程序设计与计算思维

第 25 讲：C++ 面向对象编程

韩文弢

清华大学

2026 年 5 月

本讲内容

问题与目标

问题引入

随着程序规模增长，函数和变量的数量急剧膨胀。裸函数和全局变量之间没有明确的归属关系，代码变成一团乱麻——难以理解、难以修改、难以复用。

面向对象编程（OOP）的出发点是：将**数据**和**操作数据的函数**捆绑在一起，形成一个整体——**对象**。对象自己管理自己的状态，外部通过接口与之交互。

问题引入

OOP 的三个核心优势：

1. **封装（encapsulation）**：对象隐藏内部细节，只暴露接口。调用者不需要知道实现，只需知道“能做什么”
2. **继承（inheritance）**：子类复用基类的代码，避免重复。`Dog` 自动获得 `Animal` 的所有功能
3. **多态（polymorphism）**：同一接口，不同行为。`simulate(rv)` 能处理 `Gaussian`、`Bernoulli` 等所有分布，无需逐一判断类型

问题引入

在第 12 讲中，我们用 Python 的 OOP 构建了随机变量库：`RandomVariable` → `Gaussian` / `Bernoulli`。在第 22 讲中，用 `Grid` 和 `OceanModel` 类组织海洋模式的代码。

C++ 也支持面向对象，而且比 Python 更**严格**——编译器会在编译时就检查类型、访问权限、虚函数重写等。本讲介绍 C++ 面向对象编程的相关知识，并用 C++ 重新实现这些熟悉的例子，对比两种语言的 OOP 机制。

本讲目标

1. 掌握 C++ 类的定义方法：成员变量、构造函数、成员函数
2. 理解访问控制：`public`、`private`、`protected`
3. 理解动态内存管理：`new`/`delete` 与资源管理原则
4. 理解复制语义，了解 RAII 和三法则
5. 掌握继承与虚函数，理解多态的实现原理
6. 掌握运算符重载的基本方法
7. 能够用 C++ 重新实现第 12 讲和第 22 讲的面向对象设计

类与对象

第一个 C++ 类

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

class Student {
 public:
  Student(const std::string& name, int age)
      : name_(name), age_(age) {}

  void add_course(const std::string& course) {
    courses_.push_back(course);
  }

  void info() const {
```

第一个 C++ 类

```
    std::cout << name_ << ", age=" << age_ << ", courses=";
    for (const auto& c : courses_) {
      std::cout << c << " ";
    }
    std::cout << std::endl;
  }

 private:
  std::string name_;
  int age_;
  std::vector<std::string> courses_;
};

int main() {
```

第一个 C++ 类

```
    Student s("Alice", 20);
    s.add_course("Math");
    s.add_course("Physics");
    s.info();
}
```

与 Python 的对比：

概念	Python	C++
构造函数	`__init__(self, ...)`	`ClassName(...)` 构造函数
成员变量	`self.name = name`	`: name(name)` 初始化列表
成员函数	`def method(self):`	`void method() { }`
访问成员	`self.name`	`this->name` 或直接 `name`

构造函数与析构函数

构造函数在对象创建时自动调用，用于初始化；**析构函数**在对象销毁时自动调用，用于清理资源：

```
class Counter {
 public:
  Counter(int n) : count_(n) {
    std::cout << "Counter created: "
              << count_ << std::endl;
  }
  ~Counter() {
    std::cout << "Counter destroyed: "
              << count_ << std::endl;
  }
  void inc() { ++count_; }
```

构造函数与析构函数

```
 private:
  int count_;
};
```

与 Python 的对比：

概念	Python	C++
构造函数	`__init__(self)`	`ClassName(...)`
析构函数	`__del__(self)`（不推荐使用）	`~ClassName()`（RAII 核心）
调用时机	`__del__` 由垃圾回收决定	离开作用域时**确定性**调用

构造函数与析构函数

- Python 的 `__del__` 不可靠（垃圾回收时机不确定），C++ 的析构函数**确定**执行
- 析构函数是 RAII 的基础：资源释放与对象生命周期绑定
- 如果不定义析构函数，编译器自动生成一个默认版本（对于不需要手动释放资源的类足够）

构造函数与初始化列表

C++ 的**初始化列表**是一种高效的成员初始化方式：

