

原创

# C++ | 继承

2019-01-14 15:31:14 [\\_YKitty](#) 阅读数 284 [更多](#)

分类专栏：[C++](#)

版权声明：本文为博主原创文章，遵循 [CC 4.0 BY-SA](#) 版权协议，转载请附上原文出处链接和本声明。

本文链接：[https://blog.csdn.net/qq\\_40399012/article/details/86478226](https://blog.csdn.net/qq_40399012/article/details/86478226)

## 继承

---

### 1. 概念

**继承 ( inheritance )** 是面向对象程序设计**使代码复用**的手段。它允许程序员在**保持原有类特性的基础上进行扩展**，从而产生新的类，称为**派生类也叫作子类**。**继承是设计层次的复用**

### 2. 定义

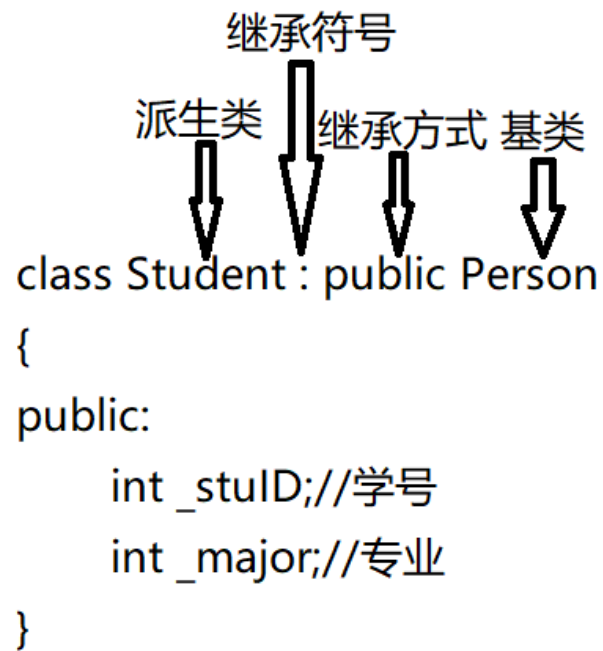
#### 2.1 定义格式

插图：定义图

继承符号

派生类      继承方式      基类

```
class Student : public Person
{
public:
    int _stuID;//学号
    int _major;//专业
}
```

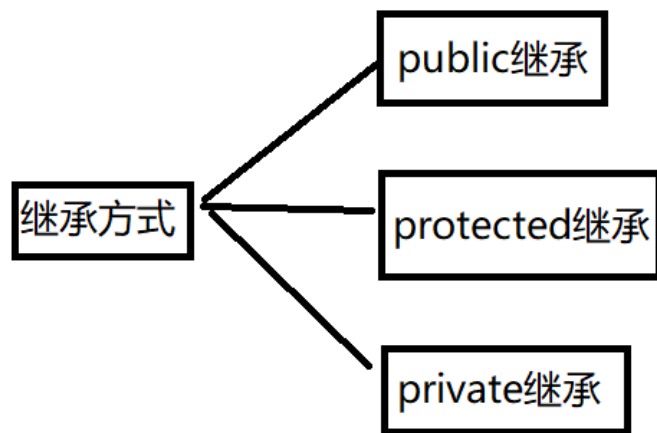


[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

**注意：**继承符号是：冒号

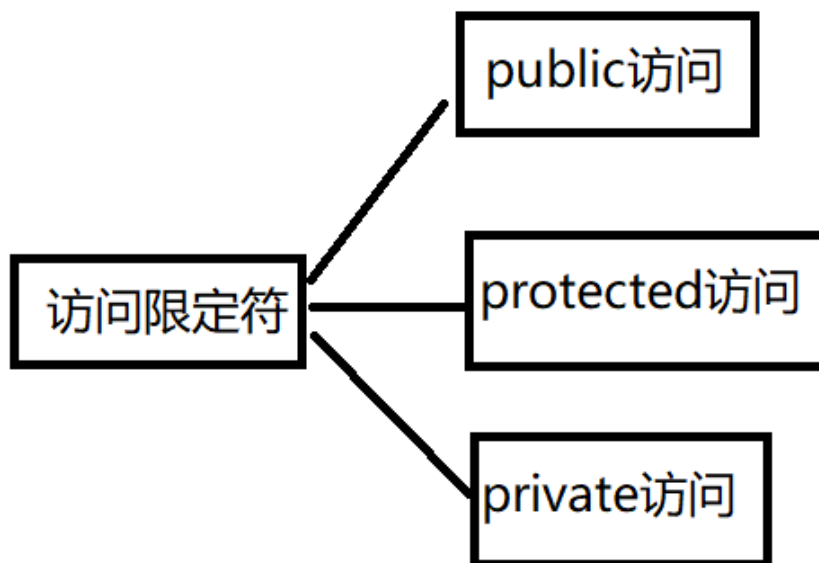
## 2.2 继承方式和访问限定符

插图：继承方式



[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

插图：访问限定符



[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

类成员/继承方式	public 继承	protected 继承	private 继承
基类的 public 成员	派生类的 public 成员	派生类的 protected 成员	派生类的 private 成员
基类的 protected 成员	派生类的 protected 成员	派生类 protected 成员	派生类的 private 成员
基类的 private 成员	在派生类中不可见	在派生类中不可见	在派生类中不可见

