Chapitre 4

Classes : Compléments

Attributs et méthodes de classe

Polymorphisme

Surcharge d’opérateurs

Attributs et méthodes de classe

Attributs de classe - Présentation du problème

On veut modifier les classes `produit` précédentes afin d'attribuer automatiquement une référence unique à tous les produits.
⇒ Les constructeurs ne prennent plus comme paramètre un `reference ref`, et l'attribut `_ref` est initialisé automatiquement par une valeur unique.

Exemple

```
produit::produit(reference ref, std::string const & nom, prix prixht
    )
  :_ref(ref), _nom(nom), _prixht(prixht) {
}
produit::produit(std::string const & nom, prix prixht)
  :_ref(???), _nom(nom), _prixht(prixht) {
}
```

Attributs de classe - Une solution

- Utiliser un compteur
  - initialisé à 0 au début de l'exécution du programme ;
  - incrémenté à chaque instanciation de `produit`.
- Ce compteur est utilisé par le constructeur de `produit` pour récupérer une valeur unique afin d'initialiser `_ref`.
- Ce compteur devrait être « caché » afin d'être visible des seules méthodes de `produit`.
  En faire un attribut `private` de `produit` ?

Attributs de classe - Une solution qui ne fonctionne pas

Exemple (ne fonctionne pas)

```
class produit {
  public:
  produit(std::string const & nom, prix prixht);
  private:
  reference _ref; std::string _nom; prix _prixht;
  reference _compteur;
};
produit::produit(string const & nom, prix prixht)
  :_ref(_compteur), _nom(nom), _prixht(prixht) {
  _compteur++;
}
```

Code incorrect

Chaque instance de `produit` dispose d'une valeur de l'attribut `_compteur`... alors qu'un compteur partagé (par toutes les instances) est requis.

Attributs de classe

Définition (Attribut/méthode d'instance)

Un **attribut d'instance** est associé à une instance : chaque instance dispose d'une valeur pour cet attribut.
Une **méthode d'instance** s'applique sur une instance.

Définition (Attribut de classe)

Un **attribut de classe** est associé à une classe : toutes les instances de cette classe « partagent » la même valeur pour cet attribut.

Le compteur est un attribut de classe, il a une unique valeur.

Attributs et méthodes de classe - Syntaxe

Syntaxe Attribut de classe

- L'emploi du mot-clef **`static`** devant la déclaration d'un attribut fait de cet attribut un attribut de classe.
- Sans mot-clef **`static`**, c'est un attribut d'instance.

Attribut de classe - Déclaration

Déclaration (produitv5.hh)

```
#pragma once
#include <string>
using reference=unsigned int;
using prix=float;
class produit {
 public:
  produit(std::string const & nom, prix prixht);
  std::string const & nom() const;
  virtual float tauxtva() const =0;
  prix prixttc() const;
  void afficher() const;
 private:
  reference _ref;
  std::string _nom;
  prix _prixht;
  static reference _compteur;
};
```

Attribut de classe - Définition et initialisation

Attention

La seule déclaration dans le fichier `.hh` ne suffit pas. Il faut aussi définir (et initialiser) cet attribut dans le fichier `.cc` correspondant.

Cette définition (et initialisation)

- doit être faite en dehors de tout bloc de code.
- a la forme syntaxique d'une déclaration (et initialisation) de variable qui aurait pour nom `NomClasse::NomAttribut`

Attribut de classe - Définition et initialisation

Définition (produitv5.cc)

```
#include "produitv5.hh"
#include <iostream>

reference produit::_compteur(0);

produit::produit(std::string const & nom, prix prixht)
  :_ref(_compteur++), _nom(nom), _prixht(prixht) {
}
```

À l'intérieur du code d'une méthode, on accède directement à un attribut de classe de cette classe.
Écrire `this->_compteur` n'a aucun sens (même si ce n'est pas faux).

