Lecture 5

TOC

1. 继承
2. 继承和多态
3. 虚函数和函数重写(override)
4. 纯虚函数和抽象类
5. 虚继承
6. 访问基类成员
7. RTTI和类型转换
8. RAII(Resource Acquisition Is Initialization)
9. RC&GC
10. Lambda 表达式
11. 拓展
12. Assignment 2
13. Q&A

继承

C++ 中的继承是一种创建新类的方法，新类可以继承现有类的属性和行为。继承分为 `public`、`protected` 和 `private` 三种。

```
class Base {
public:
    int pub;
protected:
    int pro;
private:
    int pri;
};

class Derived_pub : public Base {
    // pub is public
    // pro is protected
    // pri is not accessible
};
```

```
class Derived_pro : protected Base {
    // pub is protected
    // pro is protected
    // pri is not accessible
};

class Derived_pri : private Base {
    // pub is private
    // pro is private
    // pri is not accessible
};
```

继承和多态

通过继承，派生类可以重写基类的方法，实现多态。

eg:

- `truck` 和 `taxi` 都继承自 `car`，但是 `truck` 重写了 `run` 方法，实现了多态。

对于派生类，可以直接绑定给基类指针或引用

```
Base *ptr = new Derived_pub();
Derived_pro d;
Base &ref = d;
```

启用虚函数后，调用 `ptr→run()` 时，会调用 `Derived_pub` 的 `run` 方法，而不是 `Base` 的 `run` 方法。

虚函数和函数重写(override)

在基类中，可以使用 `virtual` 关键字声明虚函数，派生类可以重写这个函数。

```
class Base {
public:
    virtual void run() {
        std::cout << "Base::run()" << std::endl;
    }
};

class Derived : public Base {
public:
    void run() final override {
        std::cout << "Derived::run()" << std::endl;
    }
};
```

- 这里的 `override` 关键字是 C++11 引入的，用于标记函数重写，如果标记为 `override` 但是没有重写基类的函数，编译器会报错。主要用于预防函数名拼写错误。
- 函数重写需要保证和基类函数的签名一致，否则会被视为新函数。
- 如果 `Derived` 仍有派生子类，但不希望子类重写 `run` 函数，可以使用 `final` 关键字。

- 如果不想某个类被继承，可以在类声明时使用 `final` 关键字。

```
class Base final {
    // ...
};

class Foo : public Base {
    // error: cannot derive from 'final' base 'Base' in derived type 'Foo'
};
```

- C++ 支持多继承，即一个类可以继承多个基类。

```
class Derived : public Base1, public Base2, protected Base3 {
    // ...
};
```

- 如果不想派生类中使用或重定义某个基类的函数，可以使用 `delete` 关键字。

```
class Base {
  public:
    Base(const Base&) = delete;
    Base& operator=(const Base&) = delete;
};
```

纯虚函数和抽象类

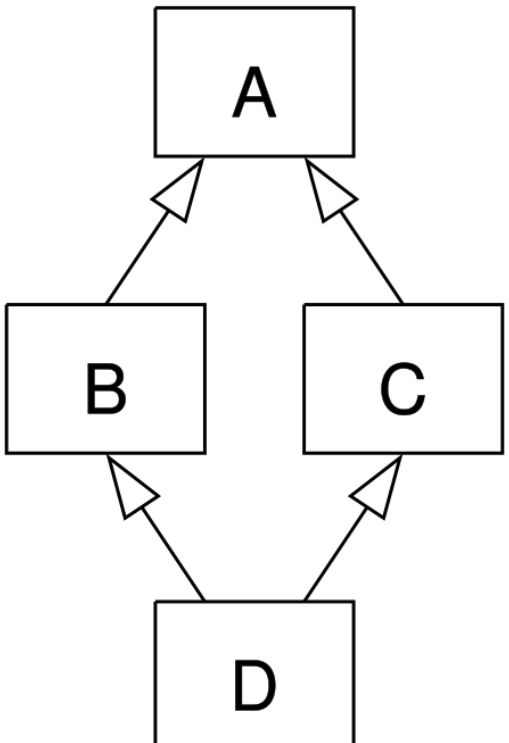
- 在基类中，我们可以只声明虚函数，而不定义它，这样的函数称为纯虚函数。
- 拥有纯虚函数的类称为抽象类，抽象类不能实例化，只能作为基类。C++的抽象类相当于 JAVA 的接口+抽象类的概念。

```
class Base {
public:
    virtual void run() = 0;
};

class Derived : public Base {
public:
    void run() override {
        std::cout << "Derived::run()" << std::endl;
    }
};

Base b; // error: cannot declare variable 'b' to be of abstract type 'Base'
Base* ptr = new Derived();
```

注意，即使声明为纯虚函数，也可以提供定义。

虚继承

虚继承是为了解决多继承时的菱形继承问题。



- 如果出现多继承，可能会导致基类被重复继承。访问时可能会出现二义性（访问 B 还是 C 的父类 A）

```
class A {
public:
    int a;
};

class B : virtual public A {
public:
    int b;
};

class C : virtual public A {
public:
    int c;
};

class D : public B, public C {
public:
    int d;
};
```

访问基类成员

- 在派生类中，可以通过 `BaseName::` 操作符访问基类的成员。其中 `BaseName` 是基类的名称。

```
class A {
public:
    int a;
};

class B : public A {
public:
    int b;
    void print() {
        std::cout << "a: " << A::a << std::endl;
    }
};

class C : public B {
public:
    int c;
    void print() {
        std::cout << "a: " << A::a << std::endl;
        std::cout << "b: " << B::b << std::endl;
    }
};
```

