Cours 02 - Concept d’encapsulation

Programmation orientée objet en C++

La dernière fois…

Généralités sur le C++.

Ce que nous avons vu :

- Compilation et outils (g++, cmake, gdb, valgrind)
- Types de base et string
- Fonctions et surcharge
- Introduction au développement scientifique

… aujourd’hui

Programme de la séance

- Concept d’objet
- Pointeurs et références
- Classes et encapsulation
- Constructeurs et destructeurs
- Quelques éléments de programmation procédurale

Introduction à la notion d’objet en C++

Introduction - Paradigmes multiples

C++ supporte des paradigmes multiples :

Paradigmes disponibles

- Programmation procédurale
- Programmation objet
- Programmation événementielle
- Programmation générique

Flexibilité et compromis

Il est possible de n'utiliser qu'un seul paradigme pour écrire un code.

Ce choix se fait souvent au détriment de la maintenabilité et de l'élégance.

Objet en C++ - Encapsulation

Le C++ supporte tous les éléments de la programmation objet :

Encapsulation des données

- Les données peuvent avoir des niveaux de visibilité différents :
  - `public` : accessible partout
  - `private` : accessible seulement dans la classe
  - `protected` : accessible dans la classe et par héritage

Important : Avec le C++, l'encapsulation est une possibilité, elle n'est jamais forcée.

Objet en C++ - Abstraction

Abstraction

Mécanisme permettant de réduire le niveau de détail d'un code.

- On regroupe les classes selon des caractéristiques communes
- Une abstraction bien conçue est généralement simple
- Elle s'utilise facilement

Exemple : Forme géométrique

```
class Forme {
    virtual double aire() = 0;
};
```

Implémentations concrètes

```
class Cercle : public Forme {
    double aire() override;
};
class Carre : public Forme {
    double aire() override;
};
```

Objet en C++ - Héritage

Héritage

Les propriétés et les fonctionnalités d'une classe existante peuvent être transmises par définition à une autre classe.

Principaux concepts

- Extensibilité
- Réutilisabilité

```
class Vehicule {
    protected:
        double vitesse;
};

class Voiture : public Vehicule {
    private:
        int nb_portes;
};
```

Objet en C++ - Polymorphisme

Polymorphisme

Capacité de manipuler à l'exécution des objets en fonction de leur type et de leur utilisation.

À la compilation

- Surcharges d'opérateurs
- Surcharges de fonctions

```
int sum(int a, int b);
double sum(double a, double b);
```

À l'exécution

- Fonctions virtuelles

```
class Base {
    virtual void draw() = 0;
};
class Derived : public Base {
    void draw() override;
};
```

Pointeurs et références

Pointeurs - Retour sur la définition

Définition

Un pointeur est une variable contenant l'adresse d'une autre variable d'un type donné.

```
int valeur = 42;
int* ptr = &valeur;  // ptr pointe vers valeur
```

Caractéristiques

- `int* ptr` : pointeur vers un `int`
- Contient l'adresse mémoire où est stocké un `int`
- Doit nécessairement être typé

Attention : Par défaut, le pointeur n'est pas initialisé !
Son contenu est celui de l'adresse mémoire avant sa création.
Un pointeur non initialisé peut pointer sur une zone mémoire du système.

Pointeurs - Comportement du compilateur

Ce que le compilateur sait

Le compilateur sait que l'objet créé est un pointeur :

- Il ne lui alloue pas de mémoire pour l'objet pointé
- Il connaît le nombre de blocs mémoire qui suivent le bloc pointé

Accéder à l'adresse d'une variable

On utilise l'opérateur `&` :

```cpp
int variable = 10;
int* pointeur = &variable;  // & donne l'adresse de variable

std::cout << "Adresse : " << pointeur << std::endl;
std::cout << "Valeur : " << *pointeur << std::endl;  // * déréférence
```

Passage d'argument par pointeur

Modifier les valeurs sans faire de copie

Passage par valeur (copie)

```cpp
void increment(int n) {
    n = n + 1;  // Modifie la copie
}

int main() {
    int x = 5;
    increment(x);
    // x vaut toujours 5
}
```

Passage par pointeur

```cpp
void increment(int* n) {
    *n = *n + 1;  // Modifie l'original
}

int main() {
    int x = 5;
    increment(&x);
    // x vaut maintenant 6
}
```

Pointeurs - À quoi ça sert ?

