Support de cours

Cours:

## Initiation à la programmation (en C++)

Vidéo:

## Pointeurs - allocation dynamique (partie 2)

Concepts (extraits des sous-titres générés automatiquement) :

**Zone mémoire capable. Nom d'un pointeur. Adresse de cette zone mémoire. Emplacement mémoire. Ligne de code suivante. Fameuse petite instruction. Allocation dynamique. Syntaxe suivante. Programme c. Variable de type pointeur. Opérateur delete. Nom de type. Opérateur new. Contre-partie. Façon sûre.**

vers la recherche de séquences vidéo
(dans Initiation à la programmation (en C++).)

vers la vidéo

EPFL

# Pointeurs : allocation dynamique
## (Partie 2)

Initiation à la programmation (C++)

Vincent Lepetit, Jean-Cédric Chappelier et Jamila Sam

...

notes

résumé

0m 0s

# Allocation de mémoire (2)

EPFL

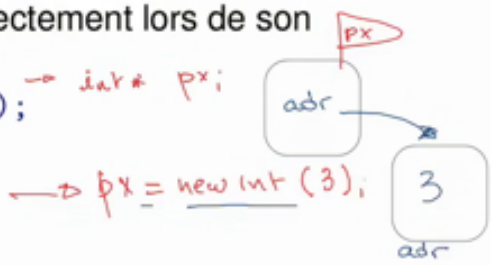
C++ possède deux opérateurs `new` et `delete` permettant d'allouer et de libérer dynamiquement de la mémoire.

```
pointeur = new type;
```

réserve une zone mémoire de type *type* et met l'adresse correspondante dans *pointeur*.

Il est également possible d'initialiser l'élément pointé directement lors de son allocation :

```
pointeur = new type(valeur);
```




Très concrètement, si vous voulez faire de l'allocation dynamique dans un programme C++, eh bien vous pourrez utiliser deux opérateurs prédéfinis par le langage : l'opérateur new qui permet d'allouer la mémoire qui vous est nécessaire, et l'opérateur delete, la contre-partie, qui permet de libérer cette mémoire lorsque vous n'en n'avez plus besoin dans le programme. L'utilisation de l'opérateur new se fait selon la syntaxe suivante : vous devez mettre le nom d'un pointeur qui va être destiné à recueillir une adresse, et vous utiliserez la syntaxe suivante : new suivi d'un nom de type. Ce que ceci va faire, c'est que ça va réserver une zone mémoire capable de contenir une donnée de ce type, et retourner l'adresse de cette zone mémoire, laquelle va pouvoir être affectée au pointeur. Soyons concret. Imaginons par exemple, que dans un programme, il y ait la ligne de code suivante : donc cette ligne, ce qu'elle fait c'est déclarer un pointeur capable de pointer sur un entier. Donc je vais maintenant adhérer à la syntaxe que l'on vient de voir ici, pour faire de l'allocation dynamique. Donc je prends mon nom de pointeur et j'écris donc px suivi de "=" et suivi enfin de ma fameuse petite instruction "new type;". Donc ici, étant donné que mon pointeur est un pointeur sur entier, eh bien il faut que "type", ici, corresponde au type sur lequel peut pointer le pointeur et donc je devrai mettre ici "int". Donc à ce moment-là, j'ai fait ce qu'on appelle une allocation dynamique. Alors comment ça va se passer, en clair ? Donc ici, au moment où vous compilez cette ligne de code, je dis bien compiler et non pas exécuter, vous aurez une allocation statique pour une variable de type pointeur, qui s'appelle px, capable de recueillir l'adresse d'un entier. Par contre, c'est au moment où vous exécutez cette ligne de code que vous aurez une allocation, cette

notes

résumé

0m 1s

## Allocation de mémoire (2)

EPFL

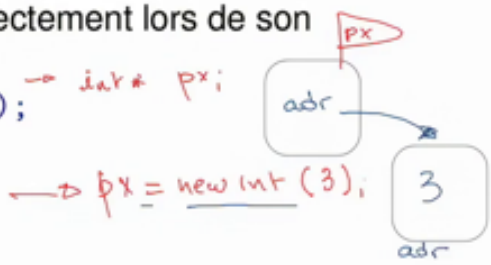
C++ possède deux opérateurs `new` et `delete` permettant d'**allouer** et de **libérer** dynamiquement de la mémoire.

```
pointeur = new type;
```

réserve une zone mémoire de type *type* et met l'adresse correspondante dans *pointeur*.