Attributs de classe

- Toutes les visibilités peuvent être appliquées aux attributs de classe.
- Si un attribut de classe est visible depuis une autre classe, on y accède en écrivant `NomClasse::NomAttribut`.
- Un attribut de classe peut être `const`ant, ce qui permet de définir des **constantes de classe**.
- La valeur d'une constante de classe peut être donnée dans le fichier `.hh` si le type de la constante est primitif.

Méthodes de classe

- Une méthode d'instance est appelée sur une instance.
- Le code d'une telle méthode peut accéder à l'objet sur lequel elle a été appelée à l'aide du pointeur `this`.

Parfois, on veut appeler une méthode sans fournir d'objet sur lequel appeler la méthode.

Méthodes de classe - Exemple

Écrire une méthode de comparaison de 2 `produits` retournant vrai si le premier est le moins cher, faux sinon.

Exemple

```
class produit {
  bool comparaison(produit const & p2) const;
};
bool produit::comparaison(produit const & p2) const {
  return _prixht < p2._prixht;
}
if (prod1.comparaison(prod2)) ...
```

$\Rightarrow$ Les deux `produits` ont le même rôle dans la comparaison, pourtant, l'un des deux est « privilégié » : on appelle la méthode sur cet objet.

Méthodes de classe

Définition (Méthode de classe)

Une **méthode de classe** est appliquée sur sa classe (et non sur une instance de sa classe).

Syntaxe méthode de classe

- L'emploi du mot-clef `static` devant la déclaration d'une méthode fait de cette méthode une méthode de classe.
- Sans mot-clef `static`, c'est une méthode d'instance.

Une méthode de classe ...

- ne peut accéder à l'objet courant `this`.
- ne peut accéder directement aux attributs (d'instance) ou appeler directement des méthodes (d'instance) sur l'objet courant.
- peut accéder aux attributs de classe ou appeler des méthodes de classe.

Méthodes de classe

Déclaration (produitv6.hh)

```
class produit {
 public:
  produit(std::string const & nom, prix prixht);
  std::string const & nom( ) const;
  virtual float tauxtva( ) const =0;
  prix prixttc( ) const;
  void afficher( ) const;
  static bool comparaison(produit const & p1, produit const & p2);
```

Définition (produitv6.cc)

```
bool produit::comparaison(produit const & p1, produit const & p2) {
  return p1._prixht < p2._prixht;
}
```

Exemple (Usage)

```
if (produit::comparaison(prod1,prod2)) ...
```

Méthodes de classe

On définira une méthode de classe :

- Quand la méthode s'applique sur plusieurs instances, et qu'aucune de ces instances n'a de rôle particulier.
- Quand la classe doit fournir un service sans nécessiter la moindre création d'instance.
- … Quand, d'un point de vue « conception », il est préférable de rattacher un comportement à la classe plutôt qu'à une instance donnée.

Polymorphisme

Polymorphisme

Lorsque des méthodes à liaison dynamique sont appelées, le simple examen du code d'un appel à une méthode ne suffit pas à déterminer quel code sera exécuté.

- Un pointeur (resp. référence) sur un objet peut en fait pointer sur (resp. référencer) une instance d'une sous-classe.
- Lors d'un appel d'une méthode à liaison dynamique sur ce pointeur (référence), la méthode redéfinie dans la sous-classe est exécutée.

Exemple. Le calcul du `prixttc` considère l'objet courant comme un `produit`, mais si une sous-classe de `produit` redéfinit `tauxtva`, c'est la méthode redéfinie qui est appelée.

Polymorphisme

Exemple

Polymorphisme - Exemple

On veut mémoriser un ensemble de `produits`.

Exemple (Une mauvaise solution)

```
class stock {
  private:
  std::vector<produitculturel> _pc;
  std::vector<produitmultimedia> _pm;
  std::vector<produitstandard> _ps;
  std::vector<produitpersissable> _pp;
  ...
};
```

- Chaque opération qui porte sur la totalité des `produits` doit faire 4 boucles.
- La définition d'une nouvelle sous-classe de `produit` demande de modifier le code de `stock`.
- À éviter

Polymorphisme - Exemple

Utiliser un seul `vector` ?