Usages principaux

- **Manipulation de données volumineuses**
  - Passage de paramètres pour les fonctions
  - Évite la copie de structures complexes
- **Tableaux dynamiques**
  - Créer des tableaux dont chaque élément a une taille différente
  - Ajouter des éléments en cours d'utilisation
- **Structures de données**
  - Créer des chaînes (listes, arbres, graphes)

```
struct Node {
    int data;
    Node* next;  // Pointeur vers le prochain noeud
};
```

Différence entre pointeur et référence

Pointeurs

```cpp
int x = 10;
int* ptr = &x;

ptr = nullptr;  // OK
ptr = &y;       // OK - réassignation
```

- Peuvent être null
- Peuvent être réassignés
- Peuvent être non-initialisés

Références

```cpp
int x = 10;
int& ref = x;

// ref = nullptr;  // ERREUR
// int& ref2;      // ERREUR
```

- Ne peuvent pas être null
- Ne peuvent pas être réassignées
- Doivent être initialisées à la création

Règles importantes

- Une référence ne peut pas être redéfinie après sa définition
- Une fois créée, elle ne peut pas être réaffectée
- Une référence se rapporte toujours à un objet
- Pour les classes, une référence s'initialise dans la liste d'initialisation

Pointeur this

Le pointeur spécial `this`

- Stocke l'adresse mémoire de la classe instanciée
- Permet l'accès par pointeur aux variables pour les fonctions de la classe
- N'est pas compté dans la taille de l'objet (`sizeof`)
- N'est pas accessible par les fonctions membres statiques
- N'est pas modifiable

Usages

```
class Point {
    int x, y;
public:
    Point& setX(int x) {
        this→x = x;  // Distingue membre/paramètre
        return *this; // Chaînage
    }
};
```

Avantages

- Identifier facilement les données membres
- Enchaîner les appels de fonctions
- Très utile pour la surcharge d'opérateurs

```
Point p;
p.setX(5).setY(10);  // Chaînage
```

Références - Syntaxe

C++ introduit un nouveau concept

Référence = pointeur déguisé

```cpp
int i = 2;
int& j = i;  // j est une référence sur i (alias)

std::cout << i << " " << j << std::endl;  // Affiche: 2 2

i = 5;
std::cout << i << " " << j << std::endl;  // Affiche: 5 5
```

Caractéristiques

- Obligatoirement initialisée à la création
- Ne voit que la variable pointée (alias)
- Si `i` est modifiée, `j` l'est également

Passage d'une référence

Transmission dans une fonction

Permet de modifier une variable externe à la fonction. Évite de transmettre un pointeur !

Avec pointeur

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 1, y = 2;
    swap(&x, &y);
}
```

Avec référence

```
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 1, y = 2;
    swap(x, y);  // Plus simple !
}
```

Passage de référence - Attention

Faire la distinction entre les deux écritures

```
int x = 10;

int* ptr;        // Pointeur sur un int (non initialisé)
int& ref = x;    // Référence sur un int (doit être initialisé)

int* ptr2 = &x;  // Pointeur initialisé avec l'adresse de x
```

Points clés :

- `int*` : pointeur sur un int
- `int&` : référence sur un int
- Le pointeur peut être réassigné, pas la référence
- La référence doit toujours être initialisée

Valeur à gauche (lvalue)

Références et modification

Référence modifiable

```cpp
int x = 5;
int& ref = x;  // lvalue

ref = 10;      // OK - modifie x
std::cout << x; // Affiche: 10
```

Une référence est une valeur à gauche. L'objet pointé par la référence est modifiable.

Référence constante

```cpp
int x = 5;
const int& ref = x;

ref = 10;      // ERREUR de compilation
```

On peut rendre l'objet pointé non modifiable avec `const`.

Passage par référence constante

Équivalent au passage par pointeur + sécurité. Intérêt pour les gros objets.

