

Support de cours

Cours:

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

Vidéo:

W15-05-polymcoll1-CPP-pt4

Concepts (extraits des sous-titres générés automatiquement) :

Deuxième aspect. Unique pointer. Bonne façon. Variable locale. Zone mémoire. Genre de chose. Aspect pointeur intelligent. Séquence vidéo. Travers de ce passage. Lignes de code. Allocation dynamique. Gros avantage. Cas d'une collection. Genre de problème. Adresse d'une variable locale.



[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 4)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



```
void creer_magicien(Jeu& jeu) {  
    Magicien mago(...);  
    jeu.ajouter_personnage(&mago);  
}  
// ...  
int main() {  
    Jeu mon_jeu;  
    creer_magicien(mon_jeu);  
    mon_jeu.afficher(); // ouille !  
    return 0;  
}
```

un nouveau « Magicien » mais qui est une variable locale.

notes

résumé

0m 1s



```
void creer_magicien(Jeu& jeu) {  
    Magicien mago(...);  
    jeu.ajouter_personnage(&mago);  
}  
// ...  
int main() {  
    Jeu mon_jeu;  
    creer_magicien(mon_jeu);  
    mon_jeu.afficher(); // ouille !  
    return 0;  
}
```

La fonction `creer_magicien` ajoute un nouveau magicien au jeu `mon_jeu`, mais par le biais d'une *variable locale* (bouh !)

Une fois l'exécution de `creer_magicien` terminée, la **variable locale est détruite** !



Attention ! Le pointeur stocké dans le vecteur `personnages` **existe toujours...**

Donc il ne faut jamais faire ce genre de chose, jamais ajouter des variables locales. Cette variable locale va cesser d'exister, « mago » ici va cesser d'exister dès que l'on va quitter la fonction. Donc on aura ajouté ici au travers de ce passage par référence un pointeur sur un « Magicien ». Mais ce « Magicien » n'existera plus dès que l'on aura quitté la fonction qui l'a créé. Donc, dès ce point-virgule ici, en toute rigueur ce « Magicien » ne devrait plus être accédé. On ne devrait plus chercher à l'afficher. Donc ici, on ne sait pas ce qu'il va se passer, on va afficher un « Magicien » qui potentiellement n'existe plus. Je dis potentiellement parce que peut-être que sur certaines machines, ce code ici étant relativement court, il ne va rien se passer de grave. Mais si l'on imagine plusieurs lignes de code avec beaucoup de manipulation entre cet appel à cette fonction et cet affichage-là, là c'est pratiquement sûr que vous aurez n'importe quoi comme résultat. Il ne faut donc vraiment jamais faire ça.

notes

résumé

0m 5s



La solution à ce problème est que l'utilisateur **alloue dynamiquement** une portion de mémoire *qui sera préservée* après la fin du bloc où l'on crée l'instance.

Exemple :

```
// définition robuste de la fonction creer_magicien
void creer_magicien(Jeu& jeu) {
    jeu.ajouter_personnage(new Magicien(...));
}
```

Grâce à l'utilisation du **new**, la **mémoire allouée dynamiquement** pour le magicien créé dans **creer_magicien** est préservée à la fin de l'exécution de cette fonction.

Jamais prendre l'adresse d'une variable locale, utiliser comme ça l'adresse d'une variable locale. La bonne façon de faire, de résoudre ce genre de problème c'est d'utiliser ce qu'on appelle l'allocation dynamique et de garantir que la zone mémoire allouée sera préservée au moins aussi longtemps que la collection qui la contient. Donc par exemple, ici, la méthode « **creer_magicien** » qui reçoit toujours un jeu

notes

résumé

1m 13s





notes

résumé

1m 37s



La solution à ce problème est que l'utilisateur **alloue dynamiquement** une portion de mémoire *qui sera préservée* après la fin du bloc où l'on crée l'instance.

Exemple :

```
// définition robuste de la fonction creer_magicien
void creer_magicien(Jeu& jeu) {
    jeu.ajouter_personnage(new Magicien(...));
}
```

Grâce à l'utilisation du `new`, la **mémoire allouée dynamiquement** pour le magicien créé dans `creer_magicien` est préservée à la fin de l'exécution de cette fonction.

par le « unique pointer » dès que la mémoire ne sera plus utilisée. Cela est un des avantages de ces « unique pointers ».

notes

résumé

2m 1s



L'utilisation des « pointeurs intelligents » `unique_ptr` présente deux avantages :

1. pas besoin de se préoccuper de la désallocation
2. l'aspect « unique » évite les références multiples et leur gestion/cohérence

☛ On a beaucoup moins de précautions à prendre et de garde-fous à programmer !

C'est donc pour cela que l'on vous conseille d'utiliser dans un premier temps des pointeurs intelligents, des « unique pointers » parce que d'abord vous n'avez pas du tout besoin de vous préoccuper de la désallocation, donc aucun risque de libération d'une zone mémoire tant qu'on l'utilise. Par contre, le pointeur sera automatiquement libéré dès qu'on n'utilise plus cette zone mémoire. Puis le deuxième aspect, c'est l'aspect « unique ». Il y a l'aspect pointeur intelligent qui permet de gérer la désallocation, et l'aspect « unique » qui évite justement d'avoir des références multiples. Alors on a vu dans l'exemple de la « boucle for » que cela donnait quelques contraintes. Mais c'est un gros avantage d'avoir globalement qu'un seul pointeur qui pointe sur l'objet en question. L'objet appartient comme ça vraiment

notes

résumé

2m 9s





notes

résumé

2m 49s

