

Support de cours

Cours:

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

Vidéo:

W17-03-surcharge-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Surcharge des opérateurs d'affichage. Tableau dynamique de pointeur. Prix du produit. Montre instance de la classe. New bracelet. Paramètre prix de base. Tableau dynamique des accessoires. Séquences vidéo précédentes. Nouveau pointeur. Accessoires de la montre. Second paramètre. Niveau des accessoires. Nom d'un fermoir. Collection de pointeurs. Première version.



[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/>

Etude de cas : surcharge d'opérateur et première version

(Partie 1)

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

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



Surcharge d'opérateur(s) et finalisation d'une première version du code



Dans les séquences vidéo précédentes,

notes

résumé

0m 1s



Rappel :

```
class Montre : public Produit {  
private:  
    unique_ptr<Mecanisme> coeur;  
    vector<unique_ptr<Accessoire>> accessoires;  
  
    Montre(const Montre&)      = delete;  
    Montre& operator=(Montre) = delete;  
};
```

nous vous avons présenté le problème général de l'étude de cas à laquelle nous nous intéressons ici, à savoir modéliser des montres avec différents mécanismes, différents accessoires. Puis nous avons abordé le problème plus spécifique de la surcharge des opérateurs d'affichage et comment faire de l'affichage polymorphe. Nous avons aussi complété la classe « Produit ». Dans cette séquence vidéo-ci, nous allons nous intéresser à la surcharge d'un autre opérateur, par exemple ici l'opérateur « += », et puis nous allons construire une première version réellement opérationnelle du programme.

notes

résumé

0m 5s



Rappel :

```
class Montre : public Produit {  
private:  
    unique_ptr<Mecanisme> coeur;  
    vector<unique_ptr<Accessoire>> accessoires;  
  
    Montre(const Montre&) = delete;  
    Montre& operator=(Montre) = delete;  
};
```

Commençons donc par ajouter un opérateur « += » à la classe « Montre », qui je vous le rappelle, héritait de « Produit », et avait un mécanisme, plus exactement un pointeur sur un mécanisme pour le polymorphisme, et surtout un tableau dynamique de pointeur sur des accessoires,

notes

résumé

0m 34s



Ajout d'accessoires (aux Montres)

Rappel :

```
class Montre : public Produit {
private:
    unique_ptr<Mecanisme> coeur;
    vector<unique_ptr<Accessoire>> accessoires;

    Montre(const Montre&)      = delete;
    Montre& operator=(Montre) = delete;
};
```

Accessoire
↑
Bracelet

On souhaite ajouter des accessoires à une montre au moyen de l'opérateur +=.

Par exemple :

```
montre += new Bracelet(...);
```

que l'on a appelé ici « accessoires ». L'idée est donc que l'on puisse rajouter des éléments à ce tableau dynamique en faisant par exemple pour une montre instance de la classe « Montre », « montre += new Bracelet (...) », où je vous rappelle que « Bracelet » est une sous-classe « d'« Accessoire » ». Donc on voudrait que ceci rajoute un nouveau pointeur sur un « Bracelet », à la collection de pointeurs sur des « Accessoires ». Un « Bracelet » étant un « Accessoire »,

notes

résumé

0m 53s



```
class Montre : public Produit {
public:
    void operator+=(Accessoire* p_accessoire) {
        accessoires.push_back(unique_ptr<Accessoire>(p_accessoire));
    }

private:
    unique_ptr<Mecanisme> coeur;
    vector<unique_ptr<Accessoire>> accessoires;

    Montre(const Montre&) = delete;
    Montre& operator=(Montre) = delete;
};
```

un pointeur sur un « Bracelet » peut être mis dans un pointeur sur un « Accessoire ». Voilà donc l'idée de cet opérateur « += ». Le prototype de cet opérateur va donc être le suivant : nous n'allons pas utiliser sa valeur de retour, nous n'allons jamais écrire des choses de type « z = (montre += new Bracelet (...)); », donc ici un type de retour « void », c'est l'opérateur « += » de la classe « Montre » et cet opérateur doit donc recevoir ici un pointeur sur un « Accessoire », donc ici « new » va renvoyer un pointeur à la C, et on veut le voir comme un pointeur sur un « Accessoire », donc ici de façon très générale, ce sera un pointeur sur un « Accessoire », ici on a ajouté un « Bracelet », mais on pourrait ajouter un « Fermoir », un « Boitier », n'importe quelle sous-classe de la classe « Accessoire ».

