

Chapitre II

Classes : Concepts de base

- 1 Introduction
- 2 Syntaxe
- 3 Encapsulation
- 4 Constructeur
- 5 Destructeur

Chapitre II

Classes : Concepts de base

- 1 Introduction
- 2 Syntaxe
- 3 Encapsulation
- 4 Constructeur
- 5 Destructeur

Classe

- Ensemble d'objets ayant des **propriétés communes**
 - Chaque objet d'une classe (**instance**) a les mêmes **attributs**.
 - Chaque objet peut avoir des **valeurs** différentes des **attributs** qui représentent l'état de l'objet.
 - Tous les objets d'une classe ont le même comportement : **méthodes**.

On appelle **membres** d'une classe les attributs et les méthodes de la classe.

Classe

- Habituellement, en C++, on sépare l'**interface** de la classe de son **implantation**.
- L'interface (la **déclaration**) d'une classe est écrite dans un fichier d'entêtes `.h`.
- L'implantation (la **définition**) (code des méthodes) est écrite dans un fichier `.cpp`.
- Il est conseillé de donner le même nom aux deux fichiers.

- Une classe est formée de *membres* : *attributs* et *méthodes*.
- L'interface d'une classe doit donner les noms des membres. . .
 - Pour les attributs, le **type** doit être donné
 - Pour les méthodes, la **signature** doit être donnée
- L'implantation d'une classe doit donner le **corps** (code) des méthodes.

Chapitre II

Classes : Concepts de base

- 1 Introduction
- 2 **Syntaxe**
- 3 Encapsulation
- 4 Constructeur
- 5 Destructeur

Classe

Syntaxe (Déclaration)

```
class NomClasse
{
    typemembre nomattribut;
    typeretour nommethode(typearg1 nomarg1, ...);
    ...
};
```

Classe

Exemple (fichier.h)

```
#ifndef fichier_h
#define fichier_h

#include <string>

class Fichier
{
private:
    std::string m_nom;
    unsigned int m_taille;

public:
    void afficher();
    unsigned int taille();
    std::string nom();
    void renommer(std::string nn);
    void fixerTaille(unsigned int nt);
};

#endif
```

Classe

Exemple (fichier.cpp)

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

void Fichier::afficher()
{
    std::cout << m_nom << " (";
    if (m_taille == 0)
        std::cout << "vide)";
    else
        std::cout << taille() << " octets)";
}

unsigned int Fichier::taille()
{
    return m_taille;
}
```

Classe

Exemple (fichier.cpp)

```
std::string Fichier::nom()
{
    return m_nom;
}

void Fichier::renommer(std::string nn)
{
    m_nom = nn;
}

void Fichier::fixerTaille(unsigned int nt)
{
    m_taille = nt;
}
```

Classe

Déclaration d'instance

Une fois définie, la classe peut être utilisée comme un type.

Syntaxe (Déclaration)

- *NomClasse NomDeLInstance* ;
- *NomClasse * NomDuPointeur* ;
Attention. Seul un pointeur (non initialisé) est déclaré.
- *NomClasse & NomDeLaReference* ;
Attention. Obligation de fournir la variable référencée (sauf dans une signature).

Classe

Accès aux membres

Syntaxe (Accès à un attribut)

NomDeLInstance.NomDeLAttribut
NomDuPointeur->NomDeLAttribut
*(*NomDuPointeur).NomDeLAttribut*
NomDeLaReference.NomDeLAttribut

(sous réserve de visibilité)

Dans une **méthode**, on peut utiliser **directement un nom d'attribut** pour désigner l'attribut correspondant **de l'instance sur laquelle la méthode a été appelée...**

Ou utiliser le pointeur `this` qui repère l'instance courante.

Classe

Accès aux membres

Syntaxe (Appel à une méthode)

