第7讲 赋值兼容与虚基类

目录 CONTENTS

1 赋值兼容规则

- 通过公有继承，派生类得到了基类中除构造函数、析构函数之外的所有成员，而且所有成员的访问控制属性也和基类完全相同。

- 这样，公有派生类实际就具备了基类的所有功能，凡是基类能解决的问题，公有派生类都可以解决。

1 赋值兼容规则

兼容规则包括：

- 派生类的对象可以赋值给基类对象。
- 派生类的对象可以初始化基类引用。
- 派生类对象的地址可以赋给指向基类指针。

- 派生类对象可以作为基类的对象使用，但只能使用从基类继承的成员。

赋值兼容规则的使用场景

具体表现在以下几个方面：

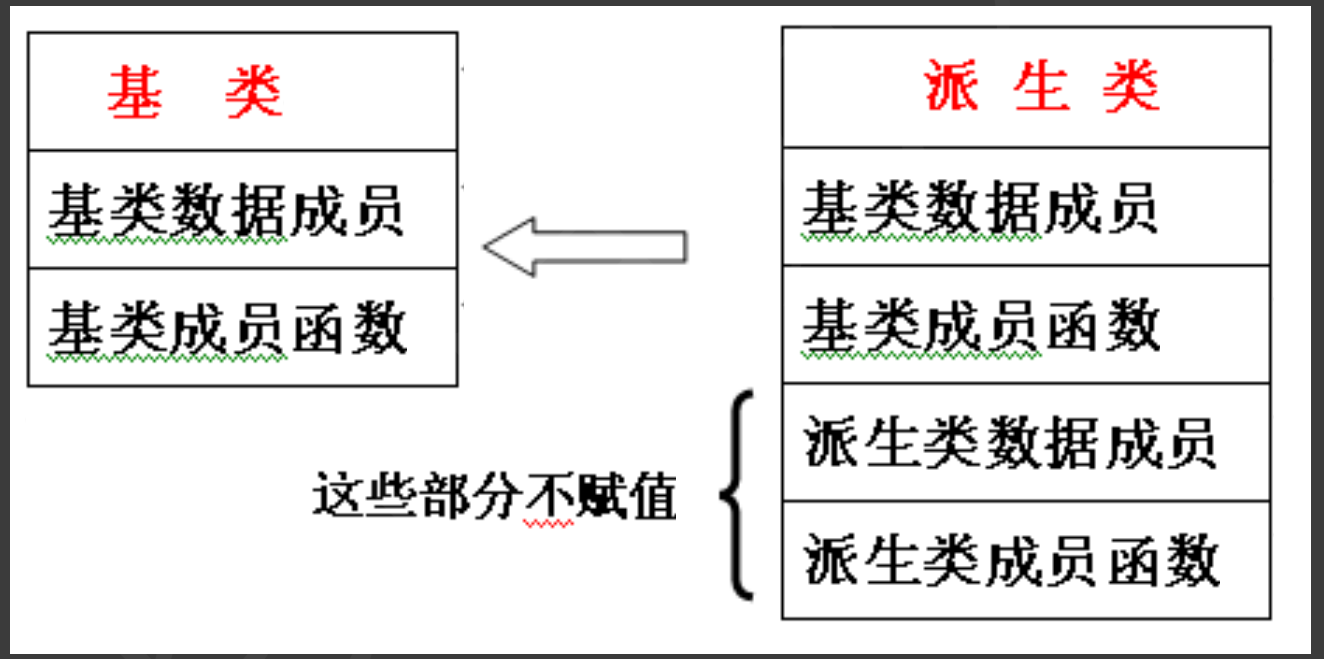
(1) 可以用派生类对象给基类对象赋值。

例如:

```
Base b;          //定义基类Base的对象b
Derived d;       //定义公有派生类Derived的对象d
b=d;
```

这样赋值的效果是，对象b中数据成员将具有对象d中对应数据成员的值。

- 所谓赋值仅仅指对基类的数据成员赋值。



赋值兼容规则的使用场景

(2) 可以用派生类对象来初始化基类对象的引用。

例如:

```
Base b;       //定义基类Base的对象b
Derived d;    //定义基类Base的公有派生类
              //Derived的对象d
Base& br=d;
```

