

Support de cours

Cours:

## Introduction à la programmation orientée objet (en C++)

Vidéo:

### W13-05-surchopcomplements-CPP-pt2

Concepts (extraits des sous-titres générés automatiquement) :

**Opérateur d'affichage. Valeur de retour de l'appel. Nouvelle copie d'un complexe. Copie supplémentaire. Opérateurs d'auto affectation. Valeur de retour. Opérateur d'affichage de opérateur. Chose de type. Instance courante. Surcharge interne. Type complexe. Copies supplémentaires. Classe de l'opérateur. Versions de l'opérateur d'auto. Premier argument.**



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en C++).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

# Surcharge d'opérateurs : compléments

## (Partie 2)

Introduction à la programmation orientée objet (en C++)

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



## Pourquoi `operator<<` retourne-t-il un `ostream&` ?

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
cout << z1 << endl;
```

Si l'on regarde ceci, on voit que cette expression, appelons-la « A »,

notes

résumé

0m 1s



## Pourquoi `operator<<` retourne-t-il un `ostream&` ?

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
cout << z1 << endl;
```

A

*A << endl;*  
*operator<<(A,*

cette ligne est donc du genre « A << endl ». C'est donc bien ici un appel à « opérateur d'affichage », « operator << (A,endl) »

notes

résumé

0m 5s



## Pourquoi `operator<<` retourne-t-il un `ostream&` ?

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
operator<<(cout << z1, endl);
```

C'est à dire si je réécris « A » pour ce qu'il est,

notes

résumé

0m 23s



## Pourquoi `operator<<` retourne-t-il un `ostream&` ?

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
operator<<(cout << z1, endl);
```

*Cout << z1 << endl*

c'est-à-dire « `cout << z1` ». L'expression « `cout << z1 << endl` »

notes

résumé

0m 29s



## Pourquoi `operator<<` retourne-t-il un `ostream&` ?

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
operator<<(operator<<(cout, z1), endl);
```

est équivalente à « `operator <<` » ; premier argument : le résultat donc de « `cout << z1` » et deuxième argument : le « `endl` ». Mais ceci, « `cout << z1` »,

notes

résumé

0m 39s



```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
operator<<(operator<<(cout, z1), endl);
```

*cout << z1 << endl;*

*void operator<<*

est lui-même l'appel de « opérateur d'affichage de « cout » et « z1 » ». Donc « cout << z1 << endl » est équivalent en fait à l'appel « opérateur d'affichage de opérateur d'affichage sur « cout » et « z1 » et ensuite « endl » » ; où l'on voit que la valeur de retour de l'appel « operator << (cout, z1) » est passé comme premier argument d'un « opérateur inférieur inférieur ». Et donc cette expression doit être ici de même type que le premier argument de « l'opérateur inférieur inférieur ». Toute cette expression doit être de type « ostream& ». Voilà pourquoi la valeur de retour de « operator affichage » doit être « ostream& », pour pouvoir comme ça, le remettre dans un deuxième appel, voir un troisième appel. Si vous aviez écrit « void » comme type de retour, il ne faut jamais faire ça

notes

résumé

0m 53s





```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
operator<<(operator<<(cout, z1), endl);
```

*void cout << z1 << endl;*

*void operator<< (...);*

parce que si vous aviez écrit « void » comme type de retour, vous ne pourriez pas écrire des expressions du genre « cout << z1 << endl ». Ce second « << » ici ne pourrait pas avoir lieu, puisque vous auriez une première expression ici qui retournerait un « void » et donc vous auriez quelque chose de type « void » suivi de « << endl », ce qui n'a absolument aucun sens.

notes

résumé

1m 48s



## Pourquoi `operator<<` retourne-t-il un `ostream&` ?

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
operator<<(operator<<(cout, z1), endl);
```

*void cout << z1 << endl;*

*void operator<< (...);*



C'est donc bien pour pouvoir enchaîner ainsi les appels à l'opérateur d'affichage

notes

résumé

2m 13s



## Quel type de retour pour `operator+=` ?

```
z1 += z2;
```

```
void Complexe::operator+=(Complexe const&);
```



que cet opérateur a besoin de retourner un type qui est compatible avec son premier argument. C'est à dire un flot dans lequel on peut écrire un « ostream » passé par référence, un « ostream& ».

