C++ et la Programmation Orienté Objet

Pr: MAJDOUB Soufyane

Université Sidi Mohamed Ben Abdellah de Fès
soufyane.majdoub@usmba.ac.ma

Laboratoire d'Informatique, Signaux, Automatique et Cognitivisme
29 mai 2025

Plan

Notion d'objet

La notion d'objet

La POO est une méthode de programmation puissante qui permet de créer des applications complexes et modulaires en utilisant des objets qui interagissent les uns avec les autres.

Notion d'objet

La notion d'objet

La POO est une méthode de programmation puissante qui permet de créer des applications complexes et modulaires en utilisant des objets qui interagissent les uns avec les autres.

- Les objets sont utilisés pour modéliser des entités du monde réel ou des concepts abstraits.
- C'est une instance d'une classe, qui est une structure de données qui définit les attributs et les méthodes de l'objet.

Notion d'objet

La notion d'objet

La POO est une méthode de programmation puissante qui permet de créer des applications complexes et modulaires en utilisant des objets qui interagissent les uns avec les autres.

- Les objets sont utilisés pour modéliser des entités du monde réel ou des concepts abstraits.
- C'est une instance d'une classe, qui est une structure de données qui définit les attributs et les méthodes de l'objet.

Exemple : Compte bancaire

Attributs

- Numéro_de_compte
- Solde
- Propriétaire

Méthodes

- retirer(montant)
- déposer(montant)
- consulter()

Le concept d'objet

La POO repose sur quatre concepts fondamentaux :

1. **L'encapsulation :** est le fait de regrouper des données et des méthodes connexes dans un même objet.
2. **L'abstraction :** permet de masquer les détails de l'implémentation d'un objet et de se concentrer sur son comportement.
3. **L'héritage :** permet de créer de nouvelles classes à partir d'une classe existante, en héritant de ses attributs et de ses méthodes.
4. **Le polymorphisme :** permet à des objets de même type d'avoir des comportements différents.

Le concept d'objet

La POO repose sur quatre concepts fondamentaux :

1. **L'encapsulation :** est le fait de regrouper des données et des méthodes connexes dans un même objet.
2. **L'abstraction :** permet de masquer les détails de l'implémentation d'un objet et de se concentrer sur son comportement.
3. **L'héritage :** permet de créer de nouvelles classes à partir d'une classe existante, en héritant de ses attributs et de ses méthodes.
4. **Le polymorphisme :** permet à des objets de même type d'avoir des comportements différents.

Résultat :

- Utilisation d'objets pour définir d'autres objets
- L'objet enfant hérite des propriétés de l'objet parent
- Fonctionnalités internes et variables ne sont pas accessibles directement
- Constructions des objets indépendantes

Contexte

- En programmation procédurale, le code est organisé en fonctions et structures de données séparées.
- La programmation orientée objet (POO) introduit la notion de **classe**, qui permet de regrouper **données** et **fonctions** au sein d'une même entité.
- Une classe sert de plan pour créer des **objets** (instances) représentant des entités du monde réel ou conceptuel.
- L'approche objet facilite la modélisation du réel, la réutilisation de code et la gestion de la complexité dans les grands programmes.

Principes de base de la POO

- **Encapsulation** : Regrouper les données et méthodes dans une classe et contrôler l'accès aux données internes via des mots-clés (`private`, `public`, etc.) pour protéger l'état de l'objet.
- **Héritage** : Définir de nouvelles classes à partir de classes existantes (relation " est-un "). La classe dérivée hérite des attributs et méthodes de la classe de base, ce qui favorise la réutilisation de code.
- **Polymorphisme** : Traiter des objets de classes différentes de manière uniforme via un type commun (généralement la classe de base). En C++, les fonctions `virtual` permettent à une même méthode d'avoir un comportement différent selon l'objet manipulé.

Qu’est-ce qu’une classe ?

