

Support de cours

Cours:

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

Vidéo:

W15-06-polymcoll2-CPP-pt3

Concepts (extraits des sous-titres générés automatiquement) :

Personnage numéro. Collection hétérogène. Unique pointers. Seul élément. Problème de l'intégrité des données. Première chose. Tableau de ses pointeurs. Deuxième personnage. Fait possibles. Cercles d'autres dessins. Fonction de votre conception. Mémoire des pointeurs. Dernier pointeur du tableau. Smart pointers. 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 : compléments avancés

(Partie 3)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



Gare aux pointeurs !

Qui dit «pointeurs», dit aussi « **bonne gestion** » et « **programmation rigoureuse** »...

En particulier pensez, si nécessaire, à la *copie profonde* et au *destructeur* pour libérer la mémoire allouée

Et n'oubliez pas la règle d'or :

c'est celui qui a alloué la mémoire (**new**) **qui est chargé de la libérer** (**delete**)

Par exemple ici, fournir une fonction

```
void Jeu::destruire_personnage(Personnage* qui);
```

ou

```
void Jeu::destruire_personnage(size_t index);
```

ou alors (voire les deux) :

```
void Jeu::destruire_tout();
```

(Rappel : `jeu.ajouter_personnage(new Magicien(...));`)

On pourrait aussi détruire un « Personnage » en indiquant quel est le numéro,

notes

résumé

0m 1s





l'index de ce « Personnage », en supposant qu'on ait numéroté, qu'on ait accès au numéro des « Personnages », détruire le personnage numéro 2, etc. Ou alors, carrément offrir une méthode qui détruit tous les « Personnages ».

notes

résumé

0m 5s



Gare aux pointeurs !

Qui dit «pointeurs», dit aussi « **bonne gestion** » et « **programmation rigoureuse** »...

En particulier pensez, si nécessaire, à la *copie profonde* et au *destructeur* pour libérer la mémoire allouée

Et n'oubliez pas la règle d'or :

c'est celui qui a alloué la mémoire (new) qui est chargé de la libérer (delete)

Par exemple ici, fournir une fonction

```
void Jeu::destruire_personnage(Personnage* qui);
```

ou

```
void Jeu::destruire_personnage(size_t index);
```

ou alors (voire les deux) :

```
void Jeu::destruire_tout();
```

(Rappel : `jeu.ajouter_personnage(new Magicien(...));`)

Ça, c'est libre à vous, c'est en fonction de votre conception, de la collection hétérogène que vous voulez créer. Mais dans tous les cas, il va falloir penser

notes

résumé

0m 22s



```
void Jeu::destruire_tout()
{
    for (auto quidam : personnages) {
        delete quidam;
    }
    personnages.clear();
}
```

à fournir des méthodes qui permettent de libérer la mémoire allouée, libérer les pointeurs qui sont stockés dans votre collection.

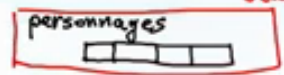
notes

résumé

0m 29s



Libération mémoire (2)



```
void Jeu::destruire_personnage(size_t lequel)
{
    delete personnages[lequel];
    // puis, en fonction des situations :

    // SOIT
    // préserve les index et la taille de la collection
    personnages[lequel] = nullptr;

    // SOIT
    // suppression efficace mais ne préserve pas l'ordre
    swap(personnages[lequel], personnages.back());
    personnages.pop_back();

    // SOIT
    // suppression plus coûteuse qui préserve l'ordre
    personnages.erase(personnages.begin() + lequel);
}
```

Par exemple, si on a décidé d'offrir la méthode qui détruit tous les objets pointés stockés dans la collection, qui détruit tous les « Personnages » pointés dans le jeu, on parcourrait comme ceci, avec une boucle « for », tous les personnages, tous les pointeurs pointés dans le tableau « personnages », et on « delete » chacun de ces pointeurs. Évidemment, à la fin, on n'oublie pas de vider le tableau de ses pointeurs, pour ne pas garder en mémoire des pointeurs qui pointent vers des zones mémoires qui ont été libérées. Une autre façon, si l'on souhaitait fournir la méthode qui permet de détruire un personnage, indiqué par sa position, par exemple détruire le deuxième personnage, on aurait la situation suivante : on a un jeu qui contient son tableau de « Personnages »

