

继承

主讲：陈笑沙

复习

以下哪个描述是错误的？

- A. 非静态方法可以调用静态方法和属性
- B. 静态方法可以调用其他静态方法和属性
- C. 静态方法可以调用非静态方法和属性
- D. 非静态方法可以调用其他非静态方法和属性

目录

- 8.1 继承的概念
- 8.2 继承的工作方式
- 8.3 继承与组合
- 8.4 多继承如何工作
- 8.5 多继承的模糊性
- 8.6 虚拟继承
- 8.7 多继承的构造顺序

8.1 继承的概念

8.1 继承的概念

- 类描述了群体的共性，通过创建类的不同对象，我们实现了代码重用。但这种重用是不充分的。

8.1 继承的概念

- 类描述了群体的共性，通过创建类的不同对象，我们实现了代码重用。但这种重用是不充分的。
- 例如，汽车类 Car 有启动，刹车，换挡，改变方向功能。当 Car 类的功能需要进行扩展为变形金刚类 Transformer，增加“变身”的功能，怎么做？

8.1 继承的概念

8.1 继承的概念

- 方法一：

8.1 继承的概念

- 方法一：
 - 创建一个新类 Transformer，在其中粘贴 Car 类的代码。

8.1 继承的概念

- 方法一：
 - 创建一个新类 Transformer，在其中粘贴 Car 类的代码。
 - 再添加新的方法 void transform(){ ... }，这时如果 Car 类增加一个功能：自动停车 void stop()，那么我们要同时更改 Car 类和 Transformer 类的代码。

8.1 继承的概念

8.1 继承的概念

- 方法二：

8.1 继承的概念

- 方法二：
 - Transformer 类“使用”了 Car 类的特性，那么不需要复制粘贴，Transformer 类就自动具有 Car 类所有的特性。

8.1 继承的概念

- 方法二：
 - Transformer 类“使用”了 Car 类的特性，那么不需要复制粘贴，Transformer 类就自动具有 Car 类所有的特性。
 - 更好的一点是，任何时候 Car 类进行修改，Transformer 类都能应用这种修改。这就是**继承与派生**。

8.1 继承的概念

现实中的继承与派生

8.1 继承的概念

现实中的继承与派生

- 猫、狗都属于哺乳动物，具备胎生、哺乳、恒温等特征，但又具有各自的特性。

8.1 继承的概念

现实中的继承与派生

- 猫、狗都属于哺乳动物，具备胎生、哺乳、恒温等特征，但又具有各自的特性。
- 这就是“继承”关系的重要性质。

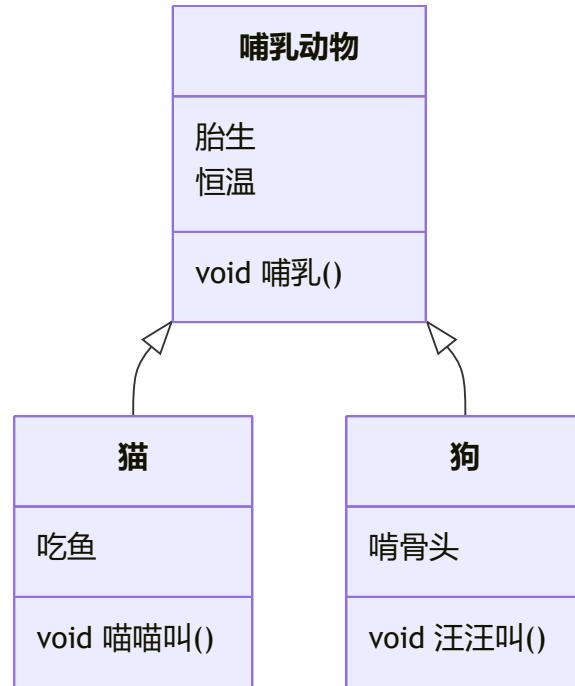
8.1 继承的概念

现实中的继承与派生

- 猫、狗都属于哺乳动物，具备胎生、哺乳、恒温等特征，但又具有各自的特性。
- 这就是“继承”关系的重要性质。
- 形成继承关系的2个类之间，具有 IS_A 的关系。

8.1 继承的概念

现实中的继承与派生



8.1 继承的概念

8.1 继承的概念

- 事物通过分类与分层相关在一起

8.1 继承的概念

- 事物通过分类与分层相关在一起
- 上层事物具有下层事物的共性

8.1 继承的概念

- 事物通过分类与分层相关在一起
- 上层事物具有下层事物的共性
- 例如

8.1 继承的概念

- 事物通过分类与分层相关在一起
- 上层事物具有下层事物的共性
- 例如
 - 四川人与浙江人都归类为中国人

8.1 继承的概念

- 事物通过分类与分层相关在一起
- 上层事物具有下层事物的共性
- 例如
 - 四川人与浙江人都归类为中国人
 - 具有黄皮肤，黑头发，会说中国话的共性

8.1 继承的概念

8.1 继承的概念

- 编程需要处理分层的事物(数据)

8.1 继承的概念

- 编程需要处理分层的事物(数据)
- 拥有上层数据或对象，用继承描述下层数据与对象

8.1 继承的概念

- 编程需要处理分层的事物(数据)
- 拥有上层数据或对象，用继承描述下层数据与对象
- 既共享了上层数据与代码获得代码重用的好处，又保持了类结构特有的封装性

8.1 继承的概念

- 编程需要处理分层的事物(数据)
- 拥有上层数据或对象，用继承描述下层数据与对象
- 既共享了上层数据与代码获得代码重用的好处，又保持了类结构特有的封装性
- 使得继承后的实体又可以作为上层数据和代码继承下去