```cpp
void afficher(const std::string& message) {  // Pas de copie !
    std::cout << message << std::endl;
}
```

Retourner une référence

Cas d'utilisation

Dans certains cas, il est nécessaire de retourner une référence.

Référence modifiable

```
class Array {
    int data[10];
public:
    int& at(int i) {
        return data[i];
    }
};

Array arr;
arr.at(0) = 42;  // Modifie data[0]
```

Référence constante

```
class Array {
    int data[10];
public:
    const int& at(int i) const {
        return data[i];
    }
};

const Array arr;
int x = arr.at(0);  // Lecture seule
```

Retour d'une référence - Danger

Erreur à éviter absolument

Mauvais exemple

```
int& dangereuse() {
    int local = 42;
    return local;  // DANGER !
}
// local est détruite en sortie
// La référence pointe vers
// une zone mémoire invalide
```

Bon exemple

```
class Container {
    int value;
public:
    int& get() {
        return value;  // OK
    }
};
// value existe tant que
// l'objet Container existe
```

Règle d'or : Il ne faut pas retourner une référence sur une variable locale qui est détruite à la sortie de la fonction. g++ génère un avertissement.

Les nouvelles références/pointeurs

C++ moderne introduit de nouveaux outils

Gestion mémoire intelligente

- `nullptr` : remplace `NULL`
- `std::unique_ptr` : possession unique
- `std::shared_ptr` : possession partagée

```cpp
#include <memory>

auto ptr = std::make_unique<int>(42);
auto shared = std::make_shared<int>(10);
```

Nouveaux concepts

- `auto` : inférence automatique de type
- Sémantique de déplacement mémoire (`&&`)

```cpp
auto x = 42;          // int
auto& ref = x;        // int&
auto ptr = &x;        // int*

std::string&& rval = std::move(str);
```

Concept de classe

Notion d’objet

Une classe d’objet est une nouvelle structure définie par l’utilisateur

Elle encapsule des membres :

- Des données
- Des fonctions
- Des types

Le concepteur doit assurer

- L’auto-consistance
- La robustesse
- La généricité

Important : La conception nécessite une réflexion approfondie.

Déclaration

Un objet est un type de variable défini par l'utilisateur

Syntaxe de déclaration

- Avec `class` ou `struct`

Opérateur d'accès à un membre

- `.` pour les objets
- `→` pour les pointeurs

```cpp
class Point {
    int x, y;
};

int main() {
    Point p;           // Objet
    Point* ptr = &p;   // Pointeur

    p.x = 10;          // Accès direct
    ptr→x = 10;        // Accès par pointeur
}
```

Déclaration - Exemple

```cpp
class Vecteur3D {
public:
    double x, y, z;

    void afficher() {
        std::cout << "(" << x << ", " << y << ", " << z << ")" << std::endl;
    }

    double norme() {
        return std::sqrt(x*x + y*y + z*z);
    }
};

int main() {
    Vecteur3D v;
    v.x = 1.0;
    v.y = 2.0;
```

Allocation

Lors de l'instanciation d'un objet

Il y a allocation automatique de l'espace nécessaire pour stocker les membres.

```
class Exemple {
    int entier;        // 4 octets
    double tableau[3]; // 3 × 8 octets
};
// Total : environ 28 octets (avec alignement)
```

À la création

- Allocation automatique de l'espace

À la destruction

- Libération automatique de l'espace alloué

Attention : La destruction ne libère pas l'espace alloué par un pointeur !
Il faut le faire manuellement avec `delete`.

Allocation - Exemple

Allocation automatique

```
class Simple {
    int data[100];
};

void fonction() {
    Simple obj;
    // Allocation automatique

}  // Libération automatique
```

Allocation dynamique

```
class Dynamique {
    int* data;
public:
    Dynamique() {
        data = new int[100];
    }
    ~Dynamique() {
        delete[] data;  // NÉCESSAIRE !
    }
};
```

Protection des attributs

Un membre d'une classe peut être déclaré

`public`

Accessible partout, par tout le monde

```
class Example {
public:
    int x;
};
```

`private`

Accessible seulement par la classe et dans la classe

```
class Example {
private:
    int x;
};
```

`protected`

Accessible dans la classe et par des relations d'héritage

```
class Example {
protected:
    int x;
};
```

Protection des attributs - Exemple

