

Support de cours

Cours:

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

Vidéo:

W14-03-2-heritageconstr-CPP-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Ordre inverse des constructeurs. Constructeur de copies. Dernier constructeur. Constructeur de copies d'une sous-classe. Grand-chose. Relation d'héritage. Nouvelle norme c. Petites choses. Coordonnées x. Premier destructeur. Position nulle. Vecteurs mathématiques. Anecdote de cette erreur. Constructeur de copies de la classe. Appel de la copie.



[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/>

Héritage : constructeurs (2)

(Partie 2)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





Pour les destructeurs, pas grand-chose à dire

notes

résumé

0m 1s



Ordre d'appel des destructeurs

Les destructeurs sont toujours appelés dans l'ordre inverse (/symétrique) des constructeurs.

Par exemple dans l'exemple précédent, lors de la destruction d'un `C`, on aura appel et exécution de :

- ▶ `C::~~C()`
- ▶ `B::~~B()`
- ▶ `A::~~A()`

(et dans cet ordre)

(puisque les constructeurs avaient été appelés dans l'ordre

- ▶ `A::A()`
- ▶ `B::B()`
- ▶ `C::C()`

)

si ce n'est qu'ils sont appelés dans l'ordre inverse des constructeurs. Les destructeurs seront toujours appelés dans l'ordre inverse des constructeurs. Donc dans l'exemple qui précède,

notes

résumé

0m 5s



Le constructeur de copie d'une sous-classe doit invoquer explicitement le constructeur **de copie** de la super-classe

☞ Sinon c'est le constructeur **par défaut** de la super-classe qui est appelé !

Exemple :

```
Rectangle(Rectangle const& autre)
: FigureGeometrique(autre),
  largeur(autre.largeur),
  hauteur(autre.hauteur)
{ }
```



nous avons d'abord appel au destructeur de « C », puisque c'est le dernier constructeur qui a été appelé, c'était le constructeur de « C », donc le premier destructeur qui est appelé, c'est le destructeur de « C », lequel destructeur de « C » appelle le destructeur de « B », lequel destructeur de « B » appelle le destructeur de « A ». Enfin, encore deux petites choses sur lesquelles nous pensons que ça vaut la peine d'attirer votre attention pour conclure cette séquence sur les constructeurs et destructeurs dans la relation d'héritage. Le premier concerne le constructeur de copies. Si jamais vous devez redéfinir le constructeur de copies, vous n'êtes pas obligé dans la plupart des cas, mais si vous pensez devoir le faire, alors faites très attention, dans le constructeur de copies d'une sous-classe à toujours penser

notes

résumé

0m 15s



Le constructeur de copie d'une sous-classe doit invoquer **explicitement** le constructeur **de copie** de la super-classe

⚠ Sinon c'est le constructeur **par défaut** de la super-classe qui est appelé !

Exemple :

```
Rectangle(Rectangle const& autre)
: FigureGeometrique(autre),
  largeur(autre.largeur),
  hauteur(autre.hauteur)
{ }
```

FigureGeometrique(1.0, 2.0, 3.5, 4.6);
 Rectangle r2(r1);
 FigureGeometrique(0.0, 0.0)
 FigureGeometrique(1.0, 2.0, 3.5, 4.6);
 Position 0
 Rectangle

à appeler explicitement le constructeur de copies de la super-classe. Par exemple, supposons que l'on ait une classe « Rectangle » qui hérite d'une classe « FigureGeometrique », laquelle classe « FigureGeometrique » a un attribut (de type) « Position », qui pourrait prendre deux coordonnées x et y, et a donc par exemple deux constructeurs, un constructeur par défaut, qui initialise la position à la position nulle (0,0) et un constructeur qui permettrait d'initialiser les coordonnées (x,y) de la position. Et supposons donc que l'on crée une instance « r1 » de la classe « Rectangle », que l'on initialise avec une position, disons première coordonnée (1), deuxième coordonnée (2) et puis une largeur de 3.5 et une hauteur de 4.6, et que l'on crée une instance « r2 » par copie de « r1 ». On s'attend donc bien sûr à ce que la copie « r2 », copie de « r1 », ait la même position, même hauteur, même largeur que « r1 ». Mais admettons que l'on ait écrit un constructeur de copies comme ceci, et que l'on n'ait pas écrit explicitement l'appel au constructeur de copies de la super-classe. Qu'est ce qui se passe à ce moment-là dans cet appel de la copie de « r1 » dans « r2 » ? On va appeler le constructeur de copies de la classe « Rectangle ». Ce constructeur de copies de la classe « Rectangle », dans sa liste d'initialisation, n'a pas d'appel explicite à aucun constructeur de la superclasse et donc appelle le constructeur par défaut de la superclasse. Ce qu'on commence par faire dans la copie de « Rectangle », c'est initialiser la position de « r2 » à 0. Donc « r2 » va se retrouver avec une position à 0 par l'appel non écrit, et donc réalisé automatiquement, du constructeur par défaut de la superclasse. Puis on copiera la largeur de «

notes

résumé

1m 1s



Le constructeur de copie d'une sous-classe doit invoquer **explicitement** le constructeur **de copie** de la super-classe

