## >>探索单继承的内存布局

### 单继承：只有一个基类和一个派生类

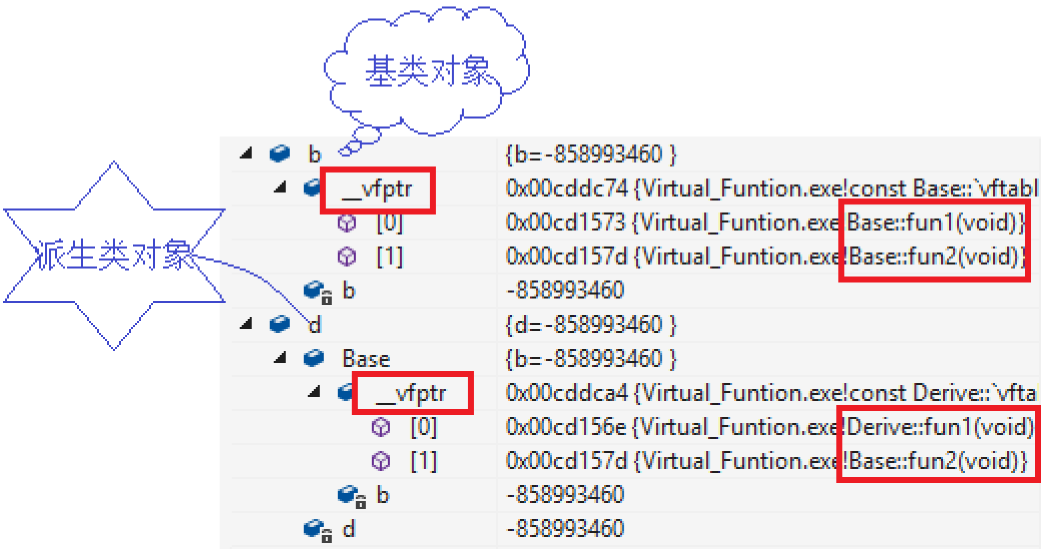
```
class Base
{
public:
    virtual void fun1()
    {
        cout << "Base::func1()" << endl;
    }
    virtual void fun2()
    {
        cout << "Base::func2()" << endl;
    }
private:
    int b;
};
class Derive :public Base
{
public:
    virtual void fun1()          //重写基类虚函数,实现多态
    {
        cout << "Derive::func1()" << endl;
    }

    virtual void fun3()
    {
        cout << "Derive::func3()" << endl;
    }
    void fun4()
    {
        cout << "Derive::func4()" << endl;
    }
private:
```

```
    int d;
};
```

## 1. 虚表就是存放虚函数的表。

## 2. 主函数中分别定义一个基类对象和一个派生类对象，通过调试窗口可以看到所谓的虚表，如下图（整型b和d未初始化）：




## >也许你会有疑问：调试窗口中派生类虚表为什么看不到Derive中的fun3()函数，这是编译器的问题，我所用的是vs2013，在调试的时候确实不见fun3()函数，所以有时编译器的调试窗口显示的也不能完全相信，那有什么办法证明fun3()函数也在派生类虚表里呢？通过打印虚表！

代码如下：

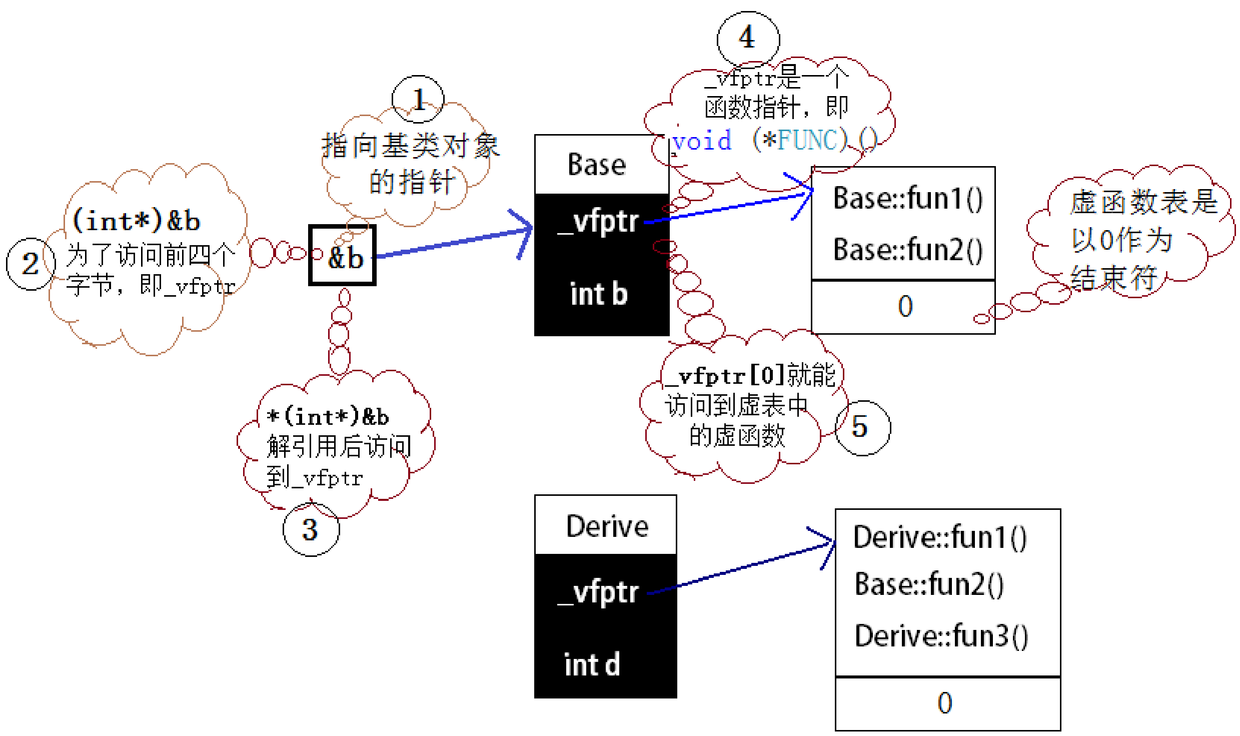
```
typedef void (*FUNC)();        //重定义函数指针，指向函数的指针
void PrintVTable(int* vTable)  //打印虚函数表
{
    if (vTable == NULL)
    {
        return;
    }
    cout << "虚函数表地址:" << vTable << endl;
    int  i = 0;
    for (; vTable[i] != 0; ++i)
    {
```

```
        printf(" 第%d个虚函数地址 :0X%x,->", i, vTable[i]);
        FUNC f = (FUNC)vTable[i];
        f();        //访问虚函数
    }
    cout << endl;
}

void Test1()
{
    Base b;
    Derive d;
    int* tmp = (int*)(*(int*)&b);     //取到虚函数的地址
    PrintVTable(tmp);
    int* tmp1 = (int*)(*(int*)&d);
    PrintVTable(tmp1);
}
```