## 2.3 继承基类成员访问方式的变化

总结：

1. 基类的 **private** 成员在派生类中无论以什么方式继承都是不可见的。不可见：基类的私有成员依旧被继承到了派生类中，但是语法上限制了派生类对象不管在类外还是类内都不能去访问它
2. 如果基类的成员不想在类外被访问，但需要在派生类中能访问，就定义为 **protected**。保护限定符是因继承才出现的
3. 对于上面表格进行总结。基类的私有成员在子类都是不可见的。基类的其他成员在子类的访问方式 = Min（成员在基类的访问限定符，继承方式），**public > protected > private**
4. 关键字 **class** 默认的继承方式是 **private**，**struct** 默认的继承方式是 **public**
5. 在实际应用中一般使用的都时 **public** 继承，几乎很少使用 **protected** 和 **private** 继承。因为 **protected** 和 **private** 继承下来的成员只能在派生类中使用，实际中扩展维护性不高

## 3. 基类和派生类对象赋值转换

- 派生类的对象可以赋值给基类的对象，基类的指针，基类的引用。形象的说法是切片
- 基类对象不可以赋值给派生类对象
- 基类的指针可以通过强制转换赋值给派生类的指针。但是必须是基类的指针指向派生类的对象时才是安全的。

例：

```
1. #include <iostream>
2. #include <string>
3.
4. class Person
5. {
6. public:
7.     void Show()
8.     {
9.         std::cout << "In Basci!" << std::endl;
10.    }
11.
12.    //private:
13.    std::string _name;
14.    int _age;
15.};
16.
17.class Student : public Person
18.{
19.public:
20.    void Display()
21.    {
22.        std::cout << "In Derived!" << std::endl;
23.    }
24.
25.    //private:
26.    int _num;
27.};
28.
29.int main()
30.{
31.    //派生类的对象可以赋值给基类的对象，引用，指针
32.    Student stu;
33.    Person p = stu;
34.    Person& p1 = stu;
35.    Person* p2 = &stu;
```

```

36.
37.     // 基类的对象不可以赋值给派生类
38.     //stu = p;
39.
40.     // 基类的指针（指向派生类对象）可以赋值给派生类，必须要进行强制类型转换
41.     Student* pStu = (Student*)p2;
42.     pStu->_num = 10;
43.     std::cout << "ok" << std::endl;
44.
45.     // 基类的指针（指向基类的对象）可以赋值给派生类，但是会发生越界问题
46.     Person pp;
47.     pStu = (Student*)&pp;
48.     pStu->_num = 10;
49.     std::cout << "odk?" << std::endl;
50.     return 0;
51. }

```

## 4. 继承中的作用域

1. 在继承体系中基类和派生类都有着自己的独立作用域
2. 子类和父类有同名的成员的话，子类成员将屏蔽父类对于同名成员的直接访问，这种情况叫做隐藏，也叫作重定向。（在子类成员函数中，可以使用基类::基类成员显式访问）
3. 实际中在继承体系中最好不要定义同名的成员

例：

```

1. #include <iostream>
2. #include <string>
3.
4. class Person
5. {

```

```

6. protected:
7.     int _num = 111;
8.     std::string _name = "yk";
9. };
10.
11. class Student : public Person
12. {
13. public:
14.     void Show()
15.     {
16.         std::cout << "_num: " << _num << std::endl;
17.         std::cout << "_name: " << _name << std::endl;
18.     }
19.
20. private:
21.     int _num = 999;
22.     std::string _name = "oodk";
23. };
24.
25. int main()
26. {
27.     Student stu;
28.     stu.Show(); // 输出 999 oodk 表示基类的成员变量被隐藏了
29.     return 0;
30. }

```

**理解重载和隐藏：**

例：

```

1. #include <iostream>
2. #include <string>
3.
4. class A
5. {
6. public:
7.     void Show()
8.     {

```

```

9.         std::cout << "In A!" << std::endl;
10.     }
11. };
12.
13. class B : public A
14. {
15. public:
16.     void Show(int i)
17.     {
18.         std::cout << "In B!" << std::endl;
19.     }
20. };
21.
22. int main()
23. {
24.     //A 和B 中的Show 函数构成了隐藏，因为不在同一个作用域中所以不是重载
25.     B b;
26.     b.Show(3);
27.     b.A::Show();//调用父类中的 Show 函数必须要显式调用，因为被隐藏了
28.     return 0;
29. }

