Chapitre 3

Héritage

Syntaxe

Exemple

Constructeurs et destructeurs

Redéfinition de méthodes

Liaison dynamique / liaison statique

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.

Héritage

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.

Syntaxe

Héritage - Syntaxe

Syntaxe héritage

```
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).

Exemple

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 20, 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.

Déclaration de produit (produitv1.hh)

```
#pragma once
#include <string>
using reference=unsigned int;
using prix=float;
class produit {
 public:
  produit(reference ref, std::string const & nom, prix prixht);
  std::string const & nom() const;
  prix prixttc() const;
  void afficher() const;
  // + accesseurs prixht, référence
 private:
  reference _ref;
  std::string _nom;
  prix _prixht;
};
```

Définition de produit (produitv1.cc)

```
#include "produitv1.hh"
#include <iostream>
produit::produit(reference ref, std::string const & nom, prix prixht
    )
  :_ref(ref), _nom(nom), _prixht(prixht) {
}
std::string const & produit::nom() const {
  return _nom;
}
float produit::prixttc() const {
  return _prixht * 1.20;
}
void produit::afficher() const {
  std::cout << _ref << " " << _nom << "\n";
  std::cout << "prix HT : " << _prixht << "\n";
  std::cout << "prix TTC : " << prixttc() << "\n";
}
```

Déclaration de produitperissable (produitperissablev1.hh)

```
#pragma once
#include "produitv1.hh"
using date=std::string;
class produitperissable: public produit {
 public:
  produitperissable(reference ref, std::string const & nom, prix
      prixht, date const & peremption);
  date const & peremption() const {
    return _peremption; }
  void afficher() const;
 private:
  date _peremption;
};
```

Définition de produitperissable (produitperissablev1.cc)

```
#include "produitperissablev1.hh"
#include <iostream>

produitperissable::produitperissable(reference ref, std::string
    const & nom, prix prixht, date const & peremption)
  :produit(ref, nom, prixht), _peremption(peremption) {
}

void produitperissable::afficher( ) const {
  produit::afficher( );
  std::cout << "peremption : " << _peremption << "\n";
}
```

Test (produittestv1.cc)

```
#include "produitv1.hh"
#include "produitperissablev1.hh"
#include <iostream>
int main() {
  produit p(1, "test", 100);
  // Ceci provoque une erreur :
  // produit p;
  // Ceci provoque une erreur :
  // produit p=new produit(1, "test", 100);
  // Ceci est inutilement compliqué :
  // produit * p = new produit(1, "test", 100);
```

Test (suite) (produittestv1.cc)

```
  p.afficher();
  produitperissable p2(2, "test2", 100, "15/10");
  std::cout << p2.nom() << "\n";
  p2.afficher();
  return 0;
}
```

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.

Constructeurs et destructeurs

Héritage et constructeurs

- Les constructeurs des classes mères ne peuvent être utilisés pour construire des instances de la classe fille.
- $\rightarrow$ 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**.

Constructeur (produitperissablev1.cc)

```
produitperissable::produitperissable(reference ref, std::string
    const & nom, prix prixht, date const & peremption)
  :produit(ref, nom, prixht), _peremption(peremption) {
}
```

Héritage et destructeurs

- 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.

Redéfinition de méthodes

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)`

Redéfinition de méthode (produitperissablev1.cc)

```
void produitperissable::afficher() const {
  produit::afficher();
  std::cout << "peremption : " << _peremption << "\n";
}
```

Liaison dynamique / liaison statique

Présentation du problème

Ajouter une méthode calculant le prix TTC et modifier la méthode d'affichage pour afficher ce prix. Dans l'exemple **`produit`** dispose déjà de cette méthode, mais on veut maintenant prendre en compte différents taux de TVA.

- Quel que soit le produit, le prix TTC est toujours obtenu par `_prixht * (1 + taux_de_tva)`
- Ce qui change est le taux de TVA, pas le calcul du prix TTC
  $\Rightarrow$ 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).

Une première (mauvaise) solution

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

Exemple (mauvaise solution)

```
prix produit::prixttc() const {
  return _prixht * (_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.

Une meilleure solution

Définir une méthode `tauxtva` dans `produit` (retourne `0.2`) et la redéfinir dans `produitculturel` (sous-classe de `produit`) (retourne `0.055`).