- Une **classe** est une construction qui permet de définir un nouveau type regroupant des **variables** (attributs) et des **fonctions** (méthodes).
- La classe représente un concept ou une entité et sert de plan pour créer des objets. Chaque **objet** (ou instance) de la classe possède ses propres valeurs d’attributs.
- Les méthodes définies dans la classe décrivent les comportements que les objets de cette classe peuvent réaliser.
- Une classe C++ peut être utilisée comme n’importe quel type (déclaration de variables, passage en paramètre de fonctions, retour de fonctions, etc.).

Terminologie des classes

Classe Plan ou prototype définissant les attributs et méthodes communs à un ensemble d'objets.

Objet (instance) Occurrence concrète d'une classe en mémoire, possédant sa propre copie des attributs définis par la classe.

Attribut (membre donné) Variable définie dans la classe, représentant une donnée propre à chaque objet.

Méthode (fonction membre) Fonction définie dans la classe, qui décrit une opération pouvant être effectuée par les objets de cette classe.

Constructeur Méthode spéciale exécutée lors de la création d'un objet, utilisée pour initialiser l'objet (porte le même nom que la classe).

Destructeur Méthode spéciale exécutée lors de la destruction d'un objet, utilisée pour libérer les ressources (nommée `~Classe`).

Implémentation des méthodes hors de la classe

- En C++, on peut déclarer les méthodes dans la définition de la classe, puis définir leur corps en dehors de la classe (souvent dans un fichier source séparé).
- La définition d'une méthode hors de la classe utilise l'opérateur de portée `::`. Par exemple, si la classe `Point` déclare une méthode `afficher()`, on peut la définir ainsi en dehors de la classe : `void Point::afficher(){ ... }`.
- Séparer l'interface (fichier `.h`) de l'implémentation (fichier `.cpp`) améliore la lisibilité et la maintenabilité du code.

Exemple de classe

```
// (Supposons que la bibliotheque iostream est incluse)
class Point {
private:
int x;
int y;
public:
// Constructeur
Point(int abs, int ord) {
x = abs;
y = ord;
}
// Methode pour afficher les coordonnees
void afficher() {
std::cout << "Point(" << x << ", " << y << ")" << std::endl;
}
// Methode pour deplacer le point
void deplacer(int dx, int dy) {
x += dx;
y += dy;
}};
```

Utilisation d'un objet

```
#include <iostream>
using namespace std;

int main() {
// Instanciation d'un objet Point
Point p(1, 2);
p.afficher();              // affiche: Point(1, 2)
// Deplacement du point
p.deplacer(3, 4);
p.afficher();              // affiche: Point(4, 6)
return 0;
}
```

Classes vs struct en C++

- En C++, `class` et `struct` sont quasiment identiques à une différence près : par défaut, les membres d'une `class` sont `private`, tandis que les membres d'une `struct` sont `public`.
- Par convention, on utilise généralement `class` pour la POO (types avec invariants, logique interne) et `struct` pour de simples structures de données sans invariants particuliers.
- De même, pour l'héritage, le mode par défaut est `private` pour une `class` dérivée et `public` pour une `struct` dérivée.

Constructeurs

- Un **constructeur** est une méthode spéciale appelée automatiquement lors de la création d'un objet (instanciation).
- Le constructeur porte le même nom que la classe et n'a pas de type de retour (même pas `void`).
- Il sert à initialiser l'objet (donner des valeurs initiales aux attributs, allouer des ressources si nécessaire).
- On peut définir plusieurs constructeurs pour une classe (plusieurs signatures) afin de créer l'objet dans différents états initiaux.
- Si aucun constructeur n'est défini, le compilateur en génère un par défaut sans paramètre (qui effectue une initialisation par défaut des attributs).