赋值兼容规则的使用场景

(3) 派生类对象的地址可以赋给指向基类对象的指针。

例如:

```
Derived d;     //定义基类Base的公有派生类
               //Derived的对象d
Base* bp=&d;
```

```
#include<iostream>
using namespace std;
class Base{
 public:
   int i;
   Base(int x) { i=x; }
   void Show(){ cout<<"Base "<<i<<endl; }
};
class Derived:public Base{
  public:
   Derived(int x):Base(x)  {  };
   void Show()
   { cout<<"Derived "<<i<<endl;}
};
```

声明基类Base

声明公有派生类Derived

```
int main()
{
  Base b1(11); b1.disp();    //①
  Derived d1(22);
  b1=d1; b1.Show();          //②
  Derived d2(33);
  Base& b2=d2; b2.Show();  //③
  Derived d3(44);
  Base* b3=&d3;
  b3->Show();                //④
  Base *b4=new Derived(55);
  b4->Show();                //⑤
  delete d4;
  return 0;
}
```

用派生类对象给基类对象赋值

用派生类对象来初始化基类的引用

把派生类对象的地址赋值给指向基类的指针

运行结果:

Base 11 ①

Base 22 ②

Base 33 ③

Base 44 ④

Base 55 ⑤

赋值兼容规则的使用场景

(4) 如果函数的形参是基类对象或基类引用，在调用函数时可以用派生类对象作为实参。

华北电力大学

赋值兼容规则的使用场景

```
class Base{                              //声明基类Base
 public:
    int i;
  …
};
class Derived:public Base {   //声明公有派生类Derived
  …
};
void fun(Base& bb)          //普通函数,形参为基类Base对象的引用
{
   cout<<bb.i<<endl;        //输出该引用所代表的对象的数据成员i
}
Derived d;
fun(d);
```

说明

(1) 声明为指向基类对象的指针可以指向它的公有派生的对象，但不允许指向私有派生的对象。

华北电力大学

说明

例如:

```
class Base{
…};
class Derive:private Base
{
…};
int main()
{ Base op1,*ptr;
  Derive op2;
  ptr=&op1;
  ptr=&op2;
   …
}
```

