

# Programmation Orientée Objet : Surcharge des opérateurs

Jean-Cédric Chappelier

Faculté I&C

# Organisation du travail (semestre)

| MOOC |          | déc.                 | cours<br>1 h<br>Jeudi 8-9 | exercices<br>2 h<br>Jeudi 9-11                  |
|------|----------|----------------------|---------------------------|-------------------------------------------------|
| 1    | 19.02.26 |                      | 0                         | Intro + compil. séparée                         |
| 2    | 26.02.26 | 1. Intro POO         | 0                         | Intro POO                                       |
| 3    | 05.03.26 | 2. Constructeurs/Des | 0                         | Constructeurs                                   |
| 4    | 12.03.26 | 3. Surcharge des opé | 0                         | Surcharge                                       |
| 5    | 19.03.26 | 4. Héritage          | 0                         | Héritage                                        |
| 6    | 26.03.26 | 5. Polymorphisme     | 0                         | Polymorphisme 1                                 |
| 7    | 02.04.26 |                      | 1                         | Polymorphisme 2 / Collections hétérogènes       |
| -    | 09.04.26 |                      | -                         | vacances Pâques                                 |
| 8    | 16.04.26 | 6. Héritage multiple | 1                         | Héritage multiple                               |
| 9    | 23.04.26 |                      | -                         | - Midtem                                        |
| 10   | 30.04.26 | (7. Etude de cas)    | -                         | Templates                                       |
| 11   | 07.05.26 |                      | -                         | Structure de données abstraites ; Bibliothèques |
| 12   | 14.05.26 |                      | -                         | (Ascension)                                     |
| 13   | 21.05.26 | (7. Etude de cas)    | -                         | Bibliothèques (fin) + Révisions                 |
| 14   | 28.05.26 |                      | -                         | - Examen                                        |

