

Support de cours

Cours:

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

Vidéo:

W15-05-polymcoll1-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Méthodes virtuelles. Collections hétérogènes. Séquence vidéo. Pointeurs c. Exemple de notre jeu. Forme d'un tableau dynamique. Façon polymorphique. Instances d'une hiérarchie de classe. Façon globale tous. Comportement spécifique différent. Collection d'objets. Unique pointers. Travers d'une méthode. Méthodes spécifiques. Solution possible.



[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/>

Collections hétérogènes

(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 jusqu'à maintenant que :

- ▶ l'héritage et les méthodes virtuelles permettent de mettre en œuvre des **traitements génériques** sur les instances d'une hiérarchie de classes (polymorphisme d'inclusion).
- ▶ les fonctions/méthodes génériques doivent utiliser des arguments **passés par référence** pour que le traitement se fasse en fonction de la *nature réelle de l'instance*

Qu'en est-il si un tel traitement (générique) doit porter sur un *ensemble* d'instances d'une hiérarchie de classe ?

- ☞ Collection *hétérogène* (au sens où le comportement spécifique de chaque instance de la collection peut être différent)

Dans cette séquence vidéo, nous allons revenir en détail sur une des applications pratiques importantes du polymorphisme que l'on appelle les collections hétérogènes.

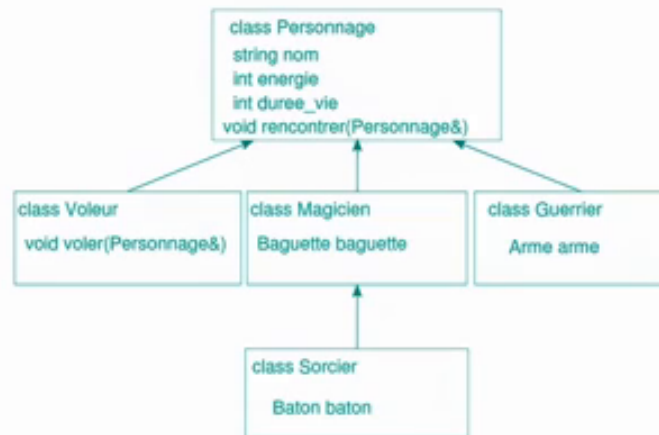
notes

résumé

0m 1s



Rappelez-vous de l'exemple du jeu avec des **Personnages** :



Comment gérer les différentes classes de **Personnages** dans le Jeu ?

Jusqu'ici nous avons vu que l'héritage et les méthodes virtuelles permettaient de définir des traitements génériques sur les instances d'une hiérarchie de classe. Ce que l'on appelle le polymorphisme. Puis l'on a vu que si l'on voulait appliquer des fonctions ou des méthodes que l'on souhaitait être polymorphiques, donc appliquer des traitements que l'on veut polymorphiques à des objets, il fallait que ces objets soient passés par référence. Qu'en est-il si l'on a un ensemble d'objets, une collection d'objets qui appartiennent à une hiérarchie et que l'on souhaite pouvoir manipuler de façon polymorphique ? C'est ce que l'on appelle une collection hétérogène. En ce sens que chaque objet stocké dans la collection, stocké dans l'ensemble peut avoir un comportement spécifique différent. Reprenons pour illustrer ceci l'exemple de notre jeu avec des personnages. Nous avons donc une classe « Personnage » avec un certain nombre d'attributs et de méthodes spécifiques et puis différentes sous-classes : « Voleurs », « Magiciens », « Guerrier », etc. qui étaient des « Personnages ».

notes

résumé

0m 13s





On voulait donc les mettre dans un jeu qui contiendrait plusieurs personnages.

