Programmation orienté objet en C++ (suite)

IUT Lyon 1

Yosra ZGUIRA

yosra.zguira@insa-lyon.fr

2016 - 2017

Les pointeurs entre les classes

Caracteristiques.h

```
#ifndef DEF_CARACTERISTIQUES
#define DEF_CARACTERISTIQUES

#include <iostream>
#include <string>

class Caracteristiques
{
   public:
   Caracteristiques();
   Caracteristiques(std::string  couleur, std::string  forme);
   void afficher() const;

   private:
         std::string C_couleur;
         std::string C_forme;
        };

#endif
```

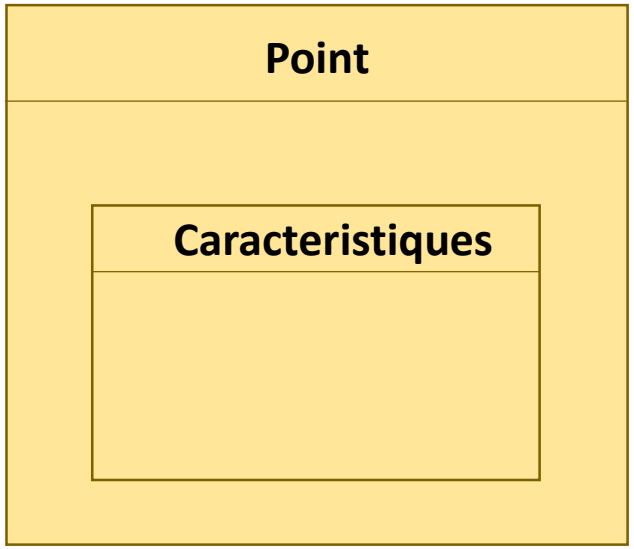
On crée une nouvelle classe ”Caracteristiques”.

- On ajoute un nouveau attribut « **coord_couleur** » de type Caracteristiques.

- Il faut ajouter donc **#include "Caracteristiques.h"** pour voir utliser un objet de type Caracteristiques.

- **Caracteristiques** est liée au **Point**, elle ne peut pas en sortir

  ➔ L'objet est contenu dans Point



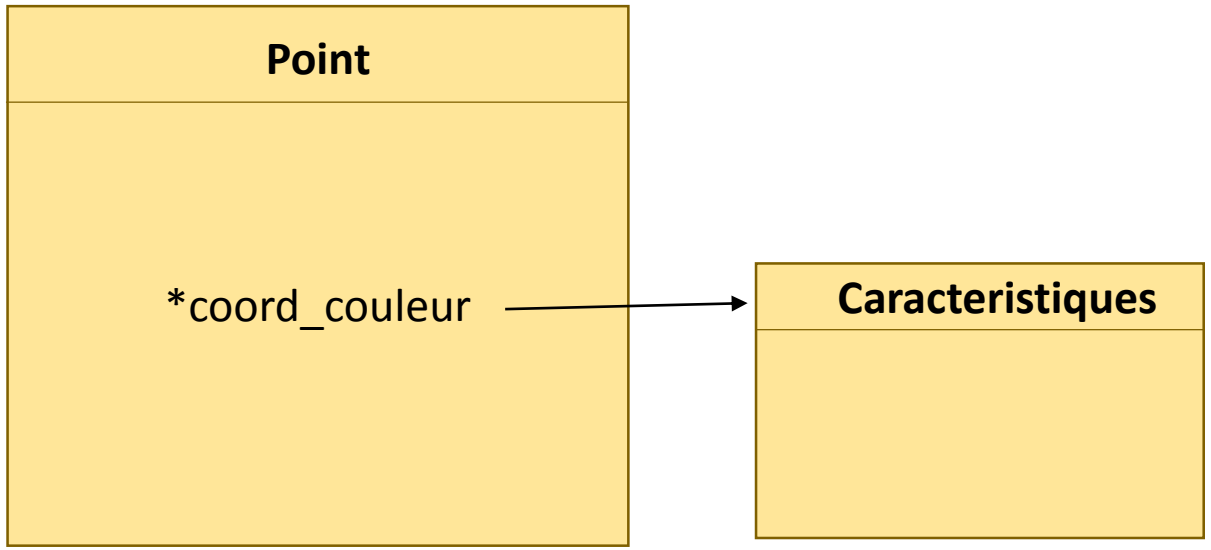
Point.h

```
#ifndef DEF_POINT
#define DEF_POINT

#include <iostream>
#include <string>
#include "Caracteristiques.h"

class Point {

private:
      int coord_x;
      int coord_y;
      Caracteristiques  coord_couleur;

public:
Point();
~Point();
void SetCoord(const int & x, const int & y);
void Afficher();

};
#endif
```

- Ne pas intégrer Caracteristiques dans Point

  ➔ Utiliser **un pointeur**

- Caracteristiques est un **pointeur**, donc l'objet n'est plus contenu dans Point



Point.h

```
#ifndef DEF_POINT
#define DEF_POINT

#include <iostream>
#include <string>
#include "Caracteristiques.h"

class Point {

private:
      int coord_x;
      int coord_y;
      Caracteristiques  *coord_couleur;

public:
Point();
~Point();
void SetCoord(const int & x, const int & y);
void Afficher();

};
#endif
```

Allocation dynamique

Allocation de mémoire de l'objet

- Caracteristiques est un ***pointeur***, il faut créer son allocation dynamique avec **new**.