8.1 继承的概念

8.1 继承的概念

- **继承**

8.1 继承的概念

- **继承**
 - 一旦指定了某种事物父代的本质特征，那么它的子代将会自动具有那些性质。

8.1 继承的概念

- **继承**

- 一旦指定了某种事物父代的本质特征，那么它的子代将会自动具有那些性质。
- 这是一种朴素的可重用的概念。

8.1 继承的概念

- **继承**

- 一旦指定了某种事物父代的本质特征，那么它的子代将会自动具有那些性质。
- 这是一种朴素的可重用的概念。
- 继承就是在一个已经存在的类的基础上建立另一个新的类。

8.1 继承的概念

- **继承**

- 一旦指定了某种事物父代的本质特征，那么它的子代将会自动具有那些性质。
- 这是一种朴素的可重用的概念。
- 继承就是在一个已经存在的类的基础上建立另一个新的类。

- **派生**

8.1 继承的概念

- **继承**

- 一旦指定了某种事物父代的本质特征，那么它的子代将会自动具有那些性质。
- 这是一种朴素的可重用的概念。
- 继承就是在一个已经存在的类的基础上建立另一个新的类。

- **派生**

- 子代可以拥有父代没有的特性，这是可扩充的概念。

8.1 继承的概念

8.1 继承的概念

- 派生类的功能主要通过以下方式来体现

8.1 继承的概念

- 派生类的功能主要通过以下方式来体现
 - 吸收基类成员

8.1 继承的概念

- 派生类的功能主要通过以下方式来体现
 - 吸收基类成员
 - 添加新成员

8.1 继承的概念

- 派生类的功能主要通过以下方式来体现
 - 吸收基类成员
 - 添加新成员
 - 改造基类成员

8.1 继承的概念

8.1 继承的概念

- 从编码的角度来看，派生类从基类中以较低的代价换来了较大的灵活性

8.1 继承的概念

- 从编码的角度来看，派生类从基类中以较低的代价换来了较大的灵活性
 - 派生类可以对继承的属性进行扩展、限制或改变

8.1 继承的概念

- 从编码的角度来看，派生类从基类中以较低的代价换来了较大的灵活性
 - 派生类可以对继承的属性进行扩展、限制或改变
 - 一旦产生了可靠的基类，只需要调试派生类中所作的修改即可

8.2 继承的工作方式

8.2 继承的工作方式

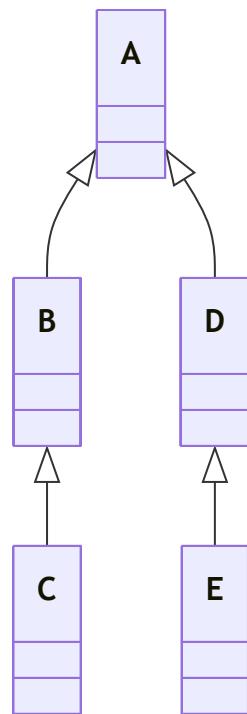
- 单继承：派生类只有一个直接基类

8.2 继承的工作方式

- 单继承：派生类只有一个直接基类
- 多继承：派生类有多个直接基类

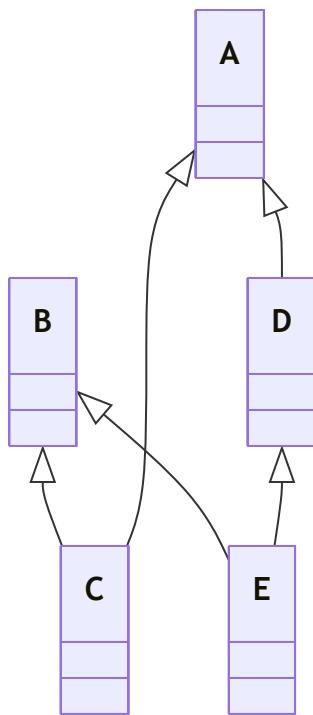
8.2 继承的工作方式

单继承



8.2 继承的工作方式

多继承



8.2 继承的工作方式

定义单继承派生类的语法格式

```
1 class 派生类名 : <继承方式> 基类名{  
2     ... //派生类修改基类的成员  
3 };
```

8.2 继承的工作方式

- 虽然继承了基类的所有成员，但是派生类并非都能访问基类的所有成员，继承方式会影响派生类对基类中各种成员的使用。

8.2 继承的工作方式

派生方式	公有成员	保护成员	私有成员
公有派生	公有成员	保护成员	不可访问成员
私有派生	私有成员	私有成员	不可访问成员
保护派生	保护成员	保护成员	不可访问成员

8.2 继承的工作方式

```
1 class Base {
2 public:
3     int publicVar = 1;
4     void publicFunc() {
5         cout << "Base::publicFunc()" << endl;
6     }
7
8 protected:
9     int protectedVar = 2;
10    void protectedFunc() {
11        cout << "Base::protectedFunc()" << endl;
12    }
13
14 private:
15     int privateVar = 3;
```

8.2 继承的工作方式

```
1 class Base {
2 public:
3     int publicVar = 1;
4     void publicFunc() {
5         cout << "Base::publicFunc()" << endl;
6     }
7
8 protected:
9     int protectedVar = 2;
10    void protectedFunc() {
11        cout << "Base::protectedFunc()" << endl;
12    }
13
14 private:
15     int privateVar = 3;
```

8.2 继承的工作方式

```
3     int publicVar = 1;
4     void publicFunc() {
5         cout << "Base::publicFunc()" << endl;
6     }
7
8 protected:
9     int protectedVar = 2;
10    void protectedFunc() {
11        cout << "Base::protectedFunc()" << endl;
12    }
13
14 private:
15     int privateVar = 3;
16     void privateFunc() {
17         cout << "Base::privateFunc()" << endl;
...     }
```