构造函数和析构函数

- 在派生类的构造函数中，可以调用基类的构造函数。
- 在派生类的析构函数中，会自动调用基类的析构函数。

```
class Base {
public:
    int a;
    Base(int a): a(a) {
        std::cout << "Base constructor\n";
    }
    virtual ~Base() {
        std::cout << "Base destructor\n";
    }
};
```

```
class Derived : public Base {
public:
    int b;
    Derived(int a, int b): Base(a), b(b) {
        std::cout << "Derived constructor\n";
    }
    ~Derived() {
        std::cout << "Derived destructor\n";
    }
};

int main() {
    Base* obj = new Derived();
    delete obj;
    return 0;
}
```

基类的析构函数应该声明为虚函数，以确保在通过基类指针删除派生类对象时，能够正确调用派生类的析构函数。

委托构造和继承构造

- C++11 引入了委托构造函数，可以在一个构造函数中调用另一个构造函数。
- 如果派生类没有新增的成员变量，可以直接使用 `using Base::Base;` 继承基类的构造函数。

```
class Base {
public:
    int a;
    Base(int a): a(a) {
        std::cout << "Base constructor\n";
    }
    Base(): Base(0) {}
};

class Derived : public Base {
public:
    using Base::Base;
};

int main() {
    Derived d(1);
    return 0;
}
```

RTTI和类型转换

- RTTI（Runtime Type Information，运行时类型信息）是 C++ 提供的在运行时获取对象的实际类型的机制。可以通过 typeid 运算符来获取类型信息，或者通过 dynamic_cast 在继承层次中进行安全的类型转换。
- 从派生类到基类的转换：这种转换是隐式的，且总是安全的。因为派生类对象可以被看作基类对象来使用，编译器自动支持这种类型转换。
- 从基类到派生类的转换：这类转换可能不安全，因为基类指针或引用不一定指向实际的派生类对象。使用 static_cast 进行这种转换是未定义行为，因为 static_cast 不会进行运行时检查。应当使用 dynamic_cast，它会在运行时检查类型的匹配性。如果转换失败，返回 nullptr（对于指针转换）或抛出 std::bad_cast 异常（对于引用转换）。

```
class Base
class Derived : public Base;

Base* b = new Derived(); // implicit conversion
Base* c = static_cast<Base*>(new Derived()); // OK

// undefined behavior
Derived* d = static_cast<Derived*>(new Base());
Derived* e = dynamic_cast<Derived*>(b); // OK

Base a;
Derived& f = dynamic_cast<Derived&>(a); // std::bad_cast
```

RAII(Resource Acquisition Is Initialization)

RAII 是一种 C++ 的资源管理技术，通过在对象的构造函数中获取资源，然后在对象的析构函数中释放资源，来确保资源的正确释放。

```
class File {
public:
    File(const char* filename) {
        file = fopen(filename, "r");
        if (!file) {
            throw std::runtime_error("Failed to open file");
        }
    }
    ~File() {
        if (file) {
            fclose(file);
        }
    }
...
};
```

RC&GC

常见的内存管理技术有两种：

- 引用计数（Reference Counting）：通过计数器来记录对象的引用次数，当引用次数为 0 时，释放对象。
- 垃圾回收（Garbage Collection）：GC 通过不同的算法（如标记-清除、分代收集等）来识别不再被使用的对象并释放它们。开发者无需手动管理内存，垃圾回收器会在适当的时间进行回收。

C++ 中，RC 通过类的构造函数和析构函数来实现。

- 在复制构造函数中，引用计数加一。
- 在移动构造函数中，引用计数不变。
- 在析构函数中，引用计数减一，当引用计数为 0 时，释放对象。

智能指针（Smart Pointer）

智能指针是 C++ 标准库中的 RAII 的实现，可以简化内存管理，避免内存泄漏。

- `std::unique_ptr`：独占所有权的智能指针，不能拷贝。
- `std::shared_ptr`：共享所有权的智能指针，可以拷贝。
- `std::weak_ptr`：弱引用智能指针，不增加引用计数。用于解决循环引用问题。

在现代 C++(非 no_std情况) 中，尽量使用智能指针来管理内存，避免使用裸指针。

Lambda 表达式

Lambda 表达式是 C++11 引入的一种匿名函数，可以在需要函数对象的地方使用。

- 本质上是一个闭包类型，可以捕获外部变量，并提供 `()` 运算符重载以供调用。
- Lambda 表达式的语法为 `[capture](parameters) → return_type { body }`,在 C++14 中，可以省略返回类型,编译器会自动推导。

```
auto add = [](int a, int b) → int {
    return a + b;
};
int c = add(1, 2); // 3
```

- 捕获列表 `capture` 可以是值捕获、引用捕获、隐式捕获等。
  - `[a, &b]`：值捕获 a，引用捕获 b
  - `[&]`：引用捕获所有外部变量
  - `[=]`：值捕获所有外部变量
  - `[=, &a]`：值捕获所有外部变量，引用捕获 a
  - `[this]`：捕获当前对象的指针，使得可以在 lambda 中访问类成员

当 lambda 表达式没有捕获列表时，可以转换为函数指针。

```
int (*add)(int, int) = [](int a, int b) -> int {
    return a + b;
};
```

注意：Lambda 的按引用捕获需要考虑声明周期问题，避免悬垂引用。

Reference

拓展

虚函数表

- C++ 中的多态是通过虚函数表（vtable）实现的。每个有虚函数的类都有自己的 vtable, 在每个类中存在一个隐藏成员 vptr 指向 vtable 的地址。
- 调用时，通过 vptr 找到 vtable，再通过 vtable 找到对应的函数，实现多态。

Assignment 2

- https://github.com/ARTINX/2025-Vision-Train-Lecture/tree/main/assignment/Assignment2

Q&A