错误,不允许将指向基类Base的指针ptr指向它的私有派生类对象op2

说明

(2) 不能将一个派生类指针指向基类对象。

华北电力大学

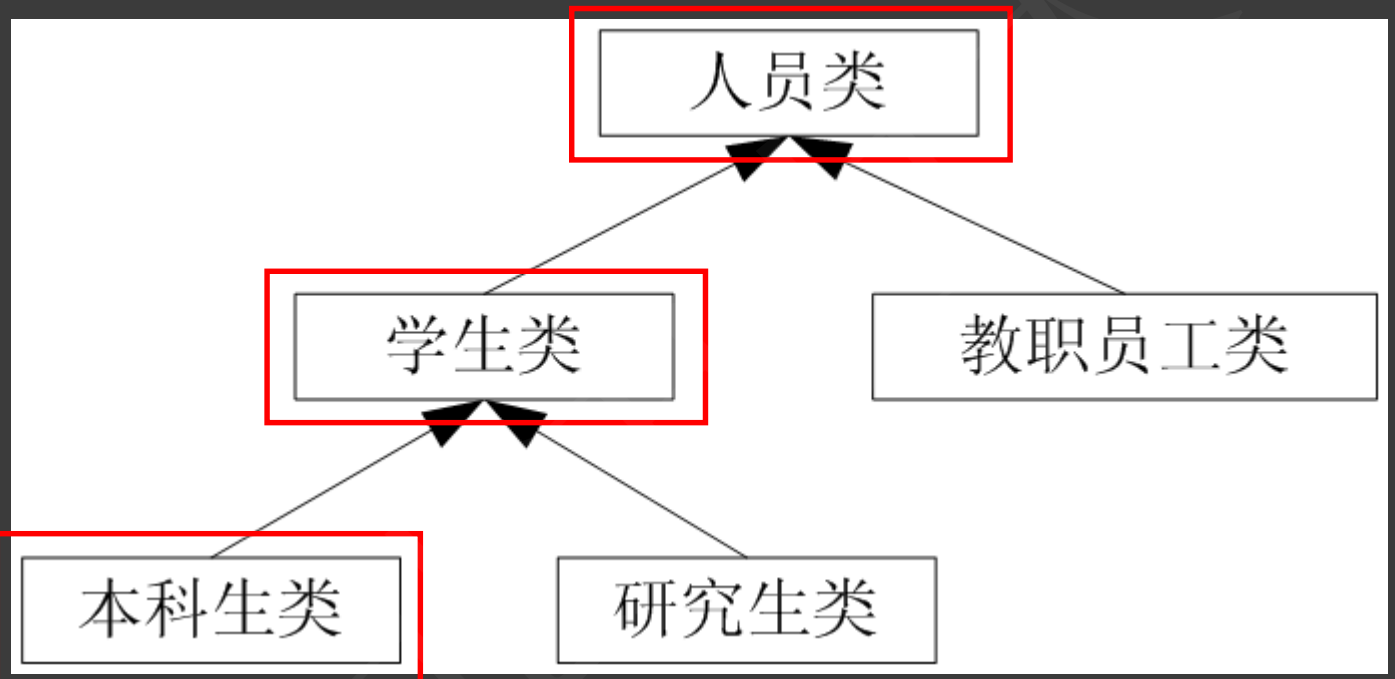
说明

例如:

```
class Base{
… };
class Derived:public Base{
…  };
int main()
{
  Base obj1;
  Derived obj2,*ptr;
  ptr=&obj2;
  ptr=&obj1;
  …
}
```

错误, 试图将指向派生类对象的指针ptr指向其基类对象obj1

2 具有多个层次的单继承



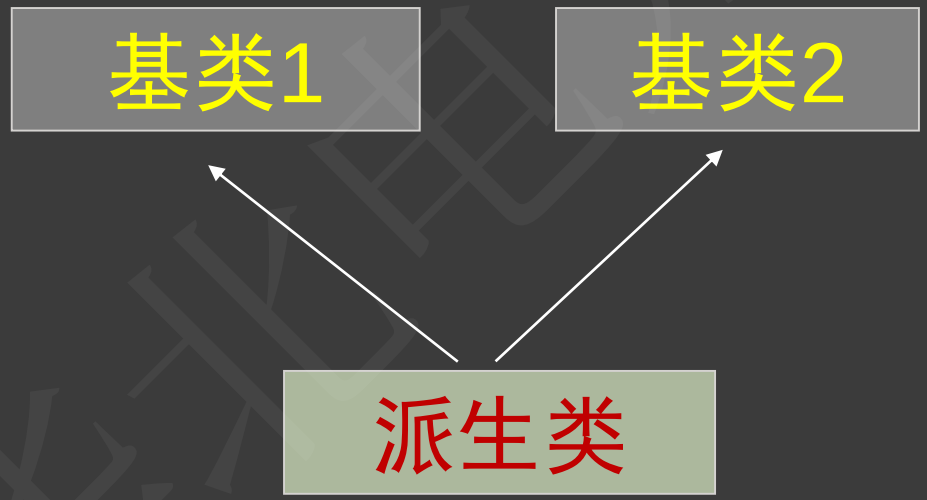
```
class A
{
  private:
    int m_a;
  public:
    A(int a) {m_a=a;}
};

class B:public A
{
  private:
    int m_b;
  public:
    B(int a, int b): A(a) {m_b=b;}
};

class C:public B
{
  private:
    int m_c;
  public:
    C(int a, int b, int c):B(a,b) {m_c=c;}
};
```

3 多重继承

- 当一个派生类具有多个基类时，这种派生方法称为多重派生或多重继承。



3.1 多重继承声明的一般形式

多重继承声明的形式如下:

```
class 派生类名：继承方式1 基类名1，…，  继承方式n 基类名n
{
    派生类新增的数据成员和成员函数
};
```

```
#include<iostream.h>
class X {
     int x;
   public:
     void SetX(int x) { this.x=x; }
     void ShowX( ){  cout<<“x="<<x<<endl; } };
class Y {
    int y;
 public:
    void SetY(int y) { this.y=y; }
    void ShowY(){cout<<“y="<<y<<endl;} };
class Z :public X, private Y{
    int z;
  public:
    void SetXYZ(int x, int y, int z)
    { setX(x); setY(y); this.z=z; }
    void ShowZ( ){ ; cout<<“z="<<z<<endl; }
};
```

```
void main( )
{    Z obj;
     obj.SetX(3);
     obj.ShowX( );
     obj.SetY(4);
     obj.ShowY( );
     obj.SetXYZ(6,8);
     obj.ShowZ( );
}
```

```
class Z：private X，public Y {

   //…
};
class Z：public X ,Y {

   //…
};
```

类Z私有继承了类X，公有继承了类Y

类X, 私有继承了Z公有继承了类类Y

在多重继承中,各种继承方式对于基类成员在派生类中的访问属性和规则与单继承相同。

```
class X{
   public:
     int f();
};
class Y{
  public:
    int f();
    int g();
};
class Z : public X，public Y{
  public:
    int g();
    int h();
};
```

```
int main( )
{
  Z obj;
  …
  obj.f();    …
}
```

二义性!

