

Support de cours

Cours:

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

Vidéo:

W17-02-affichage-CPP-pt3

Concepts (extraits des sous-titres générés automatiquement) :

Prix d'une sous-classe de produit. Méthode prix. Méthode virtuelle pure. Valeur de base. Instances possibles de produits. Comportement polymorphique. Affichage du prix. Destructeur d'un corps. Façon correcte. Paramètre du constructeur. Valeur de base du produit. Sous-classe de produit. Classe produit. Façon polymorphique. Opérateur d'affichage.



[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/>
page 1/12

Etude de cas : affichage polymorphique

(Partie 3)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



Précisons encore le comportement par défaut :

```
virtual void Produit::afficher(ostream& sortie) const {  
    sortie << prix();  
}
```

En effet, le prix d'une sous-classe de produit peut très bien différer de la valeur de base. Procéder à l'affichage du prix en utilisant la valeur de base n'est donc pas correct pour toutes les instances possibles de produits, y compris celle des sous-classes. En effet, pour une sous-classe de produit comme les bracelets, par exemple, on peut parfaitement imaginer que le prix corresponde effectivement à la valeur de base du produit. Par contre, pour les montres, une autre forme de produit, on peut imaginer que le calcul du prix soit plus complexe et que ce soit la somme des prix de tout ses composants. La façon correcte de procéder, ici, est donc d'utiliser, à la place de valeurs, la méthode `prix()`, laquelle pourra bien-sûr elle-même

notes

résumé

0m 1s



Précisons encore le comportement par défaut :

```
virtual void Produit::afficher(ostream& sortie) const {  
    sortie << prix();  
}
```

Produit p = new Mon;*

avoir un comportement polymorphique, et donc calculer le prix d'une montre si le produit affiché est une montre ou le prix d'un bracelet si le produit à afficher est un bracelet. Si l'on reprend notre exemple de départ, celui où l'on déclarait un pointeur sur un produit dans lequel on stockait

notes

résumé

0m 38s



Précisons encore le comportement par défaut :

```
virtual void Produit::afficher(ostream& sortie) const {  
    sortie << prix();  
}
```

Produit p = new Montre (...);
cout << *p << endl;*

l'adresse d'une montre, imaginez maintenant que l'on invoque l'opérateur d'affichage sur l'objet pointé par p, que se passe-t-il concrètement ?

notes

résumé

1m 0s



Affichage polymorphique

- Tous les produits sont « affichables », chacun à sa façon

☞ affichage polymorphique ?

Le plus simple est de faire appel à une méthode polymorphique définie au niveau de la super-classe :

```
class Produit {
public:
    virtual ~Produit() {}
    virtual void afficher(ostream& sortie) const ;
};

// -----
ostream& operator<<(ostream& sortie, Produit const& machin) {
    machin.afficher(sortie);
    return sortie;
}
```

Ceci se traduira par l'appel à l'opérateur d'affichage, que nous avons surchargé pour les produits

notes

résumé

1m 12s



Précisons encore le comportement par défaut :

```
virtual void Produit::afficher(ostream& sortie) const {  
    sortie << prix();  
}
```

Produit p = new Montre (...);
cout << *p << endl;
↳ operator<< (cout, *p);*

lequel va appeler la méthode afficher, sachant, maintenant que cette variable, ce paramètre, contient la référence à une montre.

notes

résumé

1m 25s



Précisons encore le comportement par défaut :

```
virtual void Produit::afficher(ostream& sortie) const {  
    sortie << prix();  
}
```

Produit p = new Montre (...);
cout << *p << endl;
↳ operator<< (cout, *p);*

La méthode afficher, telle que définie dans la classe produit va être invoquée, laquelle invoquera à son tour la méthode de calcul de prix(). Cette méthode est polymorphique, elle va donc automatiquement s'adapter à la nature réelle de l'objet auquel elle s'applique, qui se trouve être une montre, ici. La méthode prix() est, en effet, virtuelle comme on lui permet d'accéder à l'instance réelle au travers d'un pointeur, elle peut s'appliquer réellement de façon polymorphique et on aura le résultat désiré, c'est à dire que le calcul du prix se fera pour une montre comme étant, par exemple

notes

résumé

1m 34s



Supposons enfin que :

