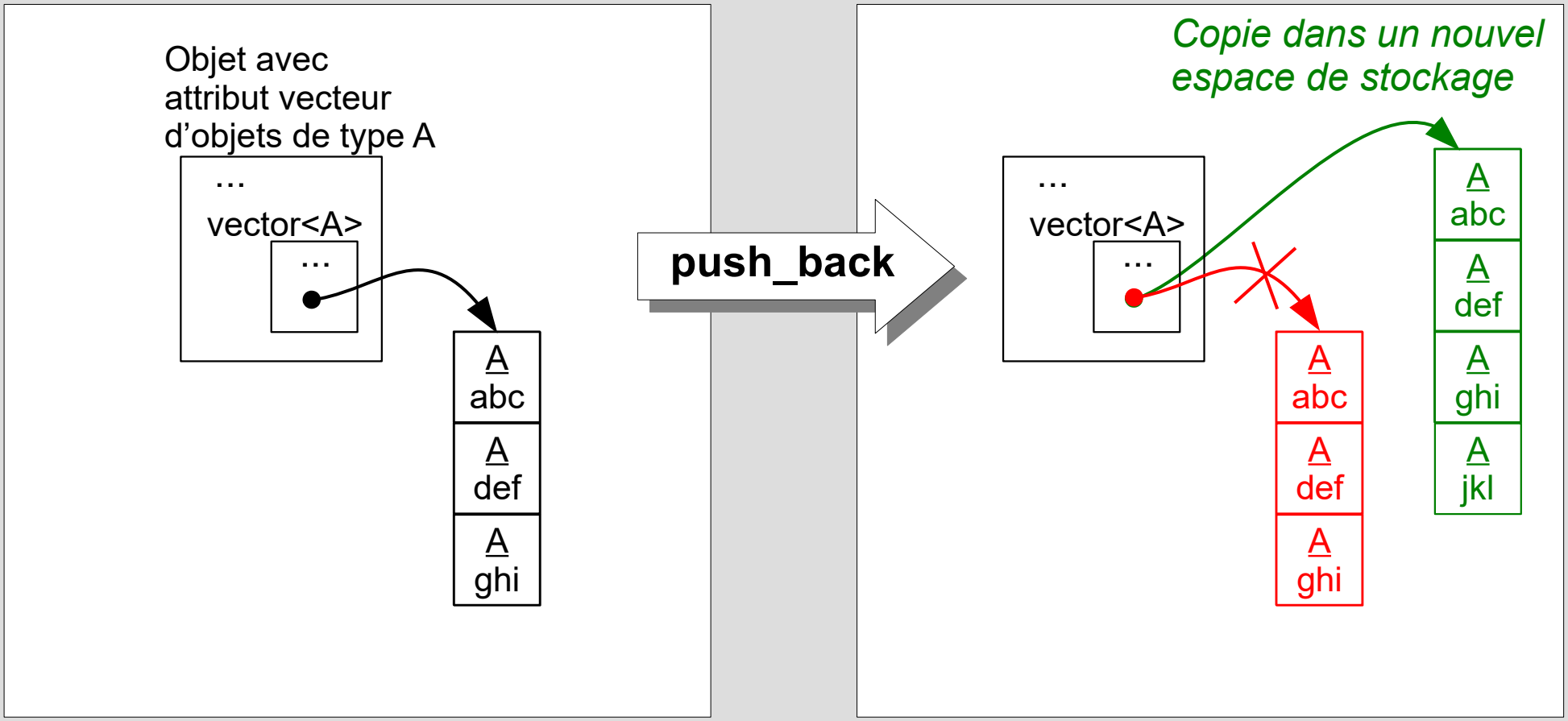
  Problème : si un objet pointé est détruit il ne peut pas facilement avertir les pointants





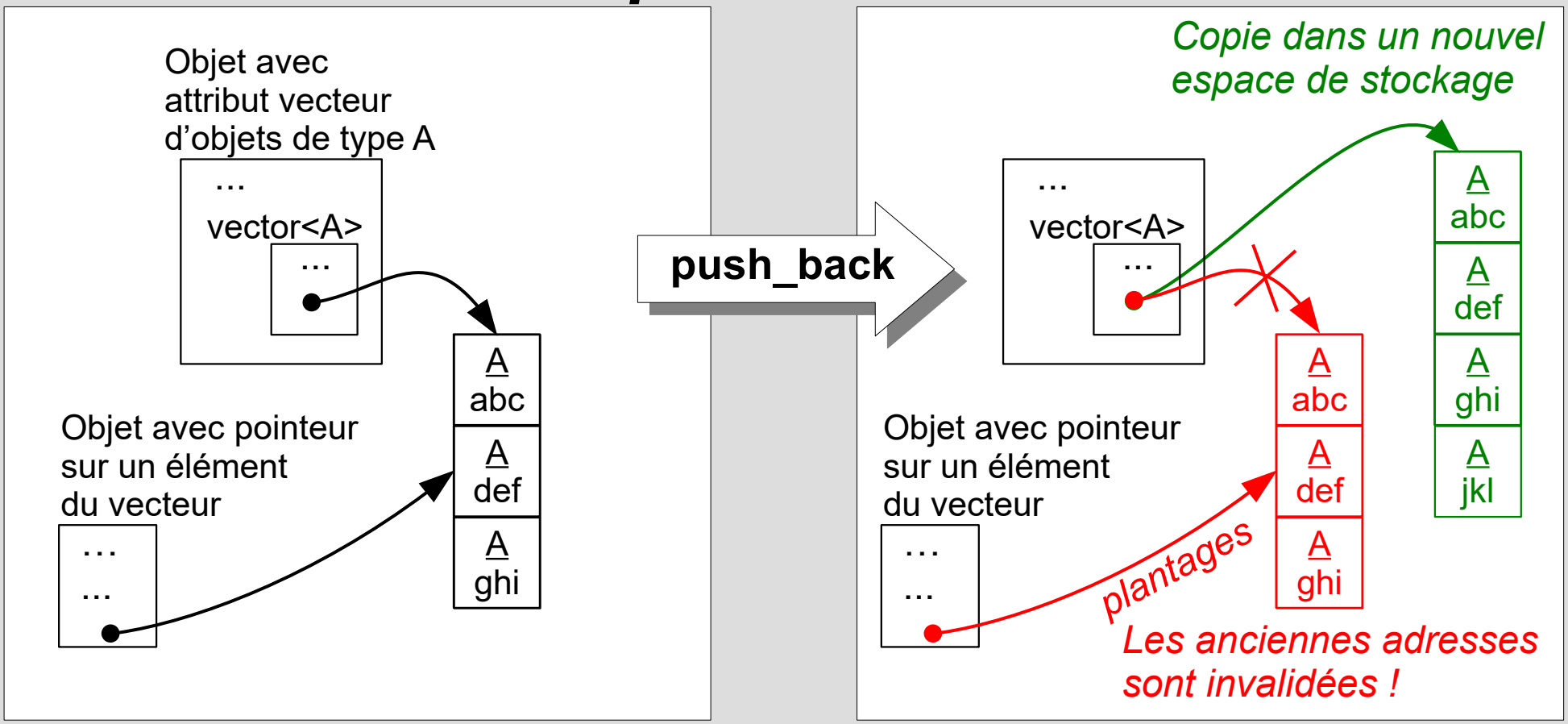
Du modèle objet au C++

- *Les vecteurs sont comme des tableaux qui grossissent magiquement mais en fait il y a ré-allocation...*



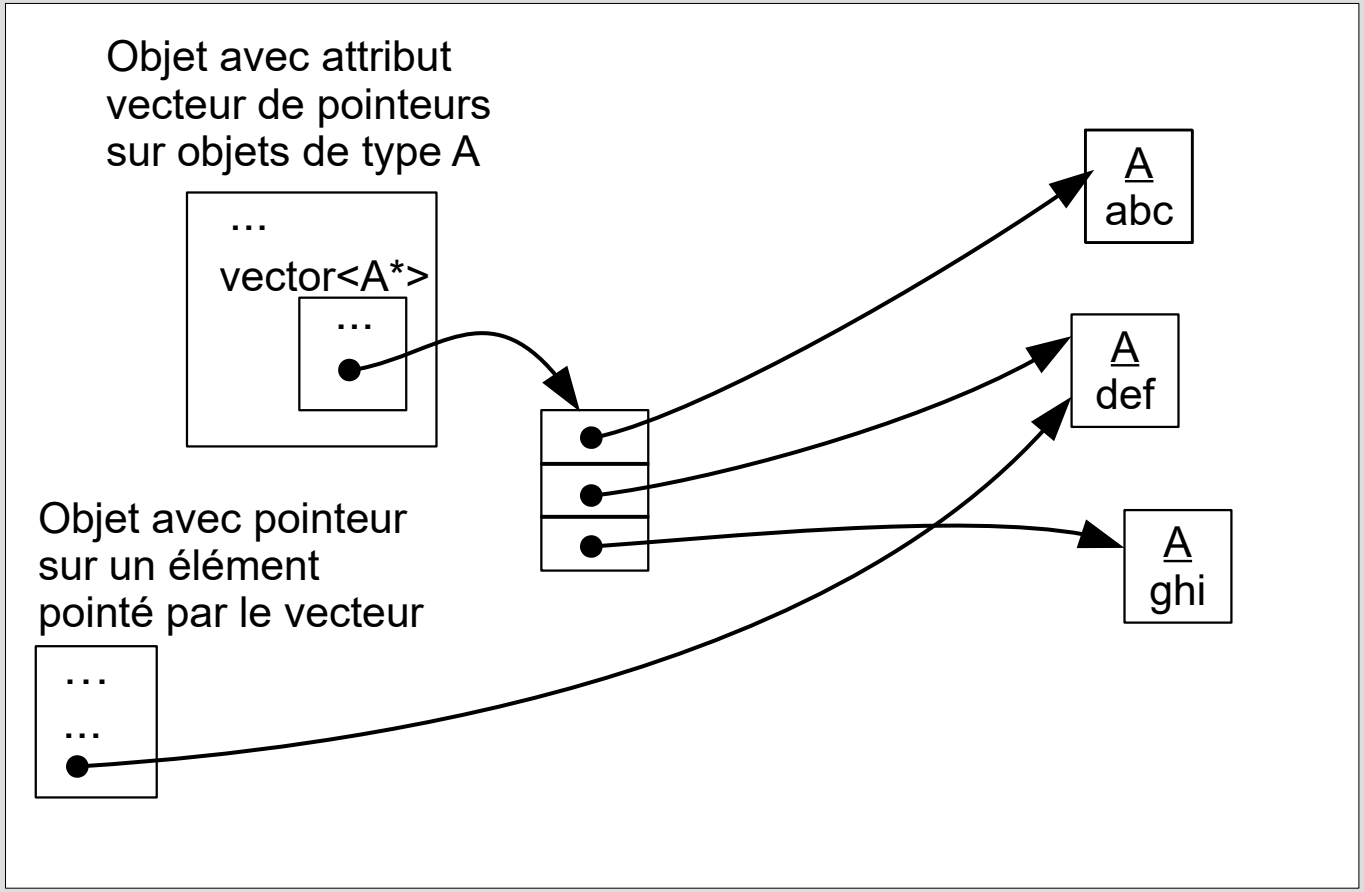
Du modèle objet au C++

- *Les vecteurs sont comme des tableaux qui grossissent magiquement mais en fait il y a ré-allocation :* ***les adresses des éléments stockés ne sont pas stables !***



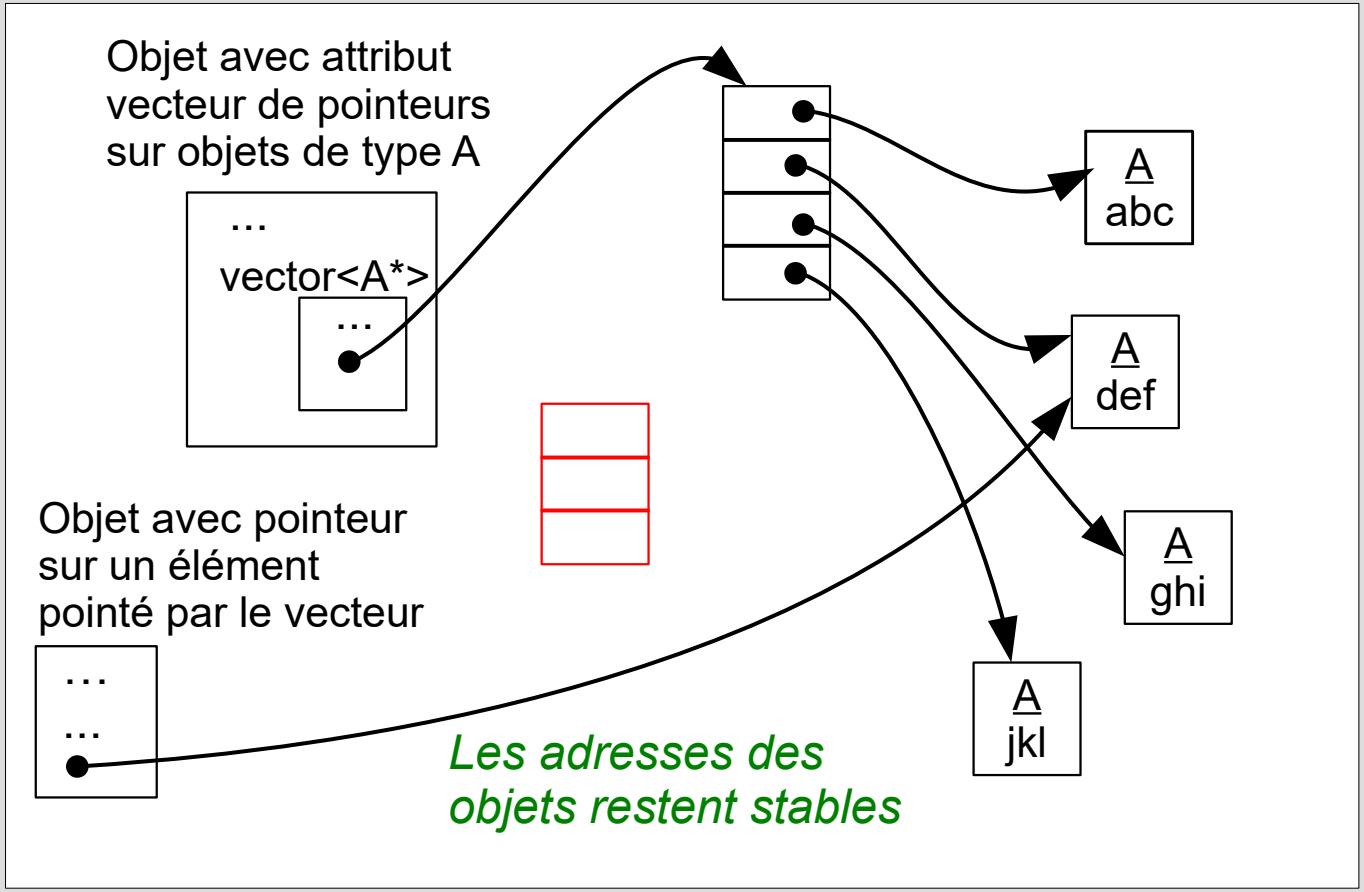
Du modèle objet au C++

- *Ce léger inconvénient n'existe pas avec d'autres conteneurs (list, map... Cours 7)*
- *Sinon stocker des **pointeurs** sur des objets...*



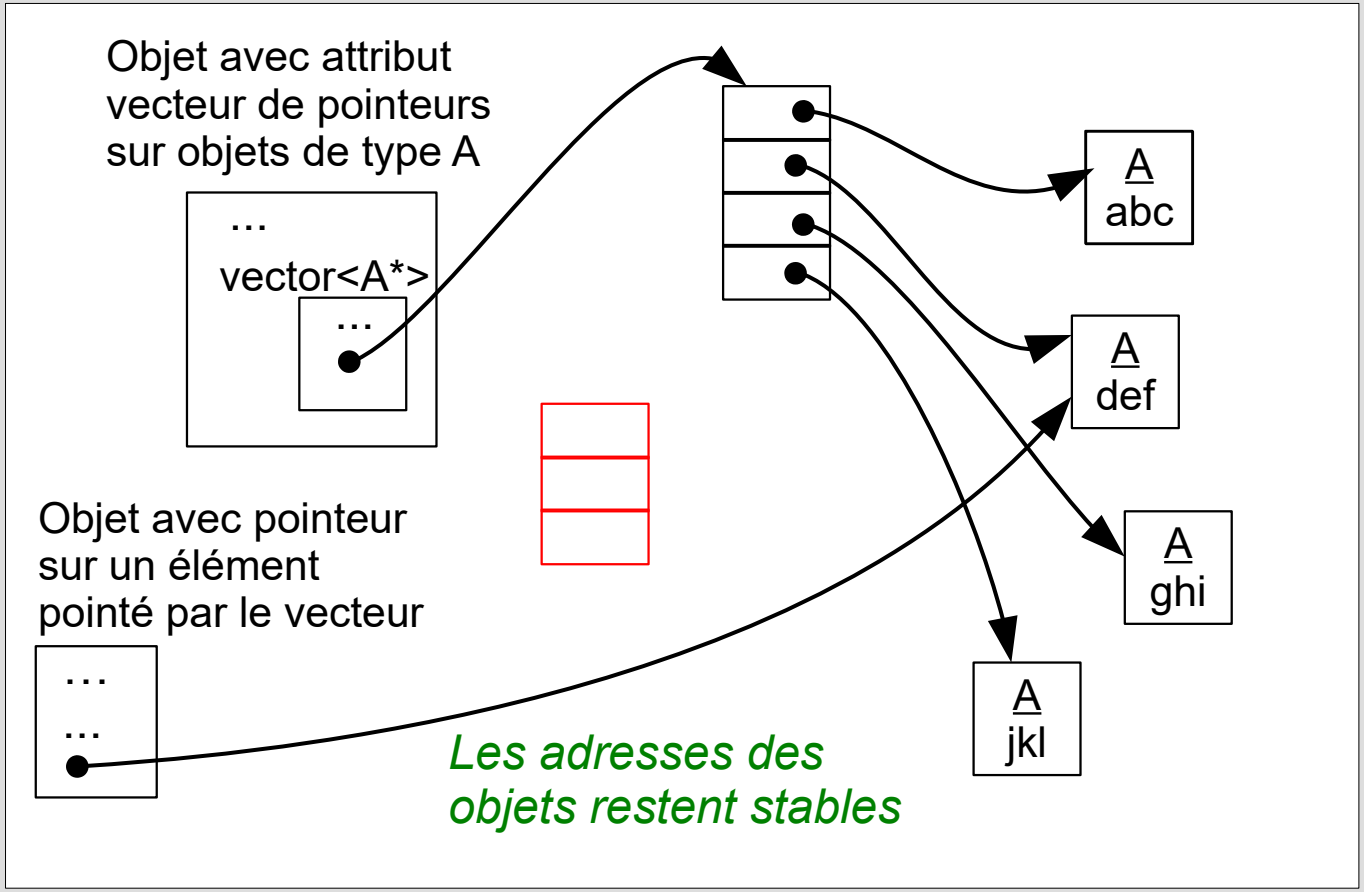
Du modèle objet au C++

- *Ce léger inconvénient n'existe pas avec d'autres conteneurs (list, map... Cours 7)*
- *Sinon stocker des **pointeurs** sur des objets...*



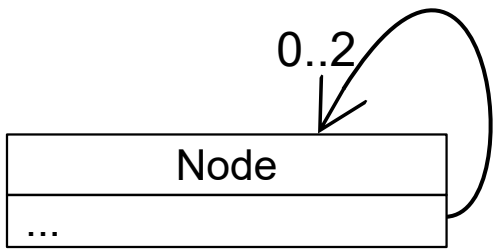
Du modèle objet au C++

- *Stocker des pointeurs sur objets plutôt que des objets implique d'utiliser l'allocation dynamique*
  - → *on préfère éviter (Cf cours 5, cycle de vie)*
  - → *pas trop le choix pour certains objets dynamiques*

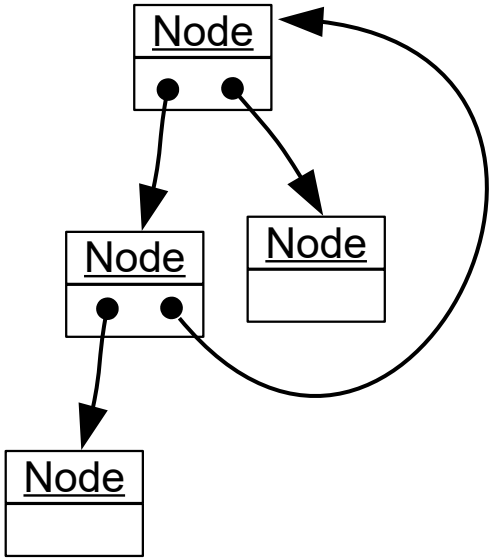


Du modèle objet au C++

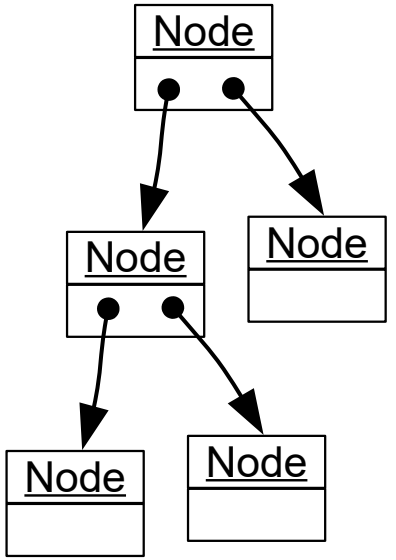
- *Dans le passage du modèle à l'implémentation on a parfois des soucis pour interpréter les contraintes de structure / topologie :* ***le modèle UML ne dit pas tout****...*
- *Cycles ou pas : pas les mêmes algos !*



Classes UML



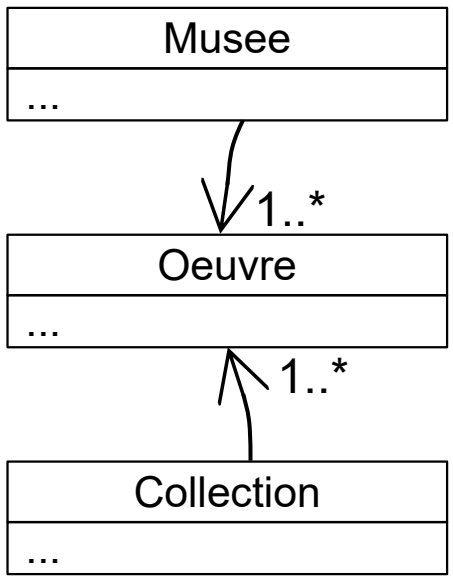
Objets : Cycles ?



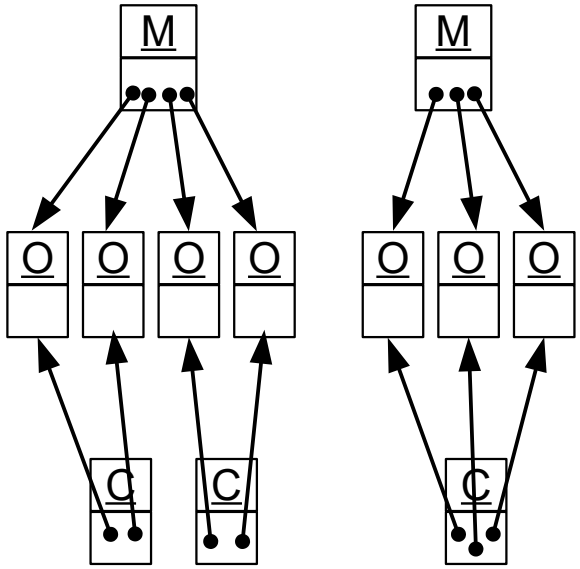
Objets : Pas cycles ?

