

Support de cours

Cours:

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

Vidéo:

W13-05-surchopcomplements-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Surcharge des opérateurs. Fonction de votre niveau de programmation. Type de retour. Expressions du genre. Opérateurs d'auto affectation. Exemple des nombres complexes. Premier temps. Valeurs de retour. Séquence vidéo. Surcharge d'opérateur. Premier niveau. Opérateur d'affichage. Conseils pratiques. Type de base. Nombre complexe.



[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 1)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





Dans cette séquence vidéo,

notes

résumé

0m 1s



En pratique, quelle surcharge des opérateurs ?

Dans votre pratique du C++, vous pouvez, en fonction de votre niveau :



nous allons donc vous apporter quelques compléments sur les aspects que nous avons laissés de côté dans les vidéos précédentes, concernant la surcharge des opérateurs. Et nous vous donnerons aussi quelques conseils pratiques.

notes

résumé

0m 5s





Au niveau suivant, vous pouvez considérer les opérateurs d'auto affectation, donc « += », *« = », mais les surcharger en ignorant leurs valeurs de retour, donc on mettant un type de retour « void ». Et enfin, un dernier niveau, vous pouvez faire la surcharge des opérateurs, en respectant strictement leurs prototypes comme ceux qui sont donnés dans la norme, et donc par exemple, pour les opérateurs d'auto affectation, retourner une référence sur une instance de la classe. Voilà donc pour ces 3 niveaux de surcharge,

notes

résumé

1m 9s



Exemples de surcharges de quelques opérateurs usuels

(au niveau ④ du tr. précédent)

```
bool operator==(Classe const&) const; // ex: p == q
bool operator<(Classe const&) const; // ex: p < q

Classe& operator+=(Classe const&); // ex: p += q
Classe& operator--(Classe const&); // ex: p -= q

Classe& operator*=(autre_type const); // ex: p *= x;

Classe& operator++(); // ex: ++p
Classe& operator++(int inutile); // ex: p++

const Classe operator-() const; // ex: r = -p;

// ===== surcharges externes =====
const Classe operator+(Classe, Classe const&); // r = p + q
const Classe operator-(Classe, Classe const&); // r = p - q

ostream& operator<<(ostream&, Classe const&); // ex: cout << p;
const Classe operator*(autre_type, Classe const&); // ex: q = x * p;
```

4 si l'on inclut aussi le fait de ne pas surcharger, que nous vous proposons en pratique donc, pour la surcharge des opérateurs. Regardons un petit peu quelques exemples en détails au dernier niveau du transparent précédent. Nous avons déjà vu dans les séquences vidéos qui précèdent, l'opérateur d'auto affectation « += » qui permet donc d'écrire des expressions du genre « p += q ». Nous avons vu aussi en surcharge externe, l'addition, qui permet donc de faire des expressions du genre « r = p + q » donc c'est ici l'opérateur d'addition. De la même façon, on pourrait définir l'opérateur de soustraction, l'opérateur d'auto-soustraction, qui permettrait d'écrire « p -= q ». En considérant la la soustraction, on a aussi le signe, c'est pas la soustraction ici, mais c'est l'opposé. Donc si par exemple j'écris « r = -p », à ce moment-là c'est aussi un opérateur -, qui sera surchargé en interne et qui ne prend pas ici d'arguments. Il n'y a pas de second argument à cet opérateur. C'est ce qu'on appelle un opérateur unaire. Nous avons aussi vu l'opérateur d'affichage, qui permet donc d'afficher les instances de nos classes, comme on affiche un « double » ou un « int ». Nous avons aussi vu donc en multiplication externe, la multiplication à gauche par un élément d'une autre classe. Donc par exemple ici « p » et « q » sont des éléments de notre classe, donc par exemple des nombres complexes, mais « x » ici est un élément qui est d'une autre classe ou d'un type de base. Donc par exemple on avait pris un « double ». On pourrait aussi par exemple avoir, en interne, la multiplication, donc justement par un de ces autres types. On pourrait ici par exemple, pour un nombre complexe « p », faire p* = x, où x est un double. D'autres exemples encore, on pourrait

notes

résumé

1m 37s



Exemples de surcharges de quelques opérateurs usuels

(au niveau ④ du tr. précédent)

```
bool operator==(Classe const&) const; // ex: p == q
bool operator<(Classe const&) const; // ex: p < q

Classe& operator+=(Classe const&); // ex: p += q
Classe& operator--(Classe const&); // ex: p -= q

Classe& operator*=(autre_type const); // ex: p *= x;

Classe& operator++(); // ex: ++p
Classe& operator++(int inutile); // ex: p++

const Classe operator-() const; // ex: r = -p;

// ===== surcharges externes =====
const Classe operator+(Classe, Classe const&); // r = p + q
const Classe operator-(Classe, Classe const&); // r = p - q

ostream& operator<<(ostream&, Classe const&); // ex: cout << p;
const Classe operator*(autre_type, Classe const&); // ex: q = x * p;
```

