Chapitre 8 : Programmation orientée objet en C++

Objectifs pédagogiques :

- Comprendre les concepts fondamentaux de la programmation orientée objet
- Savoir concevoir et implémenter des classes en C++
- Maîtriser les mécanismes d'encapsulation, d'héritage et de polymorphisme
- Utiliser les constructeurs et destructeurs pour une gestion correcte des ressources
- Modéliser des systèmes d'automatique (capteurs, actionneurs, régulateurs) sous forme d'objets

1. Introduction à la programmation orientée objet

1.1. Pourquoi l'orienté objet en automatique ?

Jusqu'à présent, nous avons utilisé la programmation procédurale : des données (variables) et des fonctions qui manipulent ces données.

En automatique, nous manipulons des entités du monde réel :

- Un capteur : a des caractéristiques (type, plage de mesure, offset, gain) et des comportements (lire une valeur, s'auto-calibrer)
- Un actionneur : a un état (marche/arrêt, vitesse, position) et des actions (démarrer, arrêter, régler)
- Un régulateur PID : a des paramètres (kp, ki, kd) et des états internes (intégrale, erreur précédente)

La programmation orientée objet (POO) permet de regrouper données et fonctions qui agissent sur ces données dans une même entité : la classe.

1.2. Programmation procédurale vs orientée objet

Approche procédurale :

```
// Données séparées
double kp, ki, kd;
double integrale = 0.0;
double erreur_prec = 0.0;

// Fonctions qui manipulent ces données
double calculer_PID(double erreur, double dt) {
    integrale += erreur * dt;
    double derivee = (erreur - erreur_prec) / dt;
    erreur_prec = erreur;
    return kp * erreur + ki * integrale + kd * derivee;
}

void reinitialiser_PID() {
    integrale = 0.0;
```

```
    erreur_prec = 0.0;
}
```

Approche orientée objet :

```
class RegulateurPID {
private:
    double kp, ki, kd;
    double integrale;
    double erreur_precedente;

public:
    // Constructeur
    RegulateurPID(double p, double i, double d)
        : kp(p), ki(i), kd(d), integrale(0.0), erreur_precedente(0.0) {}

    // Méthode (fonction membre)
    double calculer(double erreur, double dt) {
        integrale += erreur * dt;
        double derivee = (erreur - erreur_precedente) / dt;
        erreur_precedente = erreur;
        return kp * erreur + ki * integrale + kd * derivee;
    }

    void reinitialiser() {
        integrale = 0.0;
        erreur_precedente = 0.0;
    }
};

// Utilisation
RegulateurPID pid(2.0, 1.0, 0.1);
double commande = pid.calculer(erreur, dt);
```

Avantages de l'approche objet :

- Regroupement cohérent : tout ce qui concerne le PID est dans la classe
- Encapsulation : les détails internes sont cachés
- Réutilisation : on peut créer plusieurs régulateurs indépendants
- Extensibilité : facile d'ajouter des fonctionnalités

2. Concepts fondamentaux de la POO

2.1. Classe et objet

- **Classe** : un modèle, un moule qui décrit la structure et le comportement
- **Objet** : une instance concrète de la classe (un exemplaire)

```
// Définition de la classe (le modèle)
class Moteur {
    // ... attributs et méthodes ...
};

// Création d'objets (instances)
Moteur moteur1;     // Premier objet
Moteur moteur2;     // Deuxième objet, indépendant
```

2.2. Encapsulation

L'encapsulation consiste à :

- Regrouper données et méthodes
- Cacher les détails d'implémentation
- Contrôler l'accès aux données

Niveaux d'accès :

Mot-clé	Accès depuis la classe	Accès depuis les classes dérivées	Accès depuis l'extérieur
`private`	Oui	Non	Non
`protected`	Oui	Oui	Non
`public`	Oui	Oui	Oui

Principe : Les attributs sont généralement `private`, l'interface est `public`.

