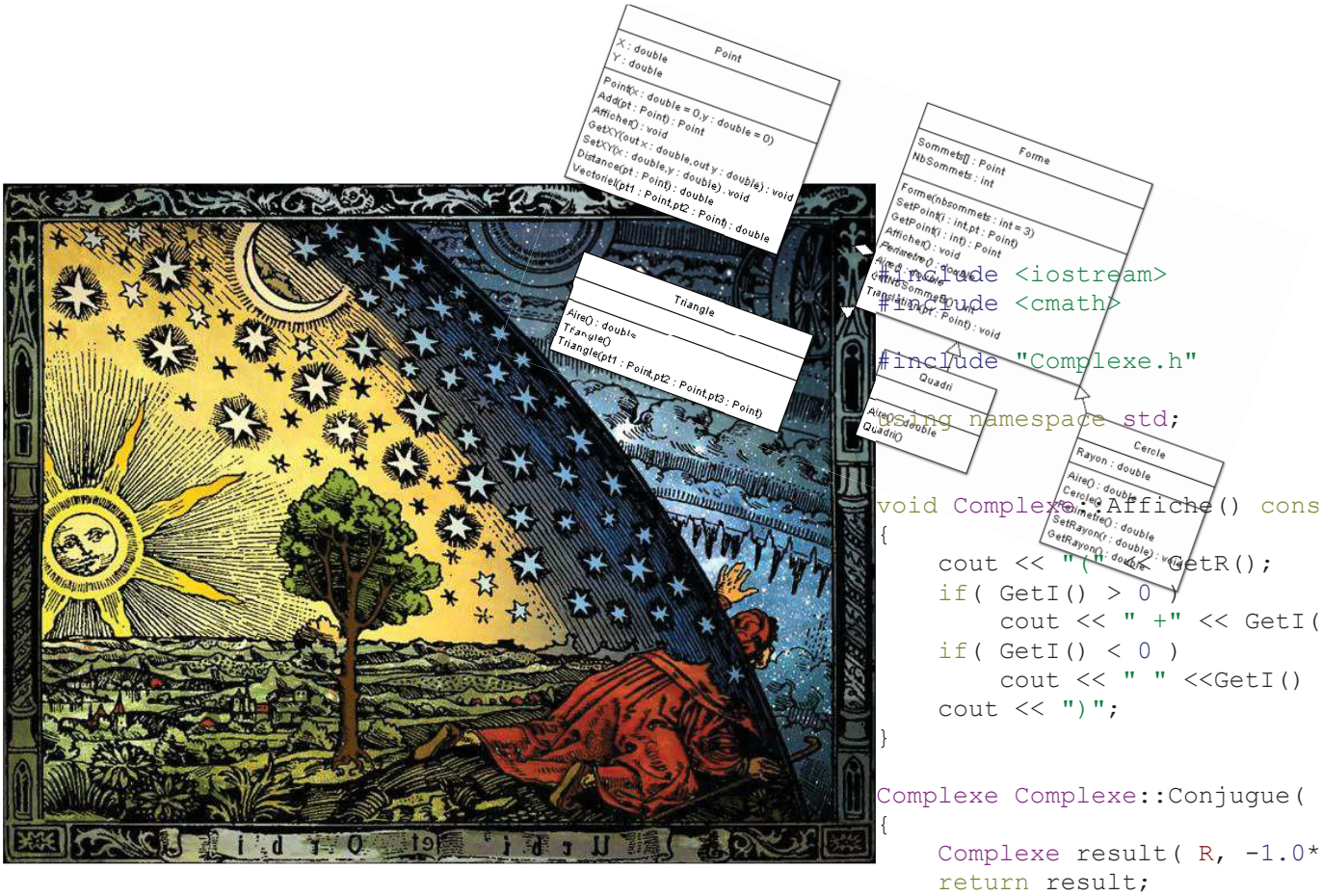
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Département
Génie Electrique

Introduction aux méthodes Orientées Objets

Modélisation avec UML 2.0

Programmation orientée objet en C++



Introduction aux méthodes Orientées Objets

Première partie

Modélisation avec UML 2.0

Programmation orientée objet en C++

Pré-requis:

maitrise des bases algorithmiques (cf. 1ier cycle),
maitrise du C (variables, fonctions, pointeurs, structures)

Thomas Grenier Insa-GE IF3

Bibliographie

- ***Le langage C++**, Grand Livre*, Micro Application, 1998
- ***Le langage C++** 3ième édition*, Bjarne Stroustrup, CampusPress, 1999
- ***Introduction à UML 2.0***, Miles & Hamilton, O'Reilly, 2006
- ***UML 2.0 en concentré***, Pilone & Pitman, O'Reilly, 2006

- Wikipedia
  - http://fr.wikipedia.org/wiki/Programmation_orientée_objet
  - http://fr.wikipedia.org/wiki/Unified_Modeling_Language
  - http://fr.wikipedia.org/wiki/Chronologie_des_langages_de_programmation
- Web divers
  - http://www.langpop.com/ (language popularity)
  - http://www.tiobe.com/ (language index)
  - http://www.nawouak.net/?cat=informatics+lang=fr (informatique et C++)

- Logiciels:
  - StarUML: http://staruml.sourceforge.net/
  - C++ : **QtCreator**, Microsoft Visual C++ Express, Code::Blocks, Eclipse, …

Introduction

- Illustration du concept objet sur un exemple issu des mathématiques

- Brève histoire de la POO
  Origine du C++
- Modélisation Orienté Objet
  Présentation de UML

Socrative 846835

Introduction, concept d'objet

- Les nombres complexes

soit $z_1, z_2 \in \mathbb{C}$

$z_i = a_i + jb_i$

2 données regroupées

Par exemple : $z_1 = 1 + j$

$z_2 = 2 - 4j$

z_1 et z_2 sont indépendants l'un de l'autre

z_1 et z_2 appartiennent au même ensemble (complexe)

L'intérêt des complexes : c'est un corps commutatif (+,x)

$z_3 = z_1 + z_2 = 3 - 3j$

$z_4 = z_1 \times z_2 = 6 - 2j$

➔ Association d'opérations propres aux données

mathématique | *implémentation*

Structures du C

Méthode Orientée objet

en OO:

- z_1, z_2, z_3 et z_4 sont des **objets**
- $\mathbb{C}$ est une **classe**

Introduction POO

- La Programmation Orientée Objet (POO): origine de la démarche « *Orienté Objet* » (OO)

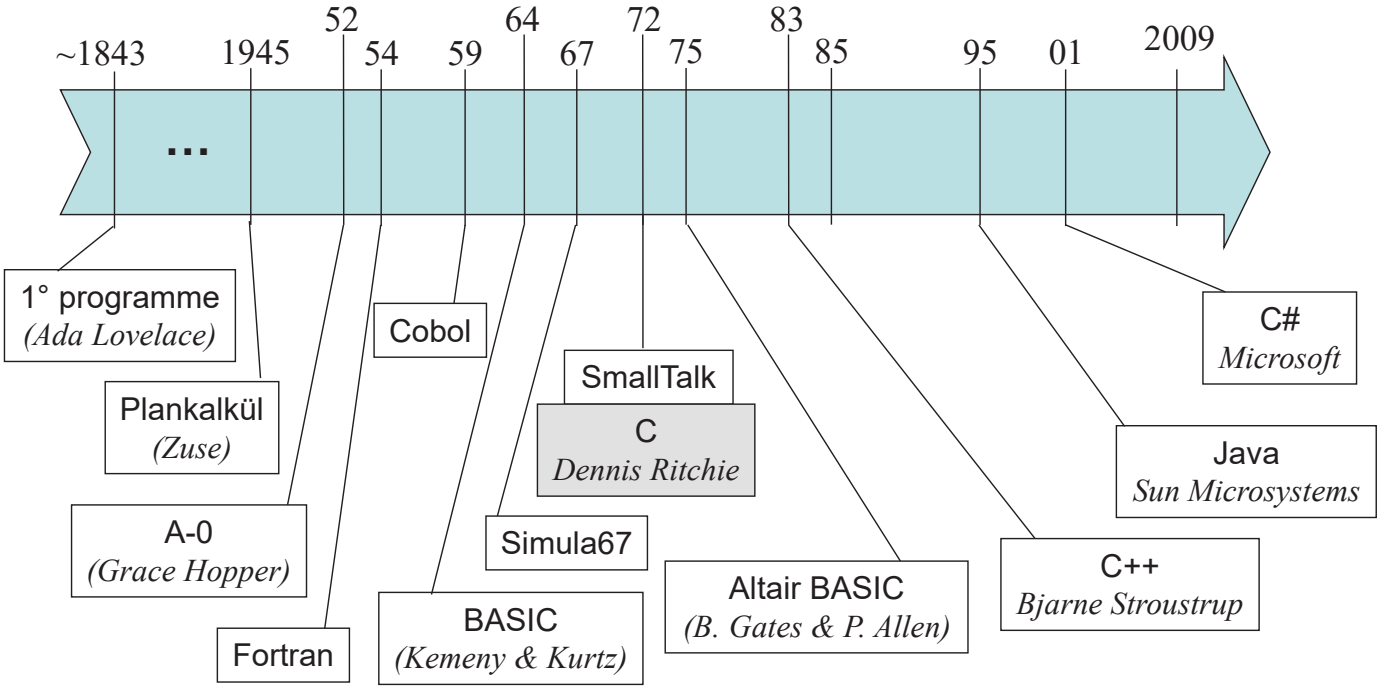
(OO)
\ ~ /

Date du premier programme (langage) informatique ?

Ada Byron, Comtesse de Lovelace
(Ada Lovelace)
1815-1852

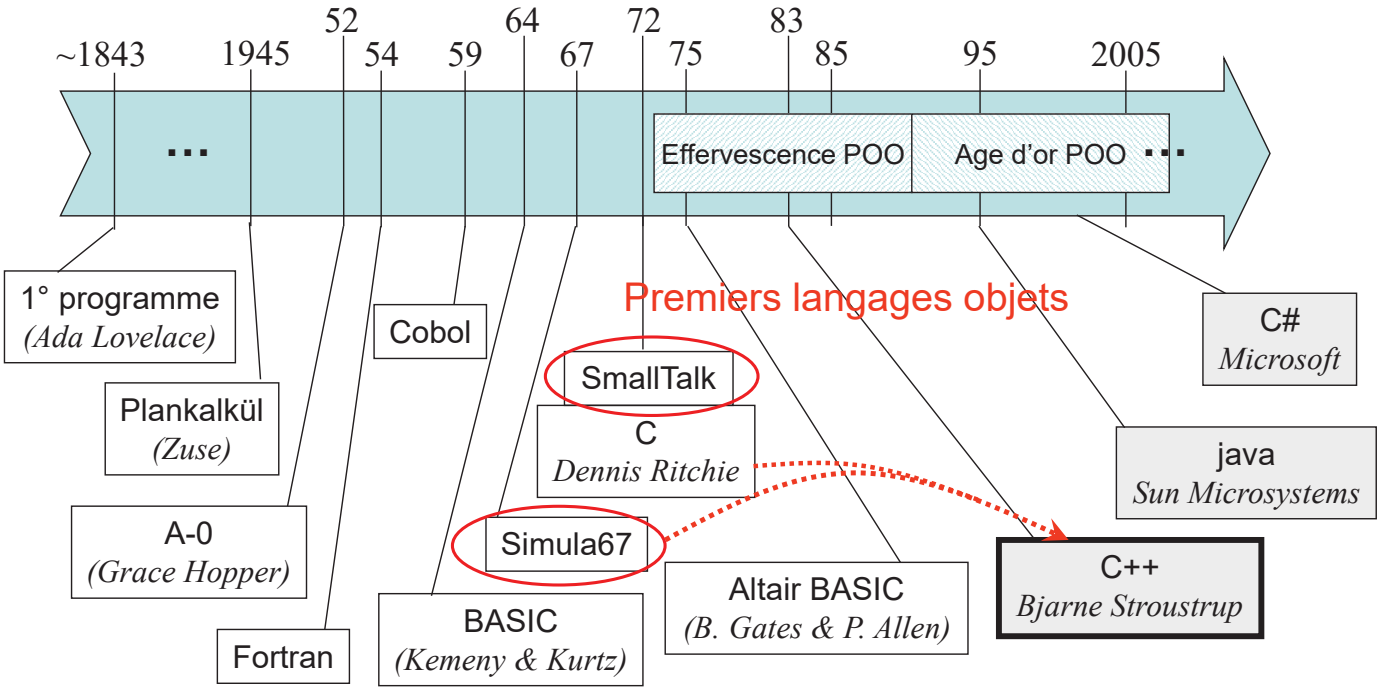
Introduction, POO

- Langages informatiques et OO:



Introduction, POO

- Langages informatiques et OO:



POO, origine de l'effervescence OO

- L'OO en informatique aujourd'hui?
  - Quasi-totalité des applications programmées en OO (POO)
  - Quasi tous les langages intègrent la notion d'objet (même fortran et cobol!)

> L'OO permet de répondre aux exigences actuelles des applications informatiques (type PC):

- Applications conviviales et graphiques (mode « fenêtre »)
- Applications complexes
  - Quantités des exigences client
  - développements, effectifs, temps
- Réutilisation du code
  - Investissement sur le développement
- Maintenance du code
  - Nombreux aller-retour entre validation et conception

Introduction

- Méthodes orientées objet :
  - Analyse (AOO)
  - Modélisation (MOO)
  - Conception (COO)
  - Bases de données (SGBDOO)
  - Programmation (POO)
  - …

➔ **UML**

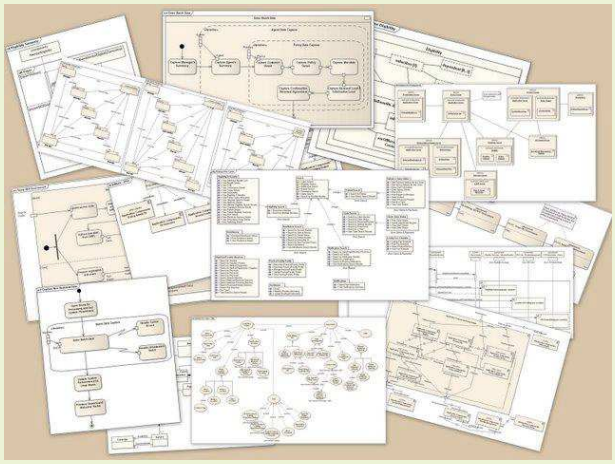
➔ **Méthodologie Orientée Objet**

Model Driven Architecture (MDA),…

Exemple typique: génération automatique de code à partir d'une modélisation UML

Introduction, UML

- UML : Unified Modeling Language
  - Langage de modélisation
    Permet de représenter un système de manière abstraite
  - Langage formel
    Concis, précis et simple
  - Langage graphique
    Basé sur des diagrammes (13 pour UML 2.0)
  - Standard industriel
    Object Management Group, 1997
  - Initié par
    Grady Booch, James Rumbaugh et Ivar Jacobson



Introduction, UML

- UML, outils d'excellence pour
  - Concevoir des logiciels informatiques
  - Communiquer sur des processus logiciels ou d'entreprise
  - Présenter, documenter un système à différents niveaux de détails
- UML, généralisation aux domaines
  - Secteur de la banque et de l'investissement
  - Santé
  - Défense
  - Informatique distribuée, Systèmes embarqués
  - Systèmes complexes (pluri techniques, multi physique)
  - Secteur de la vente et de l'approvisionnement

Introduction, UML

- « Niveaux » de détails d'UML
  - UML comme une esquisse

    (Re)-présenter les points clés, faire l'analyse, mettre en place les premières grandes idées
  - UML comme un plan

    Spécifier en détail un système (diagrammes)

    Souvent associé à des systèmes logiciels et implique une retro-ingénierie pour que le modèle reste synchronisé avec le code
  - Langage de programmation

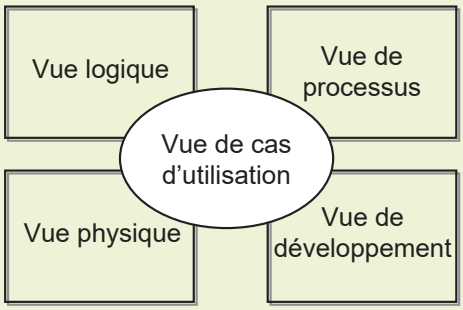
    Passer directement modèle UML → code exécutable

    Tous les aspects du système sont modélisés…

    {UML 2.0 est exécutable}

Introduction, UML

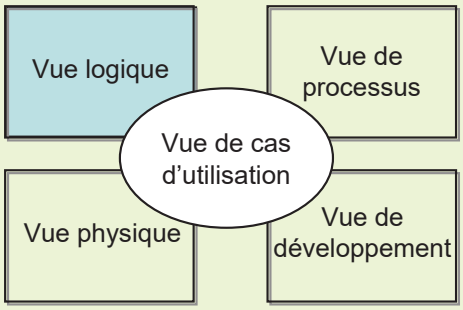
- Modèle de vue
  - organisation de l'analyse OO et des 13 diagrammes UML



Modèle de vue 4+1 de P. Kruchten

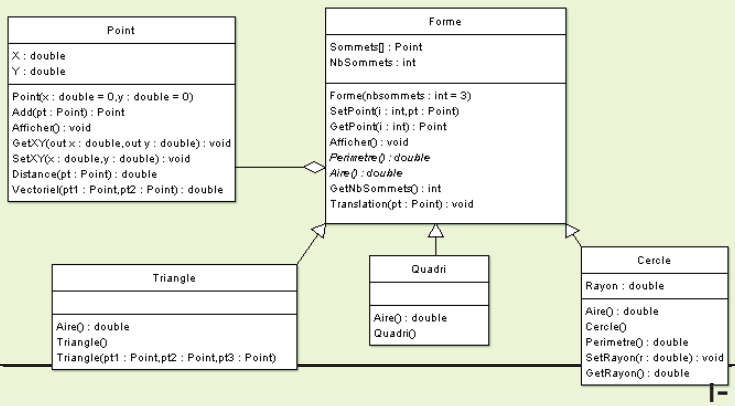
Introduction, UML

- Modèle de vue: Vue logique



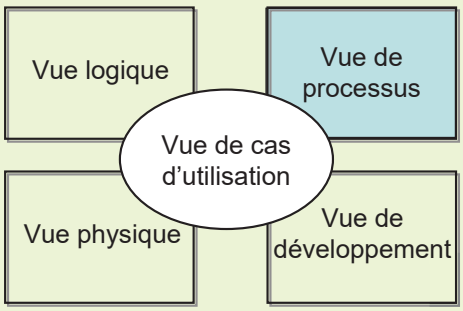
• Donne les descriptions abstraites des parties d'un système
• Sert à modéliser les composants d'un système et leurs interactions

Diagrammes associés:
- **de classes**,
- d'objets,
- de machines d'état
- d'interactions



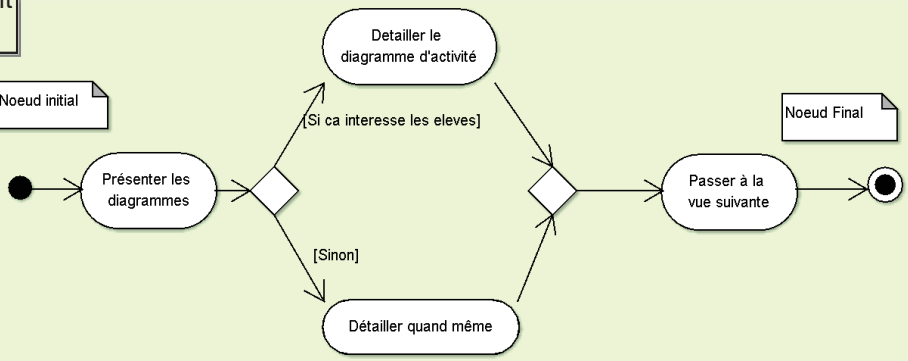
Introduction, UML

- Modèle de vue: Vue de processus



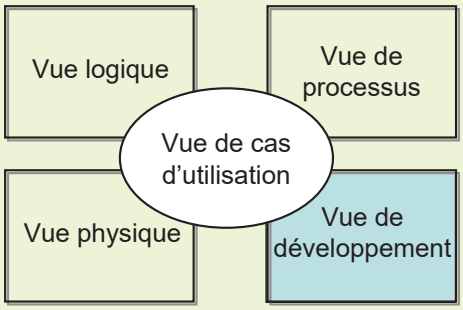
- Décrit les processus du système
- Utile pour la visualisation de l'activité du système

Diagrammes associés:
diagrammes d'activité



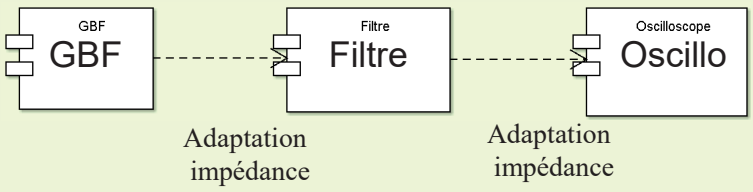
Introduction, UML

- Modèle de vue: Vue de développement



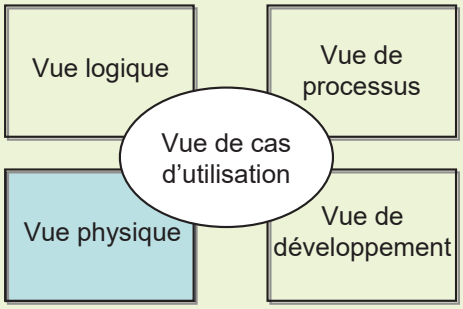
- Décrit l'organisation en modules et composants des parties du système
- Utile pour gérer les différentes couches de l'architecture du système

Diagrammes associés:
- paquetages
- **composants**



Introduction, UML

- Modèle de vue: Vue physique

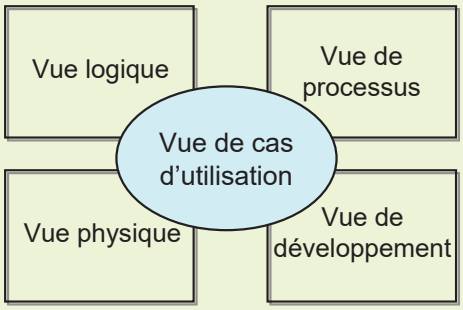


• Décrit comment la conception du système (les 3 autres vue) est mise en œuvre par un ensemble d'entités réelles

Diagramme associé:
diagramme de déploiement

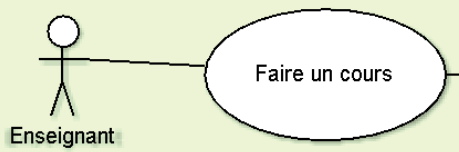
Introduction, UML

- Modèle de vue: Vue de cas d'utilisation



• Décrit la fonctionnalité du système en cours de modélisation, point de vue du monde extérieur
• Nécessaire pour décrire ce que le système est supposé faire
• Toutes les autres vues s'appuient sur celle-ci

Diagrammes associés:
- cas d'utilisation
- des descriptions
- interactions globales



Étudiants dans un amphi

Introduction, résumé

- Illustration du concept objet avec les maths

  Encapsuler données et fonctions spécifiques aux données

- Langage de POO utilisé dans le cours: C++

  Basé sur le C et Simula67 (et Algol68), écrit par Bjarne Stroustrup (1980 « C with Class »; puis 1983: C++)

- Modélisation Orienté Objet : UML

  ➔ Diagramme de classes

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet
III. Encapsulation, concept de classe
IV. Héritage (généralisation)
V. Polymorphisme
VI. Généricité (modèles de classe)

Plan

I. Premier exemple

Des fonctions aux classes… *et UML*

II. Définitions de l'Orienté Objet

III. Encapsulation, concept de classe

IV. Héritage (généralisation)

V. Polymorphisme

VI. Généricité (modèles de classe)

Premier exemple

- Nombres complexes
  - En C, écrire une **procédure** permettant de faire la somme de 2 nombres complexes

```
void Somme(double a1, double b1,
       double a2, double b2, double *a, double *b)
{
  *a = a1 + a2;
  *b = b1 + b2;
}
```

$z_1 = 1 + j$

$z_2 = 2 - 4j$

```
int main(void)
{
  double re,im;
  Somme( 1, 1, 2, -4, &re, &im);
  printf(« %lf +j %lf », re, im);
return 0;
}
```

Premier exemple

- Nombres complexes
  - En C++, écrire une **procédure** permettant de faire la somme de 2 nombres complexes

```
void Somme(double a1, double b1,
       double a2, double b2, double *a, double *b)
{
  *a = a1 + a2;
  *b = b1 + b2;
}
```

$z_1 = 1 + j$

$z_2 = 2 - 4j$

```
int main(void)
{
  double re,im;
  Somme( 1, 1, 2, -4, &re, &im);
  cout<< re << "+j" << im << endl;
return 0;
}
```


Premier exemple

- Nombres complexes, avec structure (du C)
  - Ecrire une **procédure** permettant de faire la somme de 2 nombres complexes

```
typedef struc
{ double Reel;
  double Imag;
} Complexe;
```

⇨ `Complexe` est maintenant un nouveau type

```
void Somme(Complexe z1, Complexe z2, Complexe *z3)
{
  z3->Reel = z1.Reel + z2.Reel;
  z3->Imag = z1.Imag + z2.Imag;
}
```

```
void main()
{ Complexe z1, z2, z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
Somme( z1, z2, &z3);
}
```

Premier exemple

- Nombres complexes, fonction avec structure
  - Ecrire une **fonction** permettant de faire la somme de 2 nombres complexes

```
typedef struc
{ double Reel;
  double Imag;
} Complexe;
```

⇨ `Complexe` est maintenant un nouveau type

```
Complexe Somme(Complexe z1, Complexe z2)
{
Complexe s;
  s.Reel = z1.Reel + z2.Reel;
  s.Imag = z1.Imag + z2.Imag;
return s;
}
```

```
void main()
{ Complexe z1, z2, z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = Somme( z1, z2);
}
```


Premier exemple

- Nombres complexes, du C au C++

Langage C

Langage C++

Données

```
typedef struc
{ double Reel;
  double Imag;
} Complexe;
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe() {Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
  {
    Complexe s;
    s.Reel = Reel + z.Reel;
    s.Imag = Imag + z.Imag;
    return s;
  }
};
```

Champs

constructeur

Méthodes

Fonctions

```
Complexe Somme(Complexe z1, Complexe z2)
{
Complexe s;
  s.Reel = z1.Reel + z2.Reel;
  s.Imag = z1.Imag + z2.Imag;
return s;
}
```

NB: Il s'agit d'un premier exemple qui n'est pas un modèle de programmation à bien des égards

