

Support de cours

Cours:

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

Vidéo:

W15-02-polymvirt-CPP-pt3

Concepts (extraits des sous-titres générés automatiquement) :

Objet de type mammifere. Aspect pointeur. Virtualité de la méthode. Type de la variable. Code de la classe mammifere. Objet de type dauphin. Méthode f. Méthode virtuelle. Résolution dynamique des liens. Second aspect. Exécution de cette instruction. Fin de vie de cet objet. Message propre. Tour des méthodes virtuelles. Aspect virtuel de ces 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/>

Polymorphisme : méthodes virtuelles

(Partie 3)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



virtuelle / non virtuelle : un exemple (2)

Que produit le code suivant ?

```
int main() {
    Mammifere* lui(new Dauphin());
    lui->avancer();
    lui->manger();
    delete lui;
    return 0;
}
```

Un nouveau mammifère est né !
Coui, Couic !
Je nage.

Mammifere::Mammifere()
Dauphin::Dauphin()
Dauphin::avancer()

La méthode « manger » est bel et bien appelée via un pointeur

notes

résumé

0m 1s



```

#include <iostream>
using namespace std;

class Mammifere {
public:
    Mammifere() { cout << "Un nouveau mammifère est né !" << endl; }
    virtual ~Mammifere() { cout << "Un mammifère est en train de mourir :(" << endl; }
    ✗ void manger() const { cout << "Miam... croumf !" << endl; }
    virtual void avancer() const { cout << "Un grand pas pour l'humanité." << endl; }
};

class Dauphin : public Mammifere {
public:
    Dauphin () { cout << "Coui, Couic !" << endl; }
    ~Dauphin() { cout << "Flipper, c'est fini..." << endl; }
    void manger() const { cout << "Sglups, un poisson." << endl; }
    void avancer() const { cout << "Je nage." << endl; }
};

```

mais il n'y a là qu'un seul des deux ingrédients qui est présent pour la résolution dynamique des liens, l'aspect pointeur, mais le second aspect, la virtualité de la méthode, n'est pas présent, ce qui signifie que l'on ne peut pas avoir une résolution dynamique des liens. Dans ce cas, c'est le type de la variable qui prime. « lui » est un pointeur sur Mammifère, donc le contenu pointé est perçu comme un objet de type Mammifere.

notes


résumé

0m 5s



Que produit le code suivant ?

```
int main() {  
    Mammifere* lui(new Dauphin());  
    lui->avancer();  
    lui->manger();  
    delete lui;  
    return 0;  
}
```

Un nouveau mammifère est né !
Coui, Couic !
Je nage.
Miam...  croumf !

Mammifere::Mammifere()
Dauphin::Dauphin()
Dauphin::avancer()
Mammifere::manger()

La résolution statique des liens a donc lieu dans ce cas et c'est la méthode manger de Mammifere qui va être appelée ici. L'affichage après l'exécution de cette instruction sera donc :

notes

résumé

0m 36s



```
#include <iostream>
using namespace std;

class Mammifere {
public:
    Mammifere() { cout << "Un nouveau mammifère est né !" << endl; }
    virtual ~Mammifere() { cout << "Un mammifère est en train de mourir :(" << endl; }
    void manger() const { cout << "Miam... croumf !" << endl; }
    virtual void avancer() const { cout << "Un grand pas pour l'humanité." << endl; }
};

class Dauphin : public Mammifere {
public:
    Dauphin () { cout << "Coui, Couic !" << endl; }
    ~Dauphin() { cout << "Flipper, c'est fini..." << endl; }
    void manger() const { cout << "Sglups, un poisson." << endl; }
    void avancer() const { cout << "Je nage." << endl; }
};
```

«Miam... croumf ! », qui est le message propre à la méthode manger du Mammifere. Passons à la ligne suivante. « delete lui » signifie que l'on désalloue l'emplacement mémoire alloué dynamiquement à ce stade du programme. Ceci signifie en clair, que l'on récupère l'emplacement mémoire associé à l'objet de type Dauphin précédemment alloué. « delete lui » cause donc la fin de vie de cet objet et puisqu'il y a fin de vie, il y a appel à un destructeur. Mais de quel destructeur exactement s'agit-il ? Si on examine le code de la classe Mammifere, on peut observer que le destructeur y a été déclaré comme virtuel. Ceci signifie que cette méthode pour peu qu'elle soit appliquée via un pointeur

notes

résumé

0m 49s



Que produit le code suivant ?

```
int main() {  
    Mammifere* lui(new Dauphin());  
    lui->avancer();  
    lui->manger();  
    delete lui;  
    return 0;  
}
```

Un nouveau mammifère est né !
Coui, Couic !
Je nage.
Miam... croumf !