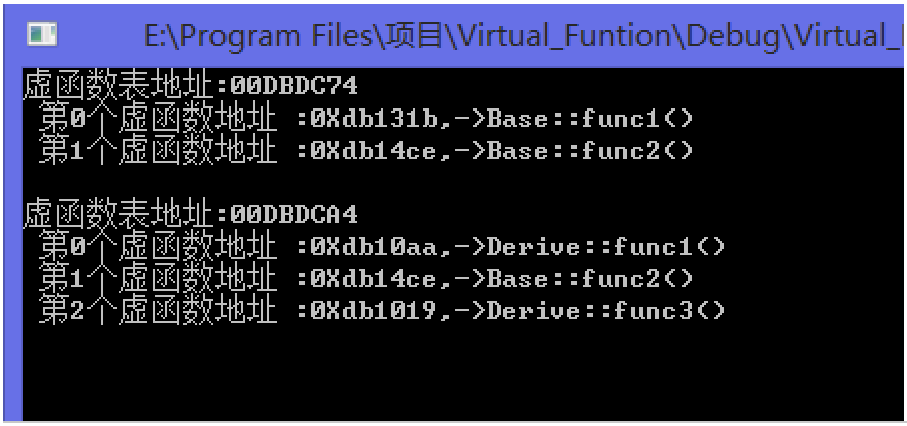
## 解析：`int* tmp = (int*)(*(int*)&b);`

## 如下图：




---

## 打印虚表：




**>注意:**

不知你是否注意到派生类中还有一个函数：void fun4();

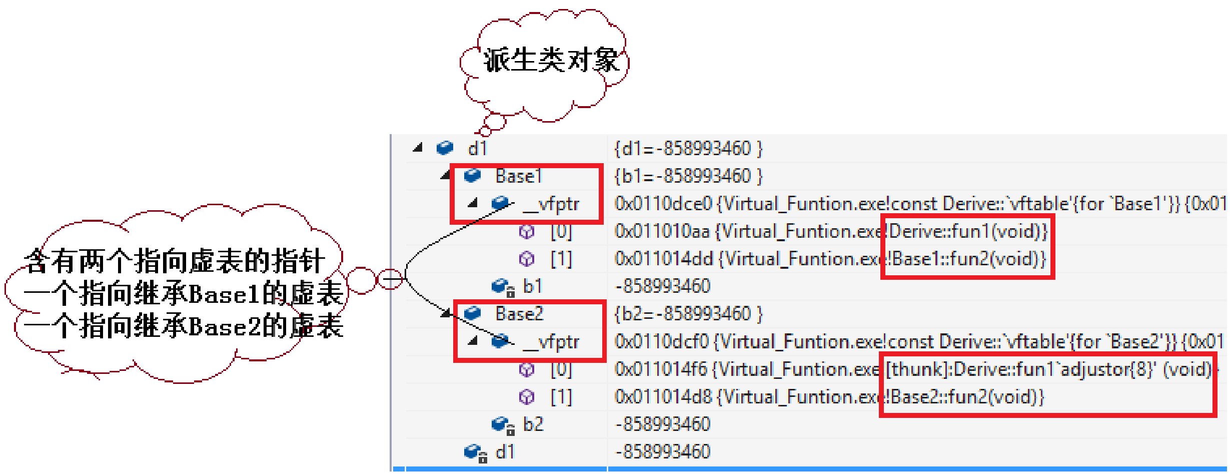
虚函数是为了实现动态多态，是当程序运行到该函数时才会去虚表里找这个函数；而函数的重载实现的是静态多态， 是在程序编译时就能找到该函数地址，而函数：void fun4();不是虚函数，自然不会在虚函数表里。