Du modèle objet au C++

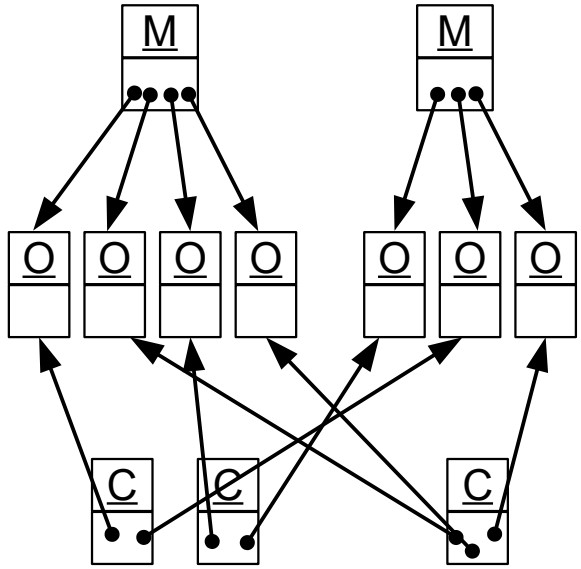
- *Dans le passage du modèle à l'implémentation on a parfois des soucis pour interpréter les contraintes de structure / topologie :* ***le modèle UML ne dit pas tout****...*
- *Groupes séparables ou pas ?*



Classes UML



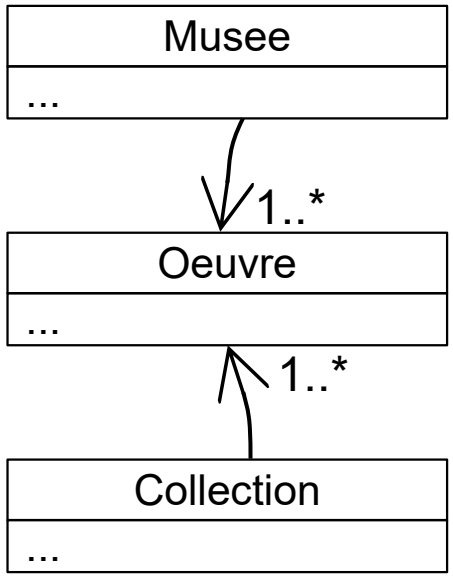
Objets : groupés ?



Objets : pas groupés ?

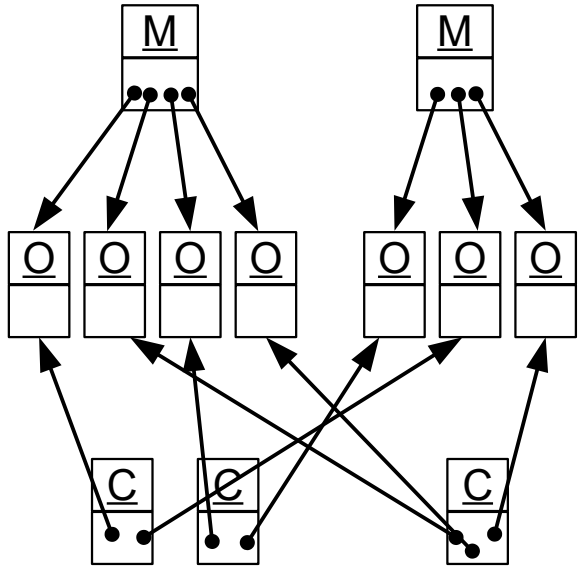
Du modèle objet au C++

- *En fait c'est le **diagramme de classes UML** spécifiquement qui ne dit pas tout, en principe un bon modèle **précise par écrit** ces aspects*



Classes UML

Bla bla bla bla bla bla bla bla bla bla bla bla bla bla une collection pourra regrouper des œuvres qui se trouvent dans différents musées bla bla bla bla bla bla bla bla bla bla bla bla



Objets : pas groupés !

Du modèle objet au C++

- *En général il faudra bien distinguer 2 façons de grouper les instances :*
- *Regroupement **par type** : **classes***
  C'est un regroupement abstrait
  Exemple « les œuvres d'art »
  Toutes dans le même sac !
- *Regroupement **en collections** : **conteneurs***
  vecteur, tableau, liste, map … (Cours 7)
  C'est un regroupement concret et plus fin
  Exemples :
  « les œuvres de la collection Christina H. Kang »
  « les œuvres de la collection François Pinault »
  ...

Du modèle objet au C++

- *En général il faudra bien distinguer 2 façons de grouper les instances :*
- *Regroupement **par type** : **classes***
- *Regroupement **en collections** : **conteneurs***

- *La confusion risque de se produire quand on a qu'une seule collection pour toutes les instances d'une classe*
- *Exemple :*
  au TD/TP 5 toutes les instances de Astre sont dans un seul et même vecteur...

Du modèle objet au C++

- *Une application orientée objet c'est*
  - *Des allocations mémoire d'objets*
  - *Des initialisations de données d'objets*
  - *Des délétions/libération mémoire d'objets*
  - *Des copies de données d'objets*
- *Des pointeurs entre des zones mémoires, certaines adresses stables, d'autres instables*
- *Trop de pointeurs, un « tissu » de pointeurs*
- *Quand on tire sur un pointeur, tout vient !*

Du modèle objet au C++

- *Pour que la transformation du modèle UML en implémentation C++ ne soit pas un fiasco :*

  - *Il faut définir des règles du jeu sur le terrain* ***qui a le ballon (le pointeur sur un objet) ?***
  - *Il faut avoir des règles claires dans les vestiaires* ***à qui est cette paire de crampons (ces 8 octets) ?***
  - *En résumé : quel objet aura, à quel moment, la responsabilité des données et de leur espace de stockage ? C'est la problématique de la* ***propriété : ownership*** *...*

COURS 6

A) Du modèle objet au C++
B) Types valeur / types entité
C) Copiabilité en C++
D) Composition en C++
E) Associations à sens unique
F) Associations à double sens

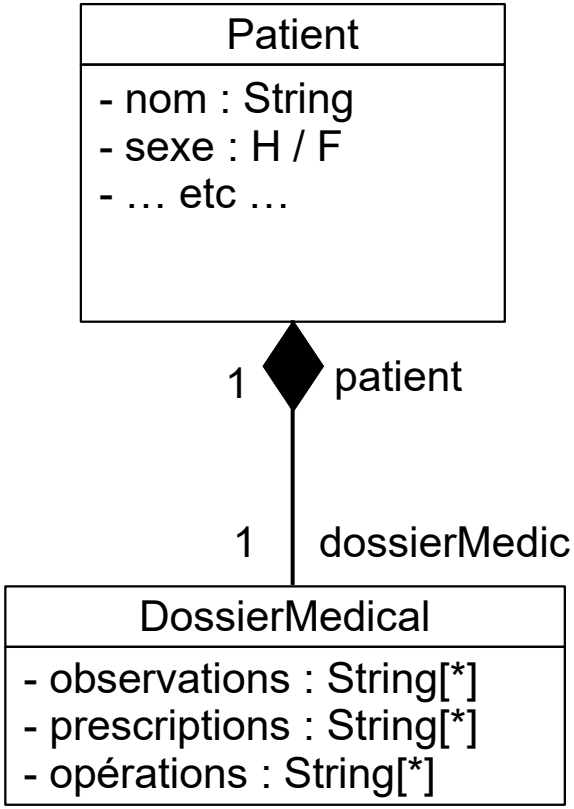
Types valeur / types entité

{25, HUN} is a composite value

The Black Pearl is an entity

Types valeur / types entité

- *Dans un diagramme de classes UML un symbole de **composition** indique un lien fort de propriété...*



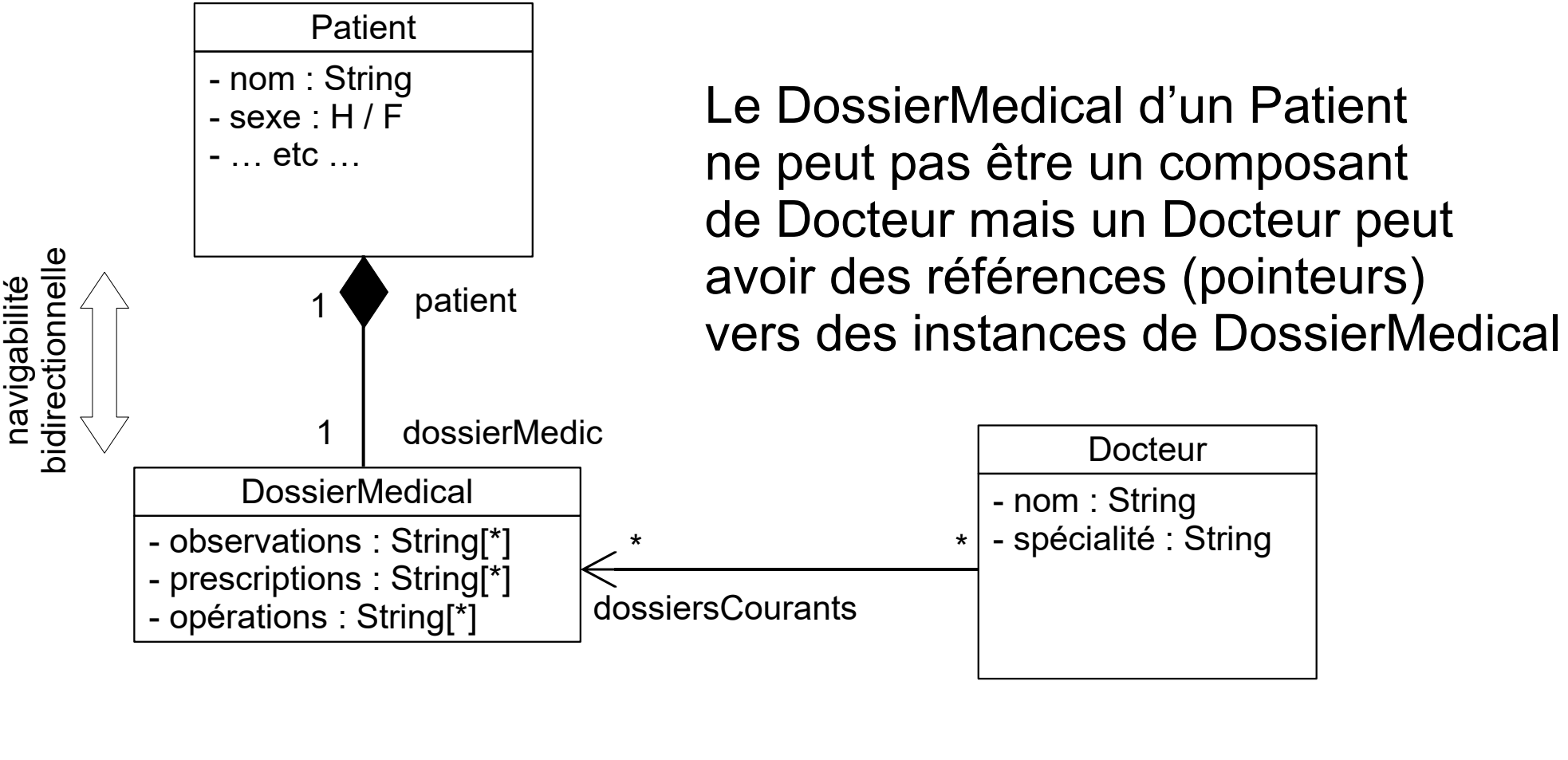
Un objet patient est **propriétaire** d'un objet DossierMedical

Un dossier médical est intimement lié à la personne qu'il représente, il ne peut pas devenir le dossier médical de quelqu'un d'autre !

Si on détruit (l'objet informatique local) Patient alors on libère (l'objet informatique local) Dossier

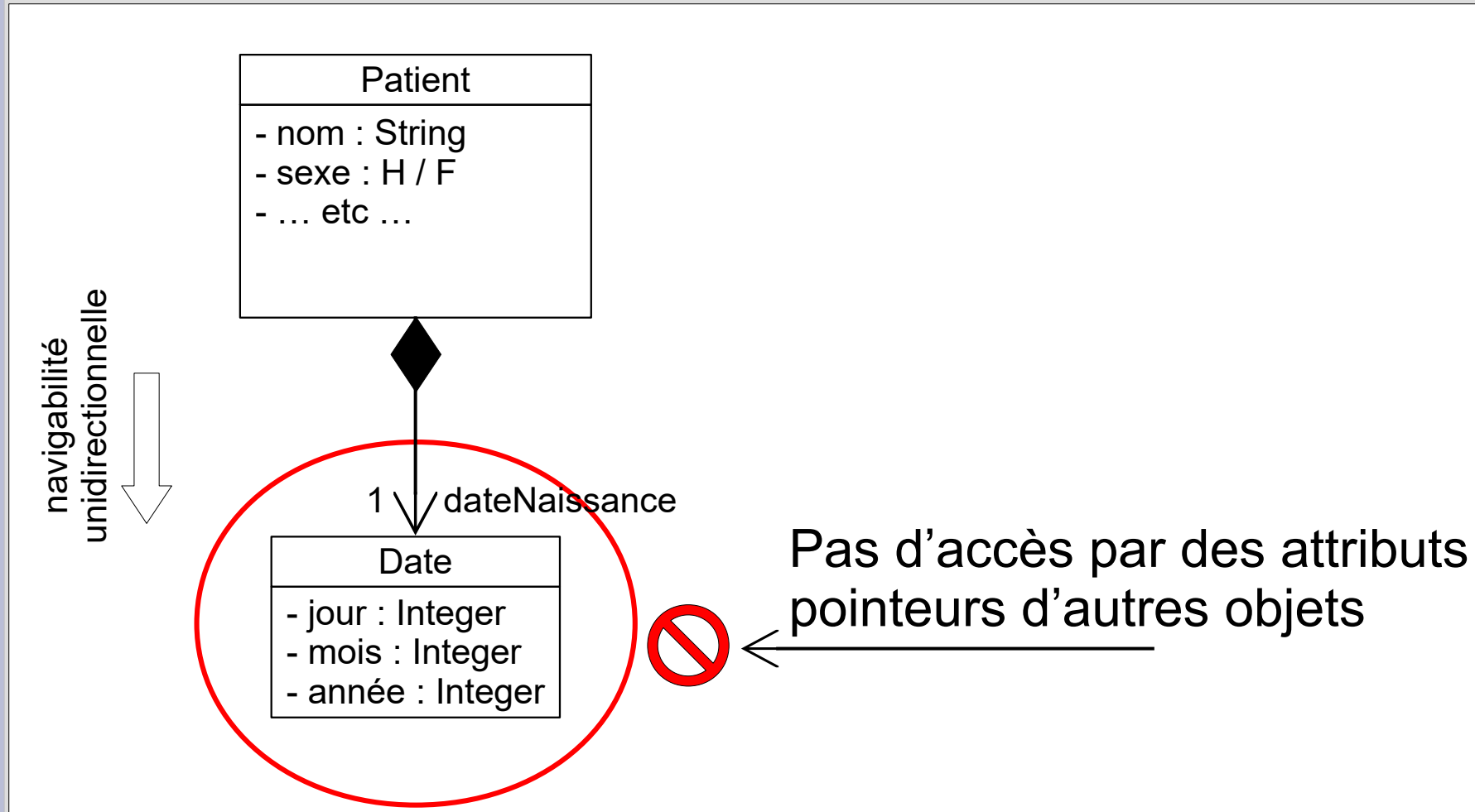
Types valeur / types entité

- *Non partageable en tant que composant, n'empêche pas d'autres objets d'être en **association** avec un composant ...*



Types valeur / types entité

- ***Certains types composants*** *n'ont pas vocation à être référencés (pointés) par d'autres objets...*



Types valeur / types entité

- *Le composite est alors le seul à « donner accès »*
- *Données identiques ≠ même objet composant !*

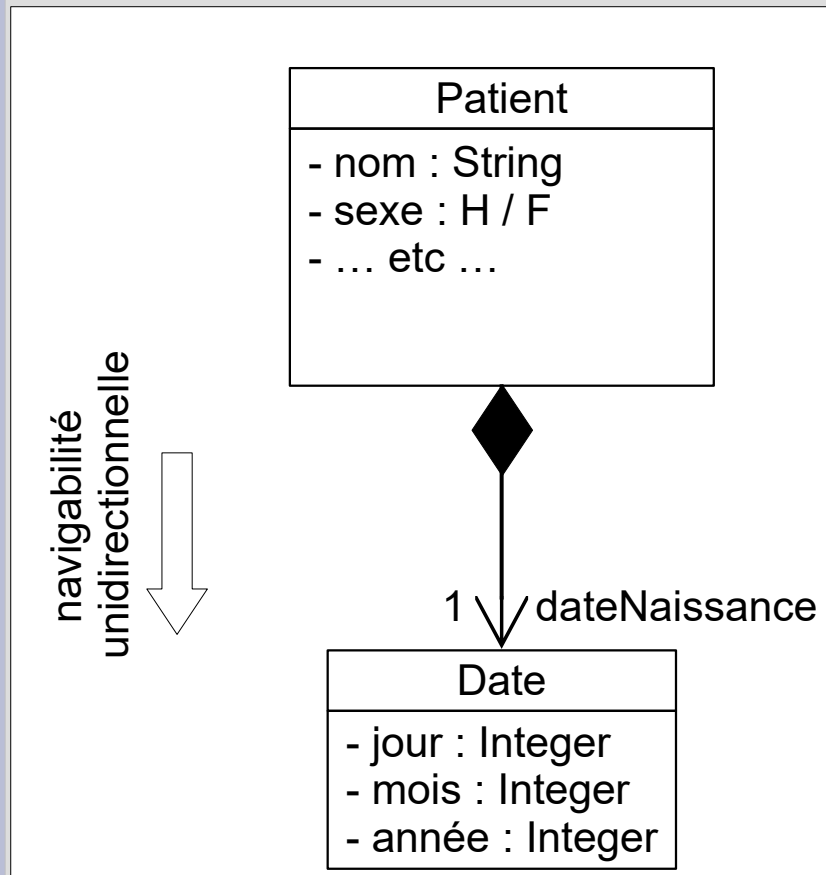


Diagramme de classes

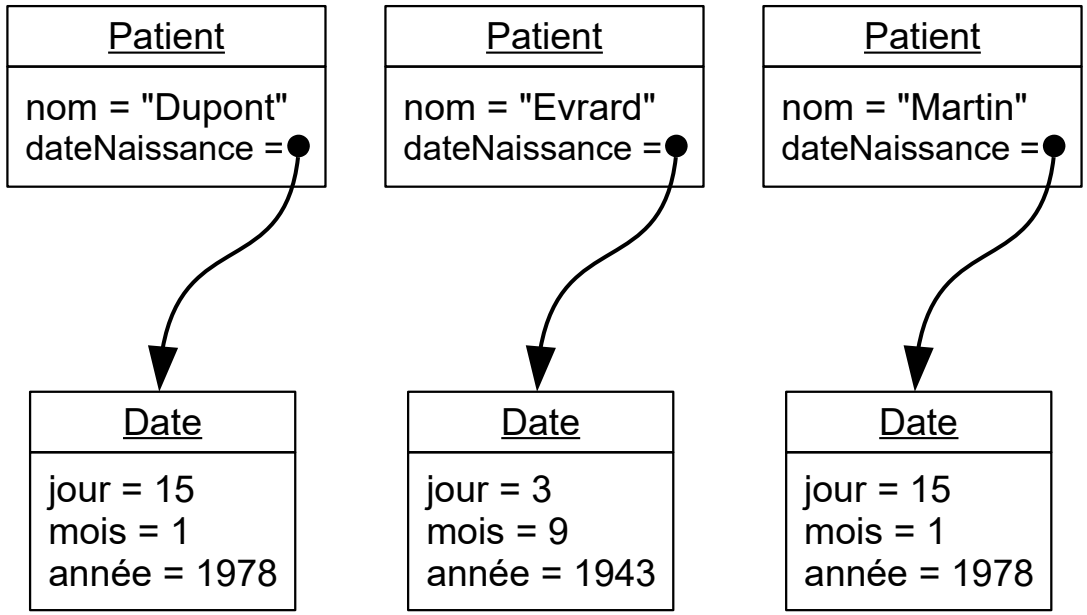


Diagramme d'objets

Types valeur / types entité

- *Le composite est alors le seul à « donner accès »*
- *Données identiques ≠ même objet composant !*

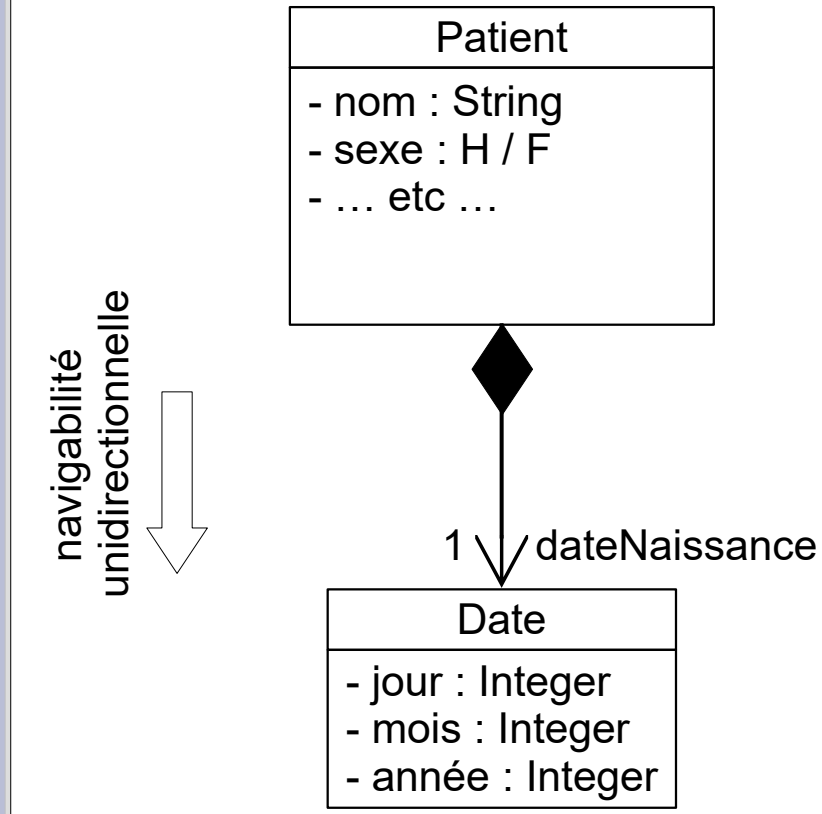
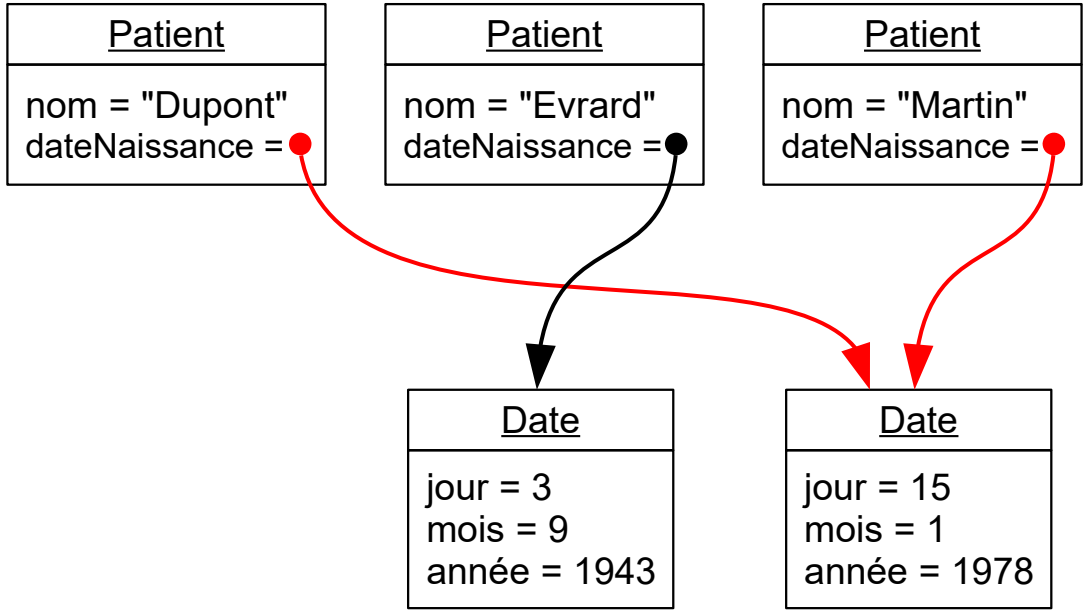


Diagramme de classes



Mauvaise implémentation :
Chaque patient devrait avoir sa dateNaissance à lui !