notes

résumé

1m 25s



```
class Montre : public Produit {
public:
    void operator+=(Accessoire* p_accessoire) {
        accessoires.push_back(unique_ptr<Accessoire>(p_accessoire));
    }

private:
    unique_ptr<Mecanisme> coeur;
    vector<unique_ptr<Accessoire>> accessoires;

    Montre(const Montre&) = delete;
    Montre& operator=(Montre) = delete;
};
```

Voyons maintenant comment écrire sa définition. Ce pointeur sur un « Accessoire », « p_accessoire », que l'on a reçu, on veut simplement l'ajouter au tableau dynamique des accessoires stockés dans la Montre, et donc on va simplement faire un « push_back » de ce pointeur,

notes

résumé

2m 15s





mais ici comme on a reçu un pointeur à la C, on va au passage transformer le pointeur à la C en un « unique_ptr », donc on fait ici un « unique_ptr » qui transforme ce pointeur à la C « p_accessoire » reçu, et que l'on rajoute « push_back » à la fin du tableau « Accessoire », dans lequel sont stockés tous les accessoires de la montre. Voilà pour cette surcharge, somme toute assez simple, de l'opérateur « += » qui nous offre maintenant

notes

résumé

2m 32s



Essayons maintenant d'avoir une première version opérationnelle de notre code, pour le moment :

- ▶ *sans* tous les mécanismes
- ▶ *sans* copie des montres

Pour cela, il nous faut encore :

- ▶ quelques *accessoires*
- ▶ terminer la classe `Montre`
- ▶ un exemple d'utilisation dans le `main()`

une écriture très compacte pour rajouter des accessoires à une montre. Essayons maintenant de faire une première version opérationnelle du programme, essayons de finaliser ce que nous avons écrit jusqu'ici,

notes

résumé

3m 1s



Quelques accessoires

Décidons par exemple que
les accessoires :

```
class Accessoire : public Produit {  
  
};
```

et pour le moment sans aucun mécanisme, et sans faire de copie de montres, nous laissons ces 2 points pour les 2 prochaines séquences vidéo. Pour pouvoir finaliser le programme que nous avons jusqu'à maintenant, il nous faudrait quelques accessoires un peu plus concrets, il faudrait terminer la classe « Montre », qui pour l'instant n'est pas encore tout à fait opérationnelle, et bien sûr utiliser tout ceci dans un « main » typique. Nous allons reprendre ces points les un après les autres,

notes

résumé

3m 13s



Quelques accessoires

Décidons par exemple que les accessoires :

- ▶ ont un nom et une valeur de base fixés au départ (sans valeur par défaut)
- ▶ s'affichent en indiquant leur nom et leur prix

```
class Accessoire : public Produit {
public:
    Accessoire(string const& un_nom,
               double prix_de_base)
        : Produit(prix_de_base), nom(un_nom)
    {}

    virtual ~Accessoire() {}

    virtual void afficher(ostream& sortie)
        const override {
        sortie << nom << " coûtant ";
        Produit::afficher(sortie);
    }

private:
    const string nom;
};
```

à commencer par quelques accessoires, pour rappel, nous avons une classe « Accessoire », qui est un « Produit », et décidons par exemple que nous ayons au niveau des accessoires un nom pour pouvoir les désigner. Pour ceci, on va dire qu'on a un nom qui ne changera pas une fois qu'il aura été fixé par le constructeur. Et nous allons ajouter un constructeur à la classe « Accessoire », qui prendra un nom pour pouvoir nommer cet accessoire. Et il ne faut pas oublier que tout produit peut être vendu et a donc un prix, et recevra donc un second paramètre pour pouvoir initialiser le prix du produit. La section « liste d'initialisation » du constructeur « Accessoire » va commencer par appeler le constructeur de la super-classe « Produit », en passant ici le paramètre prix de base que l'on a reçu, et ensuite on initialisera l'attribut « nom » avec le nom reçu. Voilà pour le constructeur qu'on a décidé de donner à tout les accessoires. Et décidons maintenant d'une façon d'afficher les accessoires, car je vous rappelle que tous les produits sont capables de s'afficher de façon polymorphique. Nous allons donc ici redéfinir la méthode afficher que nous avons héritée de « Produit », on dit qu'on la redéfinit en ajoutant le mot clé « override ». Et décidons qu'ils s'affichent en indiquant le nom et leur prix. Donc on va afficher simplement le nom, puis on va indiquer qu'il coûte, et nous savons déjà afficher le prix en appelant directement la méthode afficher de la super-classe « Produit ». Ici, on utilise l'opérateur de résolution de portée pour démasquer la méthode afficher héritée de la super-classe « Produit », parce que bien sûr si on écrit « afficher » tout seul, sans démasquage, ce sera la méthode « afficher » de la classe en question