Destructeurs

- Le **destructeur** est une méthode spéciale appelée automatiquement lorsque l'objet est détruit (fin de vie de la variable ou appel de `delete`).
- Le destructeur a le même nom que la classe précédé d'un tilde (`~`). Il n'a ni paramètre ni valeur de retour.
- Il sert à libérer les ressources détenues par l'objet (mémoire allouée avec `new`, descripteurs de fichier, etc.) pour éviter les fuites de ressources.
- Si aucun destructeur n'est fourni, le compilateur en génère un par défaut (qui appelle les destructeurs des attributs objets et de la classe de base).
- La destruction d'un objet est automatique pour les objets locaux (à la fin du bloc), et manuelle via `delete` pour les objets alloués dynamiquement.

Exemple : Constructeur et Destructeur (Classe)

```
// Classe avec constructeurs et destructeur
class Truc {
public:
Truc() { // constructeur par defaut
std::cout << "Constructeur par defaut" << std::endl;
}
Truc(int x) { // constructeur avec parametre
std::cout << "Constructeur avec param: " << x << std::endl;
}
~Truc() { // destructeur
std::cout << "Destructeur" << std::endl;
}
};
```

Exemple : Utilisation de Constructeur/Destructeur

```
#include <iostream>
using namespace std;

int main() {
cout << "Debut main" << endl;
Truc a;              // Appel du constructeur par defaut
Truc b(5);           // Appel du constructeur avec parametre
Truc *p = new Truc();  // Appel du constructeur par defaut (
    objet dynamique)
delete p;            // Appel du destructeur pour l'objet pointe
    par p
cout << "Fin main" << endl;
return 0;
}
// --- Sortie console: ---
// Debut main
// Constructeur par defaut
// Constructeur avec param: 5
// Constructeur par defaut
// Destructeur
// Fin main
```

Durée de vie des objets

- Un objet alloué **statiquement** (par exemple, une variable locale sur la pile) est détruit automatiquement à la fin de sa portée. Son destructeur est appelé à la fin du bloc où l'objet a été créé.
- Un objet alloué **dynamiquement** avec `new` (sur le tas) reste en mémoire tant qu'on ne l'a pas libéré avec `delete`. Le destructeur n'est appelé qu'au moment du `delete`.
- Il est important d'appeler `delete` sur les objets alloués dynamiquement une fois qu'on n'en a plus besoin, afin d'éviter les fuites de mémoire.
- Exemple : dans le code précédent, `a` et `b` sont des objets automatiques détruits en fin de `main`, tandis que l'objet créé par `new Truc()` est détruit explicitement lors du `delete p`.

Encapsulation

- **Encapsulation** : principe consistant à cacher les détails internes d'une classe en contrôlant l'accès à ses données. Les données (attributs) et fonctions (méthodes) sont regroupées dans la classe.
- Une classe définit une **interface publique** (méthodes `public`) et maintient ses données sensibles en `private`. Le code utilisateur ne peut accéder aux attributs qu'au travers des méthodes prévues.
- On utilise des méthodes **accesseurs** (getters) et **mutateurs** (setters) pour consulter ou modifier les attributs de manière contrôlée.
- L'encapsulation garantit l'intégrité de l'objet en empêchant les modifications non contrôlées de son état interne (exemple : éviter qu'un attribut `age` ne prenne une valeur négative en validant les données dans un setter).

Spécificateurs d'accès

- `public` : le membre est accessible partout (interface publique de la classe).
- `private` : le membre est accessible uniquement au sein de la classe (les codes externes ne peuvent pas y accéder directement).
- `protected` : le membre est accessible dans la classe elle-même **et** dans ses classes dérivées, mais pas depuis l'extérieur de la hiérarchie.
- Par défaut, les membres d'une classe C++ sont `private` si aucun spécificateur n'est indiqué (dans une `struct`, le défaut est `public`).

