

Support de cours

Cours:

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

Vidéo:

### W15-02-polymvirt-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

**Méthodes virtuelles. Résolution dynamique des liens. Utilisation de méthodes virtuelles. Objet de type sous-classe. Hiérarchie de classes. Séquence vidéo précédente. Biais du passage des arguments. Super-classe. Question de quelle méthode. Déclaration de la virtualité d'une méthode. Première ligne du programme. Type de la variable. Explication de ce transparent. Instance d'origine. Sous-classe dauphin.**



[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 1)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





Dans la séquence vidéo précédente, nous avons vu

notes

---

---

---

---

---

---

---

---

---

---

résumé

0m 1s



---

---

---

---

---

---

---

---

---

---

Il pourrait dans certains cas sembler plus naturel de choisir la méthode correspondant à la *nature réelle de l'instance*.

Dans ces cas, il faut permettre la **résolution dynamique des liens** :

- Le *choix de la méthode* à exécuter se fait à *l'exécution*, en fonction de la *nature réelle des instances*

2 ingrédients pour cela :

**références/pointeurs**

et

**méthodes virtuelles**

que le polymorphisme était une notion centrale en programmation orientée objet. Nous avons vu que pour mettre en oeuvre une solution polymorphique, il fallait faire appel à ce qu'on appelle la résolution dynamique des liens. L'objectif de cette séquence est de vous montrer comment on met en oeuvre en C++ la résolution dynamique des liens en utilisant les méthodes virtuelles.

notes

résumé

0m 5s



Une instance de sous-classe **B** est substituable à une instance de super-classe **A**.  
Que se passe-t-il lorsque **B** redéfinit une méthode de **A** ?

```
class Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};
```

```
void faire_rencontrer(
    Personnage const& un,
    Personnage const& autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;

    faire_rencontrer(g, v);
}
```

Plus précisément, nous avons conclu la séquence précédente en indiquant qu'en C++ pour que la résolution dynamique des liens puisse avoir lieu, il faut réunir deux ingrédients. À savoir l'utilisation de méthodes virtuelles sur des références et des pointeurs. C'est à l'explication de ce transparent qu'est dédiée cette séquence. Pour rappel, nous étions partis d'un exemple où nous avons une hiérarchie de classes dans laquelle une sous-classe, la sous-classe Guerrier, redéfinit une méthode déjà présente dans la super-classe. Un objet de type sous-classe était, par le biais du passage des arguments, mis dans une variable de type super-classe. Nous nous étions alors posé la question de quelle méthode rencontrer était appelée ici. La méthode rencontrer de la super-classe ou alors celle de la sous-classe.

notes

résumé

0m 27s



## Déclaration des méthodes virtuelles

- ▶ En C++, on indique au compilateur qu'une méthode peut faire l'objet d'une résolution dynamique des liens en la déclarant comme **virtuelle** (mot clé `virtual`)
- ▶ Cette déclaration doit se faire dans la classe la plus **générale** qui admet cette méthode (c'est-à-dire lors du *prototypage d'origine*)
- ▶ Les redéfinitions éventuelles dans les sous-classes seront aussi considérées comme *virtuelles par transitivité*.

### Syntaxe :

`virtual Type nom_fonction(liste de paramètres) [const];`

### Exemple :

```
class Personnage {
    // ...
    virtual void rencontrer(Personnage& autre) const
    { cout << "Bonjour !" << endl; }
};
```

