

Support de cours

Cours:

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

Vidéo:

W11-03-publicprivate-CPP-pt3

Concepts (extraits des sous-titres générés automatiquement) :

Premier temps. Programmeur utilisateur de la classe. Variable i. Lignes de code. Notions d'encapsulation. Exemple des attributs. Nom du paramètre. Portée de cette variable. Méthodes de ces attributs. Premier cours d'introduction. Problème d'une telle écriture. Gros programmes. Utilisation de l'interface. Bonne programmation. Ensemble des lignes.



[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/>

public: **et** private: (Partie 3)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



« Accesseurs », « manipulateurs » et encapsulation

Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en **public** ?

```
class Rectangle
{
public:
    double largeur;
    double hauteur;
    string label;
};
```



Une question que me posent souvent mes étudiants, c'est :

notes

résumé

0m 1s



« Accesseurs », « manipulateurs » et encapsulation

Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en **public** ?

```
class Rectangle
{
public:
    double largeur;
    double hauteur;
    string label;
};
```

mais dans ce cas ...

```
Rectangle rect;
rect.hauteur = -36;
cout << rect.label.size() << endl;
```

« Mais, Monsieur, pourquoi faire tout ceci, c'est trop compliqué, » « pourquoi pas simplement tout mettre en public, et auquel cas on pourrait » « accéder directement aux différents champs, on n'aurait pas besoin de mettre » « toutes ces méthodes get/set ; et surtout on n'aurait pas besoin » « de réfléchir auxquelles mettre et quelles méthodes bien choisir dans l'interface ? » Mais si on fait ceci -- et supposons donc par exemple qu'on ait aussi un label qui donne un nom, par exemple, au Rectangle ou qui est un outil dont on a besoin en interne --, dans ce cas, on pourrait tout à fait déclarer des instances comme ça, « rect », de la classe « Rectangle » et directement écrire par exemple « rect.hauteur = -36 », mettre donc des hauteurs qui ne sont pas vérifiées, qui ne sont pas valides. Ici on n'a absolument aucun contrôle sur ce que fait un autre programmeur utilisateur de la classe par rapport à la représentation de cette classe. Alors si l'on obligeait les programmeurs utilisateurs de notre classe à passer par des méthodes comme par exemple ici une méthode « setHauteur », on aurait toute latitude pour vérifier les données que l'on a reçues, pour garantir l'intégrité de nos données, l'intégrité de nos Rectangles. Donc par exemple, ici avec un test sur la valeur reçue, par exemple si elle est inférieure ou égale à 0 alors on générerait l'erreur, ici avec un message d'erreur, ou peu importe... Et puis donc si on a reçu quelque chose de correct, alors seulement dans ce cas-là, on peut transmettre la valeur reçue « h » à notre hauteur, ce qui garantit que notre Rectangle a toujours une hauteur qui fait sens. Pour prendre un autre exemple, on aurait donc aussi accès à tous les attributs, par exemple des attributs plus sophistiqués comme le label ici, et donc on

notes

résumé

0m 5s



Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en **public** ?

```
class Rectangle
{
public:
    double largeur;
    double hauteur;
    string label;
};
```

mais dans ce cas ...

```
Rectangle rect;
rect.hauteur = -36;
cout << rect.label.size() << endl;
```

aurait accès à toutes les méthodes de ces attributs, donc par exemple ici à la méthode « size » de ce label. Et puis donc imaginez que vous avez déjà écrit 300 lignes de code comme ça, qui utilisent le label de Rectangle, et puis qu'un jour le programmeur de la classe Rectangle décide

notes

résumé



de changer ce label, donc jeter vos 300 lignes de codes pour réécrire d'autres lignes. Alors que si vous vous limitez à l'utilisation de l'interface, ce qui a été spécifiée au préalable et qui ne doit pas être changée, si vous utilisez en externe l'interface qui a été spécifiée, et que donc le programmeur de la classe, lui, décide de changer ces labels, -- libre à lui ! -- c'est lui qui devra tout reprogrammer dans sa classe, vous, comme vous n'utilisez que l'interface, vous ne verrez rien d'autre à modifier, vous pourrez donc continuer à utiliser la classe, avec vos lignes de codes que vous avez écrites jusqu'ici.

notes

résumé

2m 13s



masquage = un identificateur « cache » un autre identificateur

```
int main() {  
    int i(120);  
  
    for (int i(1); i < MAX; ++i) {  
        cout << i << endl;  
    }  
  
    cout << i << endl;  
    return 0;  
}
```



