DAUPHINE UNIVERSITÉ PARIS | TUNIS

Le langage C++
Master Actuariat
Séance 7 : Les classes (suite)

Plan

DAUPHINE UNIVERSITÉ PARIS | TUNIS

- Rappel : Classe Complex
- Getter/Setter (Accesseurs et mutateurs)
- Encapsulation
- Constructeurs / Destructeur
- Surcharge des opérateurs

Rappel de la classe Complex

```
class Complex {
private:  //n'est pas nécessaire – Par défaut private
double a,b;
public: // Liste des méthodes pour manipuler des nbres complexes
double abs( );
...
}
```


Utilisation des nombres complexes

```
…
Complex z1, z2; //OK
z1.a = 2.2; z1.b = 3.4; //Erreur !!
```

Problème : a et b sont des membres **private** de la classe Complex ➔ Ils ne peuvent pas être lûes ni modifiées en dehors des méthodes de la classe.

Solution: définir une interface (== collection de méthodes spéciales) portant l'étiquette ***public***, pour lire/modifier des membres privées de la classe ➔ **Getter/Setter**

Getter/Setter (Accesseurs et Mutateurs)

- Les fonctions membres permettant d'accéder aux données membres sont appelées **Getter** (ou **accesseurs).**

- Les fonctions membres permettant de modifier les données membres sont appelées **Setter** (ou **mutateurs).**

Getter (Accesseur)

Getter : Fonction membre permettant de **récupérer le contenu d'une donnée membre protégée**. Un accesseur doit avoir comme type de retour le type de la variable à renvoyer et ne doit pas nécessairement posséder d'arguments

Syntaxe :

```
class MaClasse
{
private :
   TypeDeMaVariable  MaVariable;
   public :
TypeDeMaVariable  GetMaVariable();
};
// définition de l'accesseur GetMaVariable
TypeDeMaVariable MaClasse::GetMaVariable()
{
 return MaVariable;
 }
```

Setter (Mutateur)

Setter : fonction membre permettant de **modifier le contenu d'une donnée membre protégée**. Un mutateur doit avoir comme paramètre la valeur à assigner à la donnée membre. Le paramètre doit donc être du type de la donnée membre ne doit pas nécessairement renvoyer de valeur (il possède dans sa plus simple expression le type *void*)

```
class MaClasse
{ private :
TypeDeMaVariable  MaVariable;
public :
void  SetMaVariable(TypeDeMaVariable);
};
MaClasse::SetMaVariable(TypeDeMaVariable  MaValeur)
{
MaVariable = MaValeur;
}
```


Getter/Setter: Exemple Complex

Complex.h

```
class Complex {
double a,b;
public:
double get_a();
double get_b();
void set_a(double);
void set_b(double);
};
```

Getters qui permettent d'accéder aux valeurs respectives des données membre privées _a et _b

Setters qui permettent de modifier les données membre privées _a et _b

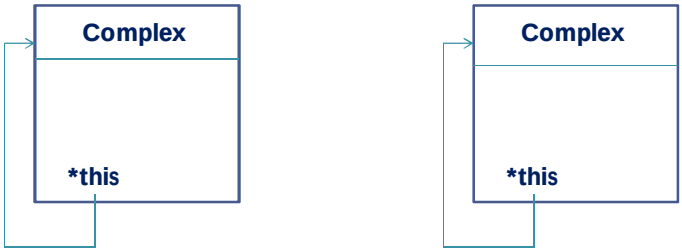
Complex.cpp

```
// Dans le fichier Complex.cpp
double Complex::get_a()
{ return _a; }
double Complex::get_b()
{ return _b; }
void Complex::set_a(double new_a)
 { _a = new_a; //ou encore this->_a=new_a; }
void Complex::set_b(double new_b)
 { _b = new_b; }
```

*Le mot clé **this** désigne un pointeur vers l'instance courante de la classe elle même.*

Le pointeur this

Dans toutes les classes, on dispose d'un pointeur ayant pour nom *this. Ce pointeur pointe **vers l'objet actuel.***



Chaque objet (ici de type Complex) possède un pointeur this qui pointe vers l'objet lui-même !