- L'allocation de mémoire pour Caracteristiques se fait dans le **constructeur**:

```
Point::Point() : x(0), y(0), coord_couleur(0)  {

    coord_couleur  =  new  Caracteristiques();
}
```

ou bien

```
Point::Point(int  abscisse, int  ordonne) : x(0), y(0), coord_couleur(0)  {

  coord_couleur  =  new  Caracteristiques(abscisse, ordonne);

}
```

NB: Par sécurité, on initialise d'abord le pointeur à **0** dans la liste d'initialisation puis on fait l'allocation avec le **new** entre les accolades du constructeur.

Désallocation de mémoire de l'objet

- Il faut utiliser un "**delete**" dans le **destructeur**.

```
Point::~Point() {

   delete   coord_couleur;

}
```

Les attributs dynamiques (1/3)

- **Les attributs dynamiques** sont alloués soit à la construction de l'objet soit pendant son utilisation.

- Ce genre d'objets aux attributs dynamiques sont très fréquemment utilisés en C++.

- **Prenons un exemple**, une classe « Etudiant » ayant deux attributs son **nom** et sa **moyenne**.

  On ne sait pas à l'avance la taille du nom de tous les objets: la chaine de caractères aura une taille différente pour chaque instance.

  ➔ **On déclare un pointeur sur une chaîne de caractères comme attribut des objets de la classe « Etudiant »**

  ➔ **Création de la classe « Etudiant » avec une gestion dynamique du nom.**

Les attributs dynamiques (2/3)

```
class Etudiant {

        private:
        char * nom;             // Pointeur sur une chaine de caractères
        double moyenne;

        public:
        Etudiant();             // Constructeur par défaut

        Etudiant(char *);   // Constructeur qui initialise le nom de l'etudiant
        …
        ~Etudiant();          // Destructeur

};
```

Les attributs dynamiques (3/3)

```
// Constructeur par défaut, le pointeur nom est initialisé au pointeur nul, la moyenne à 0
Etudiant::Etudiant() : nom(0), moyenne(0.0) {}

// Constructeur avec le nom en paramètre , il alloue un tableau de caractères de la taille du nom, puis copie la chaine passée en argument, la moyenne est initialisée à zéro
 Etudiant::Etudiant(char * name) : moyenne(0.0) {
     if(name) {
         nom = new char[strlen(name)+1];
        strcpy(nom, name);
     }
 }

// Destructeur indispensable pour libérer la chaine de caractère
Etudiant::~Etudiant() {
    delete [] nom;
}
```

Il ne faut pas oublier de supprimer les attributs alloués de façon dynamique : cela doit être fait *a priori* dans le destructeur.

Le pointeur this

Le pointeur this

- Dans toutes les classes, ils existent le pointeur "**this**" qui pointe vers l'objet actuel.

- **Le pointeur "this"** permet à une méthode de classe de renvoyer un pointeur vers l'objet auquel elle appartient.

```
Point* Point::getAdresse() const
{
    return this;
}
```

Il ne faut pas nommer vos variables class, this, new, delete, return !!

➔ **Ce sont des mots-clés-réservés au langage C++.**

Le constructeur de copie et les pointeurs

Problématique (1/4)

- **Le constructeur de copie** est indispensable dans une **classe** qui contient des **pointeurs**.

- Reprenons l'exemple de la classe « Point »:

```
int main()
{
   Point p1(5,7);

   Point p2(p1);          //On crée p2 à partir de p1.
                          // p2 sera une « copie » de p1.

   return 0;
}
```

- Le rôle du constructeur de copie est de *copier* la valeur de tous les attributs du premier objet dans le second.

Problématique (2/4)

- **Problème:** Lors de l'envoi d'un objet à une fonction sans utiliser de pointeur ni de référence, **l'objet est là aussi copié !**

➔ Il est généralement préférable d'utiliser une **référence** car l'objet n'a pas besoin d'être copié.

➔ Cela va donc plus vite et nécessite moins de mémoire.

- Si le programmeur n'écrit pas un constructeur de copie pour la classe, il sera généré automatiquement par le compilateur.

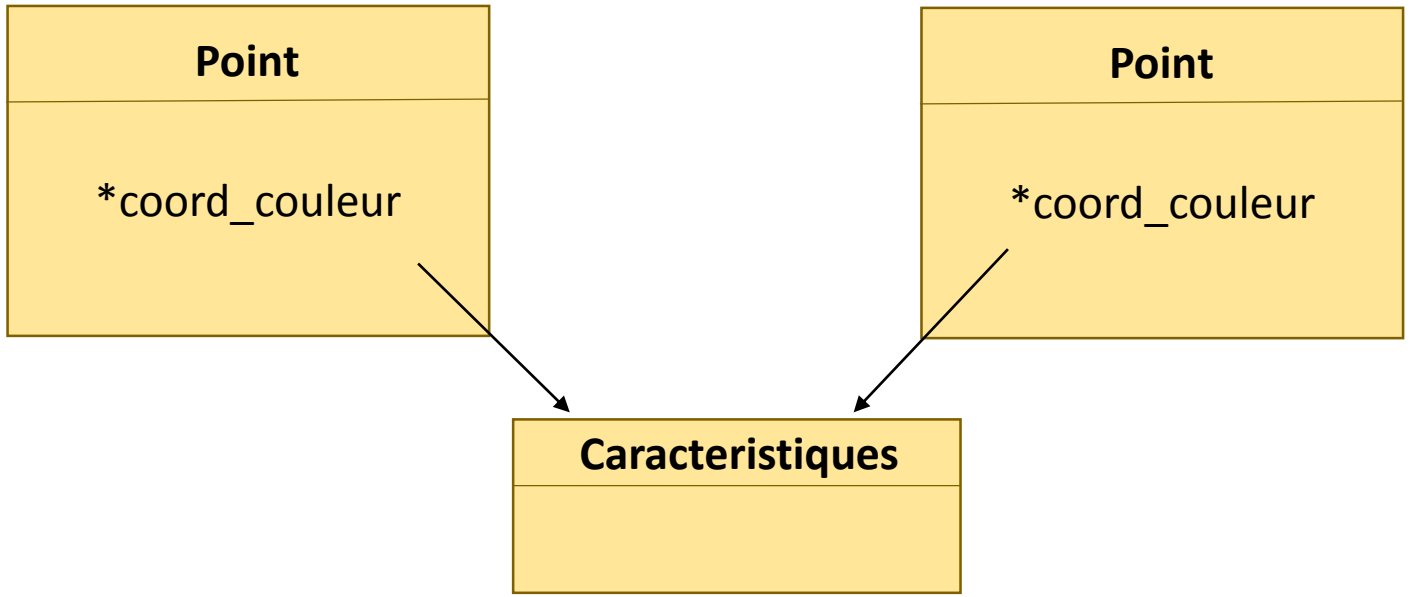
➔ **Le constructeur de copie généré se contente de copier la valeur de tous les attributs et même des pointeurs !**