Ces notions, ces contraintes d'orientée-objet ces notions d'encapsulation, d'abstraction, de limitation de l'interface, prennent donc vraiment du sens quand on écrit des gros programmes, quand on partage du code au sein d'une équipe, c'est d'ailleurs dans ce contexte-là qu'elles ont été développées et qu'elles ont été inventées et mises en place. Mais c'est comme ceci que l'on fait de la bonne programmation, que l'on écrit des programmes que l'on peut facilement faire évoluer, c'est-à-dire des programmes modulaires. Donc même si dans un premier temps ça vous semble un peu fastidieux sur des petits programmes qui sont développés dans le cadre d'exercices simples, pensez à cette pratique en vue d'une programmation plus large et d'une bonne programmation. Terminons cette séquence par un aspect technique

notes

résumé

2m 47s



masquage = un identificateur « cache » un autre identificateur

```
int main() {  
    int i(120);  
  
    for (int i(1); i < MAX; ++i) {  
        cout << i << endl;  
    }  
  
    cout << i << endl;  
    return 0;  
}
```

Situation typique en POO : un paramètre cache un attribut

```
void setHauteur(double hauteur) {  
    hauteur = hauteur; // Hmm.... pas terrible !  
}
```

qui génère souvent beaucoup de questions auprès des étudiants, et qui découle de ce l'on appelle le « masquage ». Le masquage, c'est lorsqu'un identificateur, un nom, cache, masque, un autre nom. Typiquement, on avait vu ça dans le premier cours d'introduction à la programmation, avec le masquage ici d'une variable `i` qui est masquée par une autre variable `i` locale. Je vous rappelle que la portée de cette variable ici c'est l'ensemble des lignes couvertes par le « for » et donc la portée de ce `i` ici c'était le « for » et que le `i` en question ici, c'est bien le `i` du « for » qui masquait le `i` bleu qui existe toujours mais qui dans la partie rouge, était caché ; ici c'est bien le `i` bleu, qui était caché dans la partie rouge du « for » qui était masquée par ce `i`-là. Il se produit la même chose lorsque l'on déclare des méthodes. Par exemple, supposons que l'on ait voulu déclarer la méthode « SetHauteur » qui reçoit ici une valeur pour la hauteur ; et donc on veut indiquer par un nom bien choisi que c'est une valeur pour la hauteur. Et donc le programmeur ici déclare la hauteur comme étant aussi le nom du paramètre. Donc comment on écrirait cette fois-ci le corps de la méthode de `setHauteur`,

notes

résumé

3m 25s



« Accesseurs » et « manipulateurs »

Si le programmeur *le juge utile*, il **inclut les méthodes publiques nécessaires** ...

1. Accesseurs (« méthodes get » ou « getters ») :

- Consultation (i.e. « prédicat »)
- Retour de la valeur d'une variable d'instance précise

```
double getHauteur() const { return hauteur; }
double getLargeur() const { return largeur; }
```

2. Manipulateurs (« méthodes set » ou « setters ») :

- Modification (i.e. « action »)
- Affectation de l'argument à une variable d'instance précise

```
void setHauteur(double h) { hauteur = h; }
void setLargeur(double l) { largeur = l; }
```

on écrirait « hauteur », la hauteur du Rectangle, c'est bien l'attribut hauteur, égal, ici la hauteur reçue en paramètre, donc cette hauteur ici, c'est la même que celle reçue en paramètre. Le problème d'une telle écriture, bien entendu, c'est que le mot « hauteur » désigne deux choses : il désigne à la fois l'attribut et à la fois le paramètre, que l'on a reçu ici. Ce n'est pas terrible pour le compilateur qui va vous insulter avec un message d'erreur en disant, je ne comprends pas ce que vous avez voulu écrire.

notes

résumé

4m 37s





Dans ce cas-là, ce que l'on peut faire, c'est d'abord, simplement changer le nom de l'attribut, et donc avoir un nom qui n'est pas ambigu, comme nous l'avons fait dans un transparent qui précède,

notes

résumé

5m 12s



Masquage et pointeur `this`

Si, dans une méthode, un attribut est **masqué** alors la valeur de l'attribut peut quand même être référencée à l'aide du mot réservé `this`.