Getter/Setter: Exemple Complex

```
class Complex {
double _a,_b;
public:
double get_a() { return _a; } // fonction inline
double get_b() { return _b; // ou return this->_b;}
void set_a(double new_a) { _a = new_a; }
void set_b(double new_b) { _b = new_b; }
};
```

***En règle générale,** les fonctions membres **non inline** devront être déclarées dans le fichier **source .cpp** de la classe, pour éviter d'avoir plusieurs fois la même définition au moment de l'édition des liens, et les fonctions membres **inline** devront être déclarées dans le fichier d'entête **.hpp** de la classe, pour permettre au compilateur de générer le code à chaque appel de la fonction.*

Utilisation des Getter/Setter

Le fichier main.cpp

```
Complex z1,z2;

z1.set_a(2.3);
z1.set_b(3.7);

//Maintenant z1 = 2.3 +3.7i
…
cout << "z1 = " << z1.get_a() << " + i " <<
z1.get_b();
```

Pas d'accès direct aux données membres _a et _b sur l'instance de Complex z1 en utilisant z1._a : il faut obligatoirement passer par des méthodes publiques.

Encapsulation

- Il faut **éviter de donner un accès extérieur** aux **données membres** d'un objet quelconque.
- On va **interdire l'accès à certaines données membres d'une classe ou certaines méthodes** en utilisant le mot clé **private**.
- On **ne peut accéder à une variable** (ou une méthode membre) **privée que par l'intérieur de la classe.**
- par contre, on **peut accéder librement** à **toutes** les données membres ou méthodes membres **publiques**.
- Cette technique fondamentale permet d'empêcher au programmeur de faire n'importe quoi : **il ne pourra accéder à ces données que par les méthodes publiques**

Interface et boîte noire

- **Vu de l'extérieur, on ne peut accéder à un objet donné que grâce aux méthodes publiques.**
- **L'ensemble des méthodes publiques est appelée l'interface de l'objet.**
- **De l'extérieur, l'objet peut être vu comme une boîte noire qui possède une interface d'accès.**
- **On cache ainsi à l'utilisateur de cette classe comment cette interface est implémentée : seul le comportement de l'interface est important.**

Constructeur (ctor)

- Lorsqu'on déclare un objet d'une classe donnée (ou, plus exactement, une variable de type cette classe), **cet objet n'est pas initialisé**. Ceci est particulièrement gênant pour les objets dynamiques, c'est-`a-dire ceux qui réclament une allocation dynamique de mémoire, tels que les listes chaînées.
- Cette initialisation peut se faire de façon automatique grâce à une fonction membre particulière appelée **constructeur**.

Constructeurs (ctor)

- Le constructeur est une **fonction membre** qui sert à **initialiser** les **données membres** de l'objet
  - Systématiquement appelé quand un objet est instancié.
  - N'a pas de type de retour pas même **void**
  - Porte le nom de l'objet
  - ne comporte pas d'instruction ***return***
- Une même classe peut avoir **plusieurs constructeurs** qui doivent alors avoir des signatures différentes

Constructeur par défaut

- **Constructeurs particuliers :**
  - ✓ **Constructeur par défaut**
    c'est le constructeur sans argument. Si aucun constructeur n'est présent explicitement dans la classe, il est généré par le compilateur.