```

## 5. 派生类的默认成员函数

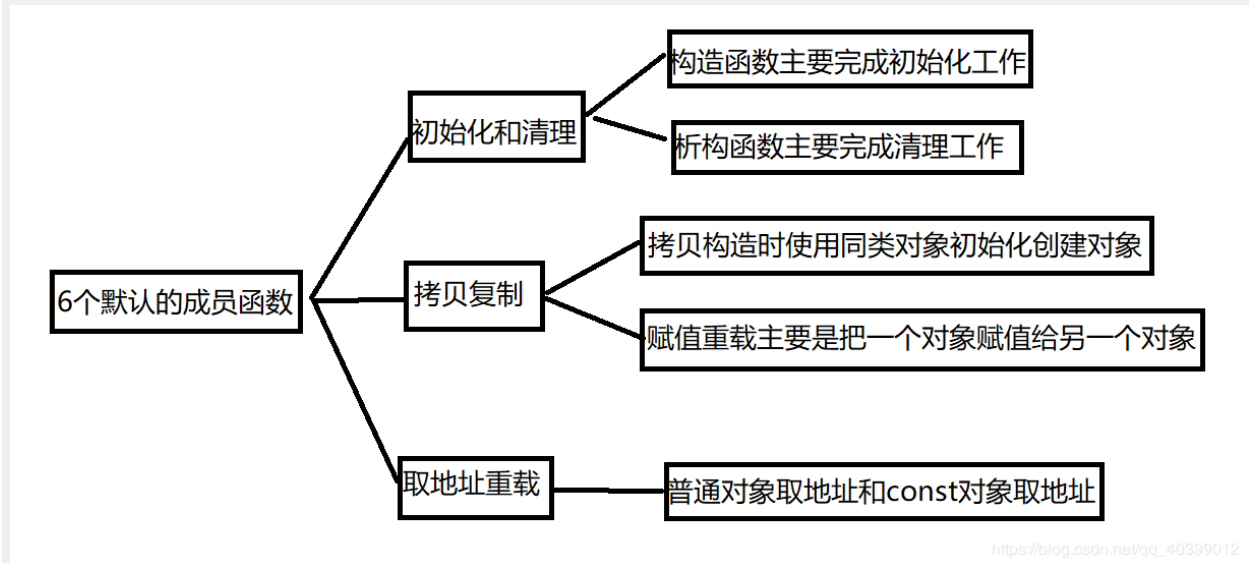
### 6 个默认的成员函数

1. 派生类的构造函数必须调用基类的构造函数初始化基类的那一部分成员。如果基类没有默认的构造函数（无参的构造函数），则必须在派生类的构造函数中初始化链表阶段显式调用（可以在初始化阶段的任何一个地方）
2. 派生类的拷贝构造函数必须调用基类的拷贝构造完成基类的拷贝初始化
3. 派生类的 operator= 必须要调用基类的 operator= 完成基类的复制

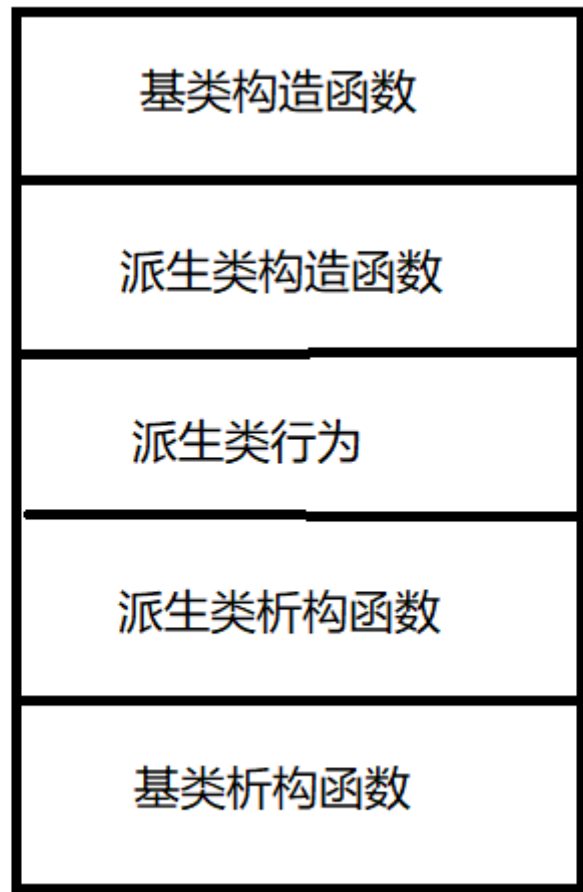


4. 派生类的析构函数会在被调用完成之后自动调用基类的析构函数清理基类成员。因为这样才能保证派生类的对象先清理派生类成员在清理基类成员的顺序

插图：六大默认成员函数



插图：默认成员函数调用顺序



[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

例：

```
1. #include <iostream>
2. #include <string>
3.
4.
5. class Person
6. {
7. public:
8.     Person(std::string name)
9.     : _name(name)
```

```

10.     {
11.         std::cout << "In Person Construction!" << std::endl;
12.     }
13.
14.     Person(const Person& p)
15.         : _name(p._name)
16.     {
17.         std::cout << "In Person Construction!=" << std::endl;
18.     }
19.
20.     Person& operator=(const Person& p)
21.     {
22.         if (this != &p)
23.         {
24.             _name = p._name;
25.             std::cout << "In Person operator!=" << std::endl;
26.         }
27.
28.         return *this;
29.     }
30.
31.     ~Person()
32.     {
33.         std::cout << "In Person Destruction!" << std::endl;
34.     }
35.
36. private:
37.     std::string _name;
38. };
39.
40.
41. class Student : public Person
42. {
43. public:
44.     Student(std::string name)
45.         : Person(name)

```

```

46.         , _name(name)
47.     {
48.         std::cout << "In Student Construction!" << std::endl;
49.     }
50.
51.     Student(const Student& p)
52.         : Person(p)
53.         , _name(p._name)
54.     {
55.         std::cout << "In Student Construction=!" << std::endl;
56.     }
57.
58.     Student& operator=(const Student& p)
59.     {
60.         if (this != &p)
61.         {
62.             // 必须先对基类的成员进行拷贝构造
63.             Person::operator=(p);
64.             _name = p._name;
65.             std::cout << "In Student operator=!" << std::endl;
66.         }
67.
68.         return *this;
69.     }
70.
71.     ~Student()
72.     {
73.         std::cout << "In Student Destruction!" << std::endl;
74.     }
75.
76. private:
77.     std::string _name;
78. };
79.
80. int main()

```

```

81.{
82.    Student s1("jack");
83.    Student s2(s1);
84.    Student s3("rose");
85.    s1 = s3;
86.    return 0;
87.}