Il est également possible d'initialiser l'élément pointé directement lors de son allocation :

```
pointeur = new type(valeur);
```




fois dynamique. Et à ce moment-là, vous avez alloué un emplacement mémoire pour un entier, qui est cet emplacement. Cet emplacement aura une adresse, et c'est précisément cette adresse que vous allez stocker dans px comme ceci. Donc ceci se passe au moment... Donc il s'agit ici d'une allocation statique. Par contre ici, vous avez une allocation dynamique. C'est l'instruction, ici "new int;", qui permet justement de faire cette allocation. Si on a recours à cette syntaxe pour faire de l'allocation dynamique, la zone nouvellement allouée est réservée, certes, mais ne contient aucune valeur. Donc si on veut, au moment de l'allocation de mémoire, mettre en plus une valeur dans la zone mémoire, eh bien il faut avoir recours à l'alternative suivante : de l'instruction d'allocation dynamique. Donc, si l'on s'en réfère à l'exemple que nous avons pris tout à l'heure, donc j'ai par exemple ici un pointeur px déclaré dans un programme. On peut utiliser cette variante de l'instruction d'allocation dynamique comme ceci. Donc : "px = new int;". Donc, je réserve un emplacement mémoire pour un entier et je mets une valeur dedans, ici par exemple, la valeur 3. Ce qui va correspondre, au moment de l'exécution de cette ligne de code, à la réservation d'un emplacement mémoire pour un entier, ce que fait le "new int;". On y met une valeur, qui est 3. Le "new int(3);" retourne l'adresse de la zone nouvellement allouée, et l'affectation va faire en sorte que cette adresse soit copiée dans px.

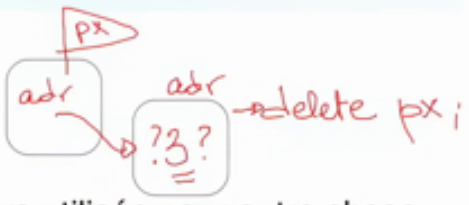
notes

résumé

# Libération de la mémoire allouée

EPFL

```
delete pointeur;
```




libère la zone mémoire allouée au pointeur *pointeur*.

C'est-à-dire que cette zone mémoire peut maintenant être utilisée pour autre chose.
Il ne faut plus y accéder !...

Conseil – Bonnes pratiques

*faire suivre tous les* `delete` *de l'instruction «* `pointeur = nullptr;` *»)*

Conseil – Bonnes pratiques

*Toute zone mémoire allouée par un* `new` *doit impérativement être libérée par un* `delete` *correspondant !*

