C++类的继承与多态 - 完整复习笔记

1. 一、类的继承与派生基础

1.1 基本概念

- **基类（Base Class）**：被继承的类，也叫超类
- **派生类（Derived Class）**：继承产生的新类，也叫子类
- **类继承层次结构**：基类和派生类的集合
- **子类型**：如果基类和派生类共享相同的公有接口，则派生类被称作基类的子类型

1.2 继承语法

```
class Derived : public/protected/private Base {
    // 派生类成员
};
```

Fence 1

1.3 继承的类型

1.3.1 单继承

- 派生类只有一个直接基类
- 一个基类可以直接派生出多个派生类

1.3.2 多重继承

- 一个派生类可以继承多个基类

```
class RobotDog : public Animal, public Machine {
public:
    void bark();  // 派生类自己的方法
};
```

Fence 2

⚠️ **注意**：需要小心处理菱形问题，可用virtual关键字避免重复继承

1.3.3 多级继承

- 派生类可以作为基类再继续派生
- **直接基类**：直接参与派生的基类
- **间接基类**：基类的基类及更深层的基类

1.3.4 类族

- 一个基类直接派生出多个派生类形成的相互关联的类族

1.4 重要关键字

1.4.1 final关键字

```
class Class final {
    // 该类不允许任何类继承
};
```

Fence 3

1.4.2 访问基类重名成员

- 不加类名限定：默认处理派生类成员
- 访问基类重名成员：`派生类对象.基类名::成员名()`

---

2. 二、派生类的访问权限

2.1 三种派生方式对比

派生方式	基类public成员	基类protected成员	基类private成员
public	仍为public	仍为protected	**不可访问**
protected	变为protected	仍为protected	**不可访问**
private	变为private	变为private	**不可访问**

Table 1

⭐ **核心要点**：三种派生方式都无法访问基类的私有成员

---

3. 三、派生类的构造函数与析构函数

3.1 派生类构造函数语法

```
派生类名(参数总表) : 初始化符表 {
    构造函数体
}
```

Fence 4

3.2 初始化符表格式

```
基类名1(基类参数表1), ..., 基类名n(基类参数表n),
对象成员名1(对象成员参数表1), ..., 对象成员名m(对象成员参数表m)
```

Fence 5

3.3 构造函数执行顺序

1. **基类优先**：调用各基类构造函数（按派生时声明顺序）
2. **对象成员**：调用对象成员构造函数（按在类中声明顺序）
3. **派生类自身**：执行派生类构造函数体

3.4 初始化符表 vs 构造函数体

3.4.1 初始化符表的作用

- **构造阶段**：用于初始化基类和本类的非静态成员
- **适用范围**：基类、本类非静态成员（包括基本类型和对象成员）
- **效率更高**：直接构造，不涉及赋值

3.4.2 构造函数体的作用

- **赋值阶段**：执行额外逻辑、赋值、条件判断等
- **访问权限**：可直接访问基类的public和protected成员
- **灵活性高**：适合动态调整值或复杂逻辑

⭐ **重要原则**：优先在初始化符表中完成初始化，在构造函数体中处理后续逻辑

3.5 构造函数的特殊情况

3.5.1 构造函数"继承"

```
class Derived : public Base {
    using Base::Base;  // 继承基类构造函数（除无参构造函数）
};
```

Fence 6

3.5.2 拷贝构造函数

```
Derived(const Derived& other) : Base(other), member{other.member} {
    // 派生类拷贝构造函数体
}
```

Fence 7

3.6 析构函数

- **执行顺序**：与构造函数相反
    1. 派生类析构函数体
    2. 对象成员析构
    3. 基类析构
- **重要提醒**：派生类析构函数一律重写，可免出错

---

4. 四、继承中的特殊成员

4.1 友元的继承

- ❌ **基类的友元不继承**：派生类不会自动获得基类的友元关系
- ✅ **友元关系被继承**：基类成员是某类的友元，作为派生类继承成员仍是该类的友元

4.2 静态成员的继承

4.2.1 继承规则

- 基类的静态成员被派生类继承，保持静态属性
- 静态成员属于类，不属于对象
- 基类和派生类的所有对象共享同一个静态成员

4.2.2 访问方式（当派生类未定义同名静态成员时）

```
Base::staticMember      // ✅ 推荐
Derived::staticMember   // ✅ 推荐
baseObj.staticMember    // ✅ 可用但不推荐
derivedObj.staticMember // ✅ 可用但不推荐
```

Fence 8

4.2.3 隐藏问题

- 如果派生类定义同名静态成员，会隐藏基类的静态成员
- 可通过`Base::staticMember`显式访问基类成员

4.3 赋值兼容性

4.3.1 赋值优先级

1. 派生类有自定义赋值运算符 → 使用自定义版本
2. 仅基类有赋值运算符 → 系统自动定义派生类版本
3. 都没有 → 使用默认按位拷贝

4.3.2 向上转型（⭐ 重要概念）

```
Base base = derived;        // ✅ 基类对象 = 派生类对象
Base* ptr = &derived;       // ✅ 基类指针 = 派生类对象地址
Base& ref = derived;        // ✅ 基类引用 = 派生类对象
```

Fence 9

5. 五、虚基类与虚拟继承

5.1 二义性问题

5.1.1 单继承重名处理

```
derived.member;          // 默认访问派生类成员
derived.Base::member;    // 显式访问基类成员
```