Méthode `produit::tauxtva` (produitv2.hh)

```
class produit {
 public:
  produit(reference ref, std::string const & nom, prix prixht);
  std::string const & nom() const;
  float tauxtva() const;
  prix prixttc() const;
```

Méthode `produit::tauxtva` (produitv2.cc)

```
float produit::tauxtva() const {
  return 0.2;
}
float produit::prixttc() const {
  return _prixht * (tauxtva() + 1);
}
```

Une meilleure solution

Méthode produitculturel::tauxtva (produitculturelv2.hh)

```
#pragma once
#include "produitv2.hh"
class produitculturel: public produit {
 public:
  produitculturel(reference ref, std::string const & nom, prix
      prixht);
  float tauxtva() const;
};
```

Méthode produitculturel::tauxtva (produitculturelv2.cc)

```
#include "produitculturelv2.hh"
produitculturel::produitculturel(reference ref, std::string const &
    nom, prix prixht)
  :produit(ref, nom, prixht) {
}
float produitculturel::tauxtva() const {
  return 0.055;
}
```

Une meilleure solution... vraiment ?

Testons les classes...

Test (produittestv2.cc)

```
#include "produitperissablev2.hh"
#include "produitculturelv2.hh"
#include <iostream>
int main() {
  produitperissable pp(1, "yaourt", 100, "15/10");
  produitculturel pc(2, "livre", 100);
  std::cout << pp.tauxtva() << " ";
  std::cout << pc.tauxtva() << "\n";
  std::cout << pp.prixttc() << " ";
  std::cout << pc.prixttc() << "\n";
  return 0;
}
```

```
0.2 0.055
120 120
```

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 `_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 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.
- Lors de la redéfinition d'une méthode **`virtual`**, utilisation du mot clef **`override`** après la déclaration de la redéfinition.

Liaison dynamique

Méthode produit::tauxtva (produitv3.hh)

```
class produit {
 public:
  produit(reference ref, std::string const & nom, prix prixht);
  std::string const & nom() const;
  virtual float tauxtva() const;
  prix prixttc() const;
```

Méthode produit::tauxtva (produitv3.cc)

```
float produit::tauxtva() const {
  return 0.2;
}
```

Liaison dynamique

Méthode produitculturel::tauxtva (produitculturelv3.hh)

```
class produitculturel: public produit {
 public:
  produitculturel(reference ref, std::string const & nom, prix
      prixht);
  float tauxtva() const override;
};
```

Méthode produitculturel::tauxtva (produitculturelv3.cc)

```
float produitculturel::tauxtva() const {
  return 0.055;
}
```

Liaison dynamique

Résultat affiché :

```
0.2 0.055
120 105.5
```

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

Liaison dynamique

Que définir `virtual` ? (produitv3.cc)

```
void produit::afficher() const {
  std::cout << _ref << " " << _nom << "\n";
  std::cout << "prix HT : " << _prixht << "\n";
  std::cout << "prix TTC : " << prixttc() << "\n";
}
```

- `prixttc` n'a pas besoin d'être déclarée `virtual`
- `afficher` non plus (vraiment ?)

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 20, 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` (20) 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` :

Calcul du prix TTC dans `produit` (produitv3.cc)

```
float produit::prixttc() const {
  return _prixht * (tauxtva() + 1);
}
```

- $\Rightarrow$ 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.

Ne pas confondre Méthode virtuelle et Méthode virtuelle pure (= Méthode abstraite).

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`.

Déclaration méthode virtuelle (produitv4.hh)

```
class produit {
 public:
  produit(reference ref, std::string const & nom, prix prixht);
  std::string const & nom() const;
  virtual float tauxtva() const =0;
```

...

Méthode virtuelle pure

- Définir `tauxtva` dans les deux classes filles de `produit`.

Produit culturel (produitculturelv4.hh)

```
class produitculturel: public produit {
 public:
  produitculturel(reference ref, std::string const & nom, prix
      prixht);
  float tauxtva() const override;
};
```

Produit culturel (produitculturelv4.cc)

```
produitculturel::produitculturel(reference ref, std::string const &
    nom, prix prixht)
  :produit(ref, nom, prixht) {
}
float produitculturel::tauxtva() const {
  return 0.055;
}
```

Méthode virtuelle pure

Produit courant (produitcourantv4.hh)

```
#pragma once
#include "produitv4.hh"
class produitcourant: public produit {
 public:
  produitcourant(reference ref, std::string const & nom, prix prixht
      );
  float tauxtva() const override;
};
```

Produit courant (produitcourantv4.cc)

```
#include "produitcourantv4.hh"
produitcourant::produitcourant(reference ref, std::string const &
    nom, prix prixht)
  :produit(ref, nom, prixht) {
}
float produitcourant::tauxtva() const {
  return 0.2;
}
```

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.