Diagramme d'objets

Types valeur / types entité

- *En **C++** on préfère alors déclarer l'attribut "**par valeur**"*
- *Attribut* `Date m_dateNaissance;` *et non pas* `Date*` *...*

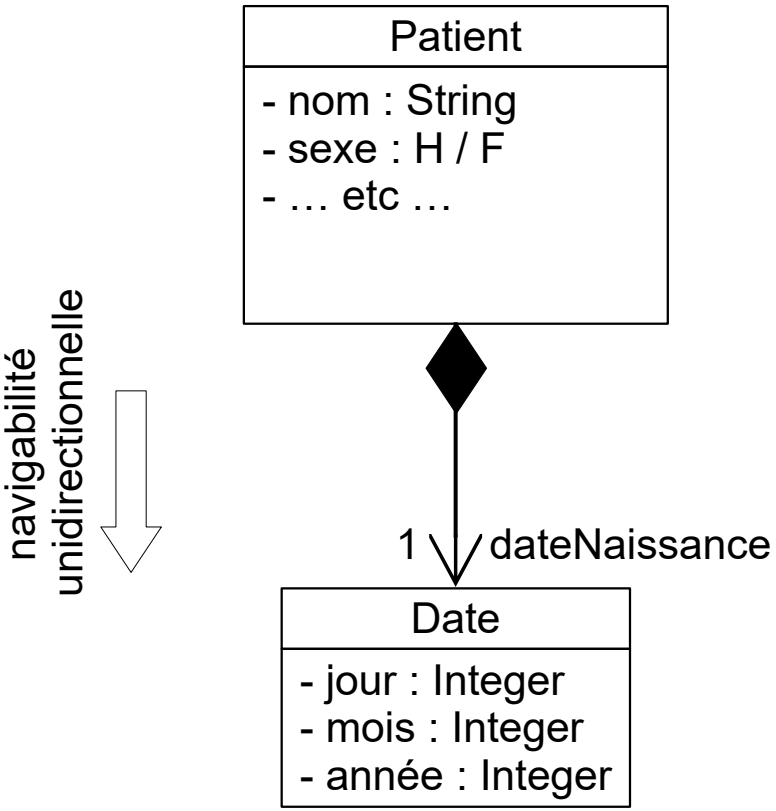


Diagramme de classes

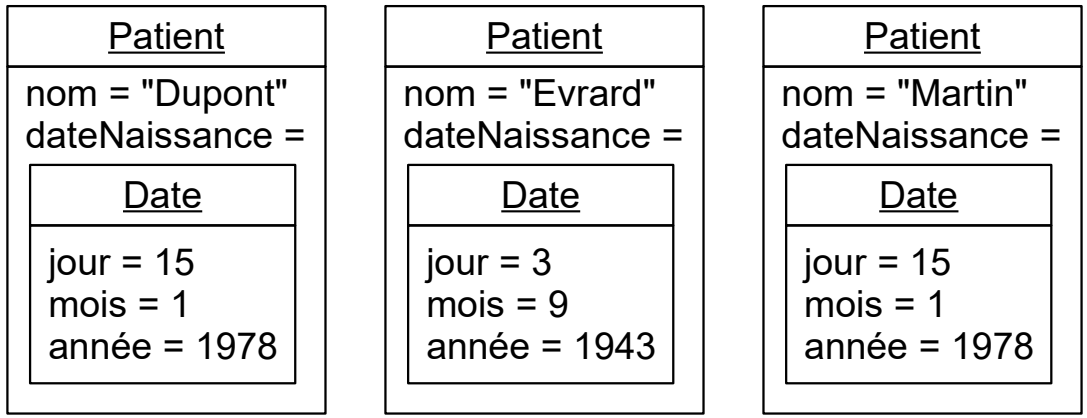
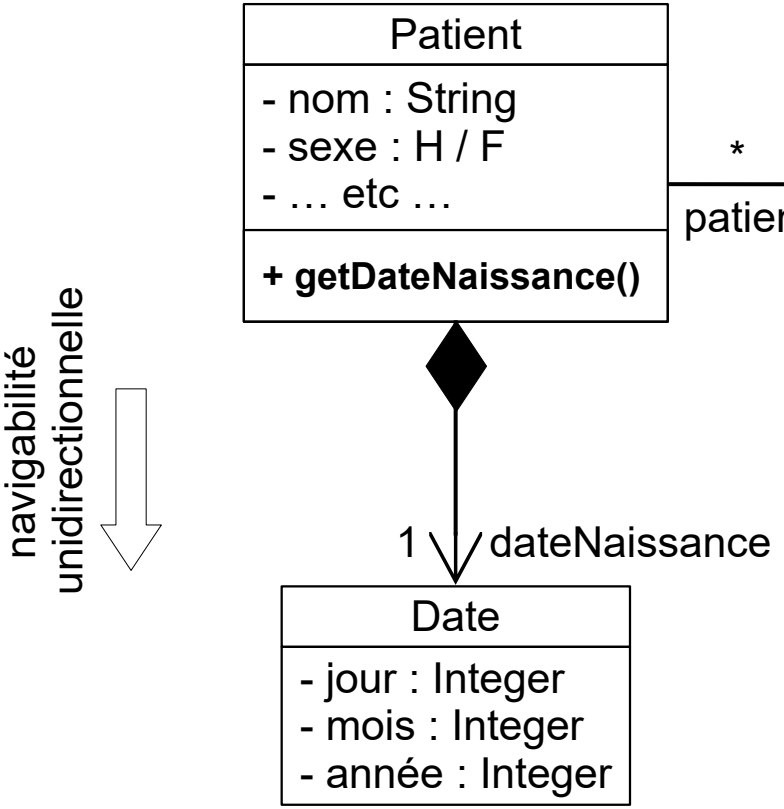


Diagramme d'objets

Types valeur / types entité

- *Ça n'interdit pas aux **méthodes** d'autres objets d'obtenir une **copie** (ou une référence temporaire) aux données : mais toujours en passant par l'objet propriétaire*

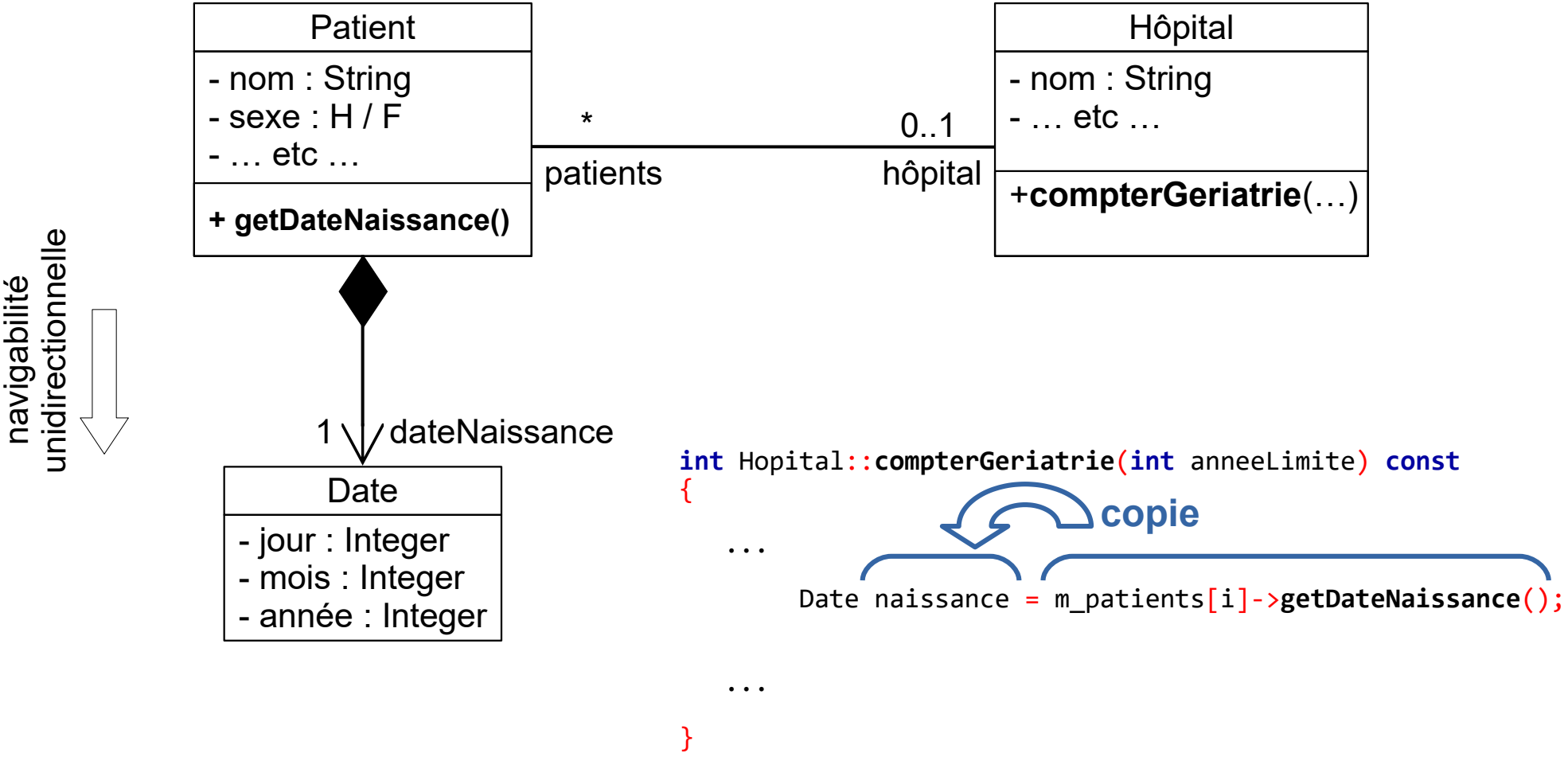



```
Date Patient::getDateNaissance() const
{
    return m_dateNaissance;
}
```

```
int Hopital::compterGeriatrie(int anneeLimite) const
{
    int cpt = 0;
    for (size_t i=0; i<m_patients.size();  ++i)
    {
        Date naissance = m_patients[i]->getDateNaissance();
        if ( naissance.getAnnee() <= anneeLimite )
            ++cpt;
    }
    return cpt;
}
```

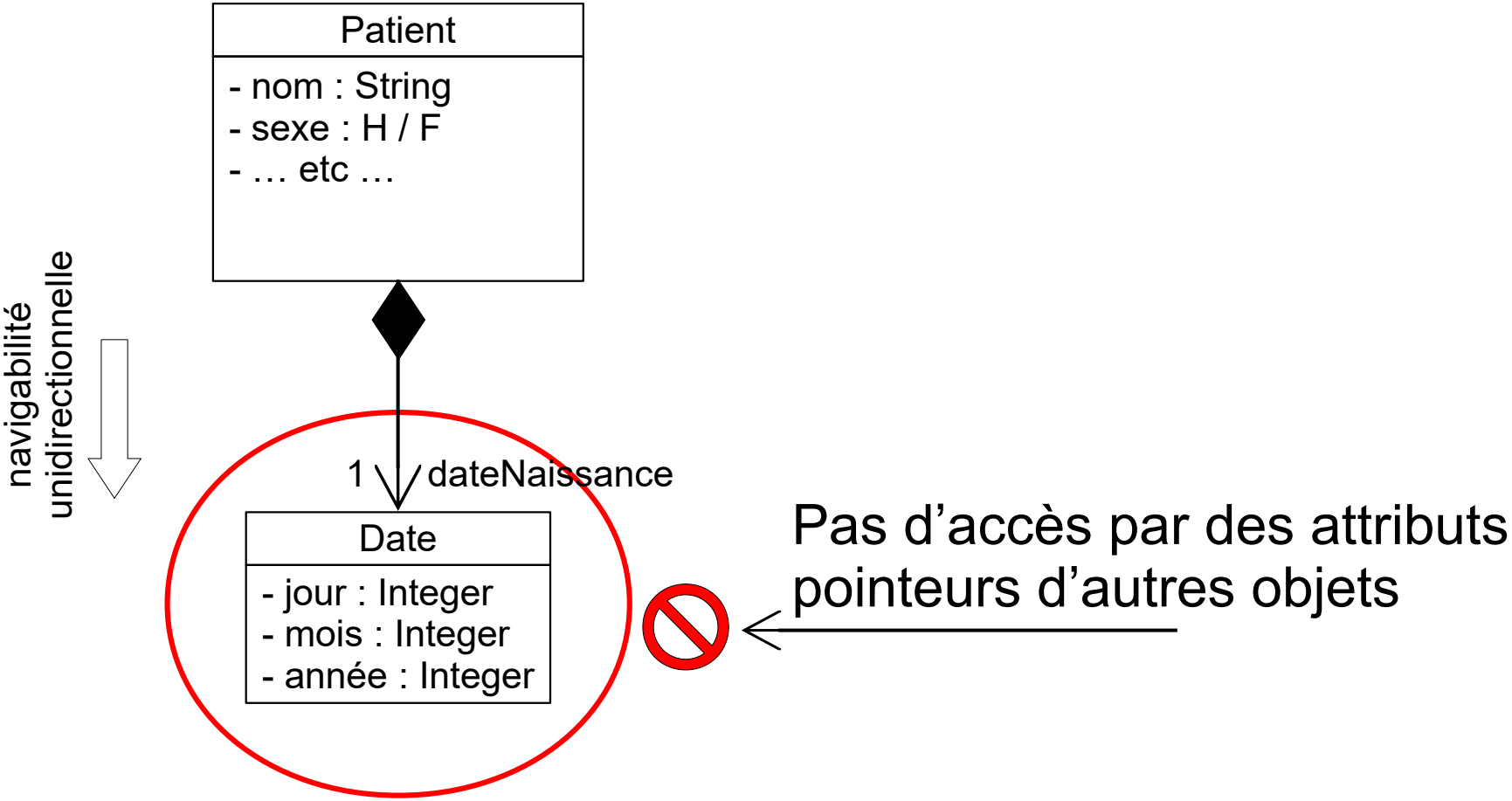
Types valeur / types entité

- *Ça n'interdit pas aux **méthodes** d'autres objets d'obtenir une **copie** (ou une référence temporaire) aux données : mais toujours en passant par l'objet propriétaire*



Types valeur / types entité

- *Les objets de ce genre de types se comportent **comme des valeurs***
  *On dira de ces types que ce sont des **types valeur***



Types valeur / types entité

- *On voudra « se débarrasser » de ces types dans les **diagrammes de classes** en les considérant directement comme des attributs*

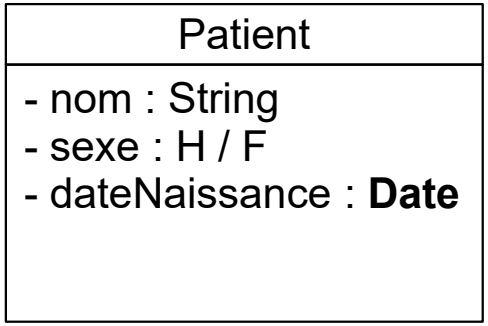
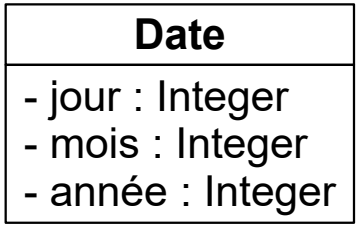


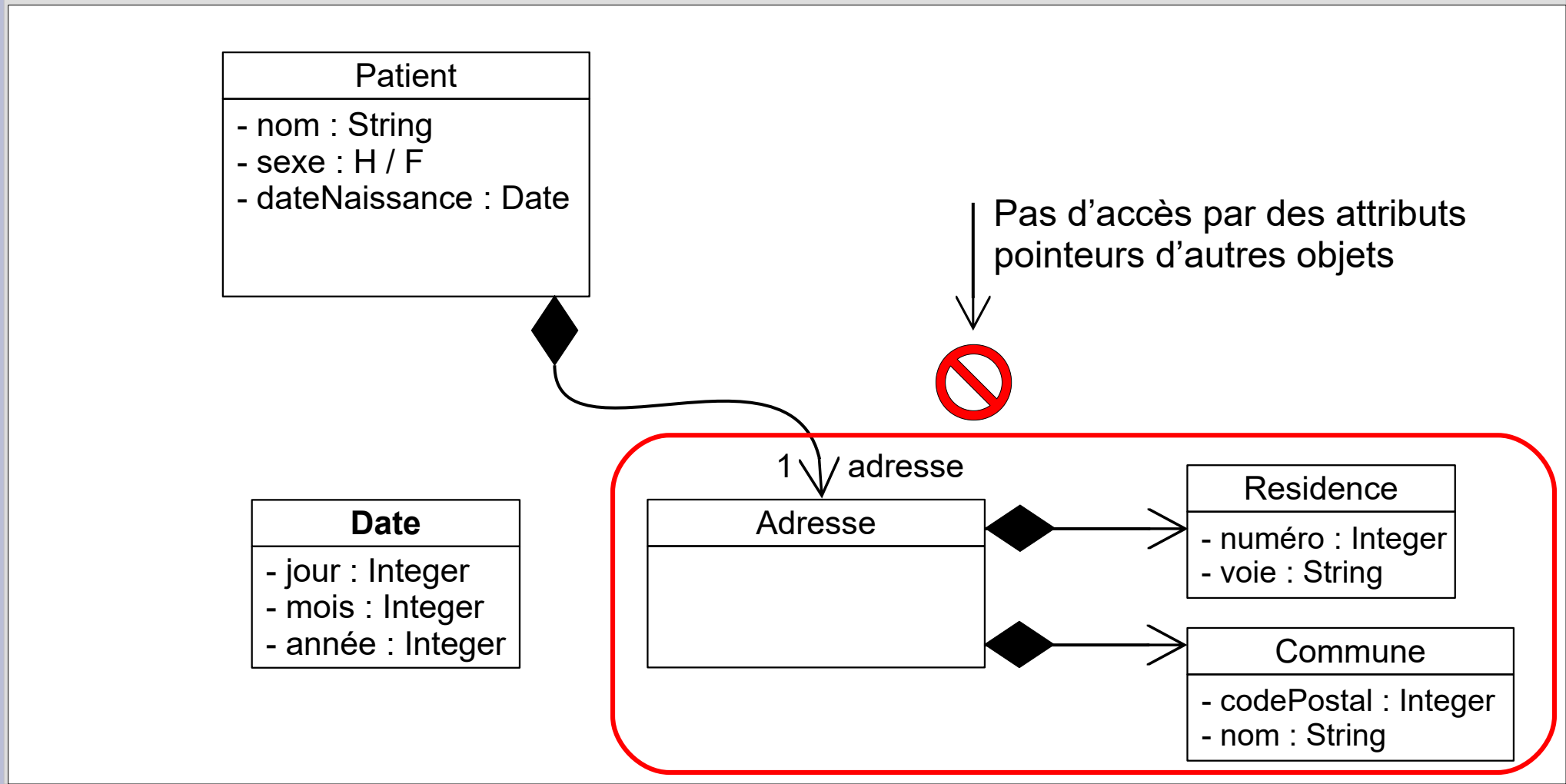
Diagramme de classes



Classes des types valeur : décrits séparément

Types valeur / types entité

- *Un type **composé** de **types valeurs**, sans pointeurs extérieurs sur lui, est lui même un **type valeur***



Types valeur / types entité

- *Un type **composé** de **types valeurs**, sans pointeurs extérieurs sur lui, est lui même un **type valeur***

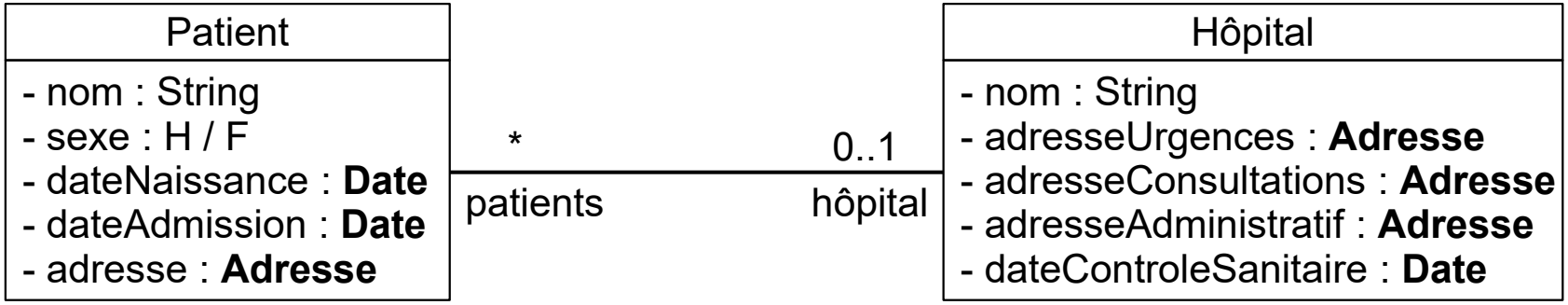
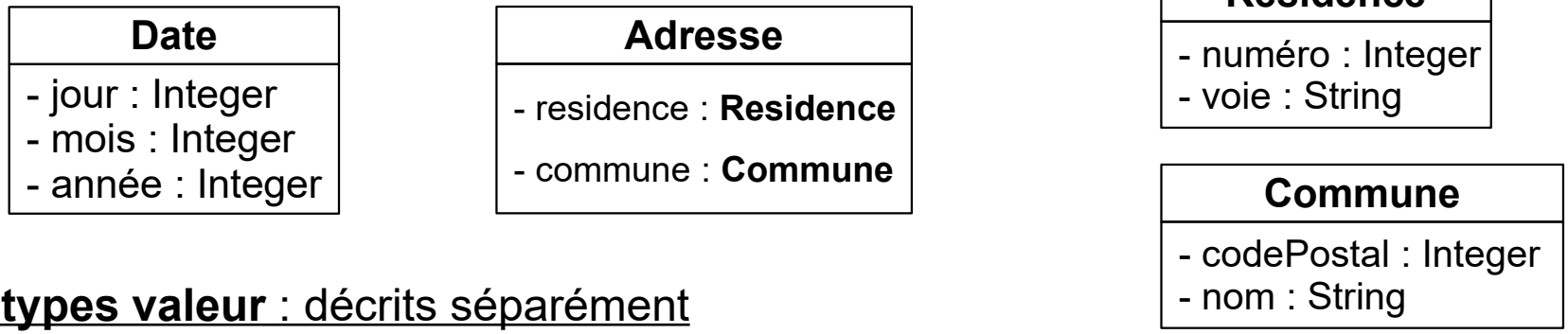


Diagramme de classes



Classes des **types valeur** : décrits séparément

Types valeur / types entité

- *Les classes qui ne sont pas des types valeur sont des **types entité** : les objets en inter-relations. Ils n'apparaissent jamais directement en attributs*