```
Student(const std::string& name, int age)
    : name_(name), age_(age) {}
```

- `: name_(name)` 表示用参数 `name` 初始化成员变量 `name_`
- 初始化列表在构造函数体执行**之前**完成初始化
- 对于 `const` 成员和引用成员，**必须**使用初始化列表

常成员函数

用 const 修饰的成员函数承诺**不修改**对象状态：

```
class Circle {
 public:
  Circle(double r) : radius_(r) {}

  double area() const {
    return 3.14159 * radius_ * radius_;
  }

  void set_radius(double r) {
    radius_ = r;
  }
```

常成员函数

```
 private:
  double radius_;
};

const Circle c(5.0);
c.area();          // OK: const 函数
c.set_radius(3);   // 错误: 非 const 函数
```

- const 成员函数内不能修改任何成员变量
- const 对象只能调用 const 成员函数
- Python 没有 const 概念，靠约定（不变量）保证
- **建议**：所有不修改对象的成员函数都声明为 const

访问控制

C++ 通过 public、private、protected 控制成员的可见性：

关键字	类内部	子类	外部
public	可访问	可访问	可访问
protected	可访问	可访问	不可访问
private	可访问	不可访问	不可访问

- Python 没有严格的访问控制，用 _ 前缀表示“约定私有”
- C++ 的 private 是**编译器强制**的，违反会报编译错误
- 建议将数据成员设为 private，通过 public 方法访问

动态内存管理

new 与 delete

C++ 通过 new 和 delete 手动管理堆内存：

```
int* p = new int(42);       // 分配一个 int
std::cout << *p;            // 42
delete p;                   // 释放内存

int* arr = new int[100];    // 分配数组
delete[] arr;               // 释放数组（注意 []）
```

- new 在**堆**上分配内存，返回指针；delete 释放内存
- 每个 new 必须有对应的 delete，否则会发生**内存泄漏（memory leak）**
- Python 用垃圾回收自动管理内存，程序员不需要关心释放

RAII：资源获取即初始化

C++ 管理资源的核心原则是 **RAII**（Resource Acquisition Is Initialization）：

1. **获取资源**在构造函数中完成
2. **释放资源**在析构函数中完成
3. 对象离开作用域时，析构函数**自动**调用

```
void foo() {
  std::ofstream out("data.txt");    // 构造：打开文件
  out << "hello";                   // 使用
}  // 离开作用域，析构函数自动关闭文件
```

- **最佳实践原则**：永远不要写裸 new/delete，用 RAII 类代替

用 SimpleString 理解 RAII

下面实现一个简化版的字符串类，展示 RAII 和资源管理：

```
#include <cstring>
#include <iostream>

class SimpleString {
 public:
  SimpleString(const char* s = "") {
    size_t len = std::strlen(s);
    data_ = new char[len + 1];
    std::strcpy(data_, s);
    std::cout << "construct: \"" << data_ << "\""
              << std::endl;
  }
```

用 SimpleString 理解 RAII

```
~SimpleString() {
  std::cout << "destroy: \"" << data_ << "\""
            << std::endl;
  delete[] data_;
}

// 复制构造函数 (copy constructor)
SimpleString(const SimpleString& other) {
  size_t len = std::strlen(other.data_);
  data_ = new char[len + 1];
  std::strcpy(data_, other.data_);
  std::cout << "copy: \"" << data_ << "\"" << std::endl;
}
```

用 SimpleString 理解 RAII

```
// 复制赋值运算符 (copy assignment)
SimpleString& operator=(const SimpleString& other) {
  if (this != &other) {
    delete[] data_;
    size_t len = std::strlen(other.data_);
    data_ = new char[len + 1];
    std::strcpy(data_, other.data_);
    std::cout << "copy assign: \"" << data_ << "\""
              << std::endl;
  }
  return *this;
}
```

用 SimpleString 理解 RAII