Problématique (3/4)

- L'attribut **coord_couleur** est un pointeur.

- L'ordinateur copie la valeur du pointeur, donc l'adresse de « Caracteristiques ».

➔ **Probléme:** Les deux objets ont un pointeur qui pointe vers le même objet de type « Caracteristiques ».

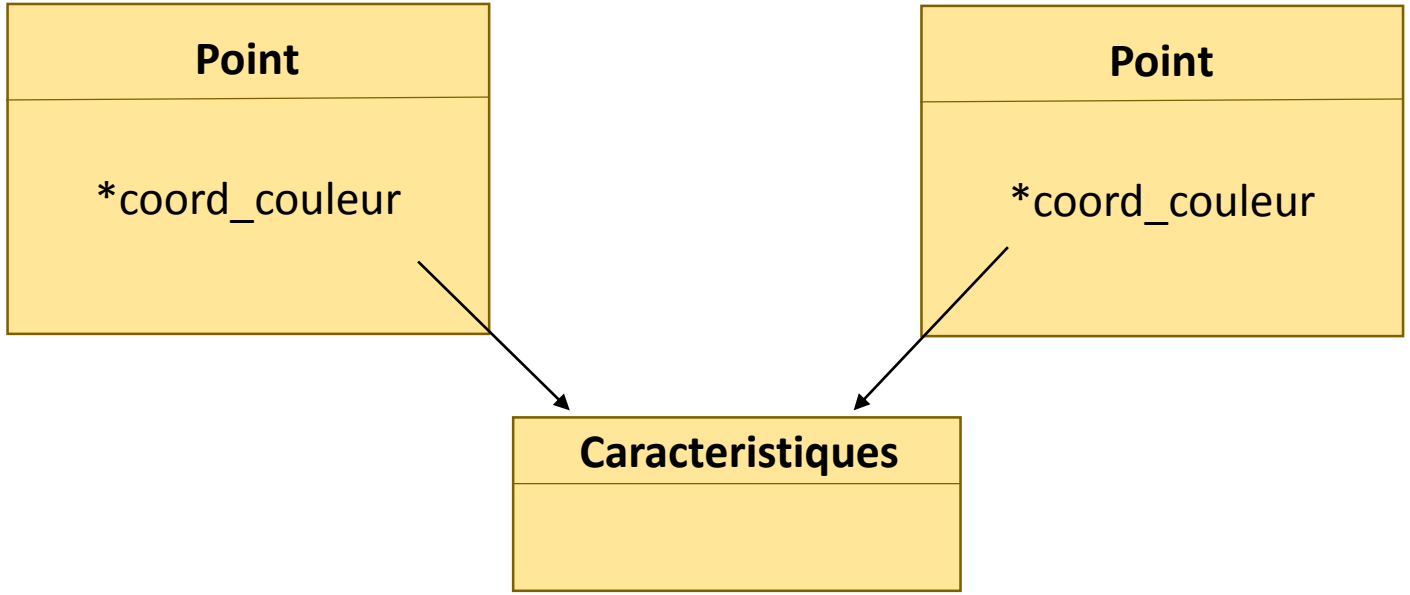



```
class Point {

private:
      int coord_x;
      int coord_y;
      Caracteristiques  *coord_couleur;

public:
Point();
~Point();
void SetCoord(const int & x, const int & y);
void Afficher();

};
```

Problématique (4/4)

- **Problème: Si** le premier « Point » est détruit ainsi que ses « Caracteristiques » car le destructeur ordonne la suppression de « Caracteristiques » avec **delete**, **alors** lors de la destruction du deuxième point, le **delete** plante car « Caracteristiques » a été déjà détruite !

➔ Le constructeur de copie généré automatiquement par le compilateur n'est pas assez intelligent pour comprendre qu'il **faut allouer de la mémoire pour une autre « Caracteristiques ».**



Création du constructeur de copie (1/5)

- Le prototype d'un constructeur de copie est:

```
Objet(Objet const& objetACopier);
```

- Exemple:

```
Point(Point const& pointACopier);
```

➔ Le **const** indique que l'on n'a pas le droit de modifier les valeurs de l'objetACopier puisque on a seulement besoin de lire ses valeurs pour le copier.