```
class Compte {
private:
    double solde;       // Protégé - pas d'accès direct

public:
    Compte(double initial) : solde(initial) {}

    void deposer(double montant) {
        if (montant > 0) {
            solde += montant;
        }
    }

    bool retirer(double montant) {
        if (montant > 0 && montant ⩽ solde) {
            solde -= montant;
            return true;
```

Fonctions membres

Déclaration dans une classe

- On peut déclarer autant de fonctions que l'on souhaite
- Si pas de protection spéciale, elles ont accès à toutes les données de la classe
- On peut renvoyer une autoréférence

```
class Point {
    int x, y;
public:
    Point& setX(int val) {
        x = val;
        return *this;  // *this est l'objet lui-même
    }

    Point& setY(int val) {
        y = val;
        return *this;  // this est le pointeur sur l'objet
    }
};

Point p;
p.setX(10).setY(20);  // Chaînage des appels
```

Fonctions membres - Exemple

```cpp
class Calculatrice {
private:
    double resultat;

public:
    Calculatrice() : resultat(0) {}

    Calculatrice& ajouter(double val) {
        resultat += val;
        return *this;
    }

    Calculatrice& multiplier(double val) {
        resultat *= val;
        return *this;
    }
```

struct et class

Quand utiliser struct vs class ?

Utiliser `struct` si :

- Les attributs peuvent évoluer indépendamment
- Pas d'invariant à maintenir
- Type simple de données (POD)

```
struct Point {
    double x, y;
};

struct Config {
    int width;
    int height;
    bool fullscreen;
};
```

Utiliser `class` si :

- Il y a un invariant à maintenir
- Distinction interface/implémentation
- Un membre est non public

```
class Rectangle {
private:
    double largeur, hauteur;
public:
    void setLargeur(double l) {
        if (l > 0) largeur = l;
    }
};
```

La seule différence : `struct` est `public` par défaut, `class` est `private` par défaut.

Structure - Exemple

```cpp
// struct pour données simples
struct Particule {
    double x, y, z;        // Position
    double vx, vy, vz;     // Vélocité
    double masse;
};

void initialiser(Particule& p, double m) {
    p.x = p.y = p.z = 0.0;
    p.vx = p.vy = p.vz = 0.0;
    p.masse = m;
}

int main() {
    Particule p;
    initialiser(p, 1.0);
```

Structuration du code - L'en-tête (.hpp)

Séparation déclaration / définition – Rendre le code lisible et modulable : la déclaration de la classe va dans un en-tête.

point.hpp – Déclaration uniquement (interface)

```
#ifndef POINT_HPP
#define POINT_HPP

class Point {
private:
    int x, y;

public:
    Point(int x, int y);
    void afficher() const;
    double distance(const Point& p) const;
};

#endif
```

- Gardes `#ifndef` / `#define` : éviter les inclusions multiples
- Seule la **signature** des membres (pas le corps des fonctions)

Structuration du code - La définition (.cpp)

point.cpp – Implémentation des fonctions déclarées dans l'en-tête

```cpp
#include "point.hpp"
#include <iostream>
#include <cmath>

Point::Point(int x, int y) : x(x), y(y) {}

void Point::afficher() const {
    std::cout << "(" << x << ", " << y << ")";
}

double Point::distance(const Point& p) const {
    int dx = x - p.x;
    int dy = y - p.y;
    return std::sqrt(dx*dx + dy*dy);
}
```

Une implémentation dans l'en-tête correspond à un `inline` implicite.

Structuration du code - Avantages

Cette séparation permet d'améliorer l'abstraction

Avantages

- Compilation séparée
- Masquage de l'implémentation
- Temps de compilation réduit
- Facilite la maintenance
- Permet le travail en équipe

Organisation typique

```
projet/
├── include/
│   └── point.hpp
├── src/
│   └── point.cpp
└── main.cpp
```

```
g++ -c src/point.cpp -Iinclude
g++ -c main.cpp -Iinclude
g++ point.o main.o -o programme
```

Déclarations anticipées

Forward declarations

Les classes peuvent être déclarées en avance :

```
class Couleur;           // Déclaration anticipée
class NuageDePoints;     // Déclaration anticipée

class Point {
    NuageDePoints& membre;       // OK - référence
    void colore(Couleur& c);     // OK - référence
    // Couleur c;                // ERREUR - type incomplet
};
```

Utilisation

- Éviter les définitions circulaires
- Réduire les temps de compilation
- Limiter les inclusions transitives

Règles

- Type incomplet : défini plus tard
- Utilisable pour références et pointeurs
- Pas utilisable pour membres ou héritage

Déclaration anticipée - Exemple

point.hpp