Premier exemple

- Nombres complexes, C++

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

```
#include <iostream>
using namespace std;
…
…
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

Objets

NB: Il s'agit d'un premier exemple qui n'est pas un modèle de programmation à bien des égards

Premier exemple

- Explications de la fonction `main` C++

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

1) 3 objets Complexe (z1, z2, z3) sont créés
   un constructeur de la classe est exécuté

En mémoire :

z1	z2	z3
Reel = 0	Reel = 0	Reel = 0
Imag = 0	Imag = 0	Imag = 0

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Premier exemple

- Explications de la fonction `main` C++

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

1) 3 objets Complexe (z1, z2, z3) sont créés
   un constructeur de la classe est exécuté

2) On modifie les champs de z1 et z2
   Accès direct aux valeurs (`public`...)

En mémoire :

z1	z2	z3
Reel = 1	Reel = 2	Reel = 0
Imag = 1	Imag = -4	Imag = 0

Premier exemple

- Explications de la fonction `main` C++

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

1) 3 objets Complexe (z1, z2, z3) sont créés
   Un constructeur de la classe est exécuté

2) On modifie les champs de z1 et z2
   Accès direct aux valeurs (`public`...)

3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3

En mémoire :

z1	z2	z3
Reel = 1	Reel = 2	Reel = 3
Imag = 1	Imag = -4	Imag = -3

Premier exemple

- Explications de la fonction `main` C++

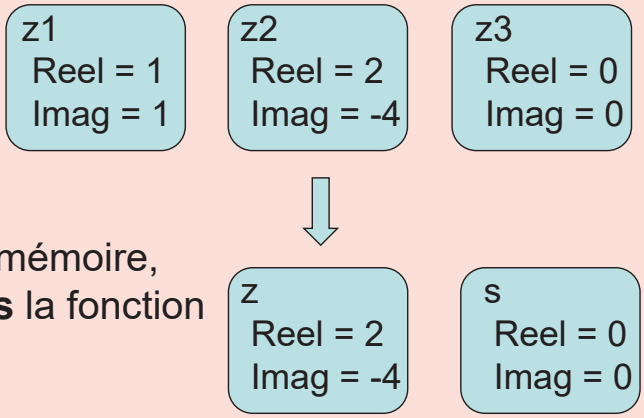
3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

```
z3 = z1.Plus(z2);
```

- En mémoire, à l'appel de la fonction *Plus*

```
z3 = z1.Plus(z2);
```



- En mémoire, **dans** la fonction *Plus*

Premier exemple

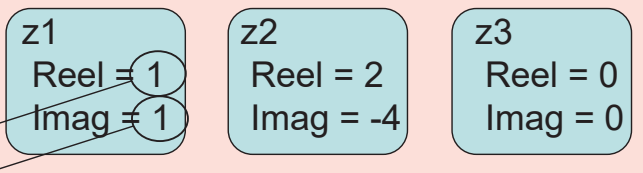
- Explications de la fonction `main` C++

3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3

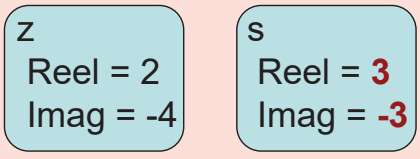
```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

- En mémoire, à l'appel de la fonction *Plus*

```
z3 = z1.Plus(z2);
```



- En mémoire, **dans** la fonction *Plus*



Premier exemple

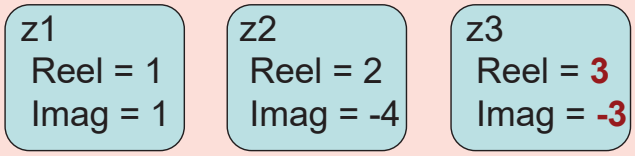
- Explications de la fonction `main` C++

3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3

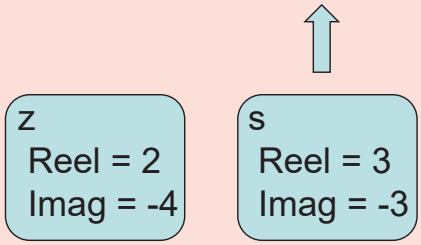
```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

- En mémoire, à l'appel de la fonction *Plus*

```
z3 = z1.Plus(z2);
```



- En mémoire, **dans** la fonction *Plus*



Premier exemple

- Explications de la fonction `main` C++

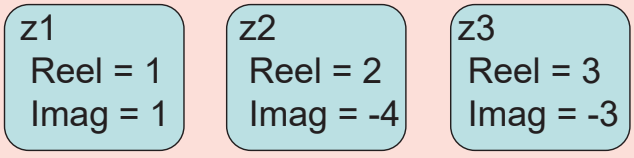
```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
 return 0;
}
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;};
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

1) 3 objets Complexe (z1, z2, z3) sont créés
   Un constructeur de la classe est exécuté
2) On modifie les champs de z1 et z2
   Accès direct aux valeurs (`public`...)
3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3
4) Affichage du résultat:

```
3
-3j
```

En mémoire :

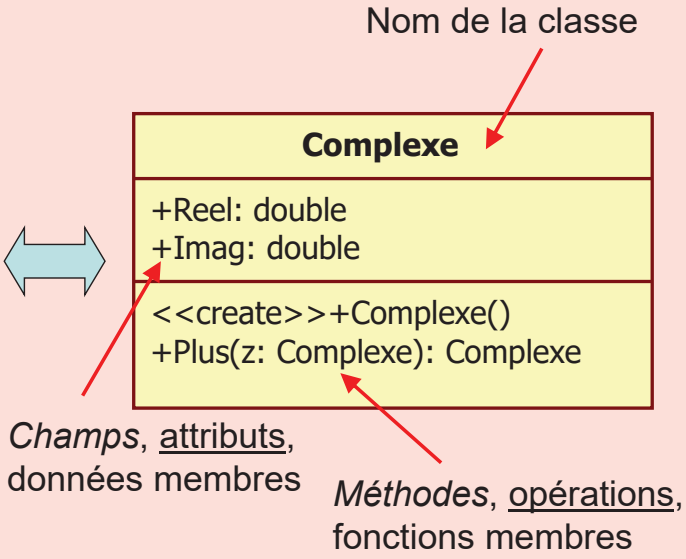


Premier exemple

- Nombres complexes, C++ et UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implémentation



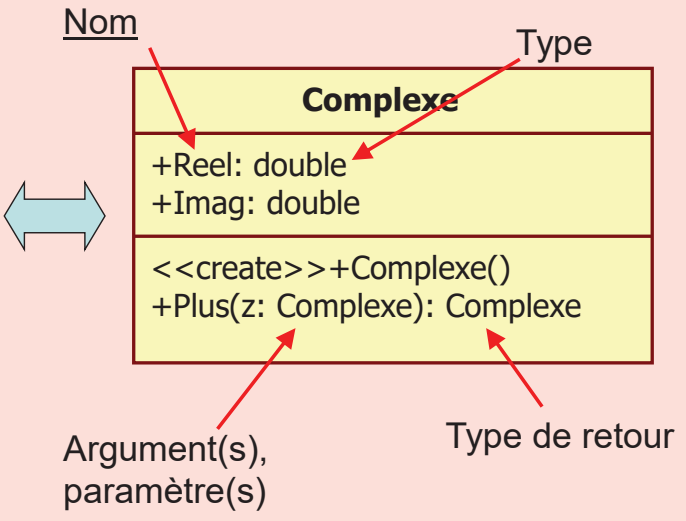
Modélisation

Premier exemple

- Nombres complexes, C++ et UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implémentation



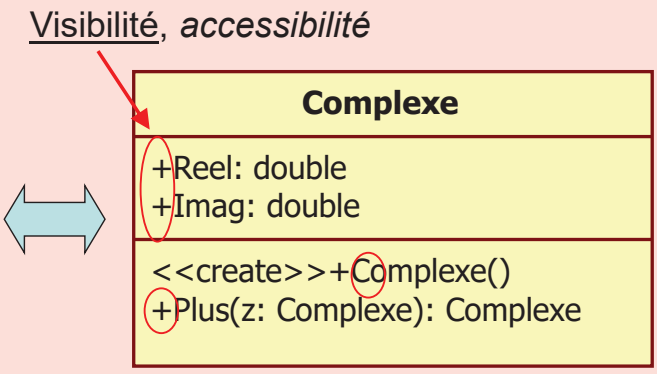
Modélisation

Premier exemple

- Nombres complexes, C++ et UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implémentation



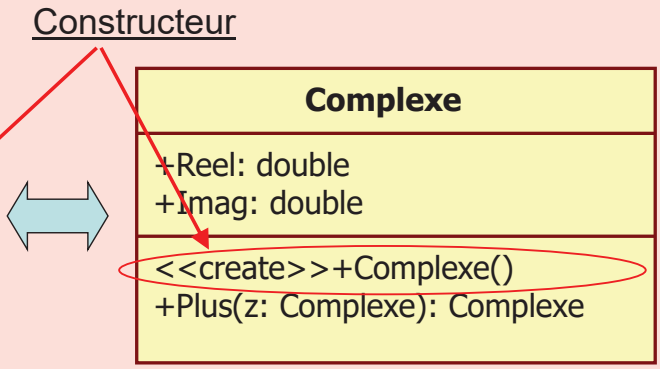
Modélisation

Premier exemple

- Nombres complexes, C++ et UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implémentation



Modélisation

Premier exemple, exo 1

- Etendre la classe Complexe aux fonctionnalités:
  - Conjuguer() : retourner le complexe conjugué
  - Moins(…): soustraction de deux complexes
  - MultiplierPar(…) : produit de deux complexes
  - DiviserPar(…): division de deux complexes
  - Retourner le module du complexe
  - Retourner l'argument du complexe (en radians)

Premier exemple, exo 2

- Modéliser et concevoir un *PorteMonnaie* permettant de gérer une somme d'argent définie au départ. On pourra:
  - ***Ajouter*** de l'argent
  - ***Enlever*** de l'argent, à condition qu'il en reste suffisamment (signaler ce problème à l'utilisateur)
  - ***Afficher*** la somme restant dans le porte monnaie

→ Il faudra interdire l'accès direct à l'attribut gérant la somme d'argent (être obliger de passer par les opérations *Ajouter* et *Enlever* pour le modifier)

Premier exemple, exo 4

- Modéliser une basse (guitare). *(pas plus d'informations…)*

→ La modélisation dépend des besoins du système!
→ Il n'y a donc pas de modélisation unique d'un même concept abstrait

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet … Deuxième partie…

III. Encapsulation, concept de classe

IV. Héritage (généralisation)

V. Polymorphisme

VI. Généricité (modèles de classe)

Introduction aux méthodes Orientées Objets

Deuxième partie

Modélisation avec UML 2.0

Programmation orientée objet en C++

Pré-requis:

maitrise des bases algorithmiques (cf. 1ier cycle),
maitrise du C (variables, fonctions, pointeurs, structures)

Thomas Grenier Insa-GE IF3

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet

III. Encapsulation, concept de classe

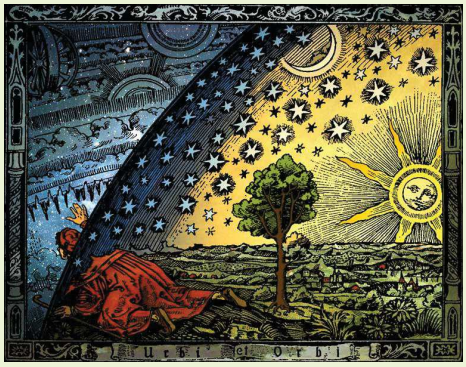
IV. Héritage (généralisation)

V. Polymorphisme

VI. Généricité (modèles de classe)

Méthode Orientée Objet

- Les paradigmes « objet »
  - Encapsulation
    - Regrouper données et opérations
  - Héritage
    - Généralisation de classes
  - Polymorphisme
    - Découle de l'héritage, permet aux classes les plus générales d'utiliser les spécifications des classes plus spécifiques
  - Généricité (extension de l'encapsulation)
    - Modèle d'encapsulation, quelque soit le type des données



Universum, C. Flammarion

Définitions

- Objet
  - Instance de classe
  - Réalisation concrète d'une description abstraite
- Classe
  - Abstraction d'un concept, ne contenant que les détails nécessaires au système
  - Encapsulation des données et des opérations

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet

III. Encapsulation, concept de classe

IV. Héritage (généralisation)

V. Polymorphisme

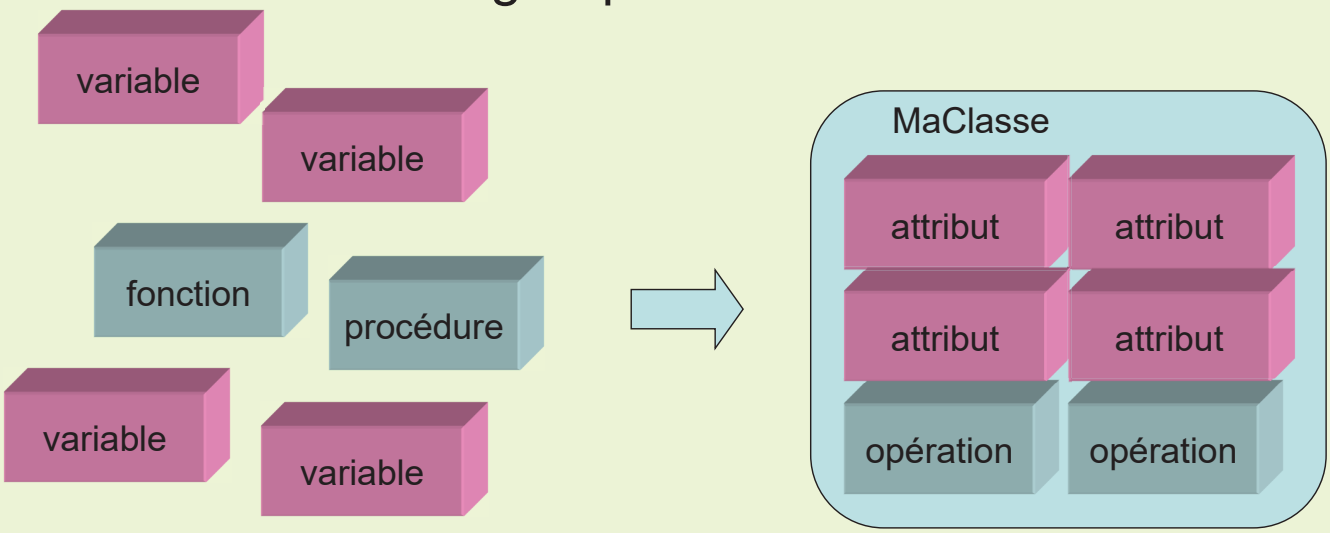
VI. Généricité (modèles de classe)

II – Encapsulation – Plan

- Rappels sur l'encapsulation
- Représentation UML et C++
  - Classes
  - Visibilité
  - Multiplicité
  - Liens entre classes
- Spécificités du C++
  - Fichiers *.h* et *.cpp*
  - Passage de paramètres :
    - Par valeur (copie), par adresse (pointeur), ***par référence***
  - Pointeur ***this***
  - Constructeurs
  - Destructeur
  - Surcharge d'opérateurs
    - Opérateur + , =

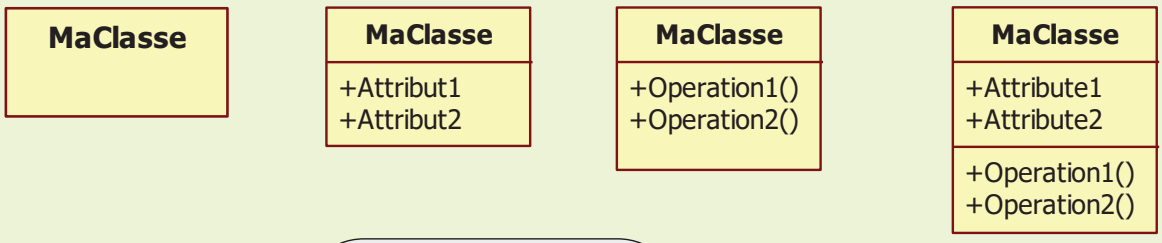
II – Encapsulation – Classes

- Encapsuler = faire une classe
  = regrouper données et fonctions



II – Encapsulation – Classes

- Représentations des classes en UML



- En C++

Tout doit être déclaré !

```
class MaClasse
{
public:
type Attribut1;
type Attribut2;
void Operation1();
void Operation2();
MaClasse();
};
```

… il faudra préciser les types et les paramètres

Un constructeur n'est pas obligatoire en C++

…Premier exemple, exo 2

- Modéliser et concevoir un *PorteMonnaie* permettant de gérer une somme d'argent définie au départ.
  On pourra:
  - Ajouter de l'argent
  - Enlever de l'argent, à condition qu'il en reste suffisamment (signaler ce problème à l'utilisateur)
  - Afficher la somme restant dans le porte monnaie

Classe *PorteMonnaie*

```
class PorteMonnaie
{

public:
double Somme;
void Afficher()
  { cout << "Somme  =" << Somme << endl; }
void Ajouter( double argent )
  { Somme += argent; }
void Enlever( double argent )
  { if( Somme-argent >= 0)
        Somme -= argent;
    else
        cout << "Pas assez d'argent !" << endl;
  }
PorteMonnaie( double somme = 100)
  { Somme = somme;}
};
```

```
int main()
{
PorteMonnaie p(50);
p.Somme = 10;
p.Ajouter( 20 );
};
```

Ok…
Somme est *public*

…Premier exemple, exo 2

- Modéliser et concevoir un *PorteMonnaie* permettant de gérer une somme d'argent définie au départ.
  On pourra:
  - Ajouter de l'argent
  - Enlever de l'argent, à condition qu'il en reste suffisamment (signaler ce problème à l'utilisateur)
  - Afficher la somme restant dans le porte monnaie

→Il faudrait interdire l'accès direct à l'attribut gérant la somme d'argent (être obligé de passer par les opérations *Ajouter* et *Enlever* pour le modifier)

```
void main()
{
PorteMonnaie p(50);
p.Somme = 10;
p.Somme = p.Somme - 20;
}
```


Classe *PorteMonnaie*

```
class PorteMonnaie
{
private:
        double Somme;
public:
void Afficher()
  { cout << "Somme  =" << Somme << endl; }
void Ajouter( double argent )
  { Somme += argent; }
void Enlever( double argent )
  { if( Somme-argent > 0)
        Somme -= argent;
    else
        cout << "Pas assez d'argent !" << endl;
  }
PorteMonnaie( double somme = 100)
  { Somme = somme;}
};
```

Ok, la visibilité privée autorise les **accès** aux champs pour les méthodes de la classe

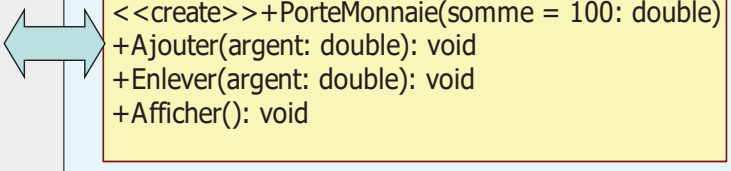
```
int main()
{
PorteMonnaie p(50);
p.Somme = 10;
p.Ajouter( 20 );
};
```

INTERDIT !!
Somme est *privée*

Classe *PorteMonnaie*

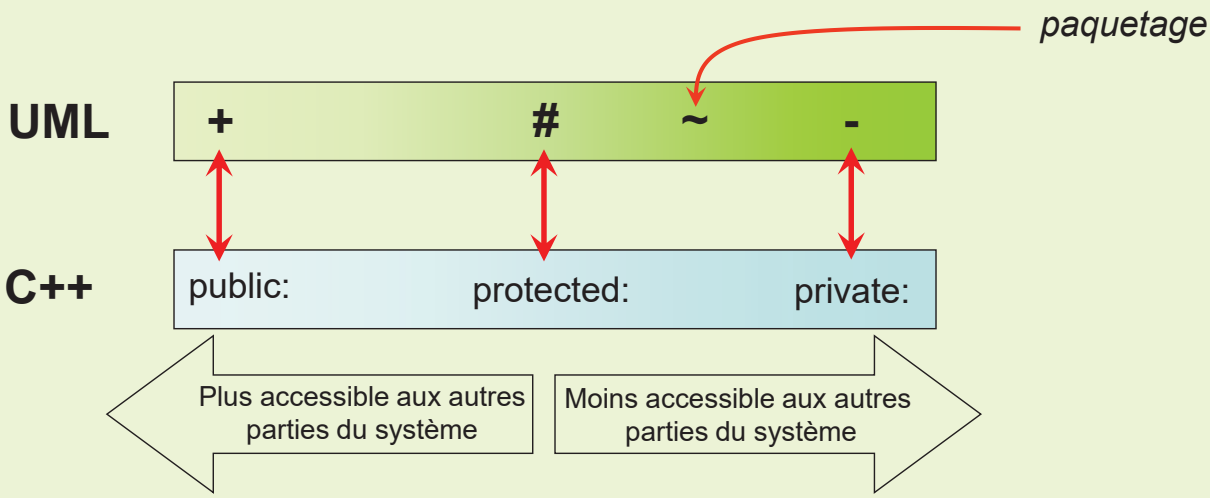
C++ UML

```
class PorteMonnaie
{
private:
        double Somme;
public:
void Afficher()
  { cout << "Somme  =" << Somme << endl; }
void Ajouter( double argent )
  { Somme += argent; }
void Enlever( double argent )
  { if( Somme-argent > 0)
        Somme -= argent;
    else
        cout << "Pas assez d'argent !" << endl;
  }
PorteMonnaie( double somme = 100)
  { Somme = somme;}
};
```



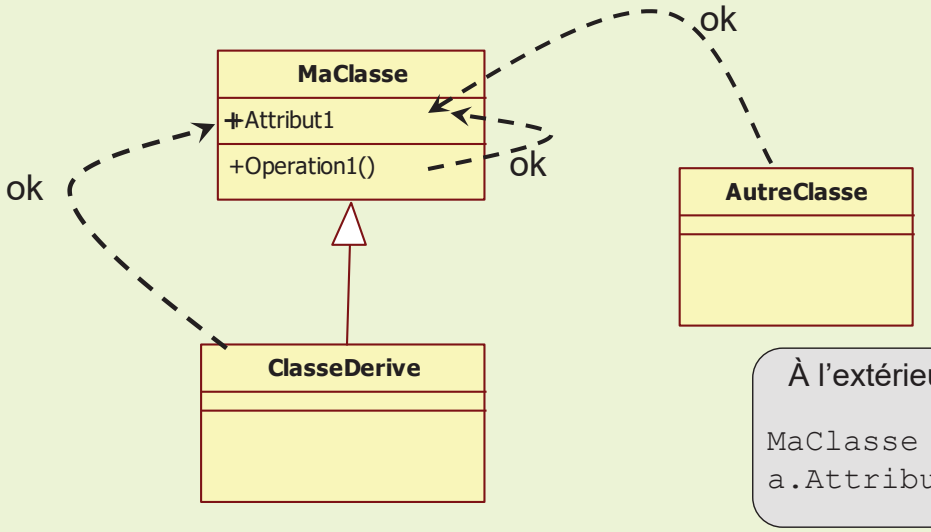
II – Encapsulation – **Visibilité**

- Visibilité des attributs et des opérations
  - 4 modificateurs en UML et 3 en C++



II – Encapsulation – Visibilité

- Visibilité *public*, (+)
  - Accessible directement par un objet
  - Accessible directement par une autre classe

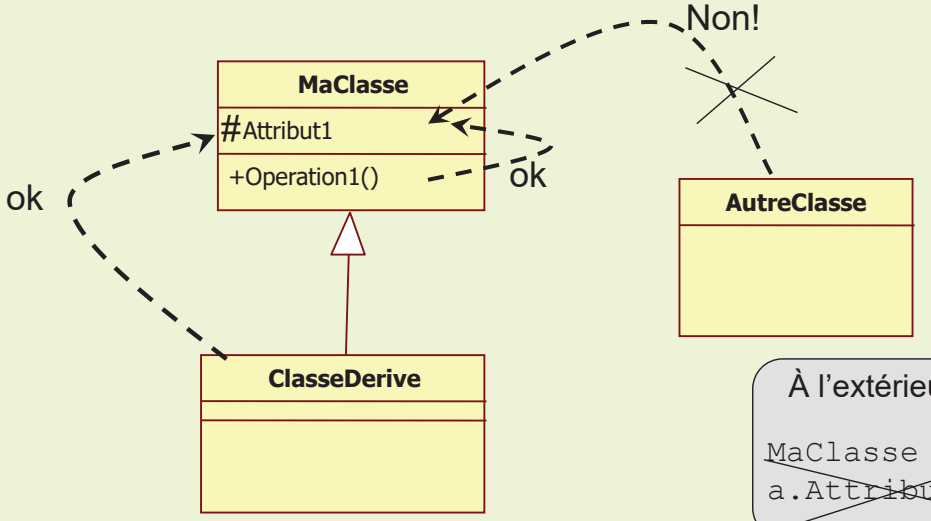


À l'extérieur de la classe

```
MaClasse a;
a.Attribut1=1;//OK
```


II – Encapsulation – Visibilité

- Visibilité *protected*, (#)
  - Non accessible directement par un objet
  - Accessible directement par une classe dérivée

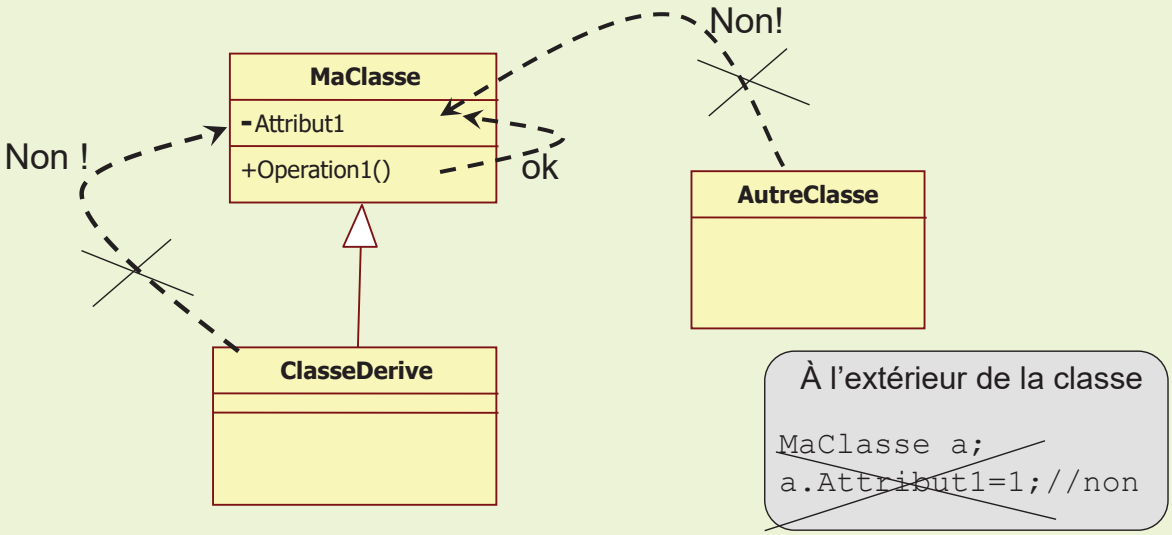


À l'extérieur de la classe

```
MaClasse a;
a.Attribut1=1;//non
```

II – Encapsulation – Visibilité

- Visibilité *private* , (-)
  - Non accessible directement par un objet
  - Non Accessible directement par une autre classe



II – Encapsulation – Visibilité

- Attributs publics…

Faut il utiliser des attributs publics ?

