

Support de cours

Cours:

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

Vidéo:

### W15-04-polymabstr-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

**Méthode virtuelle pure. Figures géométriques fermées. Première solution. Méthodes virtuelles pures. Fameux jeux. Tableau dynamique de personnages. Tableau de pointeurs. Notion de classe abstraite. Figures fermées spécifiques. Bon complément de l'abstraction. Niveau de cette classe. Niveau général d'un personnage. Méthode surface. Séquence vidéo. Mauvaise idée.**



[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/>



# Classes abstraites

## (Partie 1)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s







Dans cette séquence vidéo, nous allons voir comment

notes

---

---

---

---

---

---

---

---

---

---

résumé

0m 1s



---

---

---

---

---

---

---

---

---

---



Au sommet d'une hiérarchie de classe, il n'est pas toujours possible de :

- ▶ donner une définition générale de certaines méthodes, *compatibles avec toutes les sous-classes*,
- ▶ ...même si l'on sait que toutes ces sous-classes vont effectivement implémenter ces méthodes



le polymorphisme est un très bon complément de l'abstraction, et nous permet donc de mieux spécifier les classes au niveau le plus abstrait. Nous allons justement introduire la notion de classe abstraite et de méthode virtuelle pure.

notes

---

---

---

---

---

---

---

---

---

---

résumé

---

---

---

---

---

---

---

---

0m 5s





## Besoin de méthodes virtuelles pures : exemple

Exemple :

```
class FigureFermee {
    // ...

    // difficile à définir à ce niveau !..
    virtual double surface(...) const { ??? }

    // ...pourtant la méthode suivante en aurait besoin !
    double volume(double hauteur) const {
        return hauteur * surface();
    }
};
```



Figure Fermée  
↑  
Cercle

Définir `surface` de façon arbitraire sachant qu'on va la redéfinir plus tard n'est pas une bonne solution (source d'erreurs) !

☞ **Solution** : déclarer la méthode `surface` comme **virtuelle pure**

Mais commençons par présenter le problème auquel répondent ces méthodes virtuelles pures. Au niveau le plus élevé d'une hiérarchie, on ne sait pas forcément définir une méthode, que l'on sait pourtant qui va exister pour toutes les sous-classes. Par exemple, si l'on imagine avoir une classe très générale qui représente des figures géométriques fermées, quelconques, à ce moment là, au niveau de cette classe, il va être assez difficile de définir comment calculer la surface d'une figure géométrique quelconque, c'est certainement quelque chose de difficile à définir, on ne saurait pas bien le faire. Pourtant on sait que toute figure fermée, -- par exemple un cercle est une figure fermée -- donc on sait qu'au niveau de toutes les figures fermées spécifiques, on devrait quand même pouvoir définir la méthode `surface`. Donc, on imagine que toutes les figures fermées vont avoir une méthode `surface`, même si l'on ne sait pas bien la définir au niveau le plus abstrait. Pourtant, pour pousser le problème un petit peu plus loin, on peut même imaginer que cette méthode `surface` soit utilisée au niveau le plus élevé, par exemple pour calculer un volume engendré sur une certaine hauteur, par cette surface.

notes

résumé

0m 21s





## Besoin de méthodes virtuelles pures : autre exemple (1), EPFL

Plusieurs équipes collaborent à la création d'un jeu.

Une équipe prend en charge les classes de base suivantes :

- ▶ **Jeu:**
  - ▶ Classe pour gérer le jeu
  - ▶ Se contente ici d'afficher les personnages
- ▶ **Personnage:**
  - ▶ Classe de base pour les personnages

Une autre équipe ajoutera des sous-classes de personnages spécifiques.

A ce moment là, ce volume engendré suivant cette hauteur par cette figure géométrique, serait défini par le produit de la hauteur multiplié la surface de la figure fermée. Donc on pourrait même faire appel à cette méthode, même si on ne sait pas très bien comment la définir. Au plus haut niveau, elle serait définie pour tous les exemples concrets de figures fermées. La bonne façon de faire ceci, une méthode que l'on sait qui doit exister, mais que l'on ne sait pas forcément bien définir au niveau d'une super classe, cette façon de faire est d'introduire la méthode comme une méthode virtuelle pure. Illustrons encore ce concept de méthode virtuelle pure sur un autre exemple et revenons donc à nos fameux jeux avec des personnages,

notes

résumé

1m 37s







où nous avons donc des guerriers qui sont des personnages, des magiciens qui sont des personnages, des voleurs.. La classe « Jeu » contiendrait bien sûr plusieurs personnages, on a décidé de les mettre ici dans un tableau dynamique de personnages. Et pour avoir du polymorphisme, nous avons donc un tableau de pointeurs sur des personnages,

notes

résumé

2m 25s





## Besoin de méthodes virtuelles pures : autre exemple (2, EPFL

La classe `Jeu` développée par la première équipe :  
▮ gère un tableau de personnages et les affiche

```
class Jeu {  
public:  
    // ...  
    void afficher() const {  
        for (auto un_perso : persos) {  
            un_perso->afficher();  
            // Tous les personnages doivent pouvoir s'afficher !...  
            // ...mais comment???  
        }  
    }  
    // ...  
private:  
    vector<Personnage*> persos;  
};
```

