

Support de cours

Cours:

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

Vidéo:

W15-03-polymasq-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Masquage de méthodes. Super-classe. Classe b. Overriding des méthodes virtuelles. C telle. Méthodes virtuelles. Règles de résolution de portée. Méthode de l'instance réelle. Méthode. Surcharge de fonction. Première méthode m1. Nouveaux mots-clé. Paramètres différents. A. ces. Cours de notre apprentissage.



[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/>

Masquage, substitution et surcharge

(Partie 1)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



Nous avons rencontré **trois** concepts **différents** :

- ▶ la surcharge (*overloading*) de fonctions et de *méthodes* ;
- ▶ le masquage (*shadowing*) (en particulier de *méthodes*) ;
- ▶ (sans la nommer jusqu'ici) la substitution (ou redéfinition, *overriding*), dans les sous-classes, de nouvelles versions de **méthodes virtuelles**.



Pour les *méthodes virtuelles*, on pourrait donc avoir les trois ?!

est **qui est quoi exactement ?**

Par ailleurs, **C++11** introduit deux nouveaux mots clés, optionnels, pour justement aider le programmeur à préciser ses intentions :

override et **final**

Au cours de notre apprentissage, nous avons rencontré trois concepts différents La surcharge de fonction ou de méthode. en anglais on appelle ça overloading. Le masquage en particulier le masquage de méthodes, qu'on appelle en anglais shadowing et la substitution ou redéfinition qu'en anglais on appelle overriding des méthodes virtuelles. Le dernier ne s'applique qu'aux méthodes virtuelles. Donc pour des méthodes virtuelles on pourrait en principe avoir les trois : la surcharge, du masquage et de la substitution. Qu'est-ce qui est quoi exactement ? Pour éviter la confusion de certains programmeurs, depuis C++2011, on a deux nouveaux mots-clé dont je vous parlerai en fin de cette séquence,

notes

résumé

0m 1s



- ▶ **surcharge** : même nom, mais paramètres différents, dans la même portée
(Note : en C++, il ne peut y avoir surcharge que dans la même portée.
Cf leçon à venir sur l'héritage multiple)
- ▶ **masquage** : entités de mêmes noms mais de portées différentes, masqués par les règles de résolution de portée.
Pour les méthodes :
 - ▶ Attention aux subtilités : une seule méthode de *même nom* suffit à les masquer toutes, indépendamment des paramètres !
- ▶ **substitution/redéfinition des méthodes virtuelles**
 - ▶ résolution *dynamique* : c'est la méthode de l'instance qui est appelée (si pointeur ou référence)
 - ▶ Si l'on redéfinit qu'une seule méthode (virtuelle) surchargée, alors les autres sont masquées

qui sont override et final qui permettent de préciser les intentions du programmeur. Reprenons donc ces trois concepts en détail pour éclaircir précisément de quoi il s'agit. Avant d'illustrer ces concepts sur un exemple, rappelons leur définitions. La surcharge de fonction ou de méthode c'est lorsqu'une fonction ou une méthode a le même nom qu'une autre mais des paramètres différents et qu'elles sont dans la même portée. Attention, en C++ on ne peut avoir de la surcharge que dans la même portée. Nous y reviendrons plus tard. Le masquage c'est lorsque des entités, des variables, des fonctions et puis dans le cadre de l'orienté-objet donc des attributs, des méthodes, ont le même nom mais dans des portées différentes. Du coup par les règles de résolution de portée, en C++ on résout à la portée la plus proche, donc celui qui a la portée la plus proche masque celui qui a une portée un peu plus lointaine. Attention, une seule méthode de même nom suffit à masquer toutes les autres méthodes de portée plus lointaine, indépendamment de leurs paramètres puisque justement en ce qui concerne les paramètres c'est la notion de surcharge qui aurait lieu et la surcharge n'a lieu que dans la même portée. Donc si on est dans une autre portée, il suffit qu'on ait simplement le même nom pour masquer toutes les méthodes. Nous allons y revenir sur un exemple. Enfin la substitution ou aussi redéfinition n'est valable que pour les méthodes virtuelles et il s'agit ici de redéfinir dans une sous-classe une méthode virtuelle héritée d'une super-classe qui permet à ce moment là d'avoir ce qu'on appelle la résolution dynamique des liens et le polymorphisme.