```
NomDeLInstance.NomDeLaMethode (arguments)
NomDuPointeur->NomDeLaMethode (arguments)
(*NomDuPointeur).NomDeLaMethode (arguments)
NomDeLaReference.NomDeLaMethode (arguments)
```

(sous réserve de visibilité)

Dans une **méthode**, on peut utiliser **directement** un nom de méthode pour appeler cette méthode **sur l'instance sur laquelle la méthode a été appelée...**

Ou utiliser le pointeur `this` qui repère l'instance courante.

Classe

Exemple

```
#include "fichier.h"

int main()
{
    Fichier f;
    f.renommer("exemple.txt");
    f.fixerTaille(2567);
    f.afficher();
    return 0;
}
```

Chapitre II

Classes : Concepts de base

- 1 Introduction
- 2 Syntaxe
- 3 Encapsulation
- 4 Constructeur
- 5 Destructeur

Encapsulation

- L'**encapsulation** permet de « cacher » des membres d'une classe afin que l'**interface** de la classe ne dispose que des membres qui ont été choisis par le concepteur de la classe.
- L'encapsulation facilite la mise au point, la réutilisation, et l'évolution.

Encapsulation

- En C++, l'encapsulation permet de choisir parmi 3 niveaux de visibilité :
 - `private`
Le membre est visible dans toutes les méthodes de la classe et invisible ailleurs.
 - `protected`
Le membre est visible dans toutes les méthodes de la classe (ainsi que dans les méthodes des sous-classes), et invisible ailleurs.
 - `public`
Le membre est visible partout.

Encapsulation

Habituellement, tous les **attributs** d'une classe sont déclarés `private`, et **certaines méthodes** sont déclarées `public`.

Syntaxe (Visibilité)

```
class NomClasse
{
  [private | protected | public]:
  typemembre nommembre...;
  ...;
};
```

Tous les membres qui sont déclarés **après** une étiquette de visibilité ont cette visibilité là. Les membres qui sont déclarés avant la première étiquette de visibilité sont privés.

Exemple

`m_nom` et `m_taille` sont privés, les 5 méthodes sont publiques.

Encapsulation

Un attribut public peut être lu et modifié (comme une variable). Parfois, on veut montrer la valeur de l'attribut sans autoriser la modification.

⇒ Rendre l'attribut privé, et fournir un *accesseur* (en lecture).

Définition

- Un *accesseur* (ou accesseur en lecture) est une méthode qui retourne la valeur d'un attribut (privé).
- Un *mutateur* (ou accesseur en écriture) est une méthode qui permet de modifier la valeur d'un attribut (privé).

Exemple

`Fichier::nom()` est un accesseur, `Fichier::fixerTaille()` est un mutateur.

Chapitre II

Classes : Concepts de base

- 1 Introduction
- 2 Syntaxe
- 3 Encapsulation
- 4 **Constructeur**
 - Présentation
 - Liste d'initialisations
 - Constructeur par défaut et par copie
- 5 Destructeur

Constructeur

Un **constructeur** est une **méthode** qui est appelée pour **construire** (initialiser) une **instance** d'une classe.

Intérêt

Imposer l'exécution d'une méthode pour s'assurer de l'initialisation de l'objet.

Exemple

```
Fichier f;  
f.renommer("exemple.txt");  
f.afficher(); // f.m_taille n'est pas initialisé.
```

Constructeur

- Un constructeur est une méthode qui a comme nom **le nom de la classe** et qui **ne retourne rien** (\neq qui retourne void).
- Un constructeur est souvent public.
- Une classe peut avoir **plusieurs** constructeurs (avec des signatures différentes).
Dans ce cas, on choisit le constructeur à utiliser au moment de créer une instance.

Constructeur

Syntaxe (Constructeurs)

```
class NomClasse
{
    ...
    // Constructeur par défaut.
    NomClasse();
    // Constructeur avec paramètres.
    NomClasse(typearg1 nomarg1, ...);
    // Constructeur par copie.
    NomClasse(NomClasse const & nomarg);
    ...
};
```

Constructeur

Exemple (fichier.h)