# Objectifs de la leçon d'aujourd'hui

- ▶ Concepts fondamentaux
- ▶ Étude de cas

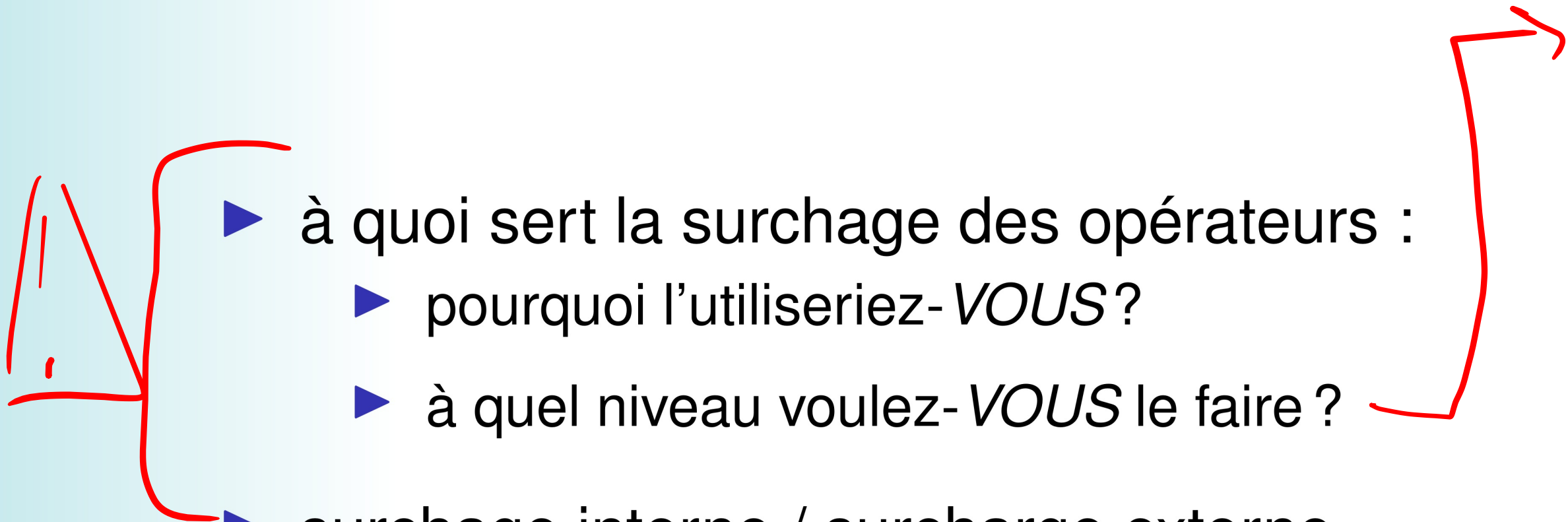
# Concepts fondamentaux

0 : pas du tout

1 : << + \* / -

2 : N1+ ↑ += \*= /= ...  
return: void

3 : N1+ ↑ += \*= /= ...  
return: Type&

- 
- ▶ à quoi sert la surcharge des opérateurs :
    - ▶ pourquoi l'utiliseriez-VOUS?
    - ▶ à quel niveau voulez-VOUS le faire?

- ▶ surcharge interne / surcharge externe

- ▶ (Attention aux copies !)  
(moins grave en **C++11**, le compilateur peut vous aider)



# 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 :

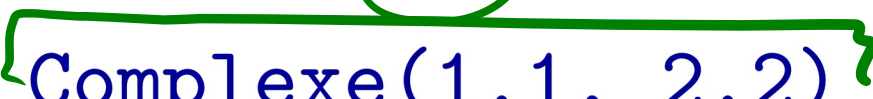
```
cout << 5.5 * ( Complexe(1.1, 2.2) * Complexe(3.3, 4.4) )  
      << endl;
```

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 );
```

# 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 :

# 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);
```

# 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 );
```



# 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 );
```

# 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 :

①

```
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; ??
```

②

# Opérateur d'affichage

 $A \perp B$  $\text{operator} \perp (A, B) \quad \text{ou} \quad A.\text{operator} \perp (B)$  $\overset{A}{\text{cout}} \perp \overset{B}{z4};$ 

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

interne  
à ostream





# Opérateur d'affichage

C

`cout << 3 ;`  
`operator<<(cout, 3);`  
*ostream, int*

`cout << z4;`

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

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

`cout << z4 << endl;`  
A B

`operator<<(cout << z4, endl);`  
de type `ostream`



# 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);`

# Opérateur d'affichage

```
cout << z4;
```

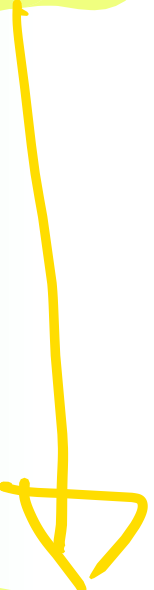
👉 `cout.operator<<(z4);`    ou    `operator<<(cout, z4);` ?