3. Création d'une classe simple

3.1. Syntaxe de base

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

class CapteurTemperature {
// Par défaut, tout est private
private:
```

```
    // Attributs (caractéristiques)
    double temperature;
    string unite;
    double offset;
    double gain;

public:
    // Méthodes (comportements)
    void initialiser(double off = 0.0, double g = 1.0, string u = "°C") {
        offset = off;
        gain = g;
        unite = u;
        temperature = 0.0;
    }

    void acquerir(double valeur_brute) {
        temperature = (valeur_brute - offset) * gain;
    }

    double lire() const {  // const = ne modifie pas l'objet
        return temperature;
    }

    string getUnite() const {
        return unite;
    }

    void afficher() const {
        cout << "Température : " << temperature << " " << unite << endl;
    }
};

int main() {
    // Création d'objets
    CapteurTemperature capteur1, capteur2;

    // Initialisation
    capteur1.initialiser(0.5, 1.0, "°C");
    capteur2.initialiser(0.0, 1.8, "°F");  // Pour un capteur en degrés Fahrenheit

    // Utilisation
    capteur1.acquerir(23.5);  // valeur brute
    capteur2.acquerir(75.2);
```

```
    capteur1.afficher();
    capteur2.afficher();

    // Accès contrôlé
    double t = capteur1.lire();
    cout << "Température lue : " << t << " " << capteur1.getUnite() << endl;

    return 0;
}
```

3.2. Séparation interface/implémentation

En pratique, on sépare souvent la déclaration (fichier .h) et l'implémentation (fichier .cpp).

capteur.h

```
#ifndef CAPTEUR_H
#define CAPTEUR_H

#include <string>

class CapteurTemperature {
private:
    double temperature;
    std::string unite;
    double offset;
    double gain;

public:
    void initialiser(double off = 0.0, double g = 1.0, std::string u = "°C");
    void acquerir(double valeur_brute);
    double lire() const;
    std::string getUnite() const;
    void afficher() const;
};

#endif
```

capteur.cpp

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

```cpp
using namespace std;

void CapteurTemperature::initialiser(double off, double g, string u) {
    offset = off;
    gain = g;
    unite = u;
    temperature = 0.0;
}

void CapteurTemperature::acquerir(double valeur_brute) {
    temperature = (valeur_brute - offset) * gain;
}

double CapteurTemperature::lire() const {
    return temperature;
}

string CapteurTemperature::getUnite() const {
    return unite;
}

void CapteurTemperature::afficher() const {
    cout << "Température : " << temperature << " " << unite << endl;
}
```

4. Constructeurs et destructeurs

Les constructeurs sont des méthodes spéciales appelées automatiquement à la création d'un objet.

4.1. Constructeur par défaut

```cpp
class Moteur {
private:
    bool en_marche;
    double vitesse;
    string nom;

public:
    // Constructeur par défaut (sans paramètres)
    Moteur() {
        en_marche = false;
        vitesse = 0.0;
        nom = "Moteur sans nom";
        cout << "Constructeur par défaut appelé" << endl;
```

```
    }

    void demarrer() { en_marche = true; }
    void arreter() { en_marche = false; }
    void setVitesse(double v) { if (en_marche) vitesse = v; }

    void afficher() const {
        cout << nom << " : " << (en_marche ? "MARCHE" : "ARRET")
             << ", vitesse=" << vitesse << endl;
    }
};

int main() {
    Moteur m;  // Constructeur par défaut appelé automatiquement
    m.afficher();
    m.demarrer();
    m.setVitesse(1500);
    m.afficher();

    return 0;
}
```

4.2. Constructeur avec paramètres

```
class Moteur {
private:
    bool en_marche;
    double vitesse;
    string nom;

public:
    // Constructeur avec paramètres
    Moteur(const string& n, double v_init = 0.0) {
        en_marche = (v_init > 0);
        vitesse = v_init;
        nom = n;
        cout << "Constructeur de " << nom << endl;
    }

    // Destructeur (vu plus tard)
    ~Moteur() {
        cout << "Destructeur de " << nom << endl;
    }
```

```
    void demarrer() { en_marche = true; }
    void arreter() { en_marche = false; vitesse = 0.0; }

    void afficher() const {
        cout << nom << " : " << (en_marche ? "MARCHE" : "ARRET")
             << ", vitesse=" << vitesse << endl;
    }
};

int main() {
    Moteur m1("Moteur principal", 1000);
    Moteur m2("Moteur secondaire");  // Utilise la valeur par défaut

    m1.afficher();
    m2.afficher();

    m2.demarrer();
    m2.setVitesse(500);
    m2.afficher();

    return 0;
}
```

4.3. Liste d'initialisation

C'est la manière préférée d'initialiser les attributs (plus efficace).

