

Chapitre III

Héritage

- 1 Syntaxe
- 2 Exemple
- 3 Constructeurs et destructeurs
- 4 Redéfinition de méthodes
- 5 Liaison dynamique / liaison statique
- 6 Classes abstraites

Héritage

- L'**héritage** permet de former une nouvelle classe à partir de classes existantes.
- La nouvelle classe (**classe fille, sous-classe**) **hérite** des attributs et des méthodes des classes à partir desquelles elle a été formée (**classes mères, super-classes**).
- De nouveaux attributs et de nouvelles méthodes peuvent être définis dans la classe fille.
- Des méthodes des classes mères peuvent être **redéfinies** dans la classe fille.
- La **redéfinition de méthode** consiste à (re)définir dans la classe fille le comportement (code de la méthode) qui existait déjà dans une classe mère.
Si une méthode d'une classe mère n'est pas redéfinie dans la classe fille, alors le code défini dans la classe mère sera utilisé « tel quel » sur les instances de la classe fille.

Intérêt : Réutilisation

Utiliser une classe existante. . .

- En lui rajoutant des membres
- et/ou en modifiant certains de ses comportements sans modifier le code de la classe existante.

Chapitre III

Héritage

- 1 Syntaxe
- 2 Exemple
- 3 Constructeurs et destructeurs
- 4 Redéfinition de méthodes
- 5 Liaison dynamique / liaison statique
- 6 Classes abstraites

Héritage

Syntaxe

Syntaxe

```
class NomClasseFille:
    public NomClasseMere1, public NomClasseMere2 ...
{
    // Déclaration des nouveaux attributs
    // Déclaration des nouvelles méthodes
    // Redéfinition de méthodes
    ...
};
```

- C++ permet aussi d'utiliser l'héritage `protected` et `private`, rarement utilisés.
- Souvent, on utilise l'**héritage simple** : une classe fille a **une** classe mère. On parle d'**héritage multiple** quand une classe fille a plusieurs classes mères.

Héritage

Intérêts

Intérêts de l'héritage :

- Favoriser la réutilisation
- **Factoriser** le code
- Catégoriser les concepts dans une relation de généralisation
Chaque concept est caractérisé par ses différences par rapport à son (ses) parent(s)

Chapitre III

Héritage

- 1 Syntaxe
- 2 Exemple
- 3 Constructeurs et destructeurs
- 4 Redéfinition de méthodes
- 5 Liaison dynamique / liaison statique
- 6 Classes abstraites

Héritage Exemple

Gérer des produits en vente dans un magasin.

- Chaque produit comporte une référence, un nom, un prix HT et un prix TTC.
- Le taux de TVA est de 19.6, sauf pour les produits culturels : 5.5.
- Les produits périssables ont une date limite de vente.
- Le matériel multimédia a une durée de garantie exprimée en années.
- Chaque produit peut afficher une « étiquette » contenant référence, prix, et informations supplémentaires.

Sans héritage :

- Définir les classes `ProduitStandard`, `ProduitCulturel`, `ProduitPerissable`, `ProduitMultimedia`... et copier/coller du code !

Héritage

Exemple

- Tous les produits ont des données communes (la référence...) et des comportements communs (l'affichage de l'étiquette, le calcul du prix TTC)
 ⇒ Créer **une** classe `Produit` qui regroupe (factorise) ce qui est commun à tous les produits et une **classe fille** `ProduitPerissable` qui contient ce qui est *particulier* à un produit périssable.
 Et on crée d'autres classes filles de `Produit` pour les autres produits particuliers.

Héritage

Exemple (produit.h)

```
#ifndef produit_h
#define produit_h
#include <string>

class Produit
{
private:
    int m_ref;
    std::string m_nom;
    float m_prixht;

public:
    Produit(int ref, std::string const & nom,
           float prixht);

    std::string const & nom();
    void afficher();
    // + accesseurs
};

#endif
```

Héritage

Exemple (produit.cpp)

```

#include "produit.h"
#include <iostream>

using namespace std;

Produit::Produit(int ref, string const & nom,
    float prixht)
    :m_ref(ref), m_nom(nom), m_prixht(prixht)
{ }

string const & Produit::nom()
{
    return m_nom;
}

void Produit::afficher()
{
    cout << m_ref << " " << m_nom << endl;
    cout << "prix HT : " << m_prixht << endl;
}

```

Héritage

Exemple (produitperissable.h)

```

#ifndef produitperissable_h
#define produitperissable_h

#include "produit.h"

class ProduitPerissable: public Produit
{
private:
    std::string m_datelimites;

public:
    ProduitPerissable(int ref, std::string const & nom,
        float prixht, std::string const & datelimites);

    void afficher();
};

#endif