```
  void print() const {
    if (data_) {
      std::cout << data_ << std::endl;
    } else {
      std::cout << "(null)" << std::endl;
    }
  }

 private:
  char* data_;
};

int main() {
  SimpleString a("hello");
```

用 SimpleString 理解 RAII

```
    SimpleString b = a;
}
```

- 构造函数用 new[] 分配内存，析构函数用 delete[] 释放
- 无论对象如何离开作用域，析构函数都会执行，确保**不泄漏**

复制语义

复制语义

当用一个对象初始化另一个对象时，编译器调用**复制构造函数**：

```
SimpleString a("hello");
SimpleString b = a;    // 复制构造：深拷贝 data_
```

复制语义要求**深拷贝**——为新对象分配独立的内存，复制内容：

操作	触发方式	行为
复制构造	`T b = a;` 或 `T b(a);`	深拷贝，两个对象各自拥有资源
复制赋值	`b = a;`	释放旧资源 → 深拷贝新资源

复制语义

- 如果不定义复制操作，编译器生成**浅拷贝**（只复制指针值）
- 浅拷贝导致两个对象指向同一块内存，析构时 `delete` 两次 → **崩溃**
- Python 中赋值都是“引用”，不存在这个问题（垃圾回收负责释放）
- 对于需要**转移**资源所有权（而非复制）的情况，C++11 引入了**移动语义**，避免了不必要的深拷贝，更加高效，具体参见参考资料

三法则

如果类需要手动管理资源，需要遵循**三法则**（Rule of Three）：

函数	说明
析构函数	释放资源
复制构造函数	深拷贝资源
复制赋值运算符	深拷贝资源

原则：定义了其中任何一个，就应该定义全部三个。

- 实际开发中，优先使用 `std::string`、`std::vector` 等标准库类型，它们已正确实现三法则（或者五法则，如果考虑移动语义）
- 只有在需要直接管理资源时才需要自己实现
- Python 程序员不需要关心这些细节——垃圾回收自动处理

运算符重载

重载 + 和 *

回顾第 12 讲的 Vector 类，现在用 C++ 实现：

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

class Vector2D {
 public:
  Vector2D(double x = 0, double y = 0) : x_(x), y_(y) {}

  double x() const { return x_; }
  double y() const { return y_; }

  Vector2D operator+(const Vector2D& o) const {
```

重载 + 和 *

```
  return Vector2D(x_ + o.x_, y_ + o.y_);
}
Vector2D operator*(double s) const {
  return Vector2D(x_ * s, y_ * s);
}

double magnitude() const {
  return std::sqrt(x_ * x_ + y_ * y_);
}
void print() const {
  std::cout << "(" << x_ << ", " << y_ << ")"
            << std::endl;
}
```

重载 + 和 *

```
 private:
  double x_;
  double y_;
};

int main() {
  Vector2D a(1.0, 2.0);
  Vector2D b(3.0, 4.0);
  Vector2D c = a + b;
  Vector2D d = a * 3.0;
  c.print();
  d.print();
  std::cout << "|c| = " << c.magnitude() << std::endl;
}
```

Python 与 C++ 运算符重载对比

运算	Python	C++
`a + b`	`def __add__(self, other)`	`operator+(const T&) const`
`a * s`	`def __mul__(self, scalar)`	`operator*(double) const`
`print(a)`	`def __str__(self)`	`friend ostream& operator<<`
`str(a)`	`def __repr__(self)`	无直接对应

- C++ 中 `const` 后缀表示该成员函数不修改对象状态
- Python 的 `self` 对应 C++ 的 `this` 指针
- C++ 的运算符重载需要**声明返回类型**

