

图说C++对象模型：对象内存布局详解

0.前言

文章较长，而且内容相对来说比较枯燥，希望对C++对象的内存布局、虚表指针、虚基类指针等有深入了解的朋友可以慢慢看。

本文的结论都在VS2013上得到验证。不同的编译器在内存布局的细节上可能有所不同。

文章如果有解释不清、解释不通或疏漏的地方，恳请指出。

1.何为C++对象模型？

引用《深度探索C++对象模型》这本书中的话：

有两个概念可以解释C++对象模型：

1. 语言中直接支持面向对象程序设计的部分。
2. 对于各种支持的底层实现机制。

直接支持面向对象程序设计，包括了构造函数、析构函数、多态、虚函数等等，这些内容在很多书籍上都有讨论，也是C++最被人熟知的地方（特性）。而对象模型的底层实现机制却是很少有书籍讨论的。对象模型的底层实现机制并未标准化，不同的编译器有一定的自由来设计对象模型的实现细节。在我看来，对象模型研究的是对象在存储上的空间与时间上的更优，并对C++面向对象技术加以支持，如以虚指针、虚表机制支持多态特性。

2.文章内容简介

这篇文章主要来讨论C++对象在内存中的布局，属于第二个概念的研究范畴。而C++直接支持面向对象程序设计部分则不多讲。文章主要内容如下：

- 虚函数表解析。含有虚函数或其父类含有虚函数的类，编译器都会为其添加一个虚函数表，vptr，先了解虚函数表的构成，有助对C++对象模型的理解。
- 虚基类表解析。虚继承产生虚基类表(vbptr)，虚基类表的内容与虚函数表完全不同，我们将在讲解虚继承时介绍虚函数表。
- 对象模型概述：介绍简单对象模型、表格驱动对象模型，以及非继承情况下的C++对象模型。
- 继承下的C++对象模型。分析C++类对象在下面情形中的内存布局：
 1. 单继承：子类单一继承自父类，分析了子类重写父类虚函数、子类定义了新的虚函数情况下子类对象内存布局。
 2. 多继承：子类继承于多个父类，分析了子类重写父类虚函数、子类定义了新的虚函数情况下子类对象内存布局，同时分析了非虚继承下的菱形继承。
 3. 虚继承：分析了单一继承下的虚继承、多重基类下的虚继承、重复继承下的虚继承。
- 理解对象的内存布局之后，我们可以分析一些问题：
 1. C++封装带来的布局成本是多大？
 2. 由空类组成的继承层次中，每个类对象的大小是多大？

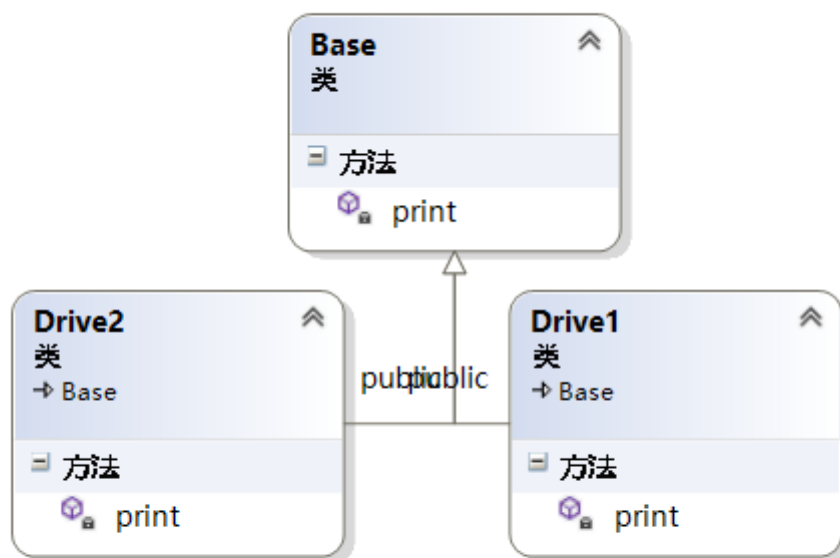
至于其他与内存有关的知识，我假设大家都有一定的了解，如内存对齐，指针操作等。本文初看可能晦涩难懂，要求读者有一定的C++基础，对概念一有一定的掌握。

3.理解虚函数表

3.1.多态与虚表

C++中虚函数的作用主要是为了实现多态机制。多态，简单来说，是指在继承层次中，父类的指针可以具有多种形态——当它指向某个子类对象时，通过它能够调用到子类的函数，而非父类的函数。

```
1 class Base {      virtual void print(void);    }
2 class Drive1 :public Base{    virtual void print(void);    }
3 class Drive2 :public Base{    virtual void print(void);    }
4 Base * ptr1 = new Base;
5 Base * ptr2 = new Drive1;
6 Base * ptr3 = new Drive2;
7 ptr1->print(); //调用Base::print()
8 prt2->print();//调用Drive1::print()
9 prt3->print();//调用Drive2::print()
```