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

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

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

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



# 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 surcharge interne ;

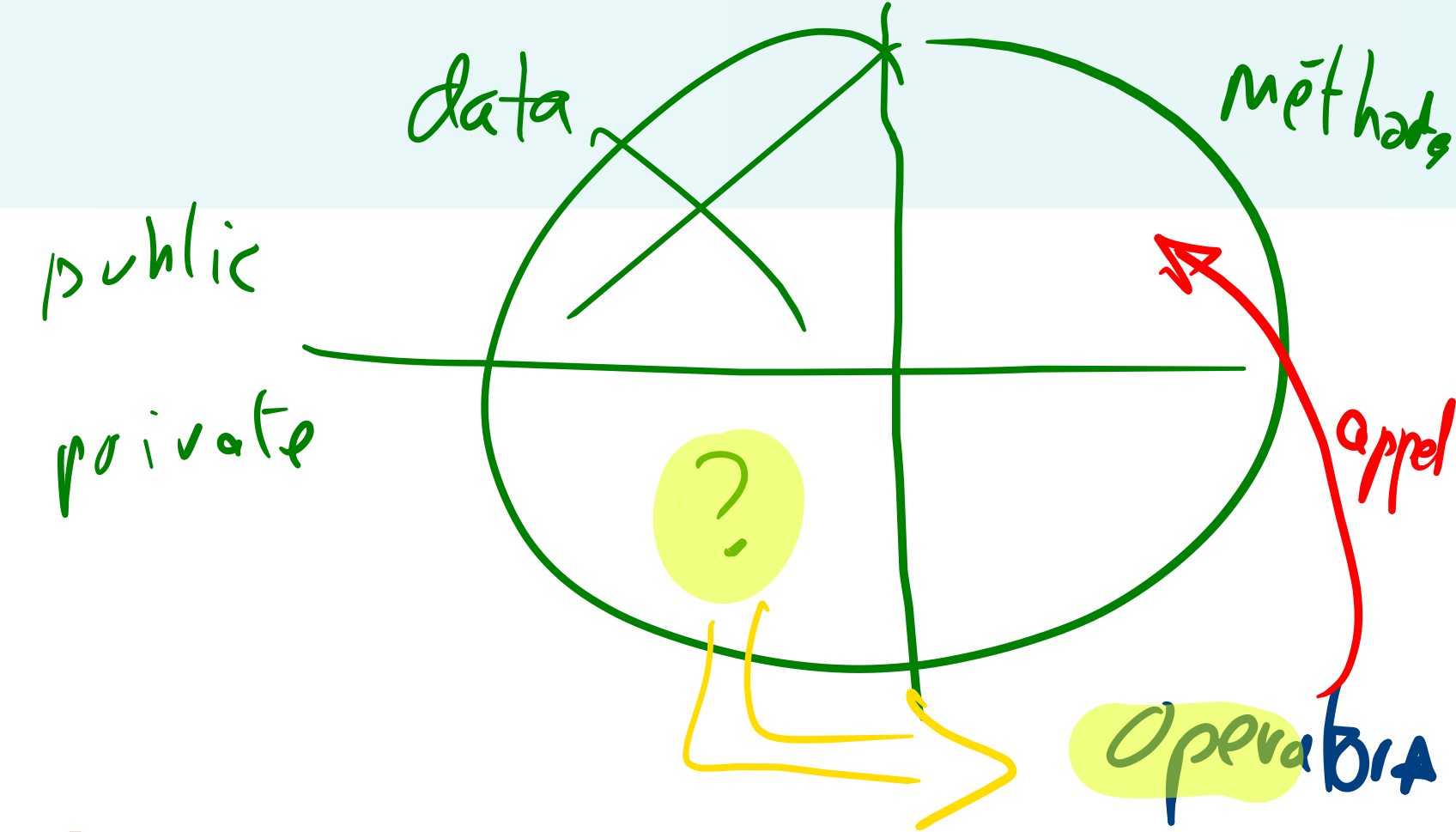
1 fonction

2. utilisez la surcharge interne si vous **devez** accéder aux parties **privées** ;

1 opérateur  
interne



# 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 : **non**, car ici la bonne façon d'écrire cette multiplication serait justement de l'*adapter* à la représentation interne : utiliser 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**

# (Apparté mathématique, version polaire)

Lequel préférez vous ? (pour des `Complex` implémentés en polaires)

$$\rho' = \rho_1 \rho_2 \quad \theta' = \theta_1 + \theta_2$$

) *operator\** ✱

OU (si `operator*` avait été écrit avec les « getters » en cartésiennes)

$$\rho' = \sqrt{(\rho_1 \cos(\theta_1) \rho_2 \cos(\theta_2) - \rho_1 \sin(\theta_1) \rho_2 \sin(\theta_2))^2 + (\rho_1 \cos(\theta_1) \rho_2 \sin(\theta_2) + \rho_1 \sin(\theta_1) \rho_2 \cos(\theta_2))^2}$$

$$\theta' = \arctan \left( \frac{\rho_1 \cos(\theta_1) \rho_2 \sin(\theta_2) + \rho_1 \sin(\theta_1) \rho_2 \cos(\theta_2)}{\rho_1 \cos(\theta_1) \rho_2 \cos(\theta_2) - \rho_1 \sin(\theta_1) \rho_2 \sin(\theta_2)} \right)$$