```

### 【面试题】：实现一个不能被继承的类

- C++98：构造函数私有化，派生类调用不到基类的构造函数，则无法继承
- C++11：使用关键字 final（最终）

## 6. 继承与友元

友元关系不能继承,也就是说基类友元不能访问子类私有和保护成员

例：

```

1. class A
2. {
3. public:
4.     friend void Display(const A& a);
5. public:
6.     void Show()
7.     {
8.         std::cout << "In A!" << std::endl;
9.     }
10.
11. private:
12.     int _num = 34;
13. };
14.
15.
16. class B : public A
17. {

```

```

18. public:
19.     void Show()
20.     {
21.         std::cout << "In B!" << std::endl;
22.     }
23.
24. private:
25.     std::string _name = "yk";
26. };
27.
28. void Display(const A& a)
29. {
30.     std::cout << "Freind function In A!" << "_num" << a._num << std::endl;
31.     //std::cout << "Freind function In A!" << "_num" << b._name << std::endl;
32. }
33.
34. int main()
35. {
36.     A a;
37.     Display(a); //输出 Friend function In A!_num34
38.     return 0;
39. }

```

## 7. 继承与静态成员

基类定义了 static 静态成员，则整个继承体系中只有一个这样的成员。无论派生出多少个子类，都只有一个 static 成员实例

例：

```

1. class A
2. {
3. public:
4.     friend void Display(const A& a);

```

```
5. public:
6.     void Show()
7.     {
8.         std::cout << "_count" << _count << std::endl;
9.     }
10.
11.     A()
12.     {
13.         _count++;
14.     }
15.
16. private:
17.     int _num = 34;
18.
19. public:
20.     static int _count;
21. };
22.
23. int A::_count = 0;
24.
25.
26. class B : public A
27. {
28. public:
29.     void Show()
30.     {
31.         std::cout << "In B!" << std::endl;
32.     }
33.
34. private:
35.     std::string _name = "yk";
36. };
37.
38. class C : public A
39. {
40.
```

```

41. private:
42.     int _c = 0;
43. };
44.
45. void Display(const A& a)
46. {
47.     std::cout << "Freind function In A!" << "_count" << a._count <<
        std::endl;
48.     //std::cout << "Freind function In A!" << "_num" << b._name << std::endl;
49. }
50.
51. int main()
52. {
53.     B b1;
54.     B b2;
55.     B b3;
56.     Display(b3); //_count 3
57.
58.     C c;
59.     c.Show(); //_count 4
60.     return 0;
61. }

```

## 8. 复杂菱形继承和菱形虚拟继承

基于代码：

```

1. class Person
2. {
3.
4. private:
5.     std::string _name;
6. };
7.

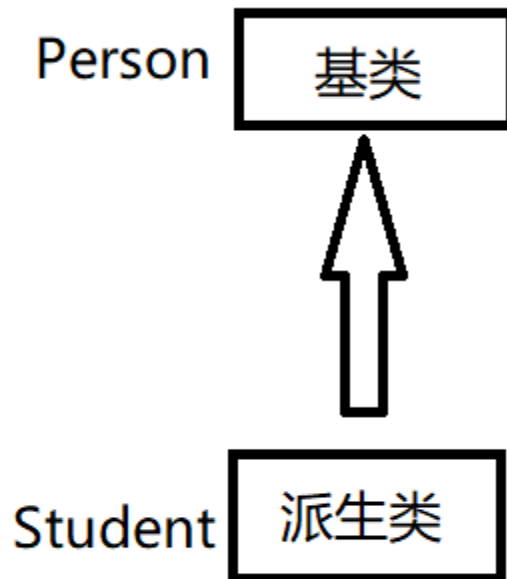
```



```
8. class Student : public Person
9. {
10.
11. private:
12.     int _num;
13. };
14.
15. class Teacher : public Person
16. {
17.
18. private:
19.     int _id;
20. };
21.
22. class Assistant : public Student, Teacher
23. {
24.
25. private:
26.     std::string _majorCourse;
27. };
```