écrire l'opérateur « == », pour par exemple faire « p == q ». Donc dans ce cas-là faites attention aussi à penser à surcharger l'opérateur « != », qui est intimement lié à l'opérateur « == », en définissant l'un à partir de l'autre. On pourrait aussi faire des opérateurs de comparaisons ; donc par exemple pour faire « p < q ». Là aussi, pensez à ce moment-là à faire les opérateurs inférieur ou égal, et si ça a un sens, les opérateurs supérieur et supérieur ou égal, en définissant l'un de ces 4 et en écrivant les 3 autres à partir de l'un de ces 4, pour garantir l'intégrité sémantique, c'est-à-dire le fait qu'ils se comportent correctement tous ensemble. Voilà donc différents exemples encore : l'opérateur d'auto-incrément, ici « ++p » comme ceci avec cette écriture.

notes

résumé

```
const Complexe operator+(Complexe, Complexe const&);
```

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

operator

Dans la suite nous allons tour à tour nous intéresser à : Pourquoi avoir mis un « `const` » au type de retour de l'addition ? Pourquoi ce type de retour pour l'opérateur d'affichage ? Quel type de retour pour l'opérateur d'auto-affectation, par exemple « `+=` » ? Pourquoi ici une référence sur une instance de classe ? Et puis nous terminerons par quelques conseils pratiques et reviendrons sur l'opérateur d'affectation « `=` ». Commençons donc par le type de retour de, par exemple, l'opérateur d'addition. Pourquoi avoir mis un « `const` » comme type de retour de cet opérateur, qui je vous le rappelle nous permet d'écrire des expressions du genre « `z3 = z1 + z2` ». Donc le type de retour ici, clairement, est un nombre complexe. Mais pourquoi avoir mis « `const` » ? Eh bien simplement pour ne pas permettre des expressions du genre « `z1 + z2` », qui est bien effectivement l'équivalent

notes

résumé

4m 13s



```
const Complexe operator+(Complexe, Complexe const&);
```

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

operator+(z1, z2)

de l'appel « `operator + (z1,z2)` »

notes

résumé

5m 11s



```
const Complexe operator+(Complexe, Complexe const&);
```

```
z3 = z1 + z2;
```

```
++(z1 + z2);
```

```
z1 + z2 • f(x);
```

$++(\text{operator}+(z_1, z_2)) ??$

Donc c'est bien le type de retour de cet opérateur. Et puis ensuite, on pourrait imaginer écrire comme ça, incrémenter cette valeur de retour. Qu'est-ce que ça aurait comme sens de faire ceci ? Qu'est-ce que ça aurait comme sens d'incrémenter l'addition de « `z1,z2` » ? Le résultat de l'addition de « `z1,z2` » c'est un résultat, c'est une constante, on ne peut pas l'incrémenter. Et donc pour empêcher justement ce genre d'écriture, on va mettre le type « `const` » pour aussi empêcher ce genre d'écriture,

notes

résumé

5m 14s



const `Complexe operator+(Complexe, Complexe const&);`

```
z3 = z1 + z2;
```

```
++(z1 + z2);
```

```
z1 + z2(= f(x)); ??
```

++ (operator+(z1, z2)) ??



de l'écrire donc du côté gauche d'une expression « = » Par exemple « `z1 + z2` » qui serait, sans « `const` », un nombre complexe, qu'on pourrait mettre ici donc du côté gauche d'un signe égal. Là aussi ce genre d'expression n'a absolument aucun sens. Donc pour empêcher le fait de le mettre de ce côté-là d'un signe égal, on va donc définir ce type de retour comme étant un « `const complexe` »,

notes

résumé

5m 49s



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

c'est-à-dire un complexe que l'on ne peut pas modifier, le résultat d'une addition, c'est ce nombre complexe. On peut pas modifier le résultat d'une addition.
Deuxième point : pourquoi l'opérateur d'affichage retourne t-il donc un « `ostream&` » ? Regardons par exemple un affichage tout à fait standard, par exemple pour un complexe « `z1` », qu'on pourrait écrire comme ça « `cout << z1 << endl` » « `cout << z1 << endl` »

notes

résumé

6m 13s