这是一种运行期多态，即父类指针唯有在程序运行时才能知道所指的真正类型是什么。这种运行期决议，是通过虚函数表来实现的。

3.2.使用指针访问虚表

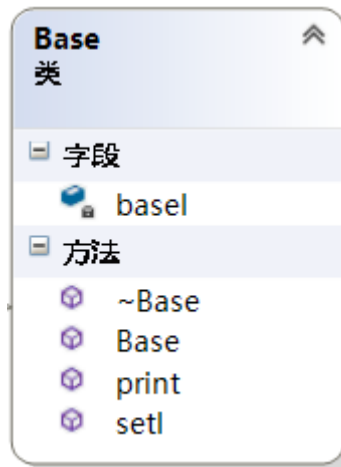
如果我们丰富我们的Base类,使其拥有多个virtual函数:

```
1 class Base
2 {
3 public:
4
5     Base(int i) :baseI(i){};
6
7     virtual void print(void){ cout << "调用了虚函数Base::print()"; }
8
9     virtual void setI(){cout<<"调用了虚函数Base::setI()";}
10
11     virtual ~Base(){}
12
```

```

13 private:
14
15     int baseI;
16
17 };

```



当一个类本身定义了虚函数，或其父类有虚函数时，为了支持多态机制，编译器将为该类添加一个虚函数指针（vptr）。虚函数指针一般都放在对象内存布局的第一个位置上，这是为了保证在多层继承或多重继承的情况下能以最高效率取到虚函数表。

当vptr位于对象内存最前面时，对象的地址即为虚函数指针地址。我们可以取得虚函数指针的地址：

```

1 Base b(1000);
2 int * vptrAdree = (int *)(&b);
3 cout << "虚函数指针（vptr）的地址是：\t"<<vptrAdree << endl;

```

我们运行代码出结果：

```

虚函数指针（vptr）的地址是：    00ABF8DC

```

我们强行把类对象的地址转换为 int* 类型，取得了虚函数指针的地址。虚函数指针指向虚函数表，虚函数表中存储的是一系列虚函数的地址，虚函数地址出现的顺序与类中虚函数声明的顺序一致。对虚函数指针地址值，可以得到虚函数表的地址，也即是虚函数表第一个虚函数的地址：

```

1 typedef void(*Fun)(void);
2 Fun vfunc = (Fun)* ( (int *)*(int*)(&b));
3 cout << "第一个虚函数的地址是：" << (int *)*(int*)(&b) << endl;
4 cout << "通过地址，调用虚函数Base::print()："；
5 vfunc();

```

- 我们把虚表指针的值取出来：*(int *)(&b)，它是一个地址，虚函数表的地址
- 把虚函数表的地址强制转换成 int*：(int *)*(int *)(&b)
- 再把它转化成我们Fun指针类型：(Fun)*(int *)*(int *)(&b)

这样，我们就取得了类中的第一个虚函数，我们可以通过函数指针访问它。

运行结果：

```

第一个虚函数的地址是：0016DC78
通过地址，调用虚函数Base::print()：调用了虚函数Base::print()。

```

同理,第二个虚函数setI()的地址为：

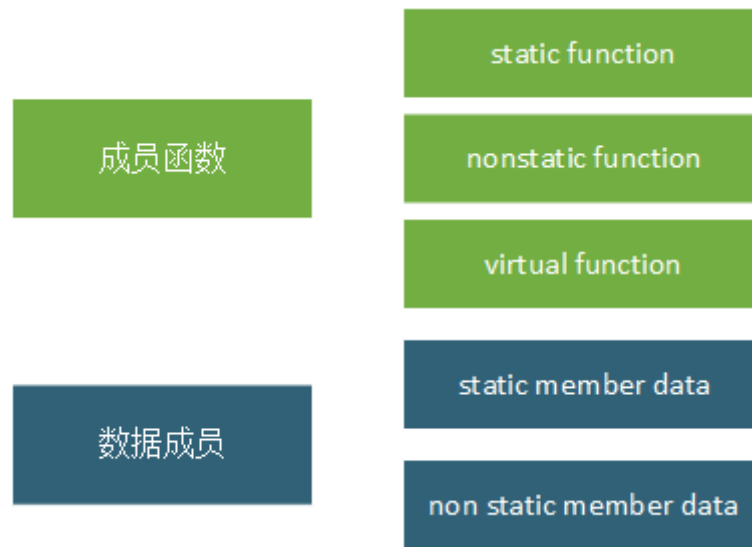
```
1 | (int * )(* (int*)(&b)+1)
```

同样可以通过函数指针访问它，这里留给读者自己试验。

到目前为止，我们知道了类中虚表指针vpert的由来，知道了虚函数表中的内容，以及如何通过指针访问虚函数表。下面的文章中将常使用指针访问对象内存来验证我们的C++对象模型，以及讨论在各种继承情况下虚表指针的变化，先把这部分的内容消化完紧接着看下面的内容。

4.对象模型概述

在C++中，有两种数据成员（class data members）：static 和nonstatic,以及三类成员函数（class member functions）:static、nonstatic和virtual:



现在有一个类Base，它包含了上面这5中类型的数据或函数：

```
1 | class Base
2 | {
3 | public:
4 |
5 |     Base(int i) :baseI(i){};
6 |
7 |     int getI(){ return baseI; }
8 |
9 |     static void countI(){};
10 |
11 |     virtual ~Base(){}
12 |
13 |     virtual void print(void){ cout << "Base::print()"; }
14 |
15 |
16 |
17 | private:
18 |
19 |     int baseI;
20 |
21 |     static int bases;
22 | };
```



