Chapitre 2

Classes : Concepts de base

Introduction

Syntaxe

Encapsulation

Constructeur

Destructeur

Méthodes constantes

Introduction

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 `.hh`.
- L'implantation (la définition) (code des méthodes) est écrite dans un fichier `.cc`.
- Il est conseillé de donner le même nom aux deux fichiers.

Classe

- 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.
  Dans le cas de méthodes simples, on s'autorise à donner le corps dans le fichier d'interface.

Syntaxe

Classe

Syntaxe Déclaration

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

Classe

Déclaration de classe (fichier1.hh)

```
#pragma once
#include <string>

class Fichier {
  public:
  void afficher();
  unsigned int taille() {
    return _taille; }
  std::string nom() {
    return _nom; }
  void renommer(std::string const & nn);
  void fixerTaille(unsigned int nt);

  private:
  std::string _nom;
  unsigned int _taille;
};
```

Classe

Définition de classe (fichier1.cc)

```
#include "fichier1.hh"
#include <iostream>

void Fichier::afficher() {
  std::cout << this->_nom << " (";
  if (_taille == 0)
     std::cout << "vide)";
  else
     std::cout << _taille << " octets)";
}

void Fichier::renommer(std::string const & nn) {
  _nom = nn;
}

void Fichier::fixerTaille(unsigned int nt) {
  _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).

C++ n'est pas Java : On préfèrera quand c'est possible manipuler les objets par valeur (et non par pointeur).

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. C'est un pointeur, donc on doit utiliser l'opérateur `->`.

Classe

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.

Instanciation et utilisation d'un classe (mainfichier1.cc)

```
#include "fichier1.hh"

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

Encapsulation

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

`_nom` et `_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.

Constructeur

Constructeur

Présentation

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._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 recopie.
  NomClasse(NomClasse const & nomarg);
  ...
};
```

Constructeur

Constructeur - déclaration (fichier.hh)

```
class Fichier {
  public:
  Fichier(std::string const & nom, unsigned int taille);
  void afficher();
```

Constructeur - définition (fichier.cc)

```
Fichier::Fichier(std::string const & nom, unsigned int taille) {
  _nom = nom;
  _taille = taille;
}
```

Constructeur

Liste d'initialisations

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.

Constructeur - définition (fichier.cc)

```
Fichier::Fichier(std::string const & nom, unsigned int taille)
  :_nom(nom),_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
    $\rightarrow$ 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(std::string const & nom, unsigned int taille) {
  _nom = nom;
  _taille = taille;
}
```

et

```
Fichier::Fichier(std::string const & nom, unsigned int taille)
  :_nom(nom),_taille(taille) {
}
```

- Dans le premier cas, le constructeur par défaut de `std::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 `_nom`.
- Dans le second cas, le constructeur de `std::string` prenant comme paramètre une `std::string` est appelé pour construire directement `_nom` avec une copie de `nom`.

Constructeur - Constructeur délégué

La liste d'initialisations peut aussi être utilisée pour définir un constructeur délégué, c'est à dire un constructeur qui fait appel à un autre constructeur (cible).
Dans ce cas, la liste d'initialisations ne contient qu'un appel à un appel à un (autre) constructeur de la même classe.

Constructeur - Constructeur délégué

Constructeur délégué (fichierconstrdelegue.hh)

```
#pragma once
#include <string>
struct fichierexterne { // struct externe à notre projet
  char const * nom;
  long taille;
};
class Fichier {
  public:
  Fichier(std::string const & nom, unsigned int taille)
   :_nom(nom), _taille(taille) {}
  Fichier(fichierexterne const & info)
   :Fichier(info.nom, static_cast<unsigned int>(info.taille)) {}
  private:
  std::string _nom;
  unsigned int _taille;
};
```

Constructeur délégué (mainfichierconstrdelegue.cc)

```
#include "fichierconstrdelegue.hh"

int main() {
  fichierexterne fe {".bashrc", 742};
  Fichier f(fe);
  return 0;
}
```

Constructeur - Liste d'initialisations

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

- Séparer les initialisations du code du constructeur.
- Plus rapide.
- Parfois indispensable : attribut référence, attribut ne disposant pas d'un constructeur par défaut.
- Quand le code du constructeur est exécuté, les attributs sont déjà initialisés.

Constructeur

Constructeur par défaut et par recopie

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.

Constructeur et usage (mainfichier.cc)

```
int main() {
  // Fichier f; // Provoque une erreur
  Fichier f1("ex", 25);
  f1.renommer("exemple.txt");
  f1.fixertaille(2567);
  f1.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 recopie

...

