Support de cours

Cours:

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

Vidéo:

## W13-03-surchppexterne-CPP-pt2

Concepts (extraits des sous-titres générés automatiquement) :

**Multiplication d'un nombre complexe. Instance de classe. Appel d'une fonction. Appel d'une méthode. Nombre complexe. Plupart des cas. Classe ostream. Type complexe. Surcharge de l'opérateur d'affichage. Surcharge externe. Écritures possibles. Premier cas. Classe complexe. Appel de la méthode de cette classe. Partie publique de la classe.**

vers la recherche de séquences vidéo
(dans Introduction à la programmation orientée objet (en C++).)

vers la vidéo

EPFL

# Surcharge d'opérateurs : surcharge externe
(Partie 2)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s

# Nécessité de la surcharge externe

Nous avons vu qu'il existe 2 surcharges, interne et externe,

notes

résumé

0m 1s

# Nécessité de la surcharge externe

EPFL

La surcharge **externe** est nécessaire pour des opérateurs concernés par une classe, mais pour lesquels la classe en question **n'est pas** l'opérande de gauche.

Exemples :

1. multiplication d'un nombre complexe par un double :

```
double x;
Complexe z1, z2;
// ...
z2 = x * z1;
```

☞ `z2 = x.operator*(z1);`
ou `z2 = operator*(x, z1);`

Le premier **n'a pas de sens** (`x` n'est pas un objet, mais de type élémentaire `double`)

2. écriture sur `cout` : `cout << z1;`
☞ `cout.operator<<(z1);` ou `operator<<(cout, z1);`,
mais on souhaite le surcharger pour la classe `Complexe` et non pas dans la classe de `cout` (`ostream`).

et que dans la plupart des cas on peut choisir entre surcharge externe et surcharge interne. Mais il y a des cas où on ne peut pas choisir et où il est absolument nécessaire d'utiliser la surcharge externe. Regardons plus en détail ces cas, le premier cas est lorsque l'on mélange différents types, par exemple ici, si l'on veut faire la multiplication d'un nombre Complexe avec un double. Par exemple, on a ici un double " x " et un nombre Complexe " z1 ", et on veut récupérer le résultat de x fois z1 dans z2. Donc x de type double, z1 de type Complexe, nous donnerait comme réécriture en principe, soit " z2 = x.operator* " avec une surcharge interne appel d'une méthode, soit appel d'une fonction " z2 = operator* " de " x, z1 ". Mais il est clair que la première ligne n'a absolument pas de sens, puisqu'ici x n'est pas une instance de classe, mais est ce qu'on appelle un type de base, et de type double, double n'est pas une classe, donc il n'y a pas de méthode dans double, et donc cette solution est impossible. Là on est obligés d'utiliser ce qu'on appelle la surcharge externe, d'utiliser des fonctions. De même, dans le cas où on voudrait afficher un complexe, par exemple faire " cout << z1 " en utilisant la surcharge de l'opérateur d'affichage inférieur-inférieur (<<). On aurait donc 2 écritures possibles, soit la méthode operator " cout est une instance, la classe ostream,

notes

résumé

0m 5s

# Exemple de la multiplication externe

```
double x;
Complexe z1, z2;
// ...
z2 = x * z1;
```

```
const Complexe operator*(double a, Complexe const& z)
{
  /* Soit l'écrire explicitement,
     soit, quand c'est possible, utiliser l'opérateur interne :
  */
  return z * a;
}
```

et donc cette écriture ici, avec un appel de la méthode de cette classe ostream, ou alors la fonction qui prend ici les 2 paramètres cout et z1. Mais ce qui nous intéresse est de surcharger la classe Complexe, certainement pas d'aller toucher à la classe ostream dont cout est une instance, et donc dans ce cas aussi, on ne va pas du tout utiliser la surcharge interne à la classe ostream, mais bien préférer la surcharge externe. Examinons ces 2 exemples en détail et commençons par la multiplication externe, où je vous rappelle, on a déclaré un doublex ici, on a 2 Complexe, z1 et z2, et on souhaite faire la multiplication de x par z1 et mettre le résultat dans z2. L'appel équivalent serait que " z2=operator*(x,z1); ", ce qui nous donne donc comme prototype que la fonction retourne un nombre Complexe, pour les mêmes raisons qui seront expliquées plus tard, on préfèrera qu'il soit const, on aura operator*, et puis ensuite on prend un double et un Complexe que pour des raisons d'optimisation, on passera par référence constante. On arrive donc au prototype suivant. Et pour la définition, on pourrait soit l'écrire explicitement, écrire toutes les opérations

notes

résumé

1m 37s

# Exemple de l'opérateur d'affichage

Exemple (d'appel) :

```
cout << z1;  // appel équivalent :  operator<<(cout, z1);
```

dont le prototype est (*hors* de la classe) :