那么，这个类在内存中将被如何表示？5种数据都是连续存放的吗？如何布局才能支持C++多态？我们的C++标准与编译器将如何塑造出各种数据成员与成员函数呢？

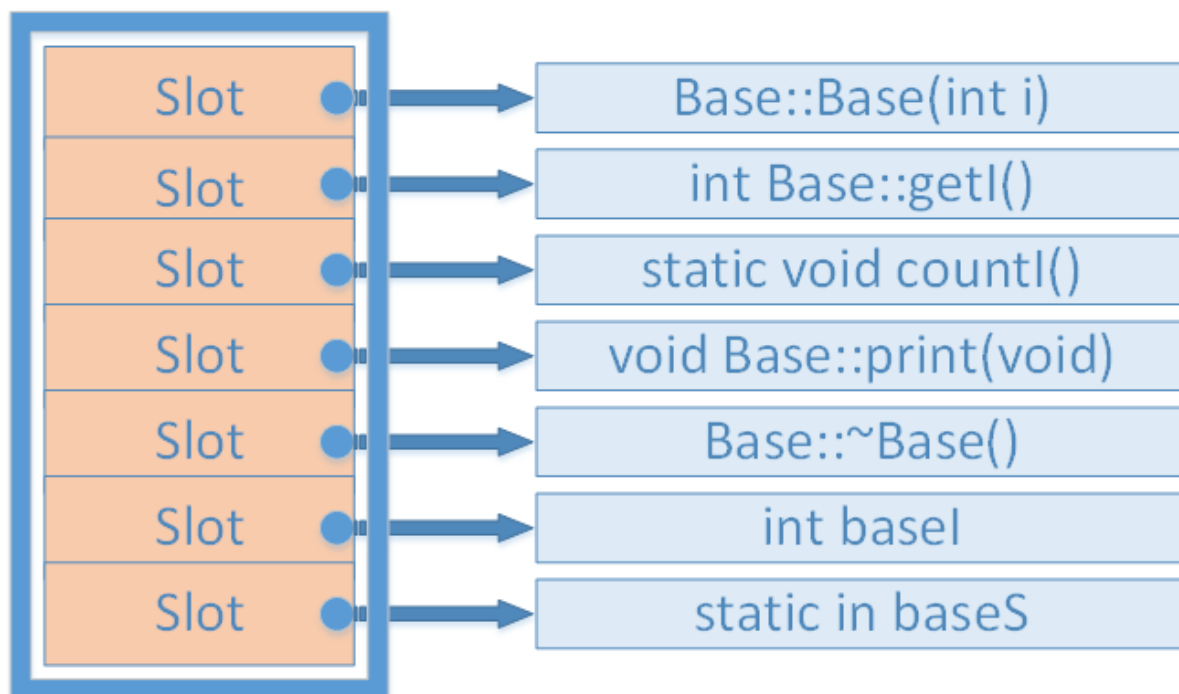
4.1.简单对象模型

说明：在下面出现的图中，用蓝色边框框起来的内容在内存上是连续的。

这个模型非常地简单粗暴。在该模型下，对象由一系列的指针组成，每一个指针都指向一个数据成员或成员函数，也即是说，每个数据成员和成员函数在类中所占的大小是相同的，都为指针的大小。这样有个好处——很容易算出对象的大小，不过赔上的是空间和执行期效率。想象一下，如果我们的Point3d类是这种模型，将会比C语言的struct多了许多空间来存放指向函数的指针，而且每次读取类的数据成员，都需要通过再一次寻址——又是时间上的消耗。

所以这种对象模型并没有被用于实际产品上。

Base类内存布局



简单对象模型

4.2.表格驱动模型

这个模型在简单对象模型的基础上又添加一个间接层，它把类中的数据分成了两个部分：数据部分与函数部分，并使用两张表格，一张存放数据本身，一张存放函数的地址（也即函数比成员多一次寻址），而类对象仅仅含有两个指针，分别指向上面这两个表。这样看来，对象的大小是固定为两个指针大小。这个模型也没有用于实际应用于真正的C++编译器上。