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H
#include <iostream>
class Complex
{
double _x,_y;
 public :
Complex(); // Constructeur par
                //défaut
  ....
};
#endif   //COMPLEX_H
```

Complex.cpp

```
#include "Complex.h"

....
Complex::Complex()
{
 _x=0;  // ou this->_x=0;
 _y=0 ;// ou this->_y=0;
}

…
```

Constructeur par défaut

Lorsqu'un constructeur par défaut existe, on peut déclarer une instance de classe **sans préciser de paramètres :**

main.cpp

```
#include " Complex.h"

int main()
{
  Complex z1; // appel du
  //contructeur par défaut
  // z1=0.0+i0.0
....

return 0;
}
```


Constructeur par défaut avec liste d'initialisation

Le C++ permet d'initialiser les attributs de la classe d'une autre manière appelée **liste d'initialisation.**

Complex.cpp

```
#include "Complex.h"

....
Complex::Complex()
{
  _x=0;  // ou this->_x=0;
  _y=0 // ou this->_y=0;
}

...
```

```
#include "Complex.h"
....
Complex::Complex():_x(0),_y(0)
{
// Rien à mettre dans le corps du constructeur, tout a //déjà été fait !
}
```

Le prototype du constructeur (dans le .h) ne change pas. Toute la partie qui suit les deux-points n'apparaît pas dans le prototype.

Surcharger le constructeur

- En C++, on a le droit de surcharger les fonctions, donc de surcharger les méthodes. Et comme le constructeur est une méthode, on a le droit de le surcharger lui aussi.
- Cela permet de créer un objet de plusieurs façons différentes.

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H
#include <iostream>
class Complex
{
double _x,_y;

public :
Complex(); //Constructeur par défaut
Complex(double a, double b);
   //ceci n'est pas un Constructeur par défaut

   ....
};
```

Complex.cpp

```
#include "Complex.h"
Complex::Complex()
{
  _x=0;
  _y=0
}

Complex::Complex(double a,double b) :
  _x(a),_y(b)
{

}

...
```

main.cpp

```
#include " Complex.h"

int main()
{
  Complex z1; // appel du
  //contructeur par défaut
   Complex z2(2.5, 6.5);

return 0;
}
```


Surcharger le constructeur

- Les constructeurs peuvent avoir des **arguments par défaut** comme toute autre fonction.

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H
#include <iostream>
class Complex
{
double _x,_y;

public :
Complex(double a=0.0, double b=0.0);
//  ceci peut être aussi //considéré comme le
   Constructeur          //par défaut
   ....
};
#endif   //COMPLEX_H
```

*Le constructeur par défaut est le constructeur qui ne **prend aucun argument** **OU** dont les arguments ont une **valeur par défaut**.*

main.cpp

```
#include " Complex.h"

int main()
{
   Complex z2(2.5, 6.5); //z2=2.5+i.5
   Complex z3;  //z3=0.0+i0.0
   Complex z4(3.3); //z4=3.3+i0.0

return 0;
}
```

Constructeur de copie

- Le compilateur crée aussi ce qu'on appelle un "**constructeur de copie**". C'est une surcharge du constructeur qui initialise notre objet en copiant les valeurs des attributs de l'autre objet.
  - Contient comme argument **un objet du même type**
  - Sert à créer des **clones d'objets**

Exemple constructeur de copie

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H
#include <iostream>
class Complex
{
double _x,_y;

public :
Complex();
Complex(double a, double b);

//constructeur de copie
Complex(const Complex & e);

   ....
};
#endif   //COMPLEX_H
```

Complex.cpp

```
#include "Complex.h"

Complex::Complex()
{
  _x=0;
  _y=0
}

Complex::Complex(double a,double b) :
  _x(a),_y(b)
{

}
Complex::Complex(const Complex & e)
{
  _x=e._x;  // this->_x=e._x;
  _y=e._y;
}


...
```

main.cpp

```
#include " Complex.h"

int main()
{
   Complex z1;
    Complex z2(2.5, 6.5);

    Complex z3(z1);
// z3 est un clone de z1
    Complex z4 = z2;
// z4 est un clone de z2

return 0;
}
```


F.CHAKER - C++ M1-MA (2014/2015) 22

Destructeur

Le destructeur est une méthode appelée lorsque l'objet est supprimé de la mémoire. Son principal rôle est de désallouer la mémoire (via des delete) qui a été allouée dynamiquement.