**单继承**：一个子类只有一个直接父类

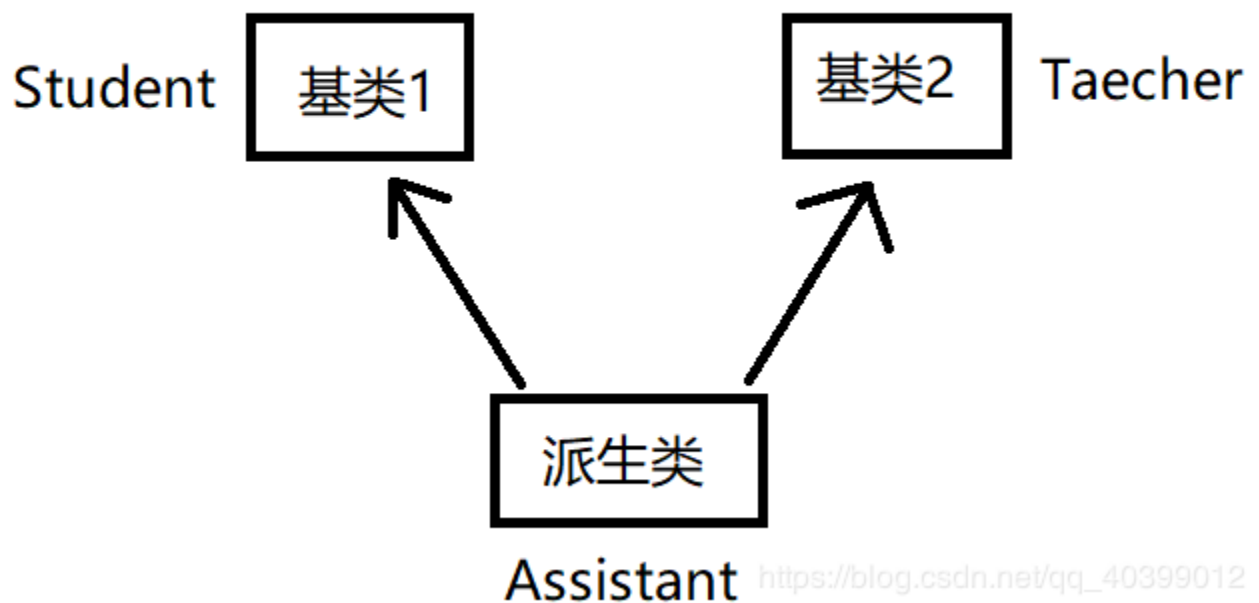
插图：



[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

**多继承：**一个子类有两个或者以上直接父类

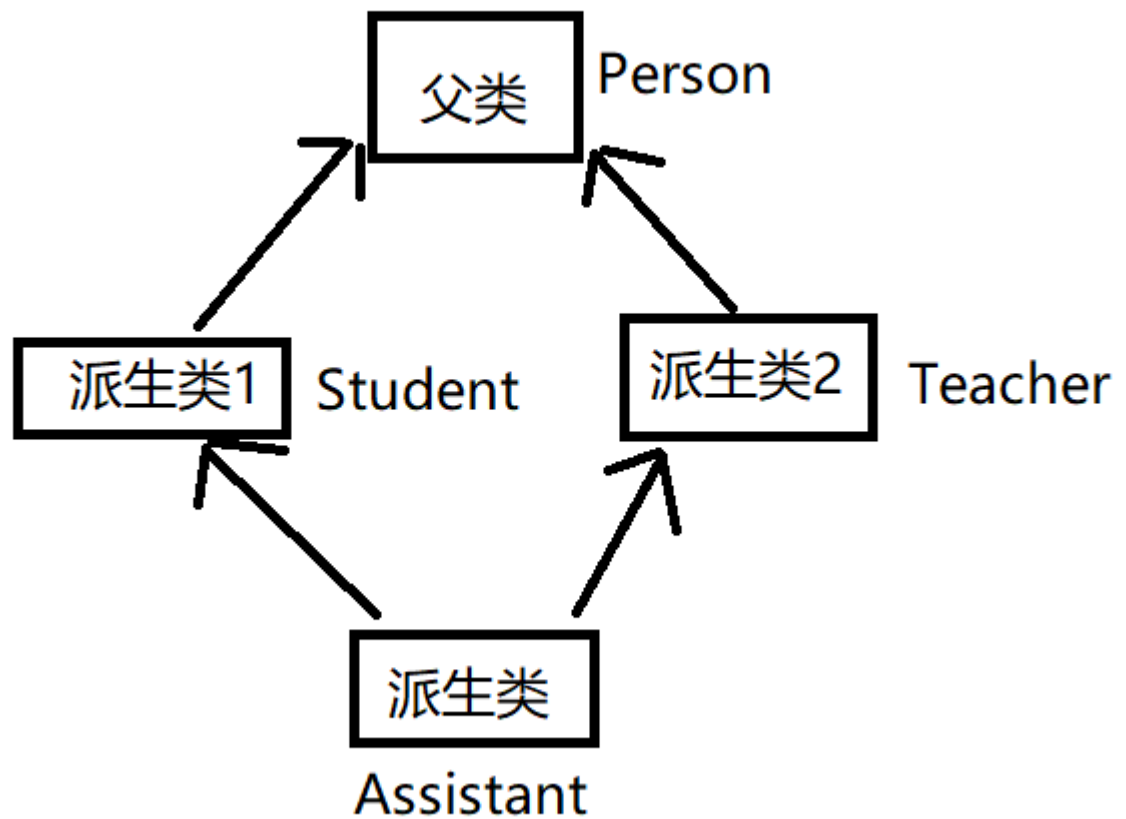
插图：



[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

**菱形继承：**菱形继承是多继承的一种特殊情况

插图：



[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

菱形继承的问题：菱形继承有数据冗余和二义性问题

例：

```
1. class Person
2. {
3.
4. public:
5.     std::string _name;
6. };
7.
8. class Student : public Person
9. {
10.
11. protected:
12.     int _num;
```

```

13.};
14.
15.class Teacher : public Person
16.{
17.
18.protected:
19.     int _id;
20.};
21.
22.class Assistant : public Student, public Teacher
23.{
24.
25.protected:
26.     std::string _majorCourse;
27.};
28.
29.int main()
30.{
31.     Assistant as;
32.     //as._name = "YK";//出现错误，因为数据二义性，必须显式调用
33.     as.Student::_name = "YK";
34.     as.Teacher::_name = "LW";
35.     return 0;
36.}

```

### 【解决方法】

**虚拟继承**可以解决菱形继承的二义性和数据冗余的问题。在上面的继承关系中，在 Student 和 Teacher 继承 Person 使用虚拟继承，即可解决问题。

**虚拟继承**就是让最终继承下来的子类之后只有一个\_name 数据，也就对不管对于改变 Student 还是 Teacher 都是将其全部都改变了。不会在出现改变 Student 中的 \_name，Teacher 中的 \_name 没有发生改变

例：

```

1. class Person
2. {
3.