(**Note** : et encore, la formule de  $\theta'$  est en fait plus compliquée et dépend des signes (et nullité) de  $x_1$ ,  $x_2$ ,  $y_1$  et  $y_2$ . Voir le cours de la première semaine pour la formule exacte.)

---

Le premiers choix ci-dessus est **ré-écrit** par le/la responsable de la classe s'il/elle change la représentation interne.

Le second ne nécessite certes aucune réécriture de `operator*` (puisqu'utilise les « getters »), mais est inefficace et plus sensible aux erreurs d'arrondis (plus de calculs).



# (Apparté mathématique, version cartésienne)

Lequel préférez vous ? (pour des `Complex` implémentés en cartésiennes)

$$x' = x_1 x_2 - y_1 y_2 \quad y' = x_1 y_2 + y_1 x_2$$

OU (si `operator*` avait été écrit avec les « getters » en polaires)

$$x' = \sqrt{(x_1^2 + y_1^2) (x_2^2 + y_2^2)} \cos \left( \arctan\left(\frac{y_1}{x_1}\right) + \arctan\left(\frac{y_2}{x_2}\right) \right)$$

$$y' = \sqrt{(x_1^2 + y_1^2) (x_2^2 + y_2^2)} \sin \left( \arctan\left(\frac{y_1}{x_1}\right) + \arctan\left(\frac{y_2}{x_2}\right) \right)$$

(**Note** : et encore, la formule de  $\theta'$  est en fait plus compliquée et dépend des signes (et nullité) de  $x_1$ ,  $x_2$ ,  $y_1$  et  $y_2$ . Voir le cours de la première semaine pour la formule exacte.)

---

Le premiers choix ci-dessus est **ré-écrit** par le/la responsable de la classe s'il/elle change la représentation interne.

Le second ne nécessite certes aucune réécriture de `operator*` (puisqu'utilise les « getters »), mais est inefficace et plus sensible aux erreurs d'arrondis (plus de calculs).

# Multiplication entre complexes

On peut changer la seconde conclusion précédente (« `operator*` doit accéder aux parties privées ») 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*=`

# Multiplication entre complexes

On peut changer la seconde conclusion précédente (« `operator*` doit accéder aux parties privées ») 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*=`

niveau 2

niveau 3

```

Complexe& Complexe::operator*=(Complexe const& autre)
{
    const 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;
}
  
```

niveau 2 :

$z^1 *= z^2;$

niveau 3

$z^4 = (z^1 *= z^2);$



# Multiplication par un double ?

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

👉 Attention ! Il y a une subtilité !



double Complote



# Multiplication par un double ?

$$z_1 = 3 + 4i$$

$$z_2 = 5$$

$$z_2 \cdot z_1 = 5 \cdot z_1$$

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 constructeur 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)
{ }
```

$$\vec{v}_2$$

$$\vec{v}_1 \in E$$

$$\bar{I} \text{ e.v. sur } K$$

$$\lambda \in K$$

$$K \text{ corps}$$

$$\lambda \vec{v}_1$$

$$\vec{v}_1 \cdot \vec{v}_2$$

$$K \times E \rightarrow E$$

$$E \times E \rightarrow K$$

$$(\lambda, 0, 0) * (v_1^1, v_1^2, v_1^3)$$

$$\lambda * (v_1^1, v_1^2, v_1^3)$$

# Multiplication par un double ?

CECI DIT : ici cette conversion implicite n'est pas dérangementante puisque justement les opérations sur les complexes de partie imaginaire nulle sont exactement 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);
```