```

Héritage

Exemple (produitperissable.cpp)

```
#include "produitperissable.h"
#include <iostream>

using namespace std;

ProduitPerissable::ProduitPerissable(int ref,
    string const & nom, float prixht,
    string const & datelimit)
    :Produit(ref, nom, prixht),
      m_datelimit(datelimit)
{
}

void ProduitPerissable::afficher()
{
    Produit::afficher();
    cout << "date limite de vente : " << m_datelimit
        << endl;
}
```

Héritage

Remarquer que la méthode `nom` n'a pas été redéfinie dans `ProduitPerissable`. Pourtant, elle peut être appelée sur les instances de `ProduitPerissable` car elle a été héritée.

Exemple

```
#include "produitperissable.h"
#include <iostream>
using namespace std;

int main()
{
    ProduitPerissable pp(42, "Yaourts", 3, "30/09");
    cout << pp.nom();
    return 0;
}
```

Chapitre III

Héritage

- 1 Syntaxe
- 2 Exemple
- 3 Constructeurs et destructeurs
- 4 Redéfinition de méthodes
- 5 Liaison dynamique / liaison statique
- 6 Classes abstraites

Héritage et constructeurs

- Les constructeurs des classes mères ne peuvent être utilisés pour construire des instances de la classe fille.
- → Il faut définir de nouveaux constructeurs.
- Les constructeurs de la classe fille font appel aux constructeurs des classes mères **au début de la liste d'initialisations**.

Exemple (produitperissable.cpp)

```
ProduitPerissable::ProduitPerissable(int ref,  
    string const & nom, float prixht,  
    string const & datelimité)  
    :Produit(ref, nom, prixht),  
      m_datelimité(datelimité)  
{ }
```

- Le destructeur des classes mères est **toujours appelé implicitement après** l'exécution du destructeur de la classe fille.
- On n'appelle **jamais explicitement** le destructeur des classes mères dans le destructeur de la classe fille.

Chapitre III

Héritage

- 1 Syntaxe
- 2 Exemple
- 3 Constructeurs et destructeurs
- 4 Redéfinition de méthodes
- 5 Liaison dynamique / liaison statique
- 6 Classes abstraites

Redéfinition

- Quand une méthode est redéfinie dans la classe fille, elle doit être déclarée **avec la même signature** que dans la classe mère.
- Le code d'une méthode redéfinie fait souvent appel à la méthode de la classe mère (*super méthode*)
NomDeLaClasseMere :: nomDeLaMethode (arguments)

Exemple (produitperissable.cpp)

```
void ProduitPerissable::afficher()
{
    Produit::afficher();
    cout << "date limite de vente : " << m_datelimites
        << endl;
}
```

Chapitre III

Héritage

- 1 Syntaxe
- 2 Exemple
- 3 Constructeurs et destructeurs
- 4 Redéfinition de méthodes
- 5 Liaison dynamique / liaison statique
- 6 Classes abstraites

Présentation du problème

Ajouter une méthode calculant le **prix TTC** et modifier la méthode d'affichage pour afficher ce prix.

- Quel que soit le produit, le prix TTC est **toujours** obtenu par $m_prixht * (1 + taux_de_tva)$
- Ce qui change est le **taux de TVA**, pas le calcul du prix TTC
⇒ La méthode `prixTTC` sera définie dans `Produit` et ne sera pas redéfinie dans les sous-classes.
- Cette méthode a besoin du taux de TVA appliqué au produit (qui, lui, est spécifique au produit).

Présentation du problème

Une première (mauvaise) solution

- Ajouter un attribut `m_tauxtva` dans `Produit`
- Modifier les constructeurs avec un paramètre supplémentaire.
- Ajouter la méthode suivante dans `Produit` :

Exemple (produitperissable.cpp)

```
float Produit::prixTTC()
{
    return m_prixht * (m_tauxtva + 1);
}
```

Défauts

- Nécessité de fournir le taux de TVA à chaque création de `Produit`
- Chaque instance de `Produit` contient une valeur « différente »
- Que faire si le taux de TVA est mis à jour ?

Le taux de TVA n'est pas une information associée à un(e instance de) `Produit` mais à une **classe** (ensemble d'instances) de produits.

Présentation du problème

Une meilleure solution

Définir une méthode `tauxTVA` dans `Produit` (retournant 0.196) et la redéfinir dans `ProduitCulturel` qui est une sous-classe de `Produit` (retournant 0.055).