访问哪个父类中的f()?

```
class X{
   public:
     int f();
};
class Y{
  public:
    int f();
    int g();
};
class Z : public X，public Y{
  public:
    int g();
    int h();
};
```

```
int main( )
{
  Z obj;
  …
  obj.f();    …
}
```

使用成员名限定可以消除二义性，例如：

obj.**X**::f()；//调用类X的f()

obj.**Y**::f()；//调用类Y的f()

3.2多继承的构造函数与析构函数

多重继承构造函数定义的一般形式如下：

```
派生类名(参数总表)：基类名1(参数表1)，基类名2(参数表2)，…，基类名n(参数表)
{
    派生类新增成员的初始化语句
}
```

多重继承构造函数的执行顺序

- 多重继承构造函数的执行顺序与单继承构造函数的执行顺序相同:
  - 先执行基类的构造函数;
  - 再执行对象成员的构造函数;
  - 最后执行派生类构造函数。
- 析构函数的执行顺序则与构造函数的执行顺序相反。

```
class X{
   public:
      X(){cout<<"constructor X."<<endl;}
};
class Y{
   public:
      Y(){ cout<<"constructor Y."<<endl;}
};
class Z :public Y, public X{
   public:
       Z(){ cout<<"constructor Z."<<endl;}
};
void main()
{    Z zz;    }
```

运行结果?

多个基类构造函数的执行顺序，严格按照派生类声明时从左到右的排列顺序来执行。

运行结果：
constructor Y.
constructor X.
constructor Z.

4 虚基类

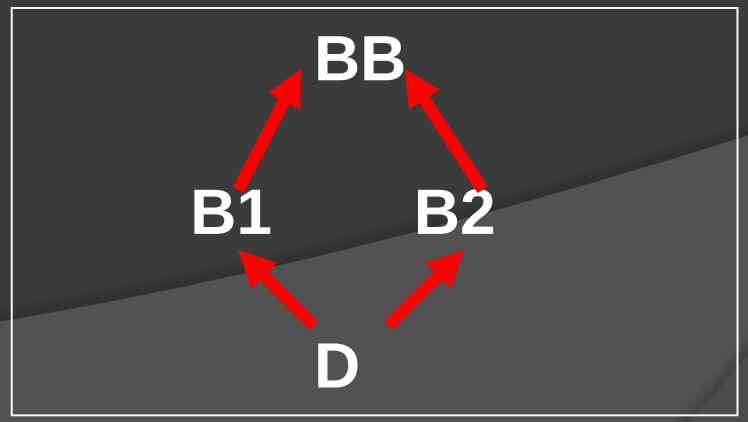
1.为什么要引入虚基类

如果一个派生类是从多个基类派生出来的，而这些基类又有一个共同的基类，

则在这个派生类中访问这个共同的基类中的成员时，可能会产生二义性。

```
class  B{
  protected:  int a;
  public: B( ){ ...}
};
class B1: public  B{
  public: base1( ){...}
};
class B2: public  B{
  public: B2( ){...}
};
class D:public B1,public B2{
. . .
}
```

派生类D是两个基类B1和 B2派生出来的，而这些基类又有一个共同的基类B,则在派生类D中访问共同基类B中的成员a时，可能会产生二义性。



类D有类B1的数据成员a和类B2的数据成员a,即B1::a和B2::a。因此如果在类D中输出a的值,而不加“::”,将出现二义性。

```
class B {
  protected:
   int a;
  public:
   B(){ a=5; cout<<"B a="<<a<<endl;}  };
class B1:public B{
   public:
    B1(){ a=a+10; cout<<"B1 a="<<a<<endl;} };
class B2:public B{
  public:
    B2(){ a=a+20; cout<<"B2 a="<<a<<endl;} };
class D:public B1,public B2{
  public:
    D( ){ cout<<"D a="<< a<<endl;} };
int main( )
{ D  obj; return  0;  }
```