Création du constructeur de copie (2/5)

- L'implémentation du constructeur est comme suit:

```
Point::Point(Point const& pointACopier) : x(pointACopier.x), y(pointACopier.y), coord_couleur(0)
{

}
```

➔ Tous les attributs du pointACopier sont copiés dans le Point actuel.

➔ Les attributs x et y du PointACopier sont privés, normalement on ne peut y accéder depuis l'extérieur !!

Un objet de type X peut accéder à tous les éléments (même privés) d'un autre objet du même type X.

Création du constructeur de copie (3/5)

- Il nous reste à copier « coord_couleur »:

```
coord_couleur = pointACopier.coord_couleur;
```

➔ **Même erreur que le compilateur:** on ne copie que l'adresse de l'objet de type « Caracteristiques » et non l'objet en entier !!

➔ Il faut donc copier l'objet de type « Caracteristiques » en faisant une **allocation dynamique** avec un **new**.

```
coord_couleur = new Caracteristiques();
```

➔ On crée une nouvelle « Caracteristiques » mais en utilisant le constructeur par défaut !

➔ Il faut appeler le constructeur de copie de « Caracteristiques » en passant en paramétre l'objet à copier.

Création du constructeur de copie (4/5)

```
coord_couleur = new Caracteristiques(pointACopier.coord_couleur);
```

➔**coord_couleur** est un **pointeur** et le prototype du constructeur de copie est sous cette forme:

```
Caracteristiques(Caracteristiques const& caracteristiques);
```

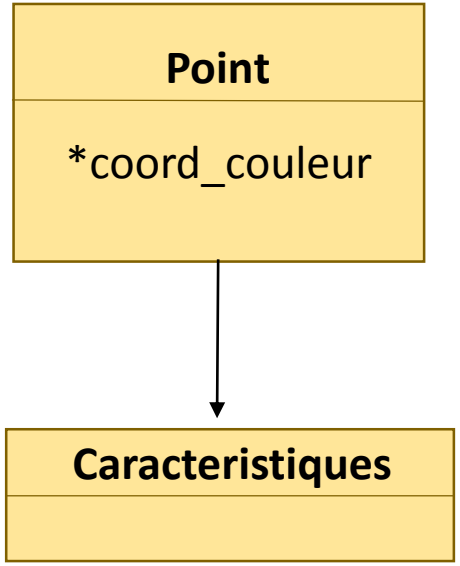
➔Donc, il faut envoyer l'objet lui-même et non son adresse.

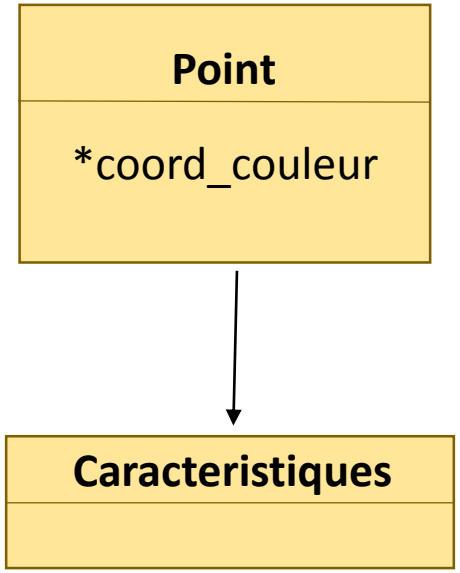
```
coord_couleur = new Caracteristiques(*(pointACopier.coord_couleur));
```

Création du constructeur de copie (5/5)

- Le constructeur de copie final:

```
Point::Point(Point const& pointACopier) : x(pointACopier.x), y(pointACopier.y), coord_couleur(0)
{
    coord_couleur  =  new  Caracteristiques(*(pointACopier.coord_couleur));

}
```





➔ Les deux points ont chacun une « Caracteristiques » identique mais dupliquée.

L'opérateur d'affectation (1/3)

- L'opérateur d'affectation est l'opérateur « **operator=** », vu dans la partie de la surcharge des opérateurs.

- La méthode **operator=** sera appelée lors de l'affectation d'une valeur à un objet.

```
Point p2 = p1;          //Constructeur de copie

p2 = p1;                //operator=
```

- Il ne faut pas confondre entre le constructeur de copie et la surcharge de l'opérateur **operator= !**

➔ ***Le constructeur de copie*** **est appelé lors de l'initialisation (à la création de l'objet)**

➔ ***L'opérateur « operator= »*** **est appelé lors de l'affectation à un autre objet après son initialisation.**

L'opérateur d'affectation (2/3)

- La méthode « **operator=** » effectue la même tâche que le constructeur de copie:

```
Point& Point::operator=(Point const& pointACopier)
{
   if(this != &pointACopier)
   //Vérifier que l'objet n'est pas le même que celui reçu en argument
   {
      x = pointACopier.x;          //Copier tous les champs
      y = pointACopier.y;

      delete  coord_couleur;
      coord_couleur = new  Caracteristiques(*(pointACopier.coord_couleur));
   }
   return *this;                   //Renvoyer l'objet lui-même
}
```

L'opérateur d'affectation (3/3)

- Il y a des différences entre le constructeur de copie et la méthode « operator=»:

  ✓ Il faut vérifier que se sont **deux objets distincts**, par la vérification de leurs adresses mémoires (this et &pointACopier) soient différentes.

  ✓ Il faut supprimer l'ancienne **« coord_couleur »** avant de créer la nouvelle.