Sans encapsulation : un problème

```
#include <iostream>
using namespace std;

struct Person {
int age;
};

int main() {
Person p;
p.age = -5;  // Valeur d'age incoherente autorisee
cout << "Age de p: " << p.age << endl;  // affiche "Age de p:
    -5"
return 0;
}
```

Avec encapsulation : la solution

```
#include <iostream>
using namespace std;
class Person {
private:
int age;
public:
Person() { age = 0; }
void setAge(int a) {
if(a >= 0) age = a;
else cout << "Valeur d'age invalide !" << endl;}
int getAge() {
return age;
}};
int main() {
Person p;
p.setAge(-5);  // tentative d'assigner une valeur invalide (
    message d'erreur)
p.setAge(30);  // age modifie a 30
cout << "Age de p: " << p.getAge() << endl;  // affiche 30
return 0;}
```

Héritage : concepts généraux

- L'**héritage** permet de définir une nouvelle classe en se basant sur une classe existante. La classe dérivée **hérite** des attributs et méthodes de la classe de base.
- On modélise ainsi une relation de spécialisation « est-un ». Exemple : une classe `Etudiant` peut hériter d'une classe `Personne` car un étudiant *est une* personne, mais avec des caractéristiques supplémentaires.
- L'héritage favorise la réutilisation de code : les fonctionnalités communes sont implémentées dans la classe de base, et les classes dérivées ajoutent ou redéfinissent des fonctionnalités spécifiques.
- En C++, la syntaxe pour hériter est `class Derived : public Base { ... }`. (Par défaut, une `class` hérite en mode `private`, mais on utilise `public` pour modéliser une relation d'héritage « est-un ».)
- La classe dérivée a accès aux membres `public` et `protected` de la base, mais ne peut pas accéder aux membres déclarés `private` dans la classe de base.

Exemple d'héritage (classe de base et dérivée)

```
// Classe de base
class Personne {
protected:
std::string nom;
public:
Personne(std::string n) : nom(n) {}
void sePresenter() {
std::cout << "Bonjour, je m'appelle " << nom << std::endl;
}};
// Classe derivee de Personne
class Etudiant : public Personne {
private:
std::string ecole;
public:
Etudiant(std::string n, std::string e)
: Personne(n), ecole(e) {}  // Appel du constructeur de
    Personne
void afficherEcole() {
std::cout << "Je suis etudiant a " << ecole << std::endl;
}};
```

Utilisation d'une classe dérivée

```
#include <iostream>
#include <string>
using namespace std;

int main() {
Etudiant etu("Ali", "Universite X");
etu.sePresenter();     // affiche: "Bonjour, je m'appelle Ali"
etu.afficherEcole();   // affiche: "Je suis etudiant a
    Universite X"
return 0;
}
```

Détails sur l'héritage

- Lors de la création d'un objet d'une classe dérivée, le constructeur de la classe de base est appelé en premier, puis le constructeur de la classe dérivée. (De même, à la destruction, le destructeur de la dérivée s'exécute avant celui de la base.)
- Une classe dérivée n'a pas accès aux membres `private` de la base. Elle peut toutefois accéder aux membres `protected` de la base (ainsi bien sûr qu'aux membres `public`).
- Par défaut, si on utilise le mot-clé `class` pour hériter sans préciser le mode, l'héritage est considéré comme `private`. Il est donc recommandé d'indiquer `public` pour les relations d'héritage « classiques ».
- *Remarque :* C++ autorise l'**héritage multiple** (une classe peut hériter de plusieurs bases), mais cela complexifie la conception (ex : problème du « diamant »). Ce cours se limite à l'héritage simple.