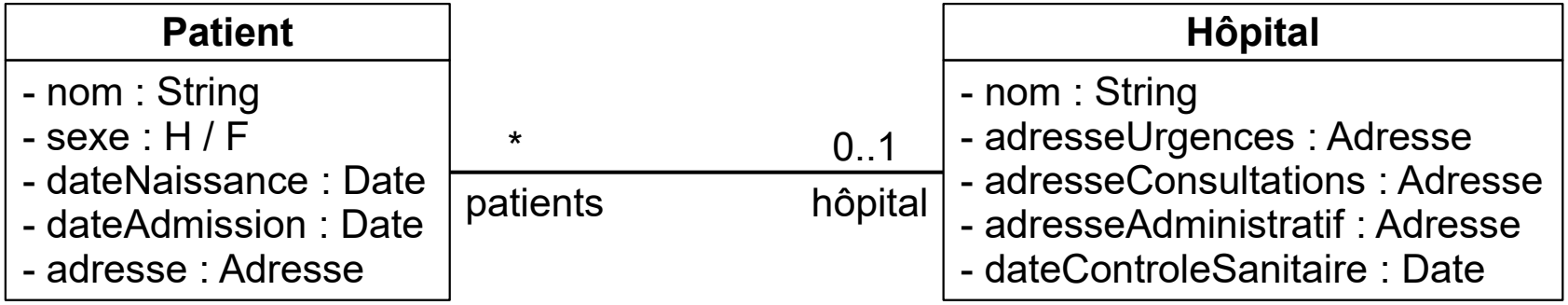
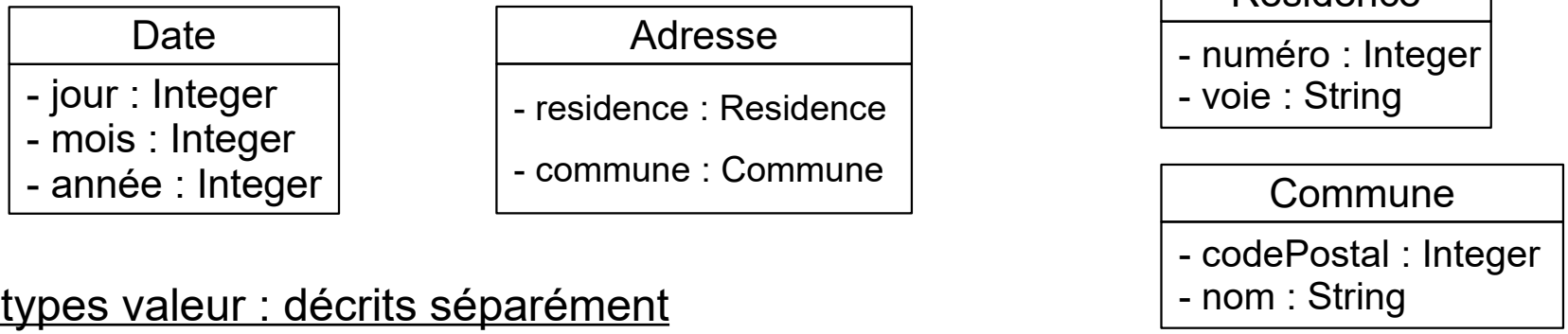


Diagramme de classes des **types entité**



Classes des types valeur : décrits séparément

Types valeur / types entité

*Les objets de **« type valeur »** n'ont pas d'identité*

- *Sont naturellement faciles à copier/détruire car :*
  - ils ne sont pas multiplement référencés
  ils ne sont pas la cible de nombreux pointeurs
  *- ce sont essentiellement des types **composés** de types valeurs élémentaires (int, float... pas de pointeurs)*
  ou d'autres types valeurs (hiérarchies de composition)
- *N'existent pas «indépendamment»*
  - soit ils ont une vie brève en tant que variable auto. / paramètre / objet anonyme temporaire
  - soit ils ont une vie longue mais en tant que valeur d'attribut d'un type entité, pas isolément

Types valeur / types entité

*Les objets de **« type entité »** ont une identité*

- *N'ont pas vocation à être copiés (unicité)*
  - *ce sont essentiellement des types **associés** (UML) à d'autres types entités*
  - *ils peuvent être multiplement référencés*
    ils sont la cible de nombreux pointeurs d'autres entités
  - *ils ont souvent des références sortantes*
    ils ciblent d'autres entités par pointeurs
- *Existent avec une adresse stable*
  - *ce sont des objets persistants : généralement alloués dynamiquement avec **new** et libérés avec **delete***
  - *on peut les mettre en correspondance directe avec des entités concrètes du modèle, 1 instance = 1 entité*

Types valeur / types entité

Distinguer les types valeur des types entité

- *Cette démarche tient une place importante dans une des méthodologies de conception orienté objet : Domain Driven Design*

  Compléments d'explications (Java/DB centrique)

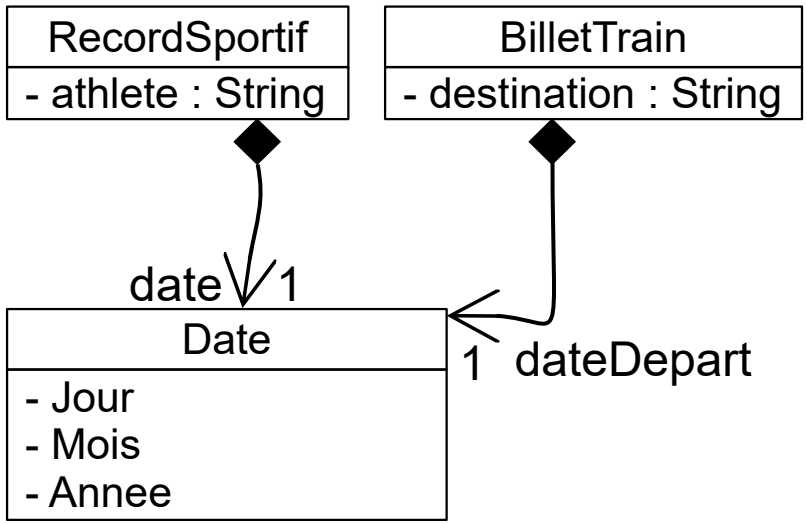
- *Attention aux raccourcis, une classe "légère et simple" n'est pas automatiquement un type valeur :*
  pour un gestionnaire de réseau (fibre optique…)
  le type Adresse est peut être un type entité !

- *A l'inverse un type "lourd ou complexe" comme Image (matrice de pixels) peut se comporter en valeur si les objets images sont des composants qui ne participent pas à la navigation dans le modèle objet*

Types valeur / types entité

- *Ceci impactera les diagrammes de classes en les débarrassant de nombreuses classes utilitaires qui se retrouvent alors comme types d'attributs, les associations graphiques sont donc réservées aux « vraies » classes d'entités*

Quelle est la relation ??



Pas terrible : Athlete et destination sont traités comme des valeurs et Date en entité !

Aucune relation !

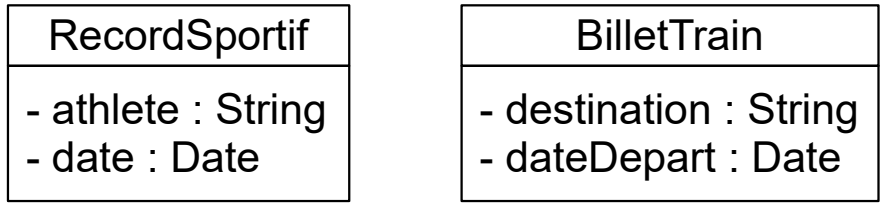


Diagramme de classes niveau entités de l'application

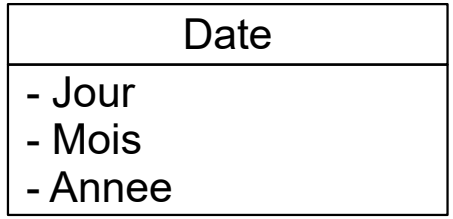
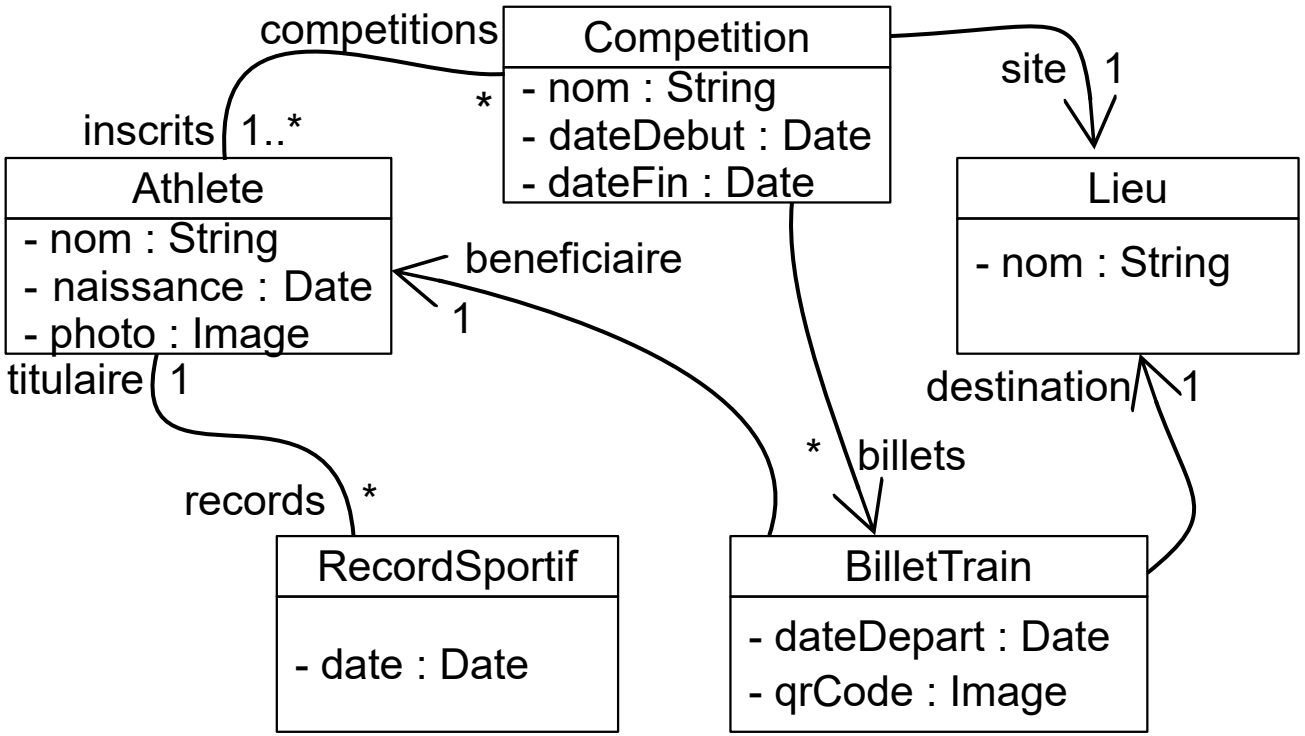


Diagramme de classes niveau bibliothèque : utilitaires / types valeur

Types valeur / types entité

- *Ceci impactera les diagrammes de classes en les débarrassant de nombreuses classes utilitaires qui se retrouvent alors comme types d'attributs, les associations graphiques sont donc réservées aux « vraies » classes d'entités*

Un organisateur de compétitions veut savoir à quels athlètes proposer des billets gratuits en fonction de leurs records...



Mieux : on voit bien le rôle réciproque des entités et Date n'encombre pas avec des liaisons graphiques !

Types valeur / types entité

- *Ceci impactera notre façon de concevoir le code*
- *C++ a une mécanique efficace de composition avec attributs membres **par valeur***
- *Les types valeurs seront codés pour être copiable facilement, efficacement, sûrement*
- *Dès qu'ils sont lourds on pourra quand même les passer par référence & en **paramètres***
- *Les types entités seront si possible déclarés non copiables !*
- *Les types entités seront manipulés par pointeurs*

COURS 6

A) Du modèle objet au C++
B) Types valeur / types entité
C) Copiabilité en C++
D) Composition en C++
E) Associations à sens unique
F) Associations à double sens

Copiabilité en C++

Copiabilité en C++

- *Le compilateur C++ fait des choses implicites*
- *Dès qu'on déclare une classe il existe une famille de **méthodes spéciales** qui sont traitées de façons spécifiques par le compilateur :*
  en l'absence de leur déclaration explicite
  elle sont générées implicitement

Function	syntax for class MyClass
Default constructor	`MyClass();`
Copy constructor	`MyClass(const MyClass& other);`
Move constructor	`MyClass(MyClass&& other) noexcept;`
Copy assignment operator	`MyClass& operator=(const MyClass& other);`
Move assignment operator	`MyClass& operator=(MyClass&& other) noexcept;`
Destructor	`~MyClass() noexcept;`

Copiabilité en C++

- *Les méthodes spéciales **implicites** sont neutralisées quand on les déclare*
- *Dans certains cas on ne veut pas de ces méthodes implicite, sans pour autant les définir*
  On l'indiquera avec **`=delete;`**
  derrière le prototype de la méthode

Function	syntax for class MyClass
Default constructor	`MyClass();`
Copy constructor	`MyClass(const MyClass& other);`
Move constructor	`MyClass(MyClass&& other) noexcept;`
Copy assignment operator	`MyClass& operator=(const MyClass& other);`
Move assignment operator	`MyClass& operator=(MyClass&& other) noexcept;`
Destructor	`~MyClass() noexcept;`

Copiabilité en C++

- *Dans certains cas on veut au contraire confirmer **explicitement** au compilateur qu'on veut ces méthodes **implicites***
  On l'indiquera avec **`=default;`**
  derrière le prototype de la méthode
  (Vous me suivez ? Les règles sont complexes...)

Function	syntax for class MyClass
Default constructor	`MyClass();`
Copy constructor	`MyClass(const MyClass& other);`
Move constructor	`MyClass(MyClass&& other) noexcept;`
Copy assignment operator	`MyClass& operator=(const MyClass& other);`
Move assignment operator	`MyClass& operator=(MyClass&& other) noexcept;`
Destructor	`~MyClass() noexcept;`

Copiabilité en C++

- *Le constructeur par défaut implicite est en général **insatisfaisant** (Cours 5)*
- *Le destructeur implicite est en général **satisfaisant** si l'objet n'a pas acquis de ressources à gestion manuelle (pas fait* new *) et qu'on a rien d'intéressant à y faire*

Function	syntax for class MyClass
Default constructor	`MyClass();`
Copy constructor	`MyClass(const MyClass& other);`
Move constructor	`MyClass(MyClass&& other) noexcept;`
Copy assignment operator	`MyClass& operator=(const MyClass& other);`
Move assignment operator	`MyClass& operator=(MyClass&& other) noexcept;`
Destructor	`~MyClass() noexcept;`

Copiabilité en C++

- *Les méthodes de déplacement (move semantics) sont une innovation du C++ 11*
  Indispensable sur du code C++ « up to date »
- On utilise quelques aspects C++ 11 mais on ne pourra pas tout couvrir… hors programme

Function	syntax for class MyClass
Default constructor	`MyClass();`
Copy constructor	`MyClass(const MyClass& other);`
Move constructor	`MyClass(MyClass&& other) noexcept;`
Copy assignment operator	`MyClass& operator=(const MyClass& other);`
Move assignment operator	`MyClass& operator=(MyClass&& other) noexcept;`
Destructor	`~MyClass() noexcept;`

Copiabilité en C++

- *Les méthodes de copie sont celles qui vont nous intéresser en particulier pour les types valeurs qui doivent être copiables !*
- On verra comment les définir en cas de besoin mais surtout comment ne pas avoir ce besoin !

Function	syntax for class MyClass
Default constructor	`MyClass();`
Copy constructor	`MyClass(const MyClass& other);`
Move constructor	`MyClass(MyClass&& other) noexcept;`
Copy assignment operator	`MyClass& operator=(const MyClass& other);`
Move assignment operator	`MyClass& operator=(MyClass&& other) noexcept;`
Destructor	`~MyClass() noexcept;`

Copiabilité en C++

- *Les méthodes de copie sont utilisées quand...*
- *Un nouvel objet est **créé** à partir d'un autre*
  => Constructeur par copie
- *Un objet existant prend la valeur d'un autre*
  => Opérateur d'affectation = (par copie)

Function	syntax for class MyClass
Default constructor	`MyClass();`
Copy constructor	`MyClass(const MyClass& other);`
Move constructor	`MyClass(MyClass&& other) noexcept;`
Copy assignment operator	`MyClass& operator=(const MyClass& other);`
Move assignment operator	`MyClass& operator=(MyClass&& other) noexcept;`
Destructor	`~MyClass() noexcept;`

Copiabilité en C++

- *Les méthodes de copie sont utilisées quand...*
- *Un nouvel objet est **créé** à partir d'un autre*
  => Constructeur par copie
- *Un objet existant prend la valeur d'un autre*
  => Opérateur d'affectation = (par copie)

```
Oeuvre inestimable{"La Joconde", "Léonard de Vinci", 1519};

/// Appel au constructeur par copie
Oeuvre vulgaireCopie1{inestimable};
Oeuvre vulgaireCopie2 = inestimable;

/// Déclaration puis appel à l'opérateur d'affectation
Oeuvre vulgaireCopie3{"", "", 0};
vulgaireCopie3 = inestimable;
```

Copiabilité en C++

- *Les méthodes de copie sont utilisées quand...*
- *Un passage de paramètre par valeur est présent => Constructeur par copie*

```
void fonctionA(Oeuvre oeuvre)
{ ... }
void fonctionB(Oeuvre& oeuvre)
{ ... }
void fonctionC(Oeuvre* poeuvre)
{ ... }
void Oeuvre::uneMethode()
{ ... }
```

```
    Oeuvre inestimable{"La Joconde", "Léonard de Vinci", 1519};

    /// Appel au constructeur par copie
    fonctionA(inestimable);

    /// Aucune copie
    fonctionB(inestimable);
    fonctionC(&inestimable);
    inestimable.uneMethode()
```

Copiabilité en C++

- *Les méthodes de copie sont utilisées quand...*
- *Un passage de paramètre par valeur est présent => Constructeur par copie*
- *Il ne faut donc pas passer par valeur des objets de classes non copiables !*
- *C'est le cas de Svgfile (TD/TP 4 et 5)*
- *Type entité : privilégier passage par référence !*

```
void fonction(Oeuvre oeuvre)
{ ... }
```

Nécessite Oeuvre copiable

```
void fonction(Oeuvre& oeuvre)
{ ... }
```

Aucune contrainte

Copiabilité en C++

- *Le constructeur par copie et l'opérateur d'affectation implicite réalisent une copie* ***membre à membre*** *(memberwise) en appelant si nécessaire les méthodes de copie des attributs de types copiables (std::string std::vector et tous nos types valeur)*
- *Ceci est le comportement souhaitable pour les types valeurs et composites de types valeurs*
- *Ceci n'est pas le comportement souhaitable pour les types qui gèrent des ressources*
- *Ceci n'est souvent pas souhaitable pour les types entités qu'il vaut mieux ne pas copier*

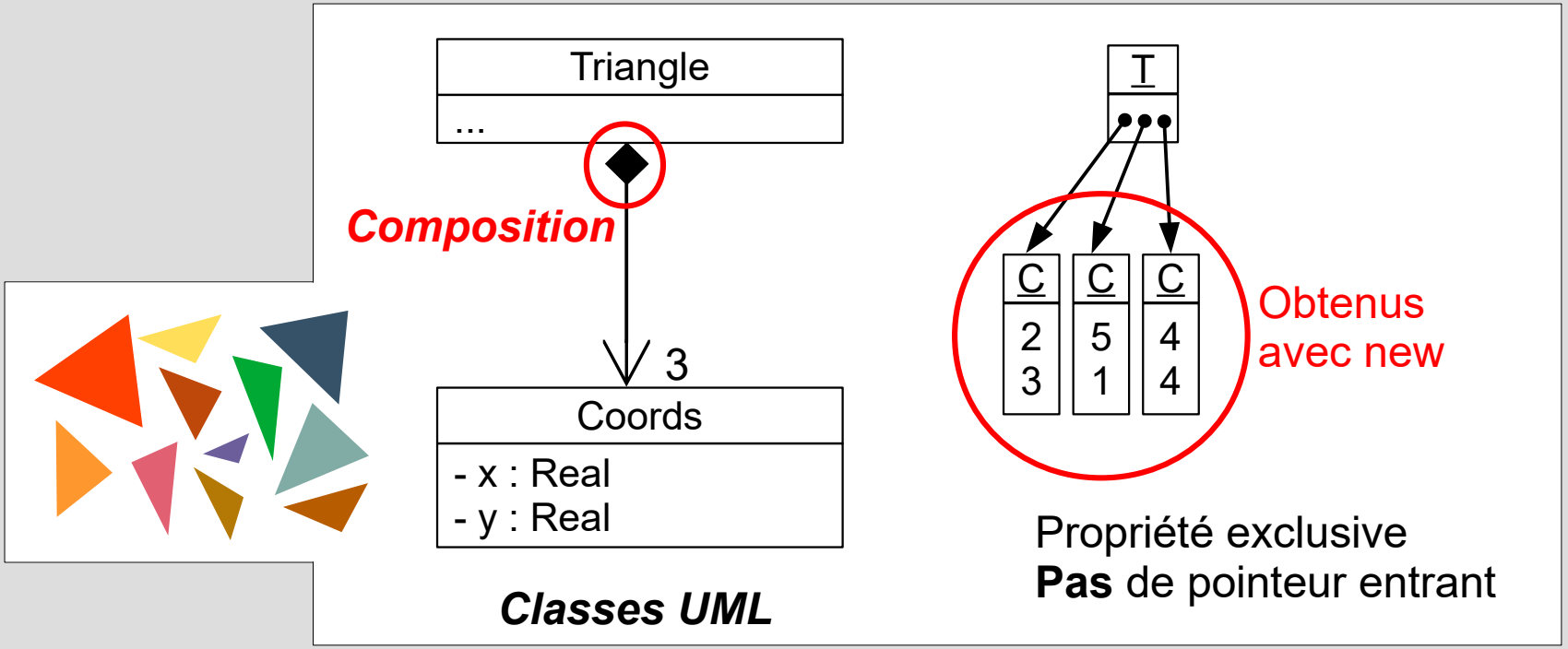
Copiabilité en C++

- *La **règle de 3** rule of three*
- *Concerne les classes gestionnaires de ressource (ce qu'on évitera de faire !)*
- *Elle dit que si UNE des 3 méthodes spéciales*
  - destructeur
  - constructeur par copie
  - opérateur d'affectation

  ne convient pas dans sa version implicite et doit être codée, alors les 3 doivent être codées

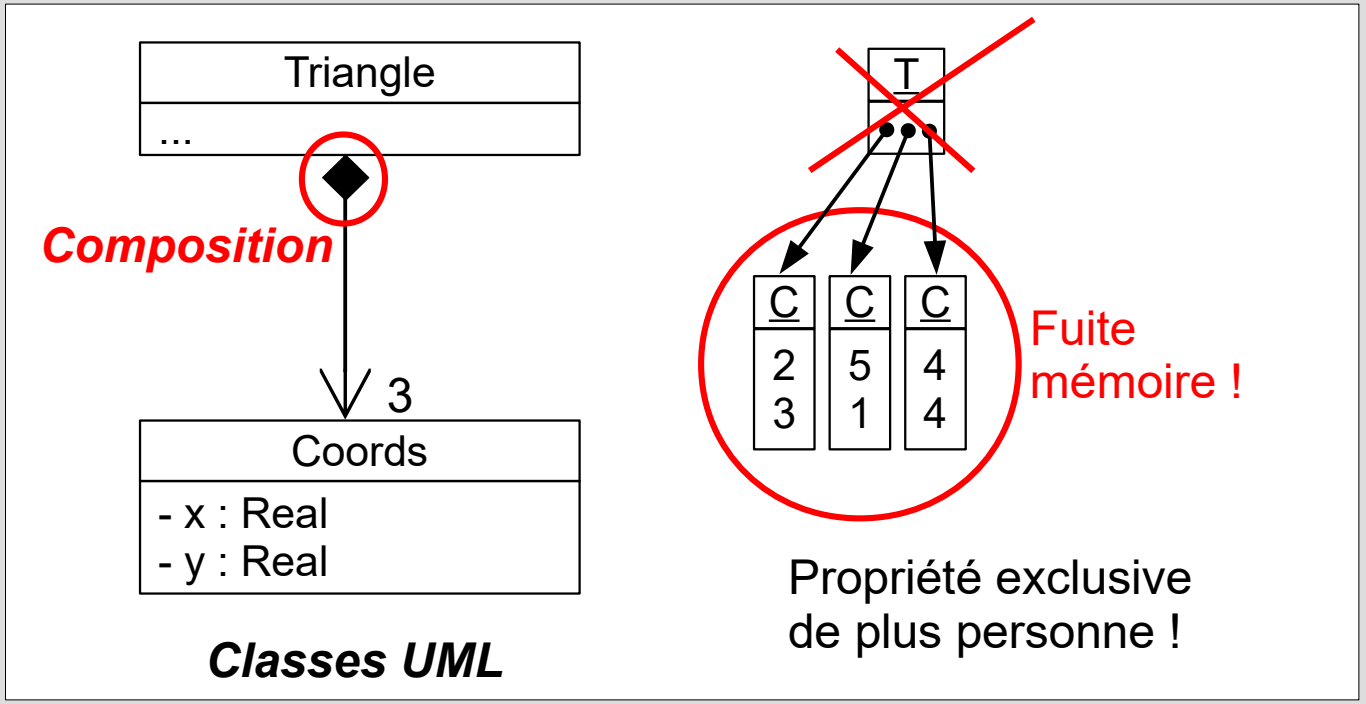
Copiabilité en C++

- *La **règle de 3** rule of three*
- *Essayons de comprendre pourquoi, supposons une classe Triangle qui **compose** 3 Coords **et qui les gère comme une ressource***
  (3 appels à new Coords dans le constructeur de Triangle)



