

Support de cours

Cours:

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

Vidéo:

W14-01-heritageintro-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Exemple des guerriers. Personnages d'un jeu. Niveau des méthodes. Méthodes communes. Notions d'encapsulation. Durée de vie. Ensemble des classes. Niveau d'énergie. Notion d'héritage. Séquence vidéo. Exemple des points de vie. Classe de type. Super-classes. Gros problèmes de maintenance. Personnage générique.



[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 : concepts

(Partie 1)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





Après les notions d'encapsulation et d'abstraction

notes

résumé

0m 1s



Exemple : classes pour les personnages

class Guerrier

```
string nom
int energie
int duree_vie
Arme arme
rencontrer(Personnage&)
```

class Voleur

```
string nom
int energie
int duree_vie
rencontrer(Personnage&)
voler(Personnage&)
```

class Magicien

```
string nom
int energie
int duree_vie
Baguette baguette
rencontrer(Personnage&)
```

class Sorcier

```
string nom
int energie
int duree_vie
Baguette baguette
Baton baton
rencontrer(Personnage&)
```

On pourrait également avoir des sorciers comme des sortes de magiciens qui en plus d'avoir une baguette auraient aussi un bâton, tout ceci, dupliquerait beaucoup de code, ça nous ferait perdre beaucoup de temps

notes

résumé

1m 13s



Exemple : classes pour les personnages

class Guerrier

```
string nom
int energie
int duree_vie
Arme arme
rencontrer(Personnage&)
```

class Voleur

```
string nom
int energie
int duree_vie
rencontrer(Personnage&)
voler(Personnage&)
```

class Magicien

```
string nom
int energie
int duree_vie
Baguette baguette
rencontrer(Personnage&)
```

class Sorcier

```
string nom
int energie
int duree_vie
Baguette baguette
Baton baton
rencontrer(Personnage&)
```

d'écrire tout cela et surtout

notes

résumé

1m 25s





poserait de gros problèmes de maintenance imaginons que l'on ne veuille plus que les personnages aient une durée de vie mais qu'ils aient par exemple des points de vie ou que l'énergie ne soit plus un « int » mais un « double » il faudrait alors corriger l'ensemble des classes que nous avons développé, c'est certainement pas une bonne façon de faire.

notes

résumé

1m 26s



Exemple : classes pour les personnages

class Guerrier

string nom
~~int~~ energie
~~int~~ duree_vie
 Arme arme
 rencontrer(Personnage&)

class Voleur

string nom
~~int~~ energie
~~int~~ duree_vie
 rencontrer(Personnage&)
 voler(Personnage&)

class Magicien

string nom
~~int~~ energie
~~int~~ duree_vie
 Baguette baguette
 rencontrer(Personnage&)

class Sorcier

string nom
~~int~~ energie
~~int~~ duree_vie
 Baguette baguette
 Baton baton
 rencontrer(Personnage&)



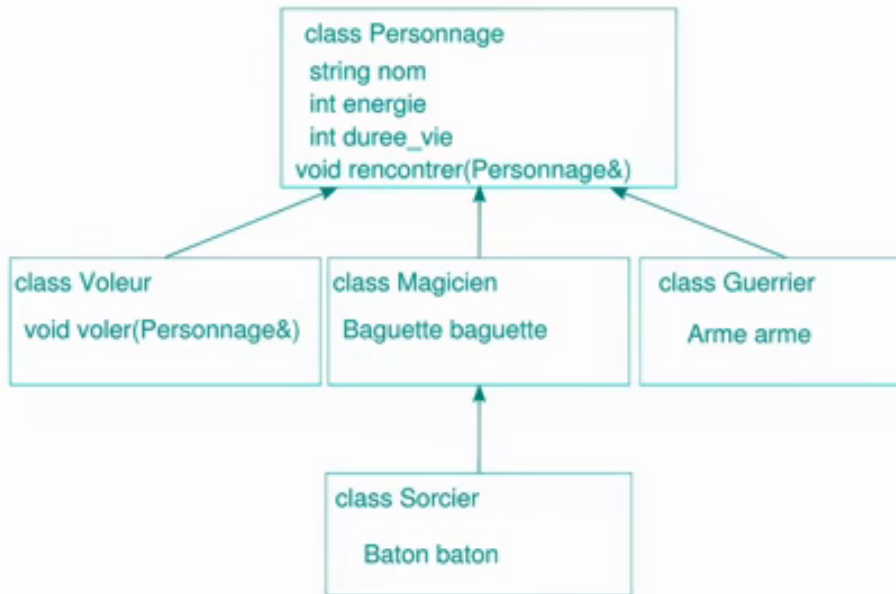
La notion d'héritage en programmation permet de répondre à ce genre de problème en essayant de regrouper au maximum les parties communes, attributs et méthodes communes dans des classes qu'on va appeler des super-classes qui regroupent des caractéristiques qui sont ensuite enrichies ou spécialisées pour d'autres classes plus spécifiques.