« Donner accès aux attributs d'une classe au reste du système ⇔ laisser la porte de sa maison ouverte et autoriser les passant à entrer sans sonner à la porte».

➔ **Usage désapprouvé par les concepteurs OO**

Mais…
➔ Il est préférable d'éviter les attributs publics,
➔ Usage autorisé pour les constantes (dialogues entre classes)
➔ Usage toléré pour les attributs n'affectant pas le comportement de la classe (ex. *Reel* et *Imag* de la classe *Complexe*)

« … » : D'après Miles et Hamilton

II – Encapsulation – **Nom et type**

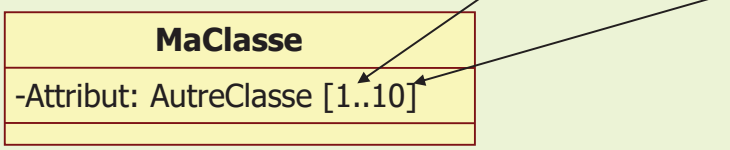
- Nom et type des attributs:
  - **Le nom** des attributs est une chaine quelconque en UML, pas en C++ (convention du langage: accents interdits, première lettre majuscule …)
    - Etre précis et clair pour le choix des noms
    - Deux attributs d'une même classe ne peuvent pas avoir le même nom
  - **Le type** des attributs est spécifié après le nom et les deux points « **:** ». Il peut s'agir :
    - D'un type classique (int, double, char, string, vector …)
    - D'une classe définie dans le système

II – Encapsulation – **Multiplicité**

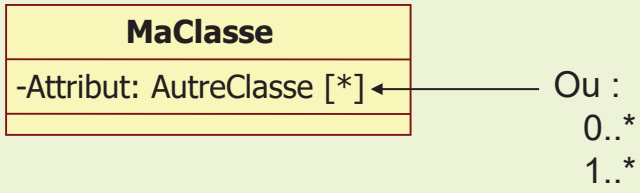
- Un attribut peut représenter un nombre quelconque d'objets de son type:

Déclaration de l'attribut comme un tableau

➔ On connaît le nombre d'éléments minimum et maximum

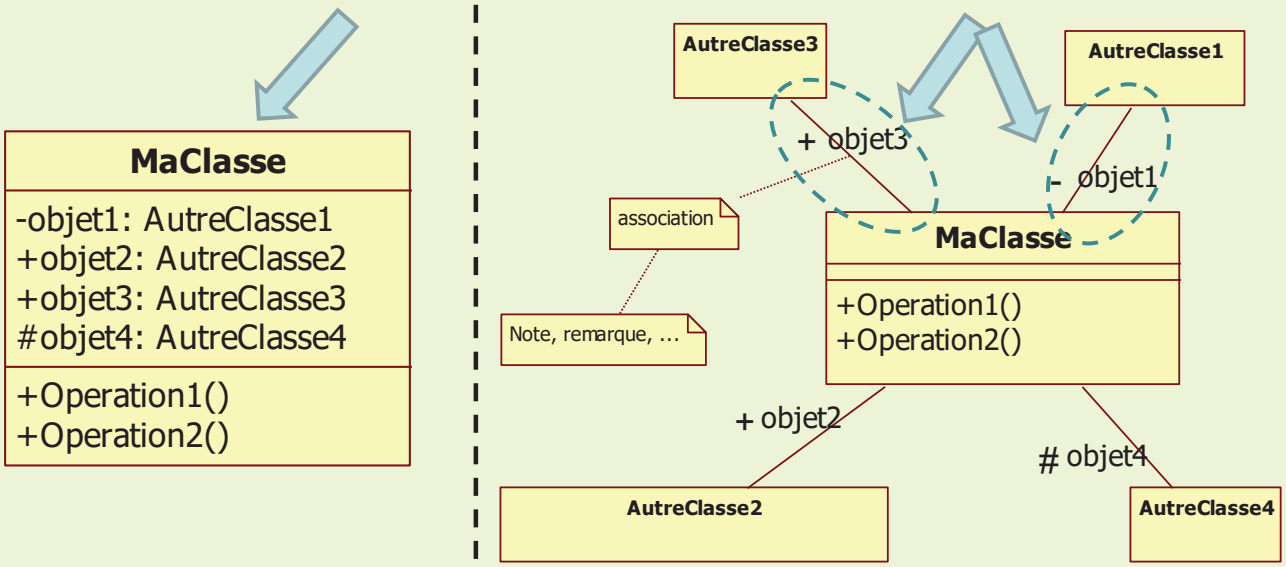


➔ On **ne** connaît **pas** le nombre d'éléments



II – Encapsulation – **Relations entre classes** (UML)

- Attributs en **ligne** vs. attributs par **RELATIONS** association

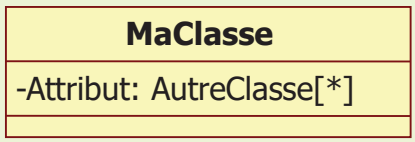


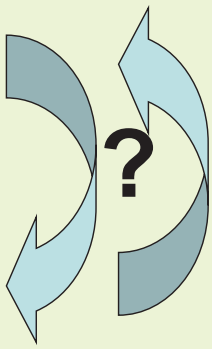
➔Représentation de la même chose (les mélanges de style sont autorisés)
➔Attention à l'objectif du diagramme (clarté *vs.* relation entre classes)

II – Encapsulation – Relations entre classes (UML)

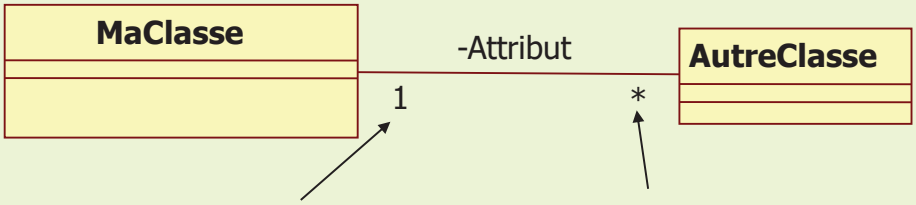
- Cas de la multiplicité

Attribut en ligne





Attribut par association



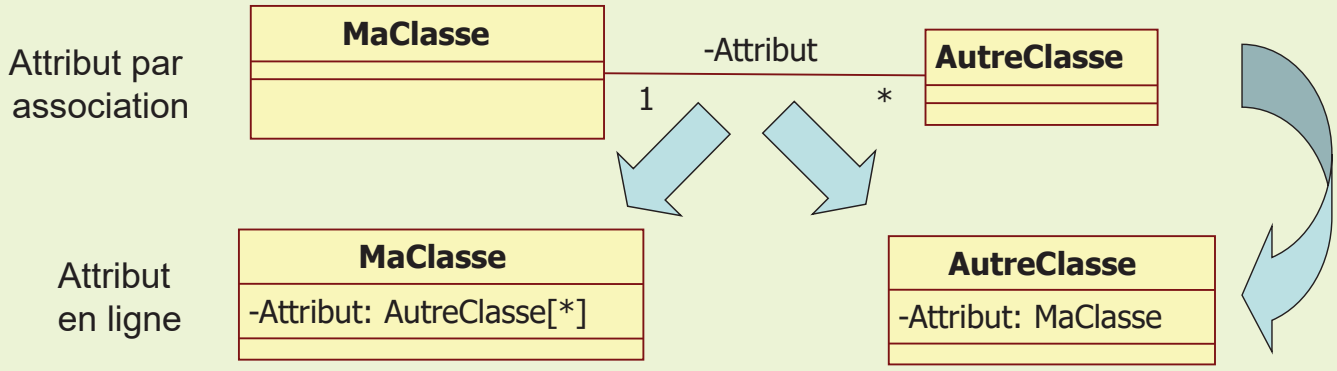
Le champs *Attribut* de *AutreClasse* est associé à **un** unique objet de type *MaClasse*

Le champs *Attribut de MaClasse* est associé à un nombre quelconque d'objets de type *AutreClasse*

➔Il y a un attribut nommé *Attribut* dans chacune des deux classes !

II – Encapsulation – Relations entre classes (UML)

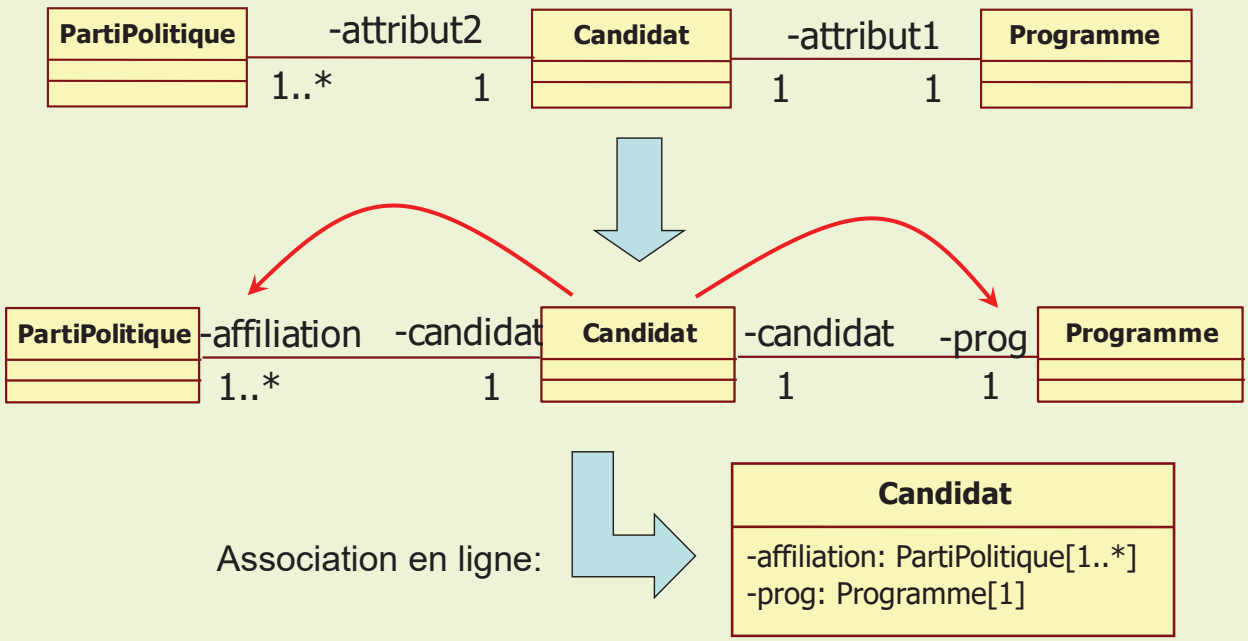
- Cas de la multiplicité



Exemple : *Modéliser les relations entre un candidat et son programme politique, puis avec les partis politiques*

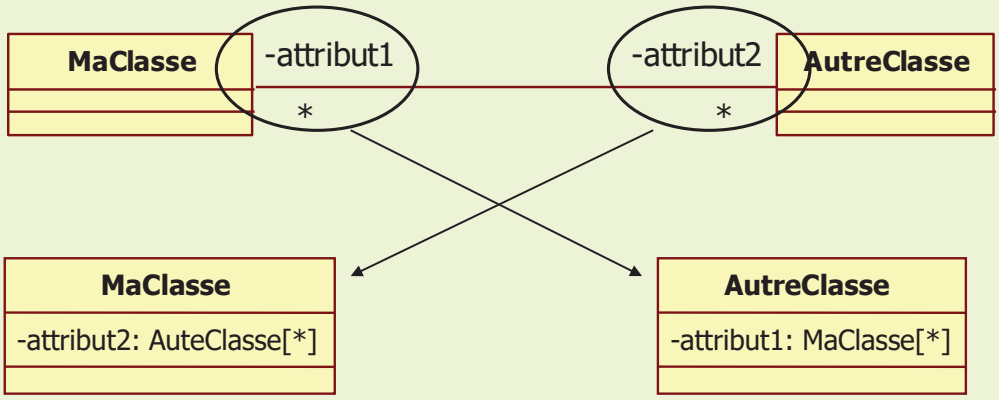
II – Encapsulation – Relations entre classes (UML)

- Exemple: Candidat, programme et partis politiques



II – Encapsulation – Relations entre classes (UML)

- Association

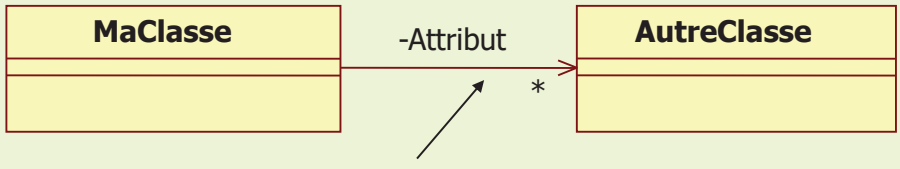


Comment rendre l'association unidirectionnelle ?
➔ rôle de la navigabilité

II – Encapsulation – Relations entre classes

- Navigabilité appliquée aux associations

UML



- *MaClasse* contient *Attribut* de type *AutreClasse*
- *AutreClasse* ne contient pas *Attribut* de type *MaClasse*

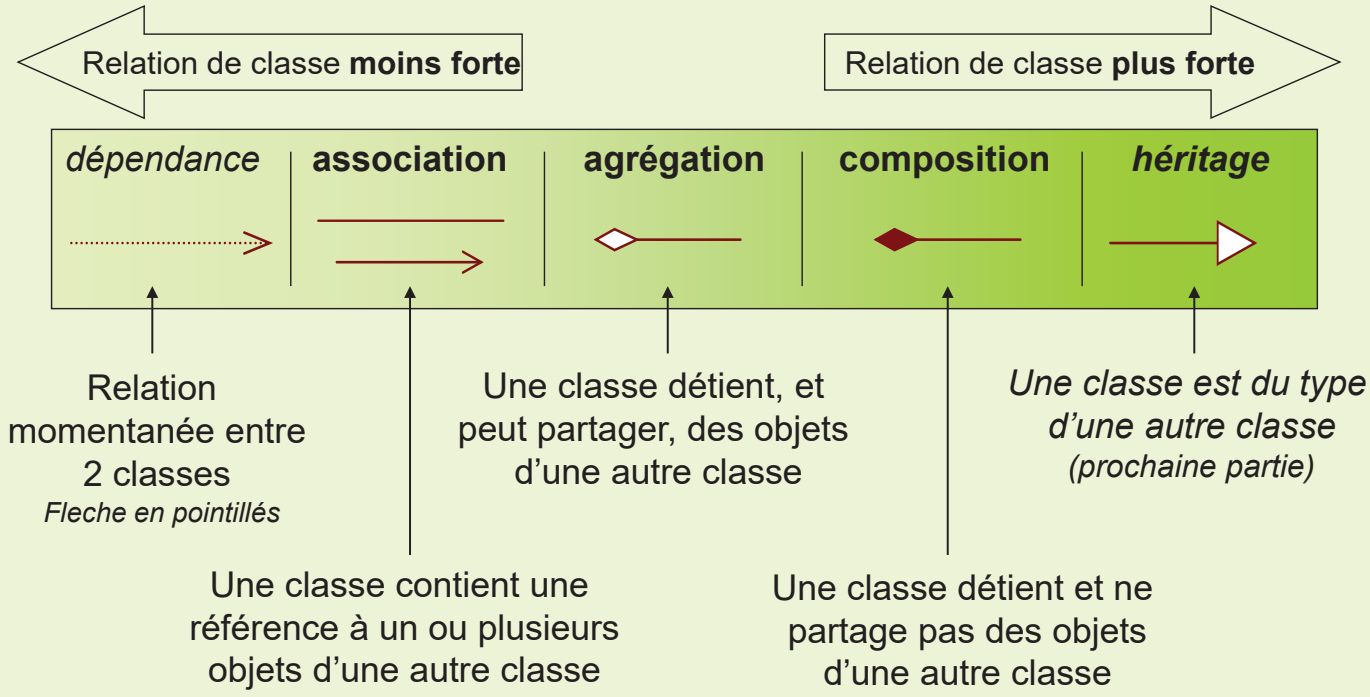
C++

```
class MaClasse
{
private:
  AutreClasse **Attribut;
…
};
```

```
class AutreClasse
{
private:
 // pas de champs Attribut
 // de type MaClasse
…
};
```

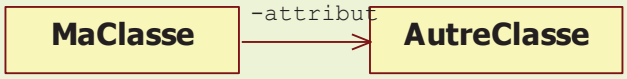
II – Encapsulation – Relations entre classes

- UML : 5 types de relations entre classe



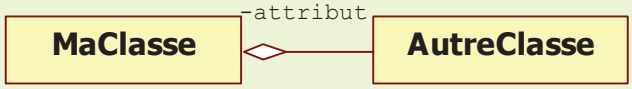
II – Encapsulation – Relations entre classes (UML)

- Association



  - *MaClasse* contient une référence à un objet de type *AutreClasse* grâce à *attribut,* elle ne modifie pas *attribut*
  - *MaClasse* **fonctionne avec un objet** de *AutreClasse*

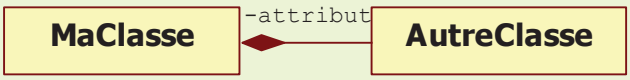
- Agrégation



  - *MaClasse* détient l'objet *attribut* et peut le partager
  - *MaClasse* est propriétaire du contenu de *attribut*

  Modification, création, suppression… d'éléments possibles

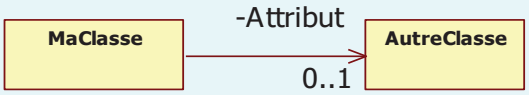
- Composition



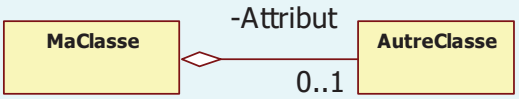
  - *MaClasse* détient et gère l'objet *attribut*: création et destruction
  - *attribut* **n'est pas partagé** avec une autre classe
  - *attribut* est une partie interne de *MaClasse*

II – Encapsulation – Les relations en C++

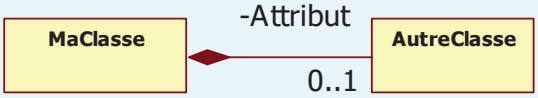
- Association



- Agrégation



- Composition

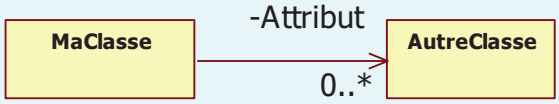



```
class MaClasse
{
private: // ou autre
  AutreClasse *Attribut;
…
};
```

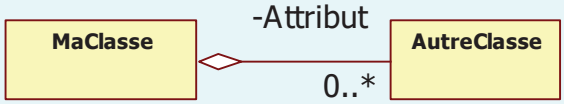
```
class MaClasse
{
private: // ou autre
  AutreClasse Attribut;
…
};
```


II – Encapsulation – Relations et multiplicité en C++

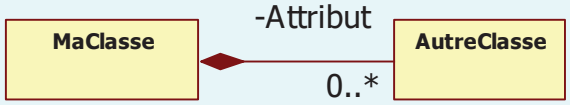
- Association



- Agrégation



- Composition



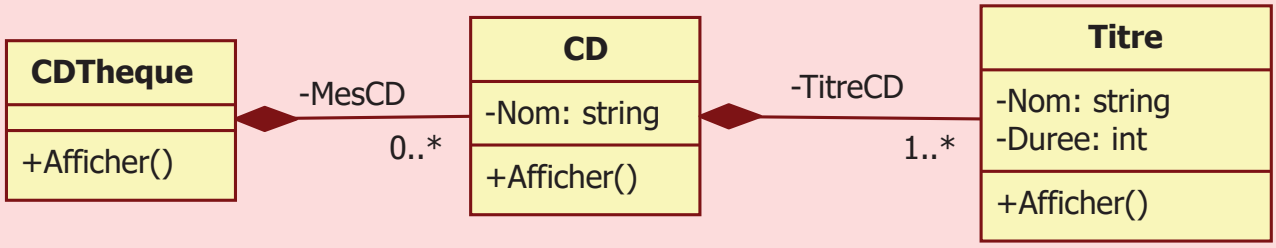

```
class MaClasse
{
private: // ou autre
  AutreClasse **Attribut;
…
};
```

```
class MaClasse
{
private: // ou autre
  AutreClasse *Attribut;
…
};
```

```
class MaClasse
{
private: // ou autre
  AutreClasse Attribut[10];
…
};
```

Exemple de relations: CDtèque

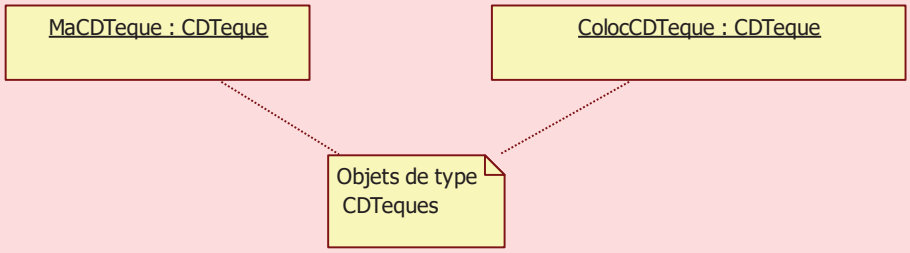
- Modéliser votre « CD thèque » étant donné que vous avez plusieurs CD et que sur chaque CD il y a plusieurs titres.
  On souhaite:
  - Afficher le nom de tous les CD de la CDthèque
  - Afficher tous les titres d'un CD et la durée de chaque titre

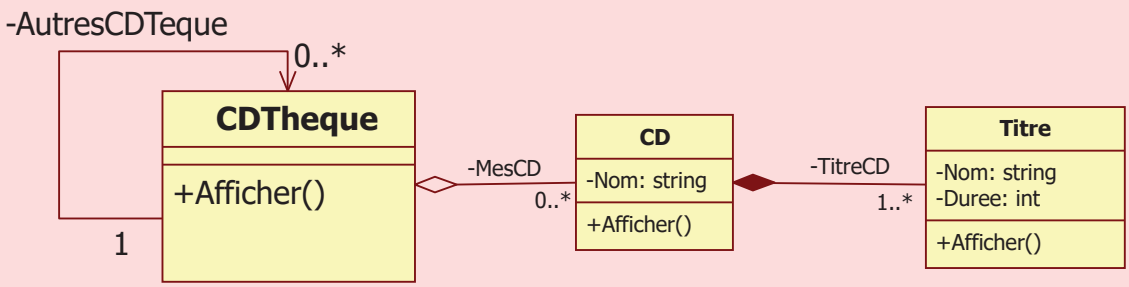


Exemple de relations: CDtèque

- Modifier votre « CD thèque » :
  vous avez un colocataire avec une autre CDthèque

Diagramme d'objets





Exemple de relations: CDtèque

- C++ de l'exemple CDteque

```
class CDTeque
{
private:
  CD *MesCD;
  CDTeque
    **AutresCDTeque;
public:
  CDTeque();
  void Afficher();
};
```

```
class CD
{
private:
  string Nom;
  Titre *TitreCD;
public:
  CD();
  void Afficher();
};
```

```
class Titre
{
private:
  string Nom;
  int Duree;
public:
  Titre();
  void Afficher();
};
```

Que manque t'il à ces classes pour pouvoir fonctionner ?

Le nombre d'éléments des tableaux !

Des méthodes d'ajout/suppression/modification d'éléments CDTeque, CD et Titre

À faire … ou utiliser ***vector<>***

II – Encapsulation – C++

- En C++, pour la lisibilité, séparation de:
  - La définition de la classe ➔ ***.h***
  - L'implémentation des méthodes ➔ ***.cpp***

.h

```
class Complexe
{
public:
  double Reel;
  double Imag;

  Complexe();
  Complexe Plus(Complexe z);
};
```

.cpp

```
Complexe::Complexe()
  {
  Reel=0;
  Imag=0;
  }
Complexe Complexe::Plus(Complexe z)
  {
   Complexe s;
    s.Reel = Reel + z.Reel;
    s.Imag = Imag + z.Imag;
   return s;
  }
```

opérateur d'appartenance

II – Encapsulation – C++

Complexe.h

```
#ifndef _Complexe_h_
#define _Complexe_h_

#include …… //si besoin

class Complexe
{
public:
  double Reel;
  double Imag;

  Complexe();
  Complexe Plus(Complexe z);
};
#endif
```

Complexe.cpp

```
#include "Complexe.h"

#include …… //si besoin

Complexe::Complexe()
  {
  Reel=0;
  Imag=0;
  }
Complexe Complexe::Plus(Complexe z)
  {
   Complexe s;
    s.Reel = Reel + z.Reel;
    s.Imag = Imag + z.Imag;
   return s;
  }
```


Classe complexe

Complexe.h

```
#ifndef _Complexe_h_
#define _Complexe_h_

class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe();
  Complexe Plus(Complexe z);
  Complexe MuliplierPar
        (Complexe z);
  Complexe DiviserPar
        (Complexe z);
  Complexe Conjuger();
  void Affiche();
  void AffichePolar();
};
#endif
```

Complexe.cpp

```
#include "Complexe.h"
#include <cmath>

Complexe::Complexe() { … }
Complexe Complexe::Conjuguer() {… }
Complexe Complexe::Plus(Complexe z) {… }
void Complexe::Affiche() {…}
Complexe Complexe::MultiplierPar(Complexe
z) {… }

Complexe Complexe::DiviserPar(Complexe z)
{
Complexe s;
s = MultiplierPar( z.Conjuguer() );
s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
return s;
}
```

Classe complexe

Complexe.h

```
#ifndef _Complexe_h_
#define _Complexe_h_

class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe();
  Complexe Plus(Complexe z);
  Complexe MuliplierPar
        (Complexe z);
  Complexe DiviserPar
        (Complexe z);
  Complexe Conjuger();
  void Affiche();
  void AffichePolar();
};
#endif
```

Complexe.cpp

```
#include "Complexe.h"
#include <cmath>
 …
Complexe Complexe::DiviserPar(Complexe z)
{…}

void Complexe::AffichePolar()
{
cout << sqrt(Reel * Reel + Imag * Imag);
cout << "ei(" <<  atan(Imag / Reel);
cout  << endl;
}
```


C++ - passage de paramètres

- Passage de paramètres par valeur (copie)
  - Idem que langage C
- Passage de paramètres par adresse (pointeur)
  - Idem que langage C
- Passage de paramètres par référence
  - Nouveau !