Polymorphisme et fonctions virtuelles

- Le **polymorphisme** permet de manipuler différents objets dérivés via un pointeur ou une référence sur leur classe de base commune, en faisant appel aux méthodes spécifiques de chaque objet réel.
- Pour obtenir ce comportement, il faut déclarer des méthodes **virtuelles** dans la classe de base. Une méthode déclarée `virtual` peut être **surchargée** (redéfinie) dans les classes dérivées.
- Lorsqu'on appelle une méthode virtuelle sur un pointeur (ou référence) de la classe de base pointant un objet dérivé, c'est la version redéfinie dans la classe dérivée qui s'exécute (liaison dynamique à l'exécution).
- Si une méthode n'est pas virtuelle, un appel sur un pointeur de base utilisera toujours l'implémentation de la classe de base (liaison statique à la compilation).
- En C++, on peut utiliser le mot-clé `override` dans la définition de la méthode dérivée pour indiquer explicitement qu'elle redéfinit une méthode virtuelle de la base (bonne pratique, évite les erreurs de signature).
- **Destructeur virtuel :** Il est conseillé de toujours déclarer un destructeur `virtual` dans une classe de base destinée à être dérivée, afin que la destruction via un pointeur de base appelle correctement le destructeur de la classe dérivée.

Classes abstraites

- Une **classe abstraite** est une classe pour laquelle on ne peut pas créer d'objets (elle n'est pas instanciable directement).
- En C++, une classe est abstraite si elle déclare au moins une méthode **virtuelle pure**, c'est-à-dire une méthode virtuelle sans implémentation, notée `=0` dans la déclaration.
- Une méthode virtuelle pure doit obligatoirement être redéfinie dans les classes dérivées concrètes. Une classe dérivée qui n'implémente pas toutes les méthodes virtuelles pures héritées reste abstraite à son tour.
- Les classes abstraites servent souvent de classes de base pour définir des interfaces. Elles décrivent un comportement générique que les sous-classes doivent spécifier. (Exemple : classe abstraite `Forme` avec méthode virtuelle pure `aire()`, implémentée différemment par `Cercle`, `Rectangle`, etc.)
- Exemple en C++ : on peut définir une classe abstraite `Animal` avec une méthode virtuelle pure `faireSon()`. Les classes `Chien` et `Chat` hériteront de `Animal` et fourniront leur propre implémentation de `faireSon()`.

Exemple de polymorphisme (classes)

```
// Classe de base abstraite
class Animal {
public:
virtual void faireSon() = 0;    // methode virtuelle pure (
    abstraite)
virtual ~Animal() {}            // destructeur virtuel (
    important pour polymorphisme)
};

class Chien : public Animal {
public:
void faireSon() override {
std::cout << "Wouf" << std::endl;
}};
class Chat : public Animal {
public:
void faireSon() override {
std::cout << "Miaou" << std::endl;
}};
```

Exemple d'utilisation polymorphique

```
#include <iostream>
using namespace std;

int main() {
Animal* a1 = new Chien();
Animal* a2 = new Chat();
a1->faireSon();  // Wouf  (appel de la methode de Chien)
a2->faireSon();  // Miaou (appel de la methode de Chat)
delete a1;  // Appelle ~Chien puis ~Animal (grace au
    destructeur virtuel)
delete a2;  // Appelle ~Chat puis ~Animal
return 0;
}
```

Conclusion

- En résumé, nous avons introduit les principaux concepts de la POO en C++ : les classes, l'encapsulation, l'héritage et le polymorphisme.
- Les **classes** permettent de définir de nouveaux types regroupant des données et des fonctions, et de créer des **objets** (instances) à partir de ces classes.
- Les **constructeurs** et **destructeurs** assurent respectivement l'initialisation et la destruction correcte des objets.
- L'**encapsulation** protège les données internes d'une classe en contrôlant les accès, et offre une interface publique claire. Elle garantit l'intégrité des objets et facilite la maintenance du code.
- L'**héritage** permet de factoriser le code en définissant des relations de spécialisation entre classes (une classe dérivée réutilise et complète les fonctionnalités d'une classe de base).
- Le **polymorphisme** permet de manipuler des objets de classes dérivées via un pointeur ou une référence de la classe de base, et d'appeler automatiquement la bonne implémentation de méthode en fonction du type réel de l'objet. Cela rend le code plus flexible et extensible.

Fin

Questions ? Commentaires ?