## >>探索多继承的内存布局

```
class Base1   //基类
{
public:
    virtual void fun1()
    {
        cout << "Base1::fun1" << endl;
    }
    virtual void fun2()
    {
        cout << "Base1::fun2" << endl;
    }
private:
    int b1;
};
class Base2  //基类
{
public:
    virtual void fun1()
    {
        cout << "Base2::fun1" << endl;
```

```
    }
    virtual void fun2()
    {
        cout << "Base2::fun2" << endl;
    }
private:
    int b2;
};
class Derive : public Base1, public Base2  //派生类
{
public:
    virtual void fun1()
    {
        cout << "Derive::fun1" << endl;
    }
    virtual void fun3()
    {
        cout << "Derive::fun3" << endl;
    }
private:
    int d1;
};
```

## 调试看结果：

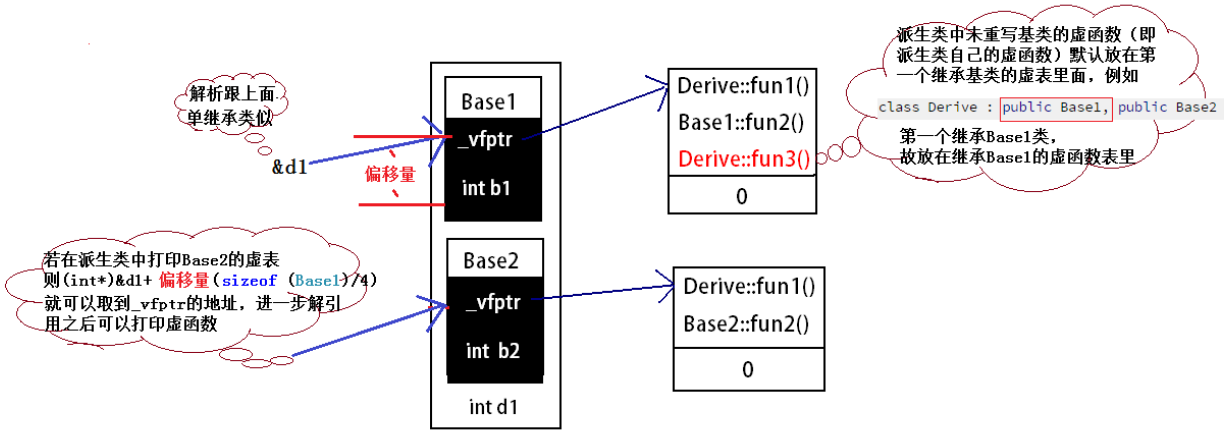



## 同样以上面单继承打印虚表函数来打印多继承虚表：

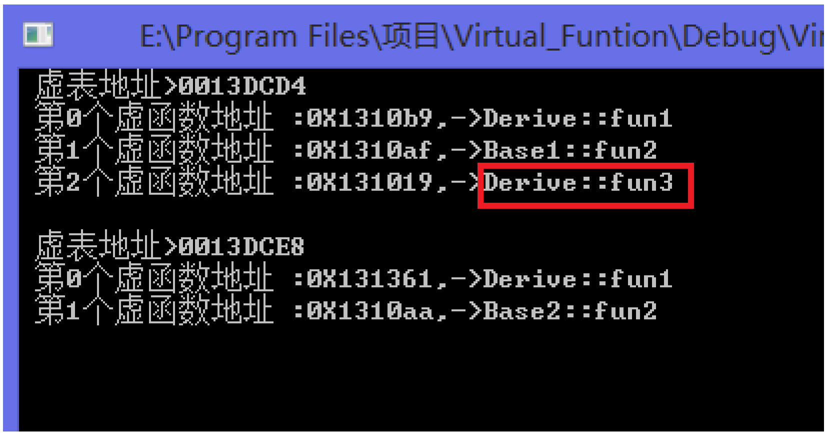
## void PrintVTable(int* vTable); //打印虚函数表

```
void Test1()
{
    Derive d1;
    int* VTable = (int*)(*(int*)&d1);
    PrintVTable(VTable);
    VTable = (int*)(*((int*)&d1 + sizeof (Base1)/4));
    PrintVTable(VTable);
}
```

## 解析：VTable = (int*)(*((int*)&d1 + sizeof (Base1)/4));




## 打印多继承虚表如下图：

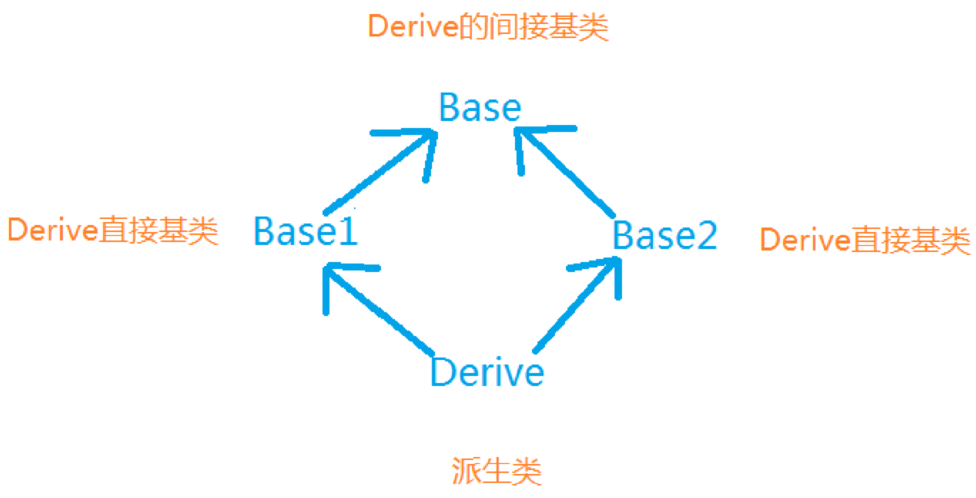



>>探索菱形继承(非虚继承)的内存布局

## 先了解什么是菱形继承？

## >>关系如下图：

**子对象重叠：当继承基类时，在派生类中就获得了基类所有数据成员的副本，该副本称为子对象。**




**两个子类继承同一个父类，而又有子类又分别继承这两个子类。**

**产生的问题，会产生二义性问题，即对于base的调用要说明作用域的情况：**

```
Derive  d,*pd;
pd = &d;
pd->Base1::b = 1;
pd->Base2::b = 2;
cout<<Base1::b;
cout<<Base2::b;
```