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

Passage par valeur (fichier.cc)

```
bool Fichier::estpluspetitque(Fichier f) {
  return _taille < f._taille;
}
```

Constructeur et usage (mainfichier.cc)

```
Fichier f2(f1);
std::cout << f1.estpluspetitque(f2) << "\n";
```

Une copie de `f2` est passée à `estpluspetitque` afin que d'éventuelles modifications du paramètre `f` dans `estpluspetitque` ne modifient pas `f2`.

$\rightarrow$ Passage par valeur = copie.

Constructeur par recopie

Toute classe dispose d'un constructeur par recopie implicite qui :

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

Il est possible de définir explicitement le constructeur par recopie.

Attention lors de la déclaration d'un constructeur par recopie

Le constructeur par recopie 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);
```

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 $\rightarrow$ À faire dans le destructeur.

Destructeur

Exemple. Rajouter une information optionnelle aux fichiers : l'identifiant du propriétaire et la date de création. Cette information est optionnelle, tous les `fichier` n'en disposent pas.

Informations sur fichiers (fichier2.hh)

```
#pragma once
#include <string>
using fichierproprietaire = unsigned int;
using fichierdate = unsigned int;
using fichiernom = std::string;
using fichiertaille = unsigned int;

struct infofichier {
  fichierproprietaire _proprietaire;
  fichierdate _creation;
};
```

Destructeur

Informations sur fichiers (suite) (fichier2.hh)

```
class fichier {
  public:
  fichier(fichiernom const & nom, fichiertaille taille);
  fichier(fichiernom const & nom, fichiertaille taille,
      fichierproprietaire proprietaire, fichierdate creation);
  void afficher();
  fichiertaille taille() {
    return _taille; }
  fichierdate creation();
  bool estpluspetitque(fichier f);
  private:
  fichiernom _nom;
  fichiertaille _taille;
  infofichier * _infosuppl;
};
```

Destructeur

Informations sur fichiers (suite) (fichier2.cc)

```
#include "fichier2.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};
}

void fichier::afficher() {
  std::cout << _nom << " (";
  std::cout << _taille << " octets)";
  if (_infosuppl)
    std::cout << " création " << creation();
}
```

Destructeur

Informations sur fichiers (suite) (fichier2.cc)

```
fichierdate fichier::creation( ) {
  if (_infosuppl)
    return _infosuppl->_creation;
  else
    return 0;
}

bool fichier::estpluspetitque(fichier f) {
  return _taille < f._taille;
}
```

Destructeur

Problème

Quand une instance de `fichier` cesse d'exister, l'`infofichier` associé continue d'exister.

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

Bonne pratique : Propriété d'un objet

Quand une instance A d'une classe est propriétaire d'une instance B, c'est (uniquement) une méthode de A qui doit créer B, et c'est (uniquement) une méthode de A qui doit libérer B.

Destructeur

destructeur (fichier2bis.hh)

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

destructeur (fichier2bis.cc)

```
fichier::~fichier() {
  delete _infosuppl;
}
```

De façon générale, tout ce qui est alloué dynamiquement dans une méthode pour être stocké dans un attribut est une propriété de l'instance, et doit donc être détruit avec l'instance, dans le destructeur.

Constructeur par recopie et destructeur

Erreur à l'exécution (mainfichier2bis.cc)

