第八章：多态与面向对象编程

- **C++中的类型**：
  - 用户自定义类型：如类（classes）、结构体（structs）、枚举（enums）等。
  - 内置类型：如 `int`、`float`、`double` 等。
  - 多态性主要通过**继承**和**虚函数**来实现。
- **公共继承**：
  - 公共继承创建子类，使派生类继承基类。
  - 这种继承方式被称为**继承多态**，派生类是基类的一种（“is-a”关系）。

Studio中的构造函数和析构函数

在Studio练习中，关于构造函数和析构函数在继承中的调用顺序有以下要点：

1. **构造函数调用顺序**：
   - 对于派生类对象，首先调用基类的构造函数，然后调用派生类的构造函数。
   - 这保证了对象的基类部分先于派生类部分初始化。
   - 输出示例：

     ```
     BaseClass constructor
     DerivedClass constructor
     ```

2. **析构函数调用顺序**：
   - 析构函数的调用顺序与构造函数相反：先调用派生类的析构函数，再调用基类的析构函数。
   - 这确保在销毁对象时，先清理派生类的部分，再清理基类的部分。
   - 输出示例：

     ```
     DerivedClass destructor
     BaseClass destructor
     ```

静态类型与动态类型：

- **静态类型**：在编译时已知的类型，基于变量的声明。
- **动态类型**：在运行时决定的对象的实际类型，尤其是基类指针或引用指向派生类对象时。

在你的练习中，当你直接对对象调用方法时，调用的是静态类型的方法；但如果使用**引用或指针**，并且该方法是虚函数，则会调用动态类型（即派生类）的版本。

虚函数

- **虚函数（virtual）**：当基类声明一个函数为 `virtual` 时，允许派生类重写该函数。在运行时，根据对象的动态类型决定调用哪个版本的函数（即**动态绑定**）。
  - 使用虚函数时，输出会是派生类的版本：

    ```
    DerivedClass display
    ```

- **非虚函数**：如果函数不是虚的，C++会使用**静态绑定**，调用静态类型（基类）的方法。
  - 示例输出：

    ```
    BaseClass display
    ```

Studio中的代码示例：

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

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

int main() {
    Base b;
    Derived d;

    // 静态绑定
    b.display();  // BaseClass display
    d.display();  // DerivedClass display

    // 动态绑定（通过指针或引用）
    Base* bp = &d;
    bp->display();  // DerivedClass display
}
```

继承的形式：

- **公有继承**：基类的 `public` 和 `protected` 成员在派生类中保持其访问权限。
- **保护继承**：基类的 `public` 成员在派生类中变为 `protected` 。
- **私有继承**：基类的所有成员（包括 `public` 和 `protected` ）在派生类中都变为 `private` 。
  在你展示的这一张图片中，主要讨论了C++中三种继承形式：**公有继承（public inheritance）**、**保护继承（protected inheritance）** 和 **私有继承（private inheritance）** 。下面是这三种继承方式的详细解释：

1. 公有继承（Public Inheritance）

- **说明**：公有继承是最常见的一种继承方式，派生类继承基类时，基类的 `public` 成员在派生类中保持 `public` ， `protected` 成员保持 `protected` 。
- **使用场景**：当派生类被看作是基类的扩展时（“is-a”关系），公有继承最为合适。
- **影响**：
  - 基类的 `public` 成员在派生类中仍然是 `public` ，这意味着派生类和外部代码都可以访问基类的 `public` 成员。
  - 基类的 `protected` 成员在派生类中仍然是 `protected` ，这意味着派生类可以访问基类的 `protected` 成员，但外部代码不能访问。
  - 基类的 `private` 成员仍然只能由基类自身访问，不能被派生类或外部代码访问。
- **示例**：

```
class A {
public:
    int i;
protected:
    int j;
private:
    int k;
};

class B : public A {
    // B中i是public的，j是protected的，k不可访问
};
```

2. 保护继承（Protected Inheritance）

- **说明**：在保护继承中，基类的 `public` 成员在派生类中变为 `protected` ，而 `protected` 成员保持不变， `private` 成员依旧不可访问。

- **使用场景**：当你希望派生类继承基类的实现，但不希望派生类暴露基类的公共接口时，使用保护继承。
- **影响**：
  - 基类的 `public` 成员在派生类中变为 `protected` ，这意味着派生类本身可以访问这些成员，但外部代码不能直接访问。
  - 基类的 `protected` 成员在派生类中保持 `protected` ，派生类可以访问，但外部代码不能访问。
  - 基类的 `private` 成员依然不可访问。
- **示例**：

```
class C : protected A {
    // C中i和j都是protected的，k不可访问
};
```

3. 私有继承（Private Inheritance）

- **说明**：私有继承将基类的所有 `public` 和 `protected` 成员都变为 `private` ，即在派生类中，这些成员都变成了私有的，外部代码无法访问这些成员。
- **使用场景**：当你想使用基类的实现，但不希望暴露任何继承的接口时，私有继承是合适的。
- **影响**：
  - 基类的 `public` 和 `protected` 成员在派生类中都变为 `private` ，只有派生类中的代码能够访问它们。
  - 外部代码不能访问这些成员，包括派生类对象中的这些成员。
  - 基类的 `private` 成员依然不可访问。
- **示例**：

```
class D : private A {
    // D中i和j都变成private，k不可访问
};
```

4. 总结

- **公有继承**保持了基类接口的公开性，派生类仍然提供基类的 `public` 接口。
- **保护继承**隐藏了基类的 `public` 接口，仅在派生类中可用，外部代码无法访问基类的 `public` 成员。
- **私有继承**完全隐藏了基类的接口，派生类对外完全封闭基类的 `public` 和 `protected` 成员。

这三种继承方式的区别主要体现在基类成员在派生类中的访问权限上，继承的访问控制决定了外部代码如何与派生类交互。在实际编程中，选择合适的继承方式可以有效控制类的接口暴露和封装。

对象切片（Object Slicing）：

- **对象切片**发生在将派生类对象按值传递给期望基类对象的函数时，派生类的部分会被“切掉”。
- 解决方法是通过引用或指针传递对象，避免切片。

智能指针与内存管理：

1. **使用智能指针**：
   - 通过 `shared_ptr` 和 `unique_ptr` ，可以避免内存泄漏。
   - 当对象超出作用域时，析构函数会自动调用。

   示例：

   ```
   std::shared_ptr<Base> ptr = std::make_shared<Derived>();
   ptr->display();  // 动态绑定：DerivedClass display
   ```

2. **使用 `new` 和 `delete` 操作符管理内存**：
   - 如果使用原始指针而不适当地 `delete` ，会导致内存泄漏。
   - 在使用多态对象时，确保基类的析构函数是虚的，否则通过基类指针删除派生类对象时可能不会调用派生类的析构函数，导致资源泄漏。

析构函数的虚拟性

如果基类的析构函数没有标记为 `virtual` ，通过基类指针删除派生类对象时，不会调用派生类的析构函数，导致资源未被正确释放。因此，在使用多态时，基类的析构函数必须声明为虚函数。