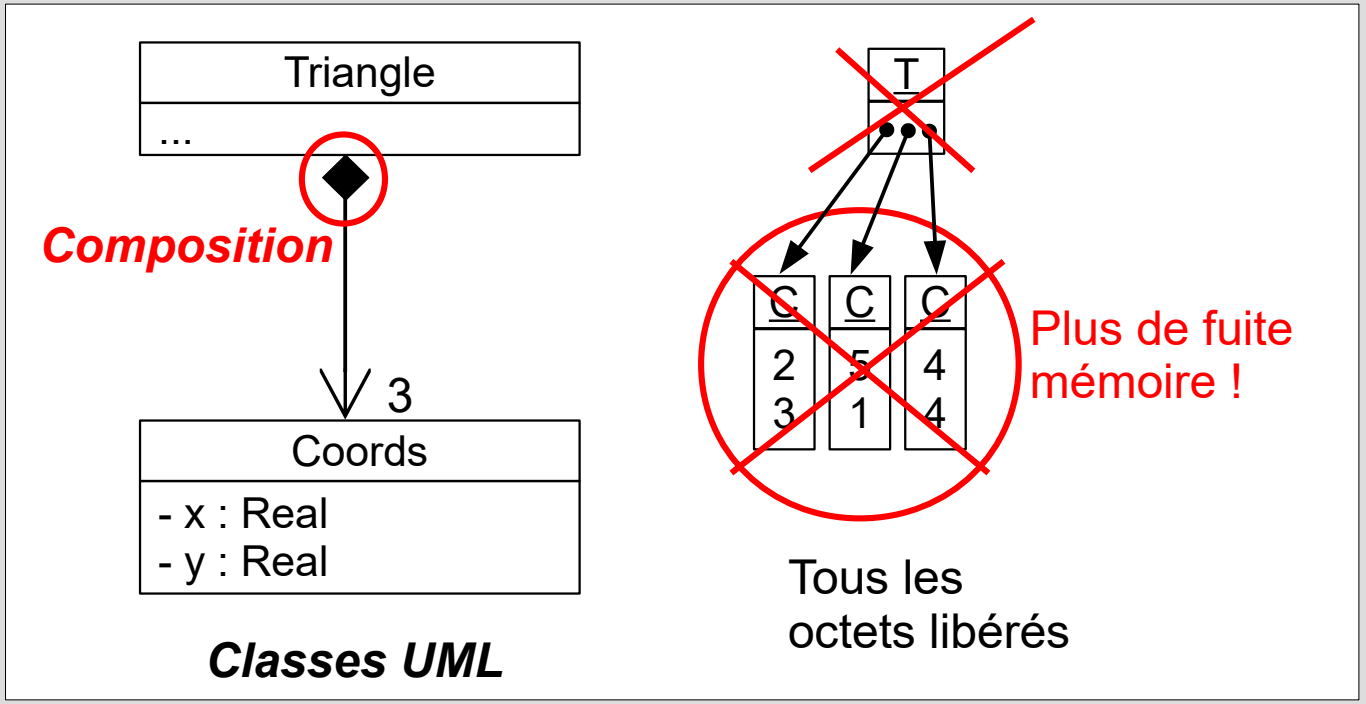
Copiabilité en C++

- *La **règle de 3** rule of three*
- *Au moment de la destruction du Triangle le destructeur implicite libère bien l'espace mémoire des 3 pointeurs mais pas des 3 Coords pointés !*



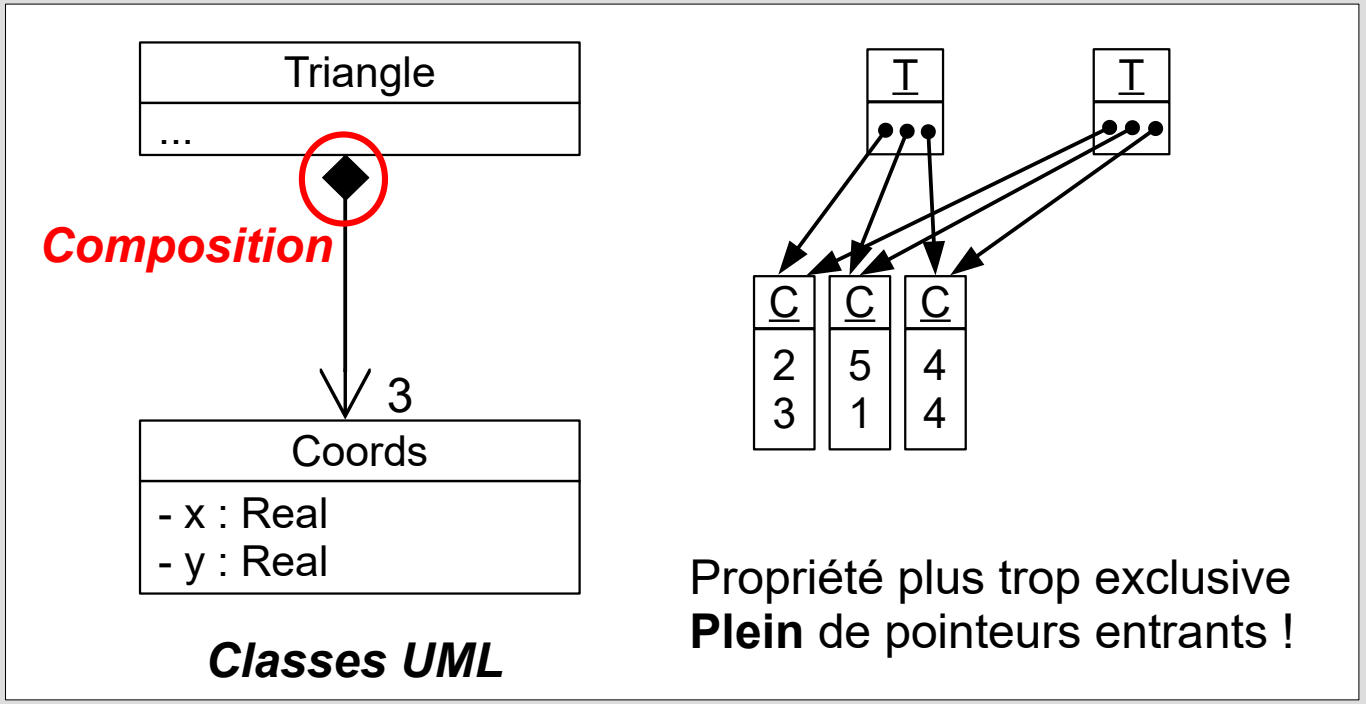
Copiabilité en C++

- *La **règle de 3** rule of three*
- *Il faut donc un destructeur explicite qui libère avec delete ce qui a été alloué avec new*



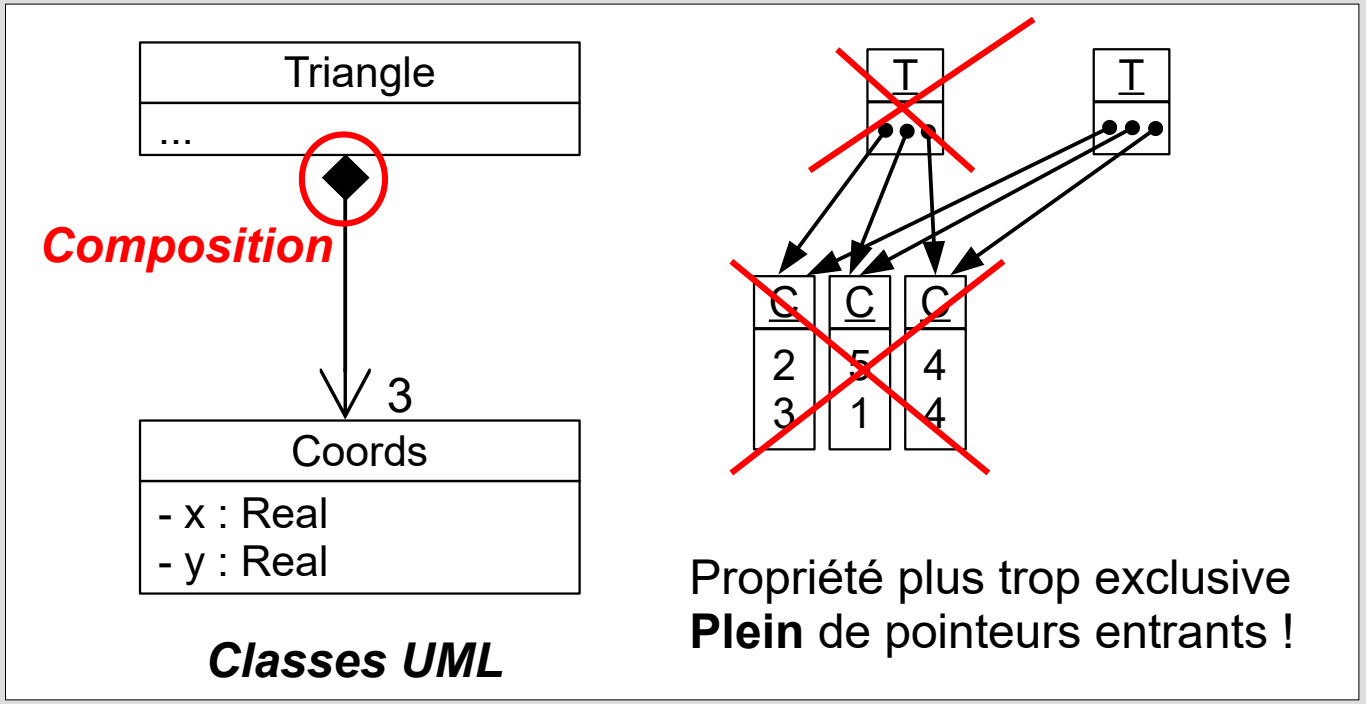
Copiabilité en C++

- *La **règle de 3** rule of three*
- *Mais dans ce cas la copie membre à membre implicite va copier des pointeurs et non les ressources pointées : **copie superficielle***
- *Ça ne convient pas pour une ressource !*



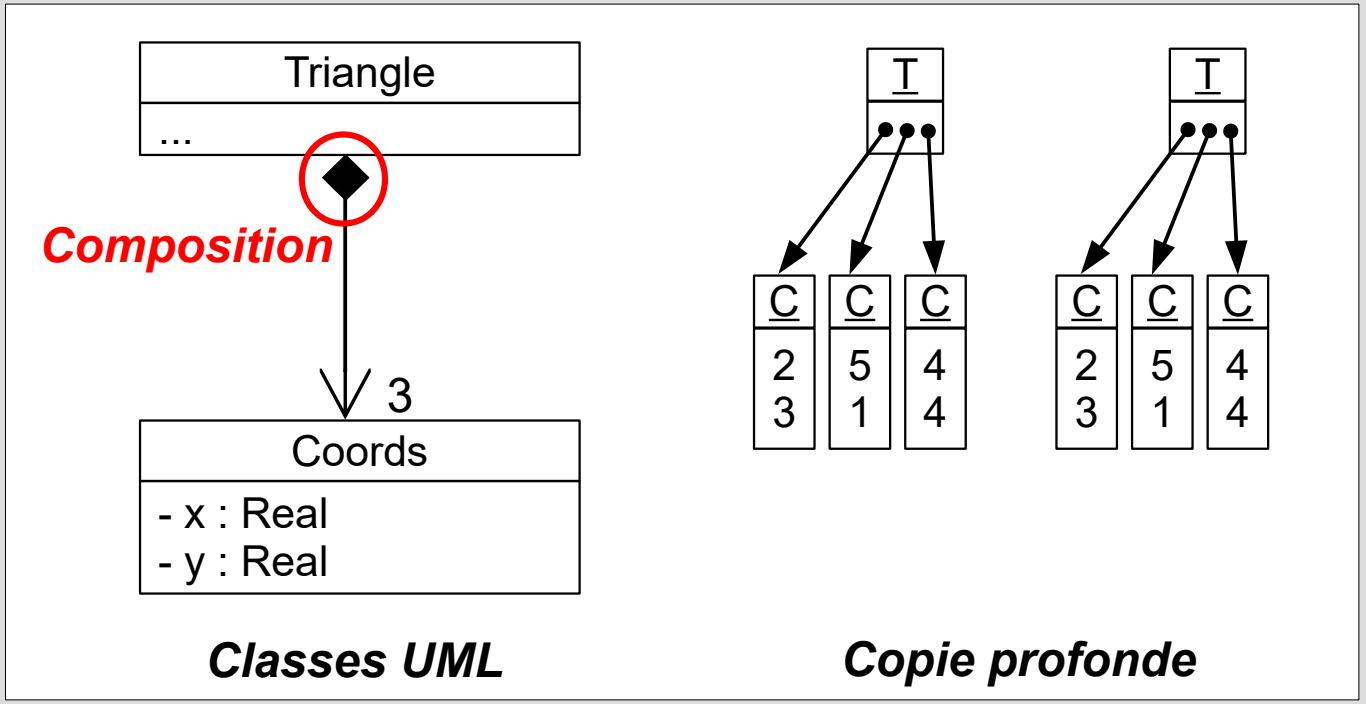
Copiabilité en C++

- *La **règle de 3** rule of three*
- *Non seulement des données exclusives sont partagées par 2 objet mais la destruction de l'un entraîne l'invalidation des pointeurs de l'autre (un chaos !)*



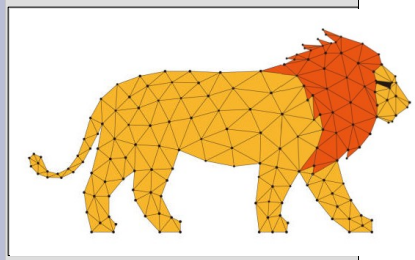
Copiabilité en C++

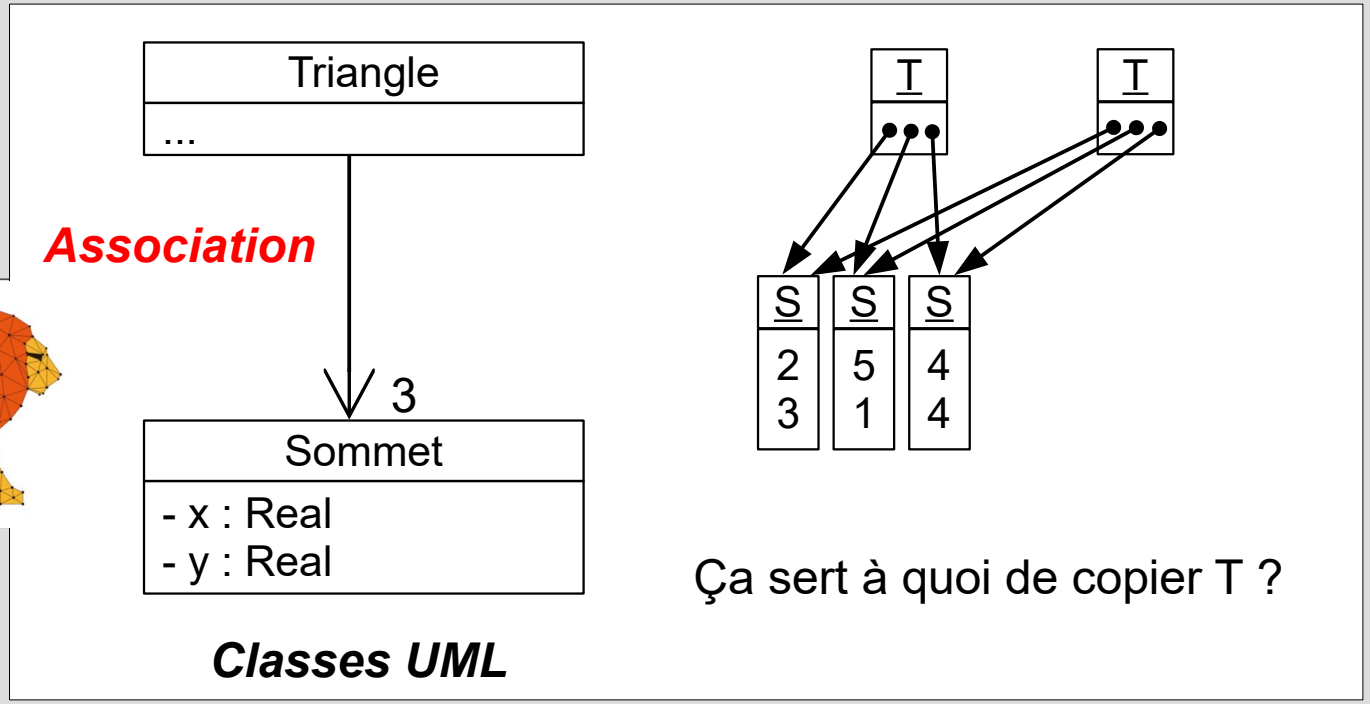
- *La **règle de 3** rule of three*
- *Donc la règle de 3 est nécessaire, il faut **coder** les 2 méthodes de copie pour obtenir une **copie profonde** (on ne voit pas ça en cours)*



Copiabilité en C++

- *Et pour une classe de type entité ?*
- *La copie membre à membre implicite fait une copie superficielle...*
- *Souvent ça ne convient pas pour une entité !*

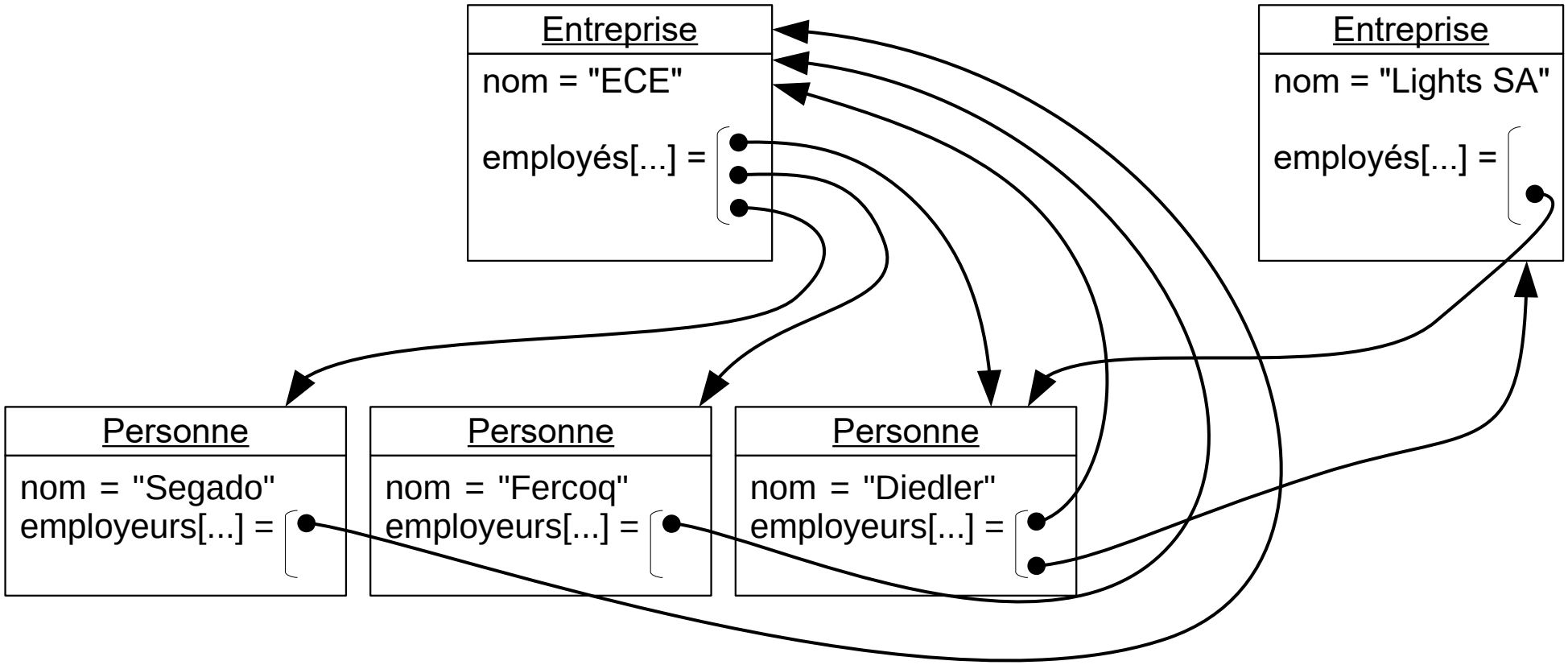




Copiabilité en C++

- *Au niveau entité tous les objets sont reliés dans un **graphe connexe** de relations*

Copier M. Fercoq ?



