高等程序设计 - Qt/C++

第 4 章：编程范式详解

王培杰

长江大学地球物理与石油资源学院

2025 年 9 月 5 日

目录

目录

编程范式概念

什么是编程范式?

- 编程的基本风格和方法论
- 解决问题的思维方式和工具
- 代码组织和结构的方法
- 不同范式的组合使用

主要编程范式

- 过程式编程（Procedural Programming）
- 面向对象编程（Object-Oriented Programming）
- 泛型编程（Generic Programming）
- 函数式编程（Functional Programming）
- 事件驱动编程（Event-Driven Programming）
- 声明式编程（Declarative Programming）
- 组件式编程（Component-Based Programming）

目录

过程式编程概述

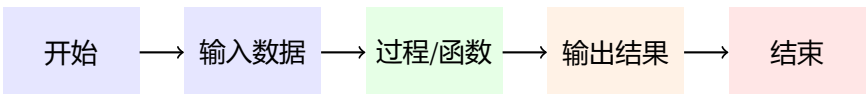
过程式编程特点

- 以过程/函数为中心
- 数据与操作分离
- 顺序执行
- 模块化设计
- 易于理解和调试

适用场景

- 算法实现
- 数据处理
- 工具函数
- 系统编程
- 性能关键代码

过程式编程示意图



过程式编程示例

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
// 过程式编程：数据处理示例
// 读取一组整数，排序后输出偶数
void inputData(std::vector<int>& data, int n) {
    std::cout << "请输入 " << n << " 个整数：" <<
    ↪ std::endl;
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        data.push_back(x);
    }
}
void sortData(std::vector<int>& data) {
    std::sort(data.begin(), data.end());
}
void printEven(const std::vector<int>& data) {
    std::cout << "排序后的偶数有：" << std::endl;
    for (int x : data) {
        if (x % 2 == 0) { std::cout << x << " "; }
    }
    std::cout << std::endl;
}
int main() {
    int n;
    std::cout << "请输入数据个数：";
    std::cin >> n;
    std::vector<int> data;
    inputData(data, n);
    sortData(data);
    printEven(data);
    std::cin.get();
    return 0;
}
```

过程式编程示例-算法实现

```cpp
#include <iostream>
#include <vector>
// 过程式编程：算法实现示例
void inputData(std::vector<int>& data, int n) {
    std::cout << "请输入 " << n << " 个整数：" <<
    ↪ std::endl;
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        data.push_back(x);
    }
}
void bubbleSort(std::vector<int>& data) {
    size_t n = data.size();
    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - 1 - i; ++j) {
            if (data[j] > data[j + 1]) {
                std::swap(data[j], data[j + 1]);
            }
        }
    }
}
void printData(const std::vector<int>& data) {
    std::cout << "排序后的结果为：" << std::endl;
    for (int x : data) {
        std::cout << x << " ";
    }
    std::cout << std::endl;
}
int main() {
    int n;
    std::cout << "请输入数据个数：";
    std::cin >> n;
    std::vector<int> data;
    inputData(data, n);
    bubbleSort(data);
    printData(data);
    std::cin.get();
    return 0;
}
```

过程式编程示例-数据处理

```cpp
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>
// 过程式编程：数据处理简化版
int main() {
    int n;
    std::cout << "请输入数据个数：";
    std::cin >> n;
    std::vector<double> data(n);
    std::cout << "请输入 " << n << " 个数据（浮点数）："
    ↪  << std::endl;
    for (auto& x : data) std::cin >> x;
    double sum = data.empty() ? 0.0 :
    ↪  std::accumulate(data.begin(), data.end(),
    ↪  0.0);
    double avg = data.empty() ? 0.0 : sum /
    ↪  data.size();
    double min = data.empty() ? 0.0 :
    ↪  *std::min_element(data.begin(), data.end());
    double max = data.empty() ? 0.0 :
    ↪  *std::max_element(data.begin(), data.end());
    std::sort(data.begin(), data.end());
    double median = data.empty() ? 0.0 :
    ↪  data[data.size() / 2];
    double std = data.empty() ? 0.0 :
    ↪  std::sqrt(std::accumulate(data.begin(),
    ↪  data.end(), 0.0, [avg](double a, double b) {
        return a + (b - avg) * (b - avg);
    }) / data.size());
    double var = data.empty() ? 0.0 :
    ↪  std::accumulate(data.begin(), data.end(),
    ↪  0.0, [avg](double a, double b) {
        return a + (b - avg) * (b - avg);
    }) / data.size();
    std::cout << "数据总和：" << sum << std::endl;
    std::cout << "数据个数：" << data.size() <<
    ↪  std::endl;
    std::cout << "平均值：" << avg << std::endl;
    std::cout << "最小值：" << min << std::endl;
    std::cout << "最大值：" << max << std::endl;
    std::cout << "中位数：" << median << std::endl;
    std::cout << "标准差：" << std << std::endl;
    std::cout << "方差：" << var << std::endl;
    std::cin.get();
    return 0;
}
```

过程式编程示例-工具函数

```cpp
#include <iostream>
#include <filesystem>
#include <string>
namespace fs = std::filesystem;
// 批量重命名文件的工具函数
void batchRenameFiles(const std::string& dirPath,
    const std::string& prefix) {
    if (!fs::exists(dirPath)
        !fs::is_directory(dirPath)) {
        std::cout << "目录不存在: " << dirPath <<
            std::endl;
        return;
    }
    int count = 1;
    for (const auto& entry :
        fs::directory_iterator(dirPath)) {
        if (entry.is_regular_file()) {
            std::string oldPath =
                entry.path().string();
            std::string extension =
                entry.path().extension().string();
            std::string newName = prefix +
                std::to_string(count) + extension;
            std::string newPath = (fs::path(dirPath)
                / newName).string();
            if (fs::exists(newPath)) {
                std::cout << "跳过已存在的文件: " <<
                    newPath << std::endl;
            } else {
                fs::rename(oldPath, newPath);
                std::cout << "重命名: " << oldPath <<
                    " -> " << newPath << std::endl;
            }
            ++count;
        }
    }
}
int main() {
    std::string dirPath, prefix;
    std::cout << "请输入要批量重命名的文件夹路径: ";
    std::getline(std::cin, dirPath);
    std::cout << "请输入新的文件名前缀: ";
    std::getline(std::cin, prefix);
    batchRenameFiles(dirPath, prefix);
    std::cout << "批量重命名完成。" << std::endl;
    std::cin.get();
```

过程式编程示例-系统编程

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <filesystem>
namespace fs = std::filesystem;

// 过程式编程：系统编程示例——统计指定目录下的文件数量和总大小
int main() {
    std::string dirPath;
    std::cout << "请输入要统计的文件夹路径：";
    std::getline(std::cin, dirPath);

    if (!fs::exists(dirPath) || !fs::is_directory(dirPath)) {
        std::cout << "目录不存在：" << dirPath << std::endl;
        return 1;
    }

    size_t fileCount = 0;
    uintmax_t totalSize = 0;

    for (const auto& entry : fs::directory_iterator(dirPath)) {
        if (entry.is_regular_file()) {
            ++fileCount;
            totalSize += entry.file_size();
        }
    }

    std::cout << "文件夹：" << dirPath << std::endl;
    std::cout << "文件数量：" << fileCount << std::endl;
    std::cout << "总大小：" << totalSize << " 字节" << std::endl;

    std::cin.get();
    return 0;
}
```

过程式编程示例-性能关键代码

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <random>

// 过程式编程：性能关键代码示例——大规模数组求和
int main() {
    const size_t N = 100000000; // 1 亿个元素
    std::vector<int> data(N);

    // 随机填充数据
    std::mt19937 rng(42);
    std::uniform_int_distribution<int> dist(1, 100);
    for (size_t i = 0; i < N; ++i) {
        data[i] = dist(rng);
    }

    // 性能计时开始
    auto start = std::chrono::high_resolution_clock::now();

    // 求和
    long long sum = 0;
    for (size_t i = 0; i < N; ++i) {
        sum += data[i];
    }

    // 性能计时结束
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;

    std::cout << "数据总和: " << sum << std::endl;
    std::cout << "耗时: " << elapsed.count() << " 秒" << std::endl;

    std::cin.get();
    return 0;
}
```

目录

面向对象编程概述

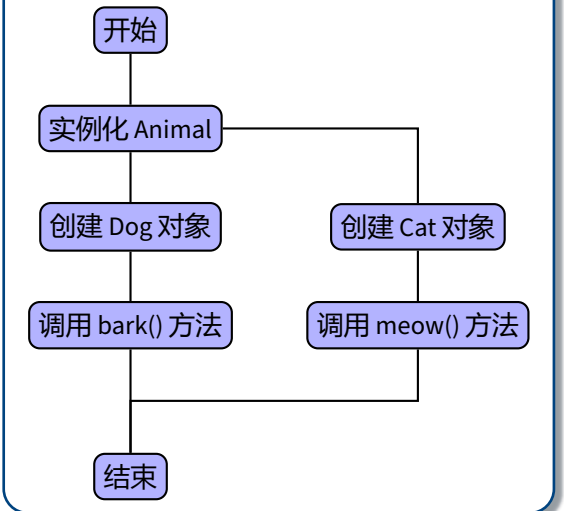
OOP 核心概念

- 封装（Encapsulation）
- 继承（Inheritance）
- 多态（Polymorphism）
- 抽象（Abstraction）

Qt 中的 OOP 应用

- QWidget 继承体系
- 信号槽机制
- 属性系统
- 事件处理

面向对象编程示意图



面向对象编程示例

```cpp
#include <iostream>
#include <string>
class Animal {
public:
    Animal(const std::string& name) : m_name(name) {}
    virtual ~Animal() {}
    virtual void speak() const = 0; // 纯虚函数
    std::string name() const { return m_name; }
protected:
    std::string m_name;
};
class Dog : public Animal {
public:
    Dog(const std::string& name) : Animal(name) {}

    void speak() const override {
        std::cout << "Dog " << m_name << " says: 汪
        ↪ 汪!" << std::endl;
    }
};
```

```cpp

class Cat : public Animal {
public:
    Cat(const std::string& name) : Animal(name) {}
    void speak() const override {
        std::cout << "Cat " << m_name << " says: 喵
        ↪ 喵!" << std::endl;
    }
};

int main() {
    Animal* dog = new Dog("小黑");
    Animal* cat = new Cat("小花");
    dog->speak();
    cat->speak();
    delete dog;
    delete cat;
    std::cin.get();
    return 0;
}
```

面向对象的适用场景

- 代码重用: 通过继承和多态实现代码重用。
- GUI 应用程序: 通过面向对象设计实现 GUI 应用程序。
- 模拟现实世界: 在游戏开发中创建角色、车辆等对象。
- 复杂系统: 通过封装和抽象实现复杂系统。
- 大型项目: 通过模块化设计实现大型项目。
- 团队协作: 通过面向对象设计实现团队协作。
- 框架开发: 通过面向对象设计实现框架开发。

面向对象编程示例-代码重用（封装）

```cpp
#include <iostream>
#include <string>

// 封装：将数据和操作数据的方法封装在类中
class Account {
public:
    Account(const std::string& owner, double balance)
        : owner(owner), balance(balance) {}