### notes

---

---

---

---

---

---

---

---

---

---

### résumé

2m 19s



---

---

---

---

---

---

---

---

## Quel type de retour pour `operator+=` ?

```
z1 += z2;
```

```
z1.operator+=(z2);
```

```
void Complexe::operator+=(Complexe const&);
```

Troisième complément, quel type de retour pour les opérateurs d'auto affectation comme par exemple l'opérateur « += » Si j'écris quelque chose du genre « `z1 += z2` », je vous rappelle que c'est équivalent à l'appel « `z1.operator+=` », surcharge interne, « de `z2` ».

### notes

### résumé

2m 32s



## Quel type de retour pour operator+= ?

```
z1 += z2;
```

```
z = z1.operator+=(z2);
```

```
void Complexe::operator+=(Complexe const&);
```

Et on voit donc ici que cet appel est autonome, il n'y a pas d'expression devant, on n'a pas écrit quelque chose du genre « `z = z1.operator+=(z2)` »

notes

résumé

2m 52s



### Quel type de retour pour `operator+=` ?

```
z1 += z2;
```

~~3~~

~~z1.operator += (z2);~~

```
void Complexe::operator+=(Complexe const&);
```

Et donc justement puisque l'on n'a pas écrit ceci, c'est pour ça qu'au premier niveau de surcharge, on vous conseille comme type de retour, de mettre simplement « void ». Ce qui correspond au premier niveau de surcharge dont nous parlions dans l'introduction.

notes

---

résumé

3m 3s



## Quel type de retour pour `operator+=` ?

```
z1 += z2;
```

```
void Complexe::operator+=(Complexe const&);
```

En C++, chaque expression **fait** quelque chose et **vaut** quelque chose :

```
x = Expression;
```

```
z3 = (z1 += z2);
```

Mais il faut savoir qu'en C++, toute expression fait quelque chose et toute expression vaut quelque chose. Par exemple, si j'ai une expression, je peux tout à fait écrire « `x =` » cette expression. Donc par exemple je peux tout à fait écrire ce que j'ai suggéré tout à l'heure, « `z3 =` » l'expression « `(z1 += z2)` ». On a ici une expression « `(z1 += z2)` » qui fait quelque chose, c'est-à-dire qu'elle ajoute « `z2` » à « `z1` », mais qui en tant que telle aussi vaut quelque chose et peut donc en tant que telle être, par exemple affectée à « `z3` ». Que vaut cette expression ? Eh bien ce qu'il faut savoir, c'est que cette expression « `(z1 += z2)` », vaut le résultat de « `z1` », la valeur de « `z1` » après l'affectation. Donc toute cette expression est bien de type complexe, puisque « `z1` » et « `z2` » sont ici des complexes. C'est pour ça qu'à un niveau avancé, le type de retour de cet opérateur « `+=` », n'est plus « `void` » mais est donc un « `complexe` ». Ceci dit ce complexe en question existe déjà ! C'est « `z1` » après le résultat de cette auto affectation. Donc ce n'est pas un nouveau complexe, c'est un complexe qui existe déjà. Et donc on va pas recréer un nouveau complexe, mais on va simplement passer une référence sur un complexe qui existe déjà. C'est donc pour ces raisons que les types de retour plus avancés,

### notes

### résumé

3m 17s



Attention de ne pas utiliser la surcharge des opérateurs à **mauvais escient** et à veiller à les **écrire avec un soin particulier**.

Les performances du programme peuvent en être gravement affectées par des opérateurs surchargés mal écrits.

