Programmation Orientée Objet : Surcharge des opérateurs

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Objectifs de la leçon d'aujourd'hui

- Concepts fondamentaux
- Étude de cas

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam
EPFL

Organisation du travail (semestre)

		MOOC	déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
1	23.02.23		0	Intro + compil. séparée	
2	02.03.23	1. Intro POO	0	Intro POO	
3	09.03.23	2. Constructeurs/De	0	Constructeurs	
4	16.03.23	3. Surcharge des op	0	Surcharge	
5	23.03.23	4. Héritage	0	Héritage	
6	30.03.23	5. Polymorphisme	0	Polymorphisme 1	
7	06.04.23		1	Polymorphisme 2 / Collections hétérogènes	
-	13.04.23		-	vacances Pâques	
8	20.04.23		-	-	Série notée
9	27.04.23	6. Héritage multiple	2	Héritage multiple	
10	04.05.23	(7. Etude de cas)	-	Templates	
11	11.05.23		-	Structure de données abstraites ; Bibliothèques	
12	18.05.23		-	(Ascension)	
13	25.05.23	(7. Etude de cas)	-	Bibliothèques (fin) + Révisions	
14	01.06.23		-	-	Examen

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Concepts fondamentaux

- à quoi sert la surchage des opérateurs :
  - pourquoi l'utiliseriez-*VOUS* ?
  - à quel niveau voulez-*VOUS* le faire ?
- surchage interne / surcharge externe
- (Attention aux copies !)
  (moins grave en 

, le compilateur peut vous aider)

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Etude de cas

Comment afficher nos nombres complexes ?

Et finalement les nombres complexes, n'est-ce pas (surtout) pour faire des calculs ?.....

Décortiquons entièrement, pas à pas, la ligne suivante :

```
cout << 5.5 * ( Complexe(1.1, 2.2) * Complexe(3.3, 4.4) )
     << endl;
```

Et, si on a le temps, aussi celles-ci :

```
Complexe z1(1.1, 2.2);
Complexe z2(3.3, 4.4);
Complexe z3( z1 *= z2 );
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Décodage

Que signifie

```
cout << 5.5 * ( Complexe(1.1, 2.2) * Complexe(3.3, 4.4) )
     << endl;
```

Essayons de le réécrire en lignes d'une seule expression :

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Décodage

Que signifie

```
cout << 5.5 * ( Complexe(1.1, 2.2) * Complexe(3.3, 4.4) )
     << endl;
```

Essayons de le réécrire en lignes d'une seule expression :

```
Complexe z1(1.1, 2.2);
Complexe z2(3.3, 4.4);
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Décodage

Que signifie

```
cout << 5.5 * ( Complexe(1.1, 2.2) * Complexe(3.3, 4.4) )
     << endl;
```

Essayons de le réécrire en lignes d'une seule expression :

```
Complexe z1(1.1, 2.2);
Complexe z2(3.3, 4.4);
Complexe z3( z1 * z2 );
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Décodage

Que signifie

```
cout << 5.5 * ( Complexe(1.1, 2.2) * Complexe(3.3, 4.4) )
     << endl;
```

Essayons de le réécrire en lignes d'une seule expression :

```
Complexe z1(1.1, 2.2);
Complexe z2(3.3, 4.4);
Complexe z3( z1 * z2 );
Complexe z4( 5.5 * z3 );
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Décodage

Que signifie

```
cout << 5.5 * ( Complexe(1.1, 2.2) * Complexe(3.3, 4.4) )
     << endl;
```

Essayons de le réécrire en lignes d'une seule expression :

```
Complexe z1(1.1, 2.2);
Complexe z2(3.3, 4.4);
Complexe z3( z1 * z2 );
Complexe z4( 5.5 * z3 );
cout << z4;
cout << endl;
// quid de cout << z4 << endl; ??
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Opérateur d'affichage

```
cout << z4;
```

☞ `cout.operator<<(z4);` ou `operator<<(cout, z4);` ?

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Opérateur d'affichage

```
cout << z4;
```

☞ `cout.operator<<(z4);` ou `operator<<(cout, z4);` ?

☞ `void operator<<(ostream&, Complexe const&);`

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Opérateur d'affichage

```
cout << z4;
```

☞ `cout.operator<<(z4);` ou `operator<<(cout, z4);` ?

☞ `void operator<<(ostream&, Complexe const&);`

```
cout << z4 << endl;
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Opérateur d'affichage

```
cout << z4;
```

☞ `cout.operator<<(z4);` ou `operator<<(cout, z4);` ?