Exemple

```
class Produit
{
  ...
  protected:
    float tauxTVA();
  ...
};

float Produit::tauxTVA()
{
  return 0.196;
}

float Produit::prixTTC()
{
  return m_prixht * (tauxTVA() + 1);
}
```

Présentation du problème

Une meilleure solution

Exemple

```
class ProduitCulturel: public Produit
{
  public:
    ProduitCulturel(int ref, std::string const & nom,
    float prixht);
  protected:
    float tauxTVA();
};

ProduitCulturel::ProduitCulturel(int ref,
  string const & nom, float prixht)
  :Produit(ref, nom, prixht)
{}

float ProduitCulturel::tauxTVA()
{
  return 0.055;
}
```

Liaison statique

Testons les classes. . .

Exemple

```
int main()
{
    ProduitPerissable pp(1, "Yaourts", 100, "30/09");
    ProduitCulturel pc(2, "Dictionnaire", 100);
    cout << pp.tauxTVA() << " ";
    cout << pc.tauxTVA() << endl;
    cout << pp.prixTTC() << " ";
    cout << pc.prixTTC() << endl;
}
```

Résultat affiché :

```
0.196 0.055
119.6 119.6
```

Liaison statique

Liaison statique

Quand une méthode à **liaison statique** est appelée sur un objet, c'est la méthode correspondant à la **classe** de cet objet **déterminée au moment de la compilation** qui est exécutée.

tauxTVA est une méthode à liaison statique.

- Dans main elle est appelée sur
 - pp qui est un ProduitPerissable
tauxTVA n'est pas redéfinie dans ProduitPerissable, c'est donc le code hérité de Produit qui est exécuté.
 - pc qui est un ProduitCulturel
tauxTVA est redéfinie dans ProduitCulturel, c'est ce code qui est exécuté.

Liaison statique

- Quand `prixTTC` est exécuté sur `pp` comme sur `pc`, c'est **le code de la méthode de `Produit`** qui est exécuté. Quand la ligne `m_prixht + (tauxTVA() + 1);` est exécutée, **l'objet courant (`this`) est un (pointeur sur un) `Produit`**. `tauxTVA` étant à liaison statique, c'est le code de **`Produit::tauxTVA`** qui est exécuté.
 - Dans le cas de `pp`, ce n'est pas gênant car `tauxTVA` n'est pas redéfinie dans `ProduitPerissable`
 - Dans le cas de `pc`, la redéfinition de `tauxTVA` est ignorée, comme si `pc` n'était qu'un `Produit`

Liaison statique

En C++, en l'absence de déclaration particulière, toute méthode est à liaison statique.

Comment (mal) régler le problème ?

Redéfinir `prixTTC` dans `ProduitCulturel` avec le même code.

Exemple

```
float ProduitCulturel::prixTTC()
{
    return m_prixht * (tauxTVA() + 1);
}
```

- Duplication de code.
- Et si on voulait afficher le prix TTC dans `afficher` ? Redéfinir `afficher` dans les sous-classes... afin qu'`afficher` appelle le « bon » `prixTTC`, etc.
- On perd tout l'avantage de l'héritage ⇒ **À éviter !**

Liaison dynamique

Liaison dynamique

Quand une méthode à **liaison dynamique** est appelée sur un objet, c'est la méthode correspondant à la **classe « réelle »** de cet objet **déterminée au moment de l'exécution** qui est exécutée.

Syntaxe (Déclaration de méthode à liaison dynamique)

- Utilisation du mot-clef `virtual` devant la **déclaration** de la méthode.
- Ce mot-clef ne doit pas être utilisé devant la définition de la méthode, ni devant les redéfinitions dans les sous-classes.

Liaison dynamique

Exemple

```
class Produit
{
    ...
    virtual float tauxTVA();
    ...
};
```

Résultat affiché :

```
0.196 0.055
119.6 105.5
```

Quand `Produit::prixTTC` est exécutée sur `pc`, elle appelle `tauxTVA`. Comme `tauxTVA` est à liaison dynamique, le code correspondant est cherché à partir de la classe « réelle » de l'objet, i.e. `ProduitCulturel`.

Liaison dynamique

- De façon générale, les méthode *susceptibles d'êtres redéfinies* dans les sous-classes devraient être définies `virtual`.
- L'appel à des méthodes à liaison dynamique est légèrement plus lent.
⇒ ne pas utiliser de façon systématique.
- Il n'est pas conseillé d'appeler une méthode à liaison dynamique dans un constructeur.