Et nous avons vu qu'en C++ il n'en était rien de particulier, c'est la résolution statique des liens qui a lieu, c'est-à-dire que c'est le type de la variable sur laquelle on appelle la méthode qui prime et qui va conditionner le choix de la méthode. En C++, si l'on veut que ce soit la résolution dynamique des liens qui ait lieu, c'est-à-dire la résolution par le biais de laquelle c'est le contenu de la variable plutôt que son type qui décide de la méthode à utiliser, alors il faut explicitement déclarer dans le programme que la méthode doit être résolue dynamiquement. Et ceci se fait en déclarant cette méthode comme virtuelle. Pour que la résolution dynamique des liens ait bel et bien lieu, il faudra donc déclarer la méthode comme virtuelle mais il faudra aussi qu'elle puisse s'appliquer via une référence ou un pointeur. C'est bien le cas ici, « un » est une référence vers l'objet passé en paramètre. En C++ on doit donc indiquer explicitement au compilateur qu'une méthode peut faire l'objet d'une résolution dynamique des liens. Et ce, en la déclarant comme virtuelle. La déclaration de la virtualité d'une méthode doit se faire dans la classe

### notes

### résumé

1m 13s



## Retour sur l'exemple (1)

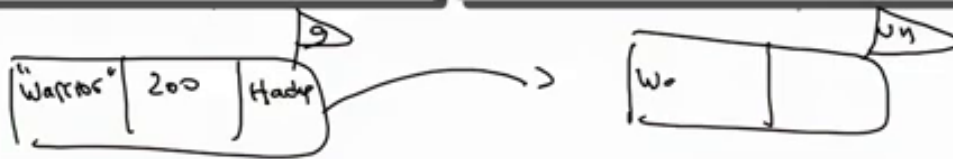
```
class Personnage {
public:
    // ...
    virtual void rencontrer(
        Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};
```

```
void faire_rencontrer(Personnage un,
    Personnage autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;

    faire_rencontrer(g, v);
}
```



la plus générale qui admet cette méthode, c'est-à-dire le plus en haut de la hiérarchie dans le prototypage d'origine. Donc dans le cas de notre exemple, dans la classe Personnage. Et la virtualité d'une méthode se déclare en C++ simplement en précédant le prototype de la méthode par le mot-clé virtual. Il faut noter que toute redéfinition dans les sous-classes d'une méthode déjà déclarée comme virtuelle dans une super-classe sera aussi considérée comme virtuelle par transitivité même si l'on ne met pas le mot-clé virtual devant son prototype. Donc concrètement, si j'ai une super-classe grand A contenant une méthode particulière m et si j'ai une sous-classe grand B qui redéfinit cette méthode. Si dans la super-classe grand A la méthode m a été déclarée comme virtuelle, alors elle l'est aussi dans la sous-classe même si le mot-clé n'apparaît pas explicitement. Si on revient à notre exemple de départ, si l'on veut donc que notre méthode fasse l'objet d'une résolution dynamique des liens, alors il faut déclarer dans la super-classe la méthode comme étant virtuelle. Mais est-ce vraiment là notre exemple de départ ? Si vous avez été attentif, vous aurez peut-être remarqué que l'exemple que vous avez ici sous les yeux est une variante de l'exemple présenté à l'origine. Une variante dans laquelle le passage des arguments de la fonction « faire\_rencontrer » est un passage d'argument par valeur. Nous avons délibérément fait cette modification pour vous montrer l'incidence qu'a le fait de ne pas utiliser une référence ou un pointeur sur la résolution dynamique des liens. Le problème est que lorsqu'on fait un passage par valeur, l'objet Guerrier va être copié dans une variable de type Personnage. La méthode « rencontrer » ne va donc pas s'appliquer à l'instance d'origine, mais à une copie de l'instance dans une variable de type Personnage. En clair, si nous avons dans la variable g

### notes

### résumé

2m 25s



## Retour sur l'exemple (1)

```

class Personnage {
public:
    // ...
    virtual void rencontrer(
        Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};

```

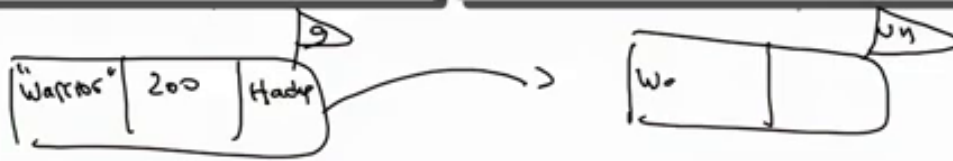
```

void faire_rencontrer(Personnage un,
    Personnage autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;

    faire_rencontrer(g, v);
}

```



un objet de type Guerrier, une fois copié dans la variable « un », qui est de type Personnage,

notes

résumé



Il pourrait dans certains cas sembler plus naturel de choisir la méthode correspondant à la *nature réelle de l'instance*.