B1::a

B2::a

出现二义性

解决办法1：
在程序中注明B1::a和B2::a

```
class B {
  protected:
   int a;
  public:
   B(){ a=5; cout<<"B a="<<a<<endl;}  };
class B1:public B{
   public:
    B1(){ a=a+10; cout<<"B1 a="<<a<<endl;} };
class B2:public B{
  public:
    B2(){ a=a+20; cout<<"B2 a="<<a<<endl;} };
class D:public B1,public B2{
  public:
  D( ){ cout<<“B1 a="<<B1::a<<endl;  cout<<“B2 a="<< B2::a<<endl;}
int main( )
{ D  obj; return  0;  }
```

运行结果：

```
B a=5
B1 a=15
B a=5
B2 a=25
B1 a=15
B2 a=25
```

解决办法2：
使从不同的路径继承的基类的成员在内存中只拥有一个拷贝,C++引入了虚基类的概念。

```
class B {
  protected:
   int a;
  public:
   B(){ a=5; cout<<"B a="<<a<<endl;}  };
class B1:public B{
   public:
    B1(){ a=a+10; cout<<"B1 a="<<a<<endl;} };
class B2:public B{
  public:
    B2(){ a=a+20; cout<<"B2 a="<<a<<endl;} };
class D:public B1,public B2{
  public:
    D( ){ cout<<"D a="<< a<<endl;} };
int main( )
{ D  obj; return  0;  }
```

出现二义性

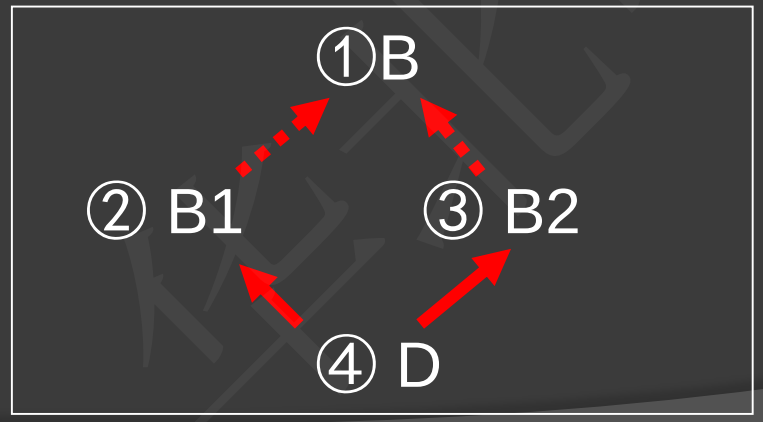
虚基类的概念

- 如果将公共基类说明为虚基类。那么，对同一个虚基类的构造函数只调用一次；
- 这样从不同的路径继承的虚基类的成员在内存中就只拥有一个拷贝。从而解决了以上的二义性问题。
- 虚基类是在派生类的声明,其语法形式如下:

```
class  派生类名:继承方式 virtual 基类名
{   … }
或
class  派生类名: virtual 继承方式 基类名
{  … }
```

```
class B{
  protected:  int a;
  public: B( ){ ...} };
class B1: public virtual B{
  public: B1( ){...} };
class B2: public virtual B{
  public: B2( ){...} };
class D:public B1,public B2{ . . .}
```

类B作为类B1和类B2的虚基类



构造函数执行顺序：
B()-> B1()-> B2()-> D()

虚基类的说明

虚基类的初始化与一般的多重继承的初始化在语法上基本上是一样的，但有一些特殊的规定：

(1) 对同一个虚基类的构造函数只调用一次，且是在第一次出现时调用；

华北电力大学

```
class B {
  protected:
   int a;
  public:
   B( ){ a=5; cout<<"B a="<<a<<endl;}  };
class B1:public virtual B{
   public:
    B1( ){ a=a+10; cout<<"B1 a="<<a<<endl;} };
class B2:public virtual B{
  public:
    B2( ){ a=a+20; cout<<"B2 a="<<a<<endl;} };
class D:public B1,public B2{
  public:
    D( ){ cout<<"D a="<< a<<endl;} };
main( )
{ D  obj; return  0;  }
```