`this` est un **pointeur sur l'instance courante**

`this` \simeq « mon adresse »

Syntaxe pour spécifier un attribut *en cas d'ambiguïté* :

`this->nom_attribut`

Exemple :

```
void setHauteur(double hauteur) {
    this->hauteur = hauteur; // Ah, là ça marche !
}
```

$A \rightarrow B$

$(*A).B$

L'utilisation de `this` est obligatoire dans les situations de **masquage** (mais évitez ces situations !)

mais l'on peut aussi désambiguïser le nom en disant que l'on veut la hauteur du Rectangle, la hauteur de cette instance, en utilisant un nom particulier qui est le pointeur « `this` » qui veut dire un pointeur sur l'instance courante. Donc « `this` », dès qu'on l'écrit, c'est un pointeur sur l'instance courante qui veut, en gros, dire « mon adresse ». Par exemple, pour un Rectangle `r1` dans une méthode `f` de `r1`, dans cette méthode, « `this` » vaut l'adresse de `r1`. Si on l'appelait sur une autre instance `r2`, « `this` », à ce moment-là, dans la méthode appelée sur l'instance `r2`, vaudrait l'adresse de `r2`. Donc « `this` » sert bien à représenter dans une méthode, l'adresse de l'instance courante. Par exemple, on pourrait écrire comme ceci : `this->hauteur` (this flèche hauteur) Je vous indique ici, puisque c'est la première fois qu'on le rencontre, que la syntaxe « flèche » comme ceci, $A \rightarrow B$ veut simplement dire $*A.B$ (étoile A point B) Si A est un pointeur, donc $*A$ est l'objet pointé et l'on prend le champ B de l'objet pointé par A . Ceci est un peu technique et n'a à mon avis pas besoin d'être retenu. Retenez peut-être la syntaxe et donc si on a le pointeur « `this` », `this->` va permettre d'accéder aux méthodes et ici aux attributs

notes

résumé

5m 23s





de l'objet pointé par « `this` », mais comme « `this` » c'est mon adresse, c'est-à-dire mes attributs. Ici `this->hauteur`, ce sera bien la hauteur de l'instance courante, ce qui permet de garder le mot hauteur tout seul sans rien devant comme le nom ici du paramètre. L'utilisation du mot-clé « `this` » est absolument nécessaire dans les cas de masquage.

notes

résumé

6m 49s



La portée des attributs dans la définition des méthodes est résumée par le schéma suivant :

```
class MaClasse {  
private:  
    int x;  
    int y;  
public:  
    void une_methode( int x ) {  
        ... y ...  
        ... x ...  
        ... this->x ...  
    }  
};
```

Ceci-dit, je vous recommande d'éviter les cas de masquage et donc de donner, tout en restant significatifs, des noms différents aux attributs et aux paramètres des méthodes qui auraient besoin de toucher à ces attributs.

notes

résumé

7m 14s



La portée des attributs dans la définition des méthodes est résumée par le schéma suivant :

```
class MaClasse {  
private:  
    int x;  
    int y;  
public:  
    void une_methode( int x ) {  
        ... y ...  
        ... x ...  
        ... this->x ...  
    }  
};
```



Pour conclure, résumons ce que nous avons appris concernant les droits d'accès et les portées des attributs et des méthodes de classes. Pour les droits d'accès, nous avons les deux mots-clés

notes

résumé

7m 28s



La portée des attributs dans la définition des méthodes est résumée par le schéma suivant :

```
class MaClasse {  
private:  
    int x;  
    int y;  
public:  
    void une_methode( int x ) {  
        ... y ...  
        ... x ...  
        ... this->x ...  
    }  
};
```

« private » et « public » qui opèrent sur l'ensemble des lignes de codes qui les suivent, « private », pour ce qui est de la partie strictement privée, c'est-à-dire l'ensemble des attributs des méthodes qui ne sont accessibles que depuis la classe elle-même, et « public » pour l'ensemble des méthodes que l'on souhaite offrir accessibles de n'importe où dans le code, dans le « main » ou dans d'autres classes. Et puis si l'on déclare ici des attributs x et y « private », ils sont accessibles n'importe où dans la classe, en particulier dans les méthodes, donc par exemple, en supposant qu'on n'ait pas déclaré d'autres y ailleurs, y référera bien sûr le y de l'instance courante. Par contre on peut avoir un phénomène de masquage lorsque l'on déclare une variable comme ceci, typiquement, un paramètre d'une méthode, qui va donc faire que, d'après les règles de résolution de portée vues dans le cours d'introduction à la programmation, va faire que ce nom se résout ici à la portée la plus proche, c'est-à-dire, au nom du paramètre, et si on souhaite pouvoir accéder à l'attribut x ici, à ce moment-là il va falloir, entre guillemets, « démasquer » ce nom, en rajoutant le pointeur « this » pour accéder à x. Ceci-dit, je ne vous recommande pas d'utiliser cette façon de faire, je ne vous recommande pas d'utiliser des pointeurs « this », mais plutôt de choisir des noms différents et pertinents, différents entre les attributs et les paramètres de vos méthodes. paramètres de vos méthodes.

notes

résumé

7m 38s