notes

résumé

0m 38s





notes

résumé

1m 25s



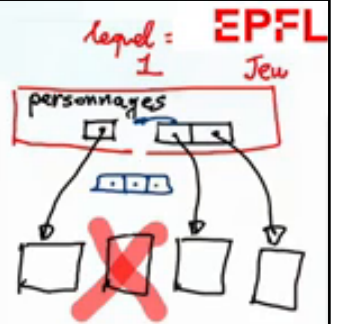
Libération mémoire (2)

```
void Jeu::destruire_personnage(size_t 1lequel) < personnages.size()
{
    delete personnages[lequel]; if (...)
    // puis, en fonction des situations :

    // SOIT
    // préserve les index et la taille de la collection
    personnages[lequel] = nullptr;

    // SOIT
    // suppression efficace mais ne préserve pas l'ordre
    swap(personnages[lequel], personnages.back());
    personnages.pop_back();

    // SOIT
    // suppression plus coûteuse qui préserve l'ordre
    personnages.erase(personnages.begin() + lequel);
}
```



Donc là, 3 solutions en fonction de ce que vous souhaitez faire. Les 3 solutions sont tout à fait possibles. Ça dépend de comment on veut gérer la collection hétérogène : soit on veut garder le même ordre et le même nombre de cases dans le tableau de « Personnages », le même nombre de personnages possibles, mais indiquer simplement que ce personnage a disparu, à ce moment là, ce qu'on pourrait faire c'est simplement dire que on remplace le pointeur invalide par une adresse nulle, en disant que ce personnage a été détruit, représenté par un « null pointer », « nullptr », ce qui donnerait donc la situation suivante en mémoire, et donc on aurait bien ici un tableau qui continue de garder 4 personnages possibles, avec un pointeur nul ici pour indiquer que le personnage à la position 1, le deuxième personnage, n'existe plus. Ça c'est une solution possible, mais qui implique que celui qui utilise le tableau prenne alors ses précautions, et teste si l'objet est bien présent, avant de, par exemple, lui appliquer une méthode. Une autre solution quand on détruit un « Personnage », ça serait de dire qu'à ce moment là, le nombre de personnages dans le jeu diminue de 1, la taille de ce tableau diminue de 1, et donc de supprimer carrément cette case. Là il y a 2 solutions possibles, une solution efficace qui ne préserve pas l'ordre, mais qui est très efficace, qui ne supprime qu'un seul élément, et une solution qui préserve l'ordre, mais qui va être coûteuse, parce qu'elle va demander de recopier tous les éléments qui suivent l'élément que l'on est en train de détruire. Regardons la première solution. La première solution consiste simplement à échanger, avec la fonction « swap » de « utility », à échanger le pointeur sur lequel on vient

notes

résumé

2m 33s



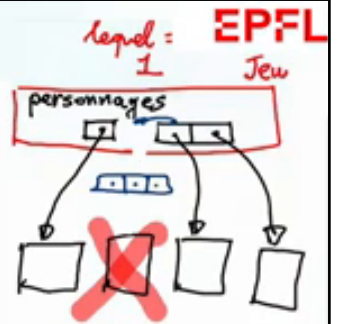
Libération mémoire (2)

```
void Jeu::destruire_personnage(size_t 1lequel) < personnages.size()
{
    delete personnages[lequel]; if (...)
    // puis, en fonction des situations :

    // SOIT
    // préserve les index et la taille de la collection
    personnages[lequel] = nullptr;

    // SOIT
    // suppression efficace mais ne préserve pas l'ordre
    swap(personnages[lequel], personnages.back());
    personnages.pop_back();

    // SOIT
    // suppression plus coûteuse qui préserve l'ordre
    personnages.erase(personnages.begin() + lequel);
}
```



d'appliquer « delete » avec le dernier pointeur du tableau, « personnages.back », je vous rappelle, c'est le dernier élément du tableau « personnages », et on va donc échanger ces 2 éléments, ce qui va donc donner ceci, si l'on échange les 2 pointeurs, c'est maintenant cette case-là qui pointera ici, et cette case-ci qui pointera là-bas. C'est l'échange de ces 2 pointeurs. Une fois que cet échange a été fait, alors à ce moment là, on peut maintenant supprimer le dernier élément du tableau, puisqu'on sait que maintenant, ce dernier élément c'est justement le pointeur qu'on ne veut pas garder. Donc avec ce « pop back », on va supprimer ici ce dernier élément. Ce qui fait qu'au final, on a effectivement plus que 3 personnages, et plus aucune indication de l'adresse de cette case mémoire que nous venons de libérer. Une troisième solution, en fonction des besoins, serait de supprimer ce pointeur, mais en gardant l'ordre, c'est-à-dire en gardant ce pointeur au début, mais celui-ci en deuxième, celui-ci en troisième, et non pas en inversant comme ceci, les ordres qu'on avait perturbés tout à l'heure. En utilisant, comme ceci, la méthode « erase » des tableaux dynamiques, des « vector », qui supprime, comme ceci, je vous le livre ici sans plus de détails, l'élément à la position « lequel », et qui va, et c'est en cela qu'elle est coûteuse, effectivement supprimer cet élément, mais copier tous les éléments qui suivent, les déplacer les uns après les autres, pour que ce tableau reste un tableau « connexe » en mémoire à 3 éléments.

