

Chapitre IV

Classes : Compléments

- 1 Attributs et méthodes de classe
- 2 Méthodes constantes
- 3 Polymorphisme
- 4 Surcharge d'opérateurs
- 5 Divers

Chapitre IV

Classes : Compléments

- 1 Attributs et méthodes de classe
- 2 Méthodes constantes
- 3 Polymorphisme
- 4 Surcharge d'opérateurs
- 5 Divers

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 `int` `ref`, et l'attribut `m_ref` est initialisé automatiquement par une valeur **unique**.

Exemple

```
class Produit
{
private:
    int m_ref; std::string m_nom; float m_prixht;
public:
    Produit(std::string const & nom, float prixht);
    ...
};
Produit::Produit(string const & nom, float prixht)
    :m_ref(???), m_nom(nom), m_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 `m_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
{
private:
    int m_ref; std::string m_nom; float m_prixht;
    int m_compteur;
public:
    Produit(std::string const & nom, float prixht);
    ...
};
Produit::Produit(string const & nom, float prixht)
    :m_ref(m_compteur), m_nom(nom), m_prixht(prixht)
{
    m_compteur++;
}
```

Code incorrect

Chaque **instance** de `Produit` dispose d'**une valeur** de l'attribut `m_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

- 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.

Exemple (produit.h)

```
class Produit
{
private:
int m_ref;
std::string m_nom;
float m_prixht;
static int s_compteur;
...
};
```

Attributs de classe

Syntaxe de déclaration et d'initialisation

Attention

La seule déclaration dans le fichier `.h` ne suffit pas. Il faut aussi **définir** (et initialiser) cet attribut dans le fichier `.cpp` 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`

Attributs de classe

Exemple

Exemple (produit.cpp)

```
#include "produit.h"

int Produit::s_compteur = 0;

Produit::Produit(string const & nom, float prixht)
    :m_ref(++s_compteur), m_nom(nom), m_prixht(prixht)
{ }
```

À l'intérieur du code d'une méthode, on accède **directement** à un attribut de classe de cette classe.

Écrire `this->s_compteur` n'a aucun sens (même si ce n'est pas faux).

Attributs de classe

Quelques remarques

- 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 constant, ce qui permet de définir des **constantes de classe**.
- La valeur d'une constante de classe peut être donnée dans le fichier `.h` si le type de la constante est primitif.

Attributs de classe

Quelques remarques

Exemple (feu.h)

```
class Feu
{
public:
    static const int ROUGE=0;
    static const int ORANGE=1;
    static const int VERT=2;
private:
    int m_etat;
public:
    Feu();
    Feu(int etat);
    ...
};
```

Attributs de classe

Quelques remarques

Exemple (feu.cpp)

```
#include "feu.h"

Feu::Feu()
    :m_etat(ROUGE)
{}

Feu::Feu(int etat)
    :m_etat(etat)
{}

```

Exemple (Déclaration d'un feu à l'état initial « vert »)

```
Feu f(Feu::VERT);
```

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);
};
bool Produit::comparaison(Produit const & p2)
{
    return m_prixht < p2.m_prixht;
}
if (prod1.comparaison(prod2)) ...
```

⇒ 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

- 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

Exemple

```
class Produit
{
    static bool comparaison(Produit const & p1,
        Produit const & p2);
};

bool Produit::comparaison(Produit const & p1,
    Produit const & p2)
{
    return p1.m_prixht < p2.m_prixht;
}

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.

Chapitre IV

Classes : Compléments

- 1 Attributs et méthodes de classe
- 2 Méthodes constantes
 - Méthodes const
 - const_cast
- 3 Polymorphisme
- 4 Surcharge d'opérateurs
- 5 Divers

Méthodes const

Définition

Quand une méthode (d'instance) est déclarée constante :

- Elle ne peut pas **modifier** les valeurs des attributs de l'objet sur lequel elle s'applique.
- Elle ne peut pas appeler sur l'objet courant une méthode qui n'est pas constante.
- Elle peut être appelée sur un objet constant (contrairement à une méthode non constante).

Syntaxe

Le mot-clef `const` doit suivre la **déclaration** de la méthode, il fait alors partie de la **signature**.