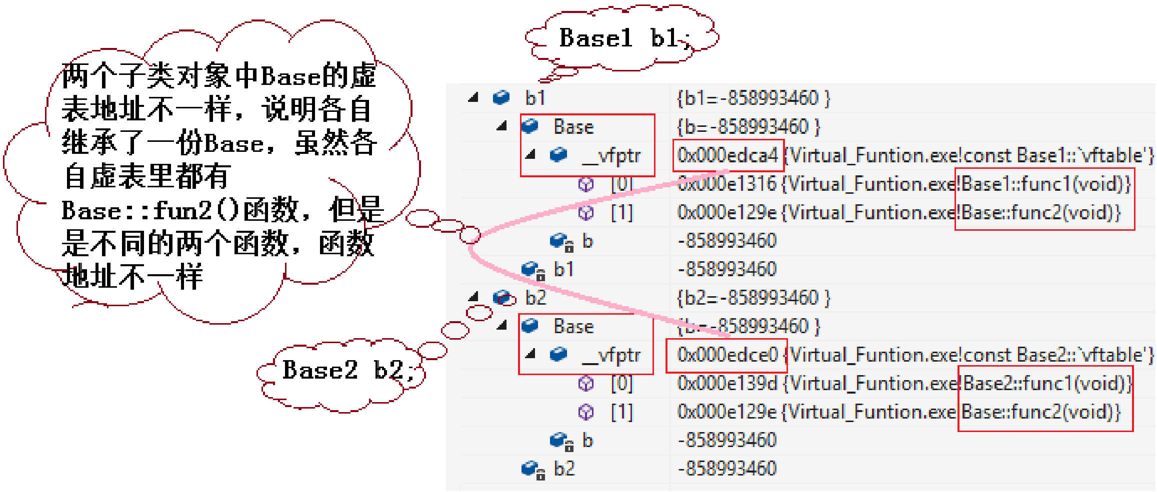
# 下列代码是菱形继承体系：

```
class Base        //Derive的间接基类
{
public:
    virtual void func1()
    {
        cout << "Base::func1()" << endl;
    }
    virtual void func2()
    {
        cout << "Base::func2()" << endl;
    }
private:
    int b;
};
class Base1 :public Base  //Derive的直接基类
{
```

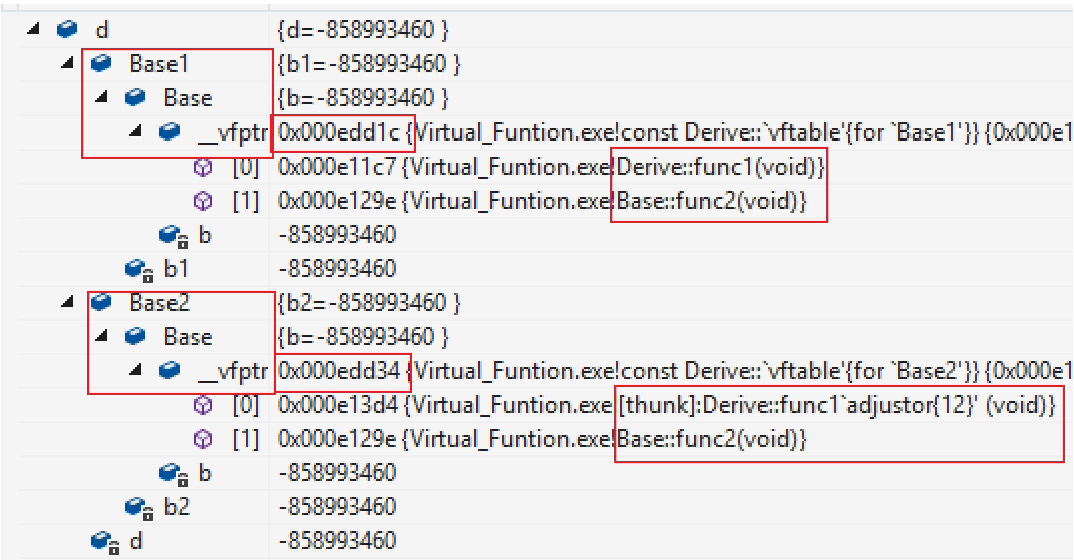
```
public:
    virtual void func1()          //重写Base的func1()
    {
        cout << "Base1::func1()" << endl;
    }
    virtual void func3()
    {
        cout << "Base1::func3()" << endl;
    }
private:
    int b1;
};
class Base2 :public Base    //Derive的直接基类
{
public:
    virtual void func1()       //重写Base的func1()
    {
        cout << "Base2::func2()" << endl;
    }
    virtual void func4()
    {
        cout << "Base2::func4()" << endl;
    }
private:
    int b2;
};
class Derive :public Base1, public Base2
{
public:
    virtual void func1()          //重写Base1的func1()
    {
        cout << "Derive::func1()" << endl;
    }
    virtual void func5()
    {
        cout << "Derive::func5()" << endl;
    }
private:
```

```
    int d;
};
```