继承与多态

基本继承

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

class Animal {
 public:
  explicit Animal(const std::string& name) : name_(name) {}

  virtual std::string speak() const { return "..."; }

  virtual ~Animal() = default;

 protected:
  const std::string& name() const { return name_; }
```

基本继承

```
 private:
  std::string name_;
};

class Dog : public Animal {
 public:
  explicit Dog(const std::string& name) : Animal(name) {}

  std::string speak() const override {
    return name() + ": Woof!";
  }
};
```

基本继承

```
class Cat : public Animal {
 public:
  explicit Cat(const std::string& name) : Animal(name) {}

  std::string speak() const override {
    return name() + ": Meow!";
  }
};

void describe(const Animal& a) {
  std::cout << a.speak() << std::endl;
}

int main() {
```

基本继承

```
    Dog dog("Buddy");
    Cat cat("Kitty");

    describe(dog);
    describe(cat);

    std::vector<Animal*> animals;
    animals.push_back(&dog);
    animals.push_back(&cat);
    for (auto* a : animals) {
      std::cout << a->speak() << std::endl;
    }
}
```

基本继承

关键点：

1. `class Dog : public Animal` 表示 Dog 继承自 Animal
2. `virtual` 声明虚函数，支持**运行时多态**
3. `override` 关键字确保正确重写了基类虚函数（C++11）
4. `describe(const Animal& a)` 接受基类引用，自动调用子类实现

为什么需要虚函数?

考虑一个问题：用基类指针调用子类方法，会执行哪个版本?

```
Animal* a = new Dog("Buddy");
a->speak();  // ???
```

C++ 默认使用**静态绑定**——编译器根据指针类型（Animal*）决定调用 Animal::speak()。但我们的意图是调用 Dog::speak()。

虚函数改变了这个行为：编译器在**运行时**根据实际对象类型（Dog）查找正确的函数版本。这就是**动态绑定**：

```
class Animal {
  virtual std::string speak() const {
    return "...";
```

为什么需要虚函数?

```
    }
};

class Dog : public Animal {
    std::string speak() const override {
        return name() + ": Woof!";
    }
};

Animal* a = new Dog("Buddy");
a->speak();  // Dog::speak()  ← 动态绑定
```

- 没有 virtual：编译器只看指针类型，调用 Animal::speak() → 输出 "..."

为什么需要虚函数?

- 加了 `virtual`：编译器生成虚函数表（vtable），运行时查表调用 `Dog::speak()` → 输出 `"Buddy: Woof!"`

虚函数让 `describe(const Animal& a)` 这样的函数能**统一处理所有子类**——这就是多态的价值。

虚函数的代价

虚函数不是免费的，它带来三重开销：

开销	说明
内存	每个含虚函数的类有一张虚函数表（vtable），每个对象多一个指向 vtable 的指针（8 字节）
时间	每次虚函数调用需要间接查表（多一次内存访问），无法内联优化
编译	编译器无法在编译时确定调用目标，部分优化（如内联）失效

虚函数的代价

C++ 默认**不使用**虚函数，正是“零开销抽象”的体现：不为没用到的功能付出代价。只有程序员显式写 `virtual`，编译器才会引入这些开销。

对比 Python 的情况：

特性	C++	Python
默认行为	静态绑定（直接调用）	动态分发（查字典）
虚函数	显式 `virtual` 才启用	所有方法默认动态分发
单次调用开销	非虚：零；虚：一次间接查表	每次调用都查 `__dict__` + 类型对象
设计哲学	不为未使用的功能付出代价	统一模型，简化思维

虚析构函数

```
virtual ~Animal() = default;
```

- 如果基类有虚函数，析构函数也应该声明为 `virtual`
- 否则通过基类指针删除子类对象时，子类的析构函数不会被调用
- `= default` 表示使用编译器生成的默认实现

纯虚函数

C++ 用 `= 0` 声明**纯虚函数**（对应 Python 的 `@abstractmethod`）：

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

class Shape {
 public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  virtual ~Shape() = default;

  void describe() const {
    std::cout << "area=" << area()
```

```
                << ", perimeter=" << perimeter() << std::endl;
    }
};

class Rectangle : public Shape {
 public:
  Rectangle(double w, double h) : width_(w), height_(h) {}

  double width() const { return width_; }
  double height() const { return height_; }

  double area() const override { return width_ * height_; }

  double perimeter() const override {
```