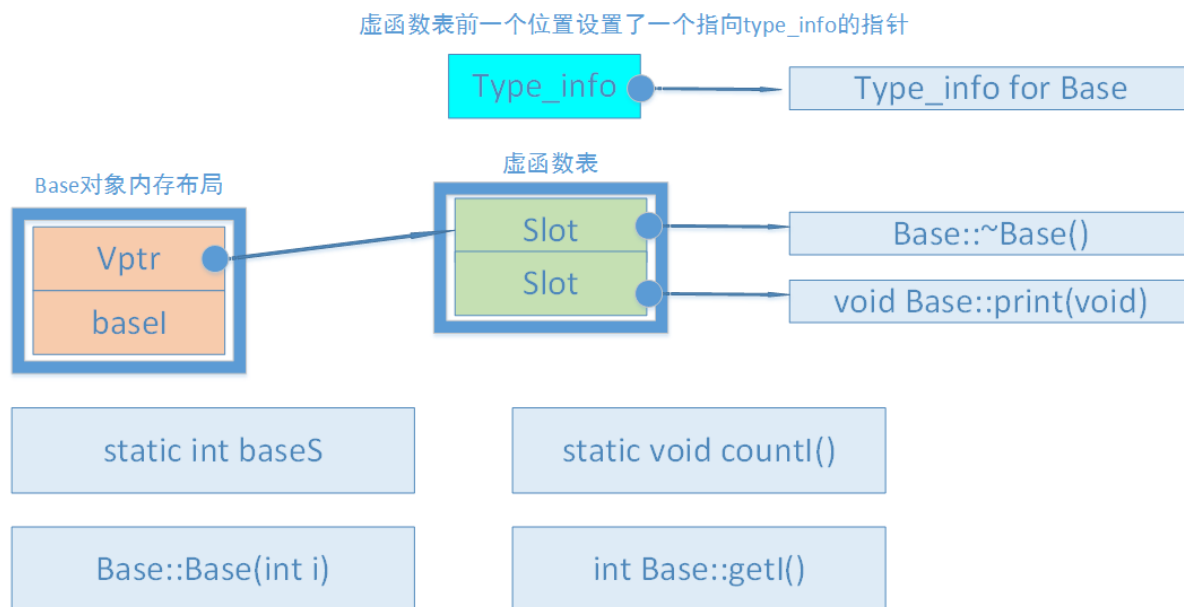
4.3.非继承下的C++对象模型

概述：在此模型下，nonstatic 数据成员被置于每一个类对象中，而static数据成员被置于类对象之外。static与nonstatic函数也都放在类对象之外，而对于virtual 函数，则通过虚函数表+虚指针来支持，具体如下：

- 每个类生成一个表格，称为虚表（virtual table，简称vtbl）。虚表中存放着一堆指针，这些指针指向该类每一个虚函数。虚表中的函数地址将按声明时的顺序排列，不过当子类有多个重载函数时例外，后面会讨论。
- 每个类对象都拥有一个虚表指针(vptr)，由编译器为其生成。虚表指针的设置与重置皆由类的复制控制（也即是构造函数、析构函数、赋值操作符）来完成。vptr的位置为编译器决定，传统上它被放在所有显示声明的成员之后，不过现在许多编译器把vptr放在一个类对象的最前端。关于数据成员布局的内容，在后面会详细分析。

另外，虚函数表的前面设置了一个指向type_info的指针，用以支持RTTI（Run Time Type Identification，运行时类型识别）。RTTI是为多态而生成的信息，包括对象继承关系，对象本身的描述等，只有具有虚函数的对象在会生成。

在此模型下，Base的对象模型如图：



C++对象模型

先在VS上验证类对象的布局：

```
1 | Base b(1000);
```

名称	值	类型
b	{baseI= 1000 }	Base
__vfptr	0x00bfdc78 {boke1.exe\const Base::`vftable' } {0x00bf1131 void **	void **
baseI	1000	int

可见对象b含有一个vfptr，即vprrt。并且只有nonstatic数据成员被放置于对象内。我们展开vfprt：

名称	值	类型
▲ b	{baseI=1000 }	Base
▲ _vfptr	0x00bdc78 {boke1.exe!const Base::vtable} {0x00bf1131}	void **
[0]	0x00bf1131 {boke1.exe!Base::~vector deleting destructor'}	void *
[1]	0x00bf123f {boke1.exe!Base::print(void)}	void *
baseI	1000	int

vfptr中有两个指针类型的数据（地址），第一个指向了Base类的析构函数，第二个指向了Base的虚函数print，顺序与声明顺序相同。

这与上述的C++对象模型相符合。也可以通过代码来进行验证：

```

1 void testBase( Base&p)
2 {
3     cout << "对象的内存起始地址: " << &p << endl;
4     cout << "type_info信息:" << endl;
5     RTTICompleteObjectLocator str = *((RTTICompleteObjectLocator*)((int*)*
(int*)&p) - 1));
6
7
8     string classname(str.pTypeDescriptor->name);
9     classname = classname.substr(4, classname.find("@@" ) - 4);
10    cout << "根据type_info信息输出类名:"<< classname << endl;
11
12    cout << "虚函数表地址:" << (int *)(&p) << endl;
13
14    //验证虚表
15    cout << "虚函数表第一个函数的地址: " << (int *)*((int*)&p) << endl;
16    cout << "析构函数的地址:" << (int *)*((int*)&p) << endl;
17    cout << "虚函数表中，第二个虚函数即print () 的地址: " << ((int*)*((int*)&p) +
1) << endl;
18
19    //通过地址调用虚函数print ()
20    typedef void(*Fun)(void);
21    Fun IsPrint=(Fun)* ((int*)*((int*)&p) + 1);
22    cout << endl;
23    cout<<"调用了虚函数";
24    IsPrint(); //若地址正确，则调用了Base类的虚函数print ()
25    cout << endl;
26
27    //输入static函数的地址
28    p.countI(); //先调用函数以产生一个实例
29    cout << "static函数countI()的地址: " << p.countI << endl;
30
31    //验证nonstatic数据成员
32    cout << "推测nonstatic数据成员baseI的地址: " << (int *)(&p) + 1 << endl;
33    cout << "根据推测出的地址，输出该地址的值: " << *((int *)(&p) + 1) << endl;
34    cout << "Base::getI():" << p.getI() << endl;
35
36 }
37 Base b(1000);
38 testBase(b);