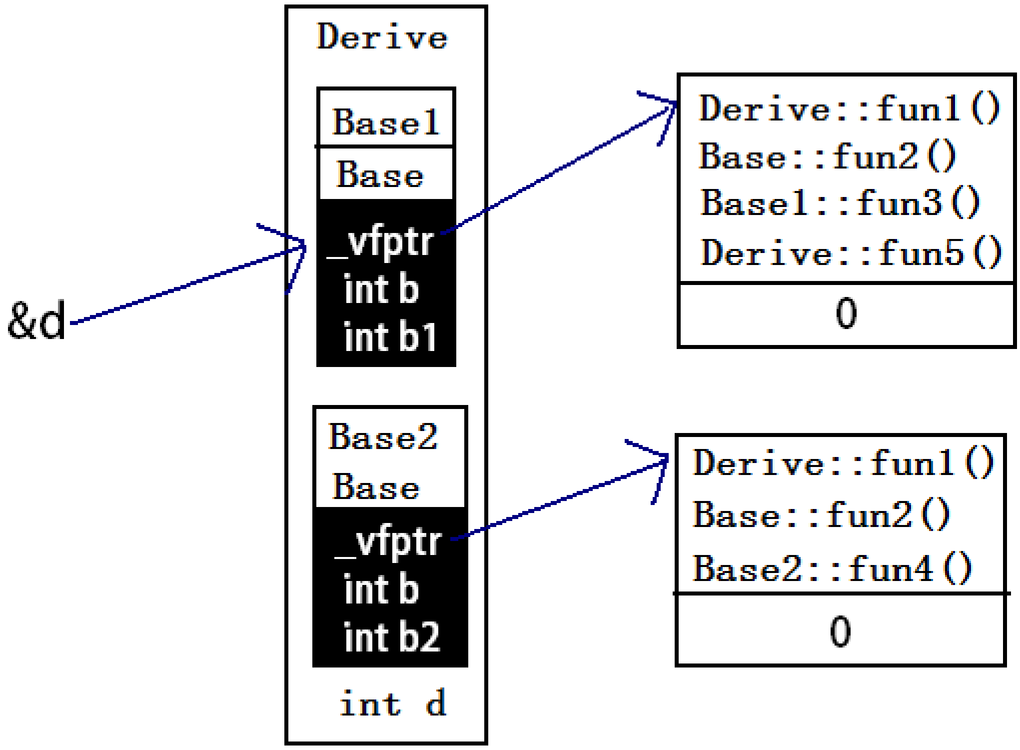
# 菱形继承其实是一个单继承与多继承的结合。




# 下面跟踪Derive对象d的内存布局：

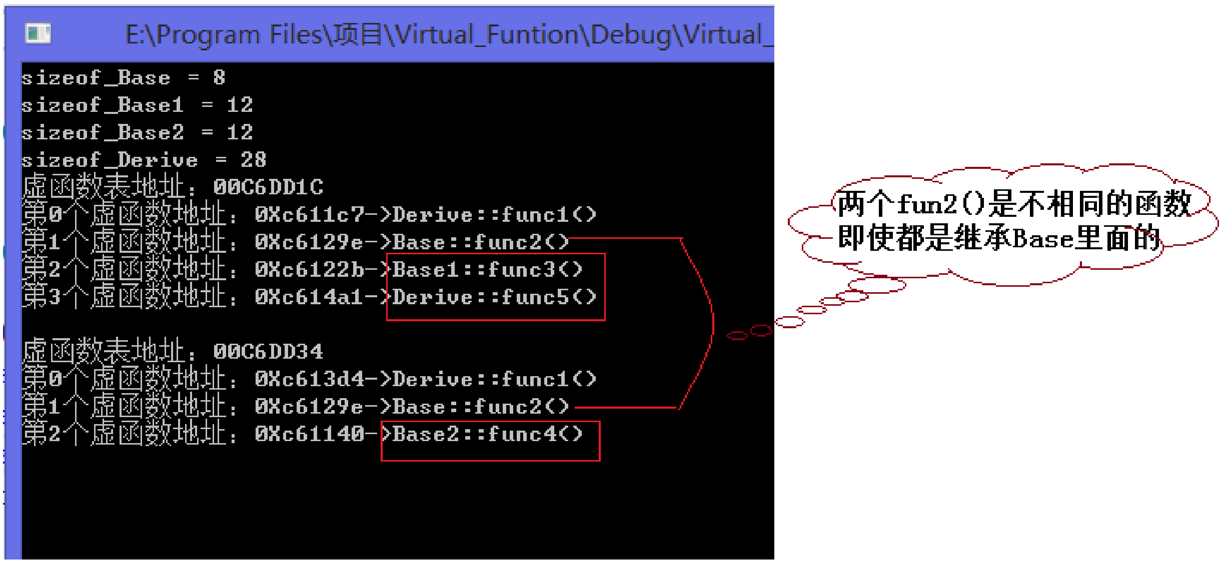



# 进一步解析如下图：




**同样以上面单继承打印虚表函数来打印多继承虚表：**

**void PrintVTable(int* vTable); //打印虚函数表**



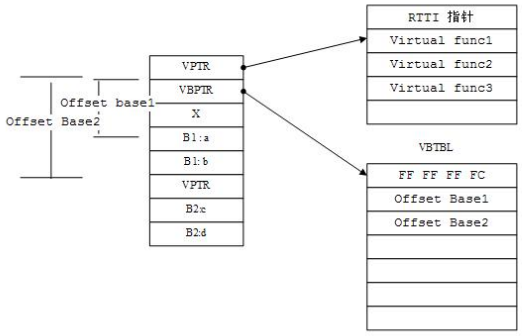

## >>探索菱形继承(虚继承)的内存布局

## >>怎么消除菱形继承的二义性？？

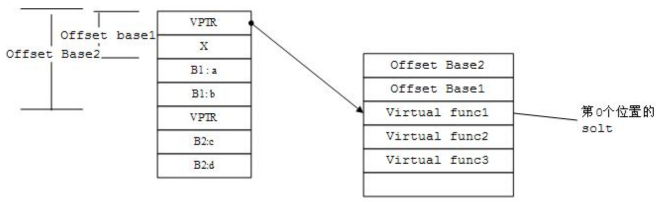
## 使用虚拟继承（虚基类）！

---

## >1. 先来看下面两个方案:




VS2003下虚继承的VBPTR及VBTBL




Bjarne的方案

## > 主要解析第一种方案:

**vs2003下虚继承的VBPTR及VBTBL:**

**在类中增加一个指针(VBPTR)指向一个VBTBL，这个VBTBL的第一项记载的是从VBPTR 与本类的偏移地址，如果本类有虚函数，那么第一项是FF FF FF FC(也就是-4)，如果没有则是零，第二项起是VBPTR与本类的虚基类的偏移值。**

## 下面这段代码与上面菱形继承（非虚继承）类似:

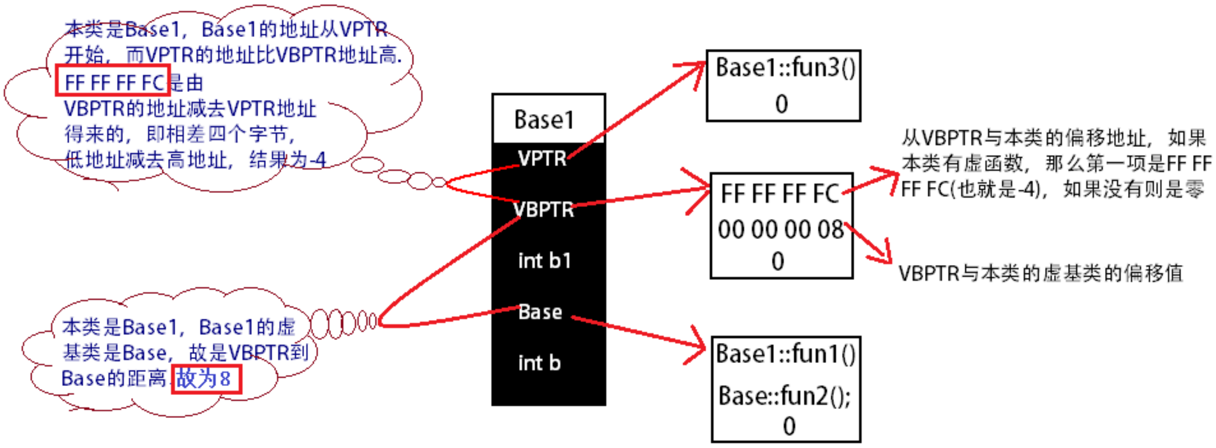
```
class Base
{
public:
    virtual void fun1()
    {
        cout << "Base::fun1()" << endl;
    }
    virtual void fun2()
    {
        cout << "Base::fun2()" << endl;
    }
private:
    int b;
};
class Base1 :virtual public Base  虚继承
{
public:
    virtual void fun1()        //重写Base的func1()
    {
        cout << "Base1::fun1()" << endl;
```

```
    }
    virtual void fun3()
    {
        cout << "Base1::fun3()" << endl;
    }
private:
    int b1;
};
class Base2 :virtual public Base  //虚继承
{
public:
    virtual void fun1()      //重写Base的func1()
    {
        cout << "Base2::fun1()" << endl;
    }
    virtual void fun4()
    {
        cout << "Base2::fun4()" << endl;
    }
private:
    int b2;
};
class Derive :public Base1, public Base2
{
public:
    virtual void fun1()         //重写Base1的func1()
    {
        cout << "Derive::fun1()" << endl;
    }
    virtual void fun5()
    {
        cout << "Derive::fun5()" << endl;
    }
private:
    int d;
};
```