  ➔ Ceci n'est pas nécessaire dans le constructeur de copie puisque le point ne possédait pas une « coord_couleur » avant.

  ✓ Il faut renvoyer ***this** ➔ c'est une règle à respecter.

L’héritage

Rappels: relations a-un, utilise-un, est-un (1/2)

- Soient les classes : **Voiture** , **Moteur** , **Route** et **Vehicule**.

  Quelles relations y a-t-il entre la classe Voiture et les trois autres classes ?

  ✓ Une voiture **a un** moteur : le moteur fait partie de la voiture, il est contenu dans celle-ci.

  ✓ Une voiture **utilise une** route, mais la route ne fait pas partie de la voiture ni inversement.

  ✓ Une voiture **est un** véhicule particulier : elle possède toutes les caractéristiques d'un véhicule, plus certaines qui lui sont propres.

Rappels: relations a-un, utilise-un, est-un (2/2)

- Du point de vue de la programmation:

  ✓ La relation **a-un** est une inclusion entre classes : un objet de type Voiture renferme une donnée membre qui est un objet de type Moteur

  ✓ La relation **utilise-un** est une collaboration entre classes indépendantes. Elle se traduit le plus souvent par un pointeur.

  ✓ La relation **est-un** s'appelle un **héritage**.

Vocabulaire et principes de l'héritage (1/3)

- Il consiste à partir d'une classe existante **Mere**, à définir une nouvelle classe **Fille**.

  ✓ La classe existante est appelée classe mère ou parente ou de base.

  ✓ La nouvelle classe est appelée classe fille ou classe dérivée.

  ✓ Mere est une super-classe de Fille.

  ✓ Fille est une sous-classe de Mere.

  ✓ La classe Fille dérive / hérite de la classe Mere.

  ✓ Une classe fille hérite automatiquement des données et méthodes de sa classe mère sans avoir à les réécrire.

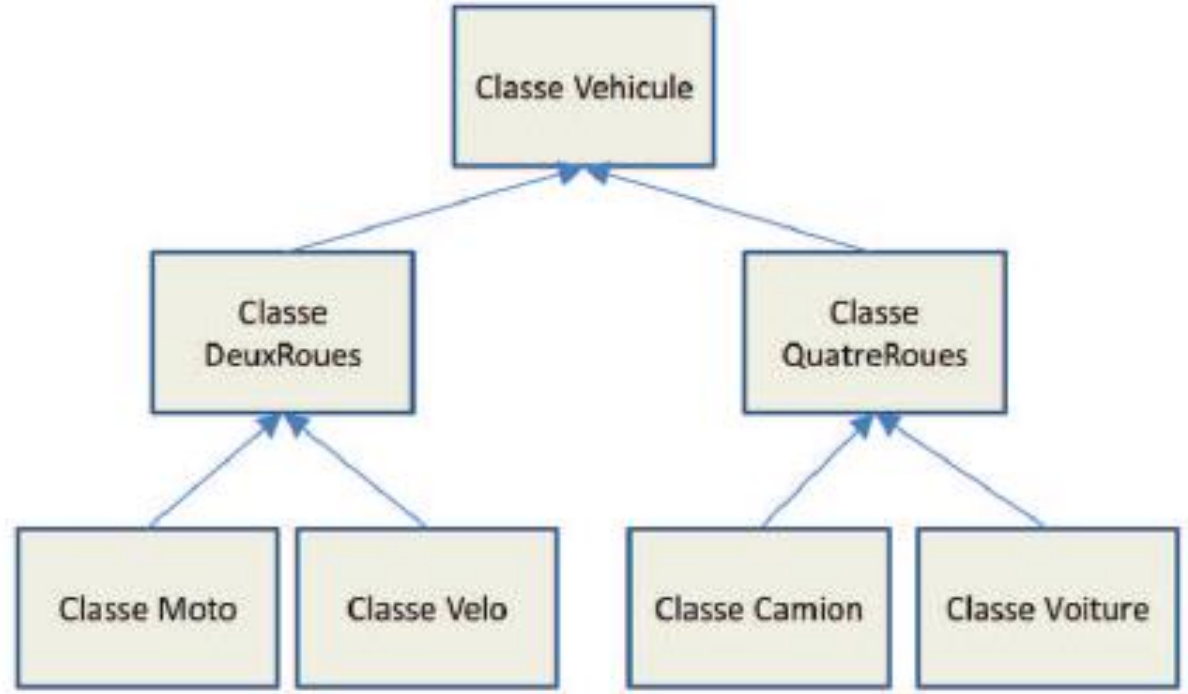
  ➔ **Attention:** la Fille n'hérite jamais les constructeurs de la Mere ni son destructeur.

Vocabulaire et principes de l'héritage (2/3)

- ✓ Une classe mère peut avoir plusieurs classes filles.
- ✓ Une classe fille peut à son tour servir de classe mère pour une autre classe fille.
- ✓ L'héritage simple, c'est quand une classe fille hérite d'une seule classe mère.
- ✓ L'héritage multiple, c'est quand une classe fille hérite simultanément de plusieurs classes mères.

- **Exemple:** Grâce à l'héritage, on peut avec peu de lignes, créer de nouvelles classes à partir de classes existantes, sans avoir à modifier ni à recompiler ces dernières.