```

```
C:\Users\jmelonstreet\documents\visual studio 2013\Projects\boke1\Deb
对象的内存起始地址: 002CFAEC
type_info信息:
根据type_info信息输出类名:Base
虚函数表地址:002CFAEC
虚函数表第一个函数的地址: 00A80DB8
析构函数的地址:00A71168
虚函数表中, 第二个虚函数即print()的地址: 00A80DBC

调用了虚函数Base::print()
static函数countI()的地址: 00A7111D
推测nonstatic数据成员baseI的地址: 002CFAF0
根据推测出的地址, 输出该地址的值: 1000
Base::getI():1000
```

结果分析:

- 通过 (int *)(&p)取得虚函数表的地址
- type_info信息的确存在于虚表的前一个位置。通过(((int)(int*)(&p) - 1))取得type_info信息, 并成功获得类的名称的Base
- 虚函数表的第一个函数是析构函数。
- 虚函数表的第二个函数是虚函数print(), 取得地址后通过地址调用它 (而非通过对象), 验证正确
- 虚表指针的下一个位置为nonstatic数据成员baseI。
- 可以看到, static成员函数的地址段位与虚表指针、baseI的地址段位不同。

好的, 至此我们了解了非继承下类对象五种数据在内存上的布局, 也知道了在每一个虚函数表前都有一个指针指向type_info, 负责对RTTI的支持。而加入继承后类对象在内存中该如何表示呢?

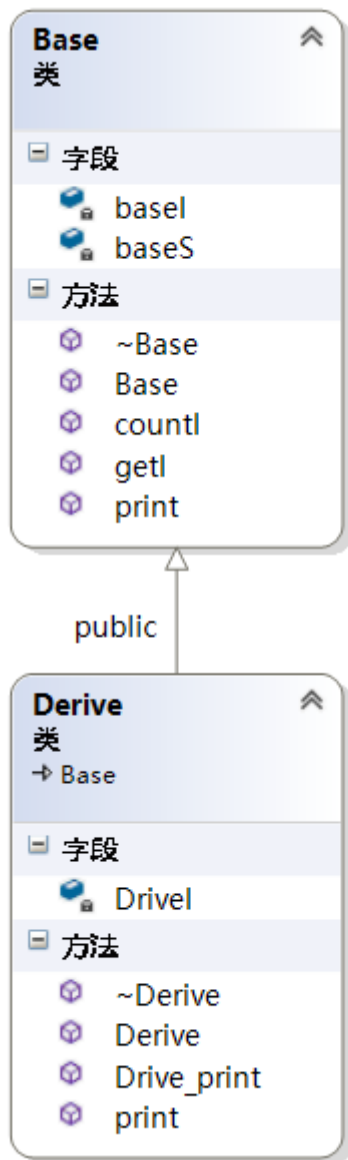
5.继承下的C++对象模型

5.1.单继承

如果我们定义了派生类

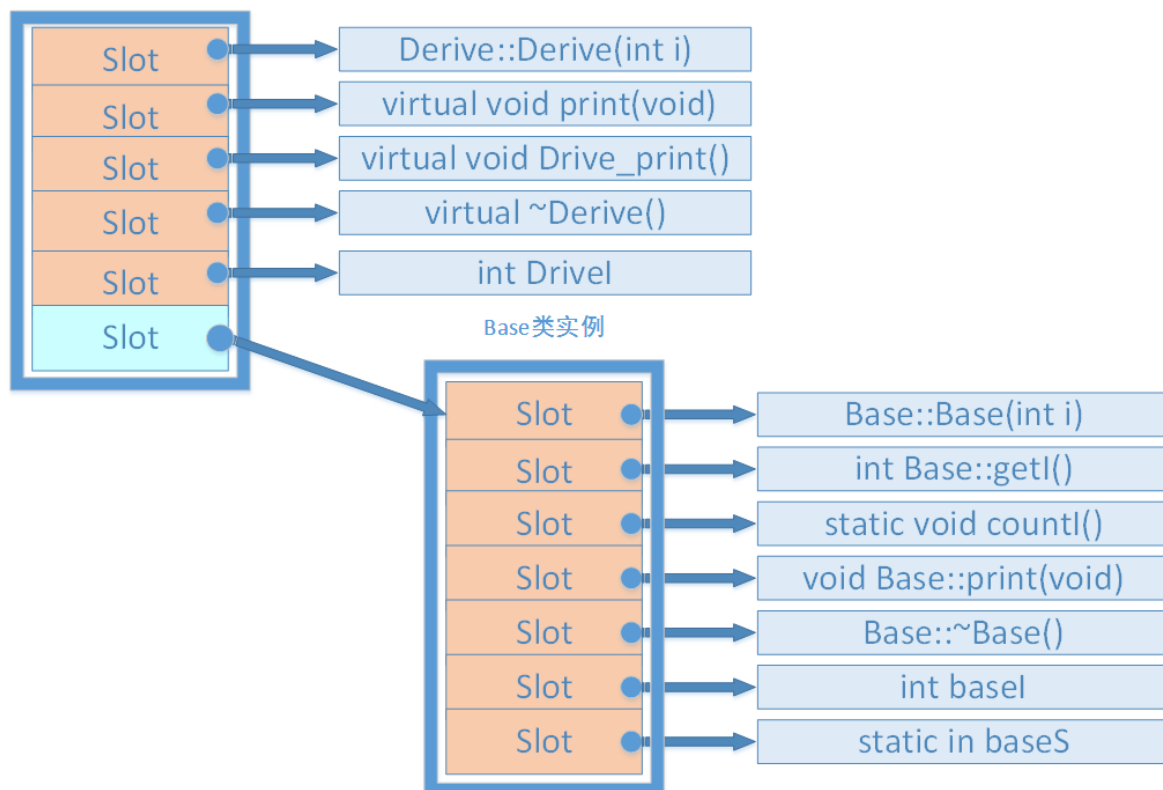
```
1  class Derive : public Base
2  {
3  public:
4      Derive(int d) :Base(1000),      DeriveI(d){};
5      //overwrite父类虚函数
6      virtual void print(void){ cout << "Drive::Drive_print()" ; }
7      // Derive声明的新的虚函数
8      virtual void Drive_print(){ cout << "Drive::Drive_print()" ; }
9      virtual ~Derive(){}
10 private:
11     int DeriveI;
12 };
```