notes

résumé



Voilà donc comment on pourrait procéder,

notes

résumé

5m 49s



Problème potentiel avec des pointeurs « à la C » :

intégrité des données

3 facettes :

1. durée de vie des données
2. désallocation
3. partage des données entre collections

unique_ptr	pointeurs « à la C »
!!	!!
✓	X !!
✓	

d'une façon ou d'une autre, pour pouvoir offrir la possibilité de supprimer les éléments de la collection. Pour résumer, les pointeurs « à la C » dans une collection hétérogène posent le problème de l'intégrité des données. En fait, il y a 3 facettes à ce problème, le premier c'est de garantir la durée de vie des données pointées, la durée de vie des personnages du jeu, au moins aussi longtemps que le jeu existe. Nous en avons déjà parlé dans le cas des pointeurs « unique pointers ». Donc nous en avons parlé dans la vidéo précédente. C'est un point que l'on trouve à la fois dans les « unique_ptr », et dans les pointeurs « à la C ». Le deuxième problème, c'est le problème de la désallocation, il faut garantir que l'on libère la mémoire que l'on a allouée, et ça, du point de vue des « unique_ptr », comme ce sont des « smart pointers », il n'y a rien à faire. Donc ici c'est tout bon, par contre, évidemment, dans les pointeurs « à la C », ce n'est pas fait. Donc il faut, bien sûr, c'est ce que nous venons juste de détailler, offrir le moyen de le faire proprement. Enfin, troisième problème, le problème de partage des données entre différentes collections, on l'a déjà brièvement évoqué avec la copie tout à l'heure, et je vais y revenir dans un instant. Ceci ne pose aucun soucis au niveau des « unique_ptr ».

notes

résumé

5m 51s



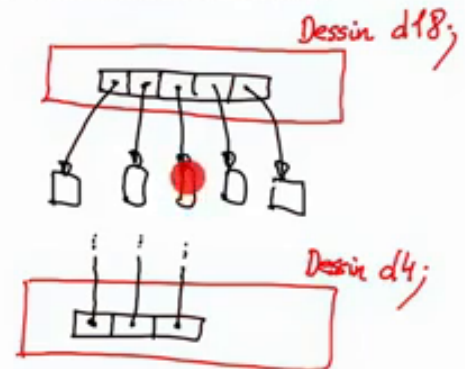
Partage des données ?

Considérons un programme de modélisation graphique manipulant des *dessins*, lesquels sont des ensembles de *figures* géométriques :

- **Figure** comme classe abstraite, avec différentes sous-classes concrètes (cercles, rectangles, carrés, ...)
- **Dessin** comme *collection hétérogène* de figures.

Le contenu d'un **Dessin** est-il **personnel** ou **partagé** ?

Par exemple, si l'on colorie en rouge le cercle 23 du dessin 18, est-ce que seul ce cercle sera rouge ou bien d'autres ? du même dessin ? d'autres dessins ?



Les réponses à ces questions dépendent du cadre général du programme et de sa **conception**, et n'ont pas de réponse unique.

C'est justement l'aspect « unique » qui empêche que l'on ait du partage de données entre les différentes collections, mais qui n'est pas du tout garanti par les pointeurs « à la C », et il va donc falloir le traiter. Pour finir, reprenons donc ce dernier point en détail. J'ai choisi, pour cela, de considérer un exemple qui me semble un tout petit peu plus illustratif. On imagine qu'on a des « dessins », qui seront notre collection hétérogène, des « dessins » étant des ensembles de « figures » géométriques. On aurait donc 2 classes, la classe « Figure » qui est une classe abstraite, dont on imagine, par exemple, qu'on aurait différentes figures, comme des rectangles, des cercles, etc. qui hériteraient. Et on aurait donc une collection hétérogène de figures, qui serait notre « Dessin », qui contiendrait donc différentes figures, lesquelles sont polymorphiques, des rectangles, des cercles, etc. La question est donc de savoir si le contenu d'un « Dessin » est personnel ou partagé. Donc si par exemple, j'imagine avoir plusieurs dessins, par exemple le dessin « d18 », qui contient une collection de pointeurs, bien sûr, puisque l'on souhaite que les figures se comportent de façon polymorphique, de pointeurs sur des figures, et imaginons que j'ai donc un autre dessin, qui contient aussi sa collection de pointeurs sur des figures. Admettons que par exemple, un de ces objets soit un cercle,