Méthodes const

Exemple

```
class Personne
{
    private:
        std::string m_nom;
        unsigned int m_age;
    public:
        Personne(std::string const & nom, unsigned int age);
        std::string const & nom() const;
        unsigned int age(); // n'est pas constante
        static bool compare(Personne const & p1,
            Personne const & p2);
};
```

Méthodes const

Exemple

```
Personne::Personne(string const & nom,
    unsigned int age)
    :m_nom(nom), m_age(age)
{}

```

```
string const & Personne::nom() const
{
    // m_nom = "test"; // ERREUR
    // cout << age(); // ERREUR
    // cout << m_age; // OK
    return m_nom;
}

```

Remarque. Si la méthode `nom()` retournerait une référence (non constante, i.e. `string &`), la ligne `return m_nom` provoquerait une erreur.

Méthodes const

Il est autorisé d'appeler une méthode `const` sur un objet sur lequel les modifications sont autorisées.

Exemple

```
unsigned int Personne::age()
{
    // m_nom = "test"; // OK
    // cout << nom(); // OK
    return m_age;
}

bool Personne::compare(Personne const & p1,
    Personne const & p2)
{
    // return p1.age() == p2.age(); // ERREUR
    return p1.nom() == p2.nom();
}

```

Méthodes const

- On définit **toujours** une méthode `const` si son comportement ne « devrait pas » modifier l'objet.
- Un paramètre passé par référence est toujours passé par **référence constante** si la méthode qui reçoit ce paramètre ne « devrait pas » modifier l'objet.

Avantage

Le compilateur vérifie **à la compilation** que l'objet n'est pas modifié.

Problème

Si le code utilisé n'emploie pas toujours `const` là où il le devrait, certaines méthodes ne peuvent pas être appelées...

Exemple. appel à `age()` dans `compare()`

const_cast

`const_cast` est un opérateur de conversion qui est capable de faire (uniquement) les conversions suivantes :

- Convertir une **référence sur un objet constant** en une référence sur un objet **non constant**.

```
Produit const & p= ...;  
cout << const_cast<Produit &>(p).age();
```
- Convertir un pointeur sur un **objet constant** en un pointeur sur un objet **non constant**.

```
Produit const * ptr= ...;  
cout << const_cast<Produit *>(ptr)->age();
```
- Convertir un pointeur constant en un pointeur **non constant**.
Rarement utilisé.

const_cast

Exemple

```
bool Personne::compare(Personne const & p1,
    Personne const & p2)
{
    return const_cast<Personne &>(p1).age() ==
        const_cast<Personne &>(p2).age();
}
```

- Dans le cas de classes bien écrites, `const_cast` est rarement utilisé.
- Pour convertir une référence (ou pointeur) sur un objet non constant en une référence (ou pointeur) sur un objet constant, aucune conversion explicite n'est nécessaire.

const_cast

- Dans une méthode `const`, `this` est un pointeur sur un objet constant...
- Un `const_cast` peut être utilisé pour appeler une méthode non `const` sur l'objet courant.
- Mais ce n'est pas conseillé (sauf si on sait ce que l'on fait, par exemple, appeler une méthode qui ne modifie pas l'objet mais qui n'est pas déclarée `const`)

Exemple

```
string const & Personne::nom() const
{
    const_cast<Personne *>(this)->m_nom = "test"; // OK
    // MAIS A ÉVITER !
    return m_nom;
}
```

Chapitre IV

Classes : Compléments

- 1 Attributs et méthodes de classe
- 2 Méthodes constantes
- 3 Polymorphisme
 - Exemple
 - Le cas particulier du destructeur
 - `dynamic_cast`
- 4 Surcharge d'opérateurs
- 5 Divers

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

On veut mémoriser un ensemble de Produits.