- ▶ l'on fixe la valeur de base d'un produit au départ et que l'on ne puisse pas en changer
- ▶ un produit ait par défaut une valeur de base nulle
- ▶ un produit n'existe pas en tant que tel : c'est une *abstraction*

```
class Produit {
public:
    Produit(double une_valeur)
        : valeur(une_valeur)
    {}

    virtual ~Produit() = 0;

    // ...

};

Produit::~~Produit() {}
```

la somme des prix de tous ses composants. Finalisons à présent notre classe produit. Supposons que l'on souhaite imposer le fait que la valeur de base d'un produit reste inchangé une fois qu'on l'a initialisé. Un produit garderait donc, comme valeur de base, la valeur d'initialisation et il ne serait pas possible d'en changer au cours de l'existence du produit. Ceci peut s'imposer en étiquetant l'attribut comme étant constant, l'attribut valeur pourra être initialisé, mais une fois cette initialisation faite, il ne pourra plus changer de valeur. Si l'on souhaite également imposer que par défaut, un produit ait une valeur de base nulle, on peut le faire en utilisant une valeur par défaut pour le paramètre du constructeur, on aurait donc à disposition un constructeur par défaut pour la classe Produit qui, lorsqu'il est invoqué, initialiserait la valeur avec la valeur 0. Enfin, imaginons que l'on souhaite modéliser le fait qu'un produit n'a pas d'existence en tant que tel, il s'agit d'une entité abstraite dans notre conception, on ne veut pas créer d'instance de produit en tant que tel. Comment faire pour imposer cette contrainte dans notre conception ? En C++, pour qu'une classe soit abstraite, il faut qu'elle contienne au moins une méthode virtuelle pure, c'est à dire qu'ici, on devrait avoir une méthode définie comme virtuelle pure. Nous n'avons pas vraiment de raison, ici, d'en créer une, en créer une artificiellement n'aurait pas vraiment de sens. Un bon candidat pour être une méthode virtuelle pure dans une classe est en fait le destructeur. Le destructeur peut être déclaré comme virtuel pur et assurer ainsi le fait que la classe devienne abstraite. Comme toute classe possède nécessairement un destructeur, le fait de les déclarer comme virtuels pur nous épargne d'avoir à créer une méthode artificielle pour le simple fait de vouloir rendre

notes

résumé

2m 1s



Finalisation de la classe Produit

Supposons enfin que :

- ▶ l'on fixe la valeur de base d'un produit au départ et que l'on ne puisse pas en changer
- ▶ un produit ait par défaut une valeur de base nulle
- ▶ un produit n'existe pas en tant que tel : c'est une *abstraction*

```
class Produit {
public:
    Produit(double une_valeur)
        : valeur(une_valeur)
    {}

    virtual ~Produit() = 0;

    // ...

};

Produit::~~Produit() {}
```

la classe en question abstraite. A noter qu'il est impératif de doter tout destructeur d'un corps et vous noterez, à ce sujet, que rien n'empêche une méthode virtuelle pure d'avoir un corps. Cette syntaxe n'est cependant pas licite en C++ donc il n'est pas possible de préciser le corps d'un destructeur virtuel pur à l'intérieur de la classe comme ceci, il faudra plutôt externaliser la définition du destructeur de la classe Produit selon cette syntaxe. C'est à dire qu'à l'intérieur de la classe Produit on déclare le destructeur virtuel pur,

notes

résumé

La classe Produit

```
class Produit {
public:
    Produit(double une_valeur = 0.0) : valeur(une_valeur) {}

    virtual ~Produit() = 0;

    virtual double prix() const { return valeur; }
    virtual void afficher(ostream& sortie) const { sortie << prix(); }

private:
    const double valeur;
};

Produit::~~Produit() {}

ostream& operator<<(ostream& sortie, Produit const& machin) {
    machin.afficher(sortie);
    return sortie;
}
```

et on en donne le corps à l'extérieur. Donc ce corps doit être donné, on doit donner une définition du destructeur même s'il n'a rien à faire. Voici à quoi nous aboutissons, au terme de cette étape, pour le codage de la classe Produit. Le constructeur de la classe permet d'initialiser son attribut valeur au moyen d'une valeur passée en paramètre. Ce constructeur peut être utilisé comme un constructeur par défaut car tous ses paramètres ont une valeur par défaut, ici la valeur 0. La classe produit offre une méthode d'affichage utilisable de façon polymorphique, cette méthode est déclarée comme virtuelle donc si elle s'applique à un pointeur ou à une référence,

notes

résumé

4m 13s





elle pourra offrir un comportement polymorphique. Elle utilise à son tour une autre méthode potentiellement polymorphique, la méthode `prix()`, aussi déclarée comme virtuelle. L'opérateur d'affichage peut être surchargé pour afficher des instances de `Produit`. Pour qu'il ait un comportement polymorphique, il faut simplement qu'il utilise, dans son corps, la méthode `afficher` polymorphique définie publiquement dans la classe `Produit`. Enfin, pour modéliser le fait qu'un produit est une entité abstraite dans notre conception, nous avons déclaré le destructeur comme virtuel pur et l'on n'oublie pas de le définir, même si l'on n'a rien de concret à lui faire faire, ici. Et ceci conclut notre séquence sur la définition des produits et leur affichage polymorphique. et leur affichage polymorphique.

notes

résumé

4m 49s