C++ - passage de paramètres

- Passage de paramètres par valeur (copie)

```
void fonction(double a)
{
  a = 3;
}
```

```
void main(void)
{
  double x = 1;
  fonction(x);
  cout << x;
}
```

→ x vaut 1

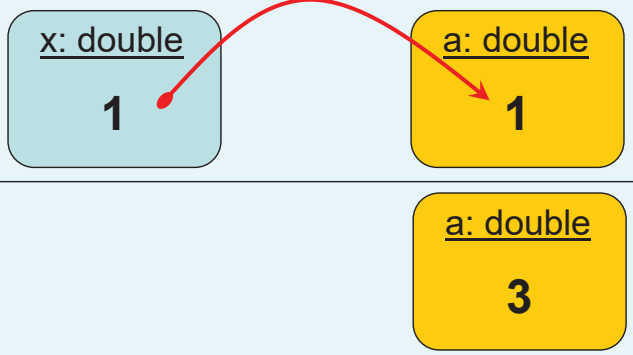
En mémoire :

Copie des valeurs

A l'appel de **`fonction(x);`**

⇩

Dans *fonction*: **`a = 3;`**



➔ Pas de modification de la valeur de x dans la fonction

C++ - passage de paramètres

- Passage de paramètres par adresse (pointeur)

```
void fonction(double *a)
{
  *a = 3;
}
```

```
void main(void)
{
  double x = 1;
  fonction(&x);
  cout << x;
}
```

→ x vaut 3

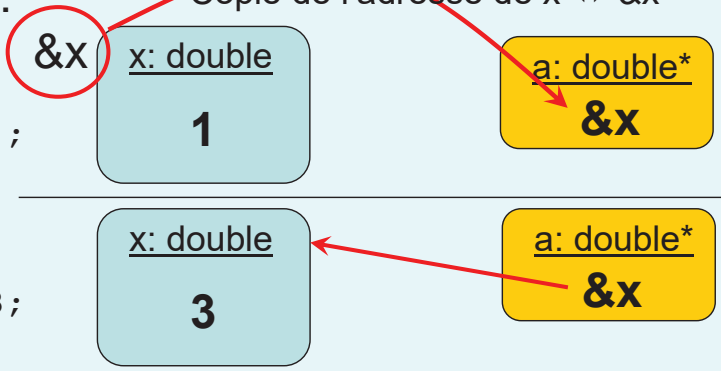
En mémoire :

Copie de l'adresse de x ⇔ &x

A l'appel de **`fonction(&x);`**

⇩

Dans *fonction*: **`*a = 3;`**



Mettre 3 dans ce que pointe a

➔ Modification de la valeur de x par la fonction

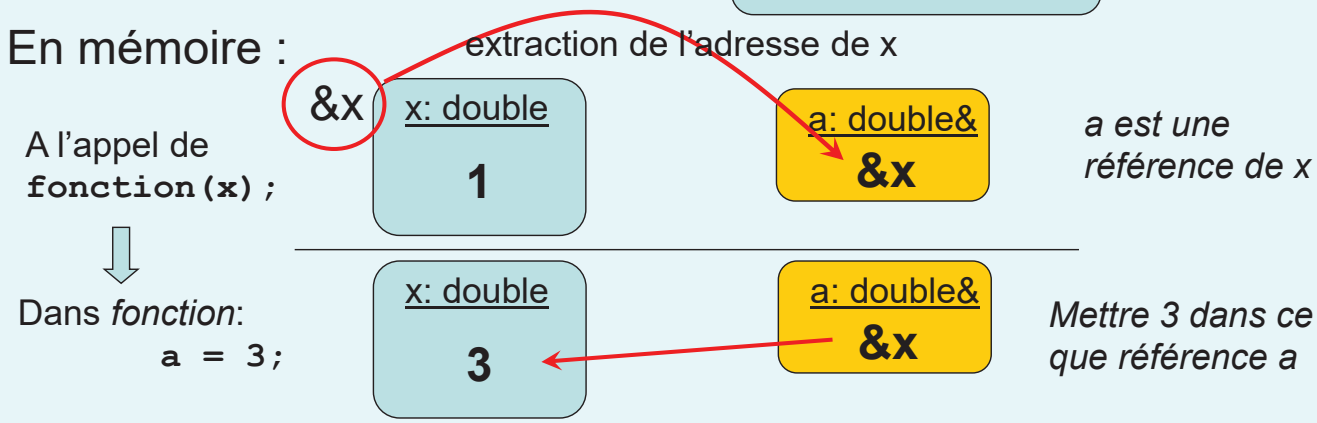
C++ - passage de paramètres

- Passage de paramètres par référence

```
void fonction(double &a)
{
  a = 3;
}
```

```
void main(void)
{
  double x = 1;
  fonction(x);
  cout << x;
}
```

→ x vaut 3



➔ Modification de la valeur de x par la fonction

C++ - passage de paramètres

- Passage de paramètres par valeur (copie)
  - **Copie** des valeurs des variables dans les paramètres
    2 espaces mémoires différents
- Passage de paramètres par adresse (pointeur)
  - Copie de l'adresse d'une variable (pas des valeurs)
  - Modification de la syntaxe pour l'accès au contenu
  - Permet de passer et manipuler des tableaux
- Passage de paramètres par référence
  - Extraction de l'adresse d'une variable
    - création d'une référence, pas de copie des valeurs
  - Transparent à l'utilisation et à la programmation d'une fonction (pas de changement de syntaxe)
    - Simplement préciser les paramètres passés par référence
  - Ne permet pas de passer ni de manipuler des tableaux

C++ - Passage de paramètres

- Comparaisons

	copie	**pointeur**	**référence**
rapidité	lent	rapide	rapide
utilisation	très simple	attention aux règles	simple
manipulation de tableau	non	oui	non
mémoire utilisée/param.	un objet	une adresse	une référence
modification paramètres	non	oui	= oui
polymorphisme	*non*	*oui*	*oui*

C++ - mot clé « const »

- But du mot clé: préciser et garantir l'interdiction de modification d'un objet
  - Applicable aux paramètres (pointeur et référence)
  - Applicable aux opérations des classes

```
void fonction(const double &a)
{
  a = 3;
}
```

Erreur de compilation :
la référence « a » est constante
et ne peut pas être modifiée

```
void Complexe::AffichePolar() const
{ … }
```

Cette opération ne modifie pas les attributs de l'objet courant

Classe complexe

Complexe.h

Version avec référence et mot clé *const*

```
#ifndef _Complexe_h_
#define _Complexe_h_

class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe();
  Complexe(const double &re, const double &im);
  Complexe Plus(const Complexe &z) const;
  Complexe MuliplierPar(const Complexe &z) const;
  Complexe DiviserPar(const Complexe &z) const;
  Complexe Conjuger() const;
  void Affiche() const;
  void AffichePolar() const;
};
#endif
```


Classe complexe

Complexe.cpp

Version avec référence et mot clé *const*

```
#include "Complexe.h"
#include <cmath>
…
Complexe Complexe::DiviserPar(const Complexe &z) const
{
Complexe s;
s = MultiplierPar( z.Conjuguer() );
s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
return s;
}

void Complexe::AffichePolar() const
{
cout << sqrt(Reel * Reel + Imag * Imag);
cout << "ei(" <<  atan(Imag / Reel);
cout  << endl;
}
```

Doit être déclarée *const*

Classe complexe

Complexe.cpp

Version avec référence et mot clé *const*

```
#include "Complexe.h"
#include <cmath>
…
Complexe Complexe::DiviserPar(const Complexe &z) const
{
Complexe s;
s = MultiplierPar( z.Conjuguer() );
s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
return s;
}

void Complexe::AffichePolar() const
{
cout << sqrt(Reel * Reel + Imag * Imag);
cout << "ei(" <<  atan(Imag / Reel);
cout  << endl;
}
```

Doit être déclarée *const*

C++ : pointeur ***this***

- *this* est un **pointeur privé** automatiquement inclus dans les classes (il n'apparaît pas dans la définition de la classe)
- *this* pointe toujours l'objet courant
  - pointeur du type de la classe (ex.: « Complexe * »)
  - initialisation automatique sur l'objet courant (ou appelant)

```
Complexe::Complexe()
{
  this->Reel=0; // ⇔ Reel=0;
  this->Imag=0; // ⇔ Imag = 0;
}
Complexe Complexe::DiviserPar(const Complexe &z) const
{
  Complexe s;
  s = this->MultiplierPar( z.Conjuguer() );
  s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
  s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
  return s;
}
```

this permet d'accéder à **tous** les éléments de l'objet courant

Rappel : (*this).Reel ⇔ this->Reel

C++ : constructeurs

- En C++, un constructeur :
  - est nécessaire à la classe (un constructeur minimum par classe, un constructeur *implicite* sans paramètre est ajouté si aucun n'est présent, un constructeur de copie est aussi toujours présent)
  - porte le nom de la classe
  - ne retourne rien de rien (même pas void !)
  - est surchargeable : Une classe peut avoir plusieurs constructeurs,
    Le type et le nombre des paramètres passés lors de la construction d'un objet permettent de désigner sans ambiguïté le constructeur à exécuter
  - est exécuté automatiquement et uniquement à la création d'un objet :

MaClasse a; // Execution du constructeur de MaClasse

NB: un constructeur n'est pas forcement public (ex. des architectures basées sur les pointeurs objets)

C++ : constructeurs

- Rôle du constructeur :
  - Doit permettre d'initialiser l'objet créé de façon à le rendre utilisable par le système. En clair initialise (tous) les champs de l'objet.
  - (le compilateur se charge de la création en mémoire des champs statiques)
- Remarque : on peut passer des instructions spécifiques lors de la création des champs statiques

```
class Complexe
{public:
  double Reel;
  double Imag;
  Complexe();
   …
};
```

Complexe.cpp

```
Complexe::Complexe():Reel(0), Imag(0){ }
```

Reel et Imag seront initialisés à 0
Attention à l'ordre !

C++ : constructeurs

- Les constructeurs du C++ :
  - Constructeur sans paramètres (ou « par défaut »)
    - Il est conseillé d'en faire un pour les définitions d'objet :
      MaClasse a;
    - Déclaration : `MaClasse();`
  - Constructeur(s) « utilisateur »
    - Autant que nécessaire, attention aux ambiguïtés d'exécution
    - **Surcharge le constructeur par défaut implicite**
    - Déclaration (exemples) :
      - `MaClasse( int a );`
      - `MaClasse( double x1, double x2, double x3=0);`
      - …
  - Constructeur de copie
    - S'il n'existe pas, automatiquement réalisé par le compilateur (copie des valeurs de tous les champs, *pb* avec les pointeurs…)
    - Déclaration : `MaClasse( const MaClasse &a);`

C++ : destructeur

- En C++, un destructeur :
  - n'est pas obligatoire
    - Si la classe n'en possède pas, le compilateur en ajoute un « simple » (donc problème si allocations d'attributs dynamiques)
    - Obligatoire dans le cas de gestion de l'allocation d'attributs dynamiques (pointeurs, tableaux dynamiques, listes, …)
  - porte le nom de la classe précédé de ~, ex: `~MaClasse();`
  - ne retourne rien de rien (même pas void)
  - n'a pas de paramètre
  - est généralement *public*
  - est **unique**
  - est exécuté automatiquement à la destruction des objets
  - peut être appelé explicitement
    delete a; delete[] a; ~~a.~MaClasse();~~
    Pas recommandé…

NB: un destructeur n'est pas forcement public (cf. t-49) mais devrait toujours être virtuel

C++ - opérateurs

- Il est possible de surcharger les opérateurs du C++ : =, +, -, /, (), [], ==, +=, …

possibilité introduite dans Algol68

  - Le nom de la fonction porte le nom ***operator*** suivi du symbole ou mot clé à (re)définir

```
Complexe Complexe::operator/(const Complexe &z) const
{
  Complexe s;
  s = this->MultiplierPar( z.Conjuguer() );
  s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
  s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
  return s;
}
```

```
void main(void)
{
Complexe a(1,1);
Complexe b(1,-4);
Complexe z;
z = a / b;
}
```

Par défaut un *operator*= est ajouté à toutes les classes

➔ à surcharger en cas d'attribut dynamique

C++ - opérateur =

- La surcharge de l'opérateur égal '=' est obligatoire si la classe gère elle-même des espaces mémoires (allocation et libération)
- La déclaration de *operator*= respecte une syntaxe stricte:

```
Complexe& Complexe::operator=(const Complexe &z);
```

  - La déclaration permet d'écrire : a=b=c=d;
  - L'implémentation doit permettre de gérer : a=a; …
- L'implémentation de *operator*= respecte un canevas précis:

```
Tableau& Tableau ::operator=(const Tableau &t)
{ if( &t != this) // « t » est il l'objet courant?
    {
    //allocation et copie des éléments de t dans l'objet courant
    }
  return *this;  // on retourne le contenu de l'objet courant
}
```

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet
III. Encapsulation, concept de classe
IV. Héritage (généralisation) *… troisième partie …*
V. Polymorphisme
VI. Généricité (modèles de classe)

Exercice

- Modéliser les différents effectifs du département GE. Pour cela, on modélisera les différents constituants ainsi :
  - un étudiant par: nom, prénom, promo, mail, groupe
  - un enseignant par: nom, prénom, mail, matière
  - une secrétaire par: nom, prénom, mail, fonction
  - un technicien par: nom, prénom, mail, spécialité

Introduction aux méthodes Orientées Objets

Troisième partie

Modélisation avec UML 2.0

Programmation orientée objet en C++

Pré-requis:

maitrise des bases algorithmiques (cf. 1ier cycle),
maitrise du C (variables, fonctions, pointeurs, structures)

Thomas Grenier Insa-GE IF3

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet
III. Encapsulation, concept de classe
IV. Héritage (généralisation)
V. Polymorphisme
VI. Généricité (modèles de classe)

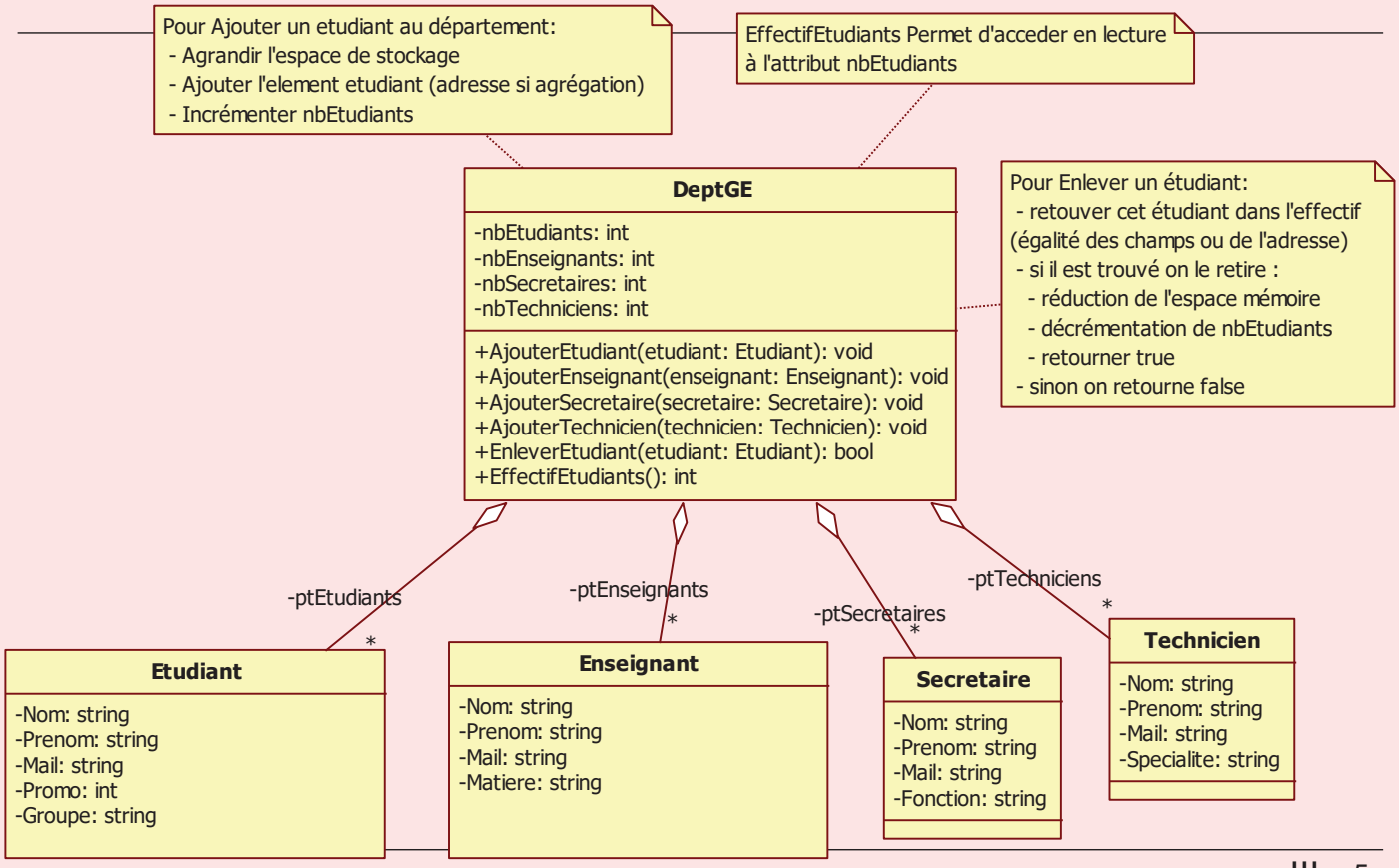
IV – Héritage – Plan

- Exemple sur exercice DeptGE
- Définitions
  - UML et C++
- Conventions C++
  - Ordre d'initialisation des objets
  - Ordre de destruction des objets

Exercice DeptGE

- Modéliser les différents effectifs du département GE. Pour cela, on modélisera les différents constituants ainsi :
  - un étudiant par: nom, prénom, promo, mail, groupe
  - un enseignant par: nom, prénom, mail, matière
  - une secrétaire par: nom, prénom, mail, fonction
  - un technicien par: nom, prénom, mail, spécialité

Exercice DeptGE



Exercice DeptGE

Etudiant

-Nom: string
-Prenom: string
-Mail: string
-Promo: int
-Groupe: string

+ModifierNom(nom: string): void
+ModifierPrenom(prenom: string): void
+ModifierMail(mail: string): void
+ModifierPromo(promo: int): void
+ModifierGroupe(groupe: string): void
+Afficher(): void
+ObtenirNom(): string
+ObtenirPrenom(): string
+ObtenirMail(): string
+ObtenirPromo(): int
+ObtenirGroupe(): string

Enseignant

-Nom: string
-Prenom: string
-Mail: string
-Matiere: string

+ModifierNom(nom: string): void
+ModifierPrenom(prenom: string): void
+ModifierMail(mail: string): void
+ModifierMatiere(matiere: string): void
+Afficher(): void
+ObtenirNom(): string
+ObtenirPrenom(): string
+ObtenirMail(): string
+ObtenirMatiere(): string

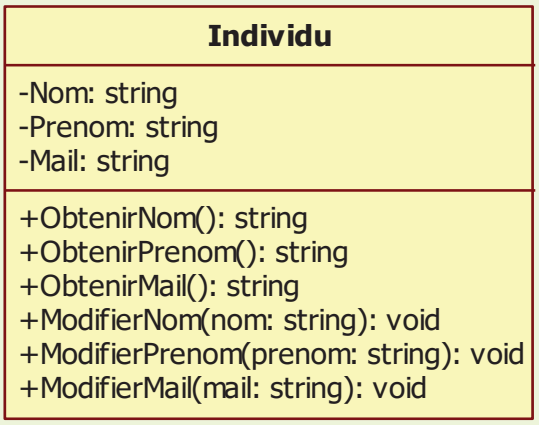
...

La même chose avec les classes:
- Secretaire
- Technicien

➔ Intérêts de regrouper les attributs et opérations identiques ? Comment ?

IV – Héritage (généralisation)

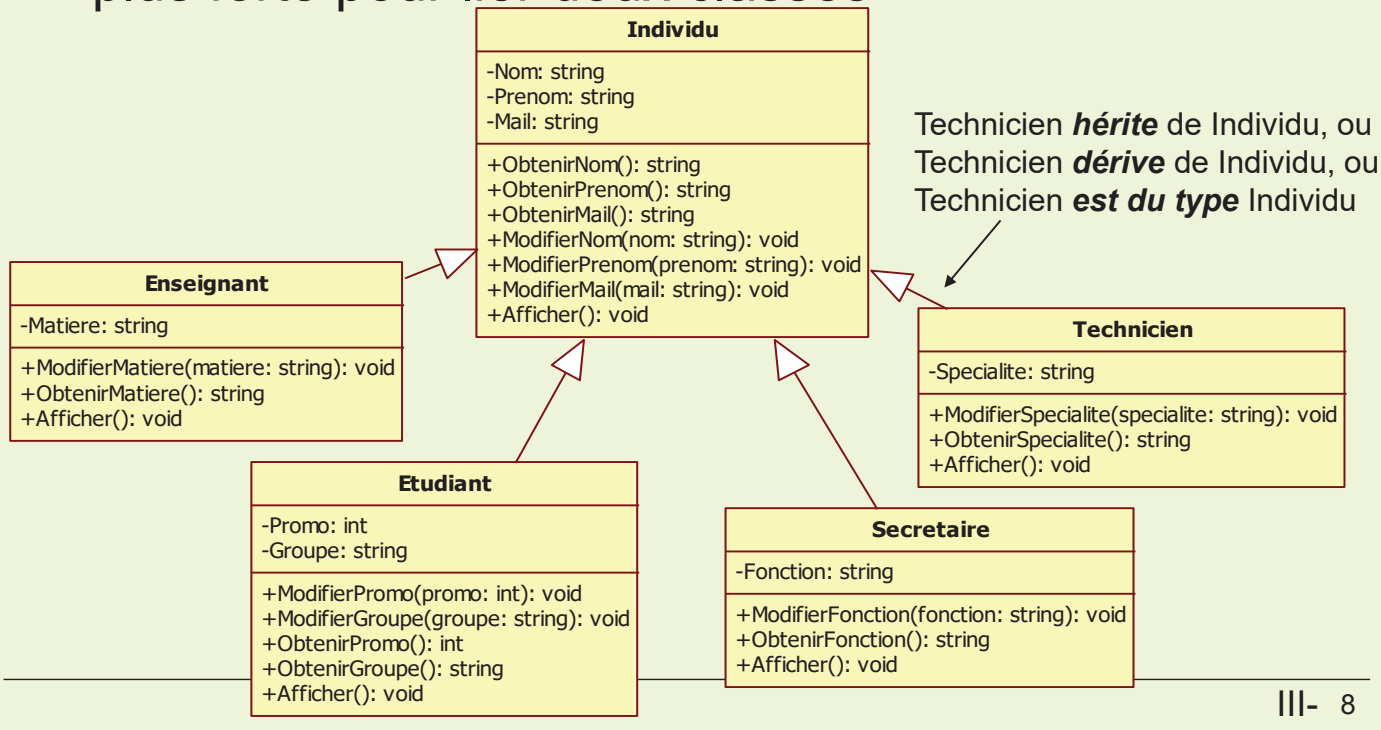
- Solution: faire une classe **plus générale*** (`Individu`) et <u>inclure</u> les fonctionnalités de cette classe dans les classes plus spécifiques:
  `Etudiant, Enseignant, Secretaire…`



* : ou « moins spécifique »

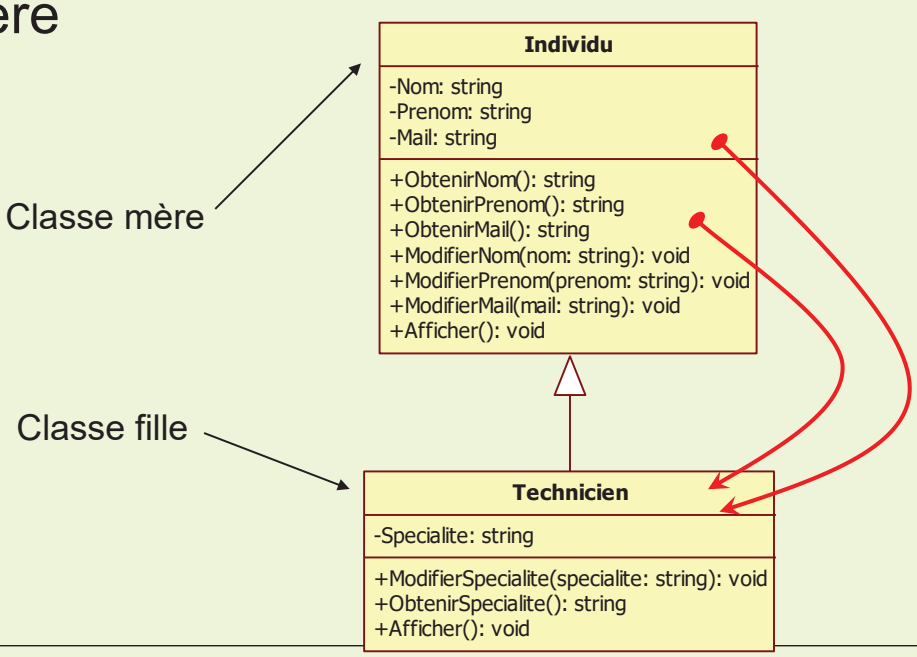
IV – Héritage (généralisation)

- La relation d'héritage (⟶) est la relation la plus forte pour lier deux classes



IV – Héritage (généralisation)

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

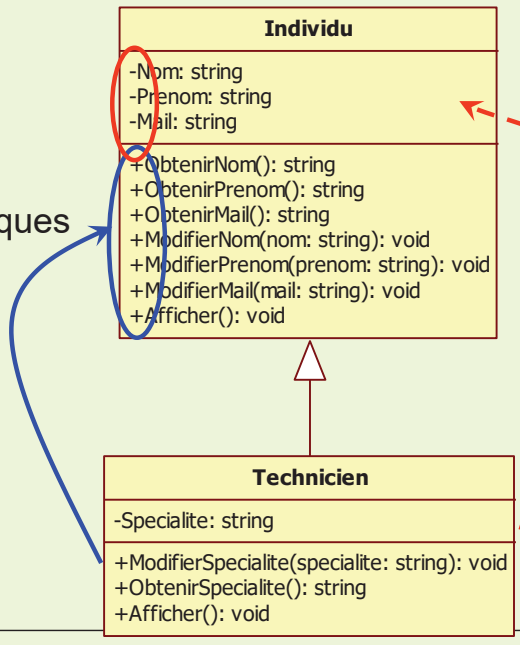


IV – Héritage (généralisation)

- La visibilité des attributs et des méthodes de la classe mère est appliquée à la classe fille

➔ Pour la classe fille
Accès direct ok:
les opérations sont publiques

➔ Pour la classe fille
Pas d'accès direct:
les attributs sont privés

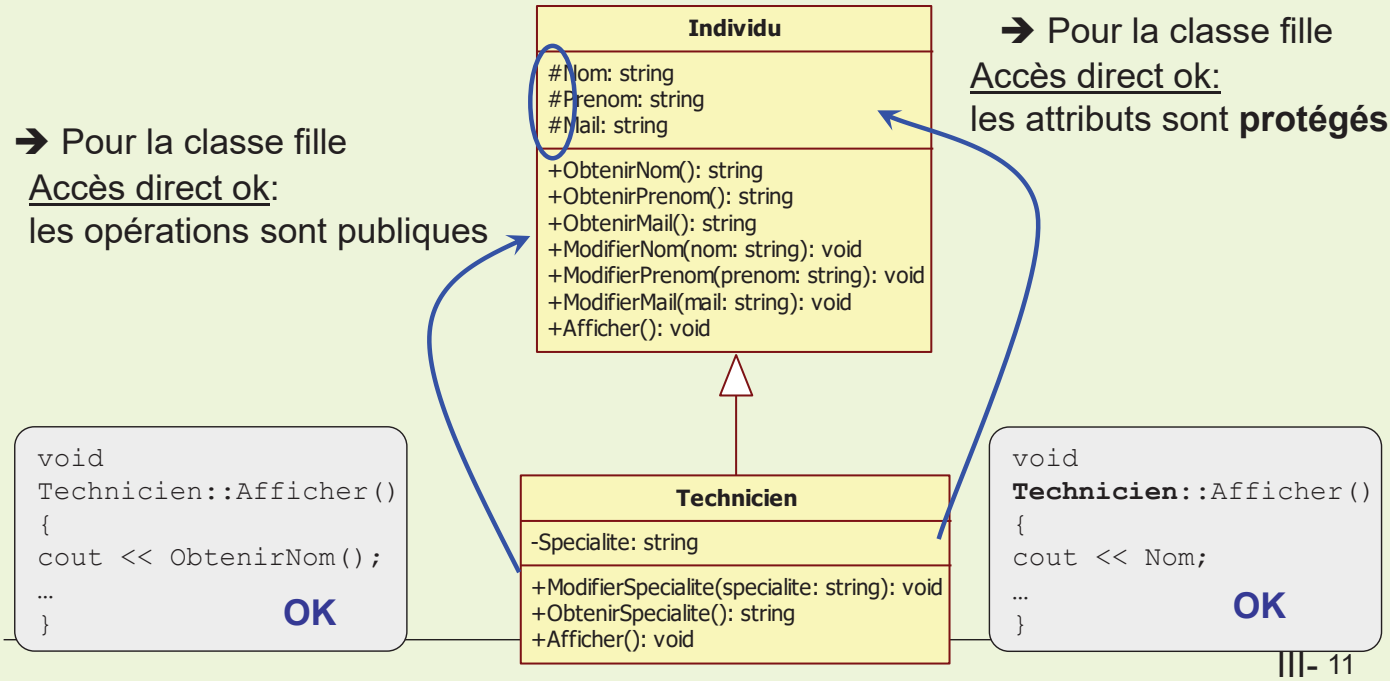



```
void
Technicien::Afficher()
{
cout << ObtenirNom();
…
}
```
OK

```
void
Technicien::Afficher()
{
cout << Nom;
…
}
```
NON !