En effet, l'utilisation inconsidérée des opérateurs peut conduire à un grand nombre de copies d'objets :

**Utiliser des références dès que cela est approprié !**

pour les opérateurs d'auto affectation, compatibles avec la norme, sont des références sur des instances, donc par exemple référence sur un complexe. Voyons maintenant comment tout ceci se définit. Donc on fait une surcharge en interne dans la classe de l'opérateur « += » qui reçoit donc le deuxième argument, la valeur que l'on veut rajouter à l'instance courante, et qui retourne donc ici, au niveau plus avancé, une référence sur un nombre complexe. On mettra une référence à chaque fois que l'on retourne un complexe qui existe déjà. Ce n'est pas une nouvelle valeur. Dans cette expression ici, on retourne la valeur de « z1 ». Et donc on le définirait de la façon suivante, avec on commence ici par l'entête, et puis ensuite comme on l'avait déjà vu donc, l'ajout de la valeur de « x de z2 » à l'attribut « x » courant, l'ajout de l'attribut « y de z2 » à l'attribut « y » courant. Mais ce que l'on rajoute, ce qui change maintenant, c'est que comme on a ici un type de retour, il faut retourner. Retourner qui ? Retourner l'instance courante. Et qui est l'instance courante ? C'est la valeur qui est stockée à mon adresse. Ici « this » est un pointeur sur l'instance courante et donc \*« this » c'est la valeur stockée à l'adresse de l'instance courante. C'est-à-dire par exemple ici, la valeur de « z1 ». Terminons par des considérations pratiques mais qui ont vraiment leur importance. Nous avons ici défini comme type de retour une référence sur un nombre complexe, parce que ce complexe de retour ici existait déjà. Si on avait pas mis de référence, on aurait alors recréé un nouveau complexe, une nouvelle copie d'un complexe qui existait déjà, ce qui est une copie supplémentaire qu'il n'est pas nécessaire de faire.

### notes

### résumé

4m 52s





Exemple : comparez le code suivant qui fait de 1 à 3 copies inutiles :

```
Complexe Complexe::operator+=(Complexe z2)
{
    Complexe z3;
    x += z2.x;
    y += z2.y;
    z3 = *this;
    return z3;
}
```

avec le code suivant qui n'en fait pas :

```
Complexe& Complexe::operator+=(Complexe const& z2)
{
    x += z2.x;
    y += z2.y;
    return *this;
}
```

Et de façon très générale donc, on peut vous donner les conseils suivants. Il faut écrire la surcharge des opérateurs avec beaucoup d'attention. En faisant extrêmement attention à ne pas faire des copies supplémentaires, à ne pas péjorer le code que vous écrivez par des opérations inutiles. Puisqu'en effet, les opérateurs ont pour but d'être utilisés très souvent, dans beaucoup d'expressions, certainement vont se répéter. Et donc quelque chose qui serait mal écrit au niveau d'un opérateur, d'une opération aussi profonde qu'un opérateur, pourrait avoir des répercussions assez importantes sur le programme de façon globale. Et donc utilisez les références à bon escient. Eviter des copies inutiles, comme dans cet exemple.

notes

résumé

6m 37s



## Avertissement (2)

Exemple : comparez le code suivant qui fait de 1 à 3 copies inutiles :

```
Complexe Complexe::operator+=(Complexe z2)
{
    Complexe z3;
    x += z2.x;
    y += z2.y;
    z3 = *this;
    return z3;
}
```

1 Copie

$z_1 += z_2;$   
 $z_1.operator+=(z_2);$

avec le code suivant qui n'en fait pas :

```
Complexe& Complexe::operator+=(Complexe const& z2)
{
    x += z2.x;
    y += z2.y;
    return *this;
}
```

