

Séance 3

Programmation C++

Classes

UE Physique numérique. M1 Physique.

M. Ismail, PHITEM, Université Grenoble Alpes

3.1

Table des matières

1	Les classes	1
2	Constructeurs	4
2.1	Définition et exemple	4
2.2	Exemple de conception	5

3.2

1 Les classes

Les classes versus structures

🔗 On remplace le mot-clé `struct` par `class` dans l'exemple précédent :

```
struct Complexe
{
    // declarations des donnees membres
    double Re;
    double Im;
    //declaration des fonctions membres
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
};
```

|structFuncMbre.C2.cpp|

🔗 La déclaration de notre classe :

```
class Complexe
{
    // declarations des donnees membres
    double Re;
    double Im;
    //declaration des fonctions membres
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
};
```

|classVstruct.C3.cpp|

🔗 On essaye de tester la classe avec ce programme simple :

```
int main ()
{
    Complexe z1;
```

```

z1.Re = 4.;
z1.affiche();
Complexe z2 = z1.conjugué();
Complexe z3 = {1.,2.};
z3 = z1;
return 0;
}

```

lclassVstruct.C3.cpp

✏️ Compilation :

```

classVstruct.C3.cpp: In member function
'Complexe Complexe::addition(const Complexe&)':
classVstruct.C3.cpp:30: error: braces around initializer for
non-aggregate type 'Complexe'
classVstruct.C3.cpp: In function 'int main()':
classVstruct.C3.cpp:7:error: 'double Complexe::Re' is private
classVstruct.C3.cpp:38:error:within this context
classVstruct.C3.cpp:14:error: 'void Complexe::affiche()' is private
classVstruct.C3.cpp:39:error:within this context
classVstruct.C3.cpp:21:error: 'Complexe Complexe::conjugué()' is private
classVstruct.C3.cpp:40:error:within this context
classVstruct.C3.cpp:41:error:braces around initializer for
non-aggregate type 'Complexe'

```

✖ Les numéros des lignes incriminées : 30, 38, 39, 40 et 41

```

28 Complexe Complexe::addition(const Complexe & z)
29 {
30     Complexe result = {Re+z.Re, Im+z.Im};
31     return result;
32 }

35 int main ()
36 {
37     Complexe z1;
38     z1.Re = 4.;
39     z1.affiche();
40     Complexe z2 = z1.conjugué();
41     Complexe z3 = {1.,2.};
42     z3 = z1;
43     return 0;
44 }

```

lclassVstruct.C3.cpp

— Rappel de la déclaration de la classe

```

4 class Complexe
5 {
6     // declarations des donnees membres
7     double Re;
8     double Im;
9     //declaration des fonctions membres
10    void affiche();
11    Complexe conjugué();
12    Complexe addition(const Complexe & z);
13 };

```

lclassVstruct.C3.cpp

3.3

Contrôle d'accès par mots-clés

Modèle

```

class A
{
    // membres (données ou fonctions) par défaut privés

```

```

public :
// membres (données ou fonctions) publics

private :
// membres (données ou fonctions) privés

};

```

Modification de la déclaration de la classe `Complexe`

- On ajoute par exemple le mot-clé `public` avant les déclarations des données et des fonctions membres

```

class Complexe
{
public :
//membres publics
double Re;
double Im;
//fonctions membres publiques
void affiche();
Complexe conjugue();
Complexe addition(const Complexe & z);
};

```

`\classPublic.C3.cpp`

- Le programme fonctionne correctement mais nous aimerions utiliser davantage les fonctionnalités du langage quant au contrôle d'accès aux données

Classe `Complexe` en représentation interne

Conditions :

- Les données membres `Re` et `Im` doivent être privées
- Les fonctions membres peuvent rester publiques

- Reprenons le programme précédent et y apportons les modifications nécessaires

```

class Complexe
{
private: // Donnees membres privees
double Re;
double Im;
public: //fonctions membres publiques
void affiche();
Complexe conjugue();
Complexe addition(const Complexe & z);
};

```

`\classeComplexeReplnt.C3.cpp`

Il faut aussi corriger les instructions devenues illégales :

- On ne peut plus utiliser `z1.Re = 4.;` et `Complexe z3 = {1.,2.};` dans la fonction `int main()`

- `z1.Re` et `z1.Im` sont inaccessibles depuis l'extérieur de la classe

- Il faut aussi modifier la fonction

```

Complexe Complexe::addition(const Complexe & z)
{
    Complexe result = {Re+z.Re, Im+z.Im};
    return result;
}

```

- Par exemple :

```

Complexe Complexe::addition(const Complexe & z)
{
    Complexe result;
    result.Re = this->Re + z.Re;
    result.Im = this->Im + z.Im;
    return result;
}

```

Exemple d'utilisation de la classe :

```
int main ()
{
    Complexe z1;
    z1.affiche();
    Complexe z2 = z1.conjugué(); //  $z_2 = \overline{z_1}$ 
    z2.affiche();
    Complexe z3;
    z3 = z2;
    z3.affiche();
    Complexe z4;
    z4 = z1.addition(z3.conjugué()); //  $z_4 = z_1 + \overline{z_3} = z_1 + \overline{z_2}$ 
    z4.affiche();
}
```

|classeComplexeReplnt.C3.cpp|

⛔ Tous ces objets n'ont pas pu être initialisés correctement. En **représentation interne**, on ne peut pas accéder directement aux membres `Re` et `Im` depuis l'extérieur de la classe.