继承类图为：



一个派生类如何在机器层面上塑造其父类的实例呢？在简单对象模型中，可以在子类对象中为每个基类子对象分配一个指针。如下图：

Derive类对象内存布局

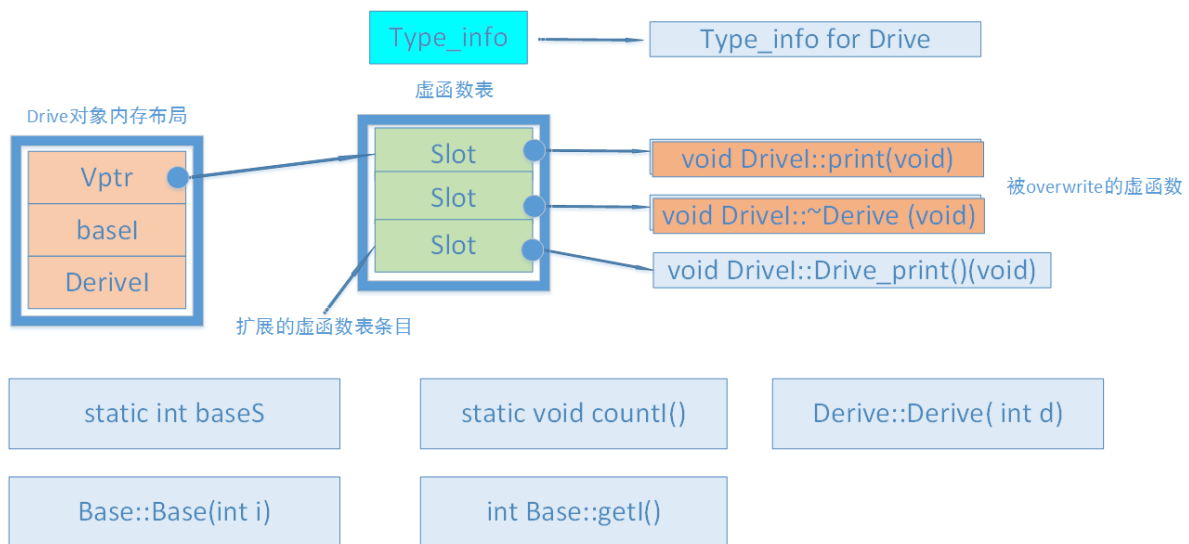


为每个父类添加一个slot

简单对象模型的缺点就是因间接性导致的空间存取时间上的额外负担，优点则是类的大小是固定的，基类的改动不会影响子类对象的大小。

在表格驱动对象模型中，我们可以为子类对象增加第三个指针：基类指针(bptr)，基类指针指向指向一个基类表(base class table),同样的，由于间接性导致了空间和存取时间上的额外负担，优点则是无须改变子类对象本身就可以更改基类。表格驱动模型的图就不再贴出来了。

在C++对象模型中，对于一般继承（这个一般是相对于虚拟继承而言），若子类重写（overwrite）了父类的虚函数，则子类虚函数将覆盖虚表中对应的父类虚函数(注意子类与父类拥有各自的一个虚函数表)；若子类并无overwrite父类虚函数，而是声明了自己新的虚函数，则该虚函数地址将扩充到虚函数表最后（在vs中无法通过监视看到扩充的结果，不过我们通过取地址的方法可以做到，子类新的虚函数确实在父类子物体的虚函数表末端）。而对于虚继承，若子类overwrite父类虚函数，同样地将覆盖父类子物体中的虚函数表对应位置，而若子类声明了自己新的虚函数，则编译器将为子类增加一个新的虚表指针vp ptr，这与一般继承不同,在后面再讨论。