Exemple (Une solution fausse)

```
class stock {
  private:
  std::vector<produit> _p;
  ...
};
```

Dans cette solution, chaque élément du `vector` est une **instance** de `produit` (et uniquement de cette classe-là).
Le `vector` ne peut donc pas contenir des `produitculturel`, etc.
Dans une solution correcte, tous les éléments du `vector` doivent être du même type (déclaration de l'attribut) mais permettre de « contenir » des types différents (sous-classes de `produit`)
Un pointeur d'une super-classe (`produit`) peut pointer sur des instances des sous-classes (`produitculturel`, etc.)

Polymorphisme - Solution utilisant le polymorphisme

Déclaration (stockv1.hh)

```
#pragma once
#include "produitv6.hh"
#include <vector>

class stock {
  public:
  ~stock();
  void afficher() const;
  float tvamoyenne() const;
  void afficherperemptions() const;
  private:
  std::vector<produit *> _prod;
};
```

Polymorphisme - Solution utilisant le polymorphisme

- Seule la classe `produit` est utilisée.
- Nécessite d'utiliser des pointeurs de `produit` et non des instances.
  Quand le polymorphisme est utilisé, on ne peut mémoriser des instances (de quelle classe ?), il faut utiliser des pointeurs (de la classe mère, qui peuvent pointer sur des instances de toutes les classes filles).
- Les instances de produits étant manipulées par pointeur et allouées dynamiquement, il faut définir un destructeur (et un constructeur par recopie, et un opérateur d'affectation, règle des 3).

Polymorphisme - Solution utilisant le polymorphisme

Définition (stockv1.cc)

```
#include "stockv1.hh"
#include "produitperissablev6.hh"
#include <iostream>
stock::~stock() {
  for (auto p : _prod)
    delete p;
}
void stock::afficher() const {
  for (auto p : _prod)
    p->afficher();
}
float stock::tvamoyenne() const {
  float s(0);
  for (auto p : _prod)
    s += p->tauxtva();
  return s/_prod.size();
}
```

Polymorphisme - Solution utilisant le polymorphisme

Problème

Lors de l'exécution d'`afficher`, les `produitperissable` n'affichent pas leur date limite de vente.

Pourquoi ? Parce que la méthode `produit::afficher` n'a pas été définie `virtual`.
⇒ La méthode de `produit` est appelée et non sa redéfinition dans `produitperissable`.

Donc...

On déclarera toujours `virtual` les méthodes pouvant être redéfinies et destinées à être utilisées par polymorphisme.

Polymorphisme

Le cas particulier du destructeur

Polymorphisme et destructeur

Quand une instance de `stock` est détruite, le contenu de l'instance est automatiquement détruit : le `vector` et les **pointeurs** qui sont dans ce `vector`...pas les objets pointés par ces pointeurs.

Il faut donc les détruire explicitement. C'est le défaut d'utiliser des pointeurs (plutôt que des instances).

Polymorphisme et destructeur

Destructeur de `stock` (stockv1.cc)

```
stock::~stock() {
  for (auto p : _prod)
    delete p;
}
```

Tous les `produits` du stock sont détruits, le destructeur est appelé sur chacun de ces produits...

Polymorphisme et destructeur

Que se passerait-il si des sous-classes de `produit` définissaient un destructeur ?

- Le destructeur est (presque) une méthode comme les autres.
- Il est donc, par défaut, à liaison statique.
- Un élément de `_prod` est de type `produit *`, c'est donc le destructeur de `produit` qui est appelé, même si l'objet détruit est une instance d'une sous-classe.→ À éviter

Donc...

- On déclarera toujours `virtual` le destructeur d'une classe qui contient au moins une méthode `virtual`.
- Sauf si la super-classe déclare déjà un destructeur `virtual`.
- Le compilateur détecte d'ailleurs cela `deleting object of abstract class type 'produit' which has non-virtual destructor will cause undefined behavior`

Polymorphisme et destructeur

Attention. Ne pas déclarer de destructeur dans une classe revient à déclarer un destructeur qui « ne fait rien »... et qui est à liaison statique.