☞ Sinon c'est le constructeur **par défaut** de la super-classe qui est appelé !

Exemple :

```
Rectangle(Rectangle const& autre)
: FigureGeometrique(autre),
  largeur(autre.largeur),
  hauteur(autre.hauteur)
{ }
```

FigureGeometrique(1.0, 2.0, 3.5, 4.6);
 Rectangle r2(r1);
 FigureGeometrique(0.0, 0.0)
 FigureGeometrique(Position 0)
 Rectangle

r1 » dans la largeur de « r2 », la hauteur de « r1 » dans la hauteur de « r2 ». Donc si on a omis cette ligne, alors « r2 » aura comme position « 0.0 , 0.0 ». Ce qu'il faut faire, c'est avoir explicitement un appel au constructeur de copies de la super-classe. A ce moment-là, ce qui va se passer, c'est que l'appel du constructeur de copies va appeler explicitement le constructeur de copies de la super-classe. Si on n'a pas écrit explicitement, on va avoir le constructeur de copies par défaut, qui va donc recopier « r1 » en tant que FigureGeometrique, je vous rappelle que le constructeur de copies de « FigureGeometrique » a le prototype suivant et donc passe une FigureGeometrique par référence constante. Ici « autre », qui est un Rectangle, est passé en tant que FigureGeometrique, ce qui pose absolument aucun problème, puisqu'un Rectangle est une FigureGeometrique. Et donc ce constructeur de copies, recopie la partie « FigureGeometrique » du Rectangle « r1 », c'est-à-dire va recopier son attribut « position » dans la position de « r2 ». Donc on aura bien ici une copie de la partie « FigureGeometrique » de « r1 », c'est-à-dire sa position « 1.0 , 2.0 » dans la partie « FigureGeometrique » de « r2 » c'est à dire dans la position héritée par « FigureGeometrique » de « r2 ».

notes

résumé

Le constructeur de copie d'une sous-classe doit invoquer **explicitement** le constructeur **de copie** de la super-classe

☞ Sinon c'est le constructeur par défaut de la super-classe qui est appelé !

Exemple :

```
Rectangle(Rectangle const& autre)
: FigureGeometrique(autre),
  largeur(autre.largeur),
  hauteur(autre.hauteur)
{ }
```

Vecteur (

↑

VecteurUnitaire

Grâce à cet appel explicite au constructeur de copies de la superclasse, on a bien « r2 » qui aura la même position que « r1 ». Et ensuite le constructeur de copies continue comme précédemment, en copiant la largeur et la hauteur. Donc il faut bien explicitement écrire l'appel au constructeur de copies de « FigureGeometrique », parce que sinon ce sera le constructeur par défaut de la super-classe qui sera appelé et ceci peut créer quelques ennuis. Je peux vous livrer une anecdote de cette erreur qui s'est passée dans des projets que je donne à mes étudiants. Ils devaient écrire des classes pour des vecteurs, des vecteurs mathématiques, des vecteurs algébriques, et ils avaient fait une sous-classe, du genre « vecteur unitaire ». Et certains s'étaient crus malins de devoir écrire le constructeur de copies des vecteurs unitaires qui héritent de la classe « Vecteur ». Et bien simplement, ils avaient oublié, dans le constructeur de copies, de faire l'appel au constructeur de copies de la super-classe « Vecteur ». Or il se trouve que la super-classe « Vecteur » avait un constructeur par défaut,

notes

résumé

4m 25s





qui construisait le vecteur nul. Et donc ils se retrouvaient, chaque fois qu'ils faisaient des copies de vecteurs unitaires, avec des vecteurs unitaires qui étaient nuls, ce qui est quand même un comble pour un vecteur unitaire. Donc évitez ce genre d'erreur, et si vous pensez devoir redéfinir le constructeur de copies dans une sous-classe, n'oubliez pas d'appeler le constructeur de copies de la super-classe.

notes

résumé

5m 25s



Les constructeurs ne sont, en général, **pas hérités** mais en C++11 on peut **demandeur leur héritage** en utilisant le mot clé « **using** ».

