

Support de cours

Cours:

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

Vidéo:

W11-02-progclasses-CPP-pt5

Concepts (extraits des sous-titres générés automatiquement) :

Meilleure lisibilité. Méthode surface. Nom de la méthode. Meilleure modularisation. Extérieur de la classe. Meilleure séparation. Définition de la méthode. Prototype de la méthode surface. Méthodes. Mot const. Syntaxe d'utilisation des attributs. Vectors de double. Classes. Opérateur de résolution de portée. Nom de la classe.



[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/>

Classes, objets, attributs et méthodes en C++ (Partie 5)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





On a vu que les méthodes sont des fonctions un peu particulières

notes

résumé

0m 1s



Définition externe des méthodes

Il est possible d'écrire les définitions des méthodes à l'extérieur de la déclaration de la classe

☞ **meilleure lisibilité du code, modularisation**

Pour relier la définition d'une méthode à la classe pour laquelle elle est définie, il suffit d'utiliser l'opérateur *::* de résolution de portée :

- ▶ La déclaration de la classe contient les *prototypes des méthodes*
- ▶ les définitions correspondantes spécifiées à l'extérieur de la déclaration de la classe se font sous la forme :

```
typeRetour NomClasse::nomFonction(type1 param1
                                , type2 param2
                                , ...)
{ ... }
```

que l'on met dans les classes, y compris avec leurs définitions. Mais, ceci rend souvent le code peu lisible. On n'a par exemple pas une vue synthétique de l'ensemble des prototypes des méthodes offertes par une classe.

notes

résumé

0m 5s





Pour cette raison, une meilleure lisibilité, mais aussi pour une meilleure modularisation, c'est-à-dire une meilleure séparation entre d'un côté l'interface et d'un autre côté l'implémentation, une meilleure séparation entre différentes parties, pour toutes ces raisons, il peut être nécessaire de définir les méthodes à l'extérieur de la classe. Cela ne change rien à tout ce que je vous ai dit, c'est simplement une organisation de l'écriture qui est un tout petit peu différente et où l'on va sortir à l'extérieur de la classe la définition de la méthode

notes

résumé

0m 19s



Définition externe des méthodes

Il est possible d'écrire les définitions des méthodes à l'extérieur de la déclaration de la classe

☞ **meilleure lisibilité du code, modularisation**

Pour relier la définition d'une méthode à la classe pour laquelle elle est définie, il suffit d'utiliser l'opérateur *::* de résolution de portée :

- ▶ La déclaration de la classe contient les *prototypes des méthodes*
- ▶ les définitions correspondantes spécifiées à l'extérieur de la déclaration de la classe se font sous la forme :

```
typeRetour NomClasse::nomFonction(type1 param1
                                , type2 param2
                                , ...)
{ ... }
```

et en gardant le prototype dans la classe. Simplement, si on sort la définition à l'extérieur de la classe, il faut bien indiquer au compilateur que c'est bien d'une méthode, d'une certaine classe, et non pas une fonction

notes

résumé

0m 49s



```
class Rectangle {  
    // ...  
    double surface(); // prototype  
};  
  
// définition  
double Rectangle::surface()  
{  
    return hauteur * largeur;  
}
```

et donc dans le nom de la méthode, on va devoir indiquer à quelle classe elle se rattache en rajoutant devant le nom de la méthode, le nom de la classe suivi de ce que l'on appelle « l'opérateur de résolution de portée :: »

notes

résumé

1m 2s



```
class Rectangle {  
    // ...  
    double surface(); // prototype  
};  
  
// définition  
double Rectangle::surface()  
{  
    return hauteur * largeur;  
}
```

Par exemple, dans notre classe « Rectangle », nous avons déclaré puis défini la méthode surface. Si l'on souhaite la définir à l'extérieur, ce que l'on va garder dans la classe « Rectangle », c'est le prototype de la méthode surface et puis, la définition à l'extérieur

notes

résumé

1m 17s





fera précéder le nom surface du nom de la classe ici suivi de « : : » On écrira donc ici la définition, mais simplement la même que précédemment, mais simplement en faisant précéder le nom surface de ce nom de classe.

notes

résumé

1m 37s



En C++, on peut distinguer les méthodes qui *modifient* l'état de l'objet (« **actions** ») de celles qui *ne changent rien* à l'objet (« **prédicats** »).