$z_3 = (z_1$

On a ici écrit deux versions de l'opérateur d'auto affectation « += ». La version standard, qui ne fait pas de copies supplémentaires et qui est la version qu'on vous a proposée dans les transparents précédents. Et puis une version ici que je vais détailler, qui fait entre une à trois copies. Donc une à trois copies, je vais pas rentrer dans les détails ici, mais ça dépend en fait des options de votre compilateur. Donc regardons en détails cette version de l'opérateur d'auto affectation. D'abord comme type de retour ici, on a choisi de retourner un nombre complexe et non pas une référence sur un nombre complexe. Puis ensuite du côté de l'argument ici, on a aussi décidé de passer un nombre complexe et de ne pas passer par référence constante. Ce qui fait déjà que lorsque l'on a une écriture, par exemple du genre «  $z_1 += z_2$  » qui correspond donc à «  $z_1.operator+=(z_2)$  ». On a ici un passage par valeur, donc on a déjà ici possiblement une première copie du passage de cet argument. Elle peut être optimisée par le compilateur mais pas forcément, donc ça ferait possiblement, une première copie ici. Et puis donc en ce qui concerne la valeur de retour ici, si on veut utiliser la valeur de retour, ça veut dire qu'on écrirait une expression du genre

notes

résumé

7m 19s





«  $z3 = (z1 += z2)$  » Et donc ici aussi nous allons avoir, par valeur de retour ici, un nouveau nombre complexe puisqu'on a pas retourné par référence qui va être créée, et donc on aura ici une deuxième copie, qui va être effectuée. Puis si l'on regarde maintenant en détails le corps de la définition de cet opérateur puisqu'on a décidé ici de retourner un nombre complexe, on se dit on va le déclarer en interne pour pouvoir le retourner. Le reste ici des opérations est comme d'habitude, mais pour pouvoir retourner ce complexe, je veux retourner l'instance courante et donc je recopie ici avec une troisième copie. Je recopie l'instance courante. Je rappelle que c'est le contenu pointé par le pointeur « this » donc c'est bien la valeur de l'instance courante que je recopie ici, dans «  $z3$  ». Donc voilà les trois copies possibles. Certaines de ces copies peuvent être optimisées par vos compilateurs. Alors que dans cette version ici on passe déjà premièrement par référence constante, donc ici on n'a pas de copie mais bien la référence sur «  $z2$  », qui est passée directement, donc pas de copie. Du côté du retour, de la même façon, on n'a pas de copie, mais on a passé la référence de l'instance courante. Et puis en interne ici, on n'a pas déclaré de complexes supplémentaires. On n'a donc pas non plus cette troisième copie. Donc tout ceci pour dire :

## notes

## résumé

8m 49s



## Opérateur d'affectation

L'opérateur d'affectation = (utilisé par exemple dans `a = b`) : `a.operator=(b);`

- ▶ est le seul opérateur universel (il est fourni de toutes façons par défaut pour toute classe)
- ▶ est très lié au constructeur de copie, sauf que le premier s'appelle lors d'une **affectation** et le second lors d'une **initialisation**
- ▶ la version par défaut, qui fait une copie de surface, est suffisante dans la très grande majorité des cas
- ▶ si nécessaire, on peut supprimer l'opérateur d'affectation :

```
class EnormeClasse {
    // ...
private:
    EnormeClasse& operator=(EnormeClasse const&) = delete;
};
```