    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }

    double getBalance() const {
        return balance;
    }

private:
    std::string owner;
    double balance; // 封装：外部无法直接访问
};

int main() {
    Account acc("张三", 1000.0);
    acc.deposit(500.0);
    acc.withdraw(200.0);
    std::cout << "最终余额：" << acc.getBalance() << "
    ↪  元" << std::endl;
    std::cin.get();
    return 0;
}
```

面向对象编程示例-代码重用（继承）

```cpp
#include <iostream>
#include <string>
class Animal {
public:
    Animal(const std::string& name) : name(name) {}
    void eat() const {
        std::cout << name << " 正在吃东西。" <<
        ↪ std::endl;
    }
    void sleep() const {
        std::cout << name << " 正在睡觉。" <<
        ↪ std::endl;
    }
protected:
    std::string name;
};
class Dog : public Animal {
public:
    Dog(const std::string& name) : Animal(name) {}
    void bark() const {
        std::cout << name << ": 汪汪！" << std::endl;
    }
};
class Cat : public Animal {
public:
    Cat(const std::string& name) : Animal(name) {}
    void meow() const {
        std::cout << name << ": 喵喵！" << std::endl;
    }
};
int main() {
    Dog dog("小黑");
    Cat cat("小花");
    dog.eat();
    dog.bark();
    dog.sleep();
    cat.eat();
    cat.meow();
    cat.sleep();
    std::cin.get();
    return 0;
```

面向对象编程示例-代码重用（多态）

```cpp
#include <iostream>
#include <vector>
class Animal {
public:
    Animal(const std::string& name) : name(name) {}
    virtual void makeSound() const { std::cout <<
    ↪  name << " 发出未知的叫声。\n"; }
    virtual ~Animal() {}
protected:
    std::string name;
};
class Dog : public Animal {
public:
    Dog(const std::string& name) : Animal(name) {}
    void makeSound() const override { std::cout <<
    ↪  name << ": 汪汪！\n"; }
};
class Cat : public Animal {
public:
    Cat(const std::string& name) : Animal(name) {}
    void makeSound() const override { std::cout <<
    ↪  name << ": 喵喵！\n"; }
};

int main() {
    Dog dog("小黑");
    Cat cat("小花");
    Animal generic("神秘动物");
    std::vector<Animal*> animals = {&dog, &cat,
    ↪  &generic};
    for (auto a : animals) a->makeSound();
    std::cin.get();
    return 0;
}
```

面向对象编程示例-GUI

```cpp
#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QVBoxLayout>
#include <QLabel>
class MyWindow : public QWidget {
    Q_OBJECT
public:
    MyWindow(QWidget* parent = nullptr) :
    ↪  QWidget(parent) {
        QVBoxLayout* layout = new QVBoxLayout(this);
        label = new QLabel(QStringLiteral("你好，Qt
        ↪  OOP GUI 示例！"), this);
        QPushButton* button = new
        ↪  QPushButton(QStringLiteral("点击我"),
        ↪  this);
        layout->addWidget(label);
        layout->addWidget(button);
        connect(button, &QPushButton::clicked, this,
        ↪  &MyWindow::onButtonClicked);
    }
private slots:
    void onButtonClicked() {
    ↪  label->setText(QStringLiteral("按钮已被点击！
    ↪  ")); }
private:
    QLabel* label;
};
#include "oop_example_gui.moc"
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MyWindow window;
    window.setWindowTitle(QStringLiteral("OOP GUI 示
    ↪  例"));
    window.resize(300, 120);
    window.show();
    return app.exec();
}
```

面向对象编程示例-模拟现实世界（图书馆-书籍-读者关系）

```cpp
#include <iostream>
#include <vector>
#include <string>
// 面向对象编程示例：模拟现实世界中的“图书馆-书籍-读者”关系
class Book {
public:
    Book(const std::string& title, const std::string& author)
        : title(title), author(author), borrowed(false) {}
    std::string getTitle() const { return title; }
    std::string getAuthor() const { return author; }
    bool isBorrowed() const { return borrowed; }
    void borrow() { borrowed = true; }
    void giveBack() { borrowed = false; }
private:
    std::string title;
    std::string author;
    bool borrowed;
};
```

```cpp

class Reader {
public:
    Reader(const std::string& name) : name(name) {}
    void borrowBook(Book& book) {
        if (!book.isBorrowed()) {
            book.borrow();
            borrowedBooks.push_back(&book);
            std::cout << name << " 借阅了《" << book.getTitle() << "》" << std::endl;
        } else {
            std::cout << "《" << book.getTitle() << "》已被借出，无法借阅。" << std::endl;
        }
    }
    void returnBook(Book& book) {
        for (auto it = borrowedBooks.begin(); it != borrowedBooks.end(); ++it) {
            if (*it == &book) {
                book.giveBack();
                borrowedBooks.erase(it);
```

面向对象编程示例-模拟现实世界（图书馆-书籍-读者关系）

```
37                  std::cout << name << " 归还了《" <<
                    ↪  book.getTitle() << "》" <<
                    ↪  std::endl;
38                  return;
39              }
40          }
41          std::cout << name << " 没有借阅《" <<
            ↪  book.getTitle() << "》，无法归还。" <<
            ↪  std::endl;
42      }
43      void listBorrowedBooks() const {
44          std::cout << name << " 当前借阅的书籍：" <<
            ↪  std::endl;
45          if (borrowedBooks.empty()) {
46              std::cout << "  无" << std::endl;
47          } else {
48              for (const auto* book : borrowedBooks) {
49                  std::cout << "  《" <<
                    ↪  book->getTitle() << "》 by " <<
                    ↪  book->getAuthor() << std::endl;
50              }
51          }
52      }
```

```
53  private:
54      std::string name;
55      std::vector<Book*> borrowedBooks;
56  };
57
58  class Library {
59  public:
60      void addBook(const Book& book) {
61          books.push_back(book);
62      }
63      Book* findBook(const std::string& title) {
64          for (auto& book : books) {
65              if (book.getTitle() == title) {
66                  return &book;
67              }
68          }
69          return nullptr;
70      }
71      void listBooks() const {
72          std::cout << "图书馆藏书列表：" << std::endl;
```

面向对象编程示例-模拟现实世界（图书馆-书籍-读者关系）

```
        for (const auto& book : books) {
            std::cout << "  《" << book.getTitle() <<
            ↪  "》 by " << book.getAuthor();
            if (book.isBorrowed()) {
                std::cout << "（已借出）";
            }
            std::cout << std::endl;
        }
    }
private:
    std::vector<Book> books;
};
int main() {
    Library library;
    library.addBook(Book("C++ Primer", "Stanley B.
    ↪  Lippman"));
    library.addBook(Book("深入理解计算机系统", "Randal
    ↪  E. Bryant"));
    library.addBook(Book("算法导论", "Thomas H.
    ↪  Cormen"));
    Reader alice("Alice");
    Reader bob("Bob");
    library.listBooks();
    std::cout << std::endl;
    Book* book1 = library.findBook("C++ Primer");
    Book* book2 = library.findBook("算法导论");
    if (book1) alice.borrowBook(*book1);
    if (book2) bob.borrowBook(*book2);
    if (book1) bob.borrowBook(*book1); // 尝试借已借出
    ↪  的书
    std::cout << std::endl;
    alice.listBorrowedBooks();
    bob.listBorrowedBooks();
    std::cout << std::endl;
    library.listBooks();
    std::cout << std::endl;
    if (book1) alice.returnBook(*book1);
    if (book1) bob.borrowBook(*book1); // 现在 Bob 可
    ↪  以借了
    std::cout << std::endl;
    library.listBooks();
    std::cin.get();
    return 0;
}
```

面向对象编程示例-复杂系统（机器人）

```cpp
#include <iostream>
#include <vector>
#include <string>

// 组件：传感器
class Sensor {
public:
    Sensor(const std::string& type) : type(type) {}
    void read() const {
        std::cout << "传感器 [" << type << "] 正在读取数据。" << std::endl;
    }
private:
    std::string type;
};

// 组件：执行器
class Actuator {
public:
    Actuator(const std::string& type) : type(type) {}
    void activate() const {
        std::cout << "执行器 [" << type << "] 被激活。" << std::endl;
    }
private:
    std::string type;
};

// 子系统：机器人手臂
class RobotArm {
public:
    RobotArm(const std::string& name) : name(name) {}
    void addActuator(const Actuator& actuator) {
        actuators.push_back(actuator);
    }
    void move() const {
        std::cout << "机器人手臂 [" << name << "] 开始移动。" << std::endl;
        for (const auto& a : actuators) {
```

面向对象编程示例-复杂系统（机器人）

```
            a.activate();
        }
    }
private:
    std::string name;
    std::vector<Actuator> actuators;
};

// 子系统：机器人传感系统
class RobotSensorSystem {
public:
    void addSensor(const Sensor& sensor) {
        sensors.push_back(sensor);
    }
    void scan() const {
        std::cout << "机器人传感系统开始扫描环境。" <<
        ↪ std::endl;
        for (const auto& s : sensors) {
            s.read();
        }
    }
private:
    std::vector<Sensor> sensors;
};

// 复杂系统：机器人
class Robot {
public:
    Robot(const std::string& name) : name(name) {}
    void addArm(const RobotArm& arm) {
        arms.push_back(arm);
    }
    void setSensorSystem(const RobotSensorSystem&
    ↪ sensorSystem) {
        this->sensorSystem = sensorSystem;
    }
    void operate() const {
        std::cout << "机器人 [" << name << "] 启动。"
        ↪ << std::endl;
```

面向对象编程示例-复杂系统（机器人）

```
        sensorSystem.scan();
        for (const auto& arm : arms) {
            arm.move();
        }
        std::cout << "机器人 [" << name << "] 操作完
        ↪  成。" << std::endl;
    }
private:
    std::string name;
    std::vector<RobotArm> arms;
    RobotSensorSystem sensorSystem;
};

int main() {
    // 创建传感器和执行器
    Sensor tempSensor("温度");
    Sensor camSensor("摄像头");
    Actuator motor("电机");
    Actuator gripper("夹爪");

    // 创建手臂并添加执行器
    RobotArm leftArm("左臂");
    leftArm.addActuator(motor);
    leftArm.addActuator(gripper);

    // 创建传感系统并添加传感器
    RobotSensorSystem sensorSystem;
    sensorSystem.addSensor(tempSensor);
    sensorSystem.addSensor(camSensor);

    // 创建机器人并组装
    Robot robot("Alpha");
    robot.addArm(leftArm);
    robot.setSensorSystem(sensorSystem);