```
class Fichier
{
    ...
    public:
    Fichier(std::string nom, unsigned int taille);
    ...
};
```

Exemple (fichier.cpp)

```
Fichier::Fichier(string nom, unsigned int taille)
{
    m_nom = nom;
    m_taille = taille;
}
```

Constructeur

Liste d'initialisations

Habituellement, un constructeur d'une classe se charge d'initialiser les attributs de l'instance en cours de construction.

Ces initialisations peuvent être réalisées **avant** même l'exécution de la première ligne de code de la méthode du constructeur.

Exemple (fichier.cpp)

```
Fichier::Fichier(string nom, unsigned int taille)
    :m_nom(nom), m_taille(taille)
{
}
```

Constructeur

Liste d'initialisations

La liste d'initialisations est formée du caractère :, des attributs à initialiser séparés par des , chaque attribut étant suivi de la valeur d'initialisation :

- Si l'attribut est d'un type primitif (`unsigned int`), syntaxe C++ d'initialisation.
- Si l'attribut est d'un type classe (`std::string`)
 - Initialisation explicite par appel à un constructeur
→ les paramètres passés doivent correspondre à un constructeur de la classe.
 - Si aucune initialisation n'est donnée, le *constructeur par défaut* (sans argument) de l'attribut est utilisé.

Ici `std::string` dispose d'un constructeur prenant comme paramètre une instance de `std::string`, c'est ce constructeur qui est appelé.

Constructeur

Liste d'initialisations

Différences entre :

```
Fichier::Fichier(string nom, unsigned int taille)
{ m_nom = nom;
  m_taille = taille; }
```

et

```
Fichier::Fichier(string nom, unsigned int taille)
:m_nom(nom), m_taille(taille)
{ }
```

- Dans le premier cas, le **constructeur par défaut** de string est utilisé, et construit une chaîne vide. Puis l'**opérateur d'affectation** = est utilisé et copie la valeur du paramètre nom dans l'attribut m_nom.
- Dans le second cas, le constructeur de string prenant comme paramètre une string est appelé pour **construire directement** m_nom avec une copie de nom.

Constructeur

Liste d'initialisation

Il est toujours préférable d'utiliser une liste d'initialisations :

- Séparer les initialisations du code du constructeur.
- Plus rapide.
- Parfois indispensable (attribut référence).
- Quand le code du constructeur est exécuté, les attributs sont déjà initialisés.

Constructeur

Si **aucun constructeur** n'est déclaré dans une classe, elle dispose tout de même d'un constructeur par défaut qui :

- Appelle le constructeur par défaut sur tous les attributs de types classes.
- Ne fait rien pour les attributs de types primitifs.

Si (au moins) **un constructeur** a été déclaré dans une classe, **le constructeur par défaut implicite n'est plus disponible**. Mais un constructeur par défaut peut être déclaré.

Exemple

```
// Fichier f; // Provoque une erreur
Fichier f("exemple.txt", 2567);
f.afficher();
```

Constructeur par recopie

Un **constructeur par recopie** permet d'effectuer une copie d'une instance pour créer une nouvelle instance.

Il est utilisé dans les cas suivants :

- Créer une nouvelle variable comme copie d'une variable existante.

Exemple

```
Fichier f2(f);
```

...

Constructeur par copie

...

- Passer une **copie** d'une variable comme **paramètre** à une fonction ou méthode.

Exemple

```
int Fichier::comparer(Fichier b)
{
    return ( b.m_taille > m_taille ) ? 2 : 1;
}
...
if (f1.comparer(f2) == 1) ...
```

Une *copie* de f2 est passée à `comparer` afin que d'éventuelles modifications de b dans `comparer` ne modifient pas f2.

Constructeur par copie

Toute classe dispose d'un constructeur par copie **implicite** qui :

- Appelle le constructeur par copie sur tous les attributs de types classes.
- Copie la valeur des attributs de types primitifs.

Il est possible de définir explicitement le constructeur par copie.

Attention lors de la déclaration d'un constructeur par copie

Le constructeur par copie ne prend pas comme paramètre une **instance** de la classe :

