

Support de cours

Cours:

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

Vidéo:

W16-02-heritmultmasquage-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Situation d'ambiguïté. Superclasses ovipare. Héritage multiple. Classe vivipare. Chaîne de caractères. Dérive de la classe ovipare. Sous-classe ovovivipare. Paramètre de type string. Exemple concret. Nouvelle question. Cadre de l'héritage multiple. Problème de masquage. Caractéristiques des instances. Classe ovovivipare. Méthodes.



[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 1)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





Nous avons vu qu'en C++,

notes

résumé

0m 1s



Comme dans le cas de l'héritage simple, une sous-classe peut accéder directement aux attributs et méthodes protégés de ses super-classes.

une classe peut hériter de plusieurs superclasses. C'est ce que l'on appelle l'« héritage multiple ». Que se passe-t-il si plusieurs superclasses offrent à une de leurs sous-classes un membre portant le même nom ? Il y a une situation d'ambiguïté, et c'est ce thème précisément que nous traitons dans la séquence qui suit. Les règles régissant les droits d'accès dans le cadre de l'héritage multiple

notes

résumé

0m 5s



Comme dans le cas de l'héritage simple, une sous-classe peut accéder directement aux attributs et méthodes protégés de ses super-classes.



... et si ces attributs/méthodes *portent le même nom* dans plusieurs super-classes ?

```
class Ovipare {
    // ...
    void afficher() const;
};
```

```
class Vivipare {
    // ...
    void afficher() const;
};
```

```
class Ovovivipare : public Ovipare,
                   public Vivipare
```

```
{ //...
```

```
};
```

```
int main()
```

```
{
```

```
    Ovovivipare o(...);
```

```
    o.afficher();
```

```
    return 0;
```

```
}
```

sont les mêmes que pour l'héritage simple, ce qui signifie qu'une sous-classe peut accéder directement aux membres (c'est-à-dire attributs et méthodes) protégés (ou publics, bien sûr) de ses superclasses. La nouvelle question qui se pose par contre dans le cadre de l'héritage multiple est : « Que se passe-t-il si plusieurs superclasses offrent à une sous-classe des attributs et des méthodes portant le même nom ? » Partons d'un exemple concret. Reprenons pour cela le cas de notre petite hiérarchie de classes zoologiques, où nous avons deux superclasses Ovipare et Vivipare, dont héritait une sous-classe Ovovivipare. Supposons maintenant que, dans chacune des superclasses, soit présente une méthode « afficher », permettant d'afficher par exemple les caractéristiques des instances de chacune de ces classes. Comme la classe Ovovivipare dérive de la classe Ovipare et de la classe Vivipare, elle va donc disposer, par héritage justement, de deux méthodes « afficher » : la méthode « afficher » de la classe Ovipare et la méthode « afficher » de la classe Vivipare ; deux méthodes, donc, ayant strictement le même nom. Supposons maintenant que, ailleurs dans le programme (par exemple dans un programme principal), nous créons une instance d'Ovovivipare, et sur cette instance, invoquons la méthode « afficher ». En clair, va se poser à nous le problème de « quelle méthode est ici invoquée ? » Celle héritée d'Ovipare ou celle héritée de Vivipare ? Le compilateur réagirait de la même façon, se poserait le même problème, et donc refuserait cette tournure.


notes

résumé

0m 25s



Accès direct ambigu (2)

 **Attention !** L'accès `o.afficher` provoquera une erreur à la compilation **même** si la méthode `afficher` n'avait **pas** les mêmes paramètres dans les deux classes `Ovipare` et `Vivipare` !!!

(La raison est qu'en C++, il n'y a **surcharge** que **dans la même portée**. Ici ce n'est pas une problème de surcharge, mais un problème de masquage [résolution de portée].)

```
class Ovipare {
    // ...
    void afficher() const;
};
```

```
class Vivipare {
    // ...
    void afficher(string const& entete) const;
};
```

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

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

    return 0;
}
```

Pour le compilateur, il y a en effet une ambiguïté : il y a un problème de masquage, dans le sens où un identificateur en masque un autre, et le compilateur ne sait pas dans quelle portée il faut résoudre cet appel. Alors attention, nous sommes bel et bien ici face à un problème de masquage : un problème de résolution de portée. En effet, l'invocation de la méthode « afficher » sur une instance d'Ovovivipare provoquera une erreur à la compilation et ce, même si la méthode « afficher » n'avait pas les mêmes paramètres dans les classes Ovipare et Vivipare, ce qui, a priori, peut sembler un peu inattendu. En clair, supposez par exemple que dans la classe Ovipare la méthode « afficher » soit sans paramètres et que dans la classe Vivipare la méthode « afficher » ait un paramètre de type string, alors on pourrait imaginer qu'en invoquant la méthode « afficher » en lui fournissant une chaîne de caractères lève toute ambiguïté : il semblerait clair ici que ce soit cette méthode qui doit être invoquée. Or, l'ambiguïté pourrait être levée ici seulement dans les situations où on aurait de la surcharge. Mais, comme vous avez eu l'occasion de le voir dans les séquences précédentes, en C++ on ne peut surcharger que dans la même portée, et donc ici, concrètement, nous n'avons pas affaire à un problème de surcharge et le compilateur va quand même refuser cette tournure. En effet, la méthode « afficher » sans paramètres est dans la portée de la classe Ovipare, la méthode « afficher » avec un paramètre de type string est dans la portée de la classe Vivipare, ces deux méthodes ne sont pas dans la même portée : on ne peut pas considérer, donc, que cette méthode « afficher » est une surcharge de cette autre méthode « afficher ». Ça n'est donc pas un mécanisme lié à la surcharge qui peut


notes

résumé

1m 49s



Accès direct ambigu (2)

 **Attention !** L'accès `o.afficher` provoquera une erreur à la compilation **même** si la méthode `afficher` n'avait **pas** les mêmes paramètres dans les deux classes `Ovipare` et `Vivipare` !!!

(La raison est qu'en C++, il n'y a **surcharge** que **dans la même portée**. Ici ce n'est pas une problème de surcharge, mais un problème de masquage [résolution de portée].)

```
class Ovipare {
    // ...
    void afficher() const;
};
```

```
class Vivipare {
    // ...
    void afficher(string const& entete) const;
};
```

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

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

    return 0;
}
```

être employé ici pour lever l'ambiguïté ; il faut plutôt résoudre la portée de façon correcte. de façon correcte.

notes

résumé