纯虚函数

```
    return 2 * (width_ + height_);
  }

 private:
  double width_;
  double height_;
};

class Square : public Rectangle {
 public:
  explicit Square(double side) : Rectangle(side, side) {}
};

int main() {
```

纯虚函数

```
    Rectangle r(3, 4);
    Square s(5);

    r.describe();
    s.describe();

    std::vector<Shape*> shapes;
    shapes.push_back(&r);
    shapes.push_back(&s);
    for (auto* sh : shapes) {
      std::cout << "area = " << sh->area() << std::endl;
    }
}
```

纯虚函数

- 含有纯虚函数的类是**抽象类**，不能直接创建对象
- 子类必须实现所有纯虚函数，否则自身也是抽象类
- `describe()` 是具体方法，子类直接继承使用

示例：随机变量库

重新实现 RandomVariable

用 C++ 重新实现第 12 讲的随机变量层次结构：

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

class RandomVariable {
 public:
  virtual double sample() = 0;
  virtual double mean() = 0;
  virtual double var() = 0;
```

重新实现 RandomVariable

```
  double std_dev() { return std::sqrt(var()); }

  virtual ~RandomVariable() = default;
};

class Gaussian : public RandomVariable {
 public:
  Gaussian(double mu = 0.0, double sigma2 = 1.0)
      : mu_(mu), sigma2_(sigma2) {}

  double mu() const { return mu_; }
  double sigma2() const { return sigma2_; }

  double sample() override {
```

重新实现 RandomVariable

```
    static std::mt19937 rng(42);
    static std::normal_distribution<double> dist(0.0, 1.0);
    return mu_ + std::sqrt(sigma2_) * dist(rng);
  }

  double mean() override { return mu_; }
  double var() override { return sigma2_; }

 private:
  double mu_;
  double sigma2_;
};

class Bernoulli : public RandomVariable {
```

重新实现 RandomVariable

```
public:
 explicit Bernoulli(double p = 0.5) : p_(p) {}

 double p() const { return p_; }

 double sample() override {
   static std::mt19937 rng(42);
   static std::uniform_real_distribution<double> dist(0.0,
                                                      1.0);
   return dist(rng) < p_ ? 1.0 : 0.0;
 }

 double mean() override { return p_; }
 double var() override { return p_ * (1.0 - p_); }
```

重新实现 RandomVariable

```
 private:
  double p_;
};

void simulate(RandomVariable& rv, int n = 10000) {
  double sum = 0;
  for (int i = 0; i < n; ++i) {
    sum += rv.sample();
  }
  std::cout << "theoretical mean: " << rv.mean()
            << std::endl;
  std::cout << "simulated mean:   " << sum / n << std::endl;
}
```

重新实现 RandomVariable


```
int main() {
  Gaussian g(0.0, 1.0);
  Bernoulli b(0.3);

  simulate(g);
  simulate(b);
}
```

设计对比

设计要素	Python (lec12)	C++ (本讲)
抽象基类	`RandomVariable(ABC)`	`class RandomVariable { virtual ... = 0; }`
抽象方法	`@abstractmethod sample()`	`virtual double sample() = 0;`
具体方法	`def std(self): return sqrt(self.var())`	`double std_dev() { return sqrt(var()); }`
子类	`class Gaussian(RV):`	`class Gaussian : public RV {`
多态调用	`simulate(rv)` 自动分发	`simulate(RandomVariable\&rv)` 引用多态

核心设计（抽象基类 + 具体子类 + 多态）在两种语言中**完全一致**，只是语法不同。

示例：海洋模式

重新实现 Grid 和 OceanModel

用 C++ 重新实现第 22 讲的网格和海洋模式：

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

class Grid {
 public:
  Grid(int N, double L) : N_(N), L_(L), dx_(L / N) {
    Nx_ = N + 2;
    Ny_ = 2 * N + 2;
  }

  int Nx() const { return Nx_; }
```