notes

résumé

1m 44s





Dans notre cas typiquement ça serait une classe « Personnage », un personnage générique qui d'ailleurs répondrait à un des problèmes de la conception présentée jusqu'ici c'est que on n'avait pas de classe de type « Personnage ». En anticipant bien sûr sur ce qui va suivre, avec l'héritage nous pourrions faire quelque chose comme ceci avoir une classe « Personnage » qui regroupe l'ensemble des caractéristiques communes :

notes

résumé

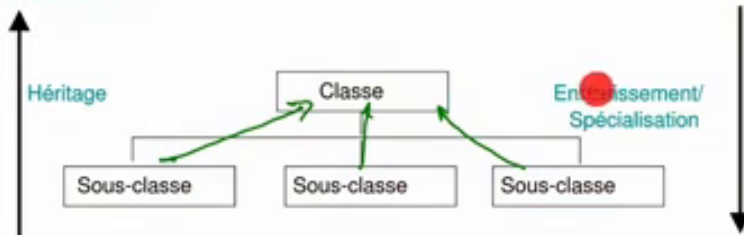
2m 4s



Après les notions d'*encapsulation* et d'*abstraction*, le troisième aspect essentiel de la « Programmation Orientée Objet » est la notion d'**héritage**.

L'héritage représente la relation «**est-un**».

Il permet de créer des classes *plus spécialisées*, appelées **sous-classes**, à partir de classes plus générales déjà existantes, appelées **super-classes**.



nom, énergie, durée de vie et aussi les méthodes donc comme par exemple rencontrer un personnage donc une classe « Personnage » qui regroupe l'ensemble des caractéristiques de ce personnage et dont hériterait le voleur, le magicien, le guerrier, ayant chacun ses spécificités, une arme pour le guerrier, une baguette pour le magicien une spécialisation ici avec une méthode « voler » pour le voleur, ... Et puis, de la classe magicien on pourrait aussi faire une extension supplémentaire, un héritage de plus, pour avoir un sorcier qui est une sorte de magicien qui aurait la baguette et en plus un bâton. Voilà ce à quoi sert l'héritage en programmation orientée objet, il représente la relation « est-un » et permet d'éviter la duplication de code, d'attributs et de méthodes. L'héritage permet ainsi de créer des classes plus spécialisées, des classes enrichies, on va parler donc de « super-classe » pour une classe à partir de laquelle on va dériver des sous-classes ; on dira que les sous-classes héritent et on mettra la flèche donc dans ce sens-là les sous-classes héritent de la super-classe et ces sous-classes seront donc des extensions

notes

résumé

2m 25s



Héritage (2)