```
class RegulateurPID {
private:
    const double kp, ki, kd;  // Constantes (doivent être initialisées)
    double integrale;
    double erreur_prec;
    double dt;

public:
    // Liste d'initialisation
    RegulateurPID(double p, double i, double d, double pas = 0.01)
        : kp(p), ki(i), kd(d), dt(pas), integrale(0.0), erreur_prec(0.0) {
        // Le corps du constructeur peut être vide
        cout << "PID créé avec kp=" << kp << ", ki=" << ki << ", kd=" << kd << endl;
    }

    double calculer(double erreur) {
```

```
        integrale += erreur * dt;
        double derivee = (erreur - erreur_prec) / dt;
        erreur_prec = erreur;
        return kp * erreur + ki * integrale + kd * derivee;
    }

    void reinitialiser() {
        integrale = 0.0;
        erreur_prec = 0.0;
    }
};

int main() {
    RegulateurPID pid(2.0, 1.0, 0.1, 0.01);

    for (int i = 0; i < 10; i++) {
        double erreur = 10.0 - i * 0.5;
        double cmd = pid.calculer(erreur);
        cout << "erreur=" << erreur << ", commande=" << cmd << endl;
    }

    return 0;
}
```

4.4. Constructeur de copie

Appelé lors de la création d'un objet à partir d'un autre.

```
class Moteur {
private:
    string nom;
    double* historique_vitesse;  // Allocation dynamique
    int taille_histo;

public:
    // Constructeur principal
    Moteur(const string& n, int taille = 10)
        : nom(n), taille_histo(taille) {
        historique_vitesse = new double[taille_histo];
        for (int i = 0; i < taille_histo; i++) {
            historique_vitesse[i] = 0.0;
        }
        cout << "Constructeur de " << nom << endl;
    }
```

```
    // Constructeur de copie
    Moteur(const Moteur& autre)
        : nom(autre.nom + "_copie"), taille_histo(autre.taille_histo) {
        // Allocation nouvelle mémoire
        historique_vitesse = new double[taille_histo];

        // Copie des données
        for (int i = 0; i < taille_histo; i++) {
            historique_vitesse[i] = autre.historique_vitesse[i];
        }
        cout << "Constructeur de copie de " << autre.nom << " -> " << nom << endl;
    }

    // Destructeur
    ~Moteur() {
        delete[] historique_vitesse;
        cout << "Destructeur de " << nom << endl;
    }

    void enregistrer_vitesse(double v, int index) {
        if (index >= 0 && index < taille_histo) {
            historique_vitesse[index] = v;
        }
    }

    void afficher() const {
        cout << nom << " - historique: ";
        for (int i = 0; i < taille_histo; i++) {
            cout << historique_vitesse[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Moteur m1("Moteur A");
    m1.enregistrer_vitesse(1000, 0);
    m1.enregistrer_vitesse(1500, 1);
    m1.afficher();

    Moteur m2 = m1;  // Constructeur de copie
    m2.afficher();   // Historique copié
```

```
    return 0;
}
```

4.5. Destructeur

Appelé automatiquement à la destruction de l'objet. Crucial pour libérer les ressources.

```
class AcquisitionBuffer {
private:
    double* buffer;
    int taille;
    int index;

public:
    AcquisitionBuffer(int t) : taille(t), index(0) {
        buffer = new double[taille];
        cout << "Buffer de " << taille << " éléments alloué" << endl;
    }

    ~AcquisitionBuffer() {
        delete[] buffer;
        cout << "Buffer libéré" << endl;
    }

    void ajouter(double valeur) {
        if (index < taille) {
            buffer[index++] = valeur;
        }
    }

    double moyenne() const {
        double somme = 0.0;
        for (int i = 0; i < index; i++) {
            somme += buffer[i];
        }
        return (index > 0) ? somme / index : 0.0;
    }
};

int main() {
    {
        AcquisitionBuffer buf(100);
        for (int i = 0; i < 50; i++) {
```