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

qui font la multiplication de ce double, ici a, par ce Complexe, z. Soit, si on suppose qu'il existe en interne, et cela est possible pour les nombres complexes, mais pas forcément possible pour toutes les classes, mais lorsque c'est possible, qu'il existe en interne (notez bien ici qu'on a changé l'ordre et qu'on a commencé par multiplier par z), donc qu'il existe ici quelque chose qui est capable de faire Complexe fois double, c'est-à-dire soit une surcharge interne dans la classe Complexe, qui prendrait un double, soit que l'on ait déjà écrit en surcharge externe le produit par un Complexe d'abord, et par un double ensuite, peu importe ici la solution choisie, mais ce qui compte c'est d'écrire que le fait de faire a fois z, est la même chose mathématiquement que de faire z fois a, et donc ne surtout pas, dans ce cas, lorsqu'on a ce genre d'égalité, réécrire 2 fois le code, mais utiliser du code que l'on a déjà écrit.

notes

résumé

3m 13s

# Définitions de l'opérateur d'affichage

Via des accesseurs :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.get_x() << ", " << z.get_y() << ')';
  return sortie;
}
```

Pour le cas de l'opérateur d'affichage, si on veut par exemple pouvoir faire " cout << z1 ", où bien sûr ici z1 est un complexe, dont l'appel équivalent, je vous rappelle, serait la surcharge externe, donc la fonction " operatorcout et z1 comme argument, ce qui nous donne donc comme prototype pour cette fonction " operatorcout, et ici un Complexe que l'on passe encore une fois, pour des raisons d'optimisation, par référence constante. À noter qu'ici, cout, qui est une instance de la classe ostream, est passé par référence. En effet, l'affichage en question, " cout << z1 ", va bien modifier cout, puisqu'on va écrire des informations sur cout, on va le modifier, et donc comme on modifie cet argument, il faut absolument le passer par référence. Le type de retour de cette fonction, je vous le donne tout de suite, pour que vous puissiez écrire directement correctement cet opérateur d'affichage, c'est aussi un " ostream& ". La raison de ce type de retour vous sera, encore une fois, expliquée dans la dernière vidéo sur la surcharge des opérateurs, qui regroupe toutes ces subtilités. Passons maintenant à la définition

notes

résumé

4m 32s

## Définitions de l'opérateur d'affichage

EPFL

Via des accesseurs :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.get_x() << ", " << z.get_y() << ')';
  return sortie;
}
```

x = 3
y = 4
(3,

de cet opérateur d'affichage pour les complexes. La première solution consiste à utiliser des accesseurs, par exemple get x, get y, qui retournent les attributs que l'on souhaite afficher, ce qui nous donnerait donc : ici on a reçu comme premier paramètre le flot de sortie sur lequel on veut écrire, et on utilise donc ce flot pour afficher ici par exemple une parenthèse, l'idée est que si j'ai un complexe dont x vaut 3, et dont y vaut 4,

notes

résumé

5m 47s

# Définitions de l'opérateur d'affichage

EPFL

Via des accesseurs :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.get_x() << ", " << z.get_y() << ')';
  return sortie;
}
```

Via une autre méthode :

string Complexe :: to_string

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << z.to_string()
  return sortie;
}
```