On peut pour cela ajouter le mot **const** après la liste des paramètres de la méthode :

```
type_retour nom_methode (type_param1 nom_param1, ...) const
```

Exemple :

```
class Rectangle {  
    // ...  
    double surface() const;  
};  
  
→ double Rectangle::surface() const  
{  
    return hauteur * largeur;  
}
```

Passons maintenant à un autre aspect des méthodes. On distingue deux sortes de méthodes : des méthodes qui peuvent modifier les instances auxquelles elles s'appliquent comme par exemple, tout à l'heure, la méthode « colorie » de la classe « FigureColoree » qui modifiait, « coloriait », l'instance à laquelle elle s'appliquait. De telles méthodes, c'est ce que l'on appelle des « actions ». On les distingue des méthodes qui ne modifient pas les instances auxquelles elles s'appliquent comme par exemple la méthode surface de la classe « Rectangle ». Calculer la surface d'un rectangle ne va pas modifier le rectangle dont on calcule la surface. Ces méthodes s'appellent des « prédicats ». Pour les prédicats, il va être bon de l'indiquer au compilateur et aux autres programmeurs qui vont lire votre code en rajoutant le mot « const » après la liste des paramètres, après ce que l'on appelle l'entête, tout à la fin, vous rajoutez le mot const pour les prédicats, pour dire qu'elle ne modifie pas les instances auxquelles elles s'appliquent. Par exemple, dans la classe « Rectangle », on définirait donc ici le prototype de la méthode surface en rajoutant derrière à la fin, ici, « const » et ce mot va bien sûr se remettre dans l'entête lorsque l'on va faire la définition de la méthode surface, ici, donc, en rajoutant ce mot aussi dans l'entête de cette définition. On voit bien en effet, ici que cette méthode surface ne modifie absolument pas ni la hauteur, ni la largeur. Elle ne fait que utiliser la hauteur et la largeur,

notes

résumé

1m 53s





elle ne modifie pas l'instance auquel elle s'applique.

notes

résumé

3m 25s



En C++, on peut distinguer les méthodes qui *modifient* l'état de l'objet (« **actions** ») de celles qui *ne changent rien* à l'objet (« **prédicats** »).

On peut pour cela ajouter le mot `const` **après** la liste des paramètres de la méthode :

```
type_retour nom_methode (type_param1 nom_param1, ...) const
```

Si vous déclarez une action en tant que prédicat (`const`), vous aurez à la compilation le message d'erreur :

```
assignment of data-member '...' in read-only structure
```

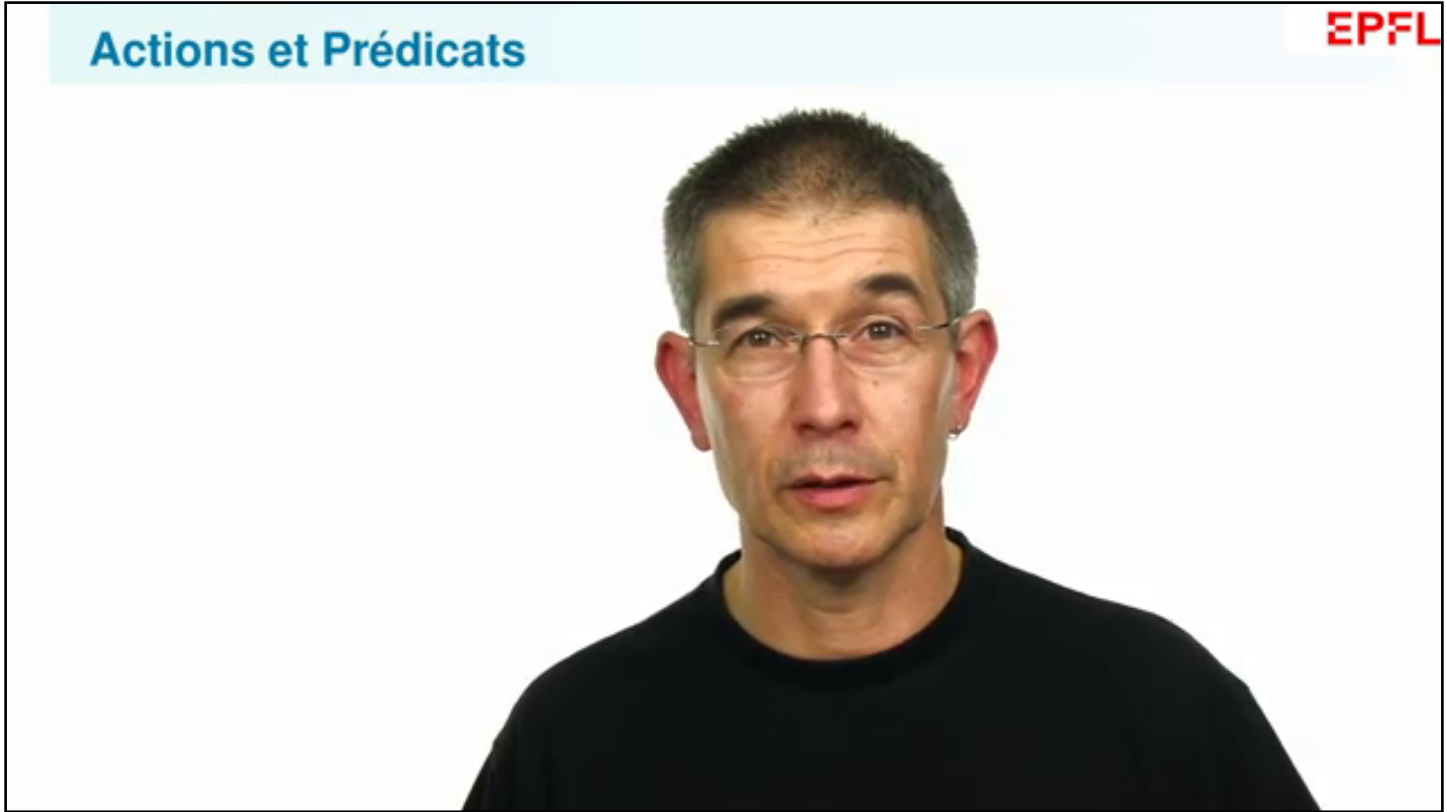
Si jamais vous déclarez une méthode avec le mot-clé « `const` », c'est-à-dire vous déclarez un prédicat mais qu'en fait, la définition de cette méthode modifie l'instance, c'est-à-dire qu'en fait c'est une action,