- **Fonction membre systématiquement appelée juste avant la destruction d'un objet**
- **Porte le nom de la classe et est précédé de ~**
- **Pas de type de retour**
- **Pas d'arguments**
- **Un seul par classe**
- **Permet de libérer les ressources**

```
class MyClass{
 public:
   ...
   ~MyClass(){
    //code ici
   }}
```


Destructeur de la classe Complex

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H
#include <iostream>
class Complex
{
private :
   double _x,_y;

public :
Complex();
Complex(double a,
   double b);

~Complex(); //destructeur

   ....
};
#endif   //COMPLEX_H
```

Complex.cpp

```
#include "Complex.h"
Complex::Complex()
{_x=0;
   _y=0
}
Complex::Complex(double
  a,double b) : _x(a),_y(b)
{
 }

Complex::~Complex()
{
//libération des ressources
....
}
```

main.cpp

```
#include  « Complex.h"

int main()
{
Complex z1;
Complex*  pZ= new Complex(2.5,
   6.5);

delete pZ; //  appelle le
   //destructeur implicitement pour
   //pZ cad pZ->~Complex() sera
   //appelé automatiquement

   return 0;

   // Le destructeur est
   //implicitement appelé pour z1.
}
```

Note : Destructeur

Le destructeur est **appelé automatiquement par le compilateur** lorsque l'objet va être détruit, et ce, que vous l'ayez appelé ou non.

```
#include <iostream>
class Test {
public: ~Test()
{ std::cout << "Destruction\n"; }
};
 int main()
 { Test t;
t.~Test(); // appelle explicite du destructeur
} // ici, le compilateur appelle à nouveau le destructeur
```

➢ Le destructeur est **appelé deux fois** et cela ne sert à rien puisque c'est le travail du compilateur.
➢ L'un des seuls cas où l'on doit appeler le destructeur explicitement est lorsqu'on a utilisé l'opérateur **new**.

Méthodes constantes

- Les méthodes constantes sont des méthodes de « lecture seule ». Elles possèdent le mot-clé **const *à la fin de leur prototype et*** *de leur déclaration.*
- Quand vous dites « ma méthode est constante », vous indiquez au compilateur que votre méthode ne modifie pas l'objet, c'est-à-dire qu'elle ne modifie la valeur d'aucun de ses attributs.

```
//Prototype de la méthode (dans le .h) :
void maMethode(int parametre) const;

//Déclaration de la méthode (dans le .cpp) :
void MaClasse::maMethode(int parametre) const
{
}
```

- **Exemple sur la classe Complex:**

```
double get_a() const;
double get_b() const;
```

En résumé

- Le constructeur est une méthode appelée automatiquement lorsqu'on crée l'objet. Le destructeur, lui, est appelé lorsque l'objet est supprimé.
- Il n'est jamais nécessaire d'appeler explicitement un destructeur (sauf si l'objet a été créé avec un placement new).
- On peut surcharger un constructeur, c'est-à-dire créer plusieurs constructeurs. Cela permet de créer un objet de différentes manières.
- Une méthode constante est une méthode qui ne change pas l'objet. Cela signifie que les attributs ne sont pas modifiés par la méthode.
- Puisque le principe d'encapsulation impose de protéger les attributs, on crée des méthodes très simples appelées accesseurs qui renvoient la valeur d'un attribut. Par exemple, get_a() renvoie la valeur de la partie real d'un complexe.

Qu'est ce qu'un opérateur ?

- **Définition** : un opérateur est une opération entre un ou deux opérandes, c'est-à-dire, entre une ou deux variables ou expressions.
- On distingue plusieurs types d'opérateurs, comme par exemple :
  - les opérateurs arithmétiques +, -, *, /, ...
  - les opérateurs logiques &&, !!, !, ...
  - les opérateurs de comparaisons > , >=, < , <=, ==, !=, ...
  - les opérateurs d'incrémentation et de décrémentation ++ et --,
  - l'opérateur d'affectation =, ...

Surcharge d'opérateurs : Motivation

- Pour certaine classe, la notion d'addition ou de multiplication est totalement naturelle. La surcharge d'opérateurs permet d'écrire directement *U+V* lorsqu'on veut additionner deux instances U et V d'une même classe A.
- **Principe** : Supposons que nous avons créé la classe Complex et que nous avons deux objets de type Complex que nous voulons additionner.
- En temps normal, il faudrait créer une fonction :