简单继承下的C++对象模型

我们使用代码来验证以上模型

```

1  typedef void(*Fun)(void);
2
3  int main()
4  {
5      Derive d(2000);
6      //[0]
7      cout << "[0]Base::vptr";
8      cout << "\t地址: " << (int *)(&d) << endl;
9      //vpvt[0]
10     cout << " [0]";
11     Fun fun1 = (Fun)*((int *)*((int *)(&d)));
12     fun1();
13     cout << "\t地址:\t" << *((int *)*((int *)(&d))) << endl;
14
15     //vpvt[1]析构函数无法通过地址调用, 故手动输出
16     cout << " [1]" << "Derive::~Derive" << endl;
17
18     //vpvt[2]
19     cout << " [2]";
20     Fun fun2 = (Fun)*((int *)*((int *)(&d)) + 2);
21     fun2();
22     cout << "\t地址:\t" << *((int *)*((int *)(&d)) + 2) << endl;
23     //[1]
24     cout << "[2]Base::baseI=" << *(int*)((int *)(&d) + 1);
25     cout << "\t地址: " << (int *)(&d) + 1;
26     cout << endl;
27     //[2]
28     cout << "[2]Derive::DeriveI=" << *(int*)((int *)(&d) + 2);
29     cout << "\t地址: " << (int *)(&d) + 2;
30     cout << endl;
31     getchar();
32 }

```

运行结果：

```
[0]Base::vptr 地址: 00D5FEC0
[0]Drive::Drive_print() 地址: 11145361
[1]Derive::~~Derive
[2]Drive::Drive_print() 地址: 11145771
[2]Base::baseI=1000 地址: 00D5FEC4
[2]Derive::DeriveI=2000 地址: 00D5FEC8
```

这个结果与我们的对象模型符合。

5.2.多继承

5.2.1一般的多重继承（非菱形继承）

单继承中（一般继承），子类会扩展父类的虚函数表。在多重继承中，子类含有多个父类的子对象，该往哪个父类的虚函数表扩展呢？当子类overwrites了父类的函数，需要覆盖多个父类的虚函数表吗？

- 子类的虚函数被放在声明的第一个基类的虚函数表中。
- overwrite时，所有基类的print()函数都被子类的print()函数覆盖。
- 内存布局中，父类按照其声明顺序排列。

