

Support de cours

Cours:

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

Vidéo:

W12-04-destructeurs-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Attributs de vie d'un objet. Variable locale de type. Moyen de pointeurs. Variante de notre fameuse classe. Séquences vidéo précédentes. Sujet de la séquence. Stade de l'exécution. Début de vie d'un objet. Programme principal. Guise d'attribut des champs. Exemple simple. Meilleure définition possible. Réservation d'un espace mémoire. Zones mémoires. Cas concret.



[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/>

Destructeurs

(Partie 1)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





Dans les séquences vidéo précédentes, nous nous sommes intéressés

notes

résumé

0m 1s



Destructeur

SI l'initialisation des attributs d'une instance implique la **mobilisation de ressources** : fichiers, périphériques, portions de mémoire (pointeurs), etc.

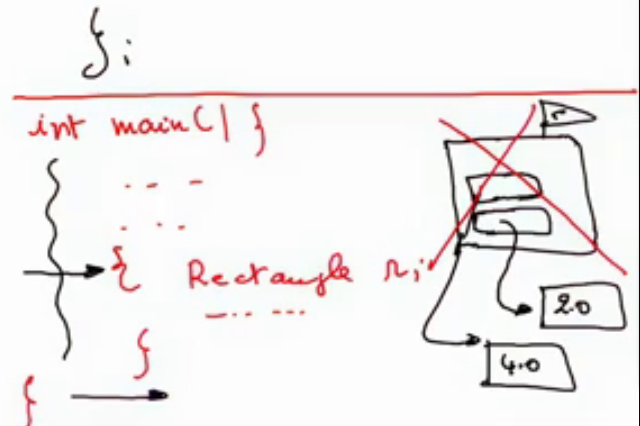
il est alors important de **libérer ces ressources** après usage !

Comme pour l'initialisation, l'*invocation explicite* de méthodes de libération n'est pas satisfaisante (fastidieuse, source d'erreur, affaiblissement de l'encapsulation).

C++ offre une méthode appelée **destructeur** invoquée automatiquement en fin de vie de l'instance.

```
class Rectangle {
public:
    Rectangle(): hauteur(new double(20)),
               largeur(new double(40))
    {}
private:
    double* hauteur;
    double* largeur;
};
```

delete ?



à gérer le début de vie d'un objet. Comment initialiser les attributs de vie d'un objet lorsqu'on commence à travailler avec. Il est temps maintenant de se préoccuper de gérer ce qui se passe en fin de vie d'un objet. Et c'est le sujet de la séquence d'aujourd'hui. Se préoccuper de gérer la fin de vie d'un objet est en fait important lorsque cet objet mobilise des ressources. Examinons un cas concret pour commencer, prenons une variante de notre fameuse classe « Rectangle », dans laquelle les attributs, au lieu d'avoir été déclarés comme des « doubles », soient maintenant déclarés comme étant des « pointeurs » sur des « doubles ». Comme ceci. Et imaginons qu'un des constructeurs de « Rectangle » alloue de la mémoire pour ces 2 attributs au moment de la création de l'objet. Par exemple, on peut imaginer qu'il y ait un constructeur par défaut qui initialiserait l'attribut hauteur en allouant dynamiquement de la mémoire pour un « double » et ferait donc la même chose pour la largeur. Cette définition au moyen de pointeurs n'est pas forcément la meilleure définition possible pour la classe « Rectangle », nous l'utilisons ici uniquement pour avoir un exemple simple. Imaginez maintenant un programme qui utiliserait la classe « Rectangle », on peut imaginer par exemple que ça se fasse simplement dans un programme principal, on aurait des instructions, puis à un moment donné, imaginez que l'on ouvre un bloc, par exemple, dans le cadre d'un « if », et dans ce bloc, on déclare une variable locale de type « Rectangle », qui serait typiquement initialisée par le constructeur par défaut, et qu'à un moment donné, on ferme ce bloc. Si j'exécute ce programme, lorsque j'arrive à ce stade de l'exécution, il y aura réservation d'un espace mémoire pour une variable « r » de type « Rectangle »

notes

résumé

0m 5s



Destructeur

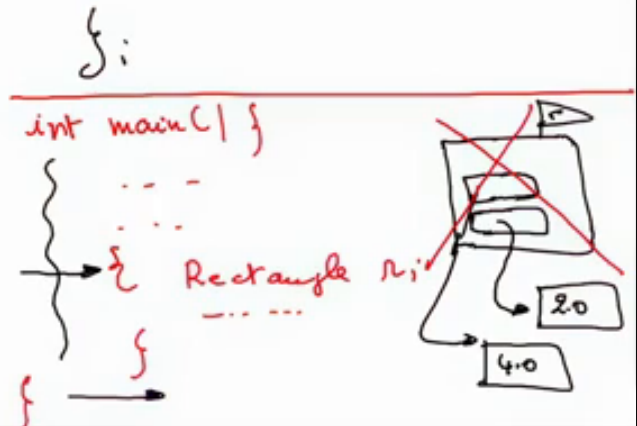
SI l'initialisation des attributs d'une instance implique la **mobilisation de ressources** : fichiers, périphériques, portions de mémoire (pointeurs), etc.

il est alors important de **libérer ces ressources** après usage !

Comme pour l'initialisation, l'*invocation explicite* de méthodes de libération n'est pas satisfaisante (fastidieuse, source d'erreur, affaiblissement de l'encapsulation).

C++ offre une méthode appelée **destructeur** invoquée automatiquement en fin de vie de l'instance.

```
class Rectangle {
public:
    Rectangle() : hauteur(new double(20)),
                largeur(new double(40))
    {}
private:
    double* hauteur;
    double* largeur;
};
```



», laquelle aura en guise d'attribut des champs largeur et hauteur dont les valeurs sont des pointeurs vers 2 zones mémoires allouées dynamiquement par le constructeur. Lorsque nous arrivons à ce stade d'exécution du programme, la variable « r », qui était déclarée localement dans ce bloc, n'existe plus et donc n'est plus utilisable dans le reste du programme. La question qui se pose est donc : Qu'advient-il des ressources en mémoire qui dépendent de cet objet « r », qui n'est plus utilisé par le programme ? Il est important de libérer les ressources après usage, comme ici ces zones mémoires dynamiquement allouées. Nous savons que dans le cadre de l'allocation de mémoire, par exemple, toute zone mémoire allouée dynamiquement au moyen d'un « new » doit être explicitement désallouée par le programmeur, au moyen d'une instruction « delete ». Et la question qui se pose, c'est qui va être en charge de ce « delete » concrètement, ici ? La solution qui consisterait à permettre au programmeur utilisateur de libérer cette mémoire, par exemple, ici, lorsqu'on sait qu'on n'a plus besoin d'utiliser la variable « r », invoquer le « delete » explicitement sur chacun des champs, sur chacun des attributs, n'est pas une bonne solution. Sauriez-vous dire pourquoi ? Sauriez-vous dire pourquoi ?

notes

résumé