🔧 **Solution** : prévoir une fonction (**public**) d'initialisation

```
public: //fonctions membres publiques
    void affiche();
    Complexe conjugué();
    Complexe addition(const Complexe & z);
    void initialise (double x=0, double y=0) {Re = x; Im = y;}
};
```

|classeComplexeReplnt.C3.cpp|

Exemple d'utilisation à partir de la fonction `main` :

```
z1.initialise(1.,2.); //  $z_1 = 1 + 2i$ 
z1.affiche();
z2 = z1.conjugué(); //  $z_2 = \overline{z_1} = 1 - 2i$ 
z2.affiche();
z4 = z1.addition(z2.conjugué()); //  $z_4 = z_1 + \overline{z_2} = z_1 + \overline{z_2}$ 
z4.affiche();
```

|classeComplexeReplnt.C3.cpp|

3.6

2 Constructeurs

2.1 Définition et exemple

Introduction

- ⛔ Rappel : par défaut les objets de type **class** désignent des variables automatiques ➡ Absence d'initialisation par défaut de ces objets lors de la déclaration : `Complex z1`
- Que faire ? une fonction d'initialisation ! pas très élégant et source d'erreur
- Meilleure approche ? permettre au programmeur de déclarer une fonction spéciale ➡ **constructeur**

3.7

Définition


Définition

Un constructeur est une fonction membre particulière (méthode). Il

- possède le même nom que la classe,
- ne possède pas de valeur de retour (même pas **void**).

3.8


Un constructeur simple

 Déclaration et définition :

```
class A
{
    //objet qui ne contient aucune donnée membre
public :
    A(); //déclaration d'un constructeur
};

A::A() //définition du constructeur
{
    cout << "Je suis un constructeur du type A" << endl;
    cout << "Je viens de creer un objet A d'adresse " << this << endl;
}
```


lconstructSimple.C3.cpp

 Utilisation

```
int main()
{
    void f();
    A a;
    A b;
    cout << endl;
    f();
    cout << endl;
    return 0;
}

void f()
{
    cout <<"Entree dans la fonction f"<< endl;
    A temp;
    cout <<"Sortie de la fonction f"<< endl;
}
```

lconstructSimple.C3.cpp




 Exécution :

```
Je suis un constructeur du type A
Je viens de creer un objet A d adresse 0x7fff9ad6a2bf
Je suis un constructeur du type A
Je viens de creer un objet A d adresse 0x7fff9ad6a2be

Entree dans la fonction f
Je suis un constructeur du type A
Je viens de creer un objet A d adresse 0x7fff9ad6a29f
Sortie de la fonction f
```

3.9


Encapsulation

- Regrouper les données et les fonctions au sein d'une classe
- Association à un système de protection par mots clés :
 -  **public** : niveau le plus bas de protection, toutes les données ou fonctions membres d'une classe sont utilisables par toutes les fonctions
 -  **private** : niveau le plus élevé de protection, données (ou fonctions membres) d'une classe utilisables uniquement par les fonctions membre de la même classe
 -  **protected** : comme **private** avec extension aux classes dérivées (voir héritage)

3.10

2.2 Exemple de conception

Classe Cmpx. Cahier des charges

 Réécriture d'une nouvelle classe modélisant les nombres complexes répondant à ces exigences :

- ☞ une représentation interne (données privées représentant le complexe) sous forme polaire : **Module** ≥ 0 et **Phase** $\in [-\pi, \pi]$
- ☞ initialisation sous la forme polaire ou cartésienne
- ☞ une initialisation par défaut
- ☞ accès au module et à la phase
- ☞ affichage sous la forme polaire et la forme cartésienne
- ☞ fonctions émulant *, / et l'exponentiation

Première approche

☞ Déclaration de la classe

```
class Cmpx
{
private :
    double Module;
    double Phase;
public :
    // Constructeur
    Cmpx(double rho = 0, double theta = 0);
    // Fonctions membres
    void affiche();
    Cmpx Multiplication(const Cmpx &);
    Cmpx Division(const Cmpx &);
    Cmpx Exponentiation(const double & exposant);
};
```

lcmpx.C3.cppl

☞ Définition du constructeur

```
Cmpx::Cmpx(double rho, double theta)
{
    Module = rho;
    Phase = theta;
}
```

lcmpx.C3.cppl

☞ La fonction exponentiation

```
Cmpx Cmpx::Exponentiation(const double & exposant)
{
    return Cmpx(pow(Module, exposant), exposant * Phase);
}
```

lcmpx.C3.cppl

☞ La fonction division

```
Cmpx Cmpx::Division(const Cmpx & z)
{
    if(!z.Module )
    {
        cout << "Division par le complexe 0" << endl;
        exit(1);
    }
    double x = Module / z.Module;
    double y = Phase - z.Phase;
    Cmpx result(x, y);
    return result;
}
```

lcmpx.C3.cppl

☞ La fonction multiplication

```
Cmpx Cmpx::Multiplication(const Cmpx & z)
{
    return Cmpx(Module * z.Module, Phase + z.Phase);
}
```