notes

résumé

7m 13s





et qu'on décide de changer un attribut de ce cercle, par exemple de colorier ce cercle en rouge. La question c'est : si je colorie ce cercle en rouge, est-ce que ce sera juste le cercle de ce dessin « d18 » qui sera colorié en rouge, ou bien aussi d'autres cercles d'autres dessins. Par exemple, est-ce que ce cercle est partagé par les dessins « d18 » et « d4 », auquel cas, si je le modifie par l'un, alors il va apparaître modifié dans tous.

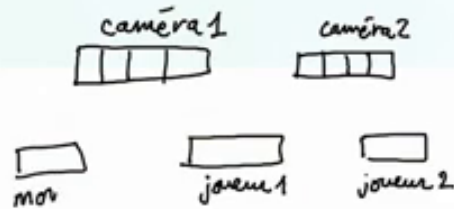
notes

résumé

8m 49s



Partage des données ?



Le contenu **personnel** ou **partagé** ?

- Dans le cas des dessins, il semble naturel que les éléments de la collection (les figures) soient uniques et personnels.
- Dans le cas des personnages *du* jeu aussi (avec qui partager ?)

☞ Ceci est *garanti* par les `unique_prt` !

La réponse à ce genre de question dépend bien évidemment du cadre dans lequel vous développez votre collection hétérogène. Ça dépend de la conception de votre collection hétérogène, mais je crois vraiment que dans le cas des dessins, on ne souhaite pas que les figures soient partagées par plusieurs dessins, mais bien que chaque dessin ait ses propres figures. Un dessin a son cercle, un autre dessin a un autre cercle, qui sont différents. Dans le cas des jeux qu'on avait pris comme premier exemple, je pense aussi que les contenus ne sont pas partagés. D'ailleurs avec qui partager ? Je pense que dans ces cas-là, on n'a qu'un seul jeu et on va éviter d'avoir des copies de jeu. Si vous voulez un exemple où l'on pourrait imaginer avoir des collections qui partagent leur contenu, ça pourrait être aussi un jeu, mais un peu plus compliqué, où on aurait différentes caméras qui pourraient voir les joueurs du jeu sous différents angles, et alors on imaginerait que chaque caméra ait par exemple une collection de « Personnages » qu'elle peut voir,

notes

résumé

9m 16s





comme ceci. Donc par exemple, la caméra 1 permettrait de voir le monstre 1 et le joueur 2, la caméra 2 permettrait de voir les 2 joueurs, et on aurait comme ceci, des éléments du jeu qui seraient partagés par plusieurs collections. Mais ce genre de conception est beaucoup plus avancé, et peut-être un peu plus rare.

notes

résumé

10m 13s



Problème potentiel avec des pointeurs « à la C » :

intégrité des données

3 facettes :

1. durée de vie des données
2. désallocation
3. partage des données entre collections

unique_ptr	pointeurs « à la C »
!!	!!
✓	X !!
✓ (unique)	X !!

Dans les exemples qu'on vous a donnés, on ne souhaiterait certainement pas partager le contenu des collections entre différentes collections, on voudrait que chaque collection ait ses propres personnages, dans le cas du jeu, ses propres figures, dans le cas des dessins. Donc, soit il faudrait le garantir au niveau du programme, donc faire des méthodes, faire des fonctions pour garantir que c'est bien le cas, soit utiliser des « unique_ptr » qui, justement, empêchent que l'on ait plusieurs pointeurs sur le même objet. Pour conclure, on vous a donc conseillé d'utiliser les « unique pointers » pour avoir plus facilement une meilleure intégrité des données. Dans tous les cas, « unique pointers » ou pointeurs « à la C », vous devez vous poser le problème de la durée de vie des données pointées pas ces collections, donc des objets contenus dans les collections. Mais dans le cas des « unique pointers », vous n'avez pas à vous préoccuper de la désallocation, ça c'est l'aspect « smart pointers » des pointeurs à la C++ 2011, ni du partage de données entre collections, ça c'est l'aspect « unique » des « unique pointers ». Si jamais vous partez pour une implémentation de pointeurs « à la C », alors il faudra, non seulement vous poser ces questions, mais aussi y apporter des réponses. mais aussi y apporter des réponses.

notes

résumé

10m 36s