ou :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  return z.affiche(sortie);
}
```

que l'on écrive " (3,4) ", c'est le format que l'on veut utiliser ici pour l'affichage. Donc on écrit la parenthèse ouvrante, puis on récupère x, du complexe que l'on a reçu comme deuxième argument à cette fonction " operator<< ". On affiche la virgule, puis on affiche la valeur récupérée pour l'attribut y de ce complexe z. Enfin on termine par la parenthèse fermante. Et on n'oublie pas, pour les raisons de ce retour, de cette valeur de retour ici, " ostream& ", de terminer cette fonction par un return du paramètre sortie, pour l'instant considérez-le comme une obligation afin d'éviter d'avoir des segmentation fault, et de pouvoir utiliser correctement l' " operator<< ", comme vous l'utilisez pour les int et pour les double. Une deuxième définition possible consiste à utiliser une méthode que l'on aurait déjà définie pour, soit convertir un Complexe en une chaîne de caractères, soit directement afficher un complexe. Regardons la première solution, ici on utiliserai de nouveau le flot que l'on a reçu pour afficher le résultat de l'appel de la méthode to_string, qui transformerait un Complexe en une chaîne de caractères, dont le prototype serait typiquement une chaîne de caractères comme valeur de retour. Donc, " string ", la méthode appartient à la classe Complexe et s'appelle " to_string ", elle ne prend pas de paramètres,

notes

résumé

6m 25s

# Définitions de l'opérateur d'affichage

et elle ne modifie pas l'instance courante, en effet, convertir un Complexe en une chaîne de caractères ne modifie pas ce Complexe. Et donc on ferait appel à cette méthode, ici, dans l' " operator << ", puis on terminerait comme précédemment, en retournant le premier argument reçu. Une autre solution pourrait consister à avoir directement une méthode affiche, qui a le même comportement que " operator << ", mais qui serait une méthode publique, dans la classe, qui reçoit le flot sur lequel on l'affiche et retourne ce flot, c'est-à-dire que son prototype serait donc similaire à celui de l'opérateur d'affichage, donc retourne un ostream, c'est une méthode de la classe complexe, qui s'appelle affiche, et qui prend comme paramètre un flot qu'elle modifie, donc on passe par référence, et puis cette méthode ne modifie pas le complexe qui est affiché, quand on affiche un Complexe, on ne le modifie pas, et donc on termine ici avec le mot clé const.

notes

résumé

8m 1s

# Définitions de l'opérateur d'affichage

EPFL

Via des accesseurs :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.get_x() << ", " << z.get_y() << ')';
  return sortie;
}
```

Via une autre méthode :

string Complexe::to_string() const;

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << z.to_string()
  return sortie;
}
```

ou :

ostream& Complexe::affiche(ostream& ) const;

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  return z.affiche(sortie);
}
```

Voilà donc 3 exemples possibles de définition de cet opérateur d'affichage pour cet exemple de classe Complexe, dont vous pouvez vous inspirer pour vos propres classes. À noter, par anticipation sur un sujet qui viendra dans plusieurs semaines,

notes

résumé

9m 8s

# Définitions de l'opérateur d'affichage

qu'il est préférable d'utiliser ce genre de méthodes toString ou affiche lorsque l'on aura affaire à utiliser le polymorphisme, qui viendra plus tard. Lorsque l'on fait ainsi de la surcharge externe des opérateurs, il est parfois nécessaire d'accéder aux attributs privés

notes

résumé

9m 22s

# Définitions de l'opérateur d'affichage

EPFL

Via des accesseurs :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.get_x() << ", " << z.get_y() << ')';
  return sortie;
}
```

Via une autre méthode :

string Complexe::to_string() const;

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << z.to_string()
  return sortie;
}
```

ou :

ostream& Complexe::affiche(ostream&) const;

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  return z.affiche(sortie);
}
```

de la classe sur laquelle on veut faire porter l'opérateur.

notes

résumé

9m 37s

## Surcharge externe / `friend`

EPFL

Parfois, il peut être nécessaire d'autoriser les opérateurs externes d'accéder à certains éléments `private`.
(mais *préférez passer par les accesseurs !*)

Dans ce cas, ajoutez, **dans** la définition de la classe, leur prototype précédé du mot clé `friend` :