8.2 继承的工作方式

```
5     cout << "Base::publicFunc()" << endl;
6 }
7
8 protected:
9     int protectedVar = 2;
10    void protectedFunc() {
11        cout << "Base::protectedFunc()" << endl;
12    }
13
14 private:
15     int privateVar = 3;
16     void privateFunc() {
17         cout << "Base::privateFunc()" << endl;
18     }
19 }
```

8.2 继承的工作方式

```
1 class Base {  
2 public:  
3     int publicVar = 1;  
4     void publicFunc() {  
5         cout << "Base::publicFunc()" << endl;  
6     }  
7  
8 protected:  
9     int protectedVar = 2;  
10    void protectedFunc() {  
11        cout << "Base::protectedFunc()" << endl;  
12    }  
13  
14 private:  
15     int privateVar = 3;
```

8.2 继承的工作方式

```
1 class PublicDerived : public Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PublicDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→公有)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PublicDerived : public Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PublicDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→公有)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PublicDerived : public Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PublicDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→公有)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PublicDerived : public Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PublicDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→公有)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PublicDerived : public Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PublicDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→公有)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PublicDerived : public Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PublicDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→公有)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PublicDerived : public Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PublicDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→公有)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class ProtectedDerived : protected Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "ProtectedDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→保护)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class ProtectedDerived : protected Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "ProtectedDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→保护)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 }
```

8.2 继承的工作方式

```
1 class ProtectedDerived : protected Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "ProtectedDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→保护)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class ProtectedDerived : protected Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "ProtectedDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→保护)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 }
```

8.2 继承的工作方式

```
1 class ProtectedDerived : protected Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "ProtectedDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→保护)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 }
```

8.2 继承的工作方式

```
1 class ProtectedDerived : protected Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "ProtectedDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→保护)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class ProtectedDerived : protected Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "ProtectedDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→保护)  
6         protectedFunc();       // ✓ 可访问 (保护→保护)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PrivateDerived : private Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PrivateDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→私有)  
6         protectedFunc();      // ✓ 可访问 (保护→私有)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PrivateDerived : private Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PrivateDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→私有)  
6         protectedFunc();       // ✓ 可访问 (保护→私有)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PrivateDerived : private Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PrivateDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→私有)  
6         protectedFunc();       // ✓ 可访问 (保护→私有)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PrivateDerived : private Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PrivateDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→私有)  
6         protectedFunc();       // ✓ 可访问 (保护→私有)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PrivateDerived : private Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PrivateDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→私有)  
6         protectedFunc();      // ✓ 可访问 (保护→私有)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PrivateDerived : private Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PrivateDerived访问基类成员：" << endl;  
5         publicFunc();           // ✓ 可访问 (公有→私有)  
6         protectedFunc();       // ✓ 可访问 (保护→私有)  
7         // privateFunc();      // ✗ 编译错误：基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 class PrivateDerived : private Base {  
2 public:  
3     void accessBaseMembers() {  
4         cout << "PrivateDerived访问基类成员: " << endl;  
5         publicFunc();           // ✓ 可访问 (公有→私有)  
6         protectedFunc();       // ✓ 可访问 (保护→私有)  
7         // privateFunc();      // ✗ 编译错误: 基类私有成员不可访问  
8     }  
9 };
```

8.2 继承的工作方式

```
1 int main() {
2     // 公有继承对象
3     PublicDerived pubObj;
4     // ✓ 外部可访问基类公有成员
5     pubObj.publicFunc();
6     // ✗ 外部不可访问基类保护成员
7     // pubObj.protectedFunc();
8
9     // 保护继承对象
10    // ✗ 外部不可访问 (基类公有成员在派生类中变为保护)
11    ProtectedDerived proObj;
12    // proObj.publicFunc();
13
14    // 私有继承对象
15    PrivateDerived priObj;
```

8.2 继承的工作方式

```
1 int main() {
2     // 公有继承对象
3     PublicDerived pubObj;
4     // ✓ 外部可访问基类公有成员
5     pubObj.publicFunc();
6     // ✗ 外部不可访问基类保护成员
7     // pubObj.protectedFunc();
8
9     // 保护继承对象
10    // ✗ 外部不可访问 (基类公有成员在派生类中变为保护)
11    ProtectedDerived proObj;
12    // proObj.publicFunc();
13
14    // 私有继承对象
15    PrivateDerived priObj;
```

8.2 继承的工作方式

```
3     PublicDerived pubObj;
4     // ✓ 外部可访问基类公有成员
5     pubObj.publicFunc();
6     // ✗ 外部不可访问基类保护成员
7     // pubObj.protectedFunc();
8
9     // 保护继承对象
10    // ✗ 外部不可访问 (基类公有成员在派生类中变为保护)
11    ProtectedDerived proObj;
12    // proObj.publicFunc();
13
14    // 私有继承对象
15    PrivateDerived priObj;
16    // ✗ 外部不可访问 (基类公有成员在派生类中变为私有)
17    // priObj.publicFunc();
18    // privateFunc priObj,
```

8.2 继承的工作方式

```
6      // ✗ 外部不可访问基类保护成员
7      // pubObj.protectedFunc();
8
9      // 保护继承对象
10     // ✗ 外部不可访问 (基类公有成员在派生类中变为保护)
11     ProtectedDerived proObj;
12     // proObj.publicFunc();
13
14     // 私有继承对象
15     PrivateDerived priObj;
16     // ✗ 外部不可访问 (基类公有成员在派生类中变为私有)
17     // priObj.publicFunc();
18
19     return 0;
20 }
```

8.3 继承与组合

```
1 // 继承示例 (is-a关系)
2 class Vehicle { // 基类
3 public:
4     virtual void start() { cout << "Vehicle启动" << endl; }
5 };
6
7 class Car : public Vehicle { // 公有继承
8 public:
9     void start() override { // 重写基类方法
10         cout << "Car启动: 踩下离合器" << endl;
11     }
12     void drive() { cout << "Car行驶中" << endl; }
13 };
14
15 // 组合示例 (has-a关系)
```