```

```

4. public:
5.     std::string _name;
6. };
7.
8. class Student : virtual public Person
9. {
10.
11. protected:
12.     int _num;
13. };
14.
15. class Teacher : virtual public Person
16. {
17.
18. protected:
19.     int _id;
20. };
21.
22. class Assistant : public Student, public Teacher
23. {
24.
25. protected:
26.     std::string _majorCourse;
27. };
28.
29. int main()
30. {
31.     Assistant as;
32.     as._name = "YK"; //可以使用了, 由于只有一个_name 数据了, 不会在产生二义性的问题
33.     as.Student::_name = "YK";
34.     as.Teacher::_name = "LW";
35.     return 0;
36. }

```

### 虚拟菱形继承的原理：

使用以下例子来再借助内存窗口观察对象成员的模型

- 菱形继承的内部结构

```
1. class A
2. {
3.
4. public:
5.     int _a;
6. };
7.
8. class B : public A
9. {
10.
11. public:
12.     int _b;
13. };
14.
15. class C : public A
16. {
17.
18. public:
19.     int _c;
20. };
21.
22. class D : public B, public C
23. {
24.
25. public:
26.     int _d;
27. };
28.
29. int main()
30. {
31.     D d;
32.     d.B::_a = 2;
33.     d.C::_a = 1;
34.     d._b = 2;
```

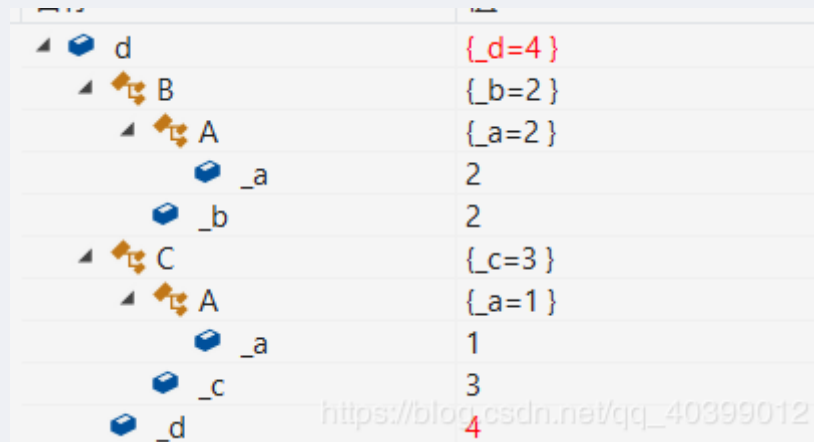
```

35.         d._c = 3;
36.         d._d = 4;
37.         return 0;;
38. }

```

这是没有虚拟继承，只是菱形继承，就会产生数据二义性问题。对于**菱形继承对于类 D 创建的对象**里面只有两个类，类 B 和类 C，但是类 B 和类 C 内部又有一个类 A，这样就会产生对于类 D 创建一个对象这个两个类 A，这样就会产生数据冗余和二义性问题。