faites bien attention lorsque vous écrivez la surcharge des opérateurs. Je voudrais conclure cette vidéo, sur les compléments de la surcharge des opérateurs, en revenant sur un opérateur un peu particulier, qui est l'opérateur « = », l'opérateur d'affectation. Cet opérateur, qui est utilisé par exemple chaque fois qu'on écrit « `a = b` », « `z1 = z2` », est d'abord un opérateur particulier, en ce sens qu'il est fourni pour toutes les classes. Il est un opérateur qu'on appelle universel. Vous n'avez pas besoin de le redéfinir, il existe déjà pour toutes les classes que vous pouvez créer. Comme nous l'avons déjà dit dans les vidéos sur la construction de copie et sur la destruction, la redéfinition possible, si vous avez besoin de cet opérateur, est intimement liée à la redéfinition du constructeur de copie. La différence entre la construction de copie et l'affectation, c'est que la construction de copie est une construction. Elle a lieu lors d'une initialisation. Alors que l'affectation opère sur des instances qui existent déjà. Comme pour le constructeur de copie et comme pour le destructeur, la version par défaut fait une copie de surface. On reviendra sur la notion de copie de surface et de copie profonde dans les prochaines semaines, mais cette copie de surface est suffisante dans la très grande majorité des cas. Donc dans la très grande majorité des cas, vous n'aurez pas besoin de redéfinir cet opérateur d'affectation. Par contre il peut être parfois nécessaire de carrément le supprimer, cet opérateur d'affectation. Supposons que l'on ait par exemple une énorme classe, c'est-à-dire une classe où, si l'on crée une instance, cette instance utilise beaucoup de place en mémoire. On voudrait absolument pas que l'on puisse copier les instances de cette classe. Une instance de cette classe nous suffit largement et on veut donc absolument éviter d'avoir comme ça des copies. Donc pour ça on va supprimer l'opérateur d'affectation,

### notes

### résumé

10m 22s



L'opérateur d'affectation = (utilisé par exemple dans a = b) : *a.operator=(b);*

- ▶ est le seul opérateur universel  
(il est fourni de toutes façons par défaut pour toute classe)
- ▶ est très lié au constructeur de copie,  
sauf que le premier s'appelle lors d'une *affectation* et le second lors d'une *initialisation*
- ▶ la version par défaut, qui fait une copie de surface, est suffisante dans la très grande majorité des cas
- ▶ si nécessaire, on peut supprimer l'opérateur d'affectation :

```
class EnormeClasse {  
    // ...  
private:  
    EnormeClasse& operator=(EnormeClasse const&) = delete;  
};
```

qui s'appelle donc « operator = » en écrivant une syntaxe qu'on avait déjà rencontrée pour supprimer le constructeur de copie qui est, en rajoutant derrière son prototype comme ça, dans la classe, c'est une surcharge interne ; en rajoutant ici « =delete ». Ça va empêcher de faire comme cela des affectations entre instances de cette énorme classe. J'en profite également pour vous montrer le prototype de cet opérateur d'affectation, qui s'appelle bien sûr « operator = » qui donc est appelé quand on a des expressions de type « a = b » c'est-à-dire que l'appel équivalent, c'est « a.operator = », surcharge interne, on ne peut pas faire de surcharge externe de l'opérateur d'affectation, « a.operator = b » Donc on prend ici quelque chose qui est de même type que « a », donc forcément, on va avoir la même classe

notes

résumé



que la classe dans laquelle on est en train de surcharger cet opérateur. Pour éviter une copie, on va le passer par référence constante et comme on a le droit en C++ d'écrire des choses du genre « `a = b = c` », sous-entendu « `a =` » le résultat de « `(b = c)` ». C'est à dire que cette expression « `(b = c)` » a également une valeur de retour, qui est la valeur de « `b` » après affectation, c'est-à-dire la valeur de « `c` » ; et qui est de même type, exactement comme « `+=` », « `-=` » de même type que la classe dans laquelle on est en train de surcharger cet opérateur. Et toujours pour les mêmes raisons, on va faire ce retour par référence, pour éviter des copies supplémentaires.

notes

## résumé

13m 13s



Si l'on doit redéfinir l'opérateur d'affectation on choisira, depuis **C++11**, le schéma suivant :

on commencera par définir une fonction `swap()` pour échanger 2 objets de la classe, (sûrement en utilisant celle de la bibliothèque `utility` (`#include <utility>`) sur les attributs)

puis on définira l'opérateur d'affectation comme suit :

```
Classe& Classe::operator=(Classe source)
// Notez le passage par VALEUR
{
    swap(*this, source);
    return *this;
}
```