8.3 继承与组合

```
1 // 继承示例 (is-a关系)
2 class Vehicle { // 基类
3 public:
4     virtual void start() { cout << "Vehicle启动" << endl; }
5 };
6
7 class Car : public Vehicle { // 公有继承
8 public:
9     void start() override { // 重写基类方法
10         cout << "Car启动: 踩下离合器" << endl;
11     }
12     void drive() { cout << "Car行驶中" << endl; }
13 };
14
15 // 组合示例 (has-a关系)
```

8.3 继承与组合

```
3 public:  
4     virtual void start() { cout << "Vehicle启动" << endl; }  
5 };  
6  
7 class Car : public Vehicle { // 公有继承  
8 public:  
9     void start() override { // 重写基类方法  
10         cout << "Car启动: 踩下离合器" << endl;  
11     }  
12     void drive() { cout << "Car行驶中" << endl; }  
13 };  
14  
15 // 组合示例 (has-a关系)  
16 class Engine { // 独立部件类  
17 public:
```

8.3 继承与组合

```
10         cout << "Car启动: 踩下离合器" << endl;
11     }
12     void drive() { cout << "Car行驶中" << endl; }
13 }
14
15 // 组合示例 (has-a关系)
16 class Engine { // 独立部件类
17 public:
18     void startEngine() { cout << "Engine点火" << endl; }
19 }
20
21 class ElectricCar { // 组合类
22 private:
23     Engine engine; // 内嵌Engine对象
24     Vehicle vehicle; // 组合其他类的对象
```

8.3 继承与组合

```
17 public:
18     void startEngine() { cout << "Engine点火" << endl; }
19 };
20
21 class ElectricCar { // 组合类
22 private:
23     Engine engine; // 内嵌Engine对象
24     Vehicle vehicle; // 组合其他类的对象
25 public:
26     void start() {
27         engine.startEngine(); // 调用部件功能
28         vehicle.start(); // 复用基类逻辑
29         cout << "ElectricCar启动完成" << endl;
30     }
31 };
32 //
```

8.3 继承与组合

继承与组合的相同点

8.3 继承与组合

继承与组合的相同点

- 两种方式都能复用已有类的功能：

8.3 继承与组合

继承与组合的相同点

- 两种方式都能复用已有类的功能：
 - 继承通过派生类直接调用基类方法（如Car调用Vehicle::start()）

8.3 继承与组合

继承与组合的相同点

- 两种方式都能复用已有类的功能：
 - 继承通过派生类直接调用基类方法（如Car调用Vehicle::start()）
 - 组合通过持有对象调用其接口（如ElectricCar调用Engine::startEngine()）

8.3 继承与组合

继承与组合的不同点

特性	继承	组合
关系类型	is-a	has-a
耦合度	高耦合（子类依赖父类实现细节，父类修改可能影响子类）	低耦合（仅通过接口交互，部件类内部修改不影响组合类）
复用方式	白盒复用（可覆盖父类方法）	黑盒复用（仅使用部件类的公开接口）
动态性	编译时确定，无法运行时切换父类	运行时可通过指针/引用动态更换部件（如更换不同类型的引擎）
设计原则	违反单一职责原则（子类可能继承不相关功能）	符合单一职责原则（各部件独立封装）

8.3 继承与组合

好的代码风格

8.3 继承与组合

好的代码风格

组合优于继承！

8.3 继承与组合

8.3 继承与组合

- 以下场景优先选择继承：

8.3 继承与组合

- 以下场景优先选择继承：
 - 需要扩展基类功能（如Car重写start()方法）

8.3 继承与组合

- 以下场景优先选择继承：
 - 需要扩展基类功能（如Car重写start()方法）
 - 需要多态特性（如通过基类指针操作不同派生类对象）

8.3 继承与组合

8.3 继承与组合

- 以下场景优先选择组合：

8.3 继承与组合

- 以下场景优先选择组合：
 - 需要动态更换组件（如支持燃油引擎和电动机自由切换）

8.3 继承与组合

- 以下场景优先选择组合：
 - 需要动态更换组件（如支持燃油引擎和电动机自由切换）
 - 避免过度层级嵌套（组合可替代多重继承的复杂场景）

8.4 多继承如何工作

- 多个不同父类派生同一个子类
- 或者说一个子类继承多个不同父类
- 根据组合与继承的关系同，既然类可以有多个成员对象，那么类也可以继承多个父类使其拥有多个父类的特质

8.4 多继承如何工作

语法：

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

8.4 多继承如何工作

```
1 #include <iostream>
2 using namespace std;
3
4 // 基类1: 飞行能力
5 class Flyable {
6 public:
7     void fly() {
8         cout << "使用喷气引擎飞行" << endl;
9     }
10};
11
12 // 基类2: 游泳能力
13 class Swimmable {
14 public:
15     void swim() {
```

8.4 多继承如何工作

```
1 #include <iostream>
2 using namespace std;
3
4 // 基类1: 飞行能力
5 class Flyable {
6 public:
7     void fly() {
8         cout << "使用喷气引擎飞行" << endl;
9     }
10};
11
12 // 基类2: 游泳能力
13 class Swimmable {
14 public:
15     void swim() {
```

8.4 多继承如何工作

```
8         cout << "使用喷气引擎飞行" << endl;
9     }
10 }
11
12 // 基类2: 游泳能力
13 class Swimmable {
14 public:
15     void swim() {
16         cout << "使用螺旋桨航行" << endl;
17     }
18 }
19
20 // 派生类: 多继承两个基类
21 class AmphibiousAircraft : public Flyable, public Swimmable
22 public:
```

8.4 多继承如何工作

```
17     }
18 }
19
20 // 派生类：多继承两个基类
21 class AmphibiousAircraft : public Flyable, public Swimmable
22 public:
23     void showMode() {
24         cout << "≡ 切换模式 ≡" << endl;
25         fly(); // 调用第一个基类方法
26         swim(); // 调用第二个基类方法
27     }
28 }
29
30 int main() {
31     AmphibiousAircraft aa;
```