Exemple

```
class produit {
 ...
 virtual ~produit() =default;
 ...
};
```

Dans `produit` le destructeur « ne fait rien », mais le déclarer `virtual` permet de le redéfinir dans les sous-classes et permet l'appel du destructeur des sous-classes par polymorphisme.

Polymorphisme

`dynamic_cast`

dynamic_cast

On suppose que la classe `produitperissable` dispose d'une méthode `date const & peremption()const` accesseur à l'attribut « date de peremption ».

Produit périssable (produitperissablev6.hh)

```
#pragma once
#include "produitcourantv6.hh"
using date=std::string;
class produitperissable: public produitcourant {
 public:
  produitperissable(std::string const & nom, prix prixht, date const
       & peremption);
  date const & peremption() const {
    return _peremption; }
  void afficher() const;
 private:
  date _peremption;
};
```

dynamic_cast

Problème.

On veut écrire dans `stock` une méthode d'affichage de toutes les dates de péremption.

Exemple (qui ne fonctionne pas)

```
void stock::afficherperemtions() const {
  for (auto p : _prod)
    std::cout << p->peremption();
}
```

dynamic_cast

Ce code est incorrect car

- On veut afficher les dates de péremption uniquement pour les `produitperissable`.
- Un élément `p` de `_prod` est un `produit *` et `peremption()` n'est pas définie dans `produit`. $\Rightarrow$ Erreur de compilation.

Comment savoir si un `produit *` pointe sur un `produitperissable` ?

dynamic_cast

Définition (dynamic_cast)

dynamic_cast est un opérateur de conversion qui permet de convertir un pointeur (une référence) sur une super-classe vers un pointeur (une référence) sur une de ses sous-classes.

- Dans le cas d'une conversion de pointeur, si la conversion n'est pas possible, retourne **nullptr**.
- Dans le cas d'une conversion de référence, si la conversion n'est pas possible, lève une exception `std::bad_cast`.

dynamic_cast

afficherperemptions (stockv1.cc)

```
void stock::afficherperemptions( ) const {
  for (auto p : _prod) {
    auto pp = dynamic_cast<produitperissable const*>(p);
    if (pp != nullptr)
      std::cout << pp->peremption( );
  }
}
```

`dynamic_cast`

Utilisation de **`dynamic_cast`** :

- Ne jamais utiliser **`dynamic_cast`** sur un pointeur valant **`nullptr`**.
- Uniquement pour convertir vers une sous-classe.
- Uniquement si la classe du pointeur (ou référence) contient au moins une méthode virtuelle.
- A un coût à l'exécution (contrairement à **`reinterpret_cast`**). Mais permet de tester et de contrôler.

Polymorphisme

Constructeur virtuel

Problèmes de la classe `stock`

- La classe `stock` ne respecte pas la règle des 3.
  Un destructeur a été défini, mais pas de constructeur par recopie (ni d'opérateur d'affectation).
  Que se passe-t-il si une copie de `stock` est faite ? Une catastrophe !
- Comment ajouter des produits au `stock` ?
  De façon robuste, simple, générique.

Ajout de produits au `stock`

version 1 - incorrect (constrvirtuel/stock.hh)

```
void ajouterproduit1(produit p) {
_prod.push_back(&p);
}
```

Erreur de compilation : `cannot declare parameter ‘p’ to be of abstract type ‘produit’`

Lors d'un appel à `ajouterproduit1` avec un `produitperissable` le constructeur par recopie de `produit` est appelé et instancie donc un `produit` copie du `produitperissable`.

Même si `produit` n'était pas abstraite, ça n'aurait aucun sens : Problème de *slicing*.

Ajout de produits au `stock`

version 2 - incorrect

(constrvirtuel/stock.hh)

```
void ajouterproduit2(produitperissable p) {
_prod.push_back(&p);
}
```

Erreur à l'exécution : ce qui est rajouté au `std::vector` est un pointeur sur une variable locale.