IV – Héritage (généralisation)

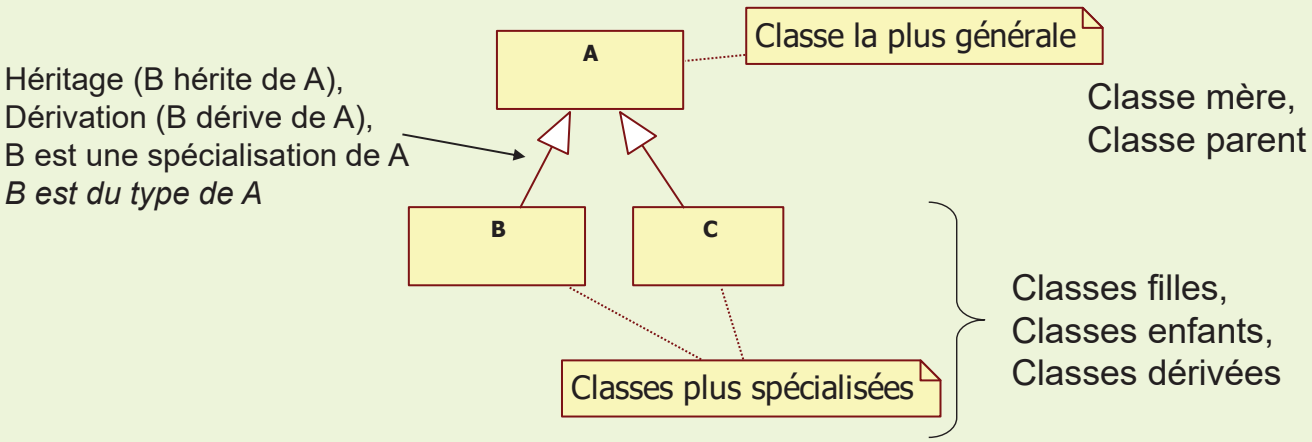
- La visibilité des attributs et des méthodes de la classe mère est appliquée à la classe fille



IV – Héritage, définition

- L'héritage (ou généralisation) permet de définir qu'une classe ***est du type*** d'une autre classe
- L'héritage est une relation unidirectionnelle

Symbole UML : 



IV – Héritage, définition

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

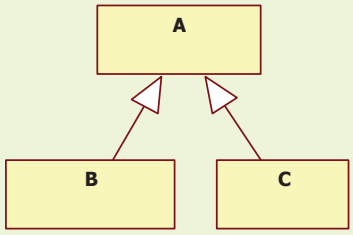
Héritage public

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

IV – Héritage, définition

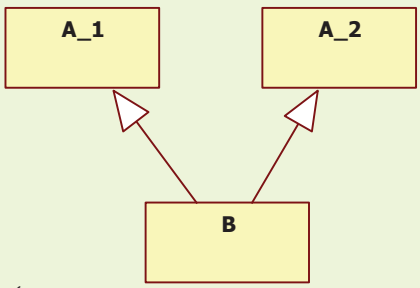
- Il existe la dérivation multiple

B hérite des membres de A
C hérite des membres de A



- Il existe l'héritage multiple

B hérite des membres de A_1 et de A_2

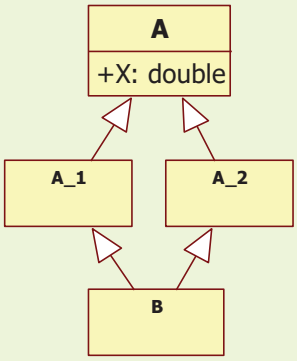


Exemple:
`Moto` hérite de `Objet2Roues` et `ObjetMotorisé`
`MotoCross` hérite de `Moto` et `MatérielToutTerrain`

IV – Héritage, mises en garde

- Une relation d'héritage n'a un sens que si elle est vraie dans une seule direction

  Il est vrai qu'un étudiant est forcément un individu, mais un individu n'est pas forcément un étudiant ➔ OK



- L'héritage multiple peut être problématique *(conflits)*

  B possède t il deux fois les membres de A ?

  B::A_1::X et B::A_2::X

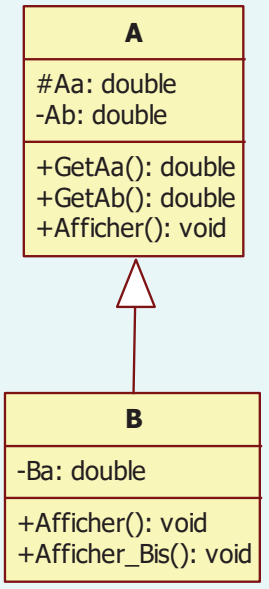
- L'héritage est mal adapté à la réutilisation d'implémentation avec évolutions

  Une classe fille est très fortement couplée à sa classe mère, si la classe mère évolue les classes filles devront probablement évoluer

  *(À moins d'avoir défini une **architecture d'objet** immuable)*

IV – Héritage en C++

UML



C++

```
class A
{
protected:
  double Aa
private:
  double Ab;
public:
  A() {};
  double GetAa()
   {return Aa;}
  double GetAb()
   {return Ab;}
  void Afficher()
   { cout<< Aa << Ab;}
};
```

```
class B : public A
{
private:
  double Ba;
public:
  B() {};
  void Afficher()
   {
   cout << Aa << GetAb();
   cout << Ba << endl;
   }
 void Afficher_Bis()
   {
   A::Afficher();
   cout << Ba << endl;
   }
};
```

IV – Héritage en C++

- Ordre des constructeurs
  - Le constructeur de la classe mère est exécuté avant celui de la classe fille
  - A la construction de la classe fille, il est possible de passer des paramètres au constructeur de la classe mère, sinon le constructeur par défaut de la classe mère est exécuté

```
class A
{
double X;
public:
  A()
   { cout << "A:defaut" << endl;}
  A(double a)
   { X=a;
     cout << "A:General" << endl;}
};
```

```
class B: public A
{
public:
  B()
   { cout << "B:defaut" << endl;}
  B(double a):A(a)
   { cout << "B:General" << endl;}
};
```

```
void main()
{
B ob1; //X=?
B ob2(3);//X=3
}
```

Affichage:
A:defaut
B:defaut

IV – Héritage en C++

- Ordre des constructeurs
  - Le constructeur de la classe mère est exécuté avant celui de la classe fille
  - A la construction de la classe fille, il est possible de passer des paramètres au constructeur de la classe mère, sinon le constructeur par défaut de la classe mère est exécuté

```
class A
{
double X;
public:
  A()
   { cout << "A:defaut" << endl;}
  A(double a)
   { X=a;
     cout << "A:General" << endl;}
};
```

```
class B: public A
{
public:
  B()
   { cout << "B:defaut" << endl;}
  B(double a):A(a)
   { cout << "B:General" << endl;}
};
```

```
void main()
{
B ob1; //X=?
B ob2(3);//X=3
}
```

Résultat:
A:General
B:General

IV – Héritage en C++

- Ordre des destructeurs
  - Le destructeur de la classe fille est exécuté avant le destructeur de la classe mère

```
class TableauDouble
{
double *Tab;
int Taille;
public:
  TableauDouble(int taille=1)
        :Taille(taille)
    { Tab = new double[Taille]; }
  double GetElement(int i) const
    { return Tab[i]; }
  void SetElement(int i, double v)
    { Tab[i] = v; }
  int GetTaille() const
    { return Taille; }
  ~TableauDouble() // destructeur
    { delete[] Tab;
   cout << "TD:Destructeur"<< endl;
    }
};
```

```
class VecteurZero:
        public TableauDouble
{
public:
  VecteurZero(int taille)
        :TableauDouble(taille)
    {for( int i=0; i<GetTaille(); i++)
        SetElement(i, 0);
    }
  ~VecteurZero() //Destructeur inutile
    {cout << "VZ:Destructeur"<< endl;}
};
```

```
void main()
{
VecteurZero a(10);
a.SetElement( 0, 1+a.GetElement(0));
}
```

Introduction aux méthodes Orientées Objets

Quatrième partie

Modélisation avec UML 2.0

Programmation orientée objet en C++

Pré-requis:

maitrise des bases algorithmiques (cf. 1ier cycle),
maitrise du C (variables, fonctions, pointeurs, structures)

Thomas Grenier Insa-GE IF3

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet
III. Encapsulation, concept de classe
IV. Héritage (généralisation)
V. Polymorphisme
VI. Généricité (modèles de classe)

V – Polymorphisme – Plan

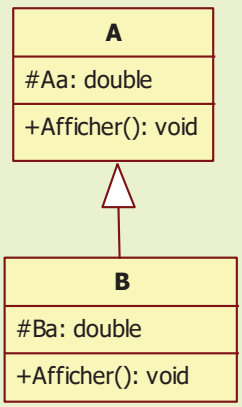
- Concept du polymorphisme
- Définitions
  - UML
  - C++
- Exercice
- Quizz

V – Polymorphisme, concept

Polymorphisme : *qui peut prendre plusieurs formes*

- Concept relatif à la surcharge des opérations des classes d'une même hiérarchie
- Extension des possibilités du mécanisme d'héritage

*Un objet d'une classe fille est **du même type** que sa classe mère*

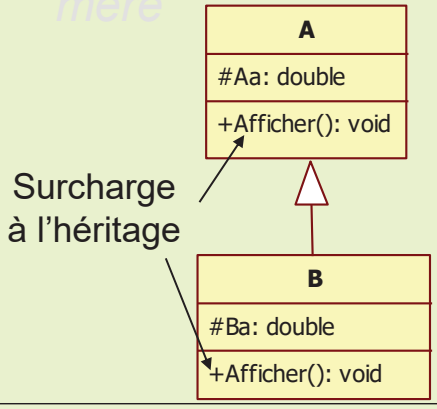


V – Polymorphisme, concept

Polymorphisme : *qui peut prendre plusieurs formes*

- Concept relatif à la surcharge des opérations des classes d'une même hiérarchie
- Extension des possibilités du mécanisme d'héritage

  *Un objet d'une classe fille est **du même type** que sa classe mère*



Surcharge de Afficher()

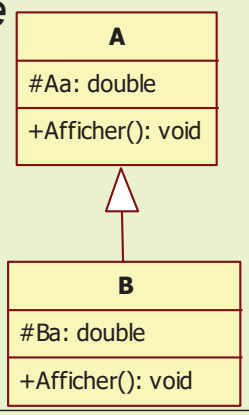
pour un objet de type B, A::Afficher() est remplacée par B::Afficher()

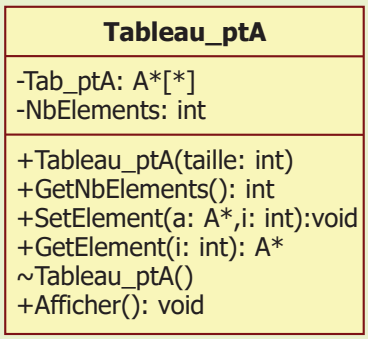
V – Polymorphisme, concept

Polymorphisme : *qui peut prendre plusieurs formes*

- Concept relatif à la surcharge des opérations des classes d'une même hiérarchie
- Extension des possibilités du mécanisme d'héritage

  *Un objet d'une classe fille est **du même type** que sa classe mère*





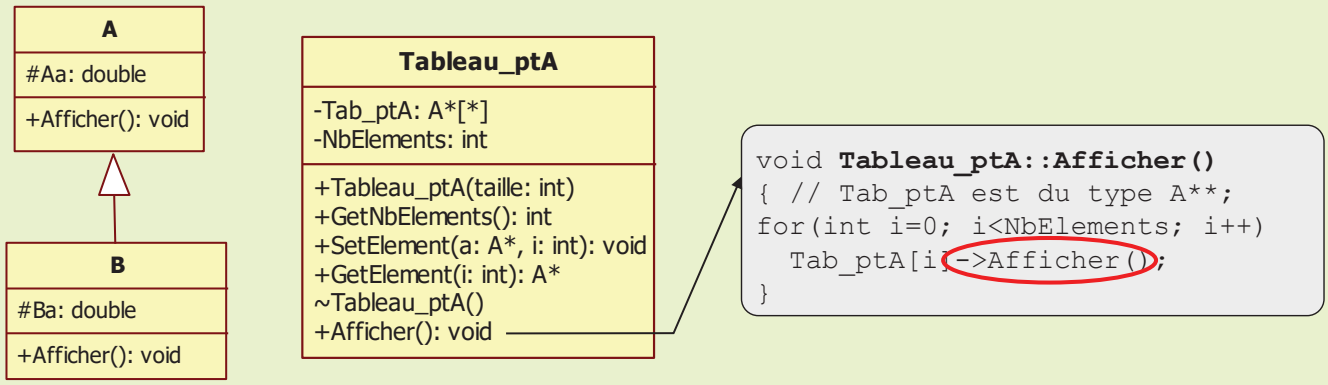
Avec un objet de Tableau_ptA on peut gérer :
- des objets de type A
- des objets de type B !

Polymorphisme

Nb : les notations A sont abusives en UML, un champ A* est une association ou une agrégation*

V – Polymorphisme, concept

Problème: surcharge des opérations à l'héritage



Quelle fonction Afficher() est appelée ?

Si l'élément *i* est un objet A ?
Si l'élément *i* est un objet B ?
} A::Afficher()

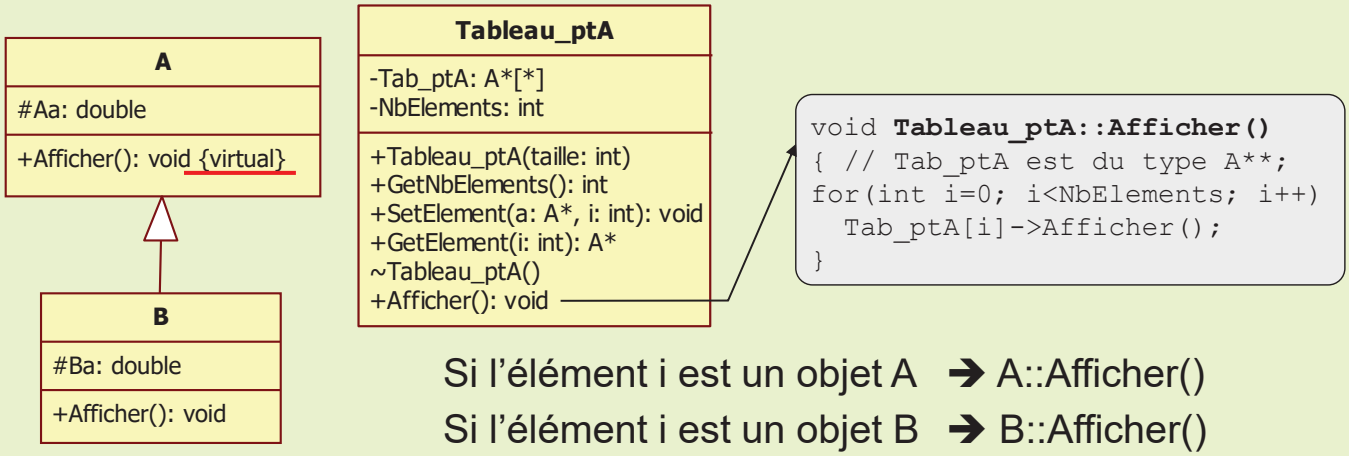
A::Afficher n'est pas la méthode la plus adaptée pour l'affichage des objets B
(le champs B::Ba ne peut pas être affiché par A::Afficher())

Polymorphisme

Nb : les notations A sont abusives en UML*

V – Polymorphisme, UML

- Opération **virtuelle** *(représentée en ~~italique~~ par {virtual}…)*
  - Opération avec implémentation (en C++)
  - Signifie: « *mes classes filles **peuvent** se charger de la ré-implémentation de ce comportement* »

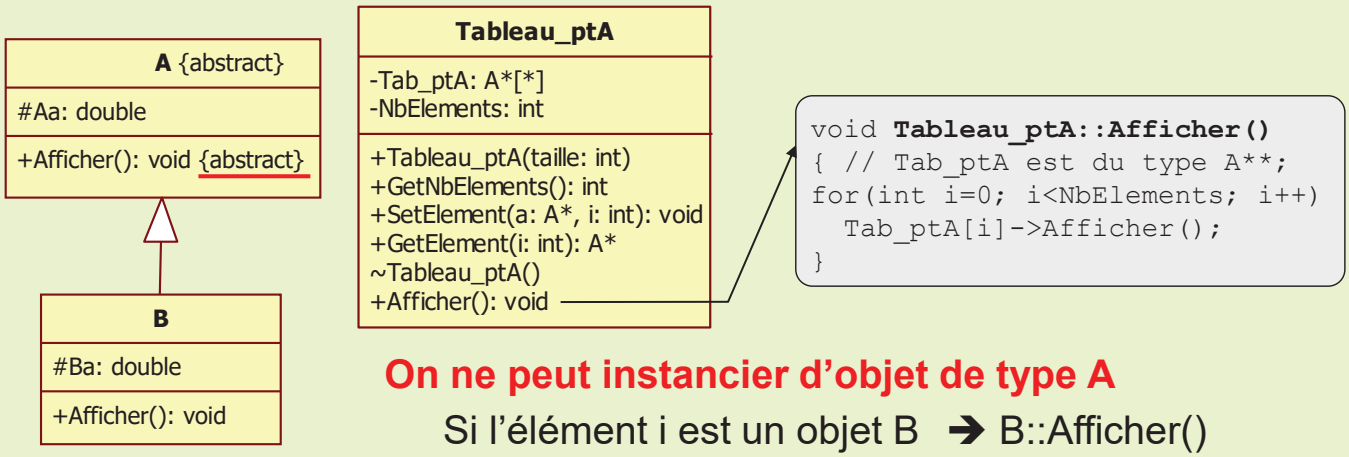


Si l'élément i est un objet A ➔ A::Afficher()
Si l'élément i est un objet B ➔ B::Afficher()

Nb : les notations A sont abusives en UML, un champ A* est une association ou une agrégation*

V – Polymorphisme, UML

- Opération **abstraite** *(représentée ~~en italique~~ par {abstract}…)*
  - Opération sans implémentation (en C++)
  - Signifie: « *mes classes filles **doivent** se charger de l'implémentation de ce comportement* »



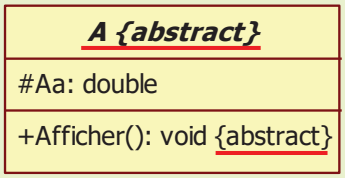
On ne peut instancier d'objet de type A

Si l'élément i est un objet B ➔ B::Afficher()

Nb : les notations A sont abusives en UML, un champ A* est une association ou une agrégation*

V – Polymorphisme, UML

- Classe abstraite *(représentée ~~en italique~~ par {abstract}…)*
  - Une classe possédant une opération abstraite est obligatoirement une classe abstraite
  - Une classe abstraite ne peut pas être instanciée (dérivation obligatoire)

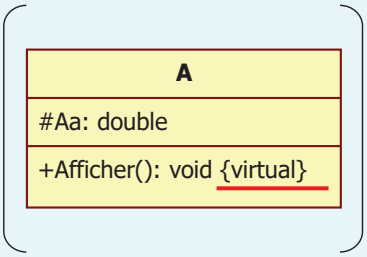


On ne peut pas instancier des objets de type A…

➔ Les classes abstraites sont les classes les plus hautes dans la hiérarchie
➔ Les classes abstraites permettent de définir une architecture et obligent les classes dérivées à respecter cette architecture

V – Polymorphisme, C++

- Méthode virtuelle ({virtual} en UML*)
  - méthode pouvant être implémentée en C++
  - Signifie: « *je propose une implémentation de ce comportement mais mes classes filles peuvent l'améliorer, l'implémentation la plus adaptée au type de l'objet sera exécutée*»



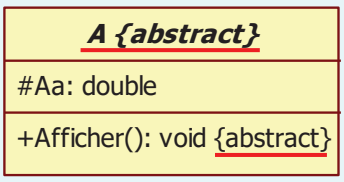

```
class A
{
protected:
  double Aa;
public:
  A(double a=1):Aa(a) {}
  virtual void Afficher()
         {cout << Aa; }
};
```

On peut instancier des objets de type A !

*: notation normalement inexistante en UML

V – Polymorphisme, C++

- Méthode virtuelle pure ou méthode abstraite
  - méthode non implémentée en C++
  - Signifie: « *mes classes filles sont chargées de l'implémentation de ce comportement* »

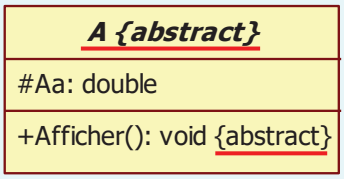



```
class A
{
protected:
  double Aa;
public:
  A(double a =1):Aa(a) {}
  virtual void Afficher() = 0;
};
```

On ne peut pas instancier des objets de type A…

V – Polymorphisme, C++

- Classe abstraite
  - Une classe possédant au moins une méthode virtuelle pure est obligatoirement une classe abstraite
  - Une classe abstraite ne peut pas être instanciée (dérivation obligatoire)

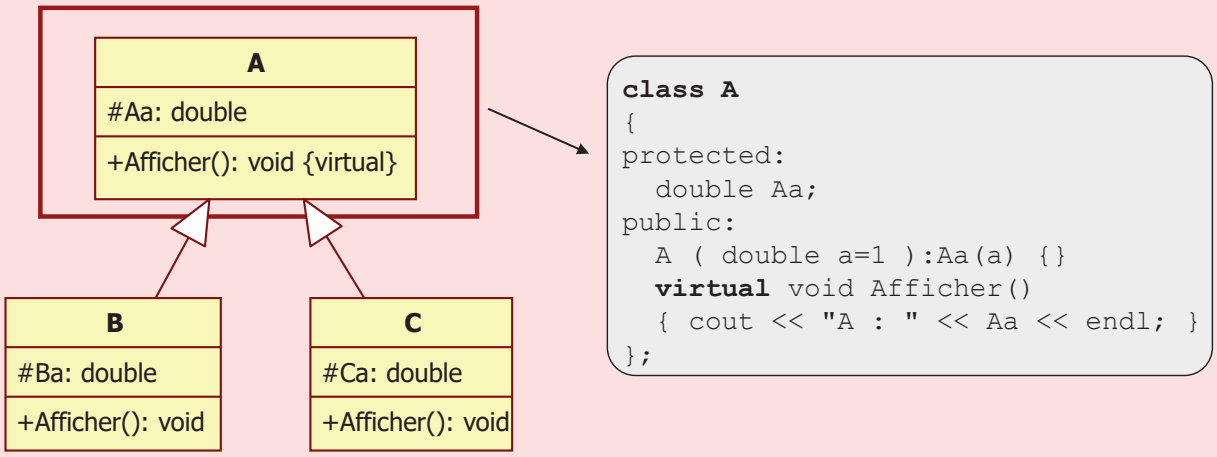



```
class A
{
protected:
  double Aa;
public:
  A(double a =1):Aa(a) {}
  virtual void Afficher() = 0;
};
```

On ne peut pas instancier des objets de type A

V – Polymorphisme, exemple

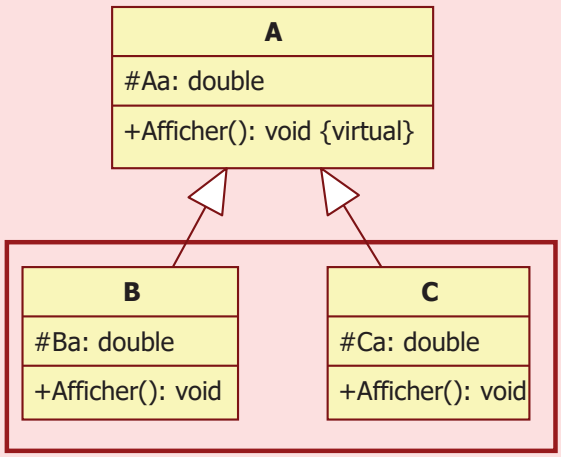
- Exemple complet:
  Tableau de classes A, B et C




```
class A
{
protected:
  double Aa;
public:
  A ( double a=1 ):Aa(a) {}
  virtual void Afficher()
  { cout << "A : " << Aa << endl; }
};
```