```
Fichier(Fichier f);
```

Mais une **référence** (constante) sur une instance de la classe :

```
Fichier(Fichier const & f);
```

Chapitre II

Classes : Concepts de base

- 1 Introduction
- 2 Syntaxe
- 3 Encapsulation
- 4 Constructeur
- 5 Destructeur

Destructeur

Le **destructeur** est une **méthode** qui est appelée **implicitement** sur une instance, quand celle-ci **cesse d'exister**.

Intérêt

Imposer l'exécution d'une méthode pour s'assurer de la destruction de l'objet.

- Fermer des fichiers qui ont été ouverts dans les méthodes de la classe.
- Fermer une connexion réseau.
- **Libérer les allocations dynamiques.**

Destructeur

- Un destructeur est une méthode qui a comme nom **le caractère ~ suivi du nom de la classe**, qui n'a **aucun paramètre** et qui **ne retourne rien** (\neq qui retourne void).
- Un destructeur est habituellement public.
- Une classe ne peut avoir qu'**un** seul destructeur.
- Le destructeur n'est **jamais appelé explicitement**.
- Quand une instance est détruite, les objets qui **composent** cette instance sont détruits implicitement.
Mais pas les objets qui sont **pointés** par un pointeur qui compose l'instance → À faire dans le destructeur.

Destructeur

Exemple. Rajouter les identifiants (unsigned int) des 20 derniers utilisateurs ayant lu le fichier.

Exemple (fichier.h)

```
class Fichier
{
private:
    ...
    unsigned int * m_utilisateurs;
    ...
};
```

Exemple (fichier.cpp)

```
Fichier::Fichier(string nom, unsigned int taille)
:m_nom(nom),m_taille(taille),
 m_utilisateurs(new unsigned int[20])
{
    for (int i=0; i<20; i++)
        m_utilisateurs[i]=0;
};
```

Destructeur

Problème

Quand une instance de `Fichier` cesse d'exister, le bloc de 20 entiers continue à être réservé.

- Il ne peut pas être détruit ailleurs
(par un `delete [] (f.m_utilisateurs);`)
car l'attribut est privé.
- On aimerait automatiser la destruction.

Destructeur

Exemple (fichier.h)

```
class Fichier
{
  ...
public:
  ~Fichier();
  ...
};
```

Exemple (fichier.cpp)

```
Fichier::~~Fichier()
{
  delete [] m_utilisateurs;
}
```

De façon générale, tout ce qui est alloué dynamiquement dans la classe pour être stocké dans les attributs de la classe doit être libéré dans le destructeur.

Constructeur par recopie et destructeur

Exemple (main)

```
{
  Fichier f1(...), f2(...);
  ...
  ... f1.comparer(f2) ...
  ...
}
```

- L'appel à comparer crée une copie de f2 (dans b).
- Le **constructeur par recopie implicite** est appelé.
- Il appelle le constructeur par recopie de la string m_nom, copie la valeur de l'entier m_taille et copie **la valeur du pointeur** m_utilisateurs.
- Quand comparer se termine, le paramètre b est détruit, le destructeur est appelé (implicitement).
- Ce destructeur libère b.m_utilisateurs... qui était la zone d'entiers utilisée par f2.
- **Tout accès à f2.m_utilisateurs est incorrect.**

Constructeur par recopie et destructeur

⇒ Définir un constructeur par recopie... qui recopie les utilisateurs (et pas simplement un pointeur sur une zone).

Exemple (fichier.h)

```
class Fichier
{
  ...
public:
  Fichier(Fichier const & f);
  ...
};
```

Exemple (fichier.cpp)

```
Fichier::Fichier(Fichier const & f)
:m_nom(f.m_nom), m_taille(f.m_taille),
 m_utilisateurs(new unsigned int[20])
{
  for (int i=0; i<20; i++)
    m_utilisateurs[i]=f.m_utilisateurs[i];
}
```