插图：内存模型

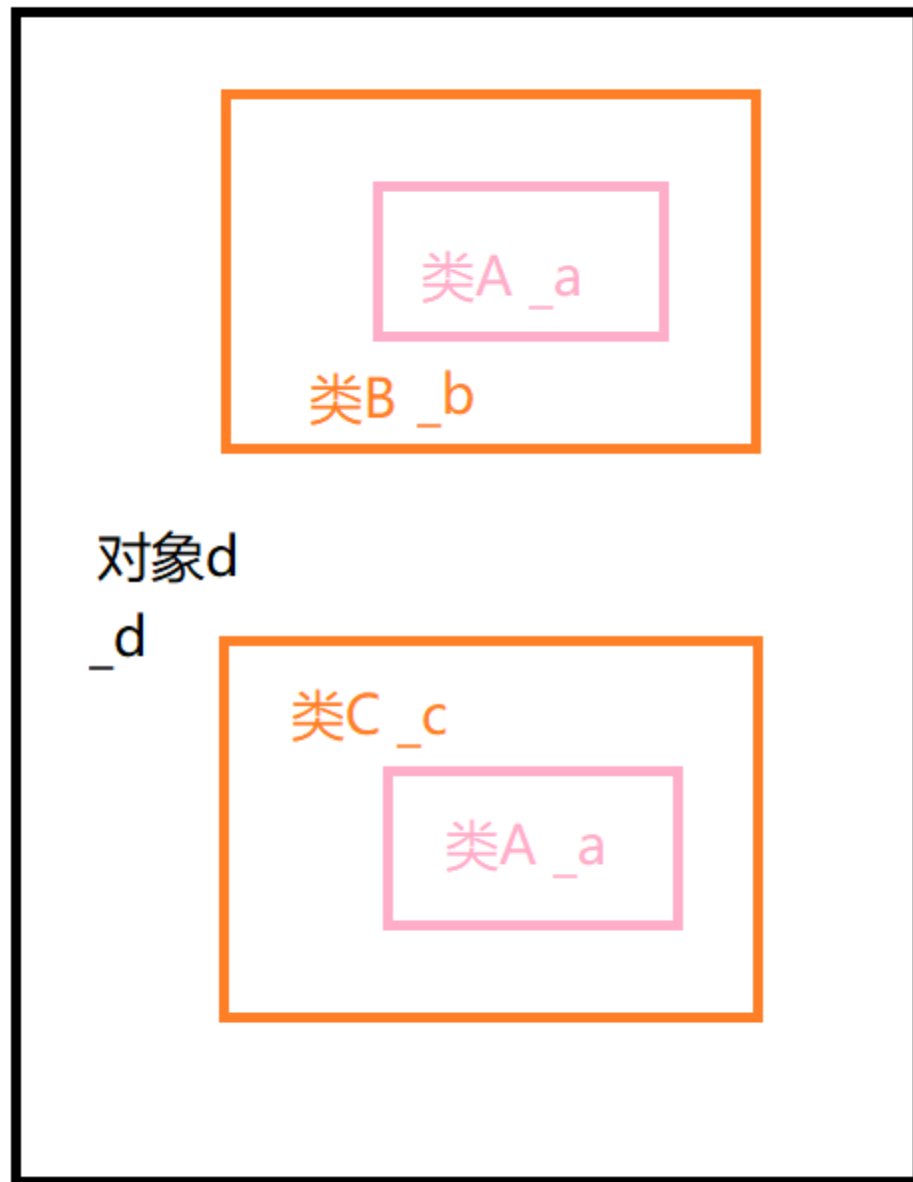


可以看到对于一个对象 d，内部有着两个类 B 和 C，但是对于 B 和 C 内部有分别有着一个类 A，这样就有两个类 A 存于对象 d 中，就会导致数据冗余和二义性。

画图理解：

插图：

## 菱形继承



[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

可以看到一个对象中有着两个\_a 数据



- 虚拟菱形继承内部结构

```
1. class A
2. {
3.
4. public:
5.     int _a;
6. };
7.
8. class B : virtual public A
9. {
10.
11. public:
12.     int _b;
13. };
14.
15. class C : virtual public A
16. {
17.
18. public:
19.     int _c;
20. };
21.
22. class D : public B, public C
23. {
24.
25. public:
26.     int _d;
27. };
28.
29. int main()
30. {
31.     D d;
32.     d._a = 0; // 可以赋值, 没有二义性问题
33.     d.B::_a = 2;
34.     d.C::_a = 1;
```

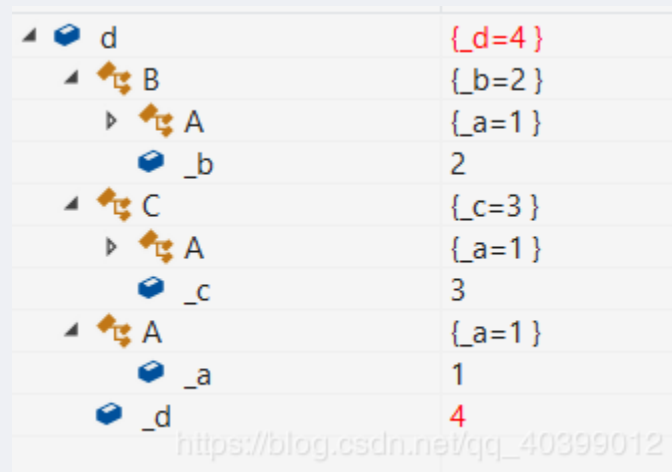
```

35.     d._b = 2;
36.     d._c = 3;
37.     d._d = 4;
38.     return 0;;
39. }

```

对于虚拟菱形继承，对于一个对象 d 来说，内部也是有着类 B 和类 C，类 B 和类 C 内部都有着一个指向类 A 的指针，**虚拟菱形继承相对于菱形继承，就是对于类 A 不再是类 B 和类 C 私有的了，类 A 对于类 B 和类 C 是共有的**

如图：内存结构



内部不仅有着类 B 类 C 还有这着一个类 B 和类 C 共有的类 A，类 B 和类 C 内部都有一个指向类 A 的指针

如何实现不存在数据二义性问题：

- 但是对于属于类 B 和类 C 的类 A 可以看到它内部没有地址空间，所以其实他只是一个指针，也就是一个地址
- 而只有共有的类 A 才有这地址空间可以存放数据\_a,所以对于类 B 和类 C 内部的类 A 的指针，指向的就是这个独立的类 A。从而实现了数据一致性了

d 对象将类 A 放到了组成的最下面，这个类 A 同时属于类 B 和类 C，那么 B 和 C 如何去找到共有的 A 呢？

- 这里是通过 B 和 C 的两个指针，指向的一张表。这两个指针叫做虚基表指针，这两个表叫做虚基表。虚基表中存在着偏移量，通过偏移量可以找到下面的类 A

- 如图：

The screenshot shows a debugger window with two memory views. The top view, '内存 2', shows addresses from 0x011DBB40 to 0x011DBB4C. The bottom view, '内存 3', shows addresses from 0x011DBB48 to 0x011DBB54. A third view on the right shows a memory dump starting at address 0x0113F77C. Blue arrows point from the '偏移量' (offset) labels in the memory views to the corresponding hex values in the dump. The dump shows a sequence of hex values: 40 bb 1d 01 (class B), 03 00 00 00 (class C), 48 bb 1d 01 (class D), 04 00 00 00 (common class A), 05 00 00 00 (common class A), 02 00 00 00 (common class A), and so on. The labels '类B', '类C', '类D', and '共有的类A' are placed next to their respective hex values.

内存地址	内存值 (Hex)	注释
0x011DBB40	00 00 00 00	
0x011DBB44	14 00 00 00	偏移量
0x011DBB48	00 00 00 00	
0x011DBB4C	0c 00 00 00	
0x011DBB48	00 00 00 00	
0x011DBB4C	0c 00 00 00	偏移量
0x011DBB50	00 00 00 00	
0x011DBB54	00 00 00 00	
0x0113F77C	40 bb 1d 01	类B
0x0113F780	03 00 00 00	类C
0x0113F784	48 bb 1d 01	类D
0x0113F788	04 00 00 00	共有的类A
0x0113F78C	05 00 00 00	共有的类A
0x0113F790	02 00 00 00	共有的类A
0x0113F794	cc cc cc cc	????
0x0113F798	1f b5 f3 2e	..??
0x0113F79C	b0 f7 13 01	??..
0x0113F7A0	4e 54 1d 01	NT..
0x0113F7A4	01 00 00 00	....
0x0113F7A8	a8 5a 18 00	?Z..
0x0113F7AC	d0 60 18 00	?^..
0x0113F7B0	08 f8 13 01	..?..