Il faut donc ajouter au `vector` un pointeur sur un objet qui n'est pas détruit à la fin de l'exécution de la méthode.

Ajout de produits au `stock`

version 3 - incorrect (constrvirtuel/stock.hh)

```
void ajouterproduit3(produitperissable & p) {
_prod.push_back(&p);
}
```

```
produitperissable pp1("pp1", 100, "10/10");
s.ajouterproduit3(pp1);
auto pp2 = new produitperissable("pp1", 100, "10/10");
s.ajouterproduit3(*pp2);
```

Le premier ajout provoquera une erreur, pas le second.

$\rightarrow$ Code peu robuste.

Ajout de produits au `stock`

version 4 - incorrect (constrvirtuel/stock.hh)

```
void ajouterproduit4(produitperissable * p) {
_prod.push_back(p);
}
```

Ce code est toujours peu robuste :

```
produitperissable pp1("pp1", 100, "10/10");
s.ajouterproduit4(&pp1);
auto pp2 = new produitperissable("pp1", 100, "10/10");
s.ajouterproduit4(pp2);
delete pp2;
```

Même en prenant un pointeur comme paramètre, aucune garantie que ce pointeur pointera toujours sur une instance dont le `stock` est propriétaire.

Ajout de produits au `stock`

Le seul moyen de s'assurer de la propriété des produits par le `stock` est de créer dynamiquement les produits dans une méthode de `stock`, de les détruire dans une méthode de `stock` et ne jamais montrer les pointeurs à l'extérieur de `stock`.

version 5

(constrvirtuel/stock.hh)

```
void ajouterproduit5a(produitperissable const & p) {
_prod.push_back(new produitperissable(p));
}
void ajouterproduit5b(std::string const & nom, prix prixht, date
    const & peremption) {
_prod.push_back(new produitperissable(nom, prixht, peremption));
}
```

Ajout de produits au `stock`

Cette solution est correcte et robuste mais...

- Nécessite de connaître les sous-classes de `produit`.
- Nouvelle sous-classe de `produit` $\rightarrow$ rajouter une méthode à `stock`.

La structure de données utilise le polymorphisme (`std::vector<produit*>`) mais pas les méthodes d'ajout.

Ajout de produits au `stock`

- On voudrait une seule méthode d'ajout → prenant comme paramètre un `produit` (passé par référence ou pointeur).
- On voudrait appeler un constructeur pour créer un nouvel objet, copie de l'objet passé en paramètre → le constructeur par recopie de la classe dont l'objet est réellement instance, pas `produit`.

On aurait donc besoin d'un *constructeur virtuel*.
Un constructeur ne peut être `virtual`.

Constructeur virtuel

Un « constructeur virtuel » est une façon de construire un nouvel objet, qui est instance de la même classe qu'un objet existant, peu importe la façon dont l'objet existant est manipulé (par exemple référence/pointeur de super-classe).

Constructeur virtuel

(constrvirtuel/produit.hh)

```
class produit {
 public:
  produit(std::string const & nom, prix prixht);
  virtual ~produit( ) =default;
  virtual produit * clone( ) const =0;
```

Constructeur virtuel

Constructeur virtuel (constrvirtuel/produitperissable.hh)

```
class produitperissable: public produitcourant {
 public:
  produitperissable(std::string const & nom, prix prixht, date const
       & peremption);
  produit * clone() const override {
  return new produitperissable(*this);
  }
```

Utilisation (constrvirtuel/stock.hh)

```
  void ajouterproduit6(produit const & p) {
  _prod.push_back(p.clone());
  }
```

Une seule méthode d'ajout, code robuste (à condition de ne jamais montrer les pointeurs du `vector`), code réutilisable (d'autres sous-classes de `produit` peuvent être utilisées sans modifier `stock`).

Retour sur la règle des 3

Faire une copie de `stock` provoque une erreur d'exécution.