Copiabilité en C++

- *Au niveau entité tous les objets sont reliés dans un **graphe connexe** de relations*
- *La copie superficielle ne veut rien dire et la copie profonde est « impossible » : il faudrait copier tout le système !*
- *On préféra donc souvent bloquer toute tentative de copie en neutralisant les méthodes de copie implicites et en évitant d'essayer de les implémenter (sauf si le CDC nous y oblige)*

```
/// Pas de constructeur par copie implicite
Oeuvre(const Oeuvre& original) = delete;

/// Pas d'opérateur d'affectation implicite
Oeuvre& operator=(const Oeuvre& original) = delete;
```

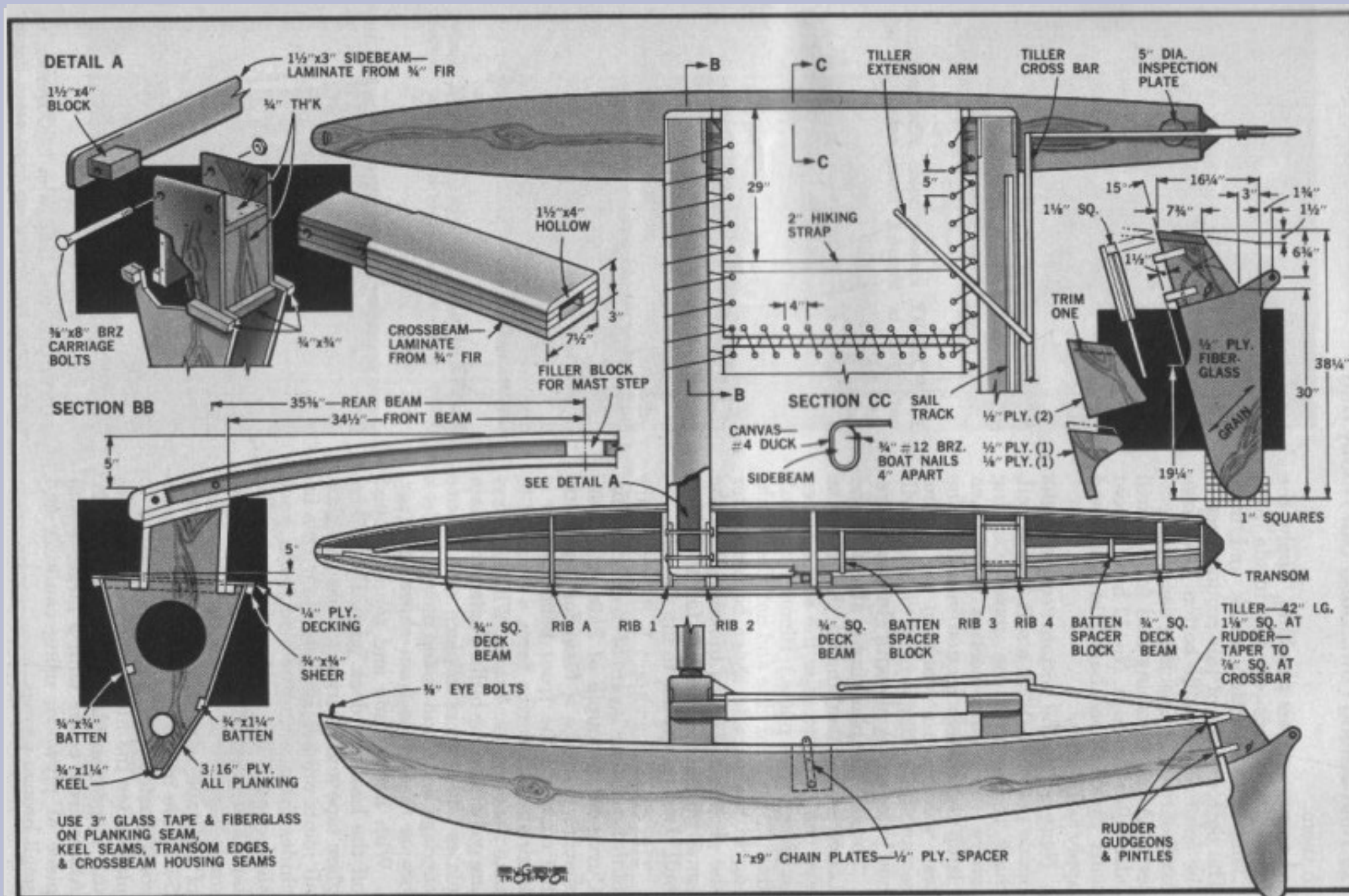
Copiabilité en C++

- *En résumé :*
- *<u>Classes de types valeur</u> :*
  les méthodes de copie implicites conviennent
- *<u>Classes gestionnaire de ressource</u> :*
  à éviter autant que possible
  la régle de 3 s'applique
  coder destructeur et 2 méthodes de copie
- *<u>Classes de types entité</u>*
  coder le destructeur si nécessaire
  => mise à jour pointeurs réciproques
  si possible mettre en =delete les méthodes de copie, la classe est non copiable

COURS 6

A) Du modèle objet au C++
B) Types valeur / types entité
C) Copiabilité en C++
D) Composition en C++
E) Associations à sens unique
F) Associations à double sens

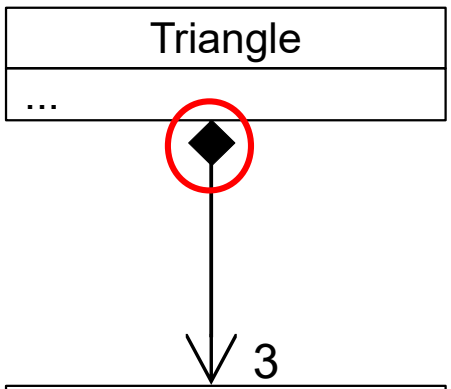
Composition en C++



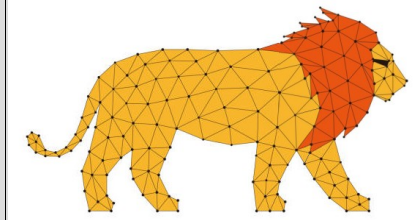
Composition en C++

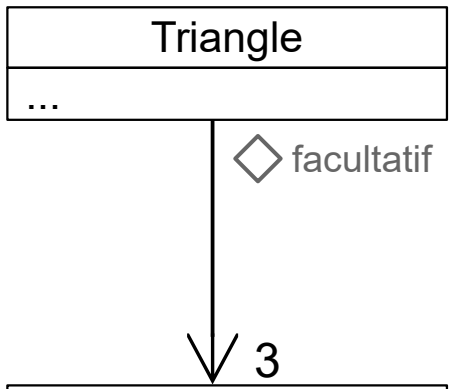
- *Dans les slides suivants on considère qu'un triangle est **composé** de 3 couples de Coords*
- *Attention ce n'est **pas** le modèle maillage (TPs) où les Sommets sont **partagés** : **association***

Composition



Association ou agrégation

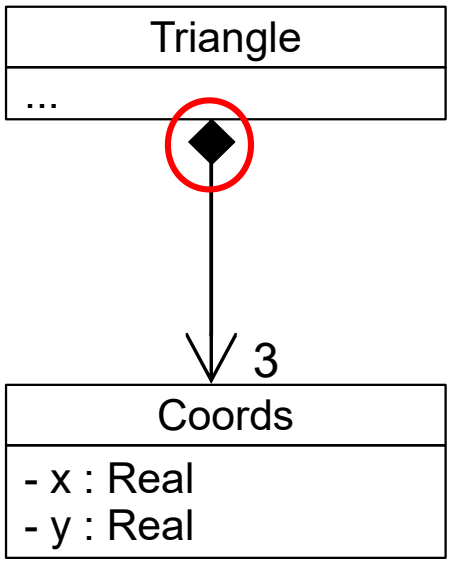




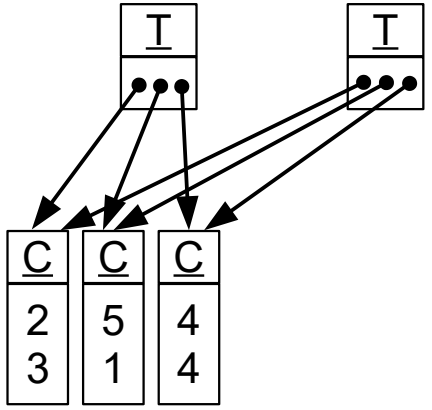
Composition en C++

- *Avec des pointeurs pour référencer des objets composants on a **par défaut** des copies de pointeurs sans copie des pointés*

Composition



Classes UML



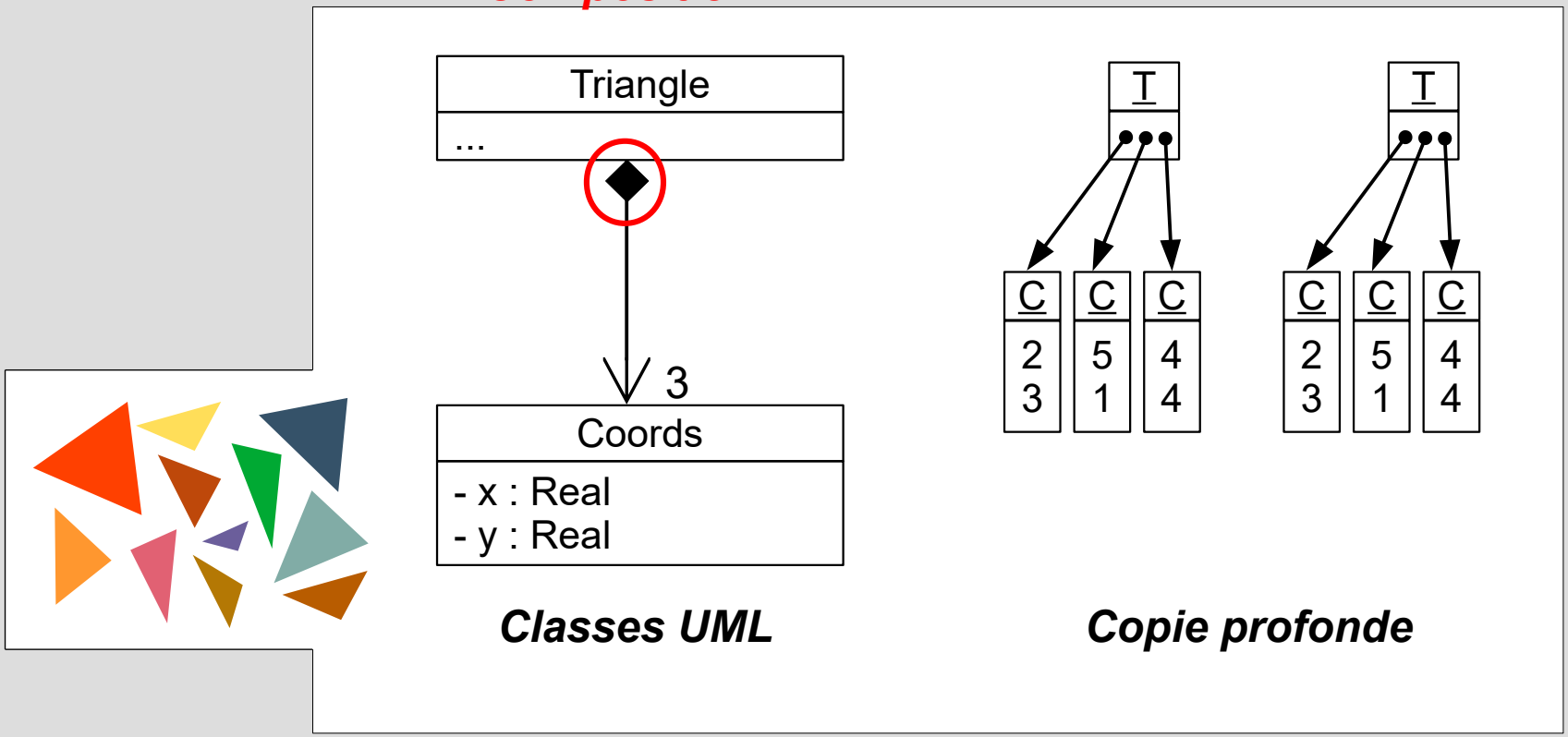
Copie superficielle

Ne convient pas du tout...

Composition en C++

- *Avec des pointeurs pour référencer des objets composants on peut **coder** pour obtenir une copie profonde avec copie des pointés*
- *On peut mais on évite si possible (règle de 3...)*

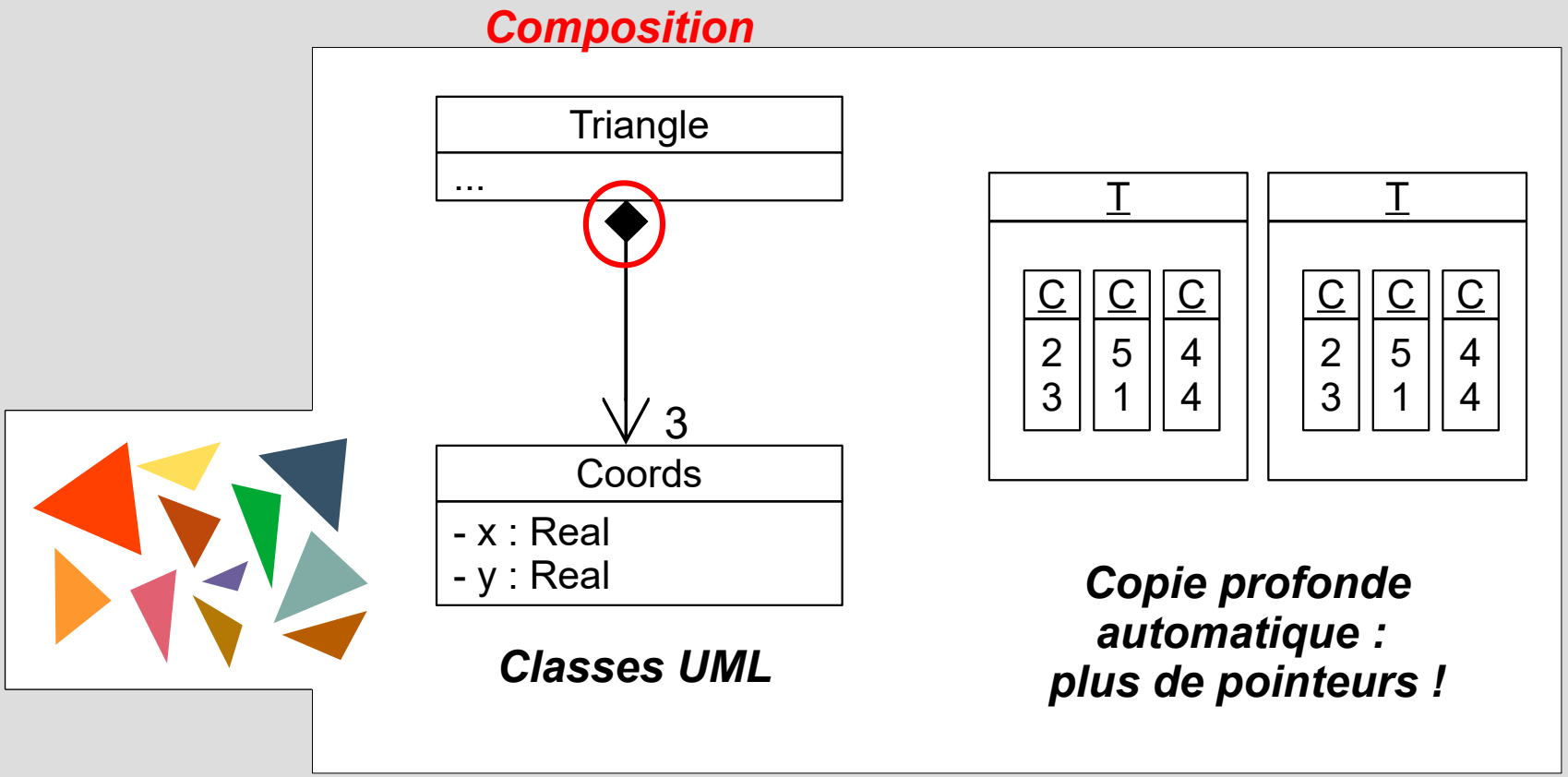
Composition



Classes UML ***Copie profonde***

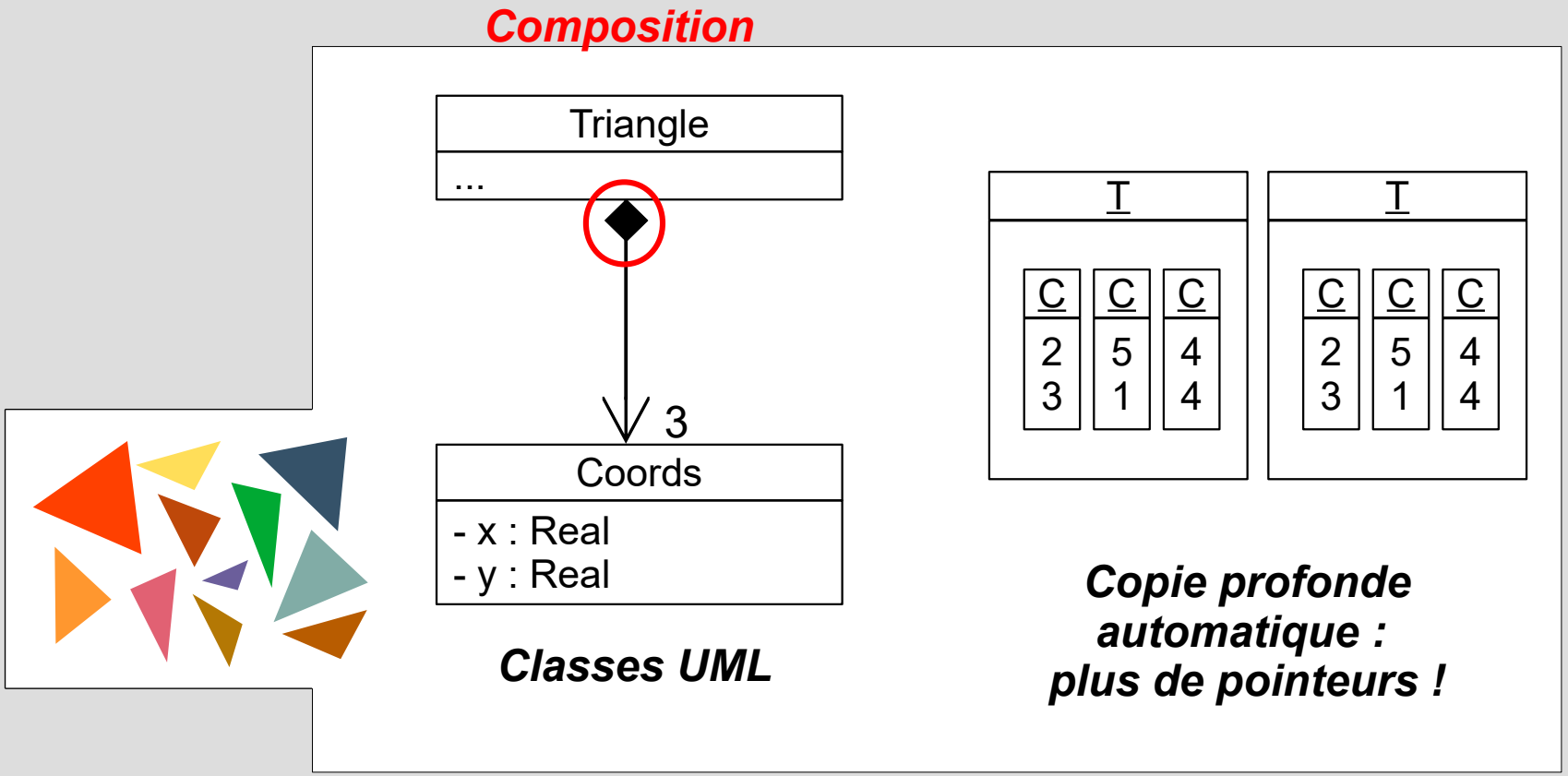
Composition en C++

- *Avec la sémantique par valeur (attributs valeurs) les données des classes composantes sont directement **dans** le même bloc mémoire que les autres attributs de la classe composite*



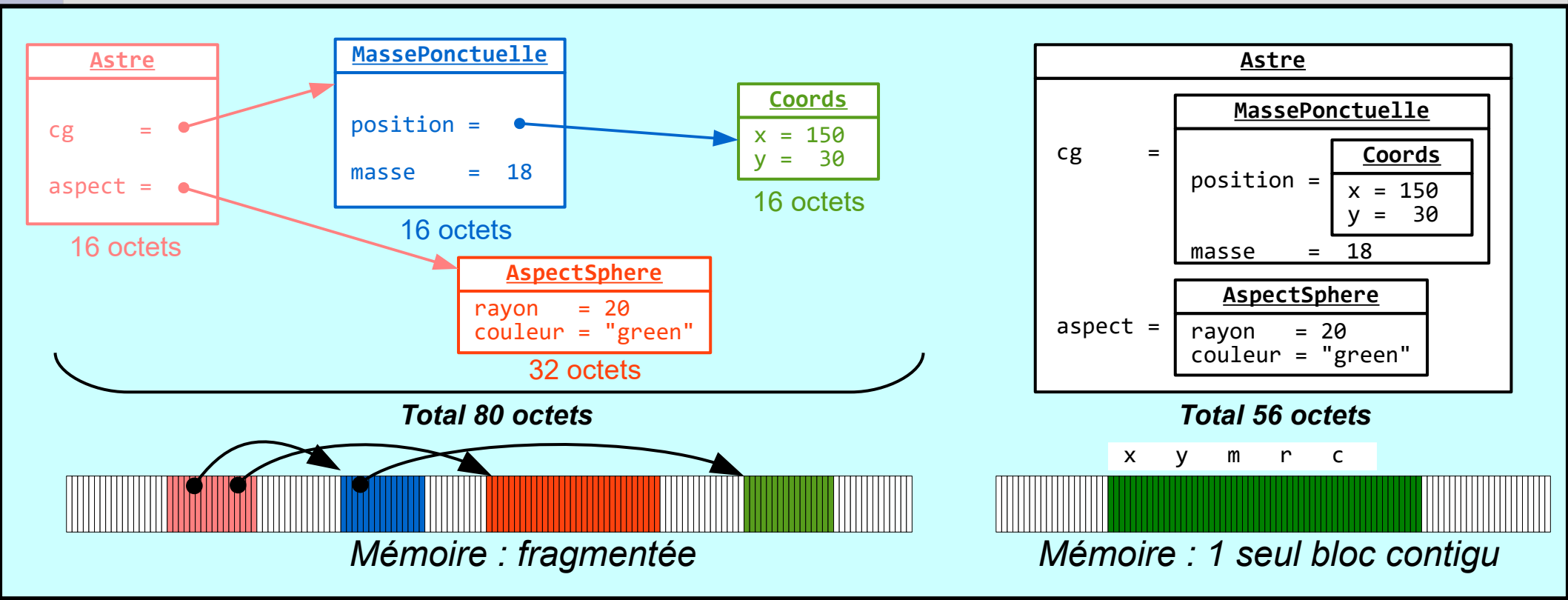
Composition en C++

- *Cette distinction copie profonde/superficielle existe dans tous les langages orientés objet*
  Seuls C++ et C# (pas Java) proposent une vraie sémantique par valeur (attributs valeurs)



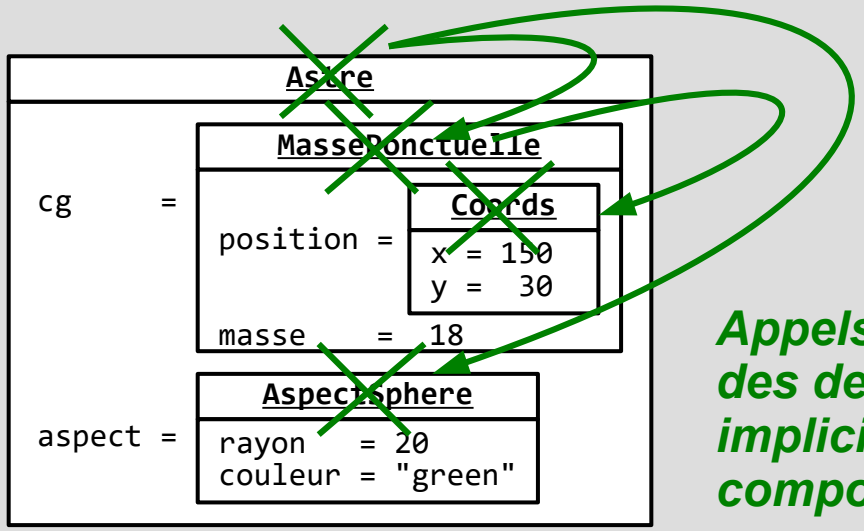
Composition en C++

- *C++ **excelle** dans les compositions par valeur*
- *Et le **hardware** adore un bloc d'octets contigus*
- *En C++ la composition par valeur est donc le choix qui s'impose **quand c'est possible***