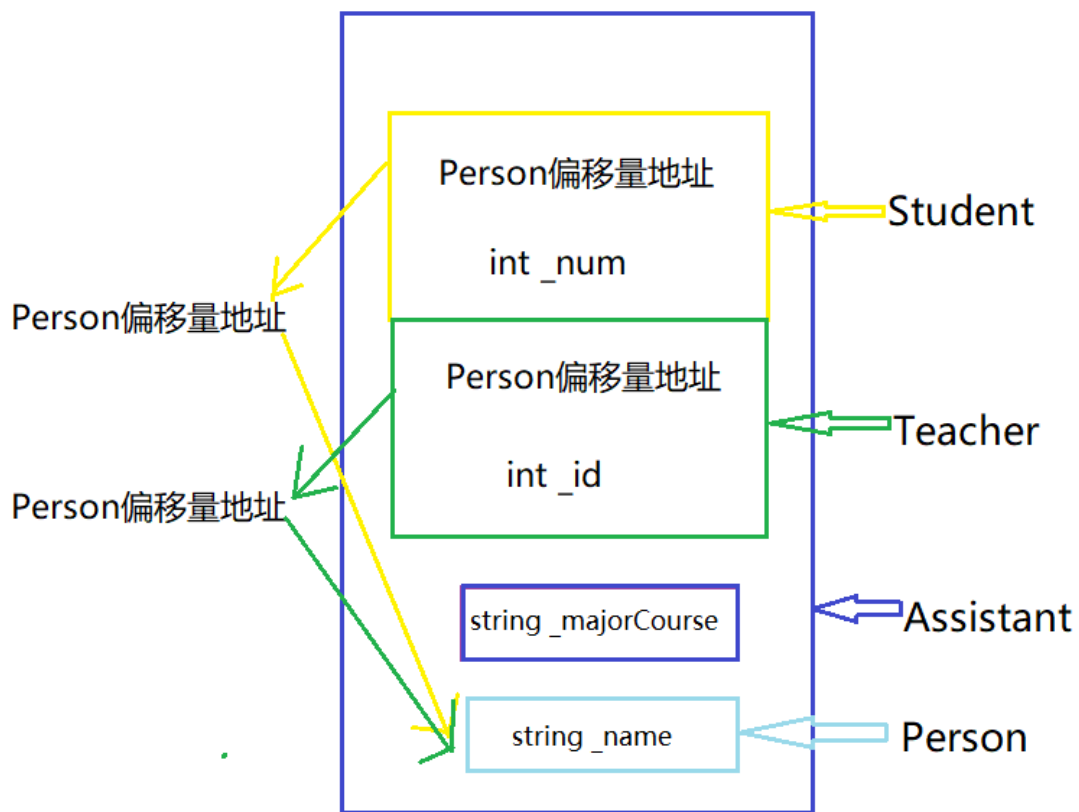
可以看到对于类 B 和类 C 都有着一个虚基表指针，指向一个虚基表，该虚基表下面就是偏移量，从该位置偏移多少就可以找到共有的类 A

**对于对象 d 中的类 B 和类 C 为什么要去找属于自己的类 A 呢？**

- 因为对于类 B 和类 C 均是类 D 的父类。如果是父类的话就有可能产生切片，所以就要找到属于自己的数据，这样的话才不会发生切片错误

对于 Person 关系菱形虚拟继承的原理解释：

如图：



[https://blog.csdn.net/qq\\_40399012](https://blog.csdn.net/qq_40399012)

## 10. 总结

1. 很多人说 C++ 语法复杂，其实多继承就是一个体现。有了多继承，就存在菱形继承，有了菱形继承就有了菱形虚拟继承，底层实现就很复杂。所以一般不建议设计出多继承，一定不要设计出菱形继承。否则在复杂度及性能上都有问题
2. 多继承可以认为是 C++ 的缺陷之一，很多后来的 OO 语言都没有多继承，如 java
3. 继承和组合

1. public 继承是一种 is-a 的关系，也就是说每个派生类对象都是一个基类对象

2. 组合是一种 has-a 的关系。假设 B 组合了 A，每个 B 对象都有着一个 A 对象
3. 优先使用对象组合，而不是类继承（高内聚，低耦合）
4. 继承允许你根据基类类的实现来定义派生类的实现，这种通过生成派生类的复用称为白箱复用(white-box reuse)。术语“白箱”是相对可视性而言：在继承方式中，基类的内部细节对子类可见。继承一定程度破坏了基类的封装，基类的改变，对于派生类有很大的影响。派生类和基类间的依赖关系很强，耦合度高。
5. 对象组合是类继承之外的另一种复用选择。对象组合要求被组合的对象具有良好定义的接口。这种复用风格称为黑箱复用(black-box reuse)，因为对象的内部细节是不可见的。对象只以“黑箱”的形式出现。组合类之间没有很强的依赖关系，耦合度低。优先使用对象组合有助于保持每一个类被封装，内聚性高。
6. 实际中尽量使用组合。组合的耦合度低，代码维护性好。不过继承也有用武之地，有些关系就适合用继承，另外实现多态，也必须要继承。类之间的关系可以使用继承，可以使用组合就用组合。