Lorsqu'une sous-classe **C1** est créée à partir d'une super-classe **C**,

- ▶ le type est *hérité* : un **C1** **est** (aussi) un **C**
- ▶ **C1** va *hériter* de l'ensemble :
 - ▶ des attributs de **C**
 - ▶ des méthodes de **C**
(sauf les constructeurs et destructeur)
- ☞ Les attributs et méthodes de **C** vont être disponibles pour **C1** sans que l'on ait besoin de les redéfinir explicitement dans **C1**.
- ▶ Par ailleurs :
 - ▶ des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe **C1**
 - ☞ **enrichissement**
 - ▶ des méthodes héritées de **C** peuvent être redéfinies dans **C1**
 - ☞ **spécialisation**

C
↑
C1

C x;
C1 y; x = y;

soit par ajout d'attributs, de méthodes, soit par spécialisation de méthodes ce seront des extensions de la super-classe. Avant de les détailler chacune sur un exemple, voyons les implications de la relation d'héritage : si on a une sous-classe « C1 » qui est créée à partir d'une classe « C », qui hérite d'une classe « C », on va dire que « C1 » est un « C », c'est-à-dire que le type est hérité, donc par exemple si j'ai une variable « x » de type « C », si j'ai une variable « y » de type « C1 », je peux tout à fait mettre « y » dans « x » : « y » est un « C », est aussi un « C » ; par contre, dans cette copie il est bien sûr copié en tant que « C », il perd sa spécificité « C1 » puisqu'il est copié dans un « C », donc il est copié ici en tant que « C », c'est simplement la partie « C » de « y » qui est recopiée dans « x ». Par ailleurs, la classe « C1 » va hériter de l'ensemble des attributs et de l'ensemble des méthodes de la classe « C » ; quand on dit l'ensemble des attributs et l'ensemble des méthodes c'est sauf le constructeur et sauf le destructeur et encore en C++ 2011, on peut hériter des constructeurs mais on en reparlera un petit peu plus tard, dans des séquences vidéo suivantes. Les attributs et les méthodes de « C » seront donc disponibles dans « C1 » sans qu'on ait besoin de les redéfinir explicitement et enfin, des attributs et des méthodes supplémentaires peuvent être ajoutés à « c1 »

notes

résumé

3m 37s

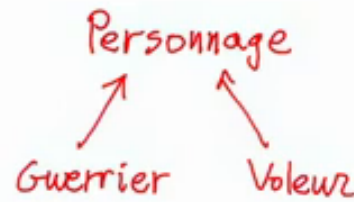


Héritage : exemple

Lorsqu'une sous-classe **C1** (ici **Guerrier** ou **Voleur**) est créée à partir d'une super-classe **C** (ici **Personnage**),

- le type est *hérité* : un **Guerrier** **est** (aussi) **un** **Personnage** :

```
Personnage p;
Guerrier g;
// ...
p = g;
// ...
void afficher(Personnage const&);
// ...
afficher(g);
```



c'est ce qu'on appelle l'enrichissement ou alors des méthodes héritées de « C » dans « C1 » peuvent être redéfinies c'est ce qu'on appelle la spécialisation. Revoyons tout ceci en détail sur un exemple : imaginons donc que l'on ait créé une super-classe « Personnage » et deux sous-classes « Guerrier » et « Voleur » par exemple donc on a dit que lorsqu'une sous-classe « C1 » par exemple ici « guerrier » est créé à partir d'une super-classe « C » donc par exemple ici « Personnage » le type est hérité. En fait, un guerrier est un personnage, est aussi un personnage, ce qui nous permet d'écrire le code suivant : supposons qu'on ait donc déclaré une variable « p » de type « Personnage », une variable « g » de type « Guerrier », on les manipule, et à un certain moment par exemple on dit que le personnage « p » est le guerrier « g », donc on met le guerrier « g » dans le personnage « p », on peut tout à fait faire ceci puisque un guerrier est un personnage, mais attention, on ne met ici de « g » dans « p » que sa partie personnage, cette affectation ici se fait en tant que personnage c'est la partie personnage de « g » qui est copiée dans « p », par exemple, les guerriers ont une arme mais pas les personnages donc ici dans cette copie l'arme ne serait pas copiée.