notes

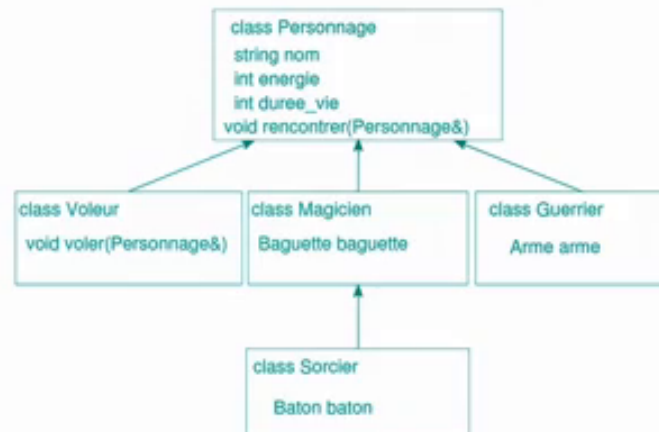
résumé

1m 13s



Collection hétérogène : exemple

Rappelez-vous de l'exemple du jeu avec des **Personnages** :



Comment gérer les différentes classes de **Personnages** dans le **Jeu** ?

Ce serait donc une collection de « Personnages », mais on souhaite que chacun des personnages se comporte pour ce qu'il est en réalité : un « Guerrier » pour un « Guerrier », un « Voleur » pour un « Voleur », etc. Donc cette collection va être hétérogène.

notes

résumé

1m 16s





Comment donc définir le « Jeu » dans ce cas là ? On pourrait bien sûr définir le « Jeu » ainsi : avoir pour chacune des sous-classes une collection spécifique, donc avoir une collection de « Guerriers » pour stocker tous les guerriers, une collection de « Magiciens » pour stocker tous les magiciens, etc. On pourrait ainsi afficher tous les guerriers au travers d'une méthode qui permet d'afficher les guerriers, afficher tous les magiciens au travers d'une méthode spécifique, etc., ajouter éventuellement un « Guerrier » au « Jeu », ajouter un « Magicien », ajouter un « Voleur », etc. C'est une solution possible et dans certains cas, ce sera nécessaire de différencier les différentes sous-classes et d'avoir des collections spécifiques pour chacune des sous-classes. Celles-ci ne sont pas des collections hétérogènes. Ce sont autant de collections spécifiques pour chacune des sous-classes. C'est une solution tout à fait possible, mais l'on pourrait aussi imaginer

notes

résumé

1m 30s



Collection de pointeurs

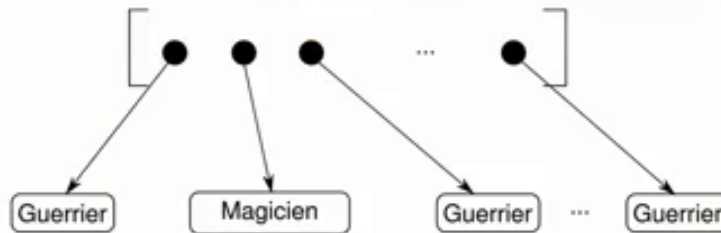
Malheureusement on ne peut pas mettre de référence dans un `vector`.

La solution à ce problème consiste donc à passer par un vecteur de **pointeurs** :

```
class Jeu {
    //...
    vector<Personnage*> personnages;
};
```

```
#include <memory>
// ...
class Jeu {
    //...
    vector<unique_ptr<Personnage>> personnages;
};
```

Notez que donc seuls les pointeurs, c'est-à-dire les *adresses des instances*, sont stocké(e)s dans la collection, et *non plus les instances* elles-mêmes :



notes

résumé

2m 25s





Si vous préférez utiliser suivant nos conseils des pointeurs intelligents à la C++2011, on ferait un tableau dynamique de « unique_pointer » de « Personnage ». Pour utiliser les « unique pointers », je vous rappelle qu'il faut inclure la bibliothèque « memory ».

notes

résumé

5m 1s



Collection de pointeurs

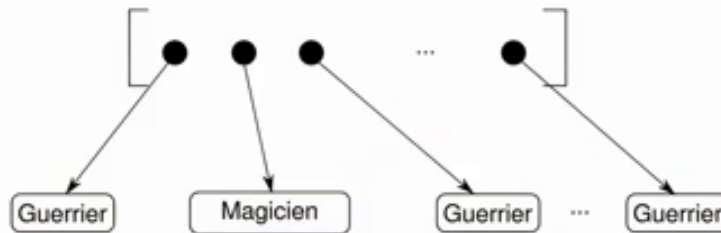
Malheureusement on ne peut pas mettre de référence dans un `vector`.