Fence 10

5.1.2 多继承重名处理

- 同单继承，通过类名限定符解决

5.1.3 菱形继承问题

```
class A { int data; };
class B : public A {};
class C : public A {};
class D : public B, public C {};  // D包含两个A的实例
```

Fence 11

5.2 虚拟继承解决方案

5.2.1 语法格式

```
class Derived : virtual public Base {
    // 派生类体
};
```

Fence 12

5.2.2 解决菱形继承

```
class A { int data; };
class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};  // D只包含一个A的实例
```

Fence 13

5.2.3 存储结构

- 普通继承：`((A B) (A C) D)` - 两个A实例
- 虚拟继承：`(((A) B C) D)` - 一个A实例

---

6. 六、多态性与虚函数

6.1 多态性的体现形式

1. **编译时多态**：函数重载、静态联编
2. **运行时多态**：动态联编、虚函数、纯虚函数

6.2 函数重载（编译时多态）

- **要求**：同名函数必须有不同的参数表
  - 参数个数不同
  - 参数类型不同
  - 参数顺序不同
- **特点**：静态联编，编译阶段确定调用哪个函数

6.3 函数超载vs函数重载

- **函数重载**：任意作用域内的同名不同参数函数
- **函数超载**：仅在基类与派生类间，完全相同的函数名、参数表、返回类型

6.4 虚函数（运行时多态）

6.4.1 定义语法

```
class Base {
public:
    virtual void func();  // 虚函数
};
```

Fence 14

6.4.2 重要特性

- 基类中定义虚函数，派生类中同原型函数默认为虚函数
- 只在派生类中定义虚函数没有意义
- 实现动态联编，运行时确定调用哪个函数

6.4.3 使用示例

```
Base* ptr;              // 基类指针
Derived1 d1; Derived2 d2;
ptr = &d1; ptr->func(); // 调用Derived1::func()
ptr = &d2; ptr->func(); // 调用Derived2::func()
```

Fence 15

6.5 纯虚函数

```
virtual 函数原型 = 0;   // 纯虚函数声明
```

Fence 16

6.5.1 特点

- 不能被直接调用
- 只规定派生类虚函数的原型规格
- 具体实现在派生类中给出

6.6 抽象基类

6.6.1 定义

- 含有纯虚函数的基类

6.6.2 特性

- ❌ 不能创建抽象基类的对象
- ✅ 可以创建指向抽象基类的指针和引用
- 只有创建派生类对象时，才有抽象基类实例伴随而生

6.6.3 继承规则

- 派生类必须实现所有纯虚函数，否则仍为抽象基类

---

7. 七、重要补充知识点

7.1 虚函数的实现机制

7.1.1 虚函数表（vtable）

- 每个包含虚函数的类都有一个虚函数表
- 虚函数表存储虚函数的地址
- 对象中包含指向虚函数表的指针（vptr）

7.1.2 动态绑定过程

```
Base* ptr = new Derived();
ptr->virtualFunc();  // 运行时通过vptr找到正确的虚函数
```

Fence 17

7.2 虚析构函数

```
class Base {
public:
    virtual ~Base() {}  // 虚析构函数
};
```

Fence 18

⭐ **重要**：当基类指针指向派生类对象时，删除对象需要虚析构函数确保正确析构

7.3 构造函数中的虚函数调用

⚠️ **注意**：构造函数中调用虚函数，调用的是当前类的版本，不是派生类版本

7.4 对象切片（Object Slicing）

```
Derived d;
Base b = d;  // 对象切片：只复制基类部分，丢失派生类特有成员
```

Fence 19

7.5 多重继承的构造顺序

- 按照派生类声明中基类的顺序调用构造函数
- 不是按照初始化列表中的顺序

7.6 名字隐藏（Name Hiding）

```
class Base {
public:
    void func(int);
    void func(double);
};
class Derived : public Base {
public:
    void func(string);  // 隐藏了Base中的所有func函数
};
```

Fence 20

解决方法：使用`using Base::func;`引入基类函数

7.7 虚继承的构造函数

- 虚基类的构造函数由最终派生类直接调用
- 中间派生类中对虚基类的构造调用被忽略

---

8. 八、重点总结与考试要点

8.1 🎯 必考知识点

1. **三种继承方式的访问权限对比**
2. **构造函数和析构函数的执行顺序**
3. **初始化符表的使用和与构造函数体的区别**
4. **虚拟继承解决菱形继承问题**
5. **虚函数实现多态的机制**
6. **抽象基类的概念和使用**
7. **虚析构函数的必要性**
8. **名字隐藏问题及解决方法**

8.2 🔍 易错点提醒

1. 所有派生方式都不能访问基类private成员
2. 构造函数执行顺序：基类→对象成员→派生类
3. 析构函数执行顺序与构造函数相反
4. virtual关键字在基类中声明，派生类自动继承虚函数属性
5. 纯虚函数不能直接调用，只能在派生类中实现
6. **构造函数中调用虚函数不会发生多态**
7. **基类指针删除派生类对象需要虚析构函数**
8. **对象赋值会发生对象切片**
9. **多重继承构造顺序按声明顺序，不按初始化列表顺序**

8.3 💡 编程实践建议

1. 优先使用初始化符表而非构造函数体赋值
2. 派生类析构函数一律重写
3. 使用虚拟继承解决菱形继承问题
4. 基类指针实现多态时注意内存管理
5. 抽象基类设计时合理规划纯虚函数接口
6. **基类析构函数声明为虚函数**
7. **使用using声明解决名字隐藏问题**
8. **避免在构造/析构函数中调用虚函数**