重新实现 Grid 和 OceanModel

```
  int Ny() const { return Ny_; }
  double dx() const { return dx_; }

  void info() const {
    std::cout << "Grid: " << Nx_ << " x " << Ny_
              << ", dx=" << dx_ << std::endl;
  }

private:
  int N_;
  double L_;
  double dx_;
  int Nx_;
  int Ny_;
```

重新实现 Grid 和 OceanModel

```
};

class OceanModel {
 public:
  OceanModel(const Grid& grid, double kappa = 1e4)
      : grid_(grid),
        kappa_(kappa),
        dt_(0.0),
        u_(grid.Nx() * grid.Ny(), 0.0),
        v_(grid.Nx() * grid.Ny(), 0.0) {}

  void set_dt(double timestep) { dt_ = timestep; }

  void info() const {
```

重新实现 Grid 和 OceanModel

```
        std::cout << "OceanModel: kappa=" << kappa_
                  << ", dt=" << dt_ << std::endl;
        grid_.info();
    }

  private:
    Grid grid_;
    double kappa_;
    double dt_;
    std::vector<double> u_;
    std::vector<double> v_;
};

int main() {
```

重新实现 Grid 和 OceanModel

```
    Grid G(50, 1.0e6);
    OceanModel model(G, 1e4);
    model.set_dt(3600.0);
    model.info();
}
```

- `Grid` 封装空间离散化信息（纯数据类）
- `OceanModel` 通过**组合**（composition）持有 `Grid` 对象
- 这与 Python 版本的设计完全一致：`OceanModel` 包含 `Grid` 成员，而非继承

组合 vs 继承

特性	继承（is-a）	组合（has-a）
关系	“Dog **是一种** Animal”	“OceanModel **包含一个** Grid”
代码复用	自动获得基类方法	通过成员变量访问
耦合度	较高（子类依赖基类）	较低（独立变化）
灵活性	编译时确定	运行时可替换
推荐场景	“is-a” 关系	“has-a” 关系

原则：优先使用组合，而非继承。

本讲小结

知识要点

1. **类与对象**：class 定义类，构造函数初始化对象，初始化列表是惯用写法
2. **构造与析构**：构造函数初始化，析构函数清理资源；析构函数在离开作用域时确定性调用
3. **常成员函数**：const 修饰的成员函数不修改对象状态，const 对象只能调用 const 成员函数
4. **访问控制**：public/private/protected 控制成员可见性，C++ 编译器强制检查
5. **动态内存管理**：new/delete 手动管理堆内存，RAII 原则确保资源不泄漏

知识要点

6. **复制语义**：深拷贝资源，遵循三法则；移动语义用于高效转移所有权（参见外部资料）
7. **运算符重载**：`operator+`、`operator*` 等，让自定义类型支持自然运算
8. **继承与虚函数**：`virtual` 实现运行时多态，`override` 确保正确重写
9. **抽象类**：`= 0` 声明纯虚函数，子类必须实现，编译时检查
10. **组合优于继承**：`OceanModel` 包含 `Grid` 成员，而非继承

C++ 面向对象设计思维

1. **封装**：数据成员设为 `private`，通过 `public` 方法访问；`protected` 仅为子类开放必要的接口
2. **RAII**：构造函数获取资源，析构函数释放资源——用对象生命周期管理资源，而非手动配对 `new/delete`
3. **所有权明确**：复制语义深拷贝（各自独立），避免浅拷贝导致的悬挂指针；资源转移可使用移动语义
4. **组合优于继承**：`OceanModel` 包含 `Grid` 成员（has-a），而非继承（is-a）；继承层次保持扁平
5. **接口与实现分离**：抽象类定义接口（`RandomVariable` 的 `sample()`、`mean()`、`var()`），子类提供具体实现