Exemple (Une mauvaise solution)

```
class Stock
{
private:
    ProduitCulturel * m_pc[20];
    ProduitMultimedia * m_pm[20];
    ProduitStandard * m_ps[20];
    ProduitPerissable * m_pp[20];
    ...
};
```

- 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

Solution utilisant le polymorphisme

Exemple (stock.h)

```
class Produit;

class Stock
{
private:
    Produit* m_prod[20];
public:
    Stock();
    ~Stock();
    float TVAMoyenne() const;
    void afficherTout() const;
};
```

Polymorphisme

Solution utilisant le polymorphisme

Exemple (stock.cpp)

```
#include "produit.h"

Stock::Stock()
{
    for (unsigned int i=0; i<20; i++)
        m_prod[i] = NULL;
}

Stock::~~Stock()
{
    for (unsigned int i=0; i<20; i++)
        if (m_prod[i] != NULL)
            delete m_prod[i];
}
```

Polymorphisme

Solution utilisant le polymorphisme

Exemple (stock.cpp)

```
// On suppose Produit::tauxTVA() publique.
float Stock::TVAMoyenne() const
{
    float somme=0; unsigned int nb=0;
    for (unsigned int i=0; i<20; i++)
        if (m_prod[i] != NULL)
        {
            somme += m_prod[i]->tauxTVA();
            nombre++;
        }
    return somme/nombre;
}

void Stock::afficherTout() const
{ for (unsigned int i=0; i<20; i++)
    if (m_prod[i] != NULL)
        m_prod[i]->afficher();
}
```

Polymorphisme

Solution utilisant le polymorphisme

Problème

Lors de l'exécution d'afficherTout, 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 et destructeur

Exemple

```
Stock::~~Stock()
{
    for (unsigned int i=0; i<20; i++)
        if (m_prod[i] != NULL)
            delete m_prod[i];
}
```

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 ?

(par exemple pour libérer une allocation dynamique)

- Le destructeur est (presque) une méthode comme les autres.
- Il est donc, par défaut, **à liaison statique**.
- `m_prod[i]` 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`.

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

Polymorphisme et destructeur

Exemple

```
class Produit
{
    ...
    virtual ~Produit();
    ...
};
```

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.

dynamic_cast

On suppose que la classe `ProduitPerissable` dispose d'une méthode `string const & DLV()`, accesseur à l'attribut « date limite de vente ».

Problème.

On veut écrire dans `Stock` une méthode d'affichage de toutes les dates limites de vente.

Exemple (Qui ne fonctionne pas)

```
void Stock::afficherDLV() const
{
    for (unsigned int i=0; i<20; i++)
        if (m_prod[i] != NULL)
            cout << m_prod[i]->DLV();
}
```

dynamic_cast

Ce code est incorrect car

- On veut afficher les DLV **uniquement** pour les `ProduitPerissable`.
- `m_prod[i]` est un `Produit *` et `DLV()` n'est pas définie dans `Produit`. ⇒ **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 NULL.
- Dans le cas d'une conversion de **référence**, si la conversion n'est pas possible, lève une exception `std::bad_cast`.

dynamic_cast

Exemple

```
void Stock::afficherDLV() const
{
    for (unsigned int i=0; i<20; i++)
        if (m_prod[i] != NULL)
        {
            ProduitPerissable const * pp =
                dynamic_cast<ProduitPerissable const *>(m_prod[i]);
            if (pp != NULL)
                cout << pp->DLV();
        }
}
```

dynamic_cast

Utilisation de `dynamic_cast` :

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

Chapitre IV

Classes : Compléments

- 1 Attributs et méthodes de classe
- 2 Méthodes constantes
- 3 Polymorphisme
- 4 Surcharge d'opérateurs
- 5 Divers

Surcharge d'opérateurs

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

Exemple

```
class Fichier
{
private:
    std::string m_nom;
    unsigned int m_taille;
    unsigned int * m_utilisateurs;
    ...
};

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

Opérateur d'affectation

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

```
f3.m_nom = f1.m_nom; f3.m_taille = f1.m_taille;
f3.m_utilisateurs = f1.m_utilisateurs;
```

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

⇒ 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

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

Exemple (Signatures habituellement utilisées)

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

Opérateur d'affectation

Exemple

```
class Fichier
{
    ...
    public:
    Fichier const & operator=(Fichier const & c);
};