notes

résumé

0m 49s



```
class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};
```

Ce sera la méthode de l'instance réelle qui sera appelée si elle est appelée au travers d'un pointeur ou d'une référence et de même que pour le masquage si l'on redéfinit une seule méthode virtuelle, même si elle n'a pas les mêmes paramètres, alors toutes les autres seront alors masquées. L'une seulement sera définie mais elle masquera simplement les autres. la règle de masquage qui s'applique aussi pour les méthodes substituées redéfinies. Repassons en détail sur un exemple concret :

notes

résumé

2m 25s



```
int main(){
    B b;
    //b.m1(2);    // NON : no matching function for call to 'B::m1(int)'
    b.A::m1(2); // ... mais elle est bien là
    b.m1("2");

    C c;
    c.m1(2); // Attention ici : c'est celle avec double !!
    //c.m1("2"); // NON : no matching function
    c.A::m1("2"); // OK
    c.A::m1(2);  // OK, et là c'est celle avec int

    return 0;
}
```

imaginons donc que nous ayons trois classes A, B et C Telle que B hérite de A et C hérite aussi de A. Ces trois classes vont donc avoir une méthode m1 sur laquelle nous allons faire de la surcharge, de la substitution et du masquage. Pour rendre l'exemple un petit peu plus concret,

notes

résumé

2m 54s



Masquage, substitution et surcharge : exemples

```

class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};

```

dans le main nous avons une instance B de la classe B et une instance C de la classe C. Nous allons regarder les différents cas d'appels de ces méthodes m1. Dans la classe A, nous avons une première méthode m1 qui prend un entier. Chacune de ces méthodes va se contenter d'afficher un message indiquant essentiellement la méthode qui est appelée. De plus, dans A nous avons aussi une autre méthode m1 qui diffère de la première par sa liste de paramètres. Ici elle prend une chaîne de caractères au lieu de prendre un entier. Ces deux méthodes m1 sont dans la même portée, la portée de la classe A. Elles diffèrent par leurs listes de paramètres alors qu'elles ont le même nom. C'est donc bien un cas classique de surcharge dont-il s'agit ici. Dans la classe B, nous avons ici substitution de l'une des deux méthodes. Nous avons ici substitution de la méthode qui a comme argument la chaîne de caractères. Cette méthode m1 redéfinie ici dans la sous-classe B de la classe A masque donc la méthode m1 de A dans B. Comme c'est la seule des deux qui a été substituée dans B, d'après les règles de résolution de portée et de masquage, celle-ci masque aussi l'autre méthode puisque dans la classe B nous n'aurons plus qu'une seule méthode m1 accessible. La classe B aura bien effectivement 3 méthodes m1. Sa propre méthode m1 et les deux méthodes m1 héritées de la super-classe A mais par les règles de résolution de portée

notes

résumé

3m 13s



```
int main(){
    B b;
    //b.m1(2);    // NON : no matching function for call to 'B::m1(int)'
    b.A::m1(2); // ... mais elle est bien là
    b.m1("2");

    C c;
    c.m1(2); // Attention ici : c'est celle avec double !!
    //c.m1("2"); // NON : no matching function
    c.A::m1("2"); // OK
    c.A::m1(2);   // OK, et là c'est celle avec int

    return 0;
}
```

lorsqu'on est dans B, on résout à la portée la plus proche, la seule qui est donc directement accessible c'est celle directement définie dans B qui masque les deux méthodes dans A. Voyons ceci sur le main.

notes

résumé

4m 49s




```
int main(){
    B b;
    //b.m1(2);    // NON : no matching function for call to 'B::m1(int)'
    b.A::m1(2); // ... mais elle est bien là
    b.m1("2");

    C c;
    c.m1(2); // Attention ici : c'est celle avec double !!
    //c.m1("2"); // NON : no matching function
    c.A::m1("2"); // OK
    c.A::m1(2);   // OK, et là c'est celle avec int

    return 0;
}
```

Nous avons ici notre instance B On ne peut pas faire un appel comme ceci. C'est pour ça que nous l'avons commenté. de la méthode m1 qui passe un entier puisque la méthode m1 qui reçoit un entier hérité de A est masquée par la méthode dans B qui prend une chaîne de caractères. On peut bien effectivement appeler la méthode m1 qui prend une chaîne de caractère. Cela ne pose aucun souci. Mais on ne peut pas appeler la méthode m1 qui prend un entier.

notes

résumé

5m 4s



Masquage, substitution et surcharge : exemples

```

class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "D::m1(double) : " << x << endl; }
};

```

On a un message qui nous indique que cette méthode n'est pas callable. Pourtant elle existe bien, si on veut l'appeler, il faudra utiliser l'opérateur de résolution de portée en disant "la méthode m1 de portée la classe A qui appartient à l'instance B de la sous-classe B, laquelle hérite effectivement cette méthode de la classe A puisque B est un A. Cette méthode est donc effectivement ici et peut être appelée mais il faut la démasquer avec l'opérateur de résolution de portée. Pour la classe C nous avons ici introduction d'une troisième méthode d'une nouvelle méthode, puisqu'on a un troisième type de paramètre avec un double ceci va redéfinir une méthode m1, une troisième, qui va, de nouveau, avec les mêmes règles de résolution de portée, masquer les deux autres héritées de A. masquer les deux autres héritées de A.

notes

résumé

5m 37s