Composition en C++

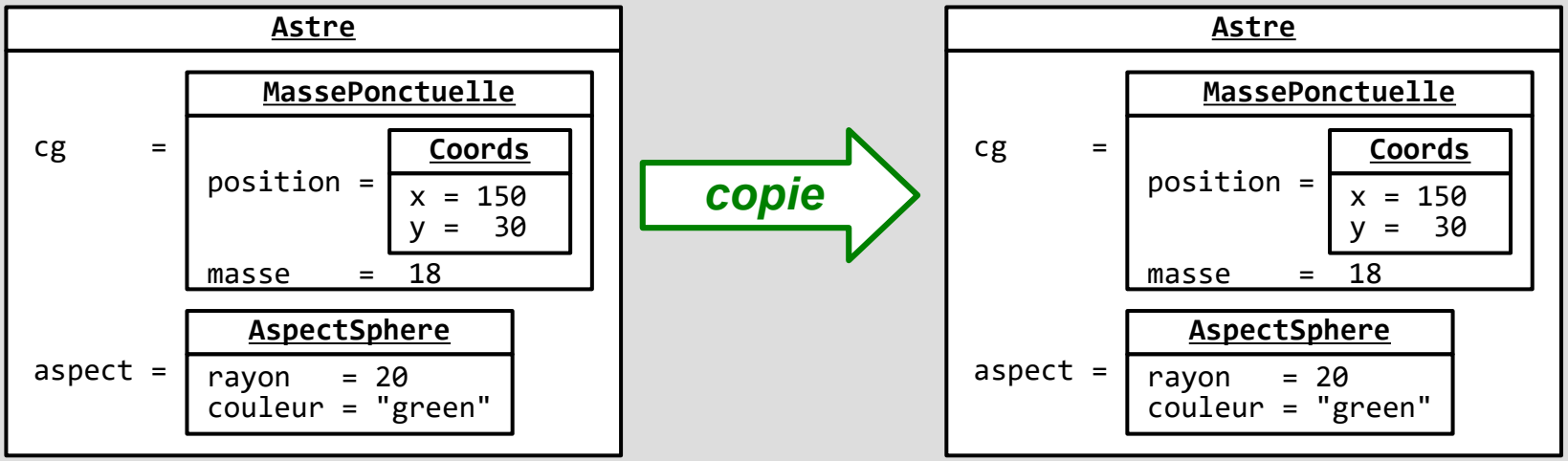
- *Une classe composite se comporte comme le propriétaire (**owner**) de ses composants*
- *Avec **attributs par valeur** sa responsabilité de terminer la vie de ses composants est **triviale** :* ***la destruction du composite entraîne automatiquement la destruction des composants :***



Appels en cascade des destructeurs implicites des composants

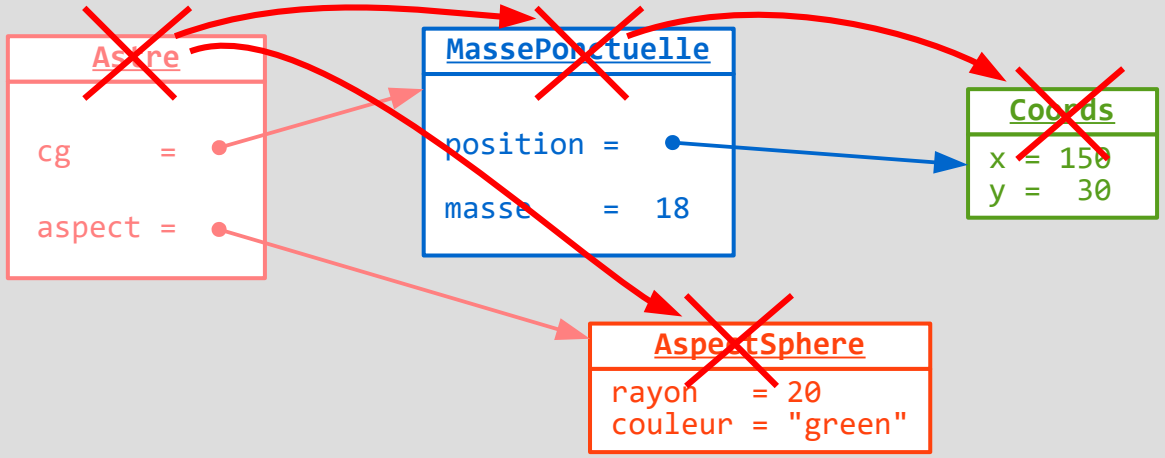
Composition en C++

- *Une classe composite se comporte comme le propriétaire (**owner**) de ses composants*
- *Avec **attributs par valeur** sa responsabilité de dupliquer ses composants est triviale :*
  constructeur par copie implicite OK
  opérateur d'affectation implicite OK



Composition en C++

- *Une classe composite se comporte comme le propriétaire (**owner**) de ses composants*
- *Avec **attributs par adresse** sa responsabilité de terminer la vie de ses composants est compliquée et risquée : **règle de 3***
  la destruction du composite n'entraîne la destruction des composants que si on le code bien :

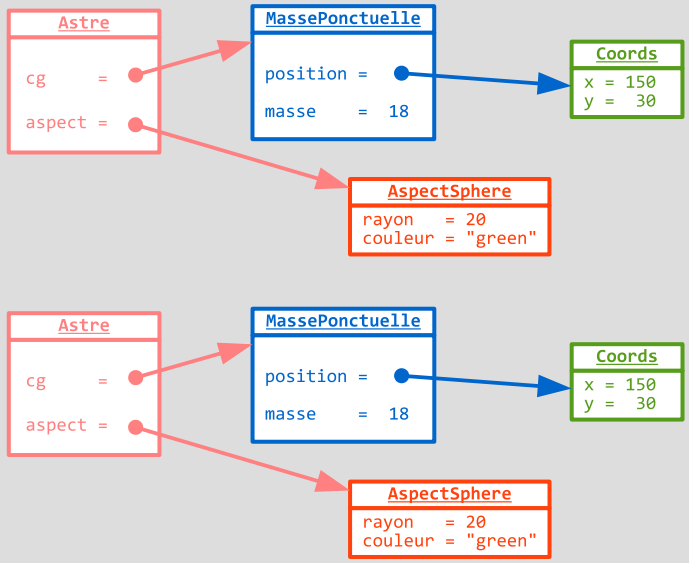


Destructeurs explicites à coder pour libérer les composants

Composition en C++

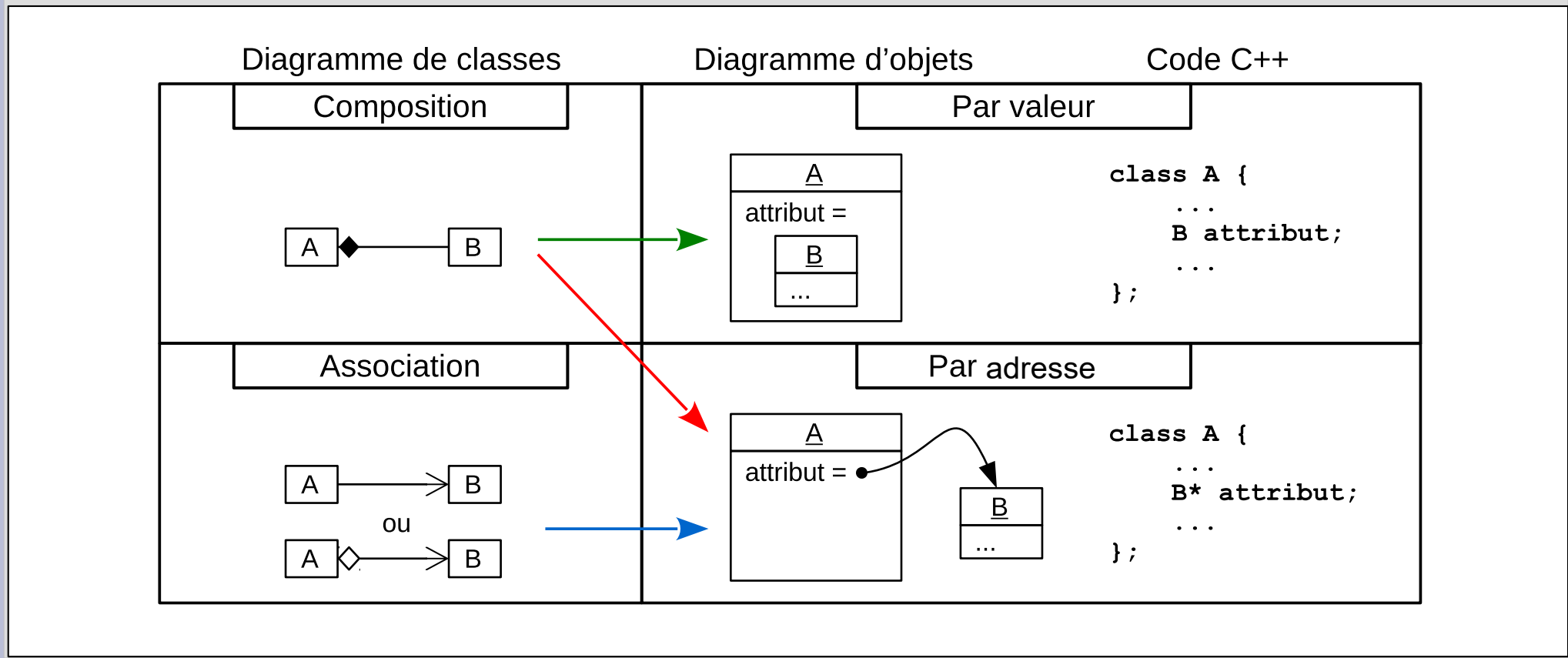
- *Une classe composite se comporte comme le propriétaire (**owner**) de ses composants*
- *Avec **attributs par adresse** sa responsabilité de dupliquer ses composants est complexe :*
  constructeur par copie implicite NON
  opérateur d'affectation implicite NON

Règle de 3 :
coder les 3 méthodes
- Destructeur
- Constructeur par copie
- Opérateur d'affectation



Composition en C++

- *En C++ on essayera toujours d'avoir une composition par valeur*
- *Mais il y a d'autres contraintes qui peuvent amener à choisir une composition par adresse*



Composition en C++

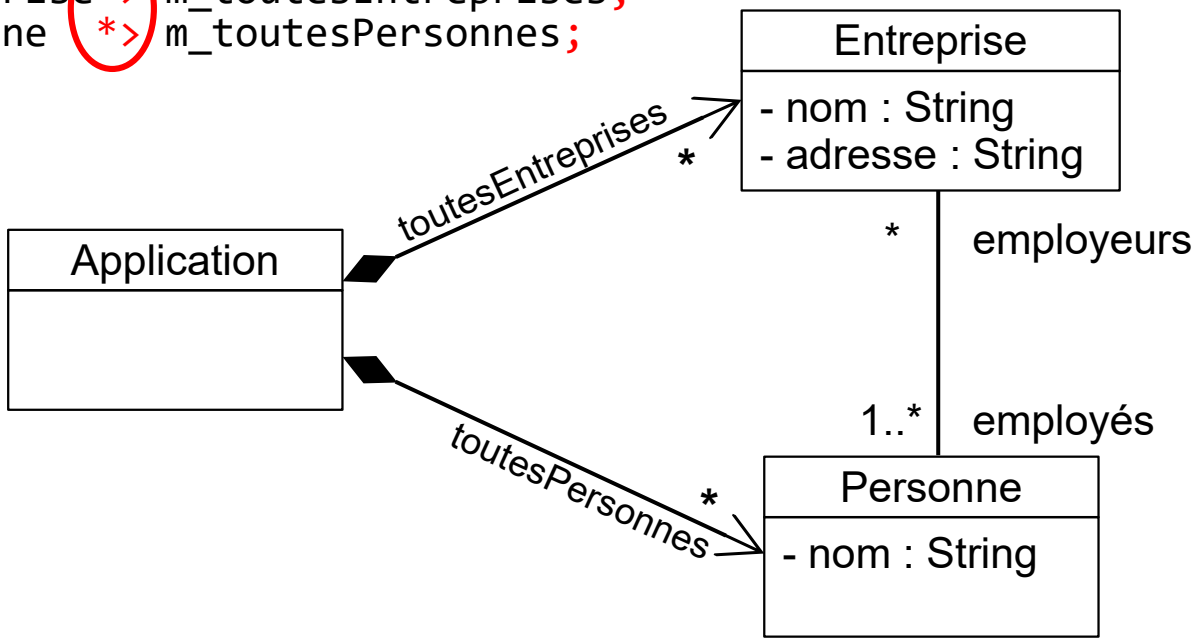
- *On envisagera ou on sera obligés de composer par adresse pour des attributs*
  - *de types entités (objets partagés, non copiables)*
  - *qui existent ou pas (nullptr dans ce cas)*
  - *qui existeront plus tard (idem précédent)*
  - *qui sont de tailles variables (réallocation)*
  - *lourds et/ou immuables (éviter des copies)*
  - *polymorphes (cours 8)*
- *Approche moderne smart pointers (hors programme)*

Composition en C++

- *Exemple typique de composition par adresse*
- *Une classe Application ou Document compose tous les objets entités alloués dynamiquement, leurs adresses doivent rester stable*

```
class Application
{
    private :
        std::vector<Entreprise*> m_toutesEntreprises;
        std::vector<Personne *> m_toutesPersonnes;
    ...
};
```

Voir slides 15 à 19



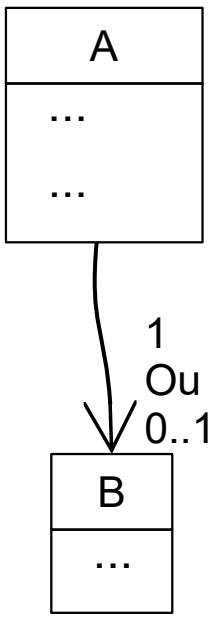
COURS 6

A) Du modèle objet au C++
B) Types valeur / types entité
C) Copiabilité en C++
D) Composition en C++
E) Associations à sens unique
F) Associations à double sens

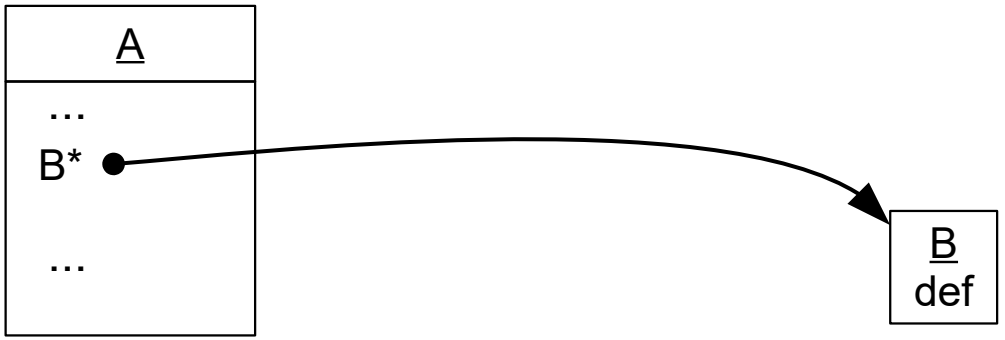
Associations à sens unique

Associations à sens unique

- *Association sens unique, multiplicité 1 ou 0..1*
- *stocker un **pointeur sur** une **entité***
- ***attribut pointeur sur***



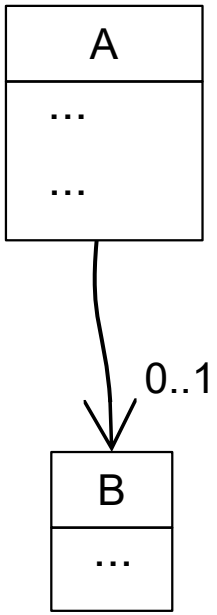
Objet avec attribut pointeur sur objet de type B



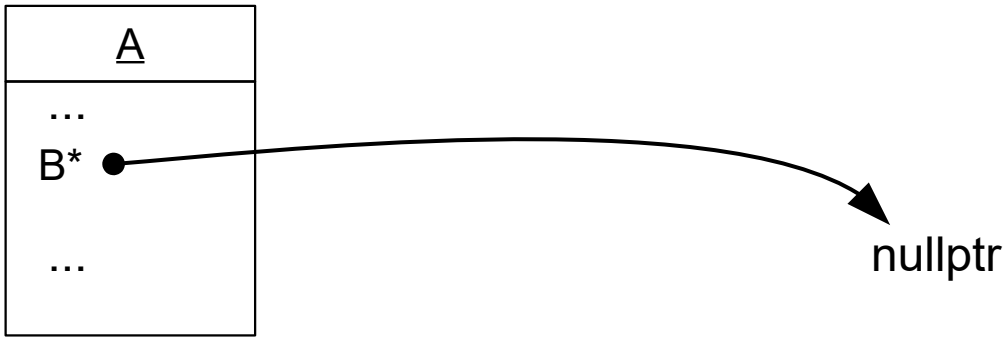

```
class A {
    ...
    B* objetB;
    ...
};
```

Associations à sens unique

- *Association sens unique, multiplicité 0..1*
- *Bricolage possible : utiliser nullptr pour indiquer qu'il n'y a pas d'objet B référencé pour l'instant. À tester systématiquement !*



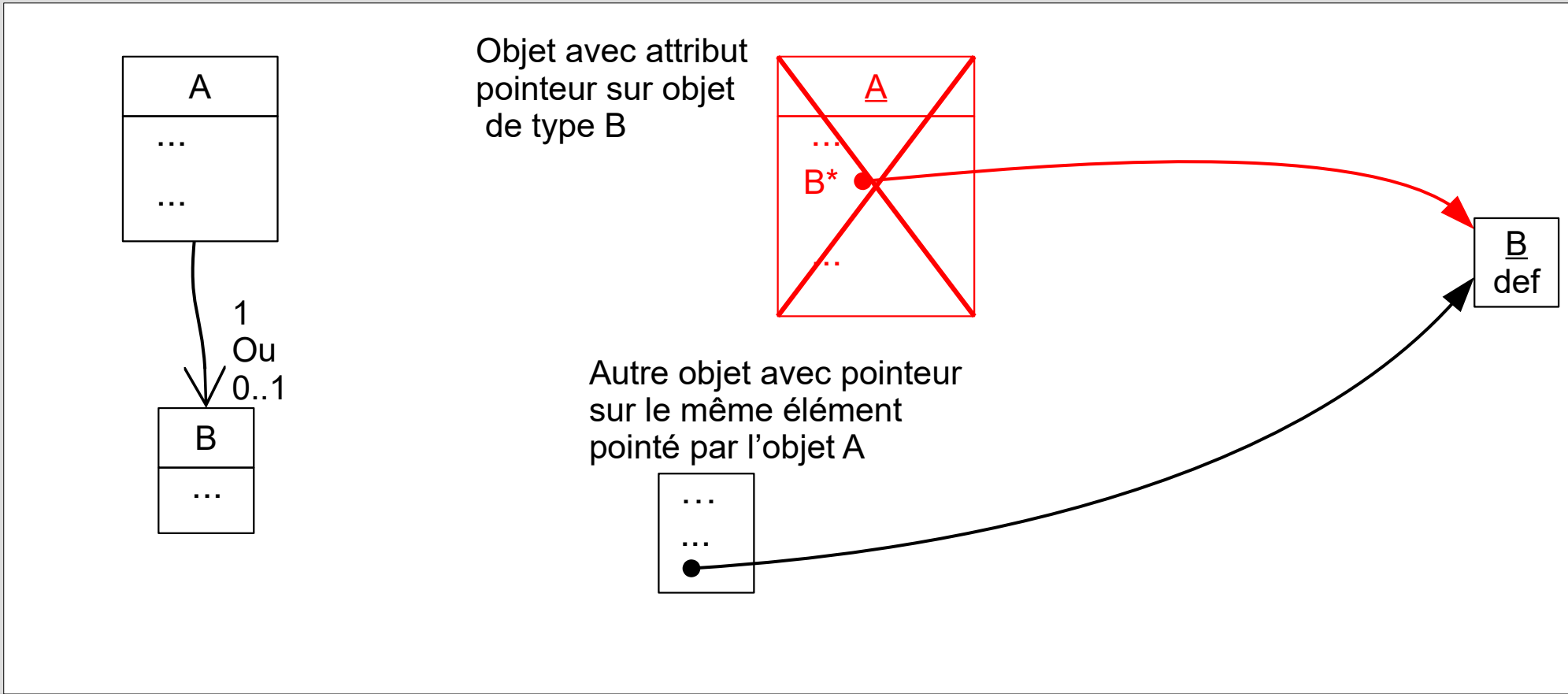
Objet avec attribut pointeur sur objet de type B




```
class A {
    ...
    B* objetB;
    ...
};
```

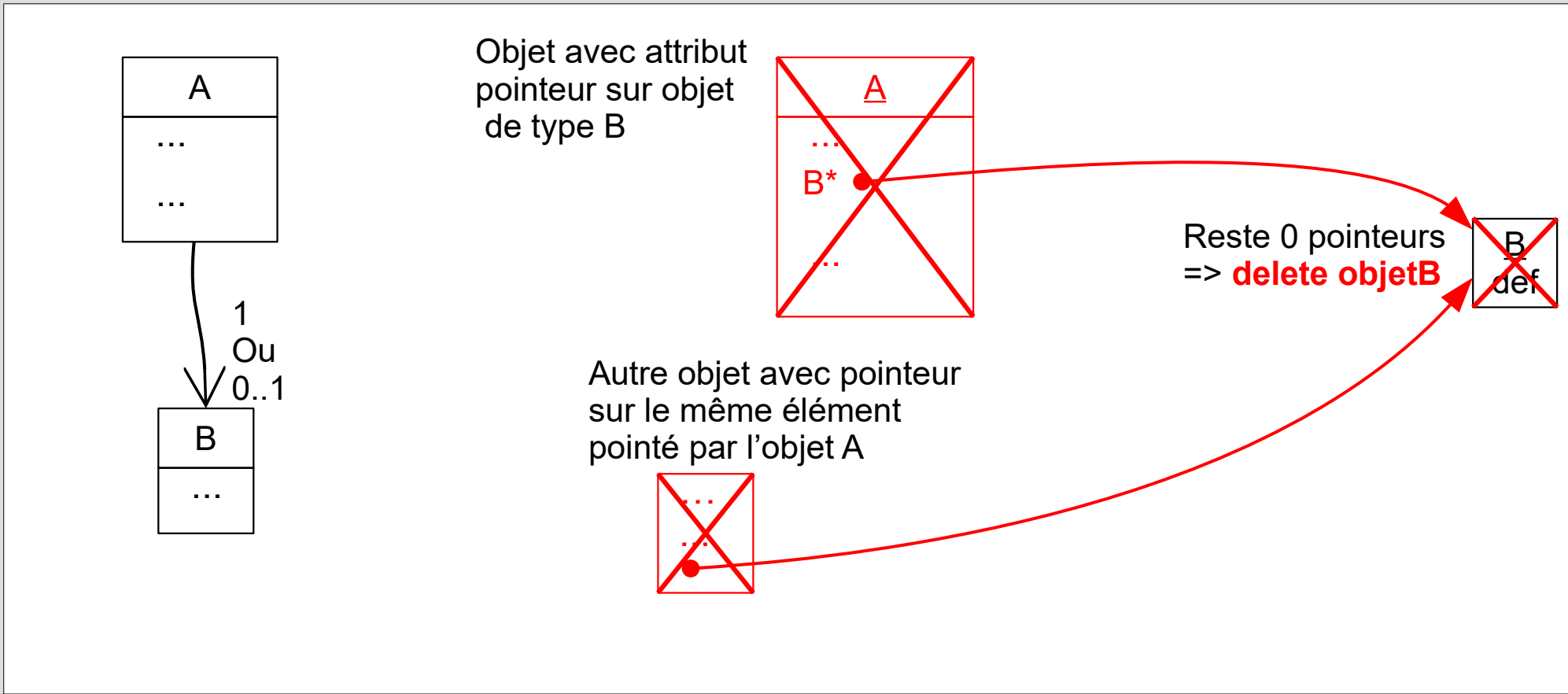
Associations à sens unique

- *Association sens unique, multiplicité 1 ou 0..1*
- *Ne pas « libérer » l'objet pointé dans le destructeur de A si l'objet B n'est pas une ressource de A ou si B est encore référencé !*



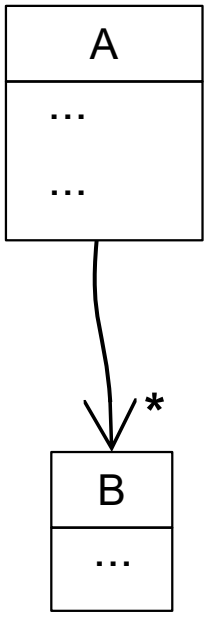
Associations à sens unique

- *Association sens unique, multiplicité 1 ou 0..1*
- *Si B est une ressource de A : voir composition*
  Si B est partagé, compter les références
  ou autre approche algorithmique...