Mammifere::Mammifere()
Dauphin::Dauphin()
Dauphin::avancer()
Mammifere::manger()

par exemple, va faire l'objet de la résolution dynamique des liens.

notes

résumé

1m 37s



```
#include <iostream>
using namespace std;

class Mammifere {
public:
    Mammifere() { cout << "Un nouveau mammifère est né !" << endl; }
    virtual ~Mammifere() { cout << "Un mammifère est en train de mourir :(" << endl; }
    void manger() const { cout << "Miam... croumf !" << endl; }
    virtual void avancer() const { cout << "Un grand pas pour l'humanité." << endl; }
};

class Dauphin : public Mammifere {
public:
    Dauphin ()          { cout << "Coui, Couic !" << endl; }
    ~Dauphin()          { cout << "Flipper, c'est fini..." << endl; }
    void manger() const { cout << "Sglups, un poisson." << endl; }
    void avancer() const { cout << "Je nage." << endl; }
};
```

Le destructeur est bel et bien invoqué via un pointeur, ce qui signifie que la résolution dynamique des liens a lieu pour la destruction aussi. L'objet va être perçu en fonction de sa nature réelle, c'est-à-dire en tant que Dauphin, et c'est le destructeur du Dauphin qui va être invoqué, ce qui cause l'affichage suivant. Nous avons vu dans une séquence précédente que l'ordre d'invocation des destructeurs est l'inverse de l'ordre de la construction, donc pour construire un objet de type Dauphin, il y a eu appel au constructeur de la super-classe Mammifere puis à celui de Dauphin. Pour la destruction, c'est l'inverse, on va avoir un appel au destructeur de Dauphin, suivi d'un appel au destructeur de Mammifere.

notes

résumé

1m 41s



Que produit le code suivant ?

```
int main() {  
    Mammifere* lui(new Dauphin());  
    lui->avancer();  
    lui->manger();  
    delete lui;  
    return 0;  
}
```

Imaginons maintenant que le destructeur de la classe Mammifere n'ait pas été déclaré comme virtuel.

notes

résumé

2m 25s



```
#include <iostream>
using namespace std;

class Mammifere {
public:
    Mammifere() { cout << "Un nouveau mammifère est né !" << endl; }
    virtual Mammifere() { cout << "Un mammifère est en train de mourir :(" << endl; }
    void manger() const { cout << "Miam... croumf !" << endl; }
    virtual void avancer() const { cout << "Un grand pas pour l'humanité." << endl; }
};

class Dauphin : public Mammifere {
public:
    Dauphin () { cout << "Coui, Couic !" << endl; }
    ~Dauphin() { cout << "Flipper, c'est fini..." << endl; }
    void manger() const { cout << "Sglups, un poisson." << endl; }
    void avancer() const { cout << "Je nage." << endl; }
};
```

Puisque le destructeur n'est pas déclaré comme virtuel dans la classe Mammifere, l'appel qui y est fait ici ne peut pas être résolu dynamiquement. Ceci signifie que c'est le type de la variable qui va primer pour le choix de la méthode destructeur,

notes

résumé

2m 31s



Et si le destructeur de `Mammifere` n'avait pas été virtuel ?

```
Un nouveau mammifère est né !  
Coui, Couic !  
Je nage.  
Miam... croumf !  
Un mammifère est en train de mourir :(
```

et donc c'est ce destructeur qui est invoqué et non plus celui-ci.

notes

résumé

2m 46s



En résumé :

Lorsqu'une *méthode virtuelle* est invoquée à partir d'une *référence* ou d'un *pointeur* vers une instance, c'est la méthode du type réel de l'instance qui sera exécutée.



Attention !

- ▶ Il est conseillé de toujours définir les **destructeurs** comme **virtuels**
- ▶ Un constructeur ne peut pas **être virtuel**
- ▶ L'aspect virtuel des **méthodes** est **ignoré** dans les constructeurs (avancé ! Détails à suivre.)