notes

résumé

3m 39s



Quelques accessoires

Décidons par exemple que les accessoires :

- ▶ ont un nom et une valeur de base fixés au départ (sans valeur par défaut)
- ▶ s'affichent en indiquant leur nom et leur prix

```
class Accessoire : public Produit {
public:
    Accessoire(string const& un_nom,
               double prix_de_base)
        : Produit(prix_de_base), nom(un_nom)
    {}

    virtual ~Accessoire() {}

    virtual void afficher(ostream& sortie)
        const override {
        sortie << nom << " coûtant ";
        Produit::afficher(sortie);
    }

private:
    const string nom;
};
```

et on aurait ici une récursion infinie, « afficher » s'appellerait elle-même, et donc pour, dans la méthode « afficher » de la classe « Accessoire », appeler la méthode « afficher » héritée de la super-classe « Produit », il faut ici démasquer cette méthode « afficher ». Enfin, comme nous avons un comportement que l'on souhaite polymorphique, avec des méthodes virtuelles, on oublie pas de rajouter le destructeur polymorphique, le destructeur virtuel, pour la classe dont on souhaite avoir un comportement polymorphique,

notes

résumé



de sorte que si on a des sous-classes qui héritent,

notes

résumé

6m 1s



Quelques accessoires

Décidons par exemple que les accessoires :

- ▶ ont un nom et une valeur de base fixés au départ (sans valeur par défaut)
- ▶ s'affichent en indiquant leur nom et leur prix

```
class Accessoire : public Produit {
public:
    Accessoire(string const& un_nom,
               double prix_de_base)
        : Produit(prix_de_base), nom(un_nom)
    {}

    virtual ~Accessoire() {}

    virtual void afficher(ostream& sortie)
        const override {
        sortie << nom << " coûtant ";
        Produit::afficher(sortie);
    }

private:
    const string nom;
};
```

et c'est ce qui sera le cas d'« Accessoire », qui ont besoin d'avoir un destructeur qui fait des comportements particuliers, ce soit effectivement leur destructeur qui soit appelé de façon polymorphique, et non pas le destructeur de la classe « Accessoire ».

notes

résumé

6m 2s



Quelques accessoires

```

class Bracelet : public Accessoire {
public:
    Bracelet(string const& un_nom, double prix_de_base)
        : Accessoire("bracelet " + un_nom, prix_de_base)
    {}
    virtual ~Bracelet() {}
};

//-----
class Fermeoir : public Accessoire {
public:
    Fermeoir(string const& un_nom, double prix_de_base)
        : Accessoire("fermeoir " + un_nom, prix_de_base)
    {}
    virtual ~Fermeoir() {}
};

// ...

```

en cuir (handwritten note with an arrow pointing to the string "bracelet " in the Bracelet constructor)