8.4 多继承如何工作

```
23     void ShowMode() {
24         cout << "==== 切换模式 ===" << endl;
25         fly(); // 调用第一个基类方法
26         swim(); // 调用第二个基类方法
27     }
28 }
29
30 int main() {
31     AmphibiousAircraft aa;
32
33     // 直接访问不同基类的成员函数
34     aa.fly(); // 输出：使用喷气引擎飞行
35     aa.swim(); // 输出：使用螺旋桨航行
36
37     // 调用派生类自有方法
38     aa.ShowMode();
```

8.4 多继承如何工作

```
27      }
28  };
29
30 int main() {
31     AmphibiousAircraft aa;
32
33     // 直接访问不同基类的成员函数
34     aa.fly();      // 输出：使用喷气引擎飞行
35     aa.swim();    // 输出：使用螺旋桨航行
36
37     // 调用派生类自有方法
38     aa.showMode();
39
40     return 0;
41 }
```

8.4 多继承如何工作

```
27     ;
28 }
29
30 int main() {
31     AmphibiousAircraft aa;
32
33     // 直接访问不同基类的成员函数
34     aa.fly();      // 输出：使用喷气引擎飞行
35     aa.swim();    // 输出：使用螺旋桨航行
36
37     // 调用派生类自有方法
38     aa.showMode();
39
40     return 0;
41 }
```



8.5 多继承的模糊性

8.5 多继承的模糊性

- 多继承必须直面父类名字冲突问题

8.5 多继承的模糊性

- 多继承必须直面父类名字冲突问题
- 子类来自父类A的成员函数与父类B的成员函数调用形式可能完全相同

8.5 多继承的模糊性

- 多继承必须直面父类名字冲突问题
- 子类来自父类A的成员函数与父类B的成员函数调用形式可能完全相同
- 应用编程不得不了解多重继承细节,与面向对象编程的宗旨背道而驰

8.5 多继承的模糊性

```
1 #include <iostream>
2 using namespace std;
3
4 // 基类1: 电子设备
5 class ElectronicDevice {
6 public:
7     void powerOn() { // 同名成员函数
8         cout << "电子设备电源启动" << endl;
9     }
10};
11
12 // 基类2: 机械装置
13 class MechanicalDevice {
14 public:
15     void powerOn() { // 同名成员函数
```

8.5 多继承的模糊性

```
1 #include <iostream>
2 using namespace std;
3
4 // 基类1: 电子设备
5 class ElectronicDevice {
6 public:
7     void powerOn() { // 同名成员函数
8         cout << "电子设备电源启动" << endl;
9     }
10};
11
12 // 基类2: 机械装置
13 class MechanicalDevice {
14 public:
15     void powerOn() { // 同名成员函数
```

8.5 多继承的模糊性

```
8         cout << "电子设备电源启动" << endl;
9     }
10 }
11
12 // 基类2: 机械装置
13 class MechanicalDevice {
14 public:
15     void powerOn() { // 同名成员函数
16         cout << "机械装置动力激活" << endl;
17     }
18 }
19
20 // 派生类: 智能机器人 (多继承)
21 class Robot : public ElectronicDevice, public MechanicalDevice {
22 public:
23     void powerOn() { // 二义性入口
```

8.5 多继承的模糊性

```
1>
20 // 派生类：智能机器人（多继承）
21 class Robot : public ElectronicDevice, public MechanicalDevice
22 public:
23     void startup() {
24         // 直接调用会产生二义性
25         // powerOn(); // ✗ 编译错误：'powerOn' is ambiguous
26
27         // 解决方案1：使用作用域限定符
28         ElectronicDevice::powerOn(); // 明确调用指定基类版本
29         MechanicalDevice::powerOn();
30
31         // 解决方案2：在派生类中重定义统一接口
32         combinedPowerOn();
33     }
34 }
```

8.5 多继承的模糊性

```
30
31     // 解决方案2: 在派生类中重定义统一接口
32     combinedPowerOn();
33 }
34
35 // 解决方案2: 创建统一接口
36 void combinedPowerOn() {
37     cout << "==== 系统启动 ===" << endl;
38     ElectronicDevice::powerOn();
39     MechanicalDevice::powerOn();
40 }
41 }
42
43 int main() {
44     // 基础多继承冲突解决
45     // Duplicating base classes // 基础多继承入
```

8.5 多继承的模糊性

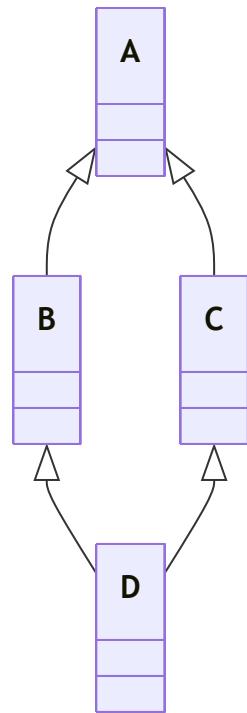
```
37     code << "启动成功" << endl;
38     ElectronicDevice::powerOn();
39     MechanicalDevice::powerOn();
40 }
41 }
42
43 int main() {
44     // 基础多继承冲突解决
45     Robot bot;
46     bot.startup();
47
48     // 外部调用时也需要明确作用域
49     bot.ElectronicDevice::powerOn();
50     bot.MechanicalDevice::powerOn();
51
52 }
```

8.5 多继承的模糊性

```
39         MechanicalDevice::powerOn();
40     }
41 }
42
43 int main() {
44     // 基础多继承冲突解决
45     Robot bot;
46     bot.startup();
47
48     // 外部调用时也需要明确作用域
49     bot.ElectronicDevice::powerOn();
50     bot.MechanicalDevice::powerOn();
51
52     return 0;
53 }
```