```
            buf.ajouter(i * 1.5);
        }
        cout << "Moyenne = " << buf.moyenne() << endl;
    }  // Le destructeur est appelé automatiquement ici

    cout << "Fin du programme" << endl;
    return 0;
}
```

5. Héritage

L'héritage permet de créer une nouvelle classe à partir d'une classe existante, en réutilisant et en spécialisant.

5.1. Concept

```
Capteur (classe de base)
          ↑
          | héritage
          |
    CapteurTemperature (classe dérivée)
```

5.2. Exemple

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

// Classe de base
class Capteur {
protected:  // accessible aux classes dérivées
    string nom;
    string unite;
    double valeur;
    bool est_initialise;

public:
    Capteur(const string& n = "Capteur", const string& u = "unités")
        : nom(n), unite(u), valeur(0.0), est_initialise(true) {
        cout << "Constructeur de Capteur: " << nom << endl;
    }

    virtual ~Capteur() {  // virtual pour le polymorphisme
        cout << "Destructeur de Capteur: " << nom << endl;
```

```
    }

    // Méthodes
    virtual void acquerir() = 0;  // Méthode virtuelle pure (classe abstraite)

    double lire() const { return valeur; }
    string getNom() const { return nom; }
    string getUnite() const { return unite; }

    virtual void afficher() const {
        cout << nom << " = " << valeur << " " << unite;
    }
};

// Classe dérivée
class CapteurTemperature : public Capteur {
private:
    double offset;
    double gain;

public:
    CapteurTemperature(const string& n = "Thermocouple", double off = 0.0, double g = 1.0)
        : Capteur(n, "°C"), offset(off), gain(g) {
        cout << "Constructeur de CapteurTemperature" << endl;
    }

    ~CapteurTemperature() {
        cout << "Destructeur de CapteurTemperature" << endl;
    }

    void acquerir() override {  // override est optionnel mais recommandé
        // Simulation : valeur brute aléatoire
        double valeur_brute = 20.0 + (rand() % 100) / 10.0;
        valeur = (valeur_brute - offset) * gain;
    }

    void etalonner(double val_reelle, double val_brute) {
        gain = val_reelle / (val_brute - offset);
    }

    void afficher() const override {
        Capteur::afficher();  // Appel de la méthode de base
        cout << " (offset=" << offset << ", gain=" << gain << ")";
```

```
    }
};

// Autre classe dérivée
class CapteurPression : public Capteur {
private:
    double pression_max;

public:
    CapteurPression(const string& n = "Manomètre", double pmax = 10.0)
        : Capteur(n, "bar"), pression_max(pmax) {}

    void acquerir() override {
        // Simulation
        valeur = (rand() % 100) / 10.0;
    }

    double getPressionMax() const { return pression_max; }

    void afficher() const override {
        Capteur::afficher();
        cout << " (max=" << pression_max << " bar)";
    }
};

int main() {
    // Capteur c;  // ERREUR : classe abstraite (méthode pure)

    CapteurTemperature t1("Thermo four");
    CapteurPression p1("Pression cuve", 16.0);

    // Polymorphisme : on peut manipuler des objets dérivés via des pointeurs de base
    Capteur* capteurs[2];
    capteurs[0] = &t1;
    capteurs[1] = &p1;

    for (int i = 0; i < 2; i++) {
        capteurs[i]->acquerir();
        capteurs[i]->afficher();
        cout << endl;
    }
    return 0;
}
```

5.3. Types d'héritage

```
class A {
private:
    int prive;
protected:
    int protege;
public:
    int publique;
};

// Héritage public (le plus courant)
class B_publique : public A {
    // prive : inaccessible
    // protege : reste protege
    // publique : reste publique
};

// Héritage protege
class B_protege : protected A {
    // prive : inaccessible
    // protege : reste protege
    // publique : devient protege
};

// Héritage prive (par défaut si non spécifié)
class B_prive : private A {
    // prive : inaccessible
    // protege : devient prive
    // publique : devient prive
};
```

5.4. Héritage multiple

```
class Imprimante {
public:
    void imprimer() { cout << "Impression..." << endl; }
};

class Scanner {
public:
    void scanner() { cout << "Scan..." << endl; }
};
```

```
class Photocopieuse : public Imprimante, public Scanner {
public:
    void photocopier() {
        scanner();
        imprimer();
    }
};

int main() {
    Photocopieuse p;
    p.photocopier();  // Utilise les deux héritages
    return 0;
}
```

Attention : L'héritage multiple peut créer des ambiguïtés (diamant).