Fichier const & Fichier::operator=(Fichier const & c)
{
    if (this != &c)
    {
        for (int i=0; i<20; i++)
            m_utilisateurs[i]=c.m_utilisateurs[i];
    }
    return *this;
}
```

Il est conseillé de définir toujours un constructeur par copie et un operator=, sauf si leur comportement par défaut est celui qui est attendu.

Opérateur de sortie

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

Problème.

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

Solution.

Les opérateurs >> et << **sur des flux** peuvent surchargés comme des **fonctions** avec la signature suivante :

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

Opérateur de sortie

Exemple

Exemple (fichier.h)

```
class Fichier
{
    ...
};
std::ostream & operator<<(std::ostream & os,
    Fichier const & f);
```

Exemple (fichier.cpp)

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

Chapitre IV

Classes : Compléments

- 1 Attributs et méthodes de classe
- 2 Méthodes constantes
- 3 Polymorphisme
- 4 Surcharge d'opérateurs
- 5 Divers
 - Déclaration avancée
 - `friend`
 - Définition de méthodes dans la déclaration de classe

Déclaration avancée

Habituellement, une classe est **déclarée** dans un fichier `.h` et **définie** dans un fichier `.cpp` de même nom.

- Pour hériter d'une classe `C`, pour déclarer un attribut de type `C`, pour utiliser `C` comme paramètre (passé par valeur) d'une déclaration de méthode ou valeur de retour (par valeur) d'une déclaration de méthode, pour appeler une méthode de `C`...
Le compilateur a besoin de la déclaration de la classe ⇒ Inclure le fichier `.h`
- Pour déclarer un pointeur (ou une référence) sur un `C`...
Le compilateur a besoin de savoir que `C` **est une classe**.
⇒ Il n'est pas nécessaire d'inclure le fichier `.h`
Une **déclaration avancée** suffit.

Déclaration avancée

Exemple (stock.h)

```
class Produit;

class Stock
{
private:
    Produit* m_prod[20];
}; ...
```

Exemple (stock.cpp)

```
#include "produit.h"

Stock::~Stock()
{
    for (unsigned int i=0; i<20; i++)
        if (m_prod[i] != NULL)
            delete m_prod[i];
}
```

friend

Problème.

Rendre visibles à une fonction *f* ou aux méthodes d'une classe *D* les membres privés d'une classe *C*... sans les montrer aux autres fonctions/classes de l'application.

Exemple.

Montrer l'attribut *m_utilisateurs* de *Fichier* dans l'opérateur de sortie de *Fichier*.

Syntaxe

La déclaration dans une classe *D* de :

- `friend NomClasse ;`
rend les membres privés de *D* visibles par toutes les méthodes de *NomClasse*.
- `friend typeretour nomfonction(arguments) ;`
rend les membres privés de *D* visibles par la fonction *nomfonction*.

Code de méthodes dans la déclaration

Il est possible de donner la **définition** d'une méthode en même temps que sa déclaration, dans un fichier .h.

Exemple (produitperissable.h)

```
class ProduitPerissable: public Produit
{
private:
    std::string m_datelimite;

public:
    ProduitPerissable(int ref, std::string const & nom,
float prixht, std::string const & datelimite)
        :Produit(ref, nom, prixht),
          m_datelimite(datelimite)
    {}
    std::string const & DLV() const
    { return m_datelimite; };
    ...
};
```

Code de méthodes dans la déclaration

Cas général.

Que se passe-t-il quand le compilateur compile un appel à une méthode ?

- Il génère un code permettant l'appel d'une méthode, en utilisant la pile.
- Si la méthode est définie dans un autre fichier, il écrit dans le fichier .o une référence au symbole correspondant à la méthode. Résolu à l'édition de liens.
- Dans le cas d'une méthode à liaison dynamique, code particulier.

Si la méthode a été écrite dans la déclaration de la classe.

- Il écrit dans le fichier .o le code compilé de la méthode.

Code de méthodes dans la déclaration

Conséquences

- L'exécutable sera plus gros.
- Le code s'exécutera plus rapidement (souvent).
- **Cela n'est pas utilisable sur des méthodes à liaison dynamique.**

On utilisera cette possibilité pour des méthodes dont le code est très court, et qui ne sont pas redéfinies dans les sous-classes.

⇒ Les accesseurs « simples ».