```
friend const Complexe operator*(double, Complexe const&);
friend ostream& operator<<(ostream&, Complexe const&);
```

Le mot clé `friend` signifie que ces fonctions, bien que *ne* faisant *pas* partie de la classe, peuvent avoir accès aux attributs et méthodes `private` de la classe.

Remarque : les définitions restent **hors** de la classe (et **sans** le mot clé `friend`)

Même si je trouve que c'est de la mauvaise programmation, et que je vous conseille vraiment de tout le temps utiliser des accesseurs, et d'utiliser l'interface, donc la partie publique de la classe. Mais néanmoins, vous pourrez rencontrer ce genre de pratique (je vous déconseille de l'écrire vous-même), et par exemple être tentés ici d'écrire " z.x " ou ici " z.y ", c'est-à-dire d'accéder directement aux attributs privés x, y, de la classe Complexe. Dans ce cas là, il va falloir donner un accès privilégié à cette fonction, qui est une fonction extérieure à la classe Complexe, lui donner un accès privilégié aux parties privées. C'est ce que l'on appelle déclarer une amitié ou un friendship par le mot clé friend. Je répète que je vous recommande très fortement de préférer passer par des accesseurs, c'est-à-dire par la partie publique de la classe, plutôt que d'accéder à la partie privée par ce moyen détourné qu'est le mot clé friend. Néanmoins comme on le rencontre, si vous deviez le faire et ne pas suivre ce conseil, vous mettriez dans la définition même de la classe, une ligne comme " friend " et ensuite le prototype de la fonction à laquelle vous voulez donner accès aux parties privées de la classe. Donc quand on fait comme ceci, " friend " suivi d'un prototype d'une fonction, cela donne à cette fonction le droit d'accéder aux méthodes et aux attributs privés de la classe. À noter que la définition de ces fonctions reste bien entendu hors de la classe,

notes

résumé

9m 41s

## friend : exemple

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.x << ", " << z.y << ')';
  return sortie;
}

// ...

class Complexe {
  friend ostream& operator<<(ostream& sortie, Complexe const& z);
  // ...
private:
  double x;
  double y;
};
```

ces fonctions n'ont rien à voir avec la classe, c'est simplement dans la classe que l'on dit qu'on offre à ces fonctions extérieures le droit d'accéder.

notes

résumé

11m 13s

## friend : exemple

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.x << ", " << z.y << ')';
  return sortie;
}

// ...

class Complexe {
  friend ostream& operator<<(ostream& sortie, Complexe const& z);
  // ...
private:
  double x;
  double y;
};
```

Regardons concrètement sur les nombres complexes ce que cela pourrait donner, on aurait donc ici la classe Complexe, avec comme parties privées les attributs x et y, et on aurait ici notre opérateur d'affichage pour les nombres complexes,

notes

résumé

11m 22s

## friend : exemple

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.x << ", " << z.y << ')';
  return sortie;
}

// ...

class Complexe {
  friend ostream& operator<<(ostream& sortie, Complexe const& z);
  // ...
private:
  double x;
  double y;
};
```

comme on l'a vu précédemment, mais on a pas suivi le conseil

notes

résumé

11m 37s

## friend : exemple

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.x << ", " << z.y << ')';
  return sortie;
}

// ...

class Complexe {
  friend ostream& operator<<(ostream& sortie, Complexe const& z);
  // ...
private:
  double x;
  double y;
};
```

(je vous déconseille, bien sûr, de le faire), et on essaie ici d'accéder aux parties privées, si on le fait sans rien faire de particulier, on va avoir une erreur de compilation ici,

notes

résumé

11m 40s

## friend : exemple

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
  sortie << '(' << z.x << ", " << z.y << ')';
  return sortie;
}

// ...

class Complexe {
  friend ostream& operator<<(ostream& sortie, Complexe const& z);
  // ...
private:
  double x;
  double y;
};
```

puisqu'on a pas le droit, avec une fonction externe, d'accéder aux parties privées. Et donc pour que tout ceci fonctionne, on rajoute dans la classe Complexe une ligne qui donne le droit d'accès, qui donne le friendship à cette fonction extérieure dont on met juste le prototype derrière le mot clé friend. En ajoutant cette ligne,

notes

résumé

11m 50s

ça veut dire que la fonction en question a maintenant le droit d'accéder aux parties privées, ce qui personnellement, je trouve, est une très mauvaise chose, puisque cela casse l'encapsulation. Ceci conclut notre séquence sur la surcharge externe. La prochaine séquence vidéo vous présentera la surcharge interne, et vous dira dans quels cas il faut préférer la surcharge interne ou la surcharge externe. Et puis dans une dernière vidéo, nous viendrons sur tous les points de détails que nous avons encore laissé en suspens. laissé en suspens.

notes

résumé

12m 11s