    // 启动机器人
    robot.operate();

    std::cin.get();
    return 0;
}
```

面向对象编程示例-大型项目（游戏引擎）

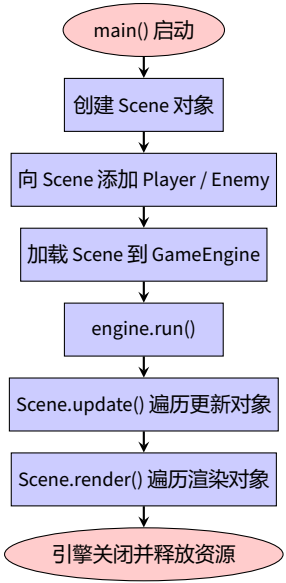
```cpp
#include <iostream>
#include <vector>
#include <string>
// 组件：游戏对象基类
class GameObject {
public:
    GameObject(const std::string& name) : name(name) {}
    virtual void update() = 0;
    virtual void render() = 0;
    virtual ~GameObject() {}
protected:
    std::string name;
};
// 组件：玩家
class Player : public GameObject {
public:
    Player(const std::string& name) : GameObject(name) {}
    void update() override {
        std::cout << "玩家 [" << name << "] 正在更新状态。" << std::endl;
    }
    void render() override {
        std::cout << "渲染玩家 [" << name << "]。" << std::endl;
    }
};
// 组件：敌人
class Enemy : public GameObject {
public:
    Enemy(const std::string& name) : GameObject(name) {}
    void update() override {
        std::cout << "敌人 [" << name << "] 正在巡逻。" << std::endl;
    }
    void render() override {
        std::cout << "渲染敌人 [" << name << "]。" << std::endl;
    }
};
```

面向对象编程示例-大型项目（游戏引擎）

```
// 子系统：场景管理
class Scene {
public:
    void addObject(GameObject* obj) {
        objects.push_back(obj);
    }
    void update() {
        std::cout << "场景更新所有对象：" << std::endl;
        for (auto obj : objects) {
            obj->update();
        }
    }
    void render() {
        std::cout << "场景渲染所有对象：" << std::endl;
        for (auto obj : objects) {
            obj->render();
        }
    }
    ~Scene() {
        for (auto obj : objects) {
            delete obj;
        }
    }
};
// 游戏引擎
class GameEngine {
public:
    void loadScene(Scene* scene) {
        this->scene = scene;
    }
    void run() {
        std::cout << "游戏引擎启动。" << std::endl;
        if (scene) {
            scene->update();
            scene->render();
        }
        std::cout << "游戏引擎关闭。" << std::endl;
    }
private:
    Scene* scene = nullptr;
};
```

面向对象编程示例-大型项目（游戏引擎）

```
80  int main() {
81      Scene* scene = new Scene();// 创建场景
82      scene->addObject(new Player("主角"));
83      scene->addObject(new Enemy("怪物 A"));
84      scene->addObject(new Enemy("怪物 B"));// 创建游戏
        ↪  引擎并加载场景
85      GameEngine engine;
86      engine.loadScene(scene);
87      engine.run();// 运行游戏
88      delete scene;// 释放资源
89      std::cin.get();
90      return 0;
```



目录

泛型编程概述

什么是泛型编程

- 泛型编程是一种编程范式，它允许你编写与类型无关的代码。例如：与类型无关的算法。
- 泛型编程的目的是提高代码的复用性和可维护性。
- 泛型编程的实现方式是使用模板。

泛型编程特点

- 类型无关的算法
- 编译时多态
- 代码复用
- 性能优化
- 类型安全

Qt 中的泛型应用

- QList 等容器类
- 算法模板
- 智能指针
- 类型推导

模板——编译时多态

模板的编译时多态

- 模板在编译阶段实例化，生成针对不同类型的高效代码，无运行时开销。
- 编译器会对模板参数进行类型检查，保证类型安全。
- 支持类型自动推导，简化模板的使用。
- 通过模板特化和重载，实现灵活的类型适配。
- C++20 引入 concepts（概念），可对模板参数类型进行约束，提升可读性和错误提示。

泛型编程示例-函数模板-单参数模板

```cpp
#include <iostream>

template<typename T>
T TemplateAdd(const T& a, const T& b) {
    return a + b;
}

int NormalAdd(const int& a, const int& b) {
    return a + b;
}

float NormalAdd(const float& a, const float& b) {
    return a + b;
}

double NormalAdd(const double& a, const double& b) {
    return a + b;
}
```

```cpp
int main() {
    int a = 10, b = 20;
    std::cout << "TemplateAdd(a, b) = " << TemplateAdd(a, b) << std::endl;
    std::cout << "NormalAdd(a, b) = " << NormalAdd(a, b) << std::endl;
    float x = 3.14, y = 2.71;
    std::cout << "TemplateAdd(x, y) = " << TemplateAdd(x, y) << std::endl;
    std::cout << "NormalAdd(x, y) = " << NormalAdd(x, y) << std::endl;
    double z = 3.14, w = 2.71;
    std::cout << "TemplateAdd(z, w) = " << TemplateAdd(z, w) << std::endl;
    std::cout << "NormalAdd(z, w) = " << NormalAdd(z, w) << std::endl;
    std::cin.get();
    return 0;
}
```

泛型编程示例-函数模板-多参数模板

```cpp
#include <iostream>

#include <type_traits>

template<typename T1, typename T2>
auto TemplateAdd(const T1& a, const T2& b) -> decltype(a + b) {
    return a + b;
}

float NormalAdd(const int& a, const float& b) {
    return a + b;
}
```

```cpp
int main() {
    int a = 10;
    float c = 10.1;
    double e = 10.3;
    std::cout << "TemplateAdd(a, c) = " << TemplateAdd(a, c) << std::endl;
    std::cout << "NormalAdd(a, c) = " << NormalAdd(a, c) << std::endl;
    std::cout << "TemplateAdd(a, e) = " << TemplateAdd(a, e) << std::endl;
    std::cout << "NormalAdd(a, e) = " << NormalAdd(a, e) << std::endl;
    std::cout << "TemplateAdd(c, e) = " << TemplateAdd(c, e) << std::endl;
    std::cout << "NormalAdd(c, e) = " << NormalAdd(c, e) << std::endl;
    std::cin.get();
    return 0;
}
```

泛型编程示例-函数模板-模板特化

```cpp
#include <iostream>
#include <type_traits>

// 通用模板
template<typename T1, typename T2>
auto TemplateAdd(const T1& a, const T2& b) -> decltype(a + b) {
    return a + b;
}

// 针对 bool 类型的重载
bool TemplateAddBool(const bool& a, const bool& b) {
    return a  b;
}
```

```cpp
int main() {
    bool a = true, b = false;
    std::cout << std::boolalpha;
    std::cout << "TemplateAdd(a, b) = " << TemplateAddBool(a, b) << std::endl;

    int x = 5, y = 7;
    std::cout << "TemplateAdd(x, y) = " << TemplateAdd(x, y) << std::endl;
    std::cin.get();
    return 0;
}
```

泛型编程示例-函数模板-可变参数模板

```cpp
#include <iostream>

// 递归终止：只有一个参数时直接返回
template<typename T>
T TemplateSum(const T& value) {
    return value;
}

// 辅助函数，用于实现递归求和
template<typename T, typename... Args>
auto TemplateSum(const T& first, const Args&... args)
    -> {
    return first + TemplateSum(args...);
}
```

```cpp
int main() {
    int a = 1, b = 2, c = 3;
    double d = 4.5, e = 5.5;
    std::cout << "TemplateSum(a, b, c) = " <<
    -> TemplateSum(a, b, c) << std::endl;
    std::cout << "TemplateSum(a, b, c, d) = " <<
    -> TemplateSum(a, b, c, d) << std::endl;
    std::cout << "TemplateSum(a, b, c, d, e) = " <<
    -> TemplateSum(a, b, c, d, e) << std::endl;
    std::cin.get();
    return 0;
}
```

泛型编程示例-函数模板-概念 (Concepts) - C++20

```cpp
#include <iostream>
#include <concepts>
#include <string>

// 定义一个概念：要求类型 T 支持加法运算
template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};
template<Addable T>
T add(T a, T b) {
    return a + b;
}
```

```cpp
int main() {
    int x = 10, y = 20;
    std::cout << "int add: " << add(x, y) <<
    ↪  std::endl;

    double d1 = 3.14, d2 = 2.71;
    std::cout << "double add: " << add(d1, d2) <<
    ↪  std::endl;

    std::string s1 = "Hello, ", s2 = "World!";
    std::cout << "string add: " << add(s1, s2) <<
    ↪  std::endl;

    // 下列代码将无法通过编译，因为指针不满足 Addable 概
    ↪  念
    // int* p1 = &x, *p2 = &y;
    // std::cout << "pointer add: " << add(p1, p2) <<
    ↪  std::endl;

    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-单参数模板

```cpp
#include <iostream>

// 类模板定义
template<typename T>
class MyPair {
public:
    MyPair(const T& first, const T& second)
        : m_first(first), m_second(second) {}

    T getFirst() const { return m_first; }
    T getSecond() const { return m_second; }

    void print() const {
        std::cout << "First: " << m_first << ",
        ↪  Second: " << m_second << std::endl;
    }

private:
    T m_first;
    T m_second;
};
```

```cpp
int main() {
    MyPair<int> intPair(10, 20);
    intPair.print();

    MyPair<double> doublePair(3.14, 2.71);
    doublePair.print();

    MyPair<std::string> stringPair("Hello", "World");
    stringPair.print();

    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-多参数模板

```cpp
#include <iostream>
#include <string>

// 多参数类模板定义
template<typename T1, typename T2>
class MyPair {
public:
    MyPair(const T1& first, const T2& second)
        : m_first(first), m_second(second) {}

    T1 getFirst() const { return m_first; }
    T2 getSecond() const { return m_second; }

    void print() const {
        std::cout << "First: " << m_first << ",
        ↪  Second: " << m_second << std::endl;
    }

private:
    T1 m_first;
    T2 m_second;
};
```

```cpp
int main() {
    MyPair<int, double> pair1(10, 3.14);
    pair1.print();

    MyPair<std::string, int> pair2("Age", 30);
    pair2.print();

    MyPair<std::string, std::string> pair3("Hello",
    ↪  "World");
    pair3.print();
    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-非类型模板参数（常量）

```cpp
#include <iostream>

template<typename T, int N>
class ArrayWrapper {
public:
    ArrayWrapper() { for (int i = 0; i < N; ++i) m_data[i] = T(); }
    T& operator[](int i) { return m_data[i]; }
    int size() const { return N; }
    void print() const {
        for (int i = 0; i < N; ++i) std::cout << m_data[i] << " ";
        std::cout << std::endl;
    }
private:
    T m_data[N];
};
```

```cpp
int main() {
    ArrayWrapper<int, 5> a;
    for (int i = 0; i < a.size(); ++i) a[i] = i * 2;
    a.print();
    ArrayWrapper<double, 3> b;
    b[0] = 3.14; b[1] = 2.71; b[2] = 1.41;
    b.print();
    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-非类型模板参数（指针和引用）

```cpp
#include <iostream>

// 非类型模板参数：指针
template<typename T, T* ptr>
class PointerWrapper {
public:
    void print() const {
        std::cout << "Pointer value: " << *ptr << std::endl;
    }
};

// 非类型模板参数：引用
template<typename T, T& ref>
class ReferenceWrapper {
public:
    void print() const {
        std::cout << "Reference value: " << ref << std::endl;
    }
};
```

```cpp
int globalInt = 42;
double globalDouble = 3.14;
int main() {
    // 指针作为非类型模板参数
    PointerWrapper<int, &globalInt> intPtrWrapper;
    intPtrWrapper.print();

    PointerWrapper<double, &globalDouble> doublePtrWrapper;
    doublePtrWrapper.print();