Vous savez désormais allouer dynamiquement de la mémoire. Que faire pour libérer cette mémoire lorsque vous n'en n'avez plus besoin ? Et d'abord, pourquoi est-il nécessaire de libérer cette mémoire ? Prenons à nouveau un exemple concret. Imaginez que dans un programme, dans un bloc, je déclare une variable, que j'utilise dans le bloc. Ce que je fais ici évidemment est une allocation statique. Puisque je suis en train de déclarer une variable, donc au moment de la compilation, je peux savoir quels sont mes besoins en mémoire. Nous savons, en vertu de la notion de portée, que cette variable n'est plus accessible en dehors du bloc. Donc, au moment où on exécute cet ensemble d'instructions, et qu'on atteint la fin du bloc, eh bien on sait que cette variable v ne sera plus jamais utilisée. Et donc, les variables statiques ont la particularité d'être désallouées automatiquement. Vous n'avez pas à vous préoccuper de ce qui va advenir de l'emplacement mémoire qui, ici, n'est plus accessible. Prenons un exemple analogue, mais où, cette fois, on fait de l'allocation dynamique. Donc, ici, imaginez que, par exemple, j'ai déclaré un pointeur px. Ensuite, dans un bloc, je décide d'allouer dynamiquement de la mémoire que je récupère dans le pointeur px. Donc, ce qu'il faut savoir, c'est que cette zone mémoire que vous avez allouée dynamiquement ici, n'est pas détruite, n'est pas libérée, tant que vous n'avez pas décidé de le faire. Donc, lorsqu'on atteint l'exécution de la fin du bloc ici, cette zone mémoire reste parfaitement disponible tant que vous ne l'avez pas désallouée. C'est ce qui fait en sorte que l'allocation dynamique permette de créer des zones mémoires, qui vivent au-delà de la portée dans laquelle elles ont été créées justement. Et donc si on veut, à un moment donné, restituer cet emplacement, c'est à nous de le

notes

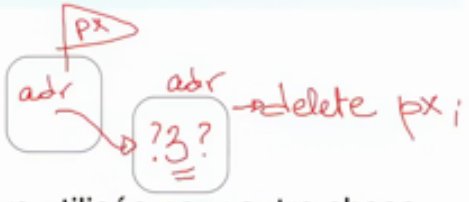
résumé

3m 49s

## Libération de la mémoire allouée

EPFL

```
delete pointeur;
```




libère la zone mémoire allouée au pointeur *pointeur*.

C'est-à-dire que cette zone mémoire peut maintenant être utilisée pour autre chose.
Il **ne** faut **plus y accéder !**...

Conseil – Bonnes pratiques

*faire suivre tous les* `delete` *de l'instruction «* `pointeur = nullptr;` *»)*

Conseil – Bonnes pratiques

*Toute zone mémoire allouée par un* `new` *doit impérativement être libérée par un* `delete` *correspondant !*

faire explicitement et à ce moment-là il va falloir avoir recours à l'opérateur delete, donc par exemple, ici, je peux décider après le bloc, car je n'ai plus besoin de la zone mémoire en question, de restituer l'emplacement mémoire en faisant un delete sur le pointeur associé. Très concrètement, imaginez que dans un programme, vous ayez un pointeur px qui pointe sur une zone mémoire dynamiquement allouée. Imaginons que cette zone mémoire contienne 3, donc bien évidemment, px contient l'adresse de cette zone mémoire. Que se passe-t-il lorsqu'on fait une désallocation de mémoire en désallouant l'emplacement mémoire associé à px par cette instruction-là ? Rappelez-vous la petite analogie avec les maisons sur un terrain, eh bien lorsqu'on fait un delete, on détruit la maison. Ce qui veut dire concrètement que le contenu de cette zone mémoire n'est plus prévisible, plus utilisable de façon sûre par le programme. Votre 3 peut parfaitement y rester un petit moment,

notes

résumé

mais vous ne pouvez plus savoir de façon sûre et prévisible quel va être le contenu de cette zone mémoire. Donc on voit ici qu'en se contentant d'un delete px, on est dans la situation où la zone mémoire ici n'a plus un contenu utilisable. Par contre, bien sûr, l'adresse existe toujours en mémoire. Et cette adresse est toujours stockée à ce stade dans le pointeur px. Donc, ça justifie une des bonnes pratiques que je vous conseille absolument de suivre c'est qu'il faut faire suivre toute instruction delete d'une instruction "pointeur = nullptr;". Ce qui va se traduire par quelque chose de cette nature : "px = nullptr;", donc ici, vous êtes en train de dire explicitement que, désormais, px ne pointe vers plus rien de valable et donc vous ne gardez pas, à l'intérieur de px, une adresse qui ne veut plus rien dire pour le programme, qui n'est plus utilisable de façon sûre pour le programme. pour le programme.

notes

résumé

6m 37s