La solution à ce problème consiste donc à passer par un vecteur de **pointeurs** :

```
class Jeu {
    //...
    vector<Personnage*> personnages;
};
```

```
#include <memory>
// ...
class Jeu {
    //...
    vector<unique_ptr<Personnage>> personnages;
};
```

Notez que donc seuls les pointeurs, c'est-à-dire les *adresses des instances*, sont stocké(e)s dans la collection, et *non plus les instances elles-mêmes* :



Pour ceux qui ont besoin de rappels sur les pointeurs, nous avons remis sur ce cours-ci l'ensemble des cours que nous avons donnés dans notre MOOC précédent sur l'initiation à la programmation. Concernant les pointeurs, je vous encourage à aller les revoir si nécessaire. Donc on pourrait ici faire l'un ou l'autre. Dans tous les cas, à noter que ce sont des pointeurs qui vont être stockés cette fois-ci dans la collection. Le tableau « personnage » ici est un tableau de pointeurs, et non plus directement d'instances. Ceci va avoir des conséquences sur notre conception et sur la vie de notre jeu, puisque maintenant les instances sont en fait en dehors du jeu. Dans le jeu, en réalité on n'a que des pointeurs sur les instances. Comment continuer le développement de notre classe « Jeu » et ajouter ces méthodes pour, par exemple, l'affichage ou l'ajout d'un personnage ?

notes

résumé

5m 20s



Exemple complet : classes

Comment l'utiliser ?

Le plus simple, comme dans la séquence vidéo précédente :

```
Jeu jeu;  
jeu.ajouter_personnage(new Guerrier(...));
```

Le plus simple est de regarder comment nous allons d'abord utiliser ce jeu.

notes

résumé

6m 13s



Exemple complet : classes

Comment l'utiliser ?

Le plus simple, comme dans la séquence vidéo précédente :

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier(...));
```

On aurait donc :

```
class Jeu {
public:
    void afficher() const;
    void ajouter_personnage(Personnage*);

private:
    vector<unique_ptr<Personnage>> personnages;
};
```

Donc, on déclarerait un jeu de l'instance de cette classe Jeu, et on imagine qu'on ajouterait un personnage. Mais on doit ajouter un pointeur. Le plus simple c'est d'ajouter un pointeur alloué dynamiquement ici par un « new » Donc par exemple ici, un nouveau « Guerrier », « new Guerrier » qui va renvoyer un nouveau pointeur sur un « Guerrier ». Donc en mémoire, on aurait la chose suivante : un « Jeu » qui contient sa collection de pointeurs. Au départ, elle est certainement vide. Et puis, on aurait donc création d'un « Guerrier » en mémoire et le new nous rendrait un pointeur sur ce « Guerrier » que l'on stockerait donc dans notre collection. On aurait donc comme prototype pour nos méthodes :

notes

résumé

6m 16s



Exemple complet : classes

Comment l'utiliser ?

Le plus simple, comme dans la séquence vidéo précédente :

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier(...));
```

On aurait donc :

```
class Jeu {
public:
    void afficher() const;
    void ajouter_personnage(Personnage*);

private:
    vector<unique_ptr<Personnage>> personnages;
};
```

ici une méthode « ajouter_personnage » qui prendrait directement un pointeur sur un « Personnage ».

notes

résumé

7m 5s



C'est ce qu'elle reçoit, new renvoie un pointeur à la C sur un personnage. Et puis la méthode afficher qui, elle, ne pose pas de souci, qui va afficher tous les personnages. Donc au niveau de son prototype, le prototype usuel de cette méthode afficher. Voyons maintenant comment définir ces méthodes. Commençons par la méthode qui ajoute un personnage.

notes

résumé

7m 14s



Exemple complet : classes

Comment l'utiliser ?