    // 引用作为非类型模板参数
    ReferenceWrapper<int, globalInt> intRefWrapper;
    intRefWrapper.print();

    ReferenceWrapper<double, globalDouble> doubleRefWrapper;
    doubleRefWrapper.print();

    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-模板模板参数（容器）

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <string>
// 模板模板参数示例：容器包装类
template <template <typename, typename> class
    Container, typename T>
class ContainerWrapper {
public:
    ContainerWrapper() = default;
    void add(const T& value) {
        m_container.push_back(value);
    }
    void print() const {
        std::cout << "Container contents: ";
        for (const auto& item : m_container) {
            std::cout << item << " ";
        }
        std::cout << std::endl;
    }
private:
    Container<T, std::allocator<T>> m_container;
};
int main() {
    // 使用 std::vector 作为模板模板参数
    ContainerWrapper<std::vector, int>
        intVectorWrapper;
    intVectorWrapper.add(1);
    intVectorWrapper.add(2);
    intVectorWrapper.add(3);
    intVectorWrapper.print();

    // 使用 std::list 作为模板模板参数
    ContainerWrapper<std::list, std::string>
        stringListWrapper;
    stringListWrapper.add("Hello");
    stringListWrapper.add("World");
    stringListWrapper.print();
    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-默认模板参数

```cpp
#include <iostream>
#include <string>

template<typename T = int, int N = 10>
class DefaultArray {
public:
    DefaultArray() { for (int i = 0; i < N; ++i) m_data[i] = T(); }
    T& operator[](int i) { return m_data[i]; }
    int size() const { return N; }
    void print() const {
        for (int i = 0; i < N; ++i) std::cout << m_data[i] << " ";
        std::cout << std::endl;
    }
private:
    T m_data[N];
};

int main() {
    DefaultArray<> a;
    for (int i = 0; i < a.size(); ++i) a[i] = i + 1;
    a.print();

    DefaultArray<double> b;
    for (int i = 0; i < b.size(); ++i) b[i] = i * 1.1;
    b.print();

    DefaultArray<std::string, 3> c;
    c[0] = "Hello"; c[1] = "C++"; c[2] = "Template";
    c.print();
    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-成员模板

```cpp
#include <iostream>
#include <string>

// 类模板，包含成员模板方法
template<typename T>
class Adder {
public:
    Adder(const T& a, const T& b) : m_a(a), m_b(b) {}
    T sum() const { return m_a + m_b; }
    void print() const { std::cout << "Sum: " << sum() << std::endl; }
    // 成员模板：可以将任意类型 U 的 Adder 与当前对象的 sum 结果相加
    template<typename U>
    auto addWith(const Adder<U>& other) const { return sum() + other.sum(); }
private:
    T m_a, m_b;
};
```

```cpp
int main() {
    Adder<int> intAdder(10, 20);
    intAdder.print();
    Adder<double> doubleAdder(3.14, 2.71);
    doubleAdder.print();
    Adder<std::string> stringAdder("Hello, ", "World!");
    stringAdder.print();
    // 使用成员模板，将不同类型 Adder 的 sum 结果相加
    auto mixedSum = intAdder.addWith(doubleAdder); // int + double
    std::cout << "intAdder + doubleAdder = " << mixedSum << std::endl;
    // 字符串与 intAdder 的 sum 结果相加（需要转换为字符串）
    // 这里演示成员模板的灵活性
    // std::string + int 需要先转换
    std::string strSum = stringAdder.sum() + std::to_string(intAdder.sum());
    std::cout << "stringAdder + intAdder = " << strSum << std::endl;
    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-继承模板

```cpp
#include <iostream>
#include <string>
// 基类模板
template<typename T>
class Base {
public:
    Base(const T& value) : m_value(value) {}
    void show() const { std::cout << "Base value: "
    ↪  << m_value << std::endl; }
protected:
    T m_value;
};
// 派生类模板，继承自基类模板
template<typename T>
class Derived : public Base<T> {
public:
    Derived(const T& value, const std::string& name)
        : Base<T>(value), m_name(name) {}
    void show() const { std::cout << "Derived name: "
    ↪  << m_name << ", value: " << this->m_value <<
    ↪  std::endl; }
    void showBase() const { Base<T>::show(); }
private:
    std::string m_name;
};

int main() {
    Base<int> baseObj(100);
    baseObj.show();
    Derived<int> derivedObj(200, "派生对象");
    derivedObj.show();
    derivedObj.showBase();
    Derived<std::string> derivedStrObj("Hello", "字符
    ↪  串派生");
    derivedStrObj.show();
    derivedStrObj.showBase();
    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-SFINAE (Substitution Failure Is Not An Error)

SFINAE (Substitution Failure Is Not An Error)

在编译时，如果模板参数不满足条件，编译器会尝试其他可能的模板参数。如果所有可能的模板参数都失败，编译器会报错。

```cpp
#include <iostream>
#include <type_traits>
#include <string>
// SFINAE 示例：只有当 T 支持 operator<< 时，才启用 print 方法
template<typename T, typename = void>
class Printer {
public:
    void print(const T&) { std::cout << "不支持输出该类型" << std::endl; }
};
// 偏特化：当 T 支持 operator<< 时，启用此版本
template<typename T>
class Printer<T,
    std::void_t<decltype(std::declval<std::ostream&>() << std::declval<T>())>> {
public:
    void print(const T& value) { std::cout << "输出: " << value << std::endl; }
};
struct NoOstreamType {};
int main() {
    Printer<int> intPrinter;
    intPrinter.print(123); // 支持输出
    Printer<std::string> strPrinter;
    strPrinter.print("Hello SFINAE"); // 支持输出
    Printer<NoOstreamType> noPrinter;
    noPrinter.print(NoOstreamType{}); // 不支持输出该类型
    std::cin.get();
    return 0;
}
```

泛型编程示例-类模板-变长模板参数（C++20）

```cpp
#include <iostream>
#include <string>
#include <tuple>
// 变长模板参数类，支持任意数量和类型的参数
template<typename... Args>
class VariadicPrinter {
public:
    // 构造函数，使用参数包初始化元组
    VariadicPrinter(const Args&... args) : m_values(args...) {}
    // 打印所有参数
    void print() const { printTuple(m_values); }
private:
    std::tuple<Args...> m_values; // 存储所有参数的元组
    // 递归打印元组中的每个元素
    template<std::size_t I = 0>
    void printTuple(const std::tuple<Args...>& t) const {
        if constexpr (I < sizeof...(Args)) { // 编译期判断是否还有元素
            std::cout << std::get<I>(t) << " "; // 打印第 I 个元素
            printTuple<I + 1>(t); // 递归打印下一个元素
        }}};
int main() {
    // 打印 int、double、string 类型的参数
    VariadicPrinter<int, double, std::string> printer(42, 3.14, "Hello");
    printer.print(); std::cout << std::endl;
    // 打印空参数（不会输出任何内容）
    VariadicPrinter<> emptyPrinter;
    emptyPrinter.print(); std::cout << std::endl;
    // 打印多个 string 和 int 类型参数
    VariadicPrinter<std::string, std::string, int> strPrinter("C++", "模板", 2023);
    strPrinter.print(); std::cout << std::endl;
    std::cin.get();
    return 0;
}
```

模板元编程概述

模板元编程（Template Metaprogramming）

- 利用 C++ 模板机制，在**编译期**进行类型和常量的计算与推导。
- 代码在编译阶段生成，**无运行时开销**，提升性能。
- 可实现**编译时多态**，如类型选择、特化、SFINAE 等。
- 支持**递归**与**条件分支**，可实现复杂的编译期逻辑。
- 常用于类型萃取（type traits）、静态断言、静态循环、类型变换等高级用法。
- 现代 C++（C++11 及以后）引入了 `constexpr`、`if constexpr`、`std::integral_constant` 等工具，极大简化了模板元编程。

典型应用场景

- 静态断言与类型检查（如 `static_assert`、SFINAE、concepts）
- 编译期常量计算（如阶乘、斐波那契数列等）
- 类型萃取与类型变换（如 `std::remove_const`、`std::is_same` 等）
- 优化泛型代码的分支与选择（如根据类型选择不同实现）

泛型编程示例-模板元编程-类型萃取

```cpp
#include <iostream>
#include <string>

// 移除 const 修饰符
template<typename T> struct remove_const { using type = T; };
template<typename T> struct remove_const<const T> { using type = T; };

// 判断类型是否相同
template<typename T, typename U> struct is_same { static constexpr bool value = false; };
template<typename T> struct is_same<T, T> { static constexpr bool value = true; };

// 判断是否为指针
template<typename T> struct is_pointer { static constexpr bool value = false; };
template<typename T> struct is_pointer<T*> { static constexpr bool value = true; };
```

```cpp
int main() {
    std::cout << std::boolalpha;
    std::cout << "remove_const<const int>::type 是否为 int: "
              << is_same<remove_const<const int>::type, int>::value << std::endl;
    std::cout << "is_same<int, int>: " << is_same<int, int>::value << std::endl;
    std::cout << "is_same<int, double>: " << is_same<int, double>::value << std::endl;
    std::cout << "is_pointer<int>: " << is_pointer<int>::value << std::endl;
    std::cout << "is_pointer<int*>: " << is_pointer<int*>::value << std::endl;
    std::cout << "is_pointer<std::string*>: " << is_pointer<std::string*>::value << std::endl;
    std::cin.get();
    return 0;
}
```

泛型编程示例-模板元编程-静态断言

```cpp
#include <iostream>
#include <type_traits>
// 1. 静态断言：编译期检查类型是否为整型
template<typename T>
void check_integral() {
    static_assert(std::is_integral<T>::value, "T 必须是整型类型");
}
// 2. 静态断言：编译期检查数组大小
template<typename T, std::size_t N>
void check_array_size(T (&)[N]) {
    static_assert(N > 3, "数组大小必须大于 3");
}
// 3. 静态断言：自定义条件
template<int N>
struct Factorial {
    static_assert(N >= 0, "N 必须为非负数");
    static constexpr int value = N * Factorial<N - 1>::value;
};
template<> struct Factorial<0> { static constexpr int value = 1; };
```

```cpp
int main() {
    // 检查类型
    check_integral<int>();
    // check_integral<double>(); // 取消注释将导致编译错误

    // 检查数组大小
    int arr[5] = {};
    check_array_size(arr);
    // int arr2[2] = {};
    // check_array_size(arr2); // 取消注释将导致编译错误

    // 编译期计算阶乘
    std::cout << "Factorial<5>::value = " << Factorial<5>::value << std::endl;
    // Factorial<-1> f; // 取消注释将导致编译错误

    return 0;
}
```

泛型编程示例-模板元编程-编译期常量计算

```cpp
#include <iostream>
// 1. 编译期计算阶乘（递归模板）
template<int N>
struct Factorial {
    static_assert(N >= 0, "N 必须为非负数");
    static constexpr int value = N * Factorial<N - 1>::value;
};
template<> struct Factorial<0> { static constexpr int value = 1; };
// 2. 编译期计算斐波那契数列（递归模板）
template<int N>
struct Fibonacci {
    static_assert(N >= 0, "N 必须为非负数");
    static constexpr int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};
template<> struct Fibonacci<1> { static constexpr int value = 1; };
template<> struct Fibonacci<0> { static constexpr int value = 0; };
```