Décidons enfin que le prix des accessoires est le même que le prix d'un produit usuel, c'est-à-dire un produit général tel que nous l'avons conçu tout en haut de la hiérarchie. À ce moment là, nous n'avons rien à faire, puisque la méthode « prix » que nous avons héritée de la classe « Produit » suffit ici, et on n'a pas besoin de la redéfinir dans la classe « Accessoire ». Nous avons donc ici une classe « Accessoire » tout à fait opérationnelle et qui correspond à ce que nous souhaitons. Définissons maintenant quelques accessoires, par exemple un « Bracelet », un « Bracelet » est un « Accessoire », donc on va avoir ici une relation d'héritage. Au niveau des accessoires, admettons que l'on souhaite que leur nom, je vous rappelle que les accessoires ont un nom, donc tous les accessoires vont hériter ce nom. Admettons que l'on souhaite donc que chacun des accessoires ait ainsi un nom qui marque ce qu'ils sont, par exemple le nom d'un bracelet va commencer par « bracelet », le nom d'un fermoir va commencer par « fermoir ». On va donc forcer ceci au niveau du constructeur, par exemple pour la sous-classe « Bracelet » de la classe « Accessoire », on recevra le nom, qui sera le complément qu'on va rajouter derrière « bracelet », par exemple « en cuir », pour que le nom complet soit « bracelet en cuir ».

notes

résumé

6m 18s





Et puis comme deuxième paramètre, le prix, que nous passons aussi au constructeur de la super-classe, dans la section liste d'initialisation. Et puis toutes ces classes pouvant potentiellement avoir des sous-classes polymorphiques, nous n'oublions pas le destructeur virtuel. De même, on pourrait définir un fermoir, qui hérite d'« Accessoire », dans lequel on définit un constructeur qui prend un complément du nom « fermoir » et qui prend un prix, pour pouvoir appeler le constructeur de la super-classe « Accessoire », où le nom complet sera donc « fermoir » avec ce complément qu'on a reçu comme premier paramètre, et comme deuxième paramètre, le prix que nous avons reçu, avec toujours le destructeur virtuel.

notes

résumé

7m 37s



Pour finaliser la classe `Montre`
(mais sans mécanisme) :

```
class Montre : public Produit {
public:
    void operator+=(Accessoire* p_accessoire) {
        accessoires.push_back(
            unique_ptr<Accessoire>(p_accessoire));
    }

private:
    unique_ptr<Mecanisme> coeur;
    vector<unique_ptr<Accessoire>> accessoires;

    Montre(const Montre&) = delete;
    Montre& operator=(Montre) = delete;
};
```

Voilà, nous en avons terminé avec les accessoires, essayons aussi de rendre la classe « Montre » utilisable sans les mécanismes et sans la copie, que nous laissons de côté pour le moment.

notes

résumé

8m 16s



Pour finaliser la classe `Montre`
(mais sans mécanisme) :

- ajoutons un constructeur (au moins remettre le constructeur par défaut)

```
class Montre : public Produit {  
public:  
    Montre() = default;  
    virtual ~Montre() {}  
  
    // ...  
  
private:  
    // unique_ptr<Mecanisme> coeur;  
  
    // ...  
};
```

Nous avons à ce stade défini simplement le contenu de la classe « Montre », nous avons empêché la copie, et nous avons ajouté l'opérateur « += », qui nous permet de rajouter des accessoires à nos montres. Pour rendre cette classe opérationnelle, il faut au moins lui ajouter un constructeur, puisque, comme nous avons touché ici au constructeur, le constructeur par défaut, fournit par défaut, ne l'est plus, nous n'avons plus le constructeur par défaut, par défaut, il faut donc au moins le rajouter ici. Evidemment le constructeur par défaut ne fera rien

notes

résumé

8m 26s



Pour finaliser la classe `Montre`
(mais sans mécanisme) :

```
class Montre : public Produit {  
public:  
    void operator+=(Accessoire* p_accessoire) {  
        accessoires.push_back(  
            unique_ptr<Accessoire>(p_accessoire));  
    }  
  
private:  
    unique_ptr<Mecanisme> coeur;  
    vector<unique_ptr<Accessoire>> accessoires;  
  
    Montre(const Montre&) = delete;  
    Montre& operator=(Montre) = delete;  
};
```

au niveau de l'initialisation des mécanismes, et donc à ce moment là, nous allons simplement commenter, pour éviter tout problème avec l'utilisation ici d'un pointeur qui ne serait pas initialisé ; nous allons donc simplement commenter cette ligne, et le constructeur par défaut,

notes

résumé

9m 1s



Pour finaliser la classe `Montre`
(mais sans mécanisme) :