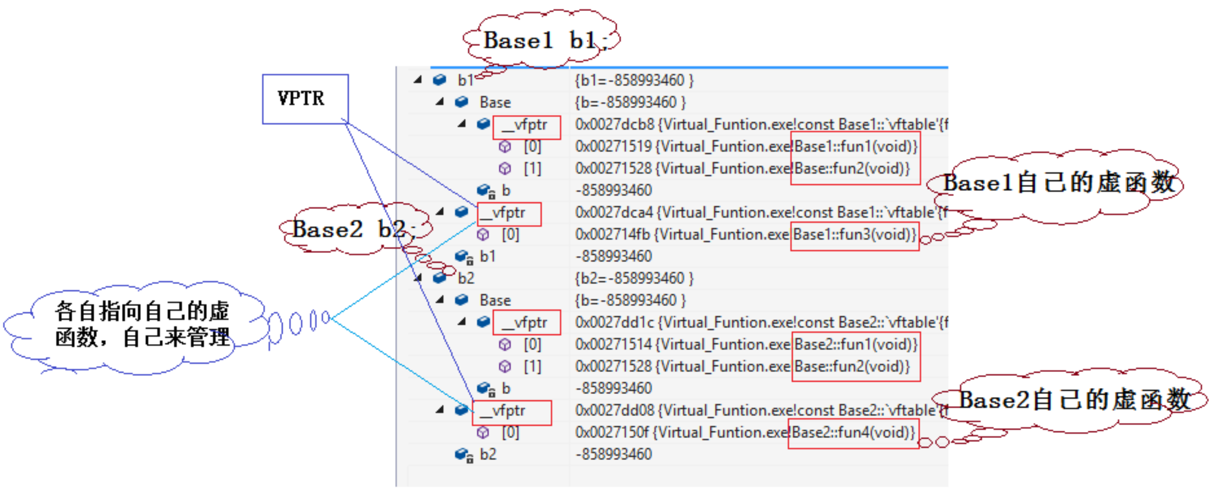
# 1. 详细地分析一下vs2003下虚继承的VBPTR及VBTBL：

## - 以Base1 b1;为例子，详细分析内存布局如下（Base2和Base1的内存布局相似）：

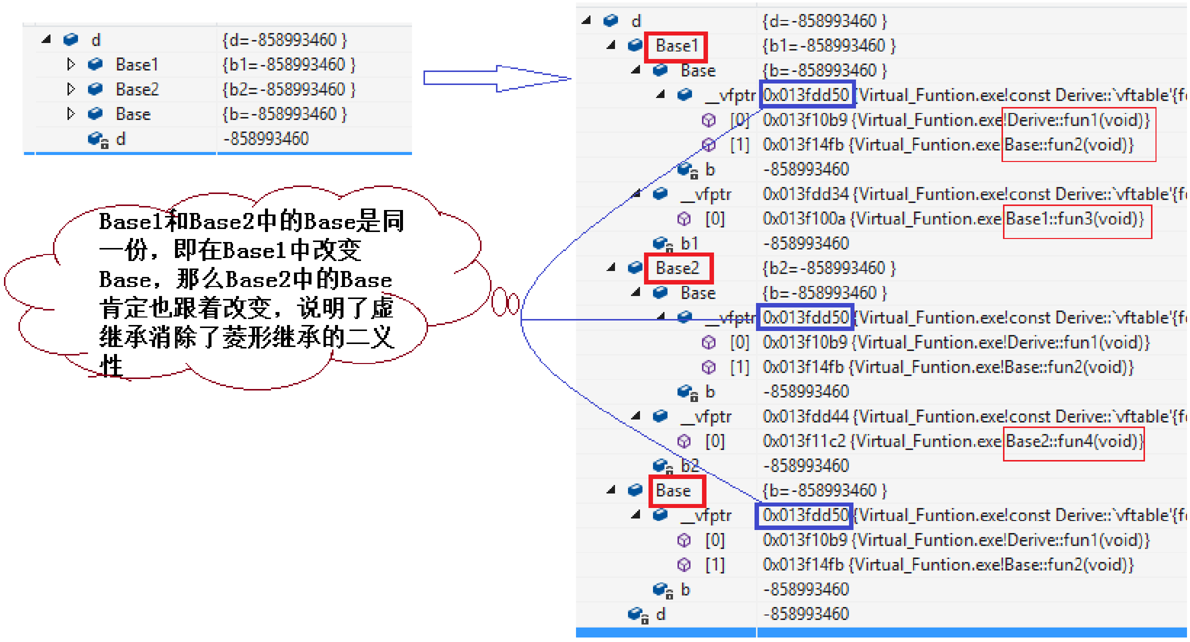
sizeof(Base1) = 20;(下图中黑色区域中所有变量所占的大小)