```cpp
// 3. 编译期判断是否为 2 的幂
template<int N>
struct IsPowerOfTwo {
    static_assert(N > 0, "N 必须为正数");
    static constexpr bool value = (N & (N - 1)) == 0;
};
int main() {
    std::cout << "Factorial<5>::value = " << Factorial<5>::value << std::endl;
    std::cout << "Fibonacci<8>::value = " << Fibonacci<8>::value << std::endl;
    std::cout << "IsPowerOfTwo<16>::value = " << std::boolalpha << IsPowerOfTwo<16>::value << std::endl;
    std::cout << "IsPowerOfTwo<18>::value = " << std::boolalpha << IsPowerOfTwo<18>::value << std::endl;
    // 编译期常量可用于数组大小等
    int arr[Factorial<4>::value] = {0};
    std::cout << "arr size = " << sizeof(arr) / sizeof(arr[0]) << std::endl;
    return 0;
}
```

泛型编程示例-模板元编程-类型变换

```cpp
#include <iostream>
#include <type_traits>
#include <string>
// 1. 移除指针类型
template<typename T> struct remove_pointer { using type = T; };
template<typename T> struct remove_pointer<T*> { using type = T; };
// 2. 添加 const 修饰符
template<typename T> struct add_const { using type = const T; };
// 3. 移除引用类型
template<typename T> struct remove_reference { using type = T; };
template<typename T> struct remove_reference<T&> { using type = T; };
template<typename T> struct remove_reference<T&&> { using type = T; };
```

```cpp
int main() {
    std::cout << std::boolalpha;
    std::cout << "remove_pointer<int*>::type 是否为 int: "
              << std::is_same<remove_pointer<int*>::type, int>::value << std::endl;
    std::cout << "add_const<int>::type 是否为 const int: "
              << std::is_same<add_const<int>::type, const int>::value << std::endl;
    std::cout << "remove_reference<int&>::type 是否为 int: "
              << std::is_same<remove_reference<int&>::type, int>::value << std::endl;
    std::cout << "remove_reference<std::string&&>::type 是否为 std::string: "
              << std::is_same<remove_reference<std::string&&>::type, std::string>::value << std::endl;
    return 0;
}
```

泛型编程示例-模板元编程-编译期循环

```cpp
#include <iostream>

// 1. 编译期循环：计算数组所有元素之和（递归模板）
template<int N>
struct Sum {
    template<typename T> static T sum(const T* arr) { return arr[0] + Sum<N - 1>::sum(arr + 1); }
};
template<>
struct Sum<0> { template<typename T> static T sum(const T*) { return T{}; } };

// 2. 编译期循环：打印数组所有元素（递归模板）
template<int N>
struct PrintArray {
    template<typename T> static void print(const T* arr) { std::cout << arr[0] << " "; PrintArray<N - 1>::print(arr + 1); }
};
template<>
struct PrintArray<0> { template<typename T> static void print(const T*) {} };
```

```cpp
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    constexpr int N = sizeof(arr) / sizeof(arr[0]);
    std::cout << "数组元素: ";
    PrintArray<N>::print(arr);
    std::cout << "\n数组元素之和: " << Sum<N>::sum(arr) << std::endl;

    double darr[4] = {1.1, 2.2, 3.3, 4.4};
    constexpr int DN = sizeof(darr) / sizeof(darr[0]);
    std::cout << "double 数组元素: ";
    PrintArray<DN>::print(darr);
    std::cout << "\ndouble 数组元素之和: " << Sum<DN>::sum(darr) << std::endl;

    std::cin.get();
    return 0;
}
```

泛型编程示例-模板元编程-编译时多态

```cpp
#include <iostream>
#include <type_traits>

// 1. 编译时多态：根据类型选择不同实现（类型分支）
template<typename T>
struct TypePrinter {
    static void print() { std::cout << "未知类型" << std::endl; }
};

template<>
struct TypePrinter<int> {
    static void print() { std::cout << "类型为 int" << std::endl; }
};

template<>
struct TypePrinter<double> {
    static void print() { std::cout << "类型为 double" << std::endl; }
};

template<>
struct TypePrinter<const char*> {
    static void print() { std::cout << "类型为 const char*" << std::endl; }
};

// 2. 利用 SFINAE 实现编译时多态（判断是否为指针类型）
template<typename T>
typename std::enable_if<std::is_pointer<T>::value, void>::type
pointer_info(T) { std::cout << "是指针类型" << std::endl; }

template<typename T>
typename std::enable_if<!std::is_pointer<T>::value, void>::type
pointer_info(T) { std::cout << "不是指针类型" << std::endl; }
```

泛型编程示例-模板元编程-constexpr

```cpp
#include <iostream>

// 1. 使用 constexpr 递归计算阶乘
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}

// 2. 使用 constexpr 递归计算斐波那契数列
constexpr int fibonacci(int n) {
    return n <= 1 ? n : fibonacci(n - 1) +
    ↪  fibonacci(n - 2);
}

// 3. 使用 constexpr 判断是否为 2 的幂
constexpr bool is_power_of_two(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}
```

```cpp
int main() {
    constexpr int fact5 = factorial(5);
    constexpr int fib8 = fibonacci(8);
    constexpr bool pow2_16 = is_power_of_two(16);
    constexpr bool pow2_18 = is_power_of_two(18);

    std::cout << "factorial(5) = " << fact5 <<
    ↪  std::endl;
    std::cout << "fibonacci(8) = " << fib8 <<
    ↪  std::endl;
    std::cout << "is_power_of_two(16) = " <<
    ↪  std::boolalpha << pow2_16 << std::endl;
    std::cout << "is_power_of_two(18) = " <<
    ↪  std::boolalpha << pow2_18 << std::endl;

    // constexpr 结果可用于数组大小等
    int arr[factorial(4)] = {0};
    std::cout << "arr size = " << sizeof(arr) /
    ↪  sizeof(arr[0]) << std::endl;

    std::cin.get();
    return 0;
}
```

泛型编程示例-模板元编程-if constexpr

```cpp
#include <iostream>
#include <type_traits>
// 1. 使用 if constexpr 实现类型分支
template<typename T>
void print_type_info(const T& value) {
    if constexpr (std::is_integral<T>::value) {
        std::cout << "类型为整型，值 = " << value <<
        ↪ std::endl;
    } else if constexpr
    ↪ (std::is_floating_point<T>::value) {
        std::cout << "类型为浮点型，值 = " << value <<
        ↪ std::endl;
    } else if constexpr (std::is_pointer<T>::value) {
        std::cout << "类型为指针，地址 = " <<
        ↪ static_cast<const void*>(value) <<
        ↪ std::endl;
    } else {
        std::cout << "未知类型" << std::endl;
    }
}
// 2. 使用 if constexpr 实现编译期递归
template<int N>
constexpr int factorial() {
    if constexpr (N <= 1) { return 1; }
    else { return N * factorial<N - 1>(); }
}
int main() {
    int a = 42;
    double d = 3.14;
    int* p = &a;
    print_type_info(a);   // 整型
    print_type_info(d);   // 浮点型
    print_type_info(p);   // 指针
    print_type_info("hello"); // 未知类型
    constexpr int fact5 = factorial<5>();
    std::cout << "factorial<5>() = " << fact5 <<
    ↪ std::endl;
    std::cin.get();
    return 0;
}
```

泛型编程示例-模板元编程-std::integral_constant

```cpp
#include <iostream>
#include <type_traits>
// 1. 使用 std::integral_constant 定义编译期常量
using Five = std::integral_constant<int, 5>;
using TrueType = std::integral_constant<bool, true>;
// 2. 编译期计算阶乘（递归模板，继承自 std::integral_constant）
template<int N>
struct Factorial : std::integral_constant<int, N * Factorial<N - 1>::value> {};
template<>
struct Factorial<0> : std::integral_constant<int, 1> {};
// 3. 编译期判断是否为偶数
template<int N>
struct IsEven : std::integral_constant<bool, (N % 2 == 0)> {};
int main() {
    // std::integral_constant 的基本用法
    std::cout << "Five::value = " << Five::value << std::endl;
    std::cout << "TrueType::value = " << std::boolalpha << TrueType::value << std::endl;
    // Factorial
    std::cout << "Factorial<5>::value = " << Factorial<5>::value << std::endl;
    std::cout << "Factorial<0>::value = " << Factorial<0>::value << std::endl;
    // IsEven
    std::cout << "IsEven<4>::value = " << std::boolalpha << IsEven<4>::value << std::endl;
    std::cout << "IsEven<7>::value = " << std::boolalpha << IsEven<7>::value << std::endl;
    // std::integral_constant 可用于类型萃取
    if constexpr (IsEven<10>::value) {
        std::cout << "10 是偶数" << std::endl;
    } else {
        std::cout << "10 是奇数" << std::endl;
    }
    std::cin.get();
    return 0;
}
```

泛型编程小结

- 泛型编程（Generic Programming）是一种以抽象和复用为核心思想的高级编程范式，强调算法与数据结构的分离，使代码能够适用于多种类型。
- C++ 通过模板（Template）机制实现泛型编程，支持函数模板、类模板、变量模板，以及模板特化、偏特化等高级特性。
- 模板不仅支持类型参数，还可结合非类型模板参数、模板模板参数，实现更高层次的抽象与灵活性。
- 泛型编程推动了 C++ 标准库（如 STL 容器、算法、迭代器等）的设计，极大提升了代码复用性和可维护性。
- 现代 C++ 泛型编程还包括模板元编程（Template Metaprogramming）、类型萃取（Type Traits）、SFINAE、constexpr 等技术，实现编译期计算与类型推导，进一步提升类型安全与性能。
- 泛型编程的优势：**代码复用**、**类型安全**、**零开销抽象**、**性能优化**，但也带来编译错误复杂、可读性下降等挑战。

泛型编程的典型应用场景

- **泛型排序算法**：如 `std::sort`、HeapSort、QuickSort 等，支持任意可比较类型。
- **泛型容器与数据结构**：如 `std::vector`、`std::map`、`std::set` 等，适用于多种类型的数据存储与管理。
- **泛型数值与科学计算**：如矩阵运算、数值积分、线性代数库等，支持不同数值类型。
- **泛型算法库**：如 STL 算法（查找、变换、归约等），可作用于任意容器和类型。
- **泛型 AI 与优化算法**：如神经网络、支持向量机、遗传算法、粒子群优化等，算法结构与数据类型解耦。
- **泛型图像与信号处理**：如图像滤波、特征提取、信号变换等，支持不同像素/信号类型。

目录

函数式编程概述

什么是函数式编程

- 函数式编程（Functional Programming）是一种编程范式，它将计算视为数学函数的求值过程。
- 函数式编程的代码通常是不可变的（Immutable），避免副作用（Side Effect），并且函数本身也是第一类对象。
- 函数式编程强调函数组合（Function Composition）和纯函数（Pure Function），避免状态变化和可变共享状态。

函数式编程特点

- 函数是一等公民
- 不可变性（Immutability）
- 无副作用
- 高阶函数
- 递归和组合

C++ 中的函数式特性

- Lambda 表达式
- std::function
- 函数对象
- 算法库
- 智能指针

函数式编程示例-纯函数