```
Complex z1, z2;
z1.addition(z2);
```

Exercice

Développer la méthode addition
– Le résultat doit être la somme du complexe courant (this) et du complexe donné en argument.

Solution

Surcharge d'opérateurs : Principe

- La surcharge des opérateurs permet de modifier la classe Complex de sorte à être capables d'écrire :

```
Complex z1, z2, resultat;
resultat = z1+z2;
```

- En pratique, l'appel à un opérateur est similaire à un **appel de méthode**.

A + B est traduite par le compilateur par : `A.operator+(B)`
A * B est traduite par le compilateur par : `A.operator*(B)`
etc...
A Op B est traduite par le compilateur par : `A.operator Op(B)`

Surcharge d'opérateurs : Principe

- La surcharge d'opérateurs consiste donc à définir une méthode **operatorOP** avec de nouveaux arguments.

- En C++, on peut surcharger presque tous les opérateurs
  - sauf les opérateurs :: (résolution de portée), ., .*, ?:, sizeof, typeid, et les opérateurs de conversions du C++

Intérêt de surcharger un opérateur

// sans surcharger l'opérateur	// En surchargeant l'opérateur
int main() **{** **Complex z1,z2,z3;** **z3=z1.addition(z2);** **}**	**int main()** **{** **Complex z1,z2,z3;** **z3=z1+z2;** **}**

Surcharge d'opérateurs : Principe

- **La surcharge peut se faire :**
  - à l'intérieur de la classe : comme fonction membre.
    Dans ce cas, si OP est un opérateur binaire, la notation **a OP b** sera interprétée par le compilateur comme **a.operatorOP(b)**.
  - ou à l'extérieur de la classe (non spécifique à une classe) : une fonction indépendante amie de la classe
    Dans ce cas la fonction operator sera interprétée par le compilateur comme **operatorOP(a,b)**.

Fonction membre ou Fonction amie ?

Faut-il spécifier un opérateur comme une fonction-membre ou comme une fonction ordinaire (éventuellement amie) ?

La règle générale est la suivante:

- Opérateur **unaire** (ex: ++)➔ Fonction membre
- Opérateur **binaire** (ex: +)➔Fonction amie

En effet:

- un opérateur unaire modifie par nature l'objet sur lequel il opère (a +=3 modifie a). Il est donc cohérent d'en faire une fonction-membre.
- Un opérateur binaire, par contre, opère sur *deux objets. En faire une fonction amie*

Surcharger un opérateur à l'intérieur de la classe (par une fonction membre de la classe)

Pour surcharger un opérateur **OP** à l'intérieure d'une classe A, il faut ajouter la définition de la méthode prédéfinie **operatorOP** dans la classe A.

type operatorOp(paramètres)

```
// A.hpp
class A
{
  // ...

public:
  // declaration de l'operateur
  typeDeRetour operatorOP(type1 parametre1, type2 parametre2,  ...);
};
```

```
// A.cpp
// definition de l'operateur
typeDeRetour A::operatorOP(type1 parametre1, type2 parametre2,  ...)
{
  // ...
}
```

Surcharger un opérateur à l'intérieur de la classe (par une méthode membre de la classe)

- Avec cette syntaxe, le **premier opérande** est toujours **l'objet** auquel cette fonction s'applique.
- Cette manière de surcharger les opérateurs est donc particulièrement bien adaptée pour les **opérateurs qui modifient l'objet sur lequel ils travaillent**, comme par exemple les opérateurs =, +=, ++, etc.
- **Les paramètres** de la fonction operator sont alors le **deuxième opérande**.
- Les opérateurs définis en interne renvoient souvent l'objet sur lequel ils travaillent (ce n'est pas une nécessité cependant). Cela est faisable grâce au pointeur **this, qui est un pointeur sur l'objet lui-même.**
- Les opérateurs [],(), ->, new et delete **doivent obligatoirement** être définies comme **fonction membres.**

E… asse(Fonction me…