notes

résumé

5m 1s



Héritage : exemple

Lorsqu'une sous-classe **C1** (ici **Guerrier** ou **Voleur**) est créée à partir d'une super-classe **C** (ici **Personnage**),

- ▶ des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe **Guerrier** : **arme**
- ▶ des méthodes héritées de **Personnage** peuvent être redéfinies dans **Voleur** : **rencontrer(Personnage&)**



Évidemment, on ne peut pas faire le contraire, on ne peut pas écrire « $g=p$ », un personnage générique n'est pas un guerrier on peut avoir des personnages qui sont des voleurs, qui seraient des magiciens, on ne pourrait pas mettre un voleur dans un guerrier, ça n'aurait pas de sens donc ici la relation est bien orientée, c'est un guerrier qui est un personnage. De même, on pourrait imaginer donc avoir une fonction ici qui permet d'afficher un personnage et on peut afficher « g », « g » qui est ici un guerrier est passé comme paramètre à la fonction « afficher » en tant que personnage, la fonction « afficher » le verra en tant que personnage, on peut faire ceci parce que un guerrier est un personnage. Deuxième aspect de l'héritage : « guerrier » va hériter c'est-à-dire recevoir, contenir, l'ensemble des attributs et des méthodes de « Personnage » donc par exemple si « Personnage » avait un nom, une énergie, une durée de vie et disons une méthode « rencontrer » et que « Guerrier » donc hérite de la classe « Personnage », alors dans « Guerrier » on aura aussi un nom, une énergie, une durée de vie sans qu'on ait besoin de les redéfinir et aussi une méthode « rencontrer », donc on pourrait faire des choses comme ceci, supposons qu'on ait ici un guerrier « g » et un voleur « v » qui est aussi un personnage « g » ici, en tant que guerrier, va hériter la méthode « rencontrer » et donc on pourrait appeler, pourvu que cette méthode soit dans l'interface « public » de « Personnage », on va pouvoir appeler « $g.rencontrer(v)$ » ici on passe donc le voleur « v » comme argument à la méthode « rencontrer » mais

notes

résumé

6m 13s



L'héritage permet donc :

- ▶ d'expliciter des relations structurelles et sémantiques entre classes
- ▶ de réduire les redondances de description et de stockage des propriétés



Attention !

- ▶ l'héritage doit être utilisé pour décrire une relation « **est-un** » ("is-a")
- ▶ il ne doit **jamais** décrire une relation « a-un »/« possède-un » ("has-a")

donc par exemple dans la classe « Guerrier » on pourrait rajouter une arme et on peut aussi donc redéfinir des méthodes ce qu'on appelle les spécialiser donc par exemple on pourrait redéfinir la méthode « rencontrer » dans la classe « Voleur » en disant que par exemple un voleur quand il rencontre un autre personnage, il le vole par exemple.