```cpp
#include <iostream>

// 纯函数：只依赖输入参数，无副作用
int add(int a, int b) {
    return a + b;
}

// 纯函数：计算平方
int square(int x) {
    return x * x;
}
```

```cpp
// 纯函数：返回较大值
int max_value(int a, int b) {
    return (a > b) ? a : b;
}

int main() {
    int x = 3, y = 5;
    std::cout << "add(3, 5) = " << add(x, y) <<
    ↪  std::endl;
    std::cout << "square(4) = " << square(4) <<
    ↪  std::endl;
    std::cout << "max_value(7, 2) = " << max_value(7,
    ↪  2) << std::endl;
    std::cin.get();
    return 0;
}
```

函数式编程示例-Lambda 表达式

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
int main() {
    // 1. 最基本的 Lambda 表达式
    auto add = [](int a, int b) { return a + b; };
    std::cout << "add(2, 3) = " << add(2, 3) <<
    ↪  std::endl;
    // 2. Lambda 作为 std::function 参数
    std::function<int(int, int)> mul = [](int a, int
    ↪  b) { return a * b; };
    std::cout << "mul(4, 5) = " << mul(4, 5) <<
    ↪  std::endl;
    // 3. Lambda 用于 STL 算法
    std::vector<int> v = {1, 2, 3, 4, 5};
    std::cout << "原始数据: ";
    for (auto x : v) std::cout << x << " ";
    std::cout << std::endl;
    // 使用 Lambda 对每个元素加 1
    std::for_each(v.begin(), v.end(), [](int &x) { x
    ↪  += 1; });
    std::cout << "每个元素加 1 后: ";
    for (auto x : v) std::cout << x << " ";
    std::cout << std::endl;
    // 4. 带值捕获的 Lambda
    int factor = 10;
    auto multiply = [factor](int x) { return x *
    ↪  factor; };
    std::cout << "multiply(6) = " << multiply(6) <<
    ↪  std::endl;
    // 5. Lambda 作为排序谓词
    std::sort(v.begin(), v.end(), [](int a, int b) {
    ↪  return a > b; });
    std::cout << "降序排序后: ";
    for (auto x : v) std::cout << x << " ";
    std::cout << std::endl;
    // 6. 引用捕获示例
    int sum = 0;
    std::for_each(v.begin(), v.end(), [&sum](int x) {
    ↪  sum += x; });
    std::cout << "所有元素之和（引用捕获 sum）: " <<
    ↪  sum << std::endl;
    std::cin.get();
    return 0;
}
```

函数式编程示例-高阶函数-函数作为参数

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

// 高阶函数：接受函数作为参数，对 vector 中的每个元素应用该函数
void apply_to_each(std::vector<int>& v, const std::function<void(int&)>& func) {
    for (auto& x : v) { func(x); }
}

// 另一个高阶函数：接受函数作为参数，返回所有元素处理后的新 vector
std::vector<int> map_vector(const std::vector<int>& v, const std::function<int(int)>& func) {
    std::vector<int> result;
    result.reserve(v.size());
    for (auto x : v) { result.push_back(func(x)); }
    return result;
}
```

```cpp
int main() {
    std::vector<int> data = {1, 2, 3, 4, 5};
    // 1. 传递 Lambda 表达式作为参数，对每个元素加 10
    apply_to_each(data, [](int& x) { x += 10; });
    std::cout << "每个元素加 10 后: ";
    for (auto x : data) std::cout << x << " ";
    std::cout << std::endl;
    // 2. 使用 map_vector，将每个元素平方，返回新 vector
    auto squared = map_vector(data, [](int x) { return x * x; });
    std::cout << "每个元素平方后: ";
    for (auto x : squared) std::cout << x << " ";
    std::cout << std::endl;
    // 3. 也可以传递普通函数指针
    auto increment = [](int& x) { x++; };
    apply_to_each(data, increment);
    std::cout << "每个元素再加 1 后: ";
    for (auto x : data) std::cout << x << " ";
    std::cout << std::endl;
    std::cin.get();
    return 0;
}
```

函数式编程示例-高阶函数-函数作为返回值

```cpp
#include <iostream>
#include <functional>

// 高阶函数：返回一个 Lambda（函数）作为返回值
std::function<int(int)> make_multiplier(int factor) {
    // 返回一个捕获 factor 的 Lambda
    return [factor](int x) {
        return x * factor;
    };
}

// 另一个示例：返回一个加法器
auto make_adder(int base) {
    return [base](int x) {
        return x + base;
    };
}
```

```cpp
int main() {
    // 1. 生成一个乘以 5 的函数
    auto times5 = make_multiplier(5);
    std::cout << "times5(3) = " << times5(3) <<
    ↪  std::endl; // 15

    // 2. 生成一个加 10 的函数
    auto add10 = make_adder(10);
    std::cout << "add10(7) = " << add10(7) <<
    ↪  std::endl; // 17

    // 3. 组合使用
    std::cout << "times5(add10(2)) = " <<
    ↪  times5(add10(2)) << std::endl; // 60

    std::cin.get();
    return 0;
}
```

函数式编程示例-递归与组合（一）

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
// 递归实现：阶乘
int factorial(int n) { return n <= 1 ? 1 : n * factorial(n - 1); }
// 递归实现：斐波那契数列
int fibonacci(int n) { return n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2); }
// 递归实现：vector 求和
int sum_recursive(const std::vector<int>& v, size_t idx = 0) {
    if (idx >= v.size()) return 0;
    return v[idx] + sum_recursive(v, idx + 1);
}
// 组合式：map（对 vector 每个元素应用函数，返回新 vector）
std::vector<int> map_vector(const std::vector<int>& v, const std::function<int(int)>& func) {
    std::vector<int> result;
    result.reserve(v.size());
    for (auto x : v) { result.push_back(func(x)); }
    return result;
}
// 组合式：filter（筛选满足条件的元素）
std::vector<int> filter_vector(const std::vector<int>& v, const std::function<bool(int)>& pred) {
    std::vector<int> result;
    for (auto x : v) { if (pred(x)) result.push_back(x); }
    return result;
}
// 组合式：reduce（归约/折叠）
int reduce_vector(const std::vector<int>& v, int init, const std::function<int(int, int)>& op) {
    int result = init;
    for (auto x : v) { result = op(result, x); }
    return result;
}
```

函数式编程示例-递归与组合（二）

```cpp
int main() {
    // 递归示例
    std::cout << "factorial(5) = " << factorial(5) << std::endl;
    std::cout << "fibonacci(8) = " << fibonacci(8) << std::endl;
    std::vector<int> data = {1, 2, 3, 4, 5};
    std::cout << "sum_recursive = " << sum_recursive(data) << std::endl;
    // map: 所有元素平方
    auto squared = map_vector(data, [](int x) { return x * x; });
    std::cout << "map_vector 平方: ";
    for (auto x : squared) std::cout << x << " ";
    std::cout << std::endl;
    // filter: 只保留偶数
    auto evens = filter_vector(data, [](int x) { return x % 2 == 0; });
    std::cout << "filter_vector 偶数: ";
    for (auto x : evens) std::cout << x << " ";
    std::cout << std::endl;
    // reduce: 求和
    int sum = reduce_vector(data, 0, [](int a, int b) { return a + b; });
    std::cout << "reduce_vector 求和: " << sum << std::endl;
    // 组合使用: 先平方再求和
    int sum_of_squares = reduce_vector(
        map_vector(data, [](int x) { return x * x; }),
        0,
        [](int a, int b) { return a + b; }
    );
    std::cout << "先平方再求和: " << sum_of_squares << std::endl;
    std::cin.get();
    return 0;
}
```

函数式编程示例-递归与组合（三）

```cpp
#include <iostream>
#include <functional>
// 函数组合器：compose(f, g) 返回一个新函数 h(x) = f(g(x))
template<typename F, typename G>
auto compose(F f, G g) {
    return [=](auto x) {
        return f(g(x));
    };
}
// 支持多函数组合（从右到左）
template<typename F, typename... Rest>
auto compose(F f, Rest... rest) {
    return [=](auto x) {
        return f(compose(rest...)(x));
    };
}
```

```cpp
int main() {
    // 定义两个简单函数
    auto add2 = [](int x) { return x + 2; };
    auto mul3 = [](int x) { return x * 3; };
    // 组合：先加 2 再乘 3
    auto add2_then_mul3 = compose(mul3, add2);
    std::cout << "add2_then_mul3(5) = " << add2_then_mul3(5) << std::endl; // (5+2)*3=21
    // 组合：先乘 3 再加 2
    auto mul3_then_add2 = compose(add2, mul3);
    std::cout << "mul3_then_add2(5) = " << mul3_then_add2(5) << std::endl; // (5*3)+2=17
    // 多函数组合：((x+1)*2)-3
    auto add1 = [](int x) { return x + 1; };
    auto mul2 = [](int x) { return x * 2; };
    auto sub3 = [](int x) { return x - 3; };
    auto complex = compose(sub3, mul2, add1);
    std::cout << "complex(4) = " << complex(4) << std::endl; // ((4+1)*2)-3=7
    std::cin.get();
    return 0;
}
```

函数式编程示例-函数对象 (Functors)-函数作为对象

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
// 定义一个函数对象（仿函数）：加法器
struct Adder {
    int operator()(int a, int b) const { return a + b; }
};
// 定义一个函数对象：判断是否为偶数
struct IsEven {
    bool operator()(int x) const { return x % 2 == 0; }
};
// 定义一个函数对象：对 vector 中每个元素加上指定值
struct AddN {
    int n;
    AddN(int n_) : n(n_) {}
    void operator()(int& x) const { x += n; }
};
```

```cpp
int main() {
    // 1. 函数对象作为可调用对象
    Adder add;
    std::cout << "Adder(3, 4) = " << add(3, 4) << std::endl;
    // 2. 函数对象用于 STL 算法
    std::vector<int> v = {1, 2, 3, 4, 5};
    // 使用 AddN 对每个元素加 10
    std::for_each(v.begin(), v.end(), AddN(10));
    std::cout << "每个元素加 10 后: ";
    for (auto x : v) std::cout << x << " ";
    std::cout << std::endl;
    // 使用 IsEven 筛选偶数
    std::cout << "偶数元素: ";
    std::for_each(v.begin(), v.end(), [](int x){
        if (IsEven{}(x)) std::cout << x << " ";
    });
    std::cout << std::endl;
    std::cin.get();
    return 0;
}
```

函数式编程示例-标准库算法（一）

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
#include <iterator>
#include <random>
#include <set>

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5};
    // 1. std::for_each: 对每个元素加 1
    std::for_each(data.begin(), data.end(), [](int& x) { x += 1; });
    std::cout << "for_each 每个元素加 1 后: ";
    for (auto x : data) std::cout << x << " ";
    std::cout << std::endl;

    // 2. std::transform: 所有元素平方，结果存入新 vector
    std::vector<int> squared(data.size());
    std::transform(data.begin(), data.end(), squared.begin(), [](int x) { return x * x; });
    std::cout << "transform 平方后: ";
    for (auto x : squared) std::cout << x << " ";
    std::cout << std::endl;

    // 3. std::count_if: 统计偶数个数
    int even_count = std::count_if(data.begin(), data.end(), [](int x) { return x % 2 == 0; });
    std::cout << "count_if 偶数个数: " << even_count << std::endl;

    // 4. std::accumulate: 求和
    int sum = std::accumulate(data.begin(), data.end(), 0);
    std::cout << "accumulate 求和: " << sum << std::endl;
```