notes

résumé

3m 28s





alors vous aurez de la part du compilateur, un message qui dira « assignment of data-member » et celui sur lequel vous avez fait la modification « in read-only structure ». Ici, le « read-only » veut bien dire qu'on considère qu'au travers de cette méthode vous ne pouvez pas modifier l'attribut en question, on peut simplement « le lire » mais on ne peut pas le modifier, l'écrire. Dans ce cas-là, deux solutions : soit vous avez fait une erreur effectivement de conception et il faut enlever le mot « const », soit vous voudriez que ce soit vraiment un prédicat et corrigez donc votre erreur. Vous avez fait à ce moment-là une erreur dans l'écriture de la définition de cette méthode.

notes

résumé

3m 38s



Appels aux méthodes

L'appel aux méthodes définies pour une instance de nom `nom_instance` se fait à l'aide d'expressions de la forme :

```
nom_instance.nom_methode(val_arg1, ...)
```

Exemple : la méthode

```
void surface() const;
```

définie pour la classe `Rectangle` peut être appelée pour une instance `rect1` de cette classe par :

```
rect1.surface()
```

Autres exemples :

```
une_figure.colorie(rouge);
```

```
i < tableau.size()
```

vector<double> tableau(5);

Voilà vous savez maintenant déclarer, définir les méthodes d'une classe mais comment les utiliser, comment faire appel à ces méthodes ? Pour cela, on va utiliser une syntaxe que vous aviez déjà rencontré par exemple avec les « vectors », avec les « strings » et qui est très similaire à la syntaxe d'utilisation des attributs, c'est simplement le nom de l'instance suivi du nom de la méthode et avec éventuellement entre parenthèses les arguments que l'on va passer à la méthode. Par exemple, pour les vectors, vous utilisiez « `tableau.size()` ». En fait, c'est parce que `vector` de `double` est une classe et que `tableau` est une instance de cette classe et dans cette classe, on a une méthode « `size()` ».

notes

résumé

4m 13s



```
// ...
class Rectangle {
    double hauteur;
    double largeur;
    double surface() const
    { return hauteur * largeur; }
};

int main()
{
    Rectangle rect1;

    rect1.hauteur = 3.0;
    rect1.largeur = 4.0;

    cout << "surface : " << rect1.surface()
        << endl;
    // ...
}
```

Pour, par exemple, appeler la méthode `surface` de la classe « Rectangle » sur une instance « `rect1` » que l'on avait déclaré de la classe « Rectangle », ce que l'on va faire, c'est « `rect1.surface()` ». Le schéma général, c'est bien `nom d'instance.nom de la méthode avec les arguments`. Ici, `surface` ne reçoit pas d'arguments, et donc on met juste les parenthèses collées comme en appel d'une fonction qui ne reçoit pas d'arguments. Voilà à ce stade, notre programme ressemblerait à la chose suivante. On déclare ici une classe « Rectangle », que l'on ferme et l'on oublie pas le point virgule dans laquelle on met un attribut `hauteur`, un attribut `largeur` tous les deux de type `double` et dans lesquels on rajoute une méthode qui ne modifie pas l'instance à laquelle elle s'applique qui s'appelle `surface`, qui ne prend pas de paramètres

notes

résumé

5m 1s





et qui retourne un double et qui calcule la surface comme étant le produit de l'attribut hauteur* l'attribut largeur, et puis donc dans le « main », on déclare ici, une instance « rect1 » de la classe « Rectangle ». On affecte la hauteur de cette instance « rect1.hauteur = 3.0 » on affecte la valeur 4 à la largeur de l'instance « rect1 » et puis, on fait un appel à la méthode surface. Ici, la méthode surface qui ne prend pas d'arguments et qui s'applique à « rect1 ». C'est donc la surface de « rect1 » ici, ça va bien afficher « surface : » et puis la valeur 12 qui est 3x4. A noter encore une fois, comme pour l'exemple du début, qu'en toute rigueur, le code donné ici ne compile pas, il manque toujours le même petit ingrédient présenté dans la prochaine séquence vidéo. présenté dans la prochaine séquence vidéo.

notes

résumé

5m 49s