V – Polymorphisme, exemple

- Exemple complet: classes A, B et C

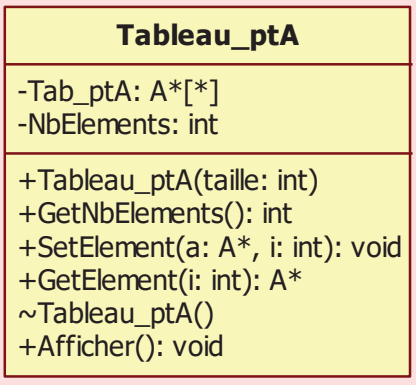



```
class B : public A
{
protected:
  double Ba;
public:
  B( double a=1, double b=1):A(a),Ba(b) {}
  void Afficher()
   { cout<<"B :"<< Aa <<", "<< Ba << endl; }
};
```

```
class C : public A
{
protected:
  double Ca;
public:
  C( double a=1, double c=1):A(a),Ca(c) {}
  void Afficher()
   { cout<<"C :"<< Aa <<", "<< Ca << endl; }
};
```


V – Polymorphisme, exemple

- Exemple complet: classe Tableau_ptA




```
class Tableau_ptA
{
  A **Tab_ptA;
  int NbElements;
public:
  Tableau_ptA(int taille)
    { Tab_ptA = new A*[taille];
      NbElements = taille;   }
  void SetElement( A *a, int i )
    { Tab_ptA[i] = a; }
  A* GetElement( int i )
    { return Tab_ptA[i]; }
  int GetNbElements()
    { return NbElements; }
  void Afficher();
  ~Tableau_ptA()
    { delete[] Tab_ptA; }
};
```

```
void Tableau_ptA::Afficher()
{
for( int i=0; i<NbElements; i++)
  {
  cout << "Tab["<< i << "] :";
  Tab_ptA[i]->Afficher();
  }
}
```

V – Polymorphisme, exemple

- Exemple complet: utilisation

```
int main(void)
{
B b1( 1, 1);
B b2( 2, 2);

C c1(-1, -1);
C c2(-2, -2);

Tableau_ptA tab(4);

tab.SetElement( &b1, 0);
tab.SetElement( &c1, 1);
tab.SetElement( &c2, 2);
tab.SetElement( &b2, 3);

tab.Afficher();

return 0;
}
```

Création des objets de types B et C

Création d'un tableau de 4 éléments

Affectation des éléments du tableau

Affichage de tous les éléments du tableau

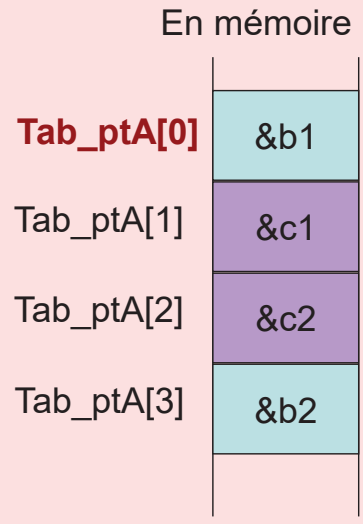
Exécution :

```
Tab[0] :B :1, 1
Tab[1] :C :-1, -1
Tab[2] :C :-2, -2
Tab[3] :B :2, 2
```


V – Polymorphisme, exemple

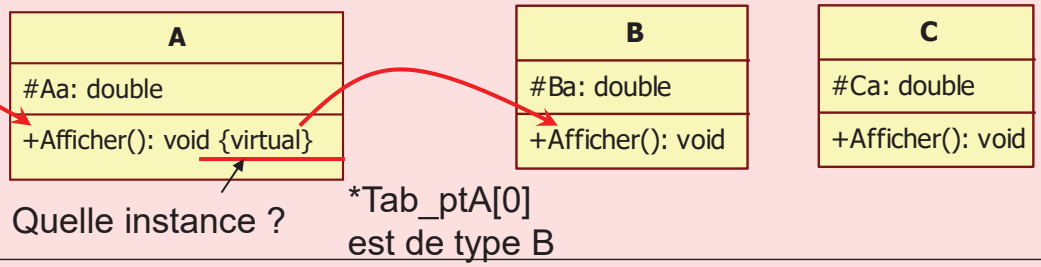
- exécution de `tab.Afficher();`

```
void Tableau_ptA::Afficher()
{
for( int i=0; i<NbElements; i++)
  {
  cout << "Tab["<< i << "] :";
  Tab_ptA[i]->Afficher();
  }
}
```



Tab_ptA[0]->Afficher()

type A*



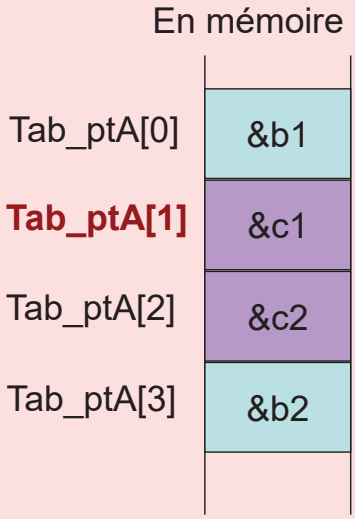
Quelle instance ?

*Tab_ptA[0] est de type B

V – Polymorphisme, exemple

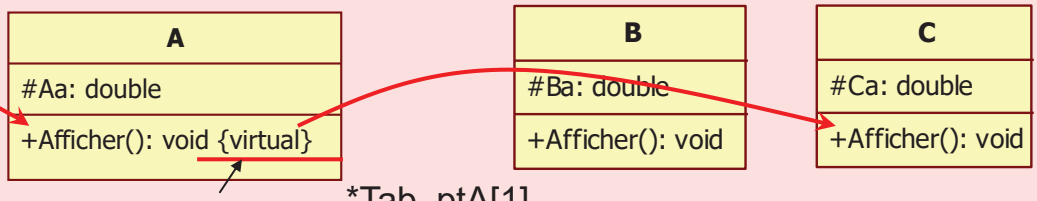
- exécution de `tab.Afficher();`

```
void Tableau_ptA::Afficher()
{
for( int i=0; i<NbElements; i++)
  {
  cout << "Tab["<< i << "] :";
  Tab_ptA[i]->Afficher();
  }
}
```



Tab_ptA[1]->Afficher()

type A*



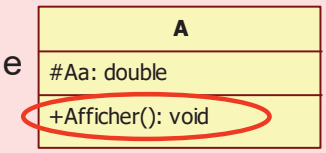
Quelle instance ?

*Tab_ptA[1] est de type C

V – Polymorphisme, exemple

- Comparaison: avec et sans méthode virtuelle

Sans méthode virtuelle




```
class A
{
protected:
  double Aa;
public:
  A ( double a=1 ):Aa(a) {}
  void Afficher()
  { cout << "A : " << Aa << endl; }
};
```

Exécution du même « main »

sans méthode virtuelle

```
Tab[0] :A : 1
Tab[1] :A : -1
Tab[2] :A : -2
Tab[3] :A : 2
```

avec méthode virtuelle

```
Tab[0] :B :1, 1
Tab[1] :C :-1, -1
Tab[2] :C :-2, -2
Tab[3] :B :2, 2
```

V – Polymorphisme - Quizz

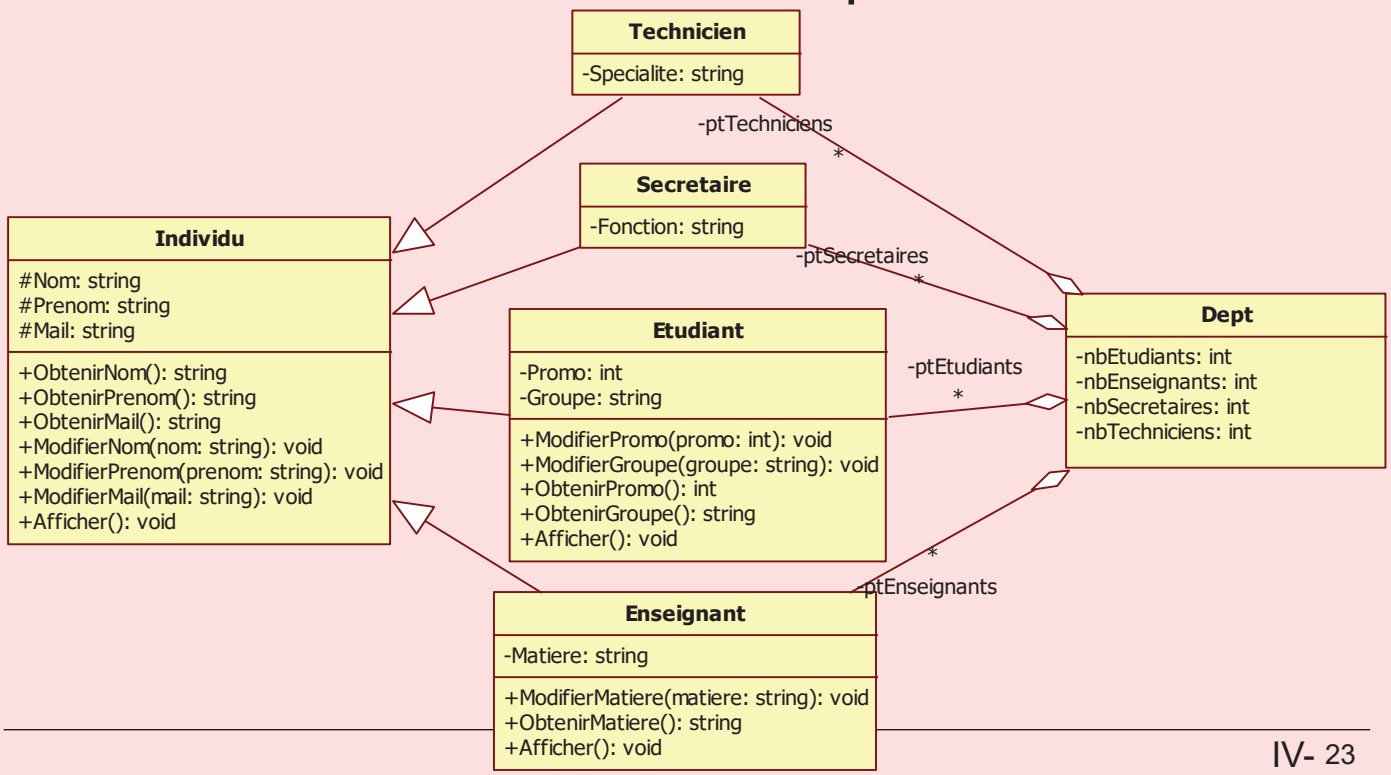
- Quels sont les objectifs/buts du polymorphisme ?
  - –
  - –
  - –
  - –
- Une méthode devrait toujours être virtuelle. Laquelle ?

Exercice équipe de sport Dept

- A partir de la modélisation d'un département (classes Dept, Individu, Etudiant, Enseignant, …), modéliser **une équipe** dont les joueurs sont un mélange d'étudiants, d'enseignants, de secrétaires et de techniciens appartenant tous à un même département.
  - On voudra afficher la liste des joueurs d'une équipe, en précisant tous les attributs de chaque joueur (méthode afficher)
  - Pour les feuilles de match, on ne souhaite afficher que le nom et le prénom (méthode AfficherFeuilleMatch())

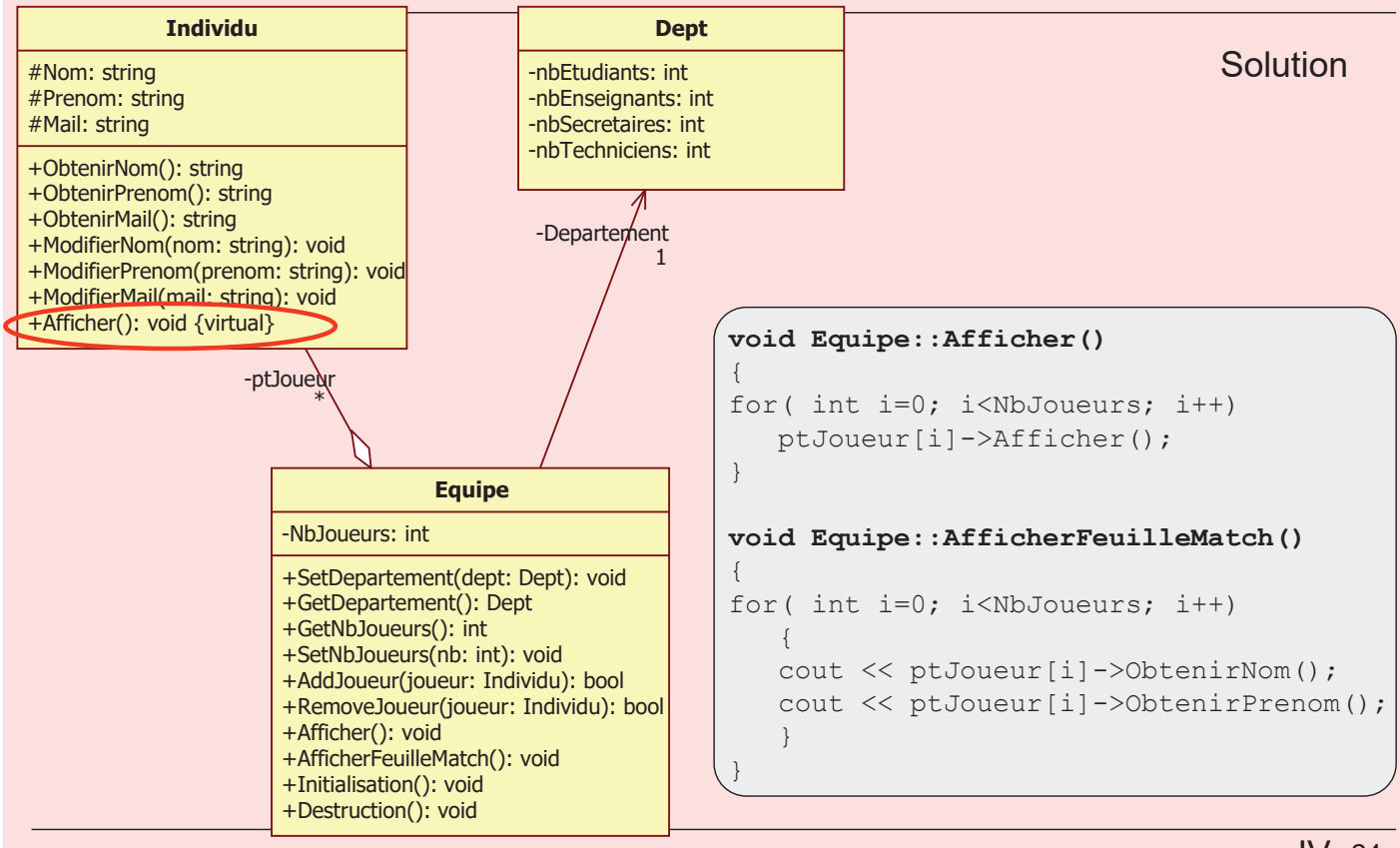
Exercice équipe de sport Dept

- Modélisation initiale d'un département :



Exercice équipe de sport Dept

Solution




```
void Equipe::Afficher()
{
for( int i=0; i<NbJoueurs; i++)
   ptJoueur[i]->Afficher();
}

void Equipe::AfficherFeuilleMatch()
{
for( int i=0; i<NbJoueurs; i++)
   {
   cout << ptJoueur[i]->ObtenirNom();
   cout << ptJoueur[i]->ObtenirPrenom();
   }
}
```

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet
III. Encapsulation, concept de classe
IV. Héritage (généralisation)
V. Polymorphisme
VI. Généricité (modèles de classe)

VI – Généricité – Plan

- Intérêt de la généricité
- Définitions
  - UML
  - C++
- Exercice

VI – Généricité – Plan

- Intérêt de la généricité
- Définitions
  - UML
  - C++
- Exercice

VI – Généricité – Intérêt

- Modéliser les classes Tableau_Double, Tableau_Int, Tableau_ptA.

  Ces classes disposeront des mêmes fonctionnalités

Tableau_Int
-Tab: int[0..*] -Taille: int
+SetElement(i: int, v: int): void +GetElement(i: int): int +GetTaille(): int +SetTaille(taille: int): void <<create>>+Tableau_Int(taille: int) <<destroy>>+Tableau_Int() +Copie(t: Tableau_Int): void

Tableau_Double
-Tab: double[0..*] -Taille: int
+SetElement(i: int, v: double): void +GetElement(i: int): double +GetTaille(): int +SetTaille(taille: int): void <<create>>+Tableau_Double(taille: int) <<destroy>>+Tableau_Double() +Copie(t: Tableau_Double): void

Tableau_ptA
-Tab: A*[0..*] -Taille: int
+SetElement(i: int, v: A*): void +GetElement(i: int): A* +GetTaille(): int +SetTaille(taille: int): void <<create>>+Tableau_ptA(taille: int) <<destroy>>+Tableau_ptA() +Copie(t: Tableau_ptA): void

Alors? qu'est ce qu'on fait? Architecture + polymorphisme ?

Le concept de polymorphisme est relatif aux opérations… et non aux types (paramètres ou attributs)

VI – Généricité – Intérêt

➢ **Obtenir une généricité du comportement d'une classe vis-à-vis des types des attributs**

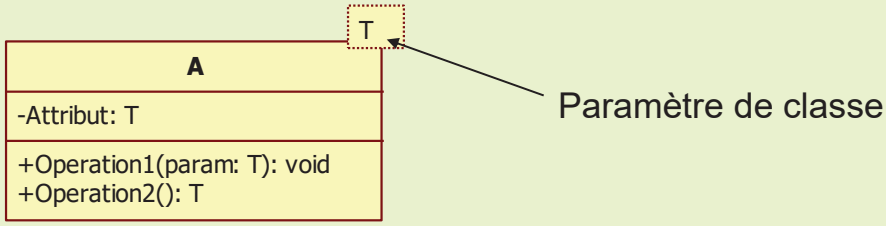
le comportement interne d'une classe « générique » doit être indépendant du type des objets manipulés

➢ Retarder le choix des classes avec lesquelles une classe fonctionnera

« je sais que cette classe fonctionnera avec d'autres classes, mais je ne connais pas, ou cela m'est égal, de connaître ces classes »

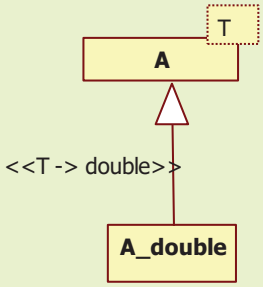
VI – Généricité – Définition UML

- modèle de classe ou classe paramétrée



- Deux manières d'associer un ensemble de classe à un modèle

Liaison d'héritage



Liaison à l'exécution

```
MonObjet:A<T->double>
```

… template

Exemple vector<T>

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
  vector<int> myvector;
  int sum (0);
  myvector.push_back (100);
  myvector.push_back (200);
  myvector.push_back (300);
  while (!myvector.empty())
             {
              sum+=myvector.back();
              myvector.pop_back();
             }
  cout << "The elements of myvector summed " << sum << endl;
  return 0;
}
```


Exemple vector<T>

```
// vector::operator[]
#include <iostream>
#include <vector>
 using namespace std;
 int main () {
 vector<int> myvector (10); // 10 zero-initialized elements
unsigned int i;
vector<int>::size_type sz = myvector.size();
// assign some values:
 for (i=0; i<sz; i++)
              myvector[i]=i;
// reverse vector using operator[]:
for (i=0; i<sz/2; i++) {
              int temp;
              temp = myvector[sz-1-i];
              myvector[sz-1-i]=myvector[i];
              myvector[i]=temp; }
cout << "myvector contains:";
for (i=0; i<sz; i++) cout << " " << myvector[i]; cout << endl;
 return 0; }
```

Introduction aux méthodes Orientées Objets

Exercices

Modélisation avec UML 2.0

Programmation orientée objet en C++

Pré-requis:

maitrise des bases algorithmiques (cf. 1ier cycle),
maitrise du C (variables, fonctions, pointeurs, structures)

Thomas Grenier Insa-GE IF3

Tableau dynamique **

Enoncé

```
class TableauDouble
{
double *Tab;
int Taille;
public:
  TableauDouble(int taille=1)
        :Taille(taille)
    { Tab = new double[Taille]; }
  double GetElement(int i) const
    { return Tab[i]; }
  void SetElement(int i, double v)
    { Tab[i] = v; }
  int GetTaille() const
    { return Taille; }
  ~TableauDouble()
    { delete[] Tab;}
};
```

1- Modéliser cette classe.

2- Ajouter la méthode *SetTaille.* Décrire son fonctionnement.

3- Réaliser la méthode *SetTaille.*

4- Réaliser la méthode *Copie* qui permet de faire une copie d'un objet *TableauDouble* passé en paramètre dans l'objet courant.

5- Donner un exemple d'utilisation

Tableau dynamique

Solution

```
class TableauDouble
{
double *Tab;
int Taille;
public:
  TableauDouble(int taille=1)
        :Taille(taille)
   { Tab = new double[Taille]; }
  double GetElement(int i) const
   { return Tab[i]; }
  void SetElement(int i, double v)
   { Tab[i] = v; }
  int GetTaille() const
   { return Taille; }
  ~TableauDouble()
   { delete[] Tab;}
};
```

1- Modéliser cette classe

–

Tableau_Double
-Tab: double[0..*] -Taille: int
+SetElement(i: int, v: double): void +GetElement(i: int): double +GetTaille(): int +SetTaille(taille: int): void <<create>>+TableauDouble(taille: int) <<destroy>>+TableauDouble()

Tableau dynamique

Solution

```
class TableauDouble
{
double *Tab;
int Taille;
public:
  TableauDouble(int taille=1)
        :Taille(taille)
   { Tab = new double[Taille]; }
  double GetElement(int i) const
   { return Tab[i]; }
  void SetElement(int i, double v)
   { Tab[i] = v; }
  int GetTaille() const
   { return Taille; }
  ~TableauDouble()
   { delete[] Tab;}
  void SetTaille(int taille);
};
```

2- Ajouter la méthode *SetTaille.* Décrire son fonctionnement.

–

Tableau_Double
-Tab: double[0..*] -Taille: int
+SetElement(i: int, v: double): void +GetElement(i: int): double +GetTaille(): int +SetTaille(taille: int): void <<create>>+TableauDouble(taille: int) <<destroy>>+TableauDouble()

SetTaille doit:

a- supprimer l'espace alloué

b- allouer un nouvel espace

c- mettre à jour le champs Taille

Tableau dynamique

Solution

```
class TableauDouble
{
double *Tab;
int Taille;
public:
  TableauDouble(int taille=1)
        :Taille(taille)
   { Tab = new double[Taille]; }
  double GetElement(int i) const
   { return Tab[i]; }
  void SetElement(int i, double v)
   { Tab[i] = v; }
  int GetTaille() const
   { return Taille; }
  ~TableauDouble()
   { delete[] Tab;}
  void SetTaille(int taille);
};
```

3- Réaliser la méthode *SetTaille.*

SetTaille doit:

a- supprimer l'espace alloué
b- allouer un nouvel espace
c- mettre à jour le champs Taille

```
void TableauDouble::SetTaille
 (int taille)
{
if( Taille != taille )
  {
  delete[] Tab;
  Tab = new double[taille];
  Taille = taille;
  }
}
```

Tableau dynamique

Solution

```
class TableauDouble
{
double *Tab;
int Taille;
public:
  TableauDouble(int taille=1)
        :Taille(taille)
   { Tab = new double[Taille]; }
  double GetElement(int i) const
   { return Tab[i]; }
  void SetElement(int i, double v)
   { Tab[i] = v; }
  int GetTaille() const
   { return Taille; }
  ~TableauDouble()
   { delete[] Tab;}
  void SetTaille(int taille);
  void Copie( const TableauDouble &t);
};
```

4- Réaliser la méthode *Copie* qui permet de faire une copie d'un objet *TableauDouble* passé en paramètre dans l'objet courant.

a- supprimer l'espace alloué
b- allouer un nouvel espace
c- mettre à jour le champs Taille
d- copier les valeurs du tableau de l'objet passé en paramètre