函数式编程示例-标准库算法（二）

```
    // 5. std::find_if: 查找第一个大于 3 的元素
    auto it = std::find_if(data.begin(), data.end(),
    ↪  [](int x) { return x > 3; });
    if (it != data.end()) {
        std::cout << "find_if 第一个大于 3 的元素: "
        ↪  << *it << std::endl;
    } else {
        std::cout << "没有找到大于 3 的元素" <<
        ↪  std::endl;
    }

    // 6. std::all_of / any_of / none_of
    bool all_positive = std::all_of(data.begin(),
    ↪  data.end(), [](int x) { return x > 0; });
    bool any_even = std::any_of(data.begin(),
    ↪  data.end(), [](int x) { return x % 2 == 0;
    ↪  });
    bool none_negative = std::none_of(data.begin(),
    ↪  data.end(), [](int x) { return x < 0; });
    std::cout << "all_of 全为正数: " << std::boolalpha
    ↪  << all_positive << std::endl;
    std::cout << "any_of 存在偶数: " << std::boolalpha
    ↪  << any_even << std::endl;
    std::cout << "none_of 没有负数: " <<
    ↪  std::boolalpha << none_negative << std::endl;
```

```
    // 7. std::remove_if + erase: 移除所有小于 3 的元素
    auto new_end = std::remove_if(data.begin(),
    ↪  data.end(), [](int x) { return x < 3; });
    data.erase(new_end, data.end());
    std::cout << "remove_if 移除小于 3 后: ";
    for (auto x : data) std::cout << x << " ";
    std::cout << std::endl;

    // 8. std::reverse: 反转 vector
    std::reverse(data.begin(), data.end());
    std::cout << "reverse 反转后: ";
    for (auto x : data) std::cout << x << " ";
    std::cout << std::endl;
```

函数式编程示例-标准库算法（三）

```cpp
    // 9. std::unique：去重（需先排序）
    std::vector<int> dup = {1, 2, 2, 3, 3, 3, 4, 5, 5};
    std::sort(dup.begin(), dup.end());
    auto last = std::unique(dup.begin(), dup.end());
    dup.erase(last, dup.end());
    std::cout << "unique 去重后：";
    for (auto x : dup) std::cout << x << " ";
    std::cout << std::endl;

    // 10. std::partition：分区（偶数在前，奇数在后）
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8};
    std::partition(nums.begin(), nums.end(), [](int x) { return x % 2 == 0; });
    std::cout << "partition 偶数在前：";
    for (auto x : nums) std::cout << x << " ";
    std::cout << std::endl;
```

```cpp
    // 11. std::min_element / std::max_element
    auto min_it = std::min_element(nums.begin(), nums.end());
    auto max_it = std::max_element(nums.begin(), nums.end());
    if (min_it != nums.end() && max_it != nums.end()) {
        std::cout << "min_element 最小值：" << *min_it << std::endl;
        std::cout << "max_element 最大值：" << *max_it << std::endl;
    }

    // 12. std::copy_if：复制所有大于 4 的元素到新 vector
    std::vector<int> greater_than_4;
    std::copy_if(nums.begin(), nums.end(), std::back_inserter(greater_than_4), [](int x) { return x > 4; });
    std::cout << "copy_if 大于 4 的元素：";
    for (auto x : greater_than_4) std::cout << x << " ";
    std::cout << std::endl;
```

函数式编程示例-标准库算法（四）

```
    // 13. std::set_union: 求两个集合的并集
    std::vector<int> a = {1, 2, 3, 5, 7};
    std::vector<int> b = {2, 3, 4, 6, 8};
    std::vector<int> union_result;
    std::set_union(a.begin(), a.end(), b.begin(),
    ↪  b.end(), std::back_inserter(union_result));
    std::cout << "set_union 并集: ";
    for (auto x : union_result) std::cout << x << "
    ↪  ";
    std::cout << std::endl;

    // 14. std::set_intersection: 求交集
    std::vector<int> intersection_result;
    std::set_intersection(a.begin(), a.end(),
    ↪  b.begin(), b.end(),
    ↪  std::back_inserter(intersection_result));
    std::cout << "set_intersection 交集: ";
    for (auto x : intersection_result) std::cout << x
    ↪  << " ";
    std::cout << std::endl;
```

```
    // 15. std::set_difference: 求差集
    std::vector<int> diff_result;
    std::set_difference(a.begin(), a.end(),
    ↪  b.begin(), b.end(),
    ↪  std::back_inserter(diff_result));
    std::cout << "set_difference 差集 (a-b): ";
    for (auto x : diff_result) std::cout << x << " ";
    std::cout << std::endl;

    // 16. std::generate: 生成序列
    std::vector<int> seq(10);
    int v = 1;
    std::generate(seq.begin(), seq.end(), [&v]() {
    ↪  return v++; });
    std::cout << "generate 生成序列: ";
    for (auto x : seq) std::cout << x << " ";
    std::cout << std::endl;
```

函数式编程示例-标准库算法（五）

```cpp
    // 17. std::shuffle: 随机打乱
    std::random_device rd;
    std::mt19937 g(rd());
    std::shuffle(seq.begin(), seq.end(), g);
    std::cout << "shuffle 打乱后: ";
    for (auto x : seq) std::cout << x << " ";
    std::cout << std::endl;

    // 18. std::fill: 填充
    std::fill(seq.begin(), seq.end(), 7);
    std::cout << "fill 填充为 7: ";
    for (auto x : seq) std::cout << x << " ";
    std::cout << std::endl;

    // 19. std::replace_if: 条件替换
    std::vector<int> rep = {1, 2, 3, 4, 5, 6};
    std::replace_if(rep.begin(), rep.end(), [](int x)
    ↪  { return x % 2 == 0; }, 0);
    std::cout << "replace_if 偶数替换为 0: ";
    for (auto x : rep) std::cout << x << " ";
    std::cout << std::endl;
```

```cpp
    // 20. std::merge: 合并有序序列
    std::vector<int> m1 = {1, 3, 5};
    std::vector<int> m2 = {2, 4, 6};
    std::vector<int> merged;
    std::merge(m1.begin(), m1.end(), m2.begin(),
    ↪  m2.end(), std::back_inserter(merged));
    std::cout << "merge 合并有序序列: ";
    for (auto x : merged) std::cout << x << " ";
    std::cout << std::endl;

    // 21. std::lower_bound / std::upper_bound
    std::vector<int> sorted = {1, 2, 4, 4, 5, 7};
    auto lb = std::lower_bound(sorted.begin(),
    ↪  sorted.end(), 4);
    auto ub = std::upper_bound(sorted.begin(),
    ↪  sorted.end(), 4);
    std::cout << "lower_bound(4) 位置: " << (lb -
    ↪  sorted.begin()) << std::endl;
    std::cout << "upper_bound(4) 位置: " << (ub -
    ↪  sorted.begin()) << std::endl;
```

函数式编程示例-标准库算法（六）

```cpp
    // 22. std::equal: 判断两个序列是否相等
    std::vector<int> eq1 = {1, 2, 3};
    std::vector<int> eq2 = {1, 2, 3};
    std::cout << "equal 两个序列是否相等: " <<
    ↪  std::boolalpha << std::equal(eq1.begin(),
    ↪  eq1.end(), eq2.begin()) << std::endl;

    // 23. std::find_end: 查找子序列最后一次出现的位置
    std::vector<int> big = {1, 2, 3, 2, 3, 4, 2, 3};
    std::vector<int> sub = {2, 3};
    auto end_pos = std::find_end(big.begin(),
    ↪  big.end(), sub.begin(), sub.end());
    if (end_pos != big.end()) {
        std::cout << "find_end 子序列最后一次出现的位
        ↪  置: " << (end_pos - big.begin()) <<
        ↪  std::endl;
    } else {
        std::cout << "find_end 未找到子序列" <<
        ↪  std::endl;
    }
```

```cpp
    // 24. std::adjacent_find: 查找相邻重复元素
    std::vector<int> adj = {1, 2, 2, 3, 4, 4, 4, 5};
    auto adj_it = std::adjacent_find(adj.begin(),
    ↪  adj.end());
    if (adj_it != adj.end()) {
        std::cout << "adjacent_find 第一个相邻重复元
        ↪  素: " << *adj_it << std::endl;
    } else {
        std::cout << "adjacent_find 未找到相邻重复元
        ↪  素" << std::endl;
    }

    // 25. std::inplace_merge: 原地合并有序区间
    std::vector<int> im = {1, 3, 5, 2, 4, 6};
    std::inplace_merge(im.begin(), im.begin() + 3,
    ↪  im.end());
    std::cout << "inplace_merge 原地合并后: ";
    for (auto x : im) std::cout << x << " ";
    std::cout << std::endl;
```

函数式编程示例-标准库算法（七）

```cpp
    // 26. std::nth_element: 部分排序
    std::vector<int> nth = {7, 2, 5, 3, 9, 1};
    std::nth_element(nth.begin(), nth.begin() + 2,
    ↪  nth.end());
    std::cout << "nth_element 第 3 小的元素: " <<
    ↪  nth[2] << std::endl;
    std::cout << "nth_element 后前 3 小: ";
    for (int i = 0; i < 3; ++i) std::cout << nth[i]
    ↪  << " ";
    std::cout << std::endl;

    // 27. std::is_sorted / std::is_sorted_until
    std::vector<int> isort = {1, 2, 3, 5, 4, 6};
    std::cout << "is_sorted 是否有序: " <<
    ↪  std::boolalpha <<
    ↪  std::is_sorted(isort.begin(), isort.end()) <<
    ↪  std::endl;
    auto until = std::is_sorted_until(isort.begin(),
    ↪  isort.end());
    std::cout << "is_sorted_until 第一个无序位置: " <<
    ↪  (until - isort.begin()) << std::endl;
```

```cpp
    // 28. std::for_each_n: 对前 n 个元素操作（C++17）
#if __cplusplus >= 201703L
    std::vector<int> nvec = {1, 2, 3, 4, 5};
    std::for_each_n(nvec.begin(), 3, [](int& x) { x
    ↪  *= 10; });
    std::cout << "for_each_n 前 3 个元素乘 10: ";
    for (auto x : nvec) std::cout << x << " ";
    std::cout << std::endl;
#endif
```

函数式编程示例-标准库算法（八）

```
    // 29. std::sample: 从序列中随机采样（C++17）
#if __cplusplus >= 201703L
    std::vector<int> pop = {1, 2, 3, 4, 5, 6, 7, 8,
    ↪  9, 10};
    std::vector<int> samp;
    std::sample(pop.begin(), pop.end(),
    ↪  std::back_inserter(samp), 4, g);
    std::cout << "sample 随机采样 4 个元素: ";
    for (auto x : samp) std::cout << x << " ";
    std::cout << std::endl;
#endif
```

```
    // 30. std::includes: 判断一个有序序列是否包含另一个
    ↪  有序序列
    std::vector<int> inc1 = {1, 2, 3, 4, 5, 6};
    std::vector<int> inc2 = {2, 4, 6};
    std::cout << "includes inc1 是否包含 inc2: " <<
    ↪  std::boolalpha
              << std::includes(inc1.begin(),
              ↪  inc1.end(), inc2.begin(),
              ↪  inc2.end()) << std::endl;

    std::cin.get();
    return 0;
}
```