Il est possible d'empêcher la copie de certains objets en « supprimant » le constructeur par recopie (implicite).

Suppression d'une méthode implicite (constrvirtuel/stock.hh)

```
stock(stock const & s) =delete;
```

Utile quand la copie des instances d'une classe n'a pas de sens.

Sinon, il faut écrire un constructeur par recopie qui fait effectivement une copie...Et créer des copies des produits contenus dans le `stock` à copier.

Constructeur virtuel

incorrect (constrvirtuel/stock.hh)

```
stock(stock const & s) {
for (auto p : s._prod)
  _prod.push_back(new produit(*p));
}
```

Appelle le constructeur de `produit`, classe abstraite.

Constructeur virtuel

Faire une suite de tests avec des **dynamic_cast** pour appeler le constructeur de la « bonne » sous-classe de **produit**.

incorrect (constrvirtuel/stock.hh)

```
stock(stock const & s) {
for (auto p : s._prod) {
  if (dynamic_cast<produitperissable const *>(p))
    _prod.push_back(new produitperissable(*(dynamic_cast<
        produitperissable const *>(p))));
  else if (dynamic_cast< ... >(p))
    ...
}
```

Lors de l'ajout d'une nouvelle sous-classe de **produit**, si on oublie de mettre à jour le constructeur par recopie de **stock**...pas d'erreur de compilation, mais comportement incorrect.

Constructeur virtuel

Solution. Appeler le constructeur virtuel.

Appel au constructeur virtuel (constrvirtuel/stock.hh)

```
stock(stock const & s) {
for (auto p : s._prod)
  _prod.push_back(p->clone());
}
```

Ainsi `stock` est une classe dont les instances peuvent être copiées. Pour respecter la règle des 3, il faudrait définir aussi l'opérateur d'affectation.

Surcharge d’opérateurs

Opérateur d'affectation

Par défaut, il est possible de copier un objet dans un autre, en utilisant l'opérateur =.

Il ne faut pas confondre la construction par recopie (création d'un *nouvel* objet par copie d'un autre objet) et l'affectation (remplacement de la valeur d'un objet existant par la copie d'un autre objet).

Exemple (Construction et affectation)

```
fichier f1;      // Constructeur par défaut
fichier f2(f1); // Constructeur par recopie
fichier f3=f1;  // Constructeur par recopie
fichier f4;
f4 = f1;         // Affectation
```

Opérateur d'affectation : Exemple

Rappel, la classe `fichier` (fichierv3.hh)

```
struct infofichier {
  fichierproprietaire _proprietaire;
  fichierdate _creation;
};
class fichier {
  public:
  fichier(fichiernom const & nom, fichiertaille taille);
  fichier(fichiernom const & nom, fichiertaille taille,
      fichierproprietaire proprietaire, fichierdate creation);
  fichier(fichier const & f);
  ~fichier();
  private:
  fichiernom _nom;
  fichiertaille _taille;
  infofichier * _infosuppl;
};
```

Opérateur d'affectation : Exemple

Rappel, la classe `fichier` (fichierv3.cc)

```
#include "fichierv3.hh"
#include <iostream>
fichier::fichier(fichiernom const & nom, fichiertaille taille)
  :_nom(nom),_taille(taille), _infosuppl(nullptr) {
}
fichier::fichier(fichiernom const & nom, fichiertaille taille,
    fichierproprietaire proprietaire, fichierdate creation)
  :fichier(nom, taille) {
  _infosuppl = new infofichier {proprietaire, creation};
}
fichier::fichier(fichier const & f)
  :fichier(f._nom, f._taille) {
  if (f._infosuppl)
    _infosuppl = new infofichier(*f._infosuppl);
}
fichier::~fichier() {
  delete _infosuppl;
}
```

Opérateur d'affectation

Par défaut, l'opérateur d'affectation copie la valeur de tous les attributs.

```
f4._nom = f1._nom; f4._taille = f1._taille;
f4._infosuppl = f1._infosuppl;
```

Ici, il copie donc la valeur de `_infosuppl`, c'est à dire le pointeur.