Vocabulaire et principes de l'héritage (3/3)



- Exemple d'héritage (UML).
- La relation "**est-un**" ou "**hérite de**" s'exprime dans le sens des flèches

Syntaxe de l’héritage

- En C++, les classes peuvent hériter d'autres classes et la relation d'héritage est exprimée à l'aide de l'opérateur de dérivation "**:**".

```
class  B :  ‘type_héritage’  A
{
    …….

};
```

Visibilité des membres d'une classe

- Il existe 3 types d'héritage **public**, **private** ou **protected** qui permettent de spécifier si oui ou non une méthode de la classe B peut modifier une donnée membre de la classe A.

- **type_héritage** est l'un des mots clés d'accès public, protected ou private.
- Les membres de A sont alors à la fois accessibles à la classe dérivée B et en dehors de ces classes selon la valeur utilisée pour type_héritage :

  ✓ Quand type_héritage est **public**, les membres de la classe A conservent leur accès (privé, protégé et public).

  ✓ Quand type_héritage est **protected**, les membres publics de la classe A deviennent protégés.

  ✓ Quand type_héritage est **private**, les membres publics et protégés de la classe A deviennent privés.

La portée protected

- La portée **protected** est un type de droit d'accès classé entre la portée public (le plus permissif) et la portée private (le plus restrictif).

- Il est généralement utilisé lors de l'héritage et donc par les classes mère. Il peut aussi être utiliser sur toutes les classes même s'il n'y a pas d'héritage.

- Les éléments qui suivent **protected** ne sont pas accessibles depuis l'extérieur de la classe, sauf si c'est une classe fille.

- Elle est équivalente à private mais elle est un peu plus ouverte : les classes filles peuvent elles aussi accéder aux éléments.

- **Conseil:** **donner toujours la portée « protected » aux attributs de vos classes !**

Exemple (1/2)

VehiculeRoulant.h

```
class VehiculeRoulant {

 public:
       VehiculeRoulant();
       void avancer();
       void accelerer();

 protected:
       int position, vitesse;

};
```

VehiculeRoulant.cpp

```
VehiculeRoulant::VehiculeRoulant() :
position(0) , vitesse(0) {

}

void VehiculeRoulant::avancer() {
        position += vitesse;
 }

void VehiculeRoulant::accelerer() {
         vitesse++;
}
```

Exemple (2/2)

Automobile.h

```
class Automobile : public VehiculeRoulant
{

public:
    Automobile(int places);

protected:
    int m_places;

};
```

Automobile.cpp

```
Automobile::Automobile(int places) :
VehiculeRoulant(), m_places(places) {

}
```

➔ **: VehiculeRoulant():** initialiser les variables héritées (position et vitesse).

Méthodes virtuelles (1/4)

- Dans l'exemple précédent, si l'on ajoute une autre méthode à la classe **VehiculeRoulant** :

VehiculeRoulant.h

```
class VehiculeRoulant {

 public:
        ...
        void accelererEtAvancer();
        ...
};
```

VehiculeRoulant.cpp

```
void  VehiculeRoulant::accelererEtAvancer() {

accelerer();
avancer();

 }
```

Méthodes virtuelles (2/4)

- On ajoute une méthode « avancer() » dans **Automobile** afin de l'avancer:

Automobile.h

```
class Automobile : public VehiculeRoulant
{

public:
     ...
    void avancer();
    ...
};
```

Automobile.cpp

```
void Automobile::avancer() {

    position += vitesse - frottementRoues;

}
```

Méthodes virtuelles (3/4)

- **Problème:** si on utilise la méthode **accelererEtAvancer()** sur un objet de classe **Automobile**, le résultat sera incorrect, car la méthode **avancer()** appelée sera celle définie par la classe **VehiculeRoulant**.

➔ Pour que ce soit celle de la classe Automobile qui soit appelée, il faut que la méthode **avancer()** soit définie comme **virtuelle** dans la classe de base VehiculeRoulant.

VehiculeRoulant.h

```
class VehiculeRoulant {

 public:
      ...
      virtual void avancer();
      ...
};
```

VehiculeRoulant.cpp

```
virtual void  VehiculeRoulant::avancer() {

   position += vitesse;

 }
```

Méthodes virtuelles (4/4)

- Par exemple, pour appeler la méthode **avancer()** de VehiculeRoulant à partir d'une méthode de Automobile:

```
void Automobile::avancer() {


    VehiculeRoulant::avancer();

    position -= frottementRoues;


}
```

Constructeur et destructeur

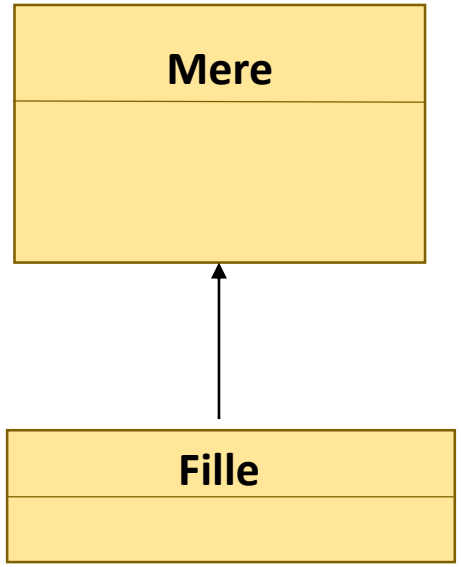
- Quand un objet Fille est créé, le constructeur de la classe Mere est d'abord appelé.
- Quand un objet Fille est détruit, le destructeur de la classe Mere est appelé après.
- Trace:

  Mere()

  Fille()

  ~Fille()

  ~Mere()



Destructeur virtuel (1/3)

- Il est important d'utiliser un **destructeur virtuel** pour toute classe qui sera dérivée (Mere).