Le plus simple, comme dans la séquence vidéo précédente :

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier(...));
```

On aurait donc :

```
class Jeu {
public:
    void afficher() const;
    void ajouter_personnage(Personnage*);

private:
    vector<unique_ptr<Personnage>> personnages;
};
```

Son prototype, c'est bien de recevoir un pointeur à la C sur un personnage, pour faire l'appel qu'on a vu juste précédemment. On commence, par mesure de précaution, par vérifier qu'on a bien reçu, au travers du pointeur « nouveau », ici, effectivement un pointeur sur un vrai personnage. Donc que ce pointeur n'est pas nul. Puis ensuite on ajoute à la collection de personnages.

notes

résumé

7m 37s




```
void Jeu::ajouter_personnage(Personnage* nouveau) {  
    if (nouveau != nullptr) {  
        personnages.push_back(unique_ptr<Personnage>(nouveau));  
    }  
}
```

Je vous rappelle que, la collection de personnages, on a décidé de l'implémenter comme un tableau dynamique de `unique_ptr` sur des « Personnages ». Donc on ajoute simplement par un « `push_back` »

notes

résumé

7m 56s



new

```
void Jeu::ajouter_personnage(Personnage* nouveau) {  
    if (nouveau != nullptr) {  
        personnages.push_back(unique_ptr<Personnage>(nouveau));  
    }  
}
```

la transformation du pointeur à la C « nouveau » reçu par l'appel de « new » en un unique_ptr.

notes

résumé

8m 9s



```
jeu. new  
void Jeu::ajouter_personnage(Personnage* nouveau) {  
    if (nouveau != nullptr) {  
        personnages.push_back(unique_ptr<Personnage>(nouveau));  
    }  
}
```

Puisque new renvoie un pointeur à la C et que nous avons décidé ici de choisir d'implémenter nos pointeurs comme des pointeurs intelligents à la C++2011, on va transformer ici ce pointeur à la C en un unique_ptr. Donc précisément en détail, l'appel précédent,

notes

résumé

8m 17s



Exemple complet : méthodes

```

    jeu.ajouter_personnage(new Guerrier(--));
void Jeu::ajouter_personnage(Personnage* nouveau) {
    if (nouveau != nullptr) {
        personnages.push_back(unique_ptr<Personnage>(nouveau));
    }
}

```

« jeu.ajouter_personnage (new Guerrier) » avec un appel au constructeur, avec des paramètres pour le constructeur de « Guerrier » va faire la chose suivante :

notes

résumé

8m 33s



Exemple complet : méthodes

```

    jeu.ajouter_personnage(new Guerrier(--));
}

void Jeu::ajouter_personnage(Personnage* nouveau) {
    if (nouveau != nullptr) {
        personnages.push_back(unique_ptr<Personnage>(nouveau));
    }
}

```

Ge

on va tout d'abord exécuter « new.guerrier » qui va nous construire en mémoire un nouveau « Guerrier ». New va nous renvoyer un pointeur sur cette zone allouée. Ce pointeur va être passé à la méthode ajouter_personnage et s'appelle « nouveau ». Ce pointeur n'est pas nul. On arrive donc ici dans la classe « Jeu » en faisant un push_back en supposant que par exemple que le tableau dynamique était vide. On va faire donc ici un push_back ce qui va ajouter une case à notre tableau dans laquelle on ajoute la transformation de ce pointeur à la C reçu de new en un unique_ptr qui pointera sur la zone mémoire allouée par le new. Voyons maintenant la méthode « afficher ». On suppose que chacun des personnages à sa propre méthode afficher qui est polymorphique. Donc il suffit de faire comme ceci, la boucle sur tous les personnages stockés dans le tableau dynamique de pointeurs sur des « Personnages », et donc d'appeler la méthode afficher au travers de ce pointeur, c'est justement pour ça qu'on avait un pointeur pour avoir ici un comportement polymorphique. Cependant, ce code ne compile pas. Cependant, ce code ne compile pas.

notes

résumé

8m 49s