cet aspect précis, tableau de pointeurs sur une classe, sera repris très en détails dans une future vidéo qui s'appelle collections hétérogènes. Ici, on a simplement notre tableau de personnages, et imaginons que dans le jeu, on souhaite afficher l'ensemble des personnages. Donc on ferait une boucle qui parcourt l'ensemble des personnages, qui parcourt ce tableau « persos »,

notes

résumé

2m 46s







et pour chacun des personnages, on afficherait ce personnage. Cette syntaxe ici, « un\_perso », c'est la variable qui parcourt le tableau de personnages, c'est donc un pointeur sur un personnage. Et donc, comme on a un pointeur, on utilise cette syntaxe ici, pour dire la méthode « afficher » de l'objet pointée par « un\_perso ». Donc ici, on afficherait bien chacun des personnages « un\_perso » dans l'ensemble des personnages stockés, ici dans l'attribut « persos » avec un 's', du jeu. Le problème, c'est comment afficher un personnage générique.

notes

résumé

3m 13s





## Besoin de méthodes virtuelles pures : autre exemple (3,

Si l'on ne met aucune méthode `afficher` dans `Personnage`, la classe `Jeu` ne compile pas :

```
class Jeu {  
    // ...  
    void afficher() const {  
        for (auto un_perso : persos) {  
            un_perso->afficher(); // ERREUR !  
        }  
    }  
    // ...  
};
```

On **doit** donc mettre une méthode `afficher` dans la classe `Personnage`...

De plus, on aimerait :

- ▶ imposer aux sous-classes (`Guerrier`, ...) d'avoir leur méthode `afficher` spécifique
- ▶ que cette méthode spécifique à la sous-classe soit exécutée (polymorphisme *donc* méthode **virtuelle**)

On ne sait pas afficher un personnage générique, on sait certainement afficher un guerrier, un magicien, un voleur, donc chacune des sous-classes spécifiques sait comment s'afficher, mais par contre, au niveau général d'un personnage, on ne sait pas forcément l'afficher, mais on sait qu'on en a besoin ici au niveau très général du jeu, de pouvoir afficher les personnages. Donc comment faire ? Bien sûr, si on ne fait rien au niveau de la classe « `Personnage` », si on ne définit pas de méthode « `afficher` » dans « `Personnage` », alors le code « `afficher` » de jeu ne compilerait pas puisque ici on appelle bien la méthode « `afficher` » d'un personnage, et le compilateur ne saurait pas quoi faire, il nous donnerait donc une erreur. On doit donc pour pouvoir écrire ceci au niveau global du jeu, on doit mettre une méthode « `afficher` » dans la classe « `Personnage` ». De plus, on aimerait imposer à chacune des sous-classes, les « `Guerriers` », « `Magiciens` », de s'afficher avec leur propre méthode « `afficher` », d'avoir une méthode spécifique, et que ce soit cette méthode spécifique

notes

résumé

3m 48s





## Besoin de méthodes virtuelles pures : autre exemple (5, EPFL)

Première « solution » :

ajouter une méthode quelconque définie arbitrairement :

```
class Personnage {  
    // ...  
    // On n'affiche rien : corps de la méthode vide  
    virtual void afficher() const { }  
    // ...  
};
```

C'est une **très mauvaise idée**

- ❌ Mauvais modèle de la réalité  
(affichage incorrect si une sous-classe ne redéfinit pas la méthode : personnages fantômes !)
- ❌ Cette solution n'impose pas que la méthode **afficher** soit redéfinie

qui soit appelée dans l'appel général, ici au niveau du jeu. On veut effectivement forcer d'avoir une méthode spécifique au niveau des classes, et que cette méthode soit polymorphique. Il faut donc que cette méthode soit virtuelle et existe au niveau des personnages. Mais comment faire si on ne sait pas, supposons qu'on ne sache pas afficher un personnage générique. De plus, comment imposer que cette méthode « afficher » soit redéfinie, forcer la redéfinition dans les sous-classes ? Une première solution pour le premier problème, comment définir l'affichage pour un personnage générique, serait d'avoir une méthode quelconque, définie arbitrairement, par exemple, supposons qu'ici, arbitrairement, on définisse que l'affichage générique d'un personnage, c'est de ne rien afficher. Cette solution est une très mauvaise idée,

notes

résumé

4m 49s







parce que d'abord c'est un mauvais modèle de la réalité, les personnages ne s'affichent pas comme du vide, donc ça ne correspond à rien, l'affichage est incorrect, et en plus l'affichage serait incorrect si une sous-classe oubliait de redéfinir la méthode, on aurait des personnages qui ne s'affichent pas, des personnages fantôme, ce qui est peut-être embêtant pour le jeu. De plus, cette solution ne répond pas au deuxième problème, elle n'oblige pas les sous-classes à redéfinir leur propre méthode « afficher ». La seule bonne solution est de signaler que la méthode « afficher » doit exister et doit être définie dans chacune des sous-classes, c'est ce qu'on appelle une méthode virtuelle pure. Voilà donc un deuxième exemple, qui j'espère, vous a fait comprendre le besoin d'avoir ces méthodes virtuelles pures dans la définition générale abstraite de super classes. de super classes.

## notes

## résumé

5m 37s