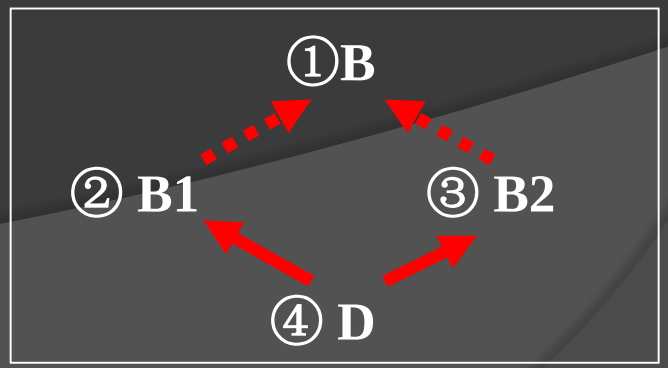
运行结果为:

B　a=5

B1　a=15

B2　a=35

D　a=35



虚基类的说明

(2) 如果在虚基类中定义有带形参的构造函数，并且没有定义无参构造函数，则整个继承结构中，所有直接或间接的派生类都必须在构造函数的成员初始化表中列出对虚基类构造函数的调用，以初始化在虚基类中定义的数据成员。

```
class B {  int a;
  public:
   B(int ca)
  { a=ca;  cout<<"Constructing B"<<endl; } };
class B1:virtual public B{  int b;
  public:
   B1(int ca,int cb):B(ca)
   { b=cb; cout<<"Constructing B1"<<endl; } };
class B2:virtual public B{  int c;
  public:
   B2(int ca,int cc):B(ca)
  { c=cc; cout<<"Constructing B2"<<endl; } };
class D:public B1,public B2 {
   int d;
  public:
   D(int ca,int cb,int cc,int cd): B(ca),B1(ca,cb),B2(ca,cc)
   { d=cd; cout<<"Constructing D"<<endl;} };
 int main()
{ D  obj(2,4,6,8); return 0; }
```

要求在派生类B1、B2和D的构造函数的初始化表中，都必须带有对B构造函数的调用。

运行结果为:

```
Constructing B
Constructing B1
Constructing B2
Constructing D
```

虚基类的说明

(3)虚基类构造函数的调用顺序：

- 若同一层次中同时包含虚基类和非虚基类,应先调用虚，再调用非虚基类的构造函数,最后调用派生基类的构造函数;
- 对于多个虚基类,构造函数的执行顺序仍然是先左后右,自上而下;
- 对于非虚基类,构造函数的执行顺序仍是先左后右,自上而下;

虚基类构造函数的调用顺序

例如：

```
class X : public Y，virtual public Z{
    //…
};
X  one;
```

先调用虚基类

定义类X的对象one后，将产生如下的调用次序。

```
Z( );
Y( );
X( );
```

虚基类的可见性

当在 D 中直接访问 a 时，会有三种可能性：
(1)如果B1和B2中都没有a的定义，那么a将被解析为B的成员，此时不存在二义性。
(2)如果B1或B2其中的一个类定义了a，也不会有二义性，派生类的a比虚基类的a优先级更高。
(3)如果B1和B2中都定义了a，那么直接访问a将产生二义性问题。

```
class B {  int a;
  public:
   B(int ca)
  { a=ca;  cout<<"Constructing B"<<endl; } };
class B1:virtual public B{  int b; int a;
  public:
   B1(int ca,int cb):B(ca)
   { b=cb; cout<<"Constructing B1"<<endl; } };
class B2:virtual public B{  int c; int a;
  public:
   B2(int ca,int cc):B(ca)
  { c=cc; cout<<"Constructing B2"<<endl; } };
class D:public B1,public B2 {
   int d;
  public:
   D(int ca,int cb,int cc,int cd): B(ca),B1(ca,cb),B2(ca,cc)
   { d=cd; cout<<"Constructing D"<<endl;} };
 int main()
{ D  obj(2,4,6,8); return 0; }
```

单选题 2分

设置

此题未设置答案，请点击右侧设置按钮

关于虚基类的说法**不正确**的是

A 引进虚基类的目的是解决二义性问题

B 虚基类被重复继承时在派生类中只产生一个副本

C 在派生类的对象被创建时，要保证其虚基类的构造函数被调用一次

D 因为虚基类可被多个派生类继承，因此在重复继承的子类中会有多个副本

提交