```
// Déclaration de la classe
class Complex {
        double a,b;
public:
        Complex(double,double); // constructeur
        void affiche() const;
        Complex operator+=(Complex ); //utilisation z3=z1+z2
};
```

```
// Définition des méthodes de la classe

Complex Complex::Complex(double a, double b):a(a),>b(b)
{
}
void Complex::affiche()
{
cout<<a<<"+i "<<b<<endl;
}
Complex Complex opérator+=(Complex z2)
{
this->a+=z2.a;
this->b+=z2.b;
return *this;
}
```

```
// Utilisation de la classe

int main()
{
Complex z1(2.2,3.3),z2(4.4,5.5),z3;
z1.affiche();
z2.affiche();
z1+=z2; / / le + sera interprété comme z1.operator+=(z2)
z1.affiche();
}
```


Opérateurs et transmission par référence

Dans l'exemple précédent, on utilise la transmission **par valeur**.
Pour des objets de grande taille, il vaut mieux utiliser la transmission **par référence**.
On peut alors protéger les objets transmis avec **const**

Complex operator+ (const Complex &);

Ex
me

Fonction

```
// Déclaration de la classe
class Complex {
        double a,b;
public:
        Complex(double,double); // constructeur
        void affiche() const;
        Complex operator+=(const Complex  &);  //utilisation z3=z1+z2
};
```

```
// Définition des méthodes de la classe

Complex Complex::Complex(double a, double b):this->a(a),this->b(b)
{
}
void Complex::affiche()
{
cout<<a<<"+i "<<b<<endl;
}
Complex Complex opérator+=(const Complex &z2)
{
this->a+=z2.a;
this->b+=z2.b;
return *this;
}
```

```
// Utilisation de la classe

int main()
{
Complex z1(2.2,3.3),z2(4.4,5.5),z3;
z1.affiche();
z2.affiche();
z3=z1+z2; // le + sera interprété comme z3=z1.operator+(z2)
z3.affiche();
}
```


Surcharge des opérateurs externes

Une deuxième possibilité nous est offerte par le langage pour surcharger les opérateurs.

- La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci.

- Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres : il n'y aura pas de paramètre implicite.

La syntaxe est la suivante :

type operatorOp(opérandes)

où opérandes est la liste complète des opérandes.

- Les opérateurs externes doivent être déclarés comme étant des **fonctions amies** (**friend) de la classe sur laquelle ils travaillent, faute** de quoi ils ne pourraient pas manipuler les attributs de leurs opérandes.
- Les opérateurs de lecture ou d'écriture sont définis comme opérateurs externes.

EC

```
// Déclaration de la classe
class Complex {
        double a,b;
public:
        Complex(double,double); // constructeur
        void affiche() const;
        friend Complex operator+(const Complex  &, const Complex &);
};
```

```
// Définition des méthodes de la classe

Complex Complex::Complex(double a, double b):this->a(a),this->b(b)
{
}
void Complex::affiche()
{
cout<<a<<"+i "<<b<<endl;
}
Complex opérator+(const Complex &z1,const Complex &z2)
{
Complex res(z1.a+z2.a,z1.b+z2.b);
return res;
}
```

```
// Utilisation de la classe

int main()
{
Complex z1(2.2,3.3),z2(4.4,5.5),z3;
z1.affiche();
z2.affiche();
z3=z1+z2; // le + sera interprété comme z3=operator+(z1,z2)
z3.affiche();
}
```