

Support de cours

Cours:

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

Vidéo:

### W16-02-heritmultmasquage-CPP-pt2

Concepts (extraits des sous-titres générés automatiquement) :

**Utilisateur de la classe ovovivipare. Cas de notre exemple. Classe vivipare. Déclaration spéciale. Problème de résolution de portée. Bonne solution. Opérateur de résolution de portée. Concepteur de la classe ovovivipare. Instance d'ovipare. Meilleure solution. Méthode. Sous-classe ovovivipare. Programme principal. Choix d'implémentation. Attributs ambigus.**



[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/>

# Héritage multiple : masquage

(Partie 2)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



Solutions ?

Première solution : utiliser l'*opérateur de résolution de portée*.

```
int main()
{
    Ovovivipare o(...);
    o.Vivipare::afficher("Un orvet : ");

    return 0;
}
```

mais...

Et quelles solutions s'offrent à nous maintenant pour résoudre ce problème de résolution de portée ? Première possibilité : simplement utiliser l'opérateur de résolution de portée. Donc, ici par exemple, dans le programme principal qui crée l'instance d'Ovipare, au moment où l'on veut appeler la méthode « afficher », il faut préciser dans quelle portée on va chercher cette méthode « afficher » ; par exemple dire

notes

résumé

0m 1s





que nous invoquons sur l'instance o la méthode « afficher » de la classe Vivipare. Nous établissons le lien entre la méthode et la classe au moyen de l'opérateur de résolution de portée. Cette façon de résoudre le problème, c'est-à-dire par l'utilisation externe de l'opérateur de résolution de portée, n'est cependant pas une bonne solution. Pourquoi ? Parce qu'elle délègue à l'utilisateur de la classe Ovovivipare

notes

résumé

0m 25s



Une des solutions consiste à lever l'ambiguïté en *indiquant explicitement* dans la sous-classe quelle(s) méthode(s) on veut invoquer :

- ajouter à la sous-classe, une *déclaration spéciale* indiquant quel(s) méthode(s)/attribut(s) seront invoqué(s) exactement.

Syntaxe :    `using NomSuperClasse::NomAttributOuMethodeAmbigu;`

Exemple :

```
class Ovovivipare : public Ovipare, public Vivipare {  
public:  
    using Vivipare::afficher;  
    // ...  
};
```

le soin de choisir comment s'affiche un ovovivipare. C'est une responsabilité qui devrait clairement plutôt incomber au concepteur de la classe Ovovivipare. L'idée ici est que, pour une meilleure solution, il faudrait que ce soit la sous-classe (ici, en l'occurrence la sous-classe Ovovivipare) qui indique quelle méthode héritée elle veut voir invoquée sur ses instances. On peut pour cela en C++ ajouter à la sous-classe une déclaration spéciale qui indique quels méthodes ou attributs ambigus, hérités de plus haut, seront invoqués exactement au travers des instances de la sous-classe. La syntaxe pour cette déclaration est la suivante : on utilise le mot réservé `using`, suivi du nom de l'attribut ou de la méthode ambiguë, et on indique, au moyen de l'opérateur de résolution de portée, dans quelle superclasse on va aller chercher cet attribut ou cette méthode. Dans le cas de notre exemple, on pourrait donc, dans la classe Ovovivipare, indiquer par une telle déclaration que c'est la méthode « afficher » de Vivipare qui va être choisie pour afficher un ovovivipare. En clair, lorsque la méthode « afficher » sera appliquée à une instance d'Ovovivipare,

notes

résumé

0m 49s



La meilleure solution consiste à ajouter dans la sous-classe une méthode définissant la bonne interprétation de l'invocation ambiguë.

Exemple :

```
class Ovovivipare: public Ovipare, public Vivipare {
public:
    // ...
    void afficher() const {
        Ovipare::afficher();
        Vivipare::afficher(" mais aussi pour sa partie"
                           " vivipare : ");
    }
    // ...
};
```

on ira rechercher cette méthode dans la classe Vivipare. Alors attention, soyez attentifs à la syntaxe : au moment de cette déclaration, on ne mettra pas de parenthèses ni de type de retour pour la méthode en question. On indique uniquement qu'on veut utiliser la méthode de nom « afficher » de la classe Vivipare. Cette solution est meilleure que la précédente, dans le sens où c'est maintenant la classe Ovovivipare qui décide elle-même de comment s'affichent ses instances. Ce choix n'est plus de la responsabilité de celui qui utilise la classe. On peut rendre les choses encore plus claires, en faisant en sorte que la classe Ovovivipare contienne une méthode dédiée qui indique en clair comment se font les affichages de ses instances. De cette façon, les choses deviennent le plus explicite possible : il n'est pas nécessaire de parcourir la classe pour voir s'il n'existe pas une déclaration using indiquant comment afficher un Ovovivipare, on sait qu'un Ovovivipare a sa propre méthode d'affichage, et donc c'est cette méthode qui va décider de comment se font ces affichages à proprement parler. On pourrait parfaitement vouloir par exemple que, pour afficher un Ovovivipare, on affiche aussi bien ses caractéristiques en tant qu'ovipare que ses caractéristiques en tant que vivipare. La méthode « afficher » d'Ovovivipare, dans ce cas, va invoquer les deux méthodes,

notes

résumé

1m 49s





en utilisant l'opérateur de résolution de portée pour lever l'ambiguïté. Ici donc, le choix d'implémentation qui est fait pour la méthode « afficher » de Ovovivipare est d'invoquer les deux méthodes « afficher » (donc méthodes de même nom héritées de plus haut) : d'abord la méthode « afficher » héritée de la classe Ovipare, puis la méthode « afficher » héritée de la classe Vivipare. Il s'agit là d'une bonne solution, où la sous-classe fournit une méthode explicite qui définit la bonne interprétation des invocations ambiguës. Et ceci conclut cette séquence sur le masquage dans le cadre de l'héritage multiple. de l'héritage multiple.

notes

résumé

3m 1s