☞ `void operator<<(ostream&, Complexe const&);`

```
cout << z4 << endl;
```

☞ `operator<<(operator<<(cout, z4), endl);`

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Opérateur d'affichage

```
cout << z4;
```

☞ `cout.operator<<(z4);` ou `operator<<(cout, z4);` ?

☞ `void operator<<(ostream&, Complexe const&);`

```
cout << z4 << endl;
```

☞ `operator<<(operator<<(cout, z4), endl);`

☞ `ostream& operator<<(ostream&, Complexe const&);`

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Multiplication entre complexes

```
z1 * z2
```

☞ surcharge interne ou externe ?

Deux principes (parfois contradictoires) :

1. préférez la surcharge externe si un nouvel objet est créé ;
   sinon la surchage interne ;
2. utilisez la surcharge interne si vous *devez* accéder aux parties privées ;

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Multiplication entre complexes

ICI :

1. est-ce qu'un nouveau complexe est créé ?

   ☞ oui, `(z1*z2)` est un nouveau complexe

   Donc de ce point de vue : clairement une **surcharge externe**

2. est-ce qu'on peut faire la multiplication sans d'accéder aux parties privées ?

   ☞ cela dépend en général de l'encapsulation, mais dans ce cas précis pas trop car ici la bonne façon d'écrire cette multiplication serait justement de l'adapter à la représentation interne : utilisez la formule avec des coordonnées cartésiennes si le nombre complexe est représenté en cartésiennes et utiliser des coordonnées polaires si le nombre complexe est représenté en polaires.

   Donc de ce point de vue : plutôt une **surcharge interne**

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Multiplication entre complexes

On peut changer la seconde conclusion précédente en offrant dans la partie *publique* une méthode que la multiplication externe pourrait utiliser : l'opérateur `*=`

La *bonne* façon de faire consiste donc à :

1. définir en interne `operator*=`
2. définir en externe `operator*` qui utilise `operator*=`

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Multiplication entre complexes

On peut changer la seconde conclusion précédente en offrant dans la partie *publique* une méthode que la multiplication externe pourrait utiliser : l'opérateur `*=`

La *bonne* façon de faire consiste donc à :

1. définir en interne `operator*=`
2. définir en externe `operator*` qui utilise `operator*=`

```
Complexe& Complexe::operator*=(Complexe const& autre)
{
  double x_old(x_);
  x_ =   x_  * autre.x_ - y_ * autre.y_;
  y_ = x_old * autre.y_ + y_ * autre.x_;
  return *this;
}

Complexe operator*(Complexe a, Complexe const& b)
{
  a *= b;
  return a;
}
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Multiplication par un `double` ?

Est-ce que l'on peut (déjà) écrire : `z4 = 5.5 * z3;` ?

☞ Attention ! Il y a une subtilité !

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Multiplication par un `double` ?

Est-ce que l'on peut (déjà) écrire : `z4 = 5.5 * z3;` ?

☞ Attention ! Il y a une subtilité !

En fait, avec ce que nous avons défini jusqu'ici (y compris la semaine passée), le code ci-dessus est interprété comme :

```
z4 = Complexe(5.5) * z3;
```

car nous avons un contructeur pour les complexes qui prend un seul `double` : on a un

```
Complexe::Complexe(double);
```

par exemple via

```
Complexe::Complexe(double = 0.0, double = 0.0);
```

Si l'on veut éviter ce genre de conversion *implicite*, il faut marquer le constructeur comme `explicit` :

```
explicit Complexe(double abscisse = 0.0, double ordonnee = 0.0)
: x_(abscisse),  y_(ordonnee)
{}
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam

Multiplication par un `double` ?

CECI DIT : ici cette conversion implicite n'est pas dérangeante puisque justement les opérations sur les complexes de partie imaginaire nulle sont exatement les mêmes que celle avec des nombres réels.

Donc ici **on peut en rester là** (*sans* `explicit`).

Mais ce n'est pas toujours le cas : par exemple pour des vecteurs, le produit scalaire entre un vecteur ayant deux coordonnées nulles et un autre vecteur n'est pas la même chose que la multiplication de cet autre vecteur par un scalaire (le résultat n'est même pas de même nature ! un scalaire dans le premier cas et un vecteur dans le second !) :

$$(x_1, 0, 0) \cdot (x_2, y_2, z_2) \neq x_1 (x_2, y_2, z_2)$$

Dans ce cas là, il faudrait certainement mettre le constructeur comme `explicit`...
...et/ou avoir par exemple :

```
double Vecteur::operator*(Vecteur const&) const;
Vecteur& Vecteur::operator*=(double);
Vecteur operator*(double, Vecteur);
```

©EPFL 2023
Jean-Cédric Chappelier
& Jamila Sam