```
void TableauDouble::Copie
 (const TableauDouble &t)
{
SetTaille( t.GetTaille() );
for(int i=0;i<Taille;i++)
  Tab[i] = t.GetElement(i);
}
```

Tableau dynamique

Solution

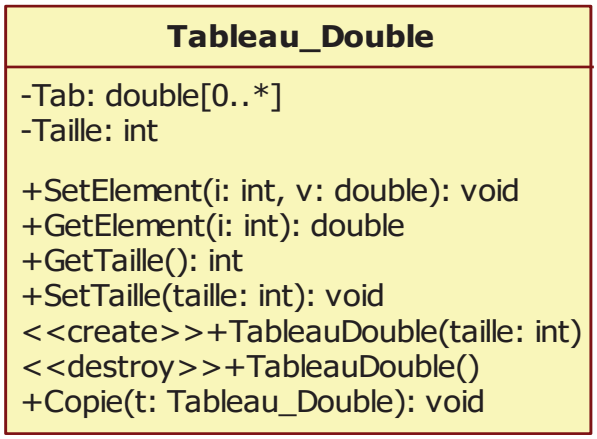
```
void main()
{
int taille;
int i;
double tmp;
TableauDouble copie;

cout << "Taille du tableau ?";
cin >> taille;
TableauDouble tab(taille);

for( i=0; i<taille; i++)
  {
  cout << "Tab[" << i << "]";
  cin >> tmp;
  tab.SetElement(i, tmp);
  }
copie.Copie( tab );
for( i=0; i<taille; i++)
  {
  cout << "Tab[" << i << "] = ";
  cout << copie.GetElement(i)<< endl;
  }
}
```

5- Donner un exemple d'utilisation

–



DeptGE *

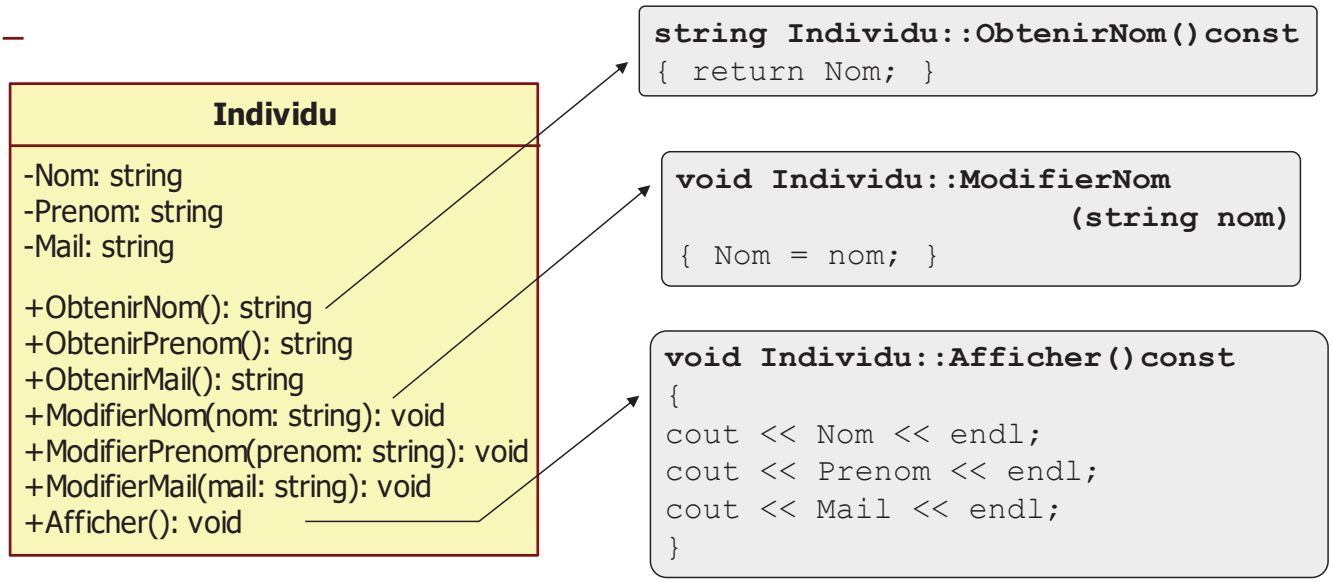
Enoncé

- Modéliser les différents effectifs du département GE. Pour cela, on modélisera les différents constituants ainsi :
  - un étudiant par: nom, prénom, promo, mail, groupe
  - un enseignant par: nom, prénom, mail, matière
  - une secrétaire par: nom, prénom, mail, fonction
  - un technicien par: nom, prénom, mail, spécialité
- On utilisera la classe Individu dont les membres sont:
  - Champs : nom, prénom et mail (de type string)
  - Fonctions d'accès en lecture et écriture à tous les champs
  - Une fonction d'affichage des 3 champs

DeptGE

Solution

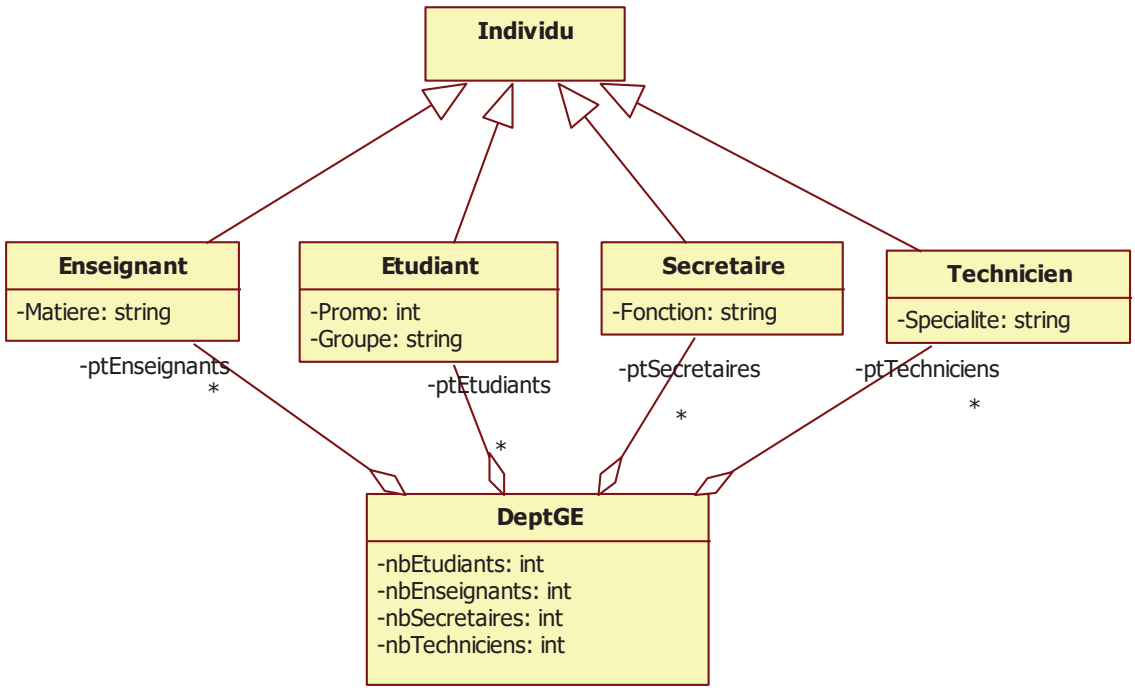
- Classe Individu



DeptGE

Solution

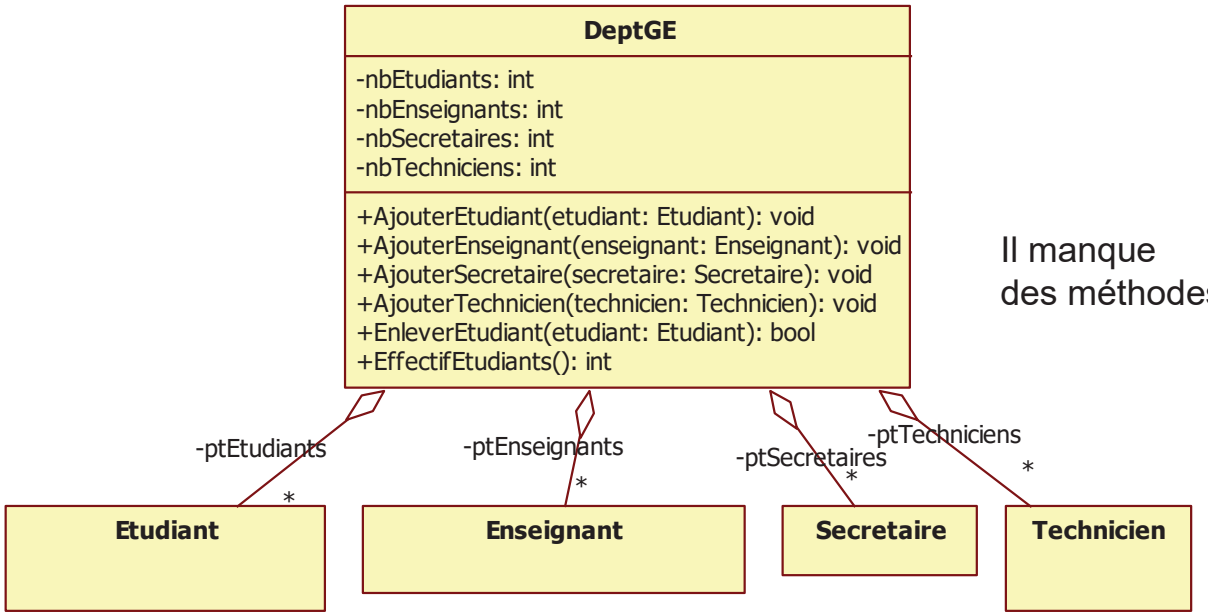
- Modélisation



DeptGE

Solution

- Modélisation



Il manque des méthodes…

DeptGE

Solution

- C++ des champs de DeptGE

```
class DeptGE
{
private:
  int nbEtudiants;
  int nbEnseignants;
  int nbSecretaires;
  int nbTechniciens;
  Enseignant **ptEnseignants;
  Etudiant   **ptEtudiants;
  Secretaire **ptSecretaires;
  Technicien **ptTechniciens;
public:
  …
  void AjouterEtudiant(Etudiants *etudiant);
  void AfficherEtudiants() const;
};
```

DeptGE

Solution

- Réalisation C++, AjouterEtudiant

```
void DeptGE::AjouterEtudiant(Etudiants *etudiant)
{
Etudiant **tmp;
tmp = new Etudiant*[ nbEtudiants+1 ];

for( int i=0;i<nbEtudiants;i++)
  tmp[i] = ptEtudiants[i];
tmp[nbEtudiants] = etudiant;

delete[] ptEtudiants;
ptEtudiants = tmp;

nbEtudiants++;
}
```

Allocation nouvel espace mémoire

Copie des adresses des anciens étudiants, puis celle du nouveau

Libération de l'ancien espace mémoire

Utilisation du nouvel espace

Mise à jour du nombre d'étudiants

DeptGE

Solution

- Réalisations C++:
  Etudiant::Afficher() et DeptGE::AfficherEtudiants()

```
void Etudiant::Afficher() const
{
  // cout << Nom << Prenom << Mail;
  Individu::Afficher();
  cout << Promo << Groupe;
}
```

Non !
Pour la classe dérivée les champs privés de la classe mère sont inaccessibles

Ok, méthode publique de la classe mère

```
void DeptGE::AfficherEtudiants() const
{
for( int i=0; i<nbEtudiants; i++)
  {
  cout << "Etudiant " << i << " : ";
  ptEtudiants[i]->Afficher();
  cout << endl;
  }
}
```

ptEtudiants est du type: Etudiant **

Exercice équipe de sport Dept ***

- A partir de la modélisation d'un département (classes Dept, Individu, Etudiant, Enseignant, …), modéliser **une équipe** dont les joueurs sont un mélange d'étudiants, d'enseignants, de secrétaires et de techniciens appartenant tous à un même département.
  - On voudra afficher la liste des joueurs d'une équipe, en précisant tous les attributs de chaque joueur (méthode afficher)
  - Pour les feuilles de match, on ne souhaite afficher que le nom et le prénom (méthode AfficherFeuilleMatch())

Article de journal ***

Enoncé

- Modéliser les différents éléments d'un article de journal. La modélisation sera focalisée sur la mise en forme du texte (Titre, SousTitre(s), Auteur(s), Illustration(s), CorpsTexte(s)). Prévoir l'accès aux différents champs et l'affichage complet de l'article.
- On dispose d'une classe *FaitActualité*. Quel lien existe-t-il entre cette classe et la classe précédente ?

Commode de rangement **

Enoncé

- Modéliser les différents éléments du système suivant:
  - une commode de 3 tiroirs permettant de ranger des vêtements (Pull, Chemise, TShirt, Chaussettes),
  - Les tiroirs ont un volume donné (20),
  - Les vêtements sont caractérisés par un nom, une couleur et un volume d'encombrement:
    - Un Pull : nom=« Pull », encombrement = 4
    - Une Chemise: nom= « Chemise », encombrement = 3
    - Un TShirt : nom=« TShirt », encombrement = 2
    - Une paire de Chaussettes : nom=« Chaussette », encombrement = 1
  - Pour chaque vêtement, on dispose d'une méthode 'Enfiler' spécifique à chaque vêtement
- On veut pouvoir Ajouter, Enlever, Chercher et Afficher le contenu d'un tiroir

Course *

Enoncé

- Il s'agit de gérer les temps et les participants d'une course. Les coureurs peuvent s'inscrire quand ils veulent et partent quand ils veulent.
- Pour gérer la course, on souhaite
  - Ajouter un participant, modifier son heure de départ et son heure d'arrivée
  - Afficher les temps de tous les participants (ordre alphabétique)
  - Savoir combien de participants sont inscrits, courent et sont arrivés
  - Savoir si la course est terminée (tout le monde est arrivé)
  - Afficher la liste des participants dans l'ordre des temps croissants

➔ Modéliser le système.

Polynôme *

Enoncé

- Modéliser la classe ***Polynome*** permettant de:
  - stocker les n+1 coefficients du polynôme d'ordre n
  - Accéder à chaque coefficient (lecture et écriture)
  - Evaluer le polynôme pour ***x*** donné (P(***x***) =…)
  - Additionner deux polynômes et retourner le nouveau polynôme
- Ecrire en C++
  - les fonctions d'accès aux coefficients
  - la fonction pour évaluer P(***x***)

$P(x)=a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \rightarrow (\ldots(a_n . x + a_{n-1}).x + \ldots + a_1)x + a_0$

TD 0 – Rappels

Préparation : relire cours IF1 et IF3, noteCpp.pdf

Exercice 1 :

1-a Ecrire un programme (en C++) convivial demandant à l'utilisateur 2 nombres entiers. Le programme affiche ensuite : la somme, la différence, le produit, le quotient, le reste de la division entière entre ces 2 nombres.

1-b Modifier ce programme en demandant à l'utilisateur quelle opération il veut réaliser. Le calcul se fera dans une fonction (on passera les 2 nombres et l'opération à faire).

Exemple d'exécution du programme à réaliser :

```
Entrer le premier nombre : 12
Entrer le deuxième nombre : 2
Opération à effectuer ? ( +, -, *, /, %) : *
12*2 = 24
```

Exercice 2 : Tableau et Fonctions

2-a Ecrire un programme permettant de saisir le contenu d'un tableau de nombres réels. Le programme affichera ensuite les valeurs entrées. Le nombre de valeurs à entrer (nombre d'éléments) est demandé à l'utilisateur (utiliser pour l'instant une allocation statique de la mémoire : `double tab[100];`).

2-b Créer la procédure **`AfficherTableau`** qui permet d'afficher tous les éléments d'un tableau (le nombre d'éléments du tableau doit nécessairement être passé en paramètre). Modifier le programme afin d'utiliser cette fonction.

2-c Créer la fonction **`SaisirNbElements`** qui demande à l'utilisateur le nombre d'éléments à entrer. Utiliser cette fonction dans le programme précédent.

2-d Ajouter au programme précédent la procédure **`SaisirTableau`**.

2-e Modifier le programme afin d'utiliser une allocation dynamique pour le tableau : le nombre d'éléments à entrer sera alors égal à la taille du tableau.

Exemple d'allocation dynamique :

```
int nb ;
double *tab = NULL ;// declaration + affectation du pointeur
cout << "Nombre d'elements ? ";
cin >> nb ;
tab = new double[nb] ;  // allocation de l'espace memoire
… // utilisation du tableau
…
delete[] tab ;  // libération de l'espace mémoire
```

2-f Faire la fonction **`AdditionneTableaux`**, permettant de faire la somme terme à terme de deux tableaux de même taille. Tester cette fonction avec 2 tableaux (tab1 + tab2) puis avec 3 tableaux (le but est de faire la somme termes à termes de trois tableaux).

TD MOO/Langage C++
Séance 1/6

TD1 – Classe Tableau

Le but : réaliser une classe qui gère automatiquement un tableau dynamique de double.
Utilisation : encapsulation (une classe), allocation dynamique, passage par référence.
Compréhension des mécanismes par défaut du C++, operator=, constructeur copie.
Rôle et intérêt du débogage.
Utilisation de QtCreator.

Préparation : TD0 et exercice 1

Exercice 1 : La classe TableauDouble, modélisation

Dans cet exercice, il s'agit de modéliser la classe `TableauDouble` permettant la gestion des tableaux dynamiques de double.

1-a Modéliser en UML la classe `TableauDouble`. Les champs seront
- *Pt* : Un pointeur sur des ***double***
- *Taille* : le nombre de ***double*** stockés
- *Nom* : un nom pour le tableau (utilisé pour l'affichage des valeurs)

Les fonctionnalités de base de la classe seront :
- l'accès en lecture **et** en écriture (cf. question 1-d) à la taille du tableau,
- l'accès en lecture **et** en écriture à chaque élément du tableau,
- l'accès en lecture **et** en écriture au nom du tableau,
- le produit scalaire avec un autre objet `TableauDouble`
- l'affichage du contenu du tableau et de son nom,
- un constructeur utilisateur, auquel on passe la taille du tableau,
- un constructeur de copie,
- un destructeur.

1-b Quelles relations et multiplicités modélisent les liens entre la classe `TableauDouble` et les classes de ses champs ? Justifier.

1-c Quelles visibilités utiliser pour les 3 champs de `TableauDouble`?

1-d En C++, quelles sont les étapes à réaliser par la méthode d'accès en écriture au champ *Taille* ?

1-e En C++, pourquoi le passage de paramètre par copie **ne** doit **pas** être utilisé pour passer des objets de type `TableauDouble`?

1-f En C++, pourquoi faut-il écrire un destructeur dans cette classe ? Quel est l'intérêt de le faire figurer dans le diagramme de classe UML ?

Exercice 2 : Réalisation et tests

2-a Récupérer les sources fournies (squelette), créer le projet sous QtCreator et ajouter les commentaires à la fonction *TableauDouble* ::*Copie(…)*.

2-b Compléter les fonctions *GetTaille* et *SetNom* dans *TableauDouble.cpp*. Utiliser le *main* fourni pour tester vos méthodes et debugger. Ci-après un exemple d'exécution du *main* : (en gras : les valeurs rentrées par l'utilisateur)

```
Entrer le nom du tableau : Toto
Entrer la taille de Toto : 4
Entrer les 4 valeurs de Toto :
Toto[0]= 1
Toto[1]= 2
Toto[2]= 3
Toto[3]= 4
-----
Toto[4] = [1, 2, 3, 4]
```

2-c Ecrire la fonction *TableauDouble* ::*SetTaille* en vous inspirant de la fonction *TableauDouble* ::*Copie*. Tester la fonction dé-commentant le *main*.

2-d Toujours en s'inspirant de la fonction *TableauDouble ::Copie*, **créer le constructeur de copie**. Pourquoi est-il obligatoire pour la classe *TableauDouble* ?

2-e Ajouter et implémenter la fonction membre ***Plus*** qui réalise la somme termes à termes de deux tableaux de même taille. Son utilisation (dans le *main*) sera :

```
TableauDouble a(10),b(10) ;
// initialisation des valeurs des tableaux
…

TableauDouble v( a.Plus(b) ) ;
v.Affiche() ;
```

(**2-e'** Sur le même principe d'utilisation, ajouter et implémenter la fonction membre ***ProduitScalaire*** pour le calcul du produit scalaire de deux tableaux.)

2-f L'utilisation des fonctions *Plus* n'est pas très commode… On préférerait utiliser les lignes suivantes :

```
TableauDouble w(1) ;
w = a.Plus(b) ;
w.Affiche() ;
```

Exécuter pas à pas (debug) les lignes précédentes et déterminer quelles sont les méthodes exécutées par ces 3 lignes de code (ordre chronologique) ? Pourquoi ce code ne marche pas?

2-g Pour résoudre le problème précédent, on propose de surcharger l'opérateur = du C++. Son rôle sera de dupliquer un objet en mémoire. Les mécanismes du C++ imposent que cet opérateur retourne une référence sur l'objet courant. Voici sa déclaration :

```
TableauDouble & operator=(const TableauDouble &t);
```

Sa définition suit le modèle recommandé d'operator=, rappelé ci-après :

```
TableauDouble & TableauDouble ::operator=(const TableauDouble &t)
{ if( this != &t )
         { // instructions pour désallouer l'objet en cours
           // puis réallouer à la taille de 't'
           // puis copier, valeurs par valeurs les éléments de t
         }
  return (*this);
}
```

➔ Implémenter et tester votre ***operator=*** .

TD 2 – Premières classes

Préparation : Exercice 1

Exercice 1 : Préparation, modélisation UML

Voici le diagramme de la classe **`Point`** ainsi que sa définition (.h) et son implémentation (.cxx). Cette classe permet de représenter et de manipuler un point du plan. Le but sera de manipuler des formes géométriques basées sur cette classe.

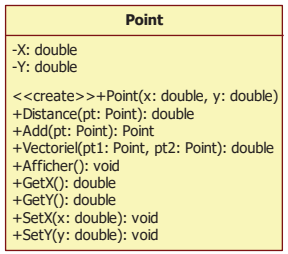


Diagramme de la classe Point

```
#ifndef _Point_h_
#define _Point_h_



class Point
{
private:
    double X;
    double Y;

 public:

 Point(double x = 0, double y = 0);

 Point Add(const Point &pt) const;

 void Afficher() const;

 double GetX() const;
 double GetY() const;

 void SetX(double x);
 void SetY(double y);

};


#endif
```

TD_Point.h

```
#include <iostream>
#include <math.h>
#include "TD_Point.h"

using namespace std ;

Point::Point(double x, double y)
{
 X=x;
 Y=y;
}

Point Point::Add(const Point &pt) const
{
 Point result(pt.X + X, pt.Y + Y);
 return result;
}

void Point::Afficher() const
{
 cout<< "(" << X << " ," << Y << " )";
}

double Point::GetX() const
{ return X; }

double Point::GetY() const
{  return Y; }

void Point::SetX(double x)
{  X = x; }

void Point::SetY(double y)
{  Y = y; }
```

TD_Point.cxx

1-a Analyser le programme suivant : comprendre chaque ligne du code et donner les affichages produits.

```
#include <iostream>
#include "TD_Point.h"

using namespace std ;

int main( void )
{
cout << " #### exo 1 #### " << endl;

Point pt1(1,1);
Point pt2;
Point * _pt;

pt2 = Point(3,1);
_pt = new Point();

cout << "Position de pt1 : ";
pt1.Afficher();
pt1.SetX( 2 ); pt1.SetY( 3 );
pt1.Afficher();
cout << endl;

cout << "Position de pt2 : ";
pt2.Afficher();
cout << endl;

*_pt = pt1.Add(pt2);

cout << "Position de pt3 : ";
_pt->Afficher();
cout << endl;

delete _pt;

