

Support de cours

Cours:

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

Vidéo:

W15-06-polymcoll2-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Collection de pointeurs. Version précédente. Tableau de pointeurs intelligent. Unique pointers. Tableau dynamique. Collection hétérogène. Ensemble d'objets polymorphiques. Zone mémoire. Interface de l'ajout d'un personnage. Première chose. Lequel pointeur. Instance de la classe. Paramètre nouveau. Version pointeur. Histoires de pointeurs uniques.



[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 : compléments avancés

(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 que si nous voulions une collection hétérogène, c'est-à-dire un ensemble d'objets polymorphiques regroupés dans un même ensemble, par exemple dans un tableau dynamique, alors, forcément, pour avoir le polymorphisme, il fallait avoir un ensemble, une collection de pointeurs sur ces objets. Et nous vous avons conseillé, jusqu'ici,

notes

résumé

0m 1s



Collection de pointeurs : lesquels ?

- Si l'on veut une collection avec comportement polymorphique des éléments, il **faut** une collection de *pointeurs*

Par exemple ici avec des pointeurs « à la C » :

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

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

Rappel d'utilisation :

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

de faire un tableau de pointeurs intelligent à la C++ 2011, un tableau de « unique pointer ». Regardons dans cette vidéo-ci comment nous pourrions le faire avec des pointeurs « à la C », et ce que cela implique. Pour rappel, nous avons pris pour exemple une classe « Jeu », qui contient une collection de « Personnages » pouvant être polymorphiques, donc pouvant être des guerriers, des voleurs, des magiciens, etc. Et donc, pour ceci, nous avons mis un tableau dynamique, ici, de pointeurs sur des « Personnages ». Dans la version précédente, nous avions mis ici des « unique pointers », regardons maintenant si on met, comme ceci, des pointeurs « à la C », puisqu'après tout, l'interface de l'ajout d'un personnage se fait avec un pointeur « à la C ». Pour rappel, l'utilisation serait d'avoir ici une instance de la classe « Jeu », et d'appeler « jeu.ajouter_personnage » avec une allocation dynamique, ici d'un guerrier, au travers d'un « new Guerrier », qui renvoie bien un pointeur sur un « Guerrier », lequel pointeur sur un « Guerrier »

notes

résumé

0m 25s



Exemple, complet ?

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

private:
    vector<Personnage*> personnages;
};

void Jeu::afficher() const {
    for (auto quidam : personnages) {
        quidam->afficher();
    }
}

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

est bien compatible avec un pointeur sur un « Personnage », puisqu'un « Guerrier » est un « Personnage », un pointeur sur un « Guerrier » est aussi un pointeur sur un « Personnage ». Comment donc, dans ce cas, avec un pointeur « à la C », écrivons-nous les 2 méthodes de la classe « Jeu » ?

notes

résumé

1m 25s



Exemple, complet ?

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

private:
    vector<Personnage*> personnages;
};

void Jeu::afficher() const {
    for (auto quidam : personnages) {
        quidam->afficher();
    }
}

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

Commençons par l'ajout d'un personnage, dont le prototype n'a pas changé. Donc on reçoit, effectivement ici, un paramètre nouveau, qui est un pointeur « à la C » sur un « Personnage », puisque ceci serait le résultat de l'appel à « new ». Et donc comme on a décidé ici d'avoir un tableau dynamique de pointeurs « à la C » sur des « Personnages », il n'y a rien de compliqué à faire. Il suffit simplement d'ajouter

notes

résumé

1m 42s



Exemple, complet ?

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

private:
    vector<Personnage*> personnages;
};

void Jeu::afficher() const {
    for (auto quidam : personnages) {
        quidam->afficher();
    }
}

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

« push_back » à la fin du tableau dynamique de « Personnages » le paramètre nouveau que l'on a reçu.

notes

résumé

2m 1s



Exemple, complet ?

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

private:
    vector<Personnage*> personnages;
};

void Jeu::afficher() const {
    for (auto quidam : personnages) {
        quidam->afficher();
    }
}

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

On peut, comme dans la version précédente, commencer par, au préalable, vérifier que l'on reçoit effectivement un vrai pointeur qui n'est pas nul, c'est-à-dire un pointeur sur un vrai objet. Cette version est un petit peu plus simple que dans le cas des « unique pointers ». De même, pour l'affichage,

notes

résumé

2m 7s



Exemple précédent (unique_ptr)

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

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

void Jeu::afficher() const {
    for (auto const& quidam : personnages) {
        quidam->afficher();
    }
}

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

ici on peut simplement effectuer une classique boucle « for auto », puisqu'il n'y a aucun soucis. Il n'y a pas de problème, d'empêchement d'avoir plusieurs pointeurs sur la même zone mémoire, comme dans le cas des « unique pointers ». Donc ici rien de particulier à faire par rapport aux « unique pointers ». Je vous laisse d'ailleurs comparer les 2 codes pour voir où sont les différences. On voit que dans la version à « unique pointer », on a dû rajouter ici « unique_ptr »,

notes

résumé

2m 25s



Exemple précédent (unique_ptr)

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

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

void Jeu::afficher() const {
    for (auto const& quidam : personnages) {
        quidam->afficher();
    }
}

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

au lieu d'avoir simplement une étoile (*). On a aussi dû, pour les histoires de pointeurs uniques, pour éviter d'avoir plusieurs pointeurs sur la même zone mémoire, rajouter ici un « const » qui n'existe pas dans la version pointeur « à la C ». Et enfin on a dû rajouter ici la conversion d'un pointeur « à la C » en un « unique pointer »,

notes

résumé

3m 1s





notes

résumé

3m 26s