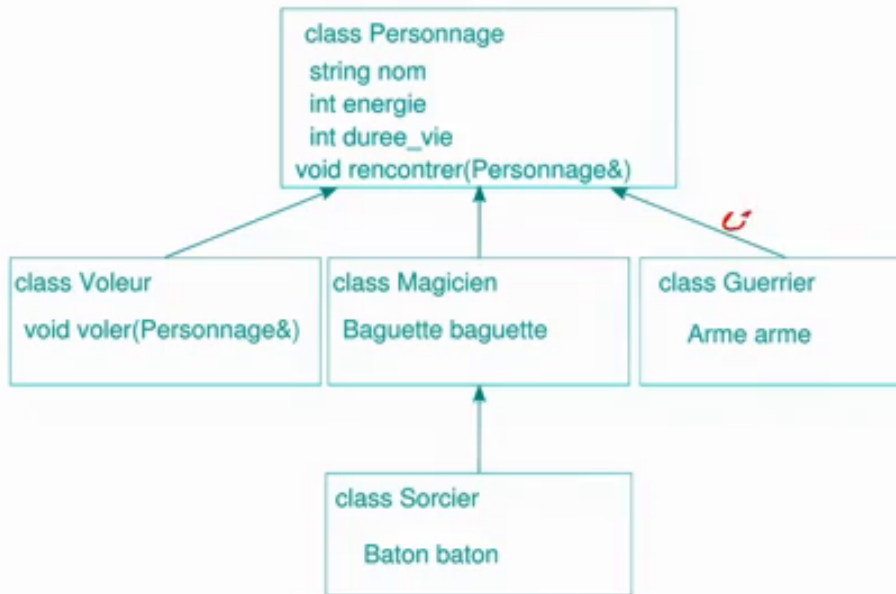
notes

résumé

8m 37s



Exemple : héritage



L'héritage est donc une notion très importante en programmation orientée objet parce qu'elle permet d'organiser le code, elle permet de rendre la conception beaucoup plus claire, elle permet d'explicitier les relations structurelles et sémantiques qui existent entre différentes classes.

notes

résumé

8m 57s



L'héritage permet donc :

ORGANISE
LE
CODE

▶ d'expliciter des relations structurelles et sémantiques entre classes

▶ de réduire les redondances de description et de stockage des propriétés



Attention !

- ▶ l'héritage doit être utilisé pour décrire une relation « **est-un** » ("is-a")
- ▶ il ne doit **jamais** décrire une relation « a-un »/« possède-un » ("has-a")

Par exemple, un guerrier est un personnage, un sorcier est une sorte de magicien

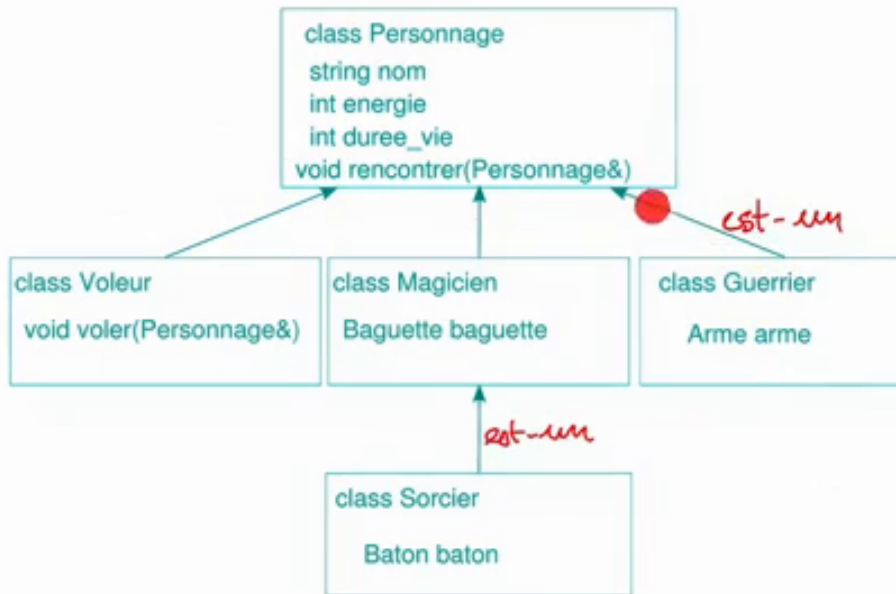
notes

résumé

9m 12s



Exemple : héritage



elle permet aussi donc d'éviter d'avoir à réécrire de grandes portions de code donc ce qu'on appelle de la redondance de code. Attention donc à bien utiliser l'héritage à bonne escient. Il faut utiliser l'héritage pour représenter la relation « est-un », un guerrier est un personnage mais on va pas dire qu'un guerrier est une arme, on dit qu'un guerrier a une arme et pour représenter donc la relation à, possède, on utilisera ce qu'on avait vu jusqu'à maintenant l'encapsulation et donc pour résumer un guerrier héritera de la classe « Personnage » mais aura, encapsulera, aura comme attribut une arme. aura comme attribut une arme.

notes

résumé

9m 17s