Dans ces cas, il faut permettre la **résolution dynamique des liens** :

- Le *choix de la méthode* à exécuter se fait à *l'exécution*, en fonction de la *nature réelle des instances*

2 ingrédients pour cela :

**références/pointeurs**

et

**méthodes virtuelles**

il n'est plus perçu que comme un personnage. Personnage qui n'a pas d'attributs pour une arme par exemple. Ce qui est donc contenu dans la variable « un » est bel et bien un objet de type Personnage. Par conséquent, même si la méthode « rencontrer » est virtuelle, donc peut faire l'objet d'une résolution dynamique des liens, et bien puisque l'objet contenu est un objet de type Personnage, c'est toujours la méthode « rencontrer » de Personnage qui va s'appliquer ici.

notes

résumé

4m 49s



## Retour sur l'exemple (1)

```

class Personnage {
public:
    // ...
    virtual void rencontrer(
        Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};

```

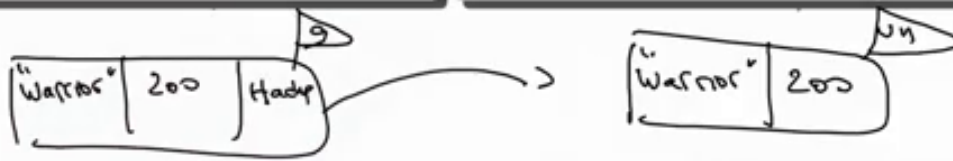
```

void faire_rencontrer(Personnage un,
    Personnage autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;

    faire_rencontrer(g, v);
}

```



Comme évoqué dans la séquence précédente, pour que la résolution dynamique des liens puisse avoir lieu, il faut vraiment que deux ingrédients soient présents, le premier est la virtualité des méthodes, ce que nous venons de voir, et il faut également que ces méthodes virtuelles puissent s'appliquer aux instances réelles via des références ou des pointeurs.

## notes

## résumé

5m 29s



## Retour sur l'exemple (2)



**Attention !** Il faut passer **un par référence** pour que la fonction **faire\_rencontrer** agisse sur l'instance d'origine !

```
class Personnage {
public:
    // ...
    virtual void rencontrer(
        Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};
```

```
void faire_rencontrer(
    Personnage const& un,
    Personnage const& autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;

    faire_rencontrer(g, v);
}
```

Cette fois tout fonctionne comme on veut !

Si ici nous faisons le choix d'un passage par référence, au moment du passage des arguments nous n'aurons pas de copie dans un objet de type `Personnage`, nous indiquons simplement que `1` est un autre nom pour la variable `g`. Nous permettons de ce fait à la méthode virtuelle d'opérer sur l'instance réelle. En permettant à une méthode virtuelle d'opérer sur l'instance réelle, nous obtenons alors l'effet voulu, à savoir une résolution dynamique des liens. Comme dit précédemment, c'est en combinant l'utilisation des références et la virtualité des méthodes que l'on obtient le résultat voulu. Ici bien sûr, on pourrait se contenter d'un passage par référence constante puisque la fonction « `faire_rencontrer` » n'a pas pour but de modifier son paramètre « `un` ».

notes

résumé

5m 48s



```
#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; }
};
```

En clair, voilà comment devrait être le code au final pour que l'on obtienne la résolution dynamique des liens dans le cadre de cet exemple. Pour permettre donc à une méthode virtuelle de travailler sur l'instance réelle, il est possible donc d'utiliser les références. On peut aussi utiliser les pointeurs comme va le montrer l'exemple qui suit. Et cette fois pour changer, un exemple zoologique où on a une sous-classe Dauphin héritant d'une super-classe Mammifere. Chacune de ces classes

## notes

## résumé

6m 44s



Que produit le code suivant ?