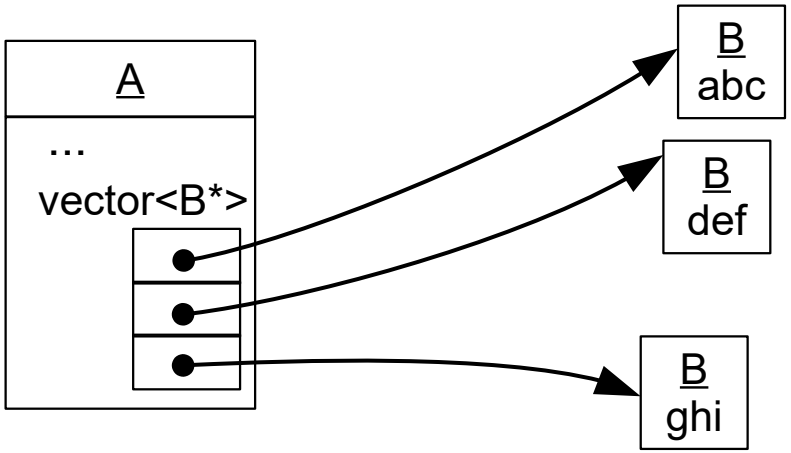


Associations à sens unique

- *Association sens unique, multiplicité **
- *stocker des* ***pointeurs sur*** *des* ***entités***
- ***attribut vecteur de pointeur sur***



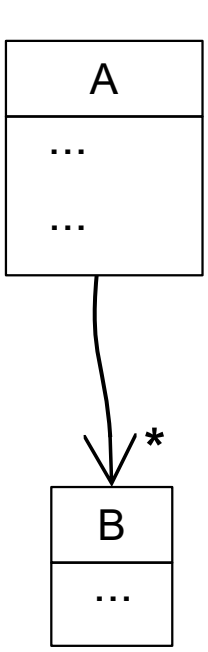
Objet avec attribut vecteur de pointeurs sur objet de type B



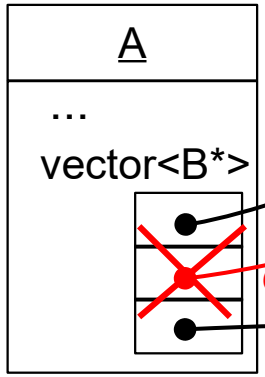

```
class A {
    ...
    std::vector<B*> objetsB;
    ...
};
```

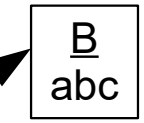
Associations à sens unique

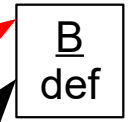
- *Association sens unique, multiplicité **
- *En général enlever un élément référencé ≠ détruire l'élément enlevé ...*

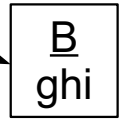


Objet avec attribut vecteur de pointeurs sur objet de type B



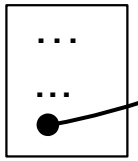






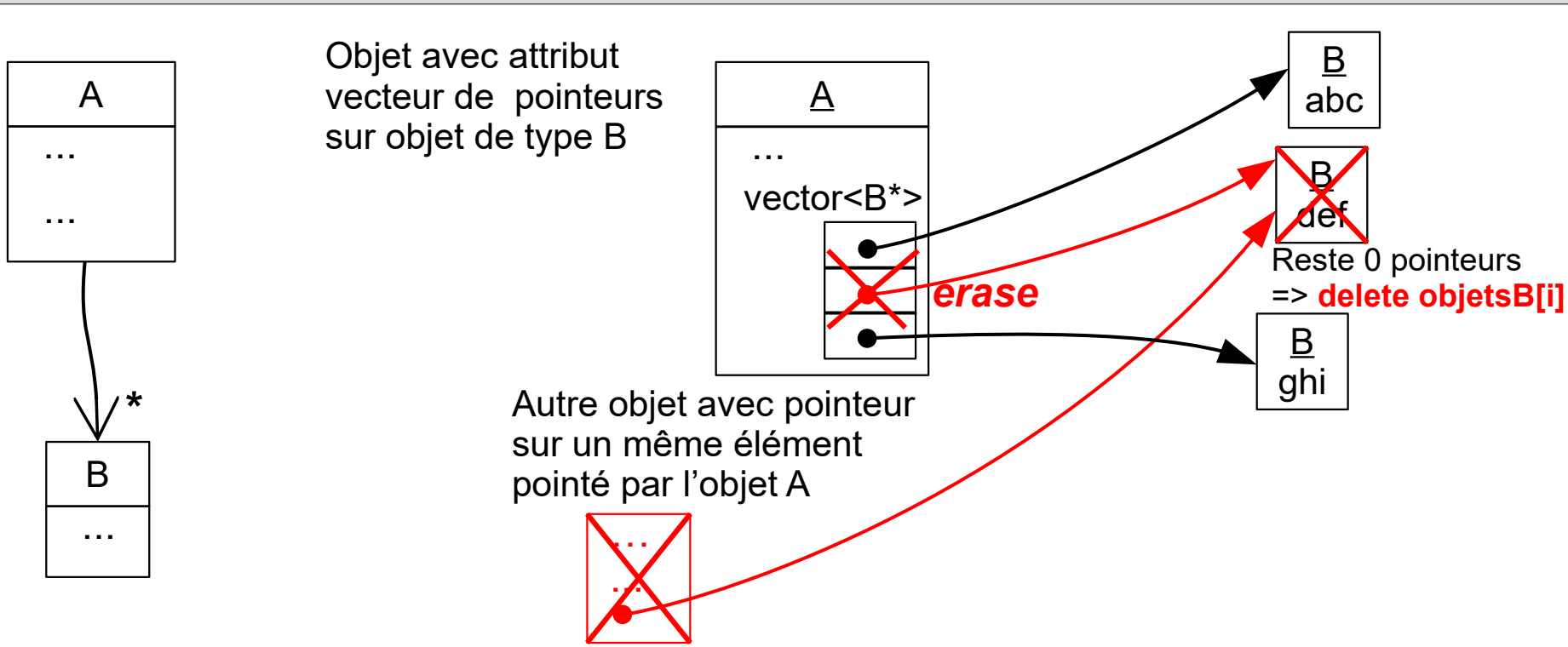
erase

Autre objet avec pointeur sur un même élément pointé par l'objet A



Associations à sens unique

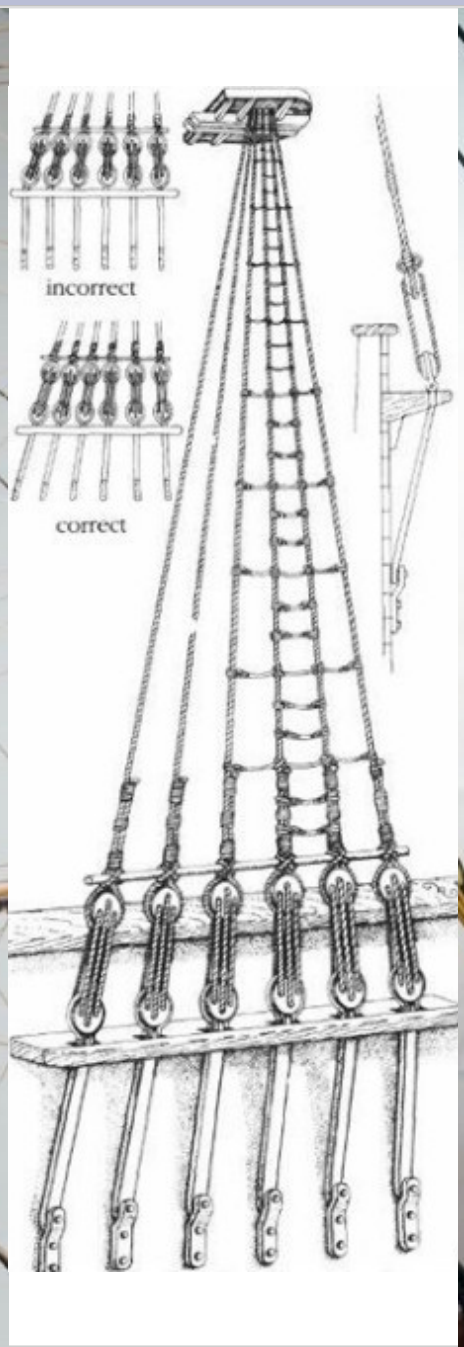
- *Association sens unique, multiplicité **
- *Si B est une ressource de A : voir composition*
  Si B est partagé, compter les références
  ou autre approche algorithmique...



COURS 6

A) Du modèle objet au C++
B) Types valeur / types entité
C) Copiabilité en C++
D) Composition en C++
E) Associations à sens unique
F) Associations à double sens

Associations à double sens



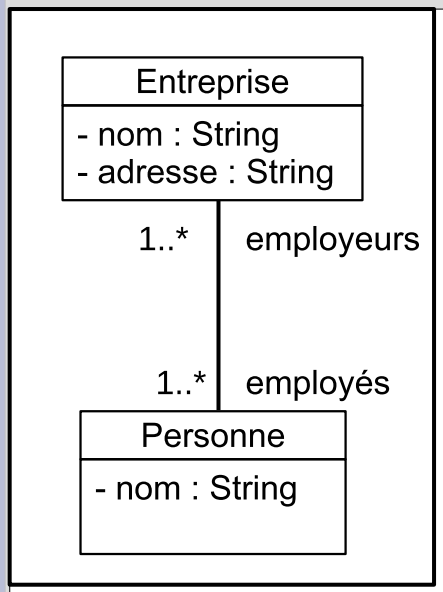
Associations à double sens

- *Les associations à double sens entre entités s'implémentent avec des pointeurs réciproques*

- *En général la double navigation implique une **redondance** des données (il y a plusieurs façon d'obtenir la même indication) et donc un risque accru sur la **cohérence** des données (non réciprocité d'un pointeur réciproque !)*

Associations à double sens

- *Les associations à double sens entre entités s'implémentent avec des pointeurs réciproques*

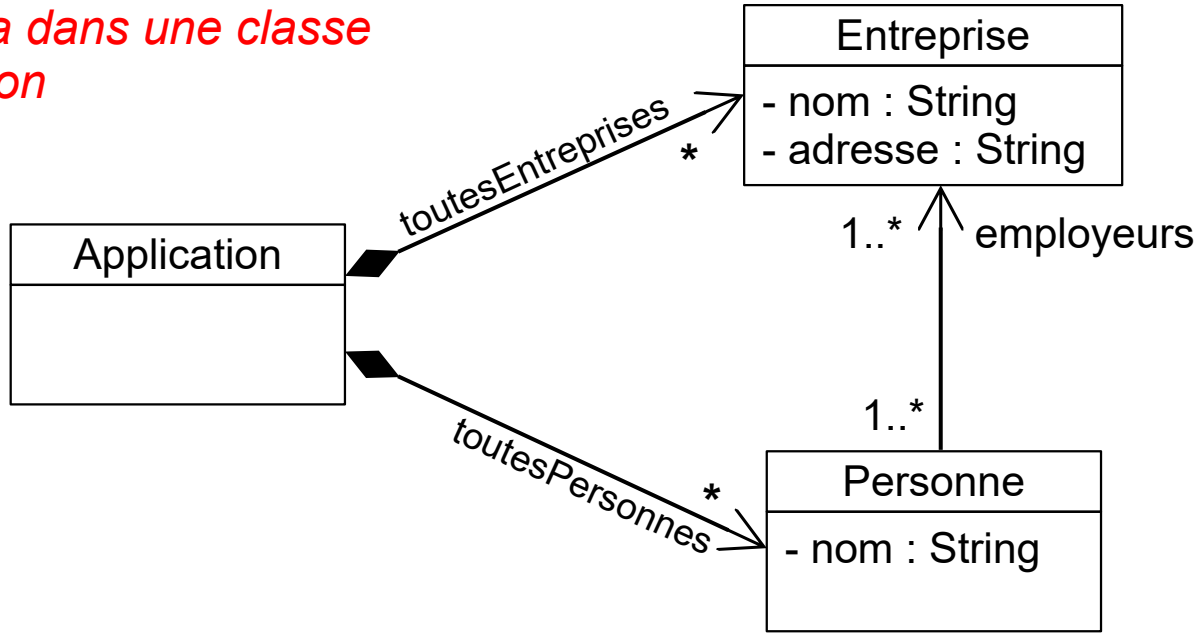


Attributs vecteurs de pointeurs

Associations à double sens

- *Vu le coût ressource et maintenance de la double navigation on envisage le sens unique*

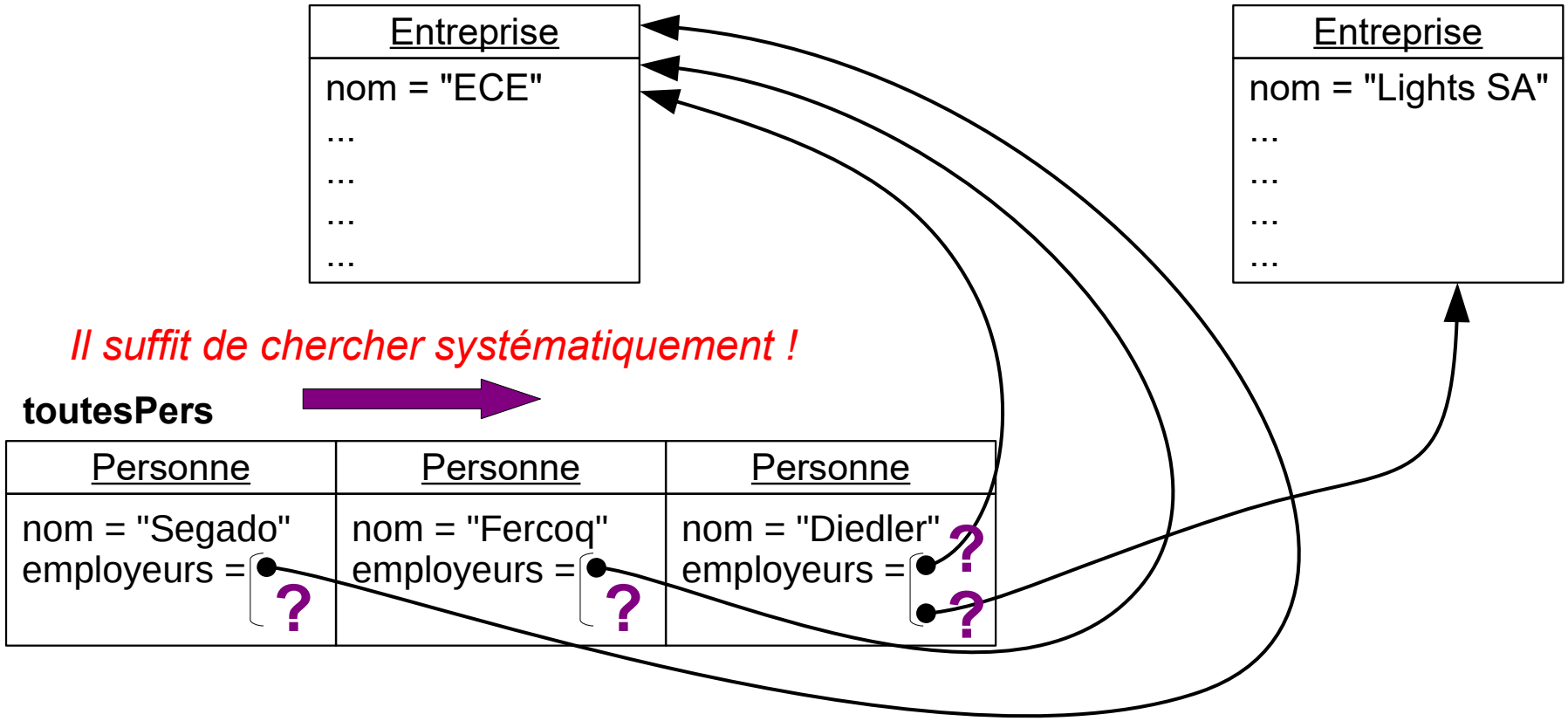
Souvent on a un ensemble de collections d'entités avec toutes les entités. On peut mettre ça dans une classe Application



Associations à double sens

- *Vu le coût ressource et maintenance de la double navigation on envisage le sens unique*

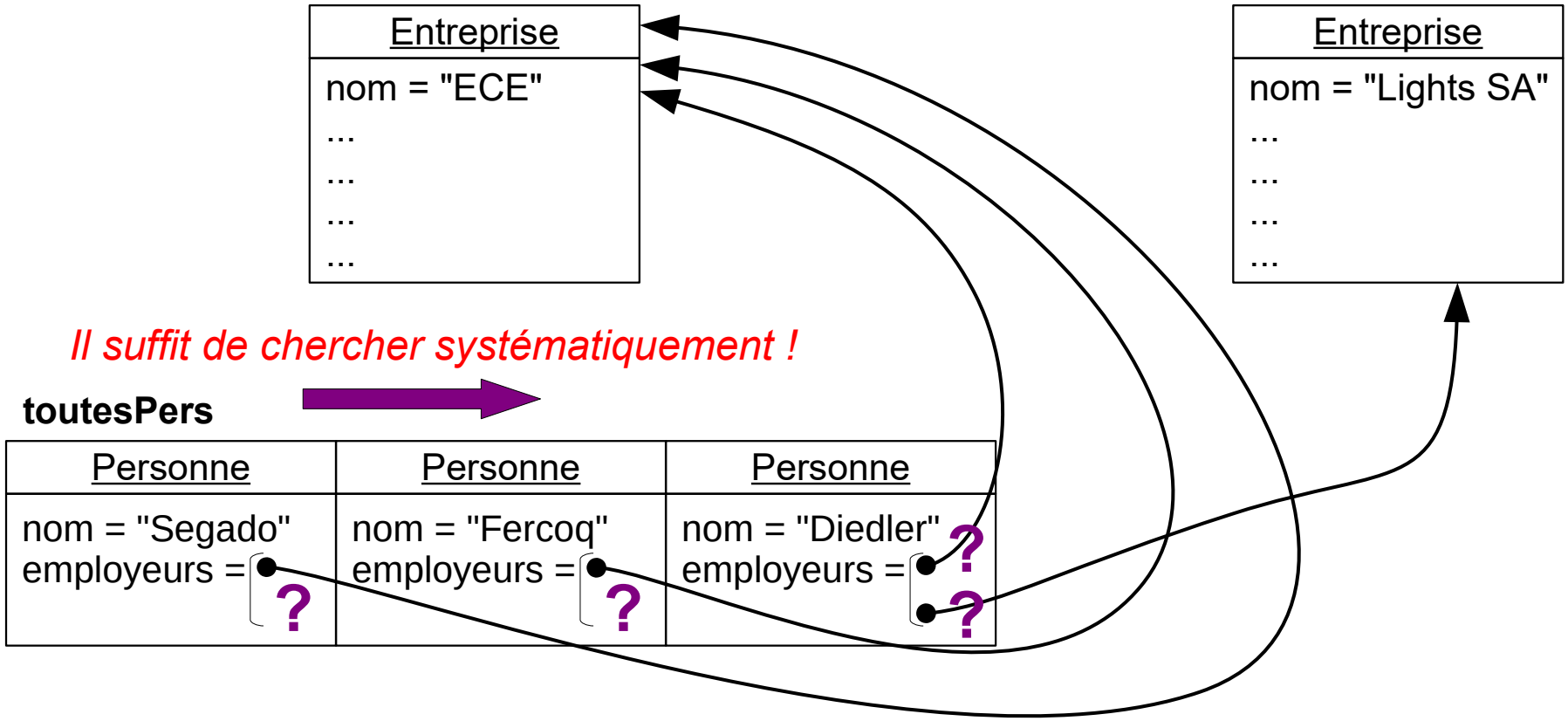
Avec les collections de toutes les entités on peut retrouver des association à contre sens : ici on va savoir qui travaille chez Lights SA même si il n'y a pas de navigation directe de Entreprise vers Personne



Associations à double sens

- *Selon le nombre d'entités et la fréquence du besoin en navigation inverse c'est envisageable*

Avec les collections de toutes les entités on peut retrouver des association à contre sens : ici on va savoir qui travaille chez Lights SA même si il n'y a pas de navigation directe de Entreprise vers Personne



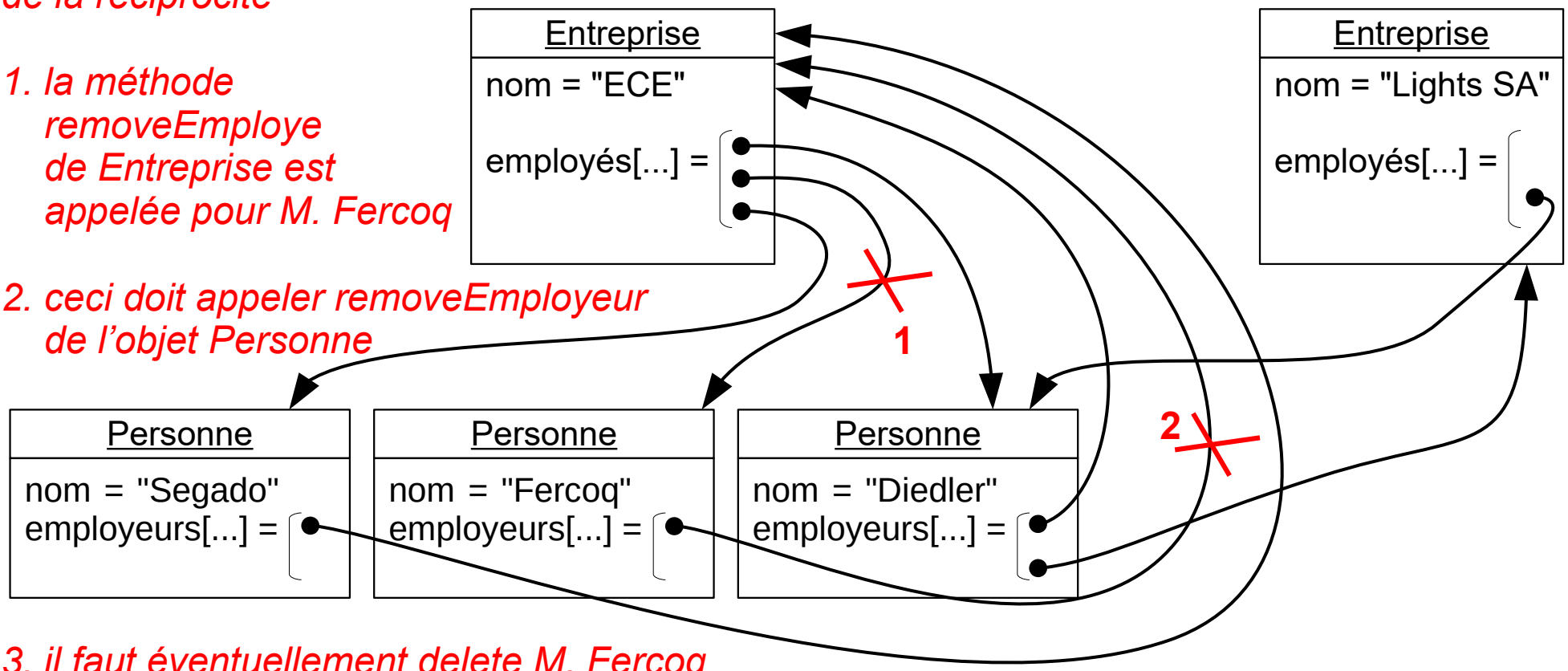
Associations à double sens

- *Si la double navigation s'impose, utiliser systématiquement des accesseurs/**mutateurs***

Les mutateurs doivent garantir la cohérence de la réciprocité

1. la méthode removeEmploye de Entreprise est appelée pour M. Fercoq

2. ceci doit appeler removeEmployeur de l'objet Personne



3. il faut éventuellement delete M. Fercoq si il ne reste plus aucun pointeur vers lui, idem pour l'objet ECE !