8.5 多继承的模糊性

菱形继承问题



8.5 多继承的模糊性

```
1 #include <iostream>
2 using namespace std;
3
4 // 公共基类
5 class Animal {
6 public:
7     int age;
8     void eat() { cout << "Animal eating" << endl; }
9 };
10
11 // 第一继承路径
12 class Mammal : public Animal {
13 public:
14     void breathe() { cout << "Mammal breathing" << endl; }
15 };
```

8.5 多继承的模糊性

```
1 #include <iostream>
2 using namespace std;
3
4 // 公共基类
5 class Animal {
6 public:
7     int age;
8     void eat() { cout << "Animal eating" << endl; }
9 };
10
11 // 第一继承路径
12 class Mammal : public Animal {
13 public:
14     void breathe() { cout << "Mammal breathing" << endl; }
15 };
```

8.5 多继承的模糊性

```
6 public:  
7     int age;  
8     void eat() { cout << "Animal eating" << endl; }  
9 };  
10  
11 // 第一继承路径  
12 class Mammal : public Animal {  
13 public:  
14     void breathe() { cout << "Mammal breathing" << endl; }  
15 };  
16  
17 // 第二继承路径  
18 class Bird : public Animal {  
19 public:  
20     void fly() { cout << "Bird flying" << endl; }
```

8.5 多继承的模糊性

```
12 class Mammal : public Animal {  
13 public:  
14     void breathe() { cout << "Mammal breathing" << endl; }  
15 };  
16  
17 // 第二继承路径  
18 class Bird : public Animal {  
19 public:  
20     void fly() { cout << "Bird flying" << endl; }  
21 };  
22  
23 // 菱形继承的派生类  
24 class Platypus : public Mammal, public Bird {  
25 public:  
26     void layEggs() { cout << "Platypus laying eggs" << endl;
```

8.5 多继承的模糊性

```
18 class Bird : public Animal {
19 public:
20     void fly() { cout << "Bird flying" << endl; }
21 };
22
23 // 菱形继承的派生类
24 class Platypus : public Mammal, public Bird {
25 public:
26     void layEggs() { cout << "Platypus laying eggs" << endl;
27 };
28
29 int main() {
30     Platypus perry;
31
32     , // 产生二义性的访问
```

8.5 多继承的模糊性

```
23 // 菱形继承的派生类
24 class Platypus : public Mammal, public Bird {
25 public:
26     void layEggs() { cout << "Platypus laying eggs" << endl;
27 }
28
29 int main() {
30     Platypus perry;
31
32     // 产生二义性的访问
33     // perry.age = 2; // ✗ 编译错误: ambiguous access of 'age'
34     // perry.eat(); // ✗ 编译错误: ambiguous access of 'eat'
35
36     // 显式指定路径可以解决
37     , perry.Mammal::age = 2; // ✓ 明确使用Mammal路径的age
```

8.5 多继承的模糊性

```
26     void layEggs() { cout << "Platypus laying eggs" << endl;
27 }
28
29 int main() {
30     Platypus perry;
31
32     // 产生二义性的访问
33     // perry.age = 2; // ✗ 编译错误: ambiguous access of 'age'
34     // perry.eat(); // ✗ 编译错误: ambiguous access of 'eat'
35
36     // 显式指定路径可以解决
37     perry.Mammal::age = 2;    // ✓ 明确使用Mammal路径的age
38     perry.Bird::eat();       // ✓ 明确使用Bird路径的eat
39
40     , // 验证两个基类副本的存在
```

8.5 多继承的模糊性

```
30     Platypus perry;
31
32     // 产生二义性的访问
33     // perry.age = 2; // ✗ 编译错误: ambiguous access of 'age'
34     // perry.eat(); // ✗ 编译错误: ambiguous access of 'eat'
35
36     // 显式指定路径可以解决
37     perry.Mammal::age = 2;    // ✓ 明确使用Mammal路径的age
38     perry.Bird ::eat();      // ✓ 明确使用Bird路径的eat
39
40     // 验证两个基类副本的存在
41     cout << "Mammal's age: " << perry.Mammal::age << endl;
42     perry.Mammal::age = 3;
43     // 输出不同值
44     cout << "Bird's age: " << perry.Bird::age << endl;
```

8.5 多继承的模糊性

```
55      // perry.age = 2; // ✗ 编译错误: ambiguous access of 'age'
34      // perry.eat(); // ✗ 编译错误: ambiguous access of 'eat'
35
36      // 显式指定路径可以解决
37      perry.Mammal::age = 2; // ✓ 明确使用Mammal路径的age
38      perry.Bird::eat();    // ✓ 明确使用Bird路径的eat
39
40      // 验证两个基类副本的存在
41      cout << "Mammal's age: " << perry.Mammal::age << endl;
42      perry.Mammal::age = 3;
43      // 输出不同值
44      cout << "Bird's age: " << perry.Bird::age << endl;
45
46      return 0;
47 }
```

8.6 虚继承

菱形继承的核心问题：

- 派生类包含两份Animal成员副本
- 直接访问公共基类成员时编译器无法确定使用哪个副本

8.6 虚继承

```
1 // 修改继承方式为虚继承
2 class Mammal : virtual public Animal {}; // 虚继承
3 class Bird : virtual public Animal {}; // 虚继承
4
5 class Platypus : public Mammal, public Bird {
6 public:
7     void layEggs() {
8         age = 2; // ✓ 可以直接访问 (唯一副本)
9         eat(); // ✓ 无二义性
10    }
11 }
12
13 int main() {
14     Platypus perry;
15     perry.age = 5; // ✓ 正常访问
```