$\Rightarrow$ Problème à la destruction ou à la modification.

Attention

Définir un constructeur par recopie ne règle pas le problème.

« Faire une affectation (sur un objet existant) » est différent de « Construire un nouvel objet par copie ».

Surcharge d'opérateurs

Il faut donc (re)définir le code devant être exécuté lors d'une affectation.

En C++, la plupart des opérateurs pouvant être appliqués à des instances d'une classe peuvent être redéfinis : `=`, `==`, `[ ]`, `<`, `<<`, `>=`, `>>`, etc.

Syntaxe Redéfinition d'opérateur

La redéfinition d'un opérateur se fait en déclarant et définissant une méthode ayant pour nom **`operator`** suivi de l'opérateur.

Surcharge d'opérateurs

La méthode **`operator`** est appelée sur l'objet présent en partie gauche de l'opérateur, et reçoit comme paramètre l'objet présent en partie droite de l'opérateur : `f4 = f1;` est un appel de la méthode **`operator`**`=` sur l'objet `f4`, en prenant comme paramètre `f1` : `f4.operator=(f1);`.

Exemple (Signatures habituellement utilisées)

```
class C {
  ...
  bool operator==(C const & c) const;
  bool operator<(C const & c) const;
  C & operator=(C const & c);
};
```

Opérateur d'affectation

Déclaration (fichierv4.hh)

```
class fichier {
  public:
  fichier(fichiernom const & nom, fichiertaille taille);
  fichier(fichiernom const & nom, fichiertaille taille,
      fichierproprietaire proprietaire, fichierdate creation);
  fichier(fichier const & f);
  ~fichier( );
  fichier & operator=(fichier const & f);
```

Opérateur d'affectation

Définition (fichierv4.cc)

```
fichier & fichier::operator=(fichier const & f) {
  if (this != &f) {
    _nom = f._nom;
    _taille = f._taille;
    delete _infosuppl;
    if (f._infosuppl)
      _infosuppl = new infofichier(*(f._infosuppl));
    else
      _infosuppl = nullptr;
  }
  return *this;
}
```

La classe `fichier` est un exemple de l'application de la règle des 3 : constructeurs par recopie, destructeur, opérateur d'affectation étaient tous les trois nécessaires.

Opérateur d'affectation et constructeur virtuel

L'opérateur d'affectation de `stock` doit être défini pour respecter la règle des 3.

operateur d'affectation

(constrvirtuel/stock.hh)

```
stock & operator=(stock const & s) {
if (this != &s) {
  for (auto p : _prod)
    delete p;
  _prod.clear();
  for (auto p : s._prod)
    _prod.push_back(p->clone());
}
return *this;
}
```

Opérateur de sortie

`<<` est un opérateur « comme les autres », il peut être redéfini.

Problème.

`std::cout << c` est un appel de l'opérateur `<<` sur `cout`.
Cela est d'ailleurs équivalent à `std::cout.operator<<(c)`.
Il faudrait donc redéfinir cet opérateur dans `std::ostream`!

Solution.

Les opérateurs `>>` et `<<` (comme les autres opérateurs) peuvent être définis comme des fonctions avec la signature suivante :

```
std::ostream & operator<<(std::ostream & os, C const & c);
std::istream & operator>>(std::istream & os, C & c);
```

Opérateur de sortie - Exemple

Déclaration (fichierv5.hh)

```
class fichier {
  public:
  fichier(fichiernom const & nom, fichiertaille taille);
  fichier(fichiernom const & nom, fichiertaille taille,
      fichierproprietaire proprietaire, fichierdate creation);
  fichier(fichier const & f);
  ~fichier();
  fichiernom const & nom() const { return _nom; }
  fichiertaille taille() const { return _taille; }
  fichier & operator=(fichier const & f);
  private:
  fichiernom _nom;
  fichiertaille _taille;
  infofichier * _infosuppl;
};
std::ostream & operator<<(std::ostream & os, fichier const & f);
```

Opérateur de sortie - Exemple

Le code de la fonction doit être donné dans le fichier `cc`, pas le fichier `hh`. Sinon, problème d'édition de liens : fonction définies plusieurs fois.