## 当在主函数中定义两个对象Base1 b1和Base2 b2时，还可通过调试进一步探索其内存布局如下：

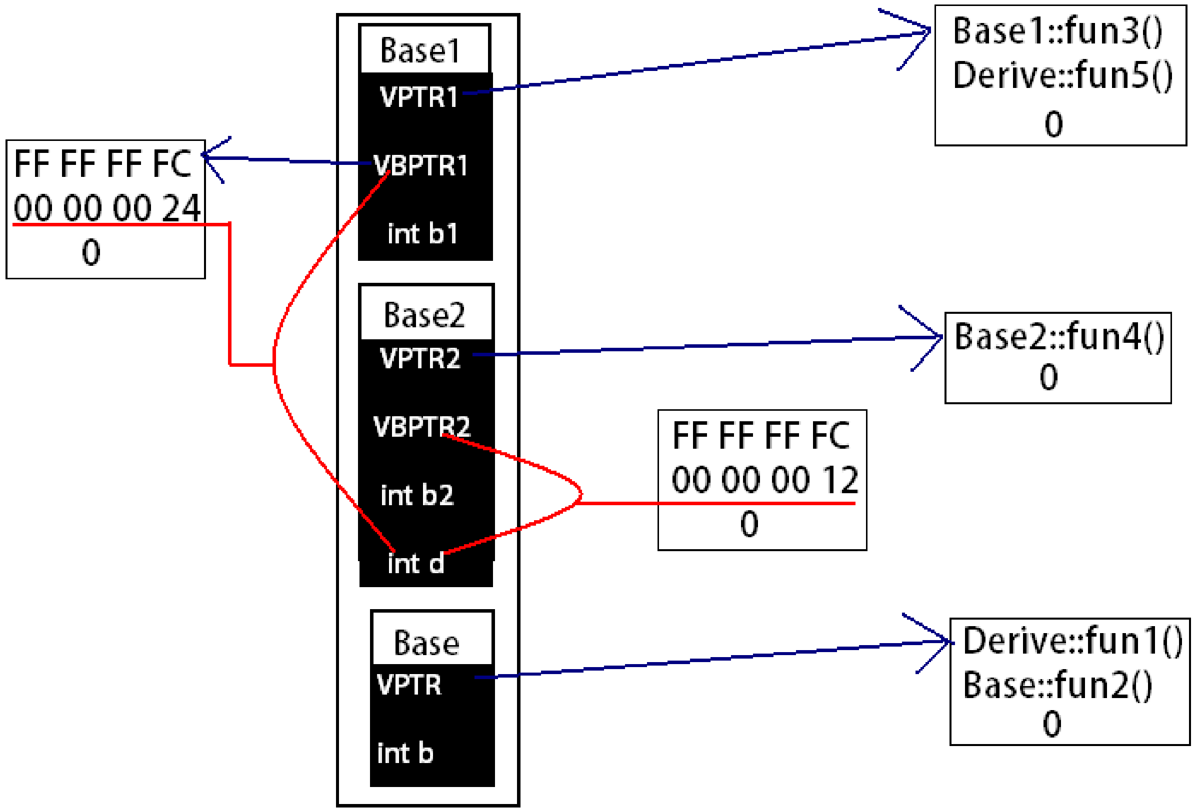



## 最后，我们再来探索一下 Derive d 的内存布局，首先我们先通过调试窗口来跟踪如下：




**通过上面调试窗口可能没办法了解全部，那么请看下图所示：**

**sizeof(Derive) = 36;(下图黑色区域所有变量的大小)**




**我们怎么验证菱形继承（虚继承）的内存布局就是这样的呢？我们可以通过打印虚表！！**

```
typedef void(*FUNC)();
void PrintVPTR(int* VPTR)                //打印虚表（虚函数）
```

```
{
    cout << "虚函数表地址: " << VPTR << endl;
    for (int i = 0; VPTR[i] != 0; ++i)
    {
        printf("第%d个虚函数地址: 0X%x->", i, VPTR[i]);
        FUNC f = (FUNC)VPTR[i];
        f();
    }
    cout << endl;
}
void PrintVBPTR(int* VBPTR)                //打印偏移地址与值
{
    cout << "虚函数表地址: " << VBPTR << endl;
    int i = 0;
    printf("与本类的偏移地址: 0X%x\n", VBPTR[i]);
    for (i = 1; VBPTR[i] != 0; i++)
    {
        cout << VBPTR[i] << " " << endl;
    }
    cout << endl;
}
```

# 主函数中的调用如下：

```
void Test1()
{
    Base b;
    Base1 b1;
    Base2 b2;
    Derive d;
    cout << "sizeof_Base = " << sizeof(Base) << endl;
    int* BvTable = (int*)(*((int*)&b));
    PrintVPTR(BvTable);
    cout << "--------------------------------" << endl;

    cout << "sizeof_Base1 = " << sizeof(Base1) << endl;
    int* BVPTR1 = (int*)(*((int*)&b1));                //存放自己的虚函数（虚表）
    PrintVPTR(BVPTR1);
    int* VBPTR1 = (int*)(*((int*)&b1 + 1));//访问偏移地址以及偏移量
    PrintVBPTR(VBPTR1);
```

```
    int* VPTR1 = (int*)(*((int*)&b1 + (*(VBPTR1 + 1)) / 4 + 1));    //在Base1中访问Base虚表
    PrintVPTR(VPTR1);
    cout << "--------------------------------" << endl;

    cout << "sizeof_Base2 = " << sizeof(Base2) << endl;
    int* BVPTR2 = (int*)(*((int*)&b2));                //存放自己的虚函数（虚表）
    PrintVPTR(BVPTR2);
    int* VBPTR2 = (int*)(*((int*)&b2 + 1));//访问偏移地址以及偏移量
    PrintVBPTR(VBPTR2);
    int* VPTR2 = (int*)(*((int*)&b2 + (*(VBPTR2 + 1)) / 4 + 1));//在Base2中访问Base虚表
    PrintVPTR(VPTR2);
    cout << "--------------------------------" << endl;

    cout << "sizeof_Derive = " << sizeof(Derive) << endl;
    int* dVPTR1 = (int*)(*((int*)&d));                //存放自己的虚函数（虚表）
    PrintVPTR(dVPTR1);
    int* dVBPTR3 = (int*)(*((int*)&d + 1));//访问偏移地址以及偏移量
    PrintVBPTR(dVBPTR3);
    int* dVPTR2 = (int*)(*((int*)&d + 3));              //在Derive中访问Base2虚表
    PrintVPTR(dVPTR2);
    int* dVBPTR = (int*)(*((int*)&d + 4));//访问偏移地址以及偏移量
    PrintVBPTR(dVBPTR);
    int* VPTR = (int*)(*((int*)&d + (*(dVBPTR3 + 1)) / 4 + 1));  //在Derive中访问Base虚表
    PrintVPTR(VPTR);
}
```

## 总结：

1. **虚基类实例地址 = 派生类虚函数指针+派生类虚函数指针到虚基类实例地址的偏移量**
2. **可以通过虚拟继承消除二义性，但是虚拟继承的开销是增加虚函数指针。**