8.6 虚继承

```
1 // 修改继承方式为虚继承
2 class Mammal : virtual public Animal {}; // 虚继承
3 class Bird : virtual public Animal {}; // 虚继承
4
5 class Platypus : public Mammal, public Bird {
6 public:
7     void layEggs() {
8         age = 2; // ✓ 可以直接访问 (唯一副本)
9         eat(); // ✓ 无二义性
10    }
11 }
12
13 int main() {
14     Platypus perry;
15     perry.age = 5; // ✓ 正常访问
```

8.6 虚继承

```
1 // 修改继承方式为虚继承
2 class Mammal : virtual public Animal {}; // 虚继承
3 class Bird : virtual public Animal {}; // 虚继承
4
5 class Platypus : public Mammal, public Bird {
6 public:
7     void layEggs() {
8         age = 2; // ✓ 可以直接访问 (唯一副本)
9         eat(); // ✓ 无二义性
10    }
11 }
12
13 int main() {
14     Platypus perry;
15     perry.age = 5; // ✓ 正常访问
```

8.6 虚继承

```
4
5 class Platypus : public Mammal, public Bird {
6 public:
7     void layEggs() {
8         age = 2;           // ✓ 可以直接访问 (唯一副本)
9         eat();            // ✓ 无二义性
10    }
11 }
12
13 int main() {
14     Platypus perry;
15     perry.age = 5;      // ✓ 正常访问
16     cout << perry.age; // 输出5 (唯一值)
17     return 0;
18 }
```

8.6 虚继承

```
1 class A {};
2 class B1 : public A {};
3 class B2 : public A {};
4
5 class C1 : public B1, public B2 {
6     // B1 is real base, B2 is fake base.
7 }
8
9 class C2 : public B2, public B1 {
10    // B2 is real base, B1 is fake base.
11 }
```

8.6 虚继承

```
1 class A {};
2 class B1 : public A {};
3 class B2 : public A {};
4
5 class C1 : public B1, public B2 {
6     // B1 is real base, B2 is fake base.
7 }
8
9 class C2 : public B2, public B1 {
10    // B2 is real base, B1 is fake base.
11 }
```

8.6 虚继承

```
1 class A {};
2 class B1 : public A {};
3 class B2 : public A {};
4
5 class C1 : public B1, public B2 {
6     // B1 is real base, B2 is fake base.
7 }
8
9 class C2 : public B2, public B1 {
10    // B2 is real base, B1 is fake base.
11 }
```

8.6 虚继承

```
1 class A {};
2 class B1 : public A {};
3 class B2 : public A {};
4
5 class C1 : public B1, public B2 {
6     // B1 is real base, B2 is fake base.
7 }
8
9 class C2 : public B2, public B1 {
10    // B2 is real base, B1 is fake base.
11 }
```

8.7 多继承的构造顺序

8.7 多继承的构造顺序

- 首先完成虚继承的构造

8.7 多继承的构造顺序

- 首先完成虚继承的构造
- 其他则按继承规则

8.7 多继承的构造顺序

- 首先完成虚继承的构造
- 其他则按继承规则
 - 首先构造非虚基类

8.7 多继承的构造顺序

- 首先完成虚继承的构造
- 其他则按继承规则
 - 首先构造非虚基类
 - 再构造成员对象

8.7 多继承的构造顺序

- 首先完成虚继承的构造
- 其他则按继承规则
 - 首先构造非虚基类
 - 再构造成员对象
 - 最后构造派生类自己的数据

8.7 多继承的构造顺序

- 首先完成虚继承的构造
- 其他则按继承规则
 - 首先构造非虚基类
 - 再构造成员对象
 - 最后构造派生类自己的数据
- 若在同一层中包含多个虚基类，那么虚基类的构造按它们说明的次序调用

8.7 多继承的构造顺序

- 首先完成虚继承的构造
- 其他则按继承规则
 - 首先构造非虚基类
 - 再构造成员对象
 - 最后构造派生类自己的数据
- 若在同一层中包含多个虚基类，那么虚基类的构造按它们说明的次序调用
- 析构的顺序与之相反

8.7 多继承的构造顺序

```
1 #include <iostream>
2 using namespace std;
3
4 // 基类
5 class Base {
6 public:
7     Base() { cout << "Base 构造函数" << endl; }
8     ~Base() { cout << "Base 析构函数" << endl; }
9 };
10
11 // 普通继承的中间类1
12 class Derived1 : public Base {
13 public:
14     Derived1() { cout << "Derived1 构造函数" << endl; }
15     ~Derived1() { cout << "Derived1 析构函数" << endl; }
```

8.7 多继承的构造顺序

```
1 #include <iostream>
2 using namespace std;
3
4 // 基类
5 class Base {
6 public:
7     Base() { cout << "Base 构造函数" << endl; }
8     ~Base() { cout << "Base 析构函数" << endl; }
9 };
10
11 // 普通继承的中间类1
12 class Derived1 : public Base {
13 public:
14     Derived1() { cout << "Derived1 构造函数" << endl; }
15     ~Derived1() { cout << "Derived1 析构函数" << endl; }
```

8.7 多继承的构造顺序

```
6 public:
7     Base() { cout << "Base 构造函数" << endl; }
8     ~Base() { cout << "Base 析构函数" << endl; }
9 };
10
11 // 普通继承的中间类1
12 class Derived1 : public Base {
13 public:
14     Derived1() { cout << "Derived1 构造函数" << endl; }
15     ~Derived1() { cout << "Derived1 析构函数" << endl; }
16 };
17
18 // 普通继承的中间类2
19 class Derived2 : public Base {
20 public:
21     Derived2() { cout << "Derived2 构造函数" << endl; }
```