La fonction affiche

```
void Cmpx::affiche()
{
    cout << "(" << Module << ", " << Phase << ")" << endl;
}
```

lcmpx.C3.cpp1

Utilisation

```
int main()
{
    const double Pi = atan(1.)*4.;

    Cmpx z1(2,Pi/4.); //  $z_1 = \sqrt{2} + i\sqrt{2}$ 
    cout << "z1 = " ; z1.affiche();

    Cmpx z;
    z = 2.; //  $z = 2e^{0i}$ ?,  $z = 2e^{2i}$ ?,  $z = 0e^{2i}$ ?
    cout << "z = " ; z.affiche();

    Cmpx z3 = z1.Multiplication(z); //  $z_3 = z_1 z$ 
    cout << "Resultat de la multiplication :";
    cout << "z3 = " ; z3.affiche();

    z3 = z1.Multiplication(2); // Conversion de 2 en Cmpx?
    cout << "Le meme produit avec un appel different :";
    cout << "z3 = " ; z3.affiche();

    Cmpx z4(2);
    cout << "z4 = " ; z4.affiche();
    (z4.Exponentiation(0.5)).affiche();

    Cmpx z5(-2);
    cout << "z5 = " ; z5.affiche();
    (z5.Exponentiation(0.5)).affiche();

    Cmpx z6;
    cout << "z6 = " ; z6.affiche();
    (z6.Exponentiation(-0.5)).affiche();

    return 0;
}
```

lcmpx.C3.cpp1

Exécution :

```
z1 = (2, 0.785398)
z = (2, 0)
Resultat de la multiplication :z3 = (4, 0.785398)
Le meme produit avec un appel different :z3 = (4, 0.785398)
z4 = (2, 0)
(1.41421, 0)
z5 = (-2, 0)
(nan, 0)
z6 = (0, 0)
(inf, -0)
```

- Interdire la conversion implicite (Cmpx z=2)
- Pourquoi nan et inf ? Enrichir les fonctions membres pour vérifier la validité des données

Mot clé explicit

- ⛔ Le mot-clé **explicit** interdit la conversion implicite. Le constructeur Cmpx ne sera appelé qu'explicitement

✍ On modifie la déclaration du constructeur :

```
class Cmpx
{
private :
    double Module;
    double Phase;
public :
    // Constructeur
    explicit Cmpx(double rho = 0, double theta = 0);
    // Fonctions membres
    void affiche();
    Cmpx Multiplication(const Cmpx &);
    Cmpx Division(const Cmpx &);
    Cmpx Exponentiation(const double & exposant);
};
```

lcmpxExplicit.C3.cpp

✗ Les instructions des lignes suivantes deviennent illégales

```
63  z = 2.; // z = 2e0i?, z = 2e2i?, z = 0e2i?
70  z3 = z1.Multiplication(2); // Conversion de 2 en Cmpx?
```

lcmpx.C3.cpp

✓ Par contre, on peut toujours demander **explicitement** au constructeur d'effectuer une telle conversion

```
z = Cmpx(2.); // z = 2 (conv. explicite, Phase = 0 par default)
z3 = z1.Multiplication(Cmpx(2)); // Conversion explicite de 2
```

lcmpxExplicit.C3.cpp

3.13

Constructeur et validation des données

Réécriture du constructeur avec :

✍ une liste d'initialisation

✍ Validation des données

```
Cmpx::Cmpx(double rho, double theta)
:Module(rho), Phase(theta)
{
    if(Module < 0)
    {
        cout << "Constructeur invoque avec un module < 0" << endl;
        exit(1);
    }
}
```

lcmpxExplicit.C3.cpp

👤 Exécution :

```
z1 = (2, 0.785398)
z = (2, 0)
Resultat de la multiplication : z3 = (4, 0.785398)
Le meme produit avec un appel different : z3 = (4, 0.785398)
z4 = (2, 0)
(1.41421, 0)
Constructeur invoque avec un module < 0
```

Rappel des résultats de l'ancienne version :

```
...
z4 = (2, 0)
(1.41421, 0)
z5 = (-2, 0)
(nan, 0)
z6 = (0, 0)
(-inf, -0)
```

3.14

Autres améliorations pour respecter le cahier des charges

La fonction exponentiation

```
Cmplx Cmplx::Exponentiation(const double & exposant)
{
    if((exposant < 0) && (Module == 0) )
    {
        cout << "Elevation de 0 a une puissance negative " << endl;
        exit(1);
    }
    return Cmplx(pow(Module, exposant), exposant * Phase);
}
```

|cmplxExplicit.C3.cpp|

ou plus “rigoureusement” :

```
. . .
static const double epsilon_double = pow(10., -16);
if((exposant < 0) && (abs(Module) <= epsilon_double) )
. . .
```