```cpp
#ifndef POINT_HPP
#define POINT_HPP

class Couleur;          // Forward
class NuageDePoints;    // Forward

class Point {
    int x, y;
    NuageDePoints& nuage;

public:
    void colore(Couleur& c);
    void afficher() const;
};

#endif
```

point.cpp

```cpp
#include "point.hpp"
#include "couleur.hpp"
#include "nuagedepoints.hpp"

void Point::colore(Couleur& c) {
    // Implémentation complète
    // nécessite la définition
}

void Point::afficher() const {
    // ...
}
```

Utilisation de const

Protection en écriture

`const` permet de protéger une variable en écriture.

Portée fichier

En C++, si un objet ayant comme portée le fichier n'est pas explicitement défini comme `extern`, il sera visible uniquement dans le fichier.

```
const int MAX = 100;
// Équivalent à :
static const int MAX = 100;
```

Pointeurs constants

```
const double* p1;
// Pointeur sur double constant

double* const p2 = &x;
// Pointeur constant sur double

const double* const p3 = &x;
// Pointeur et objet constants
```

Surcharge avec const

Comportements différents selon la constance

Définition

```
class Point {
    int a;
public:
    int getA() {
        return 1;
    }

    int getA() const {
        return 2;
    }

    int getB() const {
        return getA();  // Appelle getA() const
    }

    int getC() {
        return getA();  // Appelle getA() non-const
```

Utilisation

```
Point p;
const Point cp;

p.getA();   // Retourne 1
cp.getA();  // Retourne 2

p.getB();   // Retourne 2
p.getC();   // Retourne 1
```

Règles

- Variable lvalue non constante → méthode non-const
- Variable lvalue constante → méthode const uniquement

Qualificateur de référence

Sémantique normale vs rvalue – Sémantique différente selon objet nommé ou temporaire.

`void f() &` – instance normale (lvalue)

- Appelée sur un objet existant : `p.afficher()`

`void f() &&` – instance temporaire (rvalue)

- Appelée sur un temporaire : `Point{}.afficher()`

La surcharge avec une référence est contaminante pour l'ensemble des surcharges.

```
struct Point {
    std::string nom;
    Point(std::string n = "anonyme")
        : nom{n} {}

    void afficher() & {
        std::cout << nom
            << " - Instance normale\n";
    }
    void afficher() && {
        std::cout << nom
            << " - Temporaire\n";
    }
};
```

Constructeur/Destructeur

Constructeur

Initialisation des objets – Il est fortement conseillé d'initialiser un objet à l'aide d'un constructeur.

Caractéristiques

- Généralement déclaré `public`
- Porte le nom de la classe
- Pas de type de retour
- Liste d'initialisation `: x(x_init), y(y_init)` recommandée

```
class Point {
private:
    int x, y;
public:
    Point(int x_init, int y_init)
        : x(x_init), y(y_init) {
        std::cout << "Construction ("
                  << x << ", " << y << ")"
                  << std::endl;
    }
};

int main() {
    Point p(10, 20);
}
```

Constructeur - Exemple complet

Constructeur par défaut

```cpp
Rectangle() : largeur(1.0),
              hauteur(1.0) {
    std::cout << "Rectangle 1×1"
              << std::endl;
}

// Utilisation
Rectangle r1;  // 1×1
```

Constructeur avec paramètres

```cpp
Rectangle(double l, double h)
    : largeur(l), hauteur(h) {
    std::cout << "Rectangle "
              << l << "x" << h << std::endl;
}

// Utilisation
Rectangle r2(5.0, 3.0);
```

Surcharge de constructeur

Plusieurs constructeurs possibles – même nom de classe, signatures différentes

Par défaut et 1 paramètre

```
Vecteur() : x(0), y(0), z(0) {}

Vecteur(double val)
    : x(val), y(val), z(val) {}
```

```
Vecteur v1;       // (0, 0, 0)
Vecteur v2(5.0);  // (5, 5, 5)
```

3 paramètres