- ajoutons un constructeur (au moins remettre le constructeur par défaut)

```
class Montre : public Produit {  
public:  
    Montre() = default;  
    virtual ~Montre() {}  
  
    // ...  
  
private:  
    // unique_ptr<Mecanisme> coeur;  
  
    // ...  
};
```

en ce qui concerne le vecteur de pointeur sur des accessoires, va créer un vecteur nul, et donc là, il n'y a pas de soucis, vecteur nul que nous pourrons ensuite remplir à l'aide de notre opérateur « += ».

notes

résumé

9m 19s



Pour finaliser la classe `Montre` (mais sans mécanisme) :

- ▶ ajoutons un constructeur (au moins remettre le constructeur par défaut)
- ▶ décidons d'un calcul de prix : somme des prix des accessoires

```
// ...
virtual double prix() const override {

// Au départ, le prix est la valeur de base
double prix_final(Produit::prix());

for (auto const& p_acc : accessoires) {
    prix_final += p_acc->prix();
}

return prix_final;
}
// ...
private:
// unique_ptr<Mecanisme> coeur;
vector<unique_ptr<Accessoire>> accessoires;
// ...
};
```

Donc le constructeur par défaut nous va bien, si tant est que nous n'ayons pas ce pointeur, donc nous allons le commenter pour l'instant, jusqu'à ce que nous y revenions dans une séquence ultérieure. Voilà pour la construction des montres. Décidons ensuite du calcul du prix des montres, et admettons que ce soit la somme des prix de ses accessoires. Pour ceci, nous allons redéfinir la méthode `prix`, nous « override » cette méthode `prix` virtuelle, héritée de la super-classe « `Produit` ». Nous allons donc décider qu'au départ le prix d'un montre est le prix de base, que nous récupérerons au travers de la méthode `prix` d'origine, héritée de la super-classe « `Produit` », puis à ce prix de départ, nous allons rajouter l'ensemble des prix des différents accessoires. Pour ceci, nous parcourons la liste des accessoires, je vous rappelle que pour parcourir un tableau de « `unique_ptr` », il faut absolument utiliser cette syntaxe, passer par des références, ici, il se trouve que récupérer le prix ne modifie pas les instances parcourue, on va donc utiliser une référence constante. On va faire ici une boucle qui parcourt par référence constante les « `unique_ptr` » de ce tableau dynamique d'accessoires. Et nous rajoutons au prix, que nous avons au départ initialisé, à chaque fois le prix, c'est-à-dire l'appel à la méthode `prix` de l'objet pointé au travers de « `p_acc` » ici, donc de l'accessoire pointé au travers de ce pointeur, c'est comme ceci qu'on écrit l'appel à la méthode `prix` de quelque chose qui est un pointeur sur un objet ayant une telle méthode. Et enfin on retourne le prix que l'on vient de calculer. Décidons enfin comment afficher les montres, et par exemple nous décidons d'avoir un message pour dire qu'un montre est composée de : la liste de ses différents accessoires... Et puis à

notes

résumé

9m 31s



Pour finaliser la classe `Montre` (mais sans mécanisme) :

- ▶ ajoutons un constructeur (au moins remettre le constructeur par défaut)
- ▶ décidons d'un calcul de prix : somme des prix des accessoires

```
// ...
virtual double prix() const override {

// Au départ, le prix est la valeur de base
double prix_final(Produit::prix());

for (auto const& p_acc : accessoires) {
    prix_final += p_acc->prix();
}

return prix_final;
}
// ...
private:
// unique_ptr<Mecanisme> coeur;
vector<unique_ptr<Accessoire>> accessoires;
// ...
};
```

la fin on affiche le prix total de la montre. Et donc on va aussi ici redéfinir la méthode `afficher` virtuelle héritée de la super-classe « `Produit` », et on rappelle ici que nous sommes en train de la redéfinir, en commençant l'affichage par « Une montre composée de », et ensuite en parcourant l'ensemble des accessoires de la montre, pour chacun des accessoires, on affiche cette petite étoile, et puis on affiche ici l'accessoire en question. Une question ici : Est-ce qu'on a le droit d'écrire ceci ? Est-ce que ceci compile ? Est-ce qu'on a surchargé l'opérateur d'affichage pour ceci ? l'opérateur d'affichage pour ceci ?

notes

résumé