6. Polymorphisme

Le polymorphisme permet de traiter des objets de classes dérivées comme des objets de la classe de base, tout en conservant leur comportement spécifique.

6.1. Fonctions virtuelles

```
class Forme {
public:
    virtual double aire() const { return 0.0; }
    virtual void afficher() const { cout << "Forme" << endl; }
};

class Cercle : public Forme {
private:
    double rayon;
public:
    Cercle(double r) : rayon(r) {}

    double aire() const override { return 3.14159 * rayon * rayon; }
    void afficher() const override {
        cout << "Cercle de rayon " << rayon;
    }
};

class Rectangle : public Forme {
private:
    double largeur, hauteur;
public:
```

```
    Rectangle(double l, double h) : largeur(l), hauteur(h) {}

    double aire() const override { return largeur * hauteur; }
    void afficher() const override {
        cout << "Rectangle " << largeur << "x" << hauteur;
    }
};

void traiterForme(const Forme& f) {
    f.afficher();
    cout << " -> aire = " << f.aire() << endl;
}

int main() {
    Cercle c(5.0);
    Rectangle r(4.0, 3.0);

    traiterForme(c);  // Comportement de Cercle
    traiterForme(r);  // Comportement de Rectangle

    return 0;
}
```

6.2. Classes abstraites

Une classe avec au moins une méthode virtuelle pure (= 0) est abstraite : on ne peut pas instancier d'objets de cette classe.

```
class Regulateur {
protected:
    double consigne;

public:
    Regulateur(double c) : consigne(c) {}

    virtual double calculer(double mesure) = 0;  // Méthode pure
    virtual ~Regulateur() {}
};

class RegulateurP : public Regulateur {
private:
    double kp;

public:
```

```
    RegulateurP(double c, double p) : Regulateur(c), kp(p) {}

    double calculer(double mesure) override {
        return kp * (consigne - mesure);
    }
};

class RegulateurPI : public Regulateur {
private:
    double kp, ki;
    double integrale;
    double dt;

public:
    RegulateurPI(double c, double p, double i, double pas)
        : Regulateur(c), kp(p), ki(i), integrale(0.0), dt(pas) {}

    double calculer(double mesure) override {
        double erreur = consigne - mesure;
        integrale += erreur * dt;
        return kp * erreur + ki * integrale;
    }
};

int main() {
    // Regulateur r(10.0);  // ERREUR : classe abstraite

    RegulateurP rp(20.0, 2.0);
    RegulateurPI rpi(20.0, 1.5, 0.1, 0.01);

    vector<Regulateur*> regulateurs;
    regulateurs.push_back(&rp);
    regulateurs.push_back(&rpi);

    double mesure = 15.0;
    for (auto r : regulateurs) {
        double cmd = r->calculer(mesure);
        cout << "Commande = " << cmd << endl;
    }

    return 0;
}
```

7. Synthèse

Concept	Rôle	Syntaxe clé
Classe	Modèle d'objet	`class Nom { ... };`
Objet	Instance de classe	`Nom obj;`
Attribut	Variable membre	`type nom;`
Méthode	Fonction membre	`type nom(params) { ... }`
Constructeur	Initialisation	`Nom(params) : init(...) { ... }`
Destructeur	Nettoyage	`~Nom() { ... }`
Encapsulation	Protection données	`private:`, `protected:`, `public:`
Héritage	Réutilisation	`class Fille : public Mere { ... };`
Polymorphisme	Comportement spécifique	`virtual type methode() = 0;`
Classe abstraite	Interface	Au moins une méthode pure

Règles d'or :

- Encapsulation : attributs privés, méthodes publiques
- Constructeur : toujours initialiser tous les attributs
- Destructeur : virtuel si héritage
- Héritage : "est-un" (un CapteurTemperature EST UN Capteur)
- Composition : "a-un" (une Voiture A UN Moteur) - souvent préférable à l'héritage
- Polymorphisme : utiliser pour traiter des objets de différentes classes uniformément