函数式编程示例-模式匹配

```cpp
#include <iostream>
#include <variant>
#include <string>
// 定义一个简单的代数数据类型（ADT）：Shape
struct Circle { double radius; };
struct Rectangle { double width, height; };
// 使用 std::variant 表示 Shape 的“和类型”
using Shape = std::variant<Circle, Rectangle>;
// C++17 及以上：实现模式匹配工具
template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;
double area(const Shape& shape) {
    // 使用 std::visit 进行“模式匹配”
    return std::visit(overloaded{
        [](const Circle& c) { return 3.14159 * c.radius * c.radius; },
        [](const Rectangle& r) { return r.width * r.height; }
    }, shape);
}
```

```cpp
std::string shape_name(const Shape& shape) {
    return std::visit(overloaded{
        [](const Circle&) { return std::string("Circle"); },
        [](const Rectangle&) { return std::string("Rectangle"); }
    }, shape);
}
int main() {
    Shape s1 = Circle{2.0};
    Shape s2 = Rectangle{3.0, 4.0};
    for (const auto& s : {s1, s2}) {
        std::cout << shape_name(s) << " 的面积: " << area(s) << std::endl;
    }
    std::cin.get();
    return 0;
}
```

函数式编程示例-惰性求值

```cpp
#include <iostream>
#include <functional>
#include <vector>
// 简单的惰性求值类
template<typename T>
class LazyValue {
    std::function<T()> thunk;
    mutable bool evaluated = false;
    mutable T value;
public:
    LazyValue(std::function<T()> f) : thunk(f) {}
    T get() const {
        if (!evaluated) {
            value = thunk();
            evaluated = true;
        }
        return value;
    }
};
int main() {
    int x = 10, y = 20;
    LazyValue<int> lazy_sum([=] { std::cout << "计算 x + y\n"; return x + y; });
    std::cout << "未计算 lazy_sum\n";
    std::cout << "lazy_sum = " << lazy_sum.get() << std::endl;
    std::cout << "再次获取 lazy_sum = " << lazy_sum.get() << std::endl;
    int n = 8;
    std::vector<LazyValue<int>> fibs;
    fibs.emplace_back([] { return 0; });
    fibs.emplace_back([] { return 1; });
    for (int i = 2; i < n; ++i)
        fibs.emplace_back([i, &fibs] { return fibs[i-1].get() + fibs[i-2].get(); });
    std::cout << "斐波那契数列前" << n << "项: ";
    for (int i = 0; i < n; ++i)
        std::cout << fibs[i].get() << " ";
    std::cout << std::endl;
    return 0;
}
```

函数式编程小结

- 函数式编程是一种以数学函数为核心思想的编程范式，将计算过程抽象为函数的组合与变换。
- 代码强调不可变性（Immutable），尽量避免副作用（Side Effect），函数作为“第一类对象”可以像变量一样传递和返回。
- 推崇纯函数（Pure Function）和函数组合（Function Composition），减少状态变化和共享可变状态，使程序更易于推理和测试。
- 常见特性包括高阶函数、递归、惰性求值、模式匹配等，提升代码的表达力和可维护性。

典型应用场景

- 数学计算和科学计算
- 数据处理和转换
- 并行和并发编程
- 事件驱动编程
- 状态机和工作流
- 编译器和解释器
- 函数组合和管道
- 测试和验证

目录

事件驱动编程

什么是事件驱动编程

- 事件驱动编程（Event-Driven Programming）是一种编程范式，它将程序的执行过程看作是事件的响应和处理。
- 事件驱动编程的代码通常是异步的（Asynchronous），事件的触发和处理是分离的。
- 事件驱动编程强调事件的响应和处理，而不是按照顺序执行。

事件驱动编程特点

- 以事件为中心
- 异步处理
- 松耦合设计
- 响应式编程
- 用户交互驱动

Qt 中的事件驱动

- 信号槽机制
- 事件循环
- 事件过滤器
- 定时器事件

事件驱动编程示例

```cpp
#include <iostream>
#include <functional>
#include <map>
#include <string>
// 简单的事件驱动框架
class EventEmitter {
public:
    using Handler = std::function<void()>;
    // 注册事件处理器
    void on(const std::string& event, Handler handler) { handlers[event] = handler; }
    // 触发事件
    void emit(const std::string& event) {
        if (handlers.count(event)) { handlers[event](); }
        else { std::cout << "未找到事件处理器: " << event << std::endl; }
    }
private:
    std::map<std::string, Handler> handlers;
};
```

```cpp
int main() {
    EventEmitter emitter;
    // 注册事件及其处理器
    emitter.on("start", []() { std::cout << "程序启动事件被触发！" << std::endl; });
    emitter.on("click", []() { std::cout << "按钮点击事件被触发！" << std::endl; });
    emitter.on("exit", []() { std::cout << "程序退出事件被触发！" << std::endl; });
    // 模拟事件触发
    emitter.emit("start");
    emitter.emit("click");
    emitter.emit("exit");
    emitter.emit("unknown"); // 未注册事件
    std::cin.get();
    return 0;
}
```

声明式编程

什么是声明式编程

- 声明式编程（Declarative Programming）是一种编程范式，它将程序的执行过程看作是声明的执行。
- 声明式编程的代码通常是声明式的，而不是命令式的。
- 声明式编程强调声明的执行，而不是按照顺序执行。

声明式编程特点

- 描述” 做什么” 而非” 怎么做”
- 关注结果而非过程
- 更接近自然语言
- 减少副作用
- 提高可读性

Qt 中的声明式编程

- QML 语言
- 属性绑定
- 状态机
- 样式表

声明式编程示例

```cpp
#include <iostream>
#include <functional>
// Property 类用于实现属性的声明式绑定，支持属性值变化时自动通知回调
template<typename T>
class Property {
public:
    using Callback = std::function<void(const T&)>; // 属性变化时的回调类型
    Property(T v) : value(v) {}
    void set(T v) {
        if (value != v) {
            value = v;
            if (cb) cb(value);
        }
    }
    T get() const { return value; }
    void bind(Callback f) { cb = f; }
private:
    T value;      // 属性值
    Callback cb;  // 属性变化时的回调
};
```

```cpp
int main() {
    // 定义三个属性：width、height、area
    Property<int> width(100), height(50), area(0);
    // 定义一个函数，当 width 或 height 变化时自动更新 area
    auto updateArea = [&]() { area.set(width.get() * height.get()); };
    width.bind([&](int){ updateArea(); });
    height.bind([&](int){ updateArea(); });
    // 初始化时先计算一次 area
    updateArea();
    // 输出初始值
    std::cout << width.get() << " " << height.get() << " " << area.get() << std::endl;
    // 修改 width/height，area 会自动更新
    width.set(200); // height.set(80);
    std::cout << width.get() << " " << height.get() << " " << area.get() << std::endl;
    return 0;
```

组件式编程

什么是组件式编程

- 组件式编程（Component-Based Programming）是一种编程范式，它将程序的执行过程看作是组件的组合和交互。
- 组件式编程的代码通常是模块化的，组件之间是松耦合的。
- 组件式编程强调组件的复用和组合，而不是按照顺序执行。

组件式编程特点

- 模块化设计
- 可重用组件
- 松耦合架构
- 标准化接口
- 组合优于继承

Qt 中的组件系统

- Qt Widgets
- Qt Quick Components
- 插件系统
- 自定义组件

组件式编程示例-组件接口与实现

```cpp
#include <iostream>
#include <string>
// 组件接口
class IComponent {
public:
    virtual ~IComponent() = default;
    virtual void process() = 0;
    virtual std::string name() const = 0;
};
// 具体组件 A
class ComponentA : public IComponent {
public:
    void process() override { std::cout <<
    ↪  "ComponentA 正在处理任务" << std::endl; }
    std::string name() const override { return
    ↪  "ComponentA"; }
};
// 具体组件 B
class ComponentB : public IComponent {
public:
    void process() override { std::cout <<
    ↪  "ComponentB 正在处理任务" << std::endl; }
    std::string name() const override { return
    ↪  "ComponentB"; }
};
int main() {
    IComponent* comp1 = new ComponentA();
    IComponent* comp2 = new ComponentB();
    std::cout << comp1->name() << std::endl;
    comp1->process();
    std::cout << comp2->name() << std::endl;
    comp2->process();
    delete comp1;
    delete comp2;
    std::cin.get();
    return 0;
}
```

组件式编程示例-组件管理器

```cpp
// 组件管理器
class ComponentManager {
public:
    void registerComponent(const std::shared_ptr<IComponent>& comp) {
        components_[comp->name()] = comp;
    }
    std::shared_ptr<IComponent> getComponent(const std::string& name) const {
        auto it = components_.find(name);
        return it != components_.end() ? it->second : nullptr;
    }
    void processAll() {
        for (auto& [n, c] : components_) {
            std::cout << "[" << n << "] ";
            c->process();
        }
    }
private:
    std::unordered_map<std::string, std::shared_ptr<IComponent>> components_;
};

int main() {
    ComponentManager manager;

    manager.registerComponent(std::make_shared<ComponentA>(

    manager.registerComponent(std::make_shared<ComponentB>(
    manager.processAll();
    auto comp = manager.getComponent("ComponentA");
    if (comp) {
        std::cout << "单独处理: " << comp->name() << std::endl;
        comp->process();
    }
    return 0;
}
```

组件式编程示例-Qt 组件系统

```
// Qt 组件基类（QObject 派生）
class MyComponent : public QObject {
    Q_OBJECT
public:
    MyComponent(const QString& name, QObject* parent
    ↪  = nullptr)
        : QObject(parent), m_name(name) {}
    QString name() const { return m_name; }
public slots:
    void process() { qDebug() << m_name << "正在处理任
    ↪  务"; }
private:
    QString m_name;
};
// 组件管理器，负责管理和组合多个 Qt 组件
class QtComponentManager : public QObject {
    Q_OBJECT
public:
    void addComponent(MyComponent* comp) {
    ↪  m_components.append(comp); }
    void processAll() {
        for (auto comp : m_components) {
            comp->process();
        }
    }
private:
    QList<MyComponent*> m_components;
};
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    MyComponent compA("QtComponentA");
    MyComponent compB("QtComponentB");
    QtComponentManager manager;
    manager.addComponent(&compA);
    manager.addComponent(&compB);
    manager.processAll();
    std::cin.get();
    return 0;
}
```

目录

总结

本章要点

- 理解各种编程范式的特点和应用
- 掌握过程式编程的基本方法
- 学会面向对象编程的设计模式
- 理解泛型编程的优势和使用
- 掌握函数式编程的核心概念
- 学会事件驱动编程的实现
- 理解声明式编程的思维方式
- 掌握组件式编程的架构设计

实践建议

- 根据问题选择合适的编程范式
- 组合使用多种范式
- 注重代码的可读性和维护性
- 在实践中不断改进和优化