Concrètement donc, si le destructeur de Mammifere n'avait pas été virtuel dans notre exemple, eh bien seul le destructeur de Mammifere aurait été invoqué, le destructeur du Dauphin ne l'est pas. Ce qui signifie que l'objet alloué dynamiquement dans le programme qui est de type Dauphin, n'est que partiellement détruit. Seul la part du Mammifere dans le Dauphin aurait été détruite ici. Si l'objet de type Dauphin dynamiquement alloué ici, avait requis un certain nombre de ressources, et bien ces ressources n'auraient donc pas pu être libérées proprement. Pour résumer, lorsqu'une méthode virtuelle s'applique via une référence ou un pointeur, c'est-à-dire lorsque ces deux ingrédients sont réunis, alors le choix de la méthode se fait en fonction du type réel de l'instance. On a ce qu'on appelle la résolution dynamique des liens. Mais attention, quelques remarques importantes par rapport à la virtualité dans le cadre de la construction-destruction. Comme nous l'avons vu dans l'exemple des Mammifere et des Dauphin, pour éviter qu'une destruction d'objet ne soit que partielle, il est conseillé de toujours déclarer les destructeurs comme virtuels. Par contre, comme un constructeur a toujours pour vocation d'initialiser l'instance courante, eh bien il ne peut pas être virtuel. S'il appelle à son tour des méthodes virtuelles, alors l'aspect virtuel de ces méthodes sera tout simplement ignoré. Il s'agit là d'un aspect avancé, voyons ce qu'il signifie sur un exemple concret.

notes

résumé

2m 51s



Méthodes virtuelles et constructeurs

L'aspect polymorphique est ignoré dans les constructeurs ; c'est la méthode de la classe courante qui est appelée.

```
#include <iostream>
using namespace std;

class A {
public:
    A() { f(); }
    virtual void f() const { cout << "A::f()" << endl; }
};

class B : public A {
public:
    virtual void f() const { cout << "B::f()" << endl; }
};

int main()
{
    A a;
    B b;
    A* pa(&b);
    pa->f();
    return 0;
}
```

Handwritten annotations in red:

- A red 'X' is drawn over the `virtual` keyword in the `A::f()` definition.
- Red arrows point from the `A a;` and `B b;` lines to the handwritten text `A::f()` and `A::f()` respectively.
- A red circle is drawn around the `pa->f();` line, with a red arrow pointing to the handwritten text `A::f()`.

Voici donc un exemple un petit peu académique mais pour expliquer ce qui se passe dans ce cas de figure, où nous avons une super-classe grand A, dont hérite une sous-classe grand B. Les deux classes comportent une méthode f qui se trouve être virtuelle, et qui donc est redéfinie dans la sous-classe grand B. Lorsque nous exécutons cette ligne de code, il y a appel au constructeur par défaut de la classe grand A, lequel fait appel à la méthode f. La méthode f s'applique à l'objet this qui est de type grand A, et donc on va naturellement invoquer la méthode f de la classe grand A, ce qui aura pour conséquence d'afficher ceci : Pour la seconde ligne, nous avons construction d'un objet de type B, il n'y a pas de constructeur explicite dans la classe B, c'est donc le constructeur par défaut qui va être invoqué, lequel va aussi invoquer le constructeur par défaut de la super-classe. La méthode f s'applique donc désormais à un objet this de type grand B, cependant l'aspect virtuel de la méthode est complètement ignoré dans un constructeur. Donc cet aspect n'est pas pris en compte, et c'est toujours la méthode de la classe courante qui est appelée. Par conséquent, ici également lors de l'exécution de cette seconde ligne nous aurons aussi l'affichage suivant. La ligne suivante de code déclare une variable pa de type pointeur sur un A, et l'initialise avec l'adresse de l'objet b.

notes

résumé

4m 13s





La méthode `f` est ensuite invoquée sur l'objet pointé par le pointeur `pa`, qui est un objet de type `B`. Ici, les deux ingrédients nécessaires à la résolution dynamique des liens sont réunis. Nous avons une méthode `f` virtuelle qui est invoquée via un pointeur. La méthode `f` va donc être choisie en fonction de la nature réelle de l'instance pointée, et donc c'est la méthode `f` de la classe `B` qui va être invoquée. C'est donc ici un message différent qui va s'afficher lors de l'exécution de cette dernière ligne de code. Ici, nous avons bel et bien eu un comportement polymorphique de la méthode `f` car nous l'avons invoquée en dehors d'un constructeur. Par contre ici, l'aspect polymorphique a été complètement ignoré. Et ceci termine notre séquence sur les méthodes virtuelles qui, en C++, sont la base de la mise en oeuvre du polymorphisme. du polymorphisme.

notes

résumé

5m 49s