cout << "Appuyer sur Entree pour continuer" << endl;
cin.get();
return 0;
}
```

Programme à analyser

1-b Donner le code C++ de la fonction membre « **`Distance`** » de la classe **`Point`** qui calcule la distance entre deux points : c'est-à-dire entre le point courant et un point passé en paramètre à la fonction.

1-c Modéliser (UML) une classe **`Triangle`**, permettant de définir un triangle dans le plan à partir de 3 objets **`Point`**. Donner à cette classe les fonctionnalités suivantes :

- Les fonctions d'accès en lecture et écriture aux champs (les 3 points).
- *Affichage* : pour afficher les coordonnées des 3 points.
- *Perimetre* : qui retourne le périmètre du triangle.
- *Translation* : qui translate le triangle par un vecteur (utiliser un objet Point pour le vecteur).
- *Aire* : qui retourne l'aire du triangle (formule de Héron ou ½ périmètre).

$$Aire = \sqrt{p.(p-a).(p-b).(p-c)} \quad avec\ p = (a+b+c)/2$$

Avec a, b et c les longueurs des cotés

1-d Modéliser en UML (ne pas implémenter les fonctions) la classe **`Quadri`** reprenant toutes les fonctionnalités de la classe **triangle**, mais pour un quadrilatère… Un quadrilatère sera défini par 4 points (les cas « papillon » ne seront pas considérés).

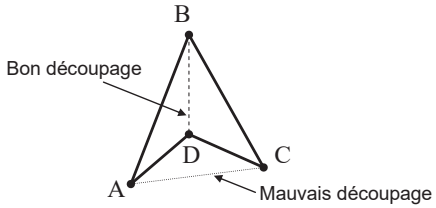
`TD_Triangle.h`, `TD_B_Triangle.cpp`, `TD_Forme.h`, `TD_Forme.cpp`), ainsi qu'un exemple d'utilisation (`TD_main.cpp`). Vous trouverez ces fichiers **dans le sous-dossier "Sources_Question_2d"** de l'archive TD_2.zip.

Remarques pour le calcul de l'aire des quadrilatères : (on ne considère pas les quadrilatères papillons, ni triangles...)

- Un quadrilatère peut se décomposer en 2 triangles et ce, de 2 manières différentes.
- Dans le cas des quadrilatères **concaves**, un seul des 2 découpages conduit au bon calcul de l'aire. Pour trouver le bon découpage il faut trouver les 2 points non connexes dont les 2 angles pour passer d'un point à l'autre sont de signes opposés. Exemple, dans la figure ci-dessous les 2 découpages possibles, et les triangles obtenus, sont :
  - segment de coupe [AC], triangles (C, A, D) et (C, A, B) : mauvais découpage,
  - segment de coupe [BD], triangles (B, D, A) et (B, D, C) : bon découpage.

Les angles pour le découpage suivant [AC] sont ADC et ABC : ils sont de même signe
➔ mauvais découpage.
Les angles pour le découpage suivant [BD] sont DAB et DCB : ils sont de signes opposés
➔ bon découpage.

Exemple de quadrilatère concave et des découpages pour le calcul de l'aire :



… Ainsi on pourra montrer que :

- pour un quadrilatère non-concave, la somme des aires obtenues pour chaque découpage est la même ;
- pour un quadrilatère concave, la somme des aires du bon découpage est plus petite que celle obtenue par le mauvais découpage…

1-e Modéliser en UML (ne pas implémenter les fonctions) une classe **`Cercle`** reprenant toutes les fonctionnalités de la classe triangle, mais pour un cercle… Un cercle sera défini par un point « centre » et un rayon.

1-f Identifier dans les classes `Triangle`, `Carre` et `Cercle` les points communs : champs et fonctions membres. Comment modifier ces classes pour rendre les fonctions membres semblables (même algorithme) ? Existe-t-il une (ou plusieurs) fonction(s) membre des trois classes non compatible avec votre schéma ?

1-g A partir des réponses précédentes, proposer une modélisation hiérarchisée de cette application.

Exercice 2 : Réalisation, premières classes

2-a Valider les résultats de la question **1-a** et créer un projet incluant les fichiers `TD_main.cxx`, `TD_Point.h` et `TD_Point.cxx`. Ces 3 fichiers sont sur le réseau (*moodle → GE → support de cours → Informatique → Moo*) **dans le sous-dossier "Sources_Question_2a"** de l'archive TD_2.zip

NB : Pour valider le déroulement du programme (constructeurs, pointeurs, appels des fonctions membres, …), utiliser le débogueur et au besoin modifier ces fichiers.

2-b Implémenter la fonction membre « **`Distance`** » de la classe **`Point`** (question **1-b**) et tester cette méthode dans le programme.

2-c Implémenter votre classe **`Triangle`** modélisée à la question **1-c**.

… et héritage

2-d Voici le diagramme de classes proposé pour mutualiser le code (correspondant à une solution des questions **1-d**, à **1-g**).

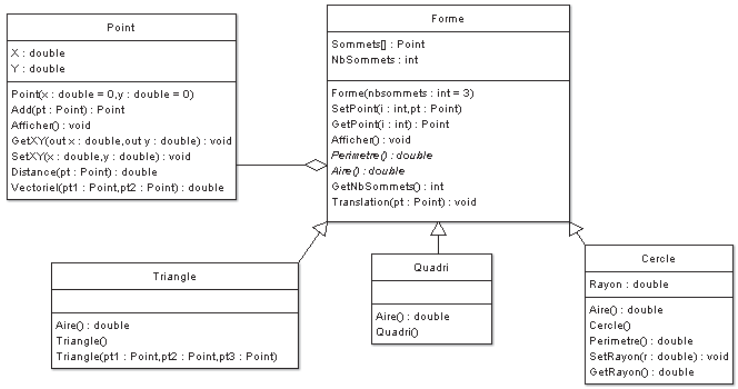


Diagramme de l'application

Ecrire en C++ les nouvelles classes **`Quadri`** et **`Cercle`** en vous appuyant sur les classes **`Point`**, **`Forme`** et **`Triangle`** fournies sous Moodle (`TD_Point.h`, `TD_Point.cpp`

TD 3 – Classe *Complexe*

Le but de ce TD est de faire une classe **Complexe** efficace en termes de :

- mémoire, (utiliser le moins possible de mémoire)
- temps d'exécution (le plus court possible)
- utilisabilité (ergonomie, intuitif, confort d'interaction avec l'utilisateur, temps minimal requis pour atteindre un résultat, grande souplesse d'utilisation, …)

Préparation : UML et rappels

1-a En s'appuyant sur l'illustration du cours, donner une modélisation complète de la classe **Complexe**. Les fonctionnalités des objets seront :

- les 4 opérations arithmétiques avec des complexes
- les 4 opérations arithmétiques avec des « double »
- calculs (par retour de valeur) des : conjugué, module, argument
- initialisation à partir de 0, 1 ou 2 *double*, et à partir d'un autre complexe
- affichage
- un test d'égalité

Pour favoriser l'utilisation, les fonctions seront le plus possible des « operator ». Par exemple : operator+, operator ==, …

1-b Comment calculer l'argument d'un ***Complexe*** ? Donner le code C++ de la fonction.

1-c Combien de fonctions « operator+ » existent dans cette classe ? Sur quoi le compilateur s'appuiera-t-il pour exécuter la bonne fonction ? Comment se nomme ce principe ?

1-d Quels sont les intérêts d'utiliser des passages des paramètres par référence pour les objets de type ***Complexe*** ? Vis-à-vis des objectifs de la classe, lequel de ces intérêts est le plus pertinent ? Pourrait-on faire autrement ?

1-e A quoi sert le mot clé ***const*** ? Pour une fonction membre, comment peut-il être utilisé ? En quoi l'ajout des mots clés ***const*** est-il pertinent pour l'utilisateur ?

Exercice 2 : Réalisation

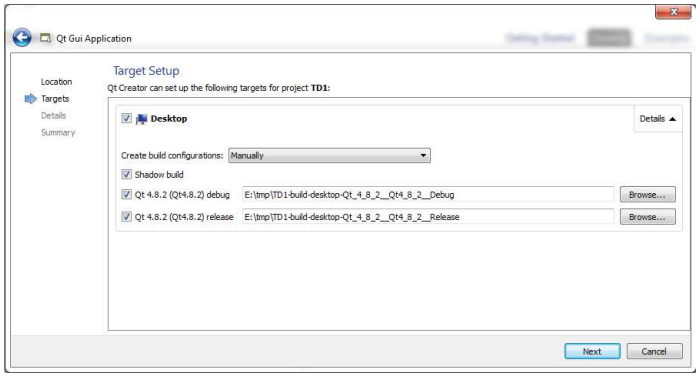
2-a Réaliser sous ***QtCreator*** votre classe. Petit à petit, c'est plus sûr… Vous pourrez utiliser le squelette *Complexe.cpp* et *Complexe.h* après les avoir téléchargés (TD3.zip).

2-b Valider votre classe ***Complexe*** avec quelques exemples simples, puis déterminer :

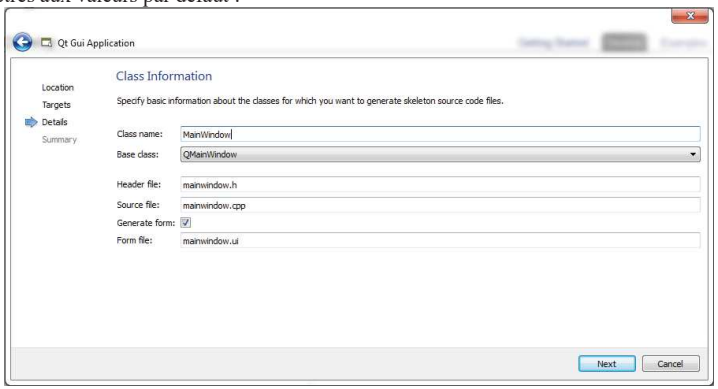
- Impédance Z à 50 Hz avec R et C en parallèle, R=50 Ohms et C=10nF
- Rotation de 45 degrés de w=1 + 0j

2-c Bon et les objectifs ? Vitesse et empreinte mémoire… et facilité d'utilisation ? Ajouter le fichier *TD3_main_Bench_CPU.cpp* et adapter le code (normalement que le nom du fichier .h inclus). Comparer les résultats entre votre classe et celle de la std. Essayer d'améliorer les performances de votre classe. Quels sont les compromis ?

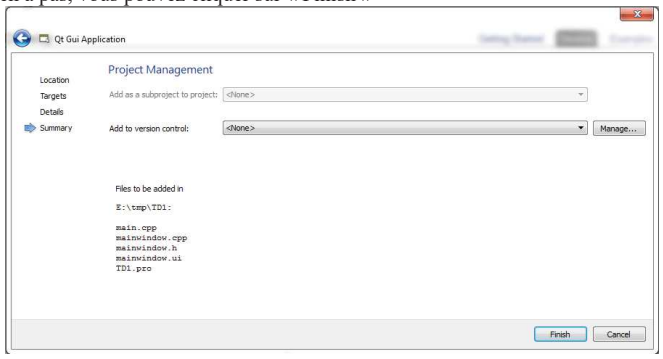
2-d Combien de constructeurs sont appelés par : ***c = c*(a/b + c) ;*** Utilisation mémoire ?



Ensuite vous pouvez choisir le nom de votre fenêtre principale. Laisser les autres paramètres aux valeurs par défaut :



Enfin, choisir un outil de gestion de version et de travail collaboratif. Dans le cadre de ces TD il n'y en a pas, vous pouvez cliquer sur « Finish »



TD 4 – Utiliser des classes existantes, IHM

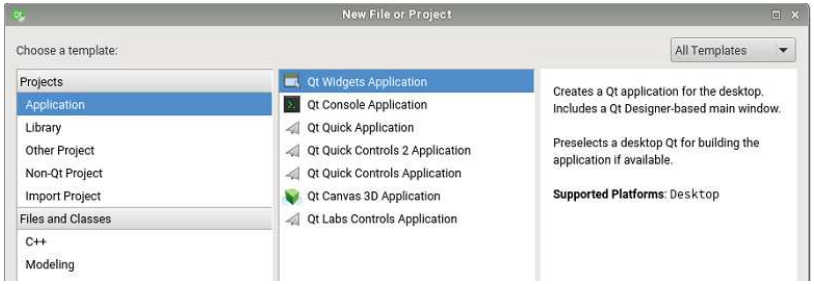
Préparation individuelle : ***maitrise pointeur/héritage,***
et *Exercice 2*

Exercice 1 : *Gui with Qt*

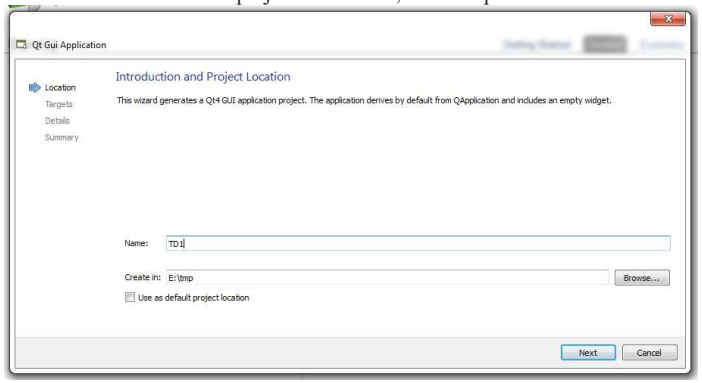
Le but de cet exercice est de faire une interface graphique pour l'exercice 1 du TD0.

1-a Faire un projet QtCreator avec IHM. Voici les étapes à suivre :

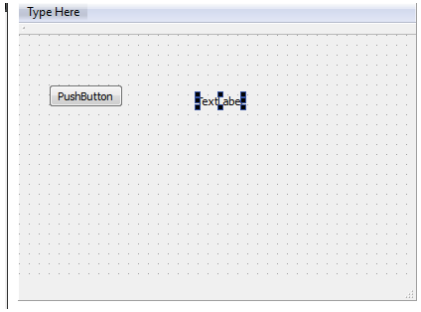
Au lancement de QtCreator, faire un nouveau projet et choisir « Qt Gui Application » ou « Qt Widgets Application » pour QtCreator 4.0 et cliquer sur « Choisir » en bas de cette fenêtre :



Donner ensuite le nom de votre projet et le chemin, terminer par « Next »



QtCreator demande ensuite de choisir les configurations de construction de votre projet. Vous pouvez laisser les valeurs par défauts. Cliquer sur « Next ».

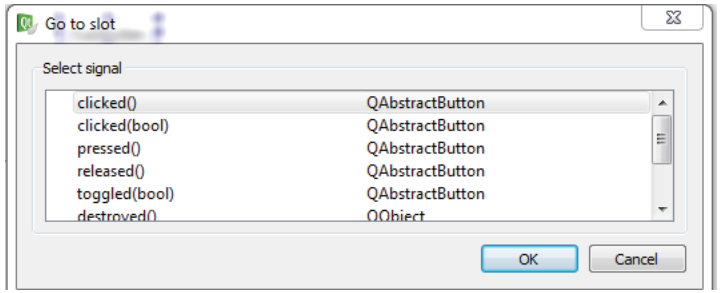


1-c Vérifier que la compilation et le lancement de l'application se fassent correctement. Rien ne se produit lors de l'appui sur le bouton puisque qu'aucun code n'a été entré lorsque l'évènement « on clique sur le bouton » se produit. C'est le but de la question suivante.

1-d Affichage d'un « Hello world » lors d'un clic sur le bouton. Pour faire ceci il faut comprendre 2 choses :

1. à chaque fois que l'on pose un *widget* sur la fenêtre, QtCreator ajoute un objet du type du *widget* dans la variable ***ui*** de la fenêtre.
2. chaque objet dispose d'évènement (cliquer, survoler, relâcher, …) sous forme de fonction que l'on va surcharger. Il s'agit des ***slot***.

Un clic bouton droit sur l'objet *PushButton*, puis « *Go to Slot* » et choisir *clicked*().

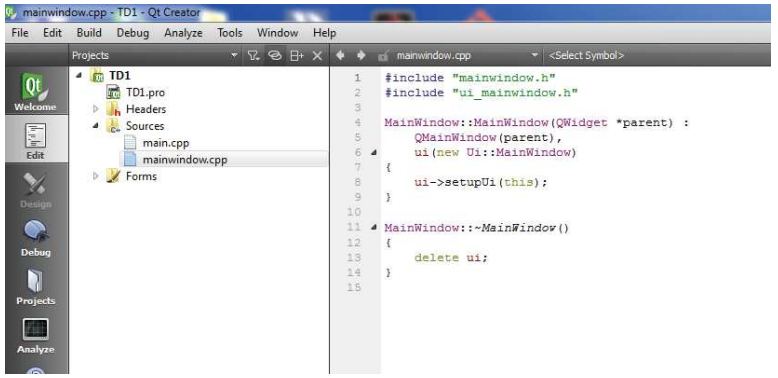


Une fonction ***on_pushButton_clicked()*** est ajoutée à la classe ***MainWindow***. Cette fonction sera exécutée lorsque le bouton sera cliqué. Ajouter les lignes suivantes à cette fonction et exécuter le programme :

```
QString txt("Hello les GE");
ui->label->setText(txt);
```

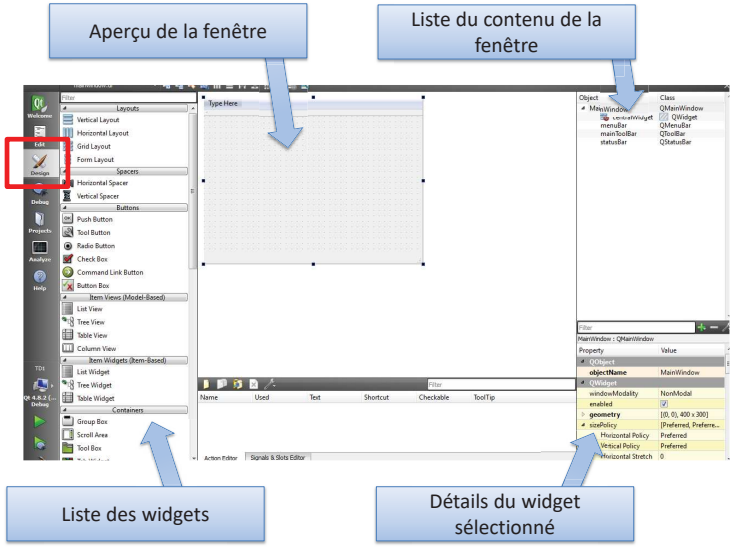
1-d Ajouter 2 ***Double Spin Box*** et une ***ComboBox*** à votre fenêtre et réaliser l'exercice 1b du TD0 (opérations +,-, /,*, %). Les calculs ne doivent pas être écrits dans la classe MainWindow mais dans un fichier cpp séparé. Votre application doit ressembler à ceci :

QtCreator crée votre projet et les fichiers de base. On obtient un affichage sensiblement similaire à celui-ci :



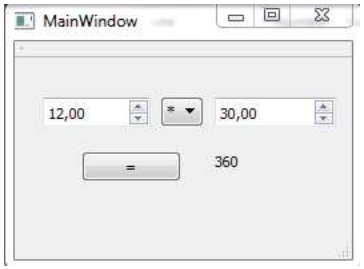
1-b Ajouter les objets à l'UI : *Label* et *PushButton*. Voici la démarche.

Cliquer sur « Form » puis double clic sur le fichier *mainwindow.ui*. Vous basculez sur le mode « Design » des interfaces graphiques dont voici un descriptif :



Il est possible de changer les noms des objets soit via la « liste du contenu de la fenêtre » soit en sélectionnant un objet et en modifiant son nom dans la zone « Détails du *widget* sélectionné ». Dans la zone « détails du widget sélectionné », il est aussi possible de paramétrer graphiquement l'objet (widget) sélectionné.

Faire un *drag and drop* d'un ***Label*** et d'un ***PushButton*** sur l'aperçu de votre fenêtre. Votre fenêtre doit ressembler à ceci (il est possible de faire plus joli…) :



Utiliser la touche « F1 » pour avoir de l'aide sur les classes des *widgets* et des classes Qt.

Ce programme n'est pas très complexe, mais la programmation n'est pas si évidente !
Quelles sont les difficultés rencontrées ?
Comment pensez-vous pouvoir devenir plus efficace ?

Exercice 2 : vector

La classe ***vector*** est une classe permettant de gérer dynamiquement et efficacement des tableaux de tous types (double, char, Complexe, …). On précise le type d'éléments lors de la création d'un objet ***vector***. Des exemples d'utilisations sont donnés dans le cours. La documentation complète de la classe ***vector*** peut être consultée ici :
http://www.cplusplus.com/reference/vector/vector/

2-a Utiliser des objets de la classe ***vector*** de la std pour réaliser les mêmes additions que la question 2-f du TD0. Vous pourrez compléter le code TD_Ex2_main_TODO.cpp fourni.

2-b Questions

- Pourquoi utiliser des passages de paramètre par référence pour des objets *vector* ?
- A quoi sert le mot clé *const* lors du passage de paramètre par référence ?

2-c Utiliser la fonction ***sort*** pour trier votre tableau en ordre croissant.

2-d Utiliser la fonction ***sort*** pour trier votre tableau en ordre décroissant.

2-e Question hors préparation. La classe ***vector*** n'implémente pas les opérations arithmétiques sur des objets ***vector***. La classe ***valarray*** est dédiée à la représentation de vecteurs et à leur manipulation mathématique. Refaire les questions 2-a, 2-c et 2-d avec des objets de la classe ***valarray***.

3-d A partir d'un objet `Trois`, comment exécuter la fonction `Affiche` de la classe `Deux` ? Et celle de la classe `Un` ?

4 Les pointeurs d'objet :

Créer un pointeur de type `Trois` et l'affecter à un espace alloué à un objet `Trois`. C'est-à-dire :

```
Trois *_pt = new Trois( 1, 2, 3) ;
_pt->Affiche() ;  // pour les questions qui suivent
delete _pt ;      // ne pas oublier !!!!
```

4-a Le nombre et l'ordre dans lequel s'effectue les constructeurs ont il changé ?

4-b Qu'en est-il pour les destructeurs ?

4-c La fonction `Affiche` appelée est elle la plus adaptée (nombre de valeurs affichées par rapport au nombre de valeurs en mémoire) ?

5 Avantages et problèmes avec les pointeurs d'objet :

Créer un pointeur de type et `Un` et l'affecter à un espace alloué à un objet `Trois`. C'est-à-dire :

```
Un *_pt = new Trois( 1, 2, 3) ; // si si
_pt->Affiche() ;  // pour les questions qui suivent
delete _pt ;      // ne pas oublier !!!!
```

5-a Le nombre et l'ordre dans lequel s'effectue les constructeurs ont il changé ?

5-b Qu'en est-il pour les destructeurs ?

5-c La fonction `Affiche` appelée est elle la plus adaptée ?

5-d Mettre le mot clé `virtual` devant la déclaration des fonctions `Affiche`. Quelle fonction `Affiche` est maintenant appelée par l'instruction : `_pt->Affiche() ;`

5-e a Mettre le mot clé `virtual` devant la déclaration des destructeurs des classes `Un`, `Deux` et `Trois`. Avec les destructeurs `virtual`, quel est le nombre et l'ordre des destructeurs utilisés par l'instruction suivante ?

```
delete _pt ;
```

5-f Trouver une application des pointeurs sur le type le plus haut dans la hiérarchie.

6 Même questions que la **5-d** avec des références :

Créer une référence de type et `Un` et l'affecter à un espace alloué d'un objet `Trois`. C'est-à-dire :

```
Trois tr( 5, 4, 3) ;
Un &_ref = tr ;
_ref.Affiche() ;  // pour les questions qui suivent
```

7 Conclure sur les conditions de mise en œuvre du polymorphisme et les intérêts du polymorphisme.

TD 5– Héritage

Le but de ce TD :

- maitriser l'écriture de nouvelles classes simples (exemple pédagogique),
- maitriser les mécanismes d'héritage et de surcharge
- mettre en œuvre le polymorphisme

Préparation : exercice 1 (et les notions de cours associées)

Exercice I : modélisation d'une hiérarchie de classes

Faire le diagramme UML des trois classes suivantes (seulement sur papier) :

- **`Un`** qui possède
  - un champ X de type double privé,
  - une fonction `GetX` qui retourne X,
  - une fonction `SetX` qui modifie X,
  - un constructeur permettant d'initialiser X
  - une fonction `Affiche` qui affiche X avec le texte « Un.X = »
- **`Deux`** qui dérive (ou hérite) de `Un` et qui possède en plus :
  - un champ Y de type double privé,
  - une fonction `GetY` qui retourne Y,
  - une fonction `SetY` qui modifie Y,
  - un constructeur permettant d'initialiser X et Y,
  - une fonction `Affiche` qui affiche X et Y avec le texte « Deux. (X, Y) = ».
- **`Trois`** qui dérive de `Deux` et qui possède en plus :
  - un champ Z de type double privé,
  - une fonction `GetZ` qui retourne Z,
  - une fonction `SetZ` qui modifie Z,
  - un constructeur permettant d'initialiser X, Y et Z,
  - la fonction `Affiche` qui affiche X, Y et Z avec le texte «Trois.(X, Y, Z)=».

Exercice II : réalisation d'une hiérarchie de classes

1 Implémenter votre solution en respectant les diagrammes de l'exercice **I**.

2 Vérifier le fonctionnement de vos trois classes dans une fonction main.

3 Modifier/débugger votre code pour répondre aux questions.

3-a Un objet de type `Deux` peut-il accéder directement au champ `X` ? Pourquoi ?

3-b Quand on crée un objet de type `Trois`, combien de constructeurs sont exécutés ? Dans quel ordre sont effectués les constructeurs ? Comment avez-vous fait pour trouver les constructeurs exécutés et l'ordre ?

3-c Quand un objet de type `Trois` est détruit, dans quel ordre sont exécutés les destructeurs ? Comment avez-vous procédé pour trouver l'ordre ?

Examen Méthodologie Orientée Objets

documents autorisés

Exercice : Calcul d'impédances R, L, C (50 minutes)

Le but est de concevoir une application capable de calculer, pour une fréquence donnée, l'impédance complexe d'un circuit. Un circuit ne sera composé que de résistances, capacités et inductances. Les composants pourront être associés en série et parallèle, ou en différents mélanges de ces constructions (pas de boucle, ni de potentiel).
La figure ci-dessous donne des exemples de circuits « calculables » avec cette application.

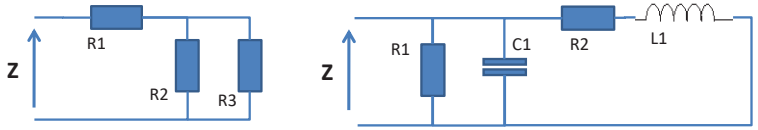


Figure 1: à gauche a) exemple avec 3 résistances; à droite, b) circuit RLC parallèle

Voici comment chaque composant sera représenté :

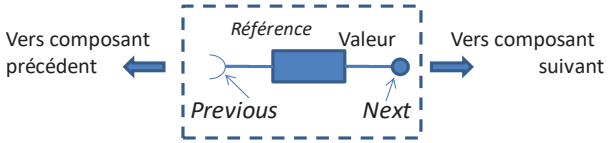


Figure 2: Schéma d'un composant

La création d'un circuit se fera via la mise bout-à-bout de plusieurs composants et par la création d'une « impedance ». C'est cette « impedance » qui permettra le calcul de l'impédance totale du circuit (à une fréquence donnée). Elle permettra également la création de branches parallèles (figure 3).

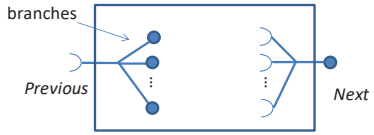
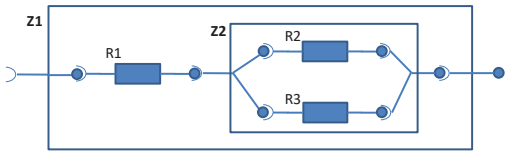


Figure 3: Schema d'"Impedance"

Comme exemple, voici la vue correspondant au premier circuit (fig1.a) et le code C++ permettant de le créer, l'afficher et le calculer pour la fréquence 0Hz.




```
Resistor R1("R1",50),
   R2("R2",100),
   R3("R3",100);
Impedance Z1("",0);
Impedance Z2("",0);
Z2.AddBranch(&R2);
Z2.AddBranch(&R3);
Z2.SetPrevious(&R1);
Z1.AddBranch(&R1);
Z1.Print();
Z1.Evaluate(0).Print();
```

Figure 4: Modélisation du circuit de la fig1.a et implémentation.

TD 6 – Un examen

Préparation individuelle :

au mieux exercice (1h max !)

au pire questions 3, 4, 6, 8 et 9 (30 min max)

Le but de la préparation est d'auto évaluer votre capacité à répondre à l'exercice d'un examen dans le temps imparti.

Au travers de l'exercice, les points suivants seront abordés :

- le polymorphisme,
- la gestion d'objets dynamique (avec la classe *std ::vector*)
- un mécanisme de chainage proche des listes.

Le but de la séance est de réaliser sous QtCreator votre application (les sources à compléter sont données).

Les étudiants les plus avancés pourront utiliser les outils graphiques vus en 3GE (IF1). pour tracer les diagrammes de Bode.
Il est possible d'étendre cette modélisation aux quadripôles, puis d'ajouter des éléments actifs (sources tension ou courant). Dans ce cas, le calcul d'impédance sera complété par ceux des tensions et courants.
Enfin, les valeurs des éléments peuvent aussi dépendre de grandeurs mesurées dans le circuit…

Partie 1 : Modélisation des composants R, L et C (15 minutes)
La modélisation des composants sera basée sur l'utilisation de la classe « ***Component*** » dont le code C++ est donné ci après.

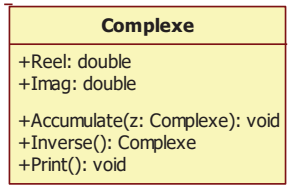
```
class Component
{
private:
      Component   *Previous;
      Component   *Next;
public:
      double      Valeur;
      string      Name;
      Component(string name, double v): Name( name ), Valeur( v )
            { Reset();  }
      void Reset()
            { Previous = NULL; Next = NULL;     }

      Component*  GetPrevious() { return Previous; }
      Component*  GetNext() { return Next; }

      void  SetPrevious( Component *comp )
            {// unlink
            if( Previous != NULL )
                  Previous->Next=comp;
            // make new link
            Previous = comp;
            Previous->Next = this;
            }
      virtual Complexe  Evaluate(double freq)=0;
      virtual void Print() { cout <<Name<<"("<<Valeur<<")"; }
};
```

1) Donner le diagramme UML de cette classe.
2) Quel type de relation lie *Previous* et *Next* à la classe « ***Component*** » ?
3) Peut-on créer un objet de type « Component » ? Quelle est la spécificité de cette classe ?
4) En vous appuyant sur la classe ***Component***, modéliser les classes ***Resistor***, ***Capacitor*** et ***Inductor***. **Justifier vos choix (méthodes, champs, associations).**
5) Ecrire le code C++ de la fonction *Evaluate* de la classe ***Capacitor***. Cette fonction simple retourne l'impédance ***Complexe*** du composant courant calculée à la fréquence passée en paramètre. La classe ***Complexe*** à utiliser est donnée ci-après.

Partie 2 : Classe Complexe (10 minutes)
On étudie la classe ***Complexe*** utilisée pour les calculs de cette application.




```
class Complexe
{
public:
      double Reel;
      double Imag;
      void Accumulate( Complexe z )
            { Reel += z.Reel;
              Imag += z.Imag; }
      Complexe Inverse() ; // = 1/z
      void Print() ;
};
```


6) En utilisant des objets de cette classe, donner le code C++ du calcul de l'impédance (complexe) totale de 3 impédances (complexes) en parallèles.
7) Modifier l'exemple de la question 6 en utilisant les fonctions *Evaluate* d'un objet ***Resistor***, un objet ***Capacitor*** et un objet ***Inductor*** ainsi qu'une fréquence (choisir les valeurs arbitrairement).

Partie 3 : Classe « Impedance » (15 minutes)
On s'intéresse maintenant à la classe ***Impedance***, qui est la classe centrale dans le calcul d'impédance des circuits série et parallèle. Cette classe utilise la classe **vector** pour la gestion dynamique des branches. On donne comme extrait de cette classe la fonction *Print()* qui affiche le nom et valeur de tous les composants contenu dans l'impédance :

```
void Impedance ::Print()
            {
            cout<<Name<<"[ ";
            if( !Branch.empty() ) // il y a des branches ?
            for(unsigned int i=0; i<Branch.size(); i++)
                  {
                  Component *comp = Branch[i];
                  do    {
                        comp->Print();
                        comp = comp->GetNext();
                        if( comp != NULL ) cout<<" + ";
                        }while( comp != NULL ); // branch pas finie ?
                  if( (Branch.size() > 1) && (i < Branch.size()-1))
                         cout<< " // ";
                  }
            cout<<"]";
            }
```

8) Justifier le fait que la classe ***Impedance*** hérite de ***Component***.
9) Donner le diagramme UML de la classe ***Impedance***. (cf. introduction !)
10) Ecrire en C++ la fonction *Evaluate()* de la classe ***Impedance***. (cf. questions 6 et 7)
*Pour le TD la classe **Impedance** est à faire.*

Fin.