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

dispose d'un constructeur par défaut explicite et d'un destructeur. Les « Mammifere » de façon générale ont une méthode par défaut pour manger. Les « Dauphin » ont une méthode spécifique pour cette même action. Et de même pour les méthodes qui permettent de faire avancer le Mammifere ou le Dauphin.

notes

résumé

7m 13s



```
#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; }
};
```

Voici maintenant un petit programme principal qui utilise la hiérarchie de classe que vous avez vue précédemment. Étudions pas à pas son déroulement pour voir ce qu'il va produire comme affichage. La première ligne du programme déclare une variable qui s'appelle « lui », qui est de type pointeur sur Mammifere. Cette variable reçoit l'adresse d'un objet de type Dauphin alloué dynamiquement. L'objet de type Dauphin alloué dynamiquement est construit au moyen du constructeur par défaut de la classe Dauphin. La variable lui contient donc l'adresse d'un objet de type Dauphin. Le constructeur par défaut de la classe Dauphin a été utilisé ici pour construire cet objet. Il s'agit donc de ce constructeur par défaut,

#### notes

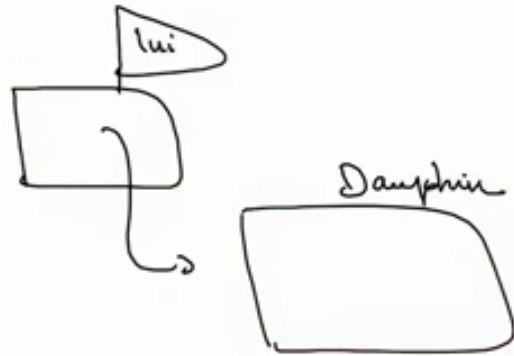
#### résumé

7m 32s



Que produit le code suivant ?

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



tel que défini dans la sous-classe Dauphin. Or le constructeur d'une sous-classe invoque toujours le constructeur de la super-classe. S'il ne le fait pas explicitement, c'est qu'il y a eu appel au constructeur par défaut de la super-classe. Ce qui veut dire concrètement qu'avant que le message « Coui, Couic ! » s'affiche, s'affiche, il y a le message « Un nouveau mammifère est né ! ».

notes

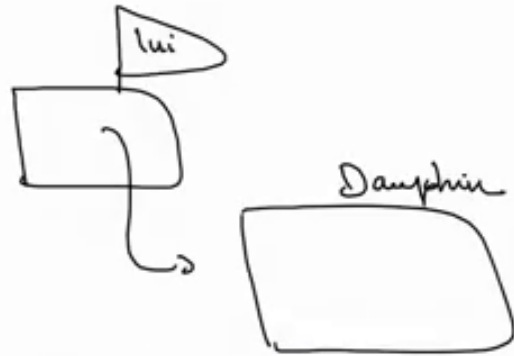
résumé

8m 25s



Que produit le code suivant ?

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



L'exécution de cette ligne de code va donc poser le premier affichage suivant : Un nouveau mammifère est né ! Pourquoi ? Parce que le constructeur par défaut de Dauphin a implicitement appelé le constructeur par défaut de Mammifere. Le constructeur de la super-classe Mammifere a été invoqué par celui de la classe Dauphin. Lorsqu'il a terminé son exécution, le corps du constructeur de Dauphin peut s'exécuter, ce qui produit l'affichage suivant. Seconde ligne. On applique à l'objet pointé par le pointeur lui, c'est-à-dire cet objet, la méthode avancer. la méthode avancer.

notes

résumé

8m 51s