8.7 多继承的构造顺序

```
13 public:
14     Derived1() { cout << "Derived1 构造函数" << endl; }
15     ~Derived1() { cout << "Derived1 析构函数" << endl; }
16 };
17
18 // 普通继承的中间类2
19 class Derived2 : public Base {
20 public:
21     Derived2() { cout << "Derived2 构造函数" << endl; }
22     ~Derived2() { cout << "Derived2 析构函数" << endl; }
23 };
24
25 // 虚继承的中间类1
26 class VirtualDerived1 : virtual public Base {
27 public:
28     VirtualDerived1() { cout << "VirtualDerived1 构造函数" << endl; }
```

8.7 多继承的构造顺序

```
21     Derived2() { cout << "Derived2 构造函数" << endl; }
22     ~Derived2() { cout << "Derived2 析构函数" << endl; }
23 }
24
25 // 虚继承的中间类1
26 class VirtualDerived1 : virtual public Base {
27 public:
28     VirtualDerived1() { cout << "VirtualDerived1 构造函数" <<
29     ~VirtualDerived1() {
30         cout << "VirtualDerived1 析构函数" << endl;
31     }
32 }
33
34 // 虚继承的中间类2
35 class VirtualDerived2 : virtual public Base {
36     VirtualDerived2() { cout << "VirtualDerived2 构造函数" <<
```

8.7 多继承的构造顺序

```
30     cout << "VirtualDerived1 构造函数" << endl;
31 }
32 }
33
34 // 虚继承的中间类2
35 class VirtualDerived2 : virtual public Base {
36 public:
37     VirtualDerived2() { cout << "VirtualDerived2 构造函数" <<
38     ~VirtualDerived2() {
39         cout << "VirtualDerived2 析构函数" << endl;
40     }
41 }
42
43 // 普通菱形继承的最终派生类
44 class Diamond : public Derived1, public Derived2 {
45     cout << "Diamond 构造函数" << endl;
46 }
```

8.7 多继承的构造顺序

```
38     ~VirtualDerived2() {
39         cout << "VirtualDerived2 析构函数" << endl;
40     }
41 }
42
43 // 普通菱形继承的最终派生类
44 class Diamond : public Derived1, public Derived2 {
45 public:
46     Diamond() { cout << "Diamond 构造函数" << endl; }
47     ~Diamond() { cout << "Diamond 析构函数" << endl; }
48 }
49
50 // 使用虚继承的最终派生类
51 class VirtualDiamond : public VirtualDerived1, public VirtualDerived2 {
52 public:
53     VirtualDiamond() { cout << "VirtualDiamond 构造函数" << endl; }
```

8.7 多继承的构造顺序

```
17 // Diamond9.cpp -- Diamond 构造函数
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48 };
49
50 // 使用虚继承的最终派生类
51 class VirtualDiamond : public VirtualDerived1, public Virtual
52 public:
53     VirtualDiamond() : Base() { // 显式调用虚基类构造函数
54         cout << "VirtualDiamond 构造函数" << endl;
55     }
56     ~VirtualDiamond() {
57         cout << "VirtualDiamond 析构函数" << endl;
58     }
59 }
60
61 int main() {
62 }
```

8.7 多继承的构造顺序

```
57         cout << "VirtualDiamond 析构函数" << endl;
58     }
59 }
60
61 int main() {
62     cout << "===== 普通菱形继承 =====\n";
63     Diamond d;
64
65     cout << "\n===== 虚继承菱形继承 =====\n";
66     VirtualDiamond vd;
67
68     cout << "\n===== 析构函数调用 =====\n";
69
70     return 0;
71 }
```

8.7 多继承的构造顺序

输出内容

```
1 ===== 普通菱形继承 =====  
2 Base 构造函数  
3 Derived1 构造函数  
4 Base 构造函数  
5 Derived2 构造函数  
6 Diamond 构造函数  
7  
8 ===== 虚继承菱形继承 =====  
9 Base 构造函数  
10 VirtualDerived1 构造函数  
11 VirtualDerived2 构造函数  
12 VirtualDiamond 构造函数  
13  
14 ===== 析构函数调用 =====  
15 VirtualDiamond 析构函数
```

8.7 多继承的构造顺序

输出内容

```
1 ===== 普通菱形继承 =====  
2 Base 构造函数  
3 Derived1 构造函数  
4 Base 构造函数  
5 Derived2 构造函数  
6 Diamond 构造函数  
7  
8 ===== 虚继承菱形继承 =====  
9 Base 构造函数  
10 VirtualDerived1 构造函数  
11 VirtualDerived2 构造函数  
12 VirtualDiamond 构造函数  
13  
14 ===== 析构函数调用 =====  
15 VirtualDiamond 析构函数
```

8.7 多继承的构造顺序

输出内容

```
3 Derived1 构造函数
4 Base 构造函数
5 Derived2 构造函数
6 Diamond 构造函数
7
8 ===== 虚继承菱形继承 =====
9 Base 构造函数
10 VirtualDerived1 构造函数
11 VirtualDerived2 构造函数
12 VirtualDiamond 构造函数
13
14 ===== 析构函数调用 =====
15 VirtualDiamond 析构函数
16 VirtualDerived2 析构函数
17 VirtualDerived1 析构函数
```

8.7 多继承的构造顺序

输出内容

```
9 Base 构造函数
10 VirtualDerived1 构造函数
11 VirtualDerived2 构造函数
12 VirtualDiamond 构造函数
13
14 ===== 析构函数调用 =====
15 VirtualDiamond 析构函数
16 VirtualDerived2 析构函数
17 VirtualDerived1 析构函数
18 Base 析构函数
19 Diamond 析构函数
20 Derived2 析构函数
21 Base 析构函数
22 Derived1 析构函数
23 Base 析构函数
+-> 退出 C++ Builder -> [关闭]
```

课外阅读

现代的GUI编程框架，有很多已经摆脱了继承，而采用组合、函数式等方式来实现。

参考资料：

- Reactive (JavaScript 前端框架)
- ftxui (C++ 的现代化 TUI 框架)