Enfin on a dit qu'il n'était pas nécessaire de redéfinir cet opérateur. Dans la très grande majorité des cas, celui qui est fourni par défaut vous suffira, mais si jamais vous deviez redéfinir cet opérateur,

notes

résumé

13m 56s





Si l'on doit redéfinir l'opérateur d'affectation on choisira, depuis **C++11**, le schéma suivant :

on commencera par définir une fonction `swap()` pour échanger 2 objets de la classe, (sûrement en utilisant celle de la bibliothèque `utility` (`#include <utility>`) sur les attributs)

puis on définira l'opérateur d'affectation comme suit :

```
Classe& Classe::operator=(Classe source)
// Notez le passage par VALEUR
{
    swap(*this, source);
    return *this;
}
```

*swap(a, b)*

*void swap(Classe&a, Classe&b)
{ swap(a.x, b.x);
}*

*class Classe
{ private:
 X x;
 Y y;
};*

alors en C++ 2011, nous vous conseillons de suivre le schéma suivant : d'utiliser ce qu'on appelle la fonction « swap ». La fonction « swap » permet d'échanger deux arguments. Donc « swap(a, b) » va échanger les valeurs entre « a » et « b ». C'est à dire que la valeur de « b » va se retrouver dans « a » et la valeur de « a » va se retrouver dans « b ». Cette fonction « swap », vous la redéfiniriez vous-même pour votre classe. Donc si par exemple, dans votre classe, vous avez les attributs « X » et « Y » de types « x, y », alors vous feriez quelque chose comme ceci : la fonction « swap » ne renvoie rien, elle prend deux instances de votre classe. Mais comme elle doit les modifier pour les échanger, elle va bien sûr les prendre par référence. Et donc par exemple, utiliser la fonction « swap » fournie par la bibliothèque « utility », donc faire un « #include » au début de son programme, pour utiliser cette fonction « swap », pour échanger donc les attributs « x » des instances « a » et « b ».

notes

résumé

14m 8s







les attributs « y » des instances « a » et « b », et ainsi de suite pour tous les attributs qu'il faudrait donc échanger. Et le schéma pour la redéfinition de l'opérateur d'affectation, que nous vous conseillons, est : et donc avec ici une petite différence pour des raisons d'optimisation en C++ 2011, qui dépasse le cadre de ce cours, mais qui permet au compilateur soit d'éviter de faire des copies, soit même d'utiliser ce qu'on appelle le déplacement. Mais ça nous amènerait un peu trop loin. Donc ça c'est vraiment une raison d'optimisation encore plus avancée. Ici on a bien effectivement pris un passage par valeurs, et donc d'utiliser cette fonction « swap » qui va échanger le contenu de l'instance courante avec le contenu de cette variable ici, qui est le paramètre de l'opérateur d'affectation qu'on est en train de redéfinir. Donc ici, ça va bien, si j'écris « a = b », ici ça va bien être un « swap », entre 'a', qui est l'instance courante, c'est bien « a.operator = b » Donc ici \*« this » qui jouerait le rôle de « a » et puis « source » qui serait ici une copie, ou alors un déplacement de « b ». Et donc on échange les deux valeurs, ce qui fait qu'on a bien effectivement la valeur de « b » qui est mise dans « a ». Et attention ici, cet échange se fait avec la copie locale source, donc c'est pas du tout la valeur de « a » qui est mise dans « b », mais la valeur de « a » est mise dans « source », c'est-à-dire ce dans quoi on a copié « b ». Donc il n'y a aucun risque de pollution de « b ». Et enfin toujours pour les mêmes raisons que pour l'opérateur « += », on retourne ici la

## notes

## résumé

15m 25s





référence sur l'instance courante, puisqu'on peut tout à fait écrire des expressions du genre «  $c = a = b$  ». Voilà tout ceci qui conclut cette dernière séquence vidéo, certainement un peu technique et avancée, sur tous les compléments, que nous voulions vous apporter concernant ce sujet assez pointu, de la surcharge des opérateurs. de la surcharge des opérateurs.

notes

---

---

---

---

---

---

---

---

---

---

résumé

---

---

---

---