```
std::cout << f1.estpluspetitque(f2) << "\n";
```

```
Error in `./a.out': double free or corruption (fasttop)
```

- L'appel à `estpluspetitque` crée une copie de `f2` (dans `f`).
- Le constructeur par recopie implicite est appelé.
- Il appelle le constructeur par recopie de `string _nom`, copie la valeur de l'`int _taille` et copie la valeur du pointeur `_infosuppl`. Un pointeur est un type primitif.
- Quand `estpluspetitque` se termine, le paramètre `f` est détruit, le destructeur est appelé (implicitement).
- Ce destructeur libère `f._infosuppl`... qui est aussi l'`infofichier` utilisé par `f2`.
- Tout accès à `f2._infosuppl` est incorrect.

Constructeur par recopie et destructeur

⇒ Définir un constructeur par recopie... qui recopie l'`infofichier` (et pas simplement un pointeur sur un `infofichier`).

constructeur par recopie (fichier2ter.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();
```

constructeur par recopie (fichier2ter.cc)

```
fichier::fichier(fichier const & f)
  :fichier(f._nom, f._taille) {
  if (f._infosuppl) // nullptr est considéré comme faux
    _infosuppl = new infofichier(*f._infosuppl); // copie de la
        struct
}
```

Constructeur par recopie et destructeur

Règle des 3

Quand dans une classe, on écrit l'une de ces méthodes, on écrit les 3 :

- Constructeur par recopie
- Destructeur
- Opérateur d'affectation (`operator=`)

Constructeur par recopie et destructeur

Mais dans la majorité des cas, on essaie d'appliquer la règle des 0.

Règle des 0

Sauf si c'est vraiment justifié, dans une classe on n'écrit ni constructeur par recopie, ni destructeur, ni opérateur d'affectation.

Ce qui ne veut pas dire qu'il n'y a pas de constructeur par recopie ou de destructeur !
Mais que c'est le comportement par défaut du constructeur par recopie/destructeur qui est attendu. Il est possible de l'écrire explicitement :

Exemple

```
class C {
  public:
  C(C const & c)=default;
  ~C( )=default;
```

Méthodes constantes

Méthodes constantes

Méthodes `const`

Méthodes `const`

Définition

Quand une méthode 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 Méthode constante

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

Méthodes const

Méthodes const (personnev1.hh)

```
#pragma once
#include <string>

class personne {
  public:
    personne(std::string const & nom, unsigned int age)
      :_nom(nom), _age(age) {}
    std::string const & nom() const;
    unsigned int age(); // n'est pas const
    bool estplusjeuneque(personne const & p2) const;
  private:
    std::string _nom;
    unsigned int _age;
};
```

Méthodes const

Exemple

```
std::string const & personne::nom() const {
 // _nom = "test"; // erreur
 // std::cout << age(); // erreur
 // std::cout << _age; // OK
 return _nom;
}
```

Remarque. Si la méthode `nom()` retournait une référence (non constante, i.e. `std::string &`), la ligne `return _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() {
 // _nom = "test"; // OK
 // std::cout << nom(); // OK
 return _age;
}
```

Méthodes const

(personnev1.cc)

```
bool personne::estplusjeuneque(personne const & p2) const {
  return age() < p2.age(); // erreur !
}
```

```
passing 'const personne' as 'this' argument discards
qualifiers
```

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 `estplusjeuneque()`.

Méthodes constantes

`const_cast`

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.

```
personne const & p= ...;
std::cout << const_cast<personne &>(p).age();
```

- Convertir un pointeur sur un objet constant en un pointeur sur un objet non constant.

```
personne const * ptr= ...;
std::cout << const_cast<personne *>(ptr)->age();
```

- Convertir un pointeur constant en un pointeur non constant. Rarement utilisé.

const_cast

Exemple

```
bool personne::estplusjeuneque(personne const & p2) const {
  return const_cast<personne *>(this)->age() < const_cast<personne
      &>(p2).age();
}
```

Il n'y a pas de raison d'écrire cela : `age()` doit être déclarée `const`.

- Dans le cas de classes bien écrites, `const_cast` est rarement (jamais) 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.