- Dans le cas contraire seul celui de la classe de base est appelé, sans libérer les membres des sous-classes (Fille).

  ➔ Cela permet de ne pas avoir de **fuite mémoire** dans le cas où on détruirait un objet Fille à partir d'un pointeur de sa Mere.

- Exemple:

Destructeur virtuel (2/3)

```
int main()
{
   Mere *f = new Fille;        // une Fille est une Mere
                               //On crée une Fille et on met son adresse dans un pointeur de
                                   Mere
...
delete f;                      // On détruit un objet de Type Mere
return 0;                      // C'est le destructeur de la Mere qui est appelé
}
```

- Trace:

  Mere()

  Fille() // pas de ~Fille() donc fuite de mémoire car f est de type Mere*

  ~Mere()

Destructeur virtuel (3/3)

- Pour supprimer le constructeur de la classe fille, il faut donc déclarer le **destructeur virtuel** , auquel cas, c'est le destructeur de Fille qui est appelée, qui génère l'appel du destructeur de Mere :

```
classe Mere
{
  public:
      virtual ~Mere()
}
```

Héritage multiple

- L'héritage multiple permet à une classe d'hériter de plusieurs super-classes.
- Ceci permet d'intégrer dans la sous-classe plusieurs concept d'abstractions qui caractérisent cette sous-classe.
- Pour cela, il suffit de déclarer les super-classes les unes après les autres.
- Si une classe C hérite de A et de B, la syntaxe sera la suivante:

```
class C : "type_héritage" A, "type_héritage" B "..."
{
    ......
 };
```

Le polymorphisme

Introduction

- **Le polymorphisme** est une fonctionnalité de l'héritage : la capacité d'appeler une méthode en fonction du type réel d'un objet (sa classe).

- Le polymorphisme rend possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérarchie des classes.

- Exemple: on suppose qu'on a les classes suivantes : **Vehicule**, **Voiture** et **Moto**.

Exemple (1/4)

```
class Vehicule
{
   public:
   void affiche() const;

   protected:
   int m_prix;
};
```

```
class Voiture : public Vehicule
{
   public:
   void affiche() const;

   private:
   int m_portes;
};
```

```
class Moto : public Vehicule
 {
   public:
   void affiche() const;

   private:
   double m_vitesse;
};
```

Exemple (2/4)

- Le corps des fonctions **afficher()** des trois classes:

```
void Vehicule::affiche() const
{
   cout << "Ceci est un vehicule." << endl;
}

void Voiture::affiche() const
{
   cout << "Ceci est une voiture." << endl;
}

void Moto::affiche() const
{
   cout << "Ceci est une moto." << endl;
}
```

➔ Chaque classe affiche un message différent.

Exemple (3/4)

- Exemple de la fonction **main()**:

```
int main()
{
   Vehicule v;
   v.affiche();    //Affiche "Ceci est un vehicule."

   Moto m;
   m.affiche();    //Affiche "Ceci est une moto."

   return 0;
}
```

Exemple (4/4)

- Ajoutons une fonction supplémentaire qui reçoit en paramètre un Vehicule et modifions la fonction **main()** précédente:

```
void presenter(Vehicule v)      //Présente le véhicule passé en argument
{
   v.affiche();
}

int main()
{
   Vehicule v;
   presenter(v);

   Moto m;
   presenter(m);

   return 0;
}
```

```
Ceci est un vehicule.

Ceci est un vehicule.
```

Problème: Le message n'est pas correct pour la moto !

Résolution statique des liens

- Comme il y a une relation d'héritage, nous savons qu'une moto est un véhicule, un véhicule amélioré en quelque sorte puisqu'il possède un attribut supplémentaire.

- La fonction **presenter()** reçoit en argument un Vehicule. Ce peut être un objet réellement de type Vehicule mais aussi une Voiture ou une Moto.

- Ce qui est important c'est que, pour le compilateur, à l'intérieur de la fonction, on manipule un Vehicule.

  ➔Peu importe sa vraie nature. Il va donc appeler la version Vehicule de la méthode **affiche()** et pas la version Moto comme on aurait pu l'espérer.

En termes techniques, on parle de **résolution statique des liens**. La fonction reçoit un Vehicule, c'est donc toujours la version Vehicule des méthodes qui sera utilisée.

Résolution dynamique des liens (1/3)

- Ce qu'on aimerait, c'est que la fonction **presenter()** appelle la bonne version de la méthode.

- Donc, il faut que la fonction connaisse la vraie nature du Vehicule.

  ➔ C'est ce qu'on appelle **la résolution dynamique des liens**.

- Lors de l'exécution, le programme utilise la bonne version des méthodes car il sait si l'objet est de type mère ou de type fille.

- Pour faire cela, il faut:

  ✓ Utiliser des **méthodes virtuelles**

  ✓ Utiliser un **pointeur** ou une **référence**

Résolution dynamique des liens (2/3)

❑ **Déclaration de la méthode virtuelle:**

```
class Vehicule
{
   public:
   virtual void affiche() const;

   protected:
   int m_prix;
};
```

```
class Voiture : public Vehicule
{
   public:
   virtual void affiche() const;

   private:
   int m_portes;
};
```

```
class Moto : public Vehicule
 {
   public:
   virtual void affiche() const;

   private:
   double m_vitesse;
};
```

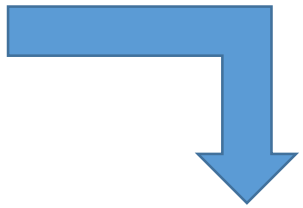
Il ne faut pas mettre virtual dans le fichier.cpp mais uniquement dans le .h !