Définition (fichierv5.cc)

```
std::ostream & operator<<(std::ostream & os, fichier const & f) {
  os << f.nom() << " " << f.taille();
  return os;
}
```

Opérateur de sortie et hiérarchie de classes

L'opérateur de sortie est une fonction, il ne peut donc pas avoir le comportement d'une méthode virtuelle.

Exemple. Écrire l'opérateur de sortie de `stock` qui sort les informations sur les produits.

opérateur de sortie (constrvirtuel/stock.hh)

```
std::ostream & operator<<(std::ostream & os, stock const & s);
```

On suppose que les classes `produit` définissent un opérateur de sortie :

```
std::ostream & operator<<(std::ostream & os, produit const & s);
std::ostream & operator<<(std::ostream & os, produitperissable const
    & s);
std::ostream & operator<<(std::ostream & os, produitmultimedia const
    & s);
...
```

Opérateur de sortie et hiérarchie de classes

opérateur de sortie

(constrvirtuel/stock.cc)

```
std::ostream & operator<<(std::ostream & os, stock const & s) {
  for (auto p : s._prod)
    os << (*p);
  return os;
}
```

(à condition que `_prod` soit public, ce qui n'est certainement pas une bonne idée)

Pas d'erreur de compilation, mais ne fait pas ce qui est attendu : appelle systématiquement l'opérateur de `produit` (car `*p` est de type `produit` pour le compilateur).

Il faut appeler une méthode virtuelle.

Opérateur de sortie et hiérarchie de classes

méthode virtuelle de sortie (constrvirtuel/produit.hh)

```
  virtual void sortie(std::ostream & os) const;
```

méthode virtuelle de sortie (constrvirtuel/produit.cc)

```
void produit::sortie(std::ostream & os) const {
  os << _ref << " " << _nom << "\n";
  os  << "prix HT : " << _prixht << "\n";
  os  << "prix TTC : " << prixttc() << "\n";
}
```

Opérateur de sortie et hiérarchie de classes

Et la redéfinir dans les sous-classes :

méthode virtuelle de sortie (constrvirtuel/produitperissable.hh)

```
  void sortie(std::ostream & os) const override;
```

méthode virtuelle de sortie (constrvirtuel/produitperissable.cc)

```
void produitperissable::sortie(std::ostream & os) const {
  produit::sortie(os);
  os <<"peremption : " << _peremption << "\n";
}
```

Opérateur de sortie et hiérarchie de classes

opérateur de sortie (constrvirtuel/stock.cc)

```
std::ostream & operator<<(std::ostream & os, stock const & s) {
  for (auto p : s._prod)
    p->sortie(os);
  return os;
}
```

On peut ainsi écrire :

```
stock s;
s.ajouterproduit( ... );
std::cout << s;
```

Pour éviter de rendre `_prod` **`public`**, il suffit d'écrire le code de sortie de `stock` dans une méthode, appelée depuis l'opérateur de sortie.

Opérateur de sortie et hiérarchie de classes

Un `stock` peut être envoyé dans un flux mais pas un `produit`

```
produitperissable pp1( ... );
produit * p = new produitperissable( ... );
std::cout << pp1;
std::cout << (*p);
```

Erreurs de compilation.

Opérateur de sortie et hiérarchie de classes

Rien n'empêche de définir l'opérateur de sortie de `produit` pour appeler `sortie`.

opérateur de sortie (constrvirtuel/produit.cc)

```
std::ostream & operator<<(std::ostream & os, produit const & p) {
  p.sortie(os);
  return os;
}
```

Et il est inutile de définir cet opérateur sur les sous-classes de `produit` car l'opérateur de sortie de `produit` peut recevoir (par référence constante) une instance de `produit` (ou de n'importe quelle sous-classe) et appelle une méthode virtuelle.