On récupère alors **tous** les constructeurs de la super-classe, i.e. on peut construire la sous-classe avec les mêmes arguments, mais...



Attention ! ces constructeurs n'initialisent donc **pas** les **attributs spécifiques de la sous-classe**.

C'est donc **très risqué**, et je vous conseille de ne l'utiliser que pour des sous-classes n'ayant **pas de nouvel attribut** (et si c'est approprié) !

Exemple :

```
class A {  
public:  
    A(int);  
    A(double, double);  
    // ...  
};
```

```
class B : public A {  
using A: A;  
/* existent alors maintenant  
    B::B(int)  
    et B::B(double, double) */  
};
```

Pour finir, je voudrais revenir sur un aspect qui a changé avec C++ 2011. Je vous ai dit en effet, que les constructeurs n'étaient pas hérités dans la relation d'héritage, une sous-classe héritait l'ensemble des attributs et des méthodes de sa super-classe, mais n'héritait pas le constructeur et les destructeurs. En C++ 2011, ils ont introduit ce qu'on appelle « l'héritage des constructeurs ». Il est donc maintenant possible de demander l'héritage d'un constructeur. Par défaut, on n'a pas d'héritage des constructeurs, mais on peut rajouter, avec une syntaxe comme ceci, le mot-clé « using », on peut rajouter tous les constructeurs de la super-classe. Et ainsi, on pourra construire la sous-classe avec tous les constructeurs et avec les mêmes arguments que les constructeurs de la super-classe. Mais faites très attention, ceci est à mon avis quelque chose de très risqué et je vous déconseille de faire ce genre de chose, de faire de l'héritage des constructeurs, parce que les constructeurs des super-classes, évidemment, n'initialisent pas les attributs de la sous-classe, et donc si on fait un appel à un constructeur de la super-classe dans la sous-classe, qui serait possible avec cet héritage des constructeurs, on n'aurait alors à ce moment-là pas d'initialisation des attributs propres de la sous-classe, c'est donc quelque chose que je vous déconseille de faire et que je ne vous conseille de faire que lorsque vous n'avez pas de nouveaux attributs dans la sous-classe, et évidemment si c'est approprié. Donc par exemple, supposons qu'on ait une super-classe « A », qui a deux constructeurs : un constructeur qui prend un « entier », et un constructeur qui prend deux « double ». Et on a une sous-classe « B » qui hérite de la super-classe « A ». On peut donc faire, à la demande, hériter les constructeurs de « A » dans la classe « B » en écrivant comme ceci « using A :

notes

résumé

5m 48s



Les constructeurs ne sont, en général, **pas hérités** mais en C++11 on peut *demandeur leur héritage* en utilisant le mot clé « **using** ».

On récupère alors **tous** les constructeurs de la super-classe, i.e. on peut construire la sous-classe avec les mêmes arguments, mais...



Attention ! ces constructeurs n'initialisent donc **pas** les **attributs spécifiques de la sous-classe**.

C'est donc **très risqué**, et je vous conseille de ne l'utiliser que pour des sous-classes n'ayant *pas de nouvel attribut* (et si c'est approprié) !

Exemple :

```
class A {  
public:  
    A(int);  
    A(double, double);  
    // ...  
};
```

```
class B : public A {  
using A: A;  
/* existent alors maintenant  
    B::B(int)  
    et B::B(double, double) */  
};
```

: A », c'est à dire en utilisant

notes

résumé



les constructeurs qui s'appellent « A » de la classe « A ». La syntaxe est peut-être un peu redondante mais c'est comme ça qu'il faut écrire, « using A : A ». Et alors à moment-là, on récupère dans « B », deux constructeurs : un constructeur avec un paramètre de type « entier » et un constructeur avec deux paramètres de types « double, double ». Mais si jamais, dans « B », on avait introduit des attributs supplémentaires, alors attention, les constructeurs en question, comme ce sont les constructeurs de « A », ne vont évidemment pas initialiser l'attribut « B ». C'est donc pour ça que je vous recommande d'utiliser ce genre de chose avec beaucoup de parcimonie, voire de l'oublier. Mais nous tenions à vous le signaler, parce que quand même c'est quelque chose qui a été introduit dans cette nouvelle norme C++ 2011 et que vous pourriez être amenés à rencontrer. Ceci conclut donc notre série de séquences vidéos sur l'héritage.

notes

résumé

7m 37s