Résolution dynamique des liens (3/3)

❑ **Utilisation d'une référence:**

- Réécrivons donc la fonction **presenter()** avec comme argument une référence.

```
void presenter(Vehicule const& v)   //Présente le véhicule passé en argument
{                                   //const: ne pas modifier l'objet dans la fonction
    v.affiche();
}

int main()
{
   Vehicule v;
   presenter(v);

   Moto m;
   presenter(m);

   return 0;
}
```



```
Ceci est un vehicule.

Ceci est une moto.
```

L’amitié

Qu'est-ce que l'amitié ?

- Dans les langages orientés objet, l'amitié est le fait de donner un accès complet aux éléments d'une classe.

- Donc si on déclare une fonction **f** amie de la classe **A**, la fonction **f** pourra modifier les attributs de la classe **A** même si les attributs sont privés ou protégés.
- La fonction **f** pourra également utiliser les fonctions privées et protégées de la classe **A**.

  ➔ On dit alors que **la fonction f est amie de la classe A**.

- En déclarant une fonction amie d'une classe, on ne respecte pas le concept de l'encapsulation de la classe puisque quelque chose d'extérieur à la classe pourra modifier ce qu'elle contient.

  ➔ Il ne faut donc pas abuser de l'amitié.

Fonctions amies

- **Les fonctions amies** se déclarent en faisant précéder la déclaration classique de la fonction du mot clé ***friend*** à l'intérieur de la déclaration de la classe cible.

- Les fonctions amies ne sont pas des méthodes de la classe correspondante (cela n'aurait pas de sens puisque les méthodes ont déjà accès aux membres de la classe).

- Exemple:

```
class A
{ int x;                              // Une donnée privée.
  friend void ecrit_a(int i);         // Une fonction amie.
};

 void ecrit_a(int i)
{  A.x=i;                             // Initialise x
   return;
}
```

Les espaces de noms (namespace)

C’est quoi les espaces des noms?

- **Les espaces des noms** ont été introduits pour permettre de faire des regroupements logiques et résoudre les collisions de noms.

- Un espace de nom permet de regrouper plusieurs déclarations de variables, fonctions et classes dans un groupe nommé.

- Un même nom pourra être déclaré dans des espaces de noms différents évitant ainsi la collision lorsqu'ils sont inclus en même temps dans une unité.

- Utiliser différents espaces des noms permet, lors d'un projet en équipe ou de l'utilisation de bibliothèques externes, de créer plusieurs entités portant le même nom.

Exemple

main.cpp

```
#include <iostream>

using namespace std;

const int livre = 8;

int main()
{
    cout << "Nombre livres = " << livre << endl;
    return 0;
}
```

bibliotheque.cpp

```
const int livre = 3;

...    //Plein de fonctions
```

➔ Le compilateur provoque une **erreur** car la variable est déclarée deux fois avec le même nom.

➔ Avec un namespace, le problème est résolu. On crée un namespace pour sa bibliothèque, l'utilisateur peut donc utiliser tout les noms qu'il veut sans risque de conflit.

Création d'un namespace

- La création d'un **namespace** se fait comme suit:

```
namespace NomDuNamespace
{
    //Contenu du namespace
}
```

➔ Il n'y a pas de point-virgule après l'accolade fermante.

- **Les déclarations de namespace** se mettent dans les fichiers header (**.h**) et dans les fichiers source (**.cpp**).

Exemple

bibliotheque.h

```
namespace Biblio
{
        class Livre
        {
            private:
            //Mes attributs
            public:
             //Le prototype de mes méthodes
        };

        //Le prototype des mes fonctions
        void afficher();

        //Des variables
        int nbrLivre = 8;
} //Fin du namespace
```

bibliotheque.cpp

```
#include  "bibliotheque.h"

namespace Biblio
{
      //Mes fonctions, méthodes
      Livre::Livre()
      {    …
       }

      void afficher()
      {
            …
      }
} //Fin du namespace
```

Remarques

- Ne pas attribuer de valeur aux variables à l'intérieur du bloc **namespace** situé dans le fichier **.cpp**.

  ➔ Avoir une erreur de type "Redefinition of...", ce qui signifie qu'on redéfinit une variable déjà définie.

- Pour modifier la variable **"nbrLivre"** du namespace **"Biblio"**, il faut accéder à la variable comme ceci :

```
Biblio::nbrLivre = 12;
```

- Tous les éléments contenus dans un namespace sont **publics**. On ne peut pas modifier leur type d'accès.

Le mot-clé using

- Pour accéder à la variable **"nbrLivre"** du namespace **"Biblio"**:

```
int x =Biblio::nbrLivre ;
```

- Il existe un autre moyen pour utiliser le namespace est le mot-clé **using**.

- On n'a plus besoin d'écrire le nom du namespace devant chaque appel aux variables et aux fonctions d'un espace de nom donnée.

```
using  namespace Biblio;
```

Le mot-clé using

- Vous placez généralement cette ligne **avant le main**.

- Vous pouvez la placer n'importe où. Elle agira alors dans le bloc où elle à été placée.

  ➔Si vous la mettez en dehors de tout bloc (par exemple avant le main), elle agira comme une variable globale statique (c'est-à-dire sur l'ensemble du fichier).

  ➔Si vous la placez dans un bloc "if" par exemple, la condition ne sera valable que pour le if en question.

- Un exemple d'espace de nom est **std**, défini par la **bibliothèque standard**, regroupant en autre les flux standards **cin**, **cout** et **cerr**.