```
Vecteur(double x, double y, double z)
    : x(x), y(y), z(z) {}
```

```
Vecteur v3(1, 2, 3);
```

Constructeur par copie

Copie d'objets – Par défaut, un constructeur par copie est créé ; il copie les membres bit à bit (danger si pointeurs).

Problème : copie par défaut

```
class Tableau {
    int* data;
    int taille;
public:
    Tableau(int n) : taille(n) {
        data = new int[n];
    }
    // Constructeur par copie implicite
    // copie le pointeur, pas les données !
};

Tableau t1(10);
Tableau t2 = t1;  // Copie superficielle !
```

Solution : constructeur explicite

```
class Tableau {
    int* data;
    int taille;
public:
    Tableau(int n) : taille(n) {
        data = new int[n];
    }

    // Constructeur par copie
    Tableau(const Tableau& autre)
        : taille(autre.taille) {
        data = new int[taille];
        std::copy(autre.data,
                  autre.data + taille,
                  data);
    }
};
```

Constructeur par copie - Invocation

Quand est-il appelé ? – Invocation implicite dans les cas suivants :

1. Initialisation par copie

```
Point p1(10, 20);
Point p2 = p1;   // Copie
Point p3(p1);    // Copie
```

2. Passage par valeur

```
void fonction(Point p) { /* ... */ }

fonction(p1);    // Copie de p1
```

3. Retour par valeur

```
Point creer() {
    Point p(5, 5);
    return p;    // Copie (ou élision)
}

Point q = creer();
```

Élision : le compilateur peut éviter la copie (optimisation).

Appel de constructeur

Différentes formes d'initialisation

```cpp
class Point {
    int x, y;
public:
    Point() : x(0), y(0) {}
    Point(int x, int y) : x(x), y(y) {}
};

int main() {
    Point p1;              // Défaut
    Point p2(10, 20);      // Directe
    Point p3 = Point(5,5); // Copie (souvent élidée)
    Point p4{15, 25};      // Liste (C++11)

    Point* p5 = new Point(1, 2);  // Dynamique
    delete p5;
}
```

Préférez l'initialisation par liste `{}` en C++ moderne : elle évite les conversions implicites dangereuses.

Destructeur

Nettoyage des ressources

Un destructeur libère l'espace alloué par l'utilisateur.

```
class Tableau {
private:
    int* data;
    int taille;

public:
    // Constructeur
    Tableau(int n) : taille(n) {
        data = new int[n];
        std::cout << "Allocation de " << n << " entiers" << std::endl;
    }

    // Destructeur
    ~Tableau() {
        delete[] data;
        std::cout << "Libération de " << taille << " entiers" << std::endl;
    }
```

Destructeur - Règles

Caractéristiques importantes

Propriétés

- Porte le nom `~ClassName`
- N'a jamais d'argument
- Est généralement `public`
- Un seul destructeur par classe
- Appelé automatiquement

Quand est-il appelé ?

- Fin de portée pour objet local
- `delete` pour objet dynamique
- Fin de programme pour variables globales

```
{
    Point p;
}  // Destructeur appelé

Point* ptr = new Point();
delete ptr;  // Destructeur appelé
```

Important : Si votre classe alloue de la mémoire dynamiquement, vous DEVEZ écrire un destructeur pour la libérer !

Conclusion

Concepts du cours

Gestion mémoire

- Pointeurs et références
- Allocation/libération
- `new` et `delete`

Encapsulation

- `public`, `private`, `protected`
- Abstraction des données
- Interface vs implémentation

Classes et objets

- Constructeurs (défaut, paramètres, copie)
- Destructeur
- Fonctions membres
- Pointeur `this`

Bonnes pratiques

- Séparation .hpp/.cpp
- Déclarations anticipées
- Utilisation de `const`

En TP

Essayez de créer :

```
class Particule {
    // Position et vitesse
    // Constructeurs
    // Méthodes de mise à jour
    // Destructeur si nécessaire
};
```

La prochaine fois

Cours 3 : Opérateurs et Héritage

Opérateurs

- Surcharge d'opérateurs
- Opérateurs arithmétiques
- Opérateurs de comparaison
- Opérateurs d'affectation

Héritage

- Classe de base et dérivée
- Héritage public/privé/protégé
- Polymorphisme
- Fonctions virtuelles

Questions ?