Liaison dynamique

Exemple

```
void Produit::afficher()
{
    cout << m_ref << " " << m_nom << endl;
    cout << "prix HT : " << m_prixht << endl;
    cout << "prix TTC : " << prixTTC() << endl;
}
```

- `prixTTC` n'a pas besoin d'être déclarée `virtual`
- `afficher` non plus (pour l'instant)

Chapitre III

Héritage

- 1 Syntaxe
- 2 Exemple
- 3 Constructeurs et destructeurs
- 4 Redéfinition de méthodes
- 5 Liaison dynamique / liaison statique
- 6 Classes abstraites

Présentation du problème

- Dans la solution précédente, tous les produits (les instances de `Produit` et les sous-classes) ont un taux de TVA de 19.6, sauf pour les instances de `ProduitCulturel`.
- On peut faire un autre choix :
 - Le taux de TVA d'un `Produit` (en général) est « indéterminé »
 - Il y a deux sous-classes de `Produit` : les `ProduitCourant` (19.6) et les `ProduitCulturel` (5.5).

Une solution (qui ne fonctionne pas)

- La classe `Produit` ne dispose pas de la méthode `tauxTVA`
- `ProduitCourant` et `ProduitCulturel` (classes-filles de `Produit`) ont une méthode `tauxTVA`
- Le calcul du prix TTC est le même, quel que soit le type de produit
Il est donc dans `Produit` :

Exemple

```
float Produit::prixTTC()
{
    return m_prixht * (tauxTVA() + 1);
}
```

- ⇒ **Erreur de compilation** : La classe `Produit` ne dispose pas d'une méthode `tauxTVA`.

Pourquoi cette erreur ?

Au moment de la compilation, le compilateur vérifie les appels des méthodes...

- `tauxTVA` est appelée sur l'objet courant, un `Produit`...
- Or, `Produit` ne dispose pas de cette méthode.
- Avec des méthodes `virtual`, c'est au moment de l'exécution que le code exécuté est choisi, mais...
Au moment de la compilation, l'existence d'une méthode (`virtual` ou pas) est testée.
- Il faut déclarer `tauxTVA` dans `Produit`.
Même si on n'est pas capable d'écrire son code.

Méthode virtuelle pure

ou Méthode abstraite

Méthode virtuelle pure

Une **méthode virtuelle pure** (**abstraite**) est une méthode déclarée dans une classe, destinée à être définie dans les sous-classes, mais pour laquelle aucun code n'est donné.

Syntaxe (Déclaration)

```
virtual typeretour nommethode(arguments) =0;
```

Pas de définition d'une méthode virtuelle pure.

Méthode virtuelle pure

- Définir `tauxTVA` comme virtuelle pure dans `Produit` règle le problème : il existe une méthode de ce nom lors de la compilation de `prixTTC`.

Exemple

```
class Produit
{
    ...
    virtual float tauxTVA() =0;
    ...
};
```

...

Méthode virtuelle pure

- Définir `tauxTVA` dans les deux classes filles de `Produit`.

Exemple

```
class ProduitCourant: public Produit
{
    ...
    float tauxTVA();
    ...
};
float ProduitCourant::tauxTVA()
{
    return 0.196;
}

class ProduitCulturel: public Produit
{
    ...
    float tauxTVA();
    ...
};
float ProduitCulturel::tauxTVA()
{
    return 0.055;
}
```

Méthode virtuelle pure

- Dans `prixTTC` (de `Produit`) l'appel à `tauxTVA` exécute le code de la méthode correspondante de la classe dont l'objet est réellement instance. (car `tauxTVA` est à liaison dynamique)
- Mais si l'objet est un `Produit` (sans être instance d'aucune classe fille), Quel code est exécuté ?

Classe abstraite

Classe abstraite

- Toute classe contenant (au moins) une méthode virtuelle pure est appelée **abstraite**.
- Une classe abstraite ne peut être instanciée.

Exemple

```
Produit p; // provoque une erreur
ProduitCourant p1; // ok
Produit * pp; // ok
Produit * pp1 = new ProduitCourant(...); // ok
```

Attention

Si une classe B est classe fille de A , si dans A est déclarée une méthode m virtuelle pure, et si B ne redéfinit pas m ... alors B contient une méthode virtuelle pure, elle est donc abstraite.

Classe abstraite

Dans l'exemple,

- `Produit` est abstraite et ne peut être instanciée.
- `ProduitCourant` et `ProduitCulturel` sont concrètes et peuvent être instanciées.

Pour finir l'exemple...

- `ProduitMultimedia` et `ProduitPerissable` peuvent être définies comme des sous-classes de `ProduitCourant`.
- Elles héritent alors du taux de TVA des `ProduitCourant`.
- Dans ces 2 classes, les constructeurs doivent être définis, et la méthode `afficher` doit être redéfinie.