其中第二点保证了父类指针指向子类对象时，总是能够调用到真正的函数。

为了方便查看，我们把代码都粘贴过来

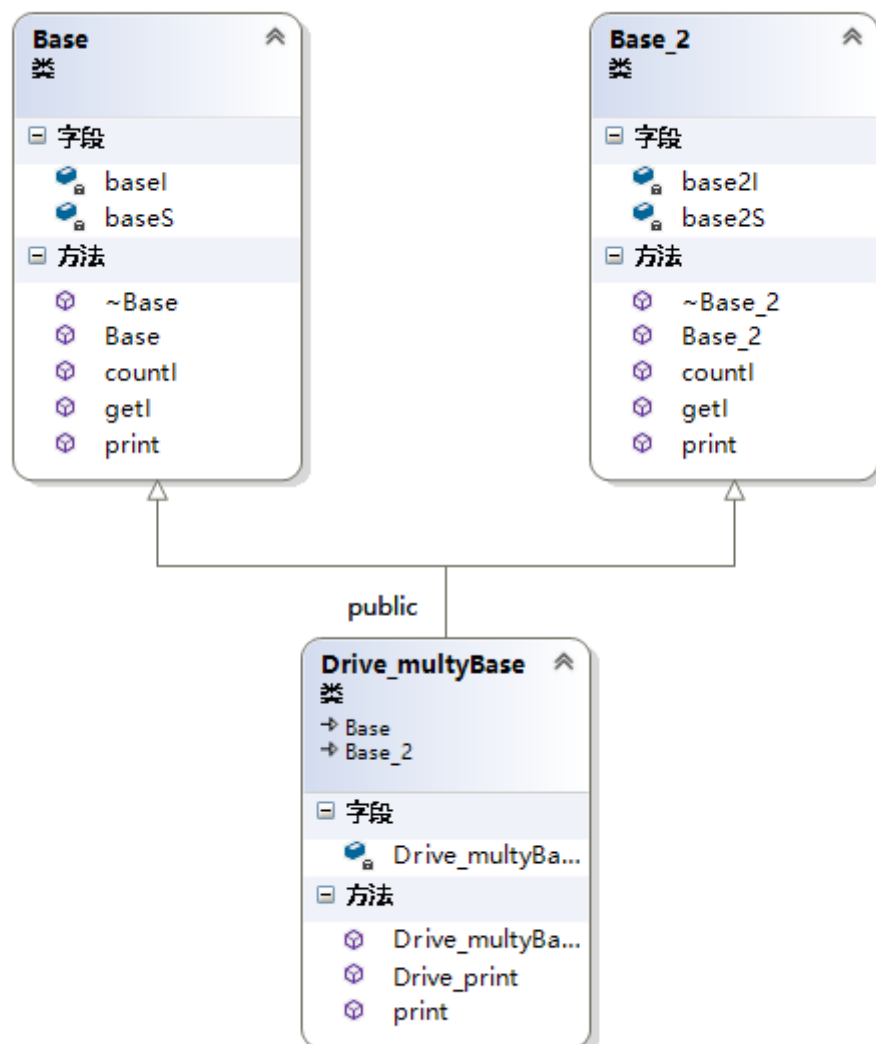
```
1  class Base
2  {
3  public:
4
5      Base(int i) :baseI(i){};
6      virtual ~Base(){}
7
8      int getI(){ return baseI; }
9
10     static void countI(){};
11
12     virtual void print(void){ cout << "Base::print()"; }
13
14 private:
15
16     int baseI;
17
18     static int bases;
19 };
20 class Base_2
21 {
22 public:
23     Base_2(int i) :base2I(i){};
24
25     virtual ~Base_2(){}
26
27     int getI(){ return base2I; }
28
29     static void countI(){};
30
```

```

31     virtual void print(void){ cout << "Base_2::print()"; }
32
33 private:
34
35     int base2I;
36
37     static int base2S;
38 };
39
40 class Drive_multyBase :public Base, public Base_2
41 {
42 public:
43
44     Drive_multyBase(int d) :Base(1000), Base_2(2000) ,Drive_multyBaseI(d){};
45
46     virtual void print(void){ cout << "Drive_multyBase::print" ; }
47
48     virtual void Drive_print(){ cout << "Drive_multyBase::Drive_print" ; }
49
50 private:
51     int Drive_multyBaseI;
52 };

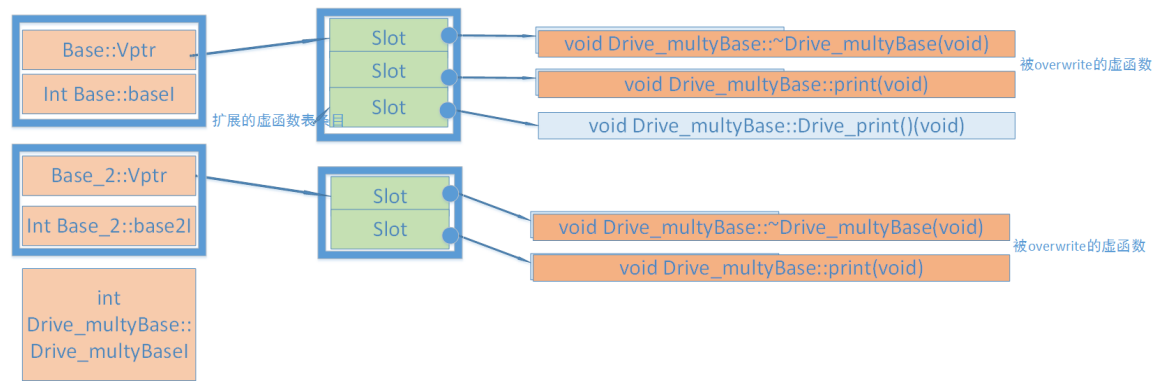
```

继承类图为：



此时Drive_multyBase 的对象模型是这样的：

Drive_multyBase类内存布局



多重继承下的对象模型

我们使用代码验证：

```
1  typedef void(*Fun)(void);
2
3  int main()
4  {
5      Drive_multyBase d(3000);
6      //[0]
7      cout << "[0]Base::vptr";
8      cout << "\t地址: " << (int*)(&d) << endl;
9
10     //vprt[0]析构函数无法通过地址调用，故手动输出
11     cout << " [0]" << "Derive::~~Derive" << endl;
12
13     //vprt[1]
14     cout << " [1]";
15     Fun fun1 = (Fun)*((int*)((int*)(&d))+1);
16     fun1();
17     cout << "\t地址:\t" << *((int*)((int*)(&d))+1) << endl;
18
19
20     //vprt[2]
21     cout << " [2]";
22     Fun fun2 = (Fun)*((int*)((int*)(&d)) + 2);
23     fun2();
24     cout << "\t地址:\t" << *((int*)((int*)(&d)) + 2) << endl;
25
26
27     //[1]
28     cout << "[1]Base::baseI=" << *(int*)((int*)(&d) + 1);
29     cout << "\t地址: " << (int*)(&d) + 1;
30     cout << endl;
31
32
33     //[2]
34     cout << "[2]Base_2::vptr";
35     cout << "\t地址: " << (int*)(&d)+2 << endl;
36
37     //vprt[0]析构函数无法通过地址调用，故手动输出
38     cout << " [0]" << "Drive_multyBase::~~Derive" << endl;
39
40     //vprt[1]
```

```

41         cout << "    [1]";
42         Fun fun4 = (Fun)*((int *)*((int *)(&d))+1);
43         fun4();
44         cout << "\t地址:\t" << *((int *)*((int *)(&d))+1) << endl;
45
46         //[3]
47         cout << "[3]Base_2::base2I=" << *(int*)((int *)(&d) + 3);
48         cout << "\t地址: " << (int *)(&d) + 3;
49         cout << endl;
50
51         //[4]
52         cout << "[4]Drive_multyBase::Drive_multyBaseI=" << *(int*)((int *)(&d) +
4);
53         cout << "\t地址: " << (int *)(&d) + 4;
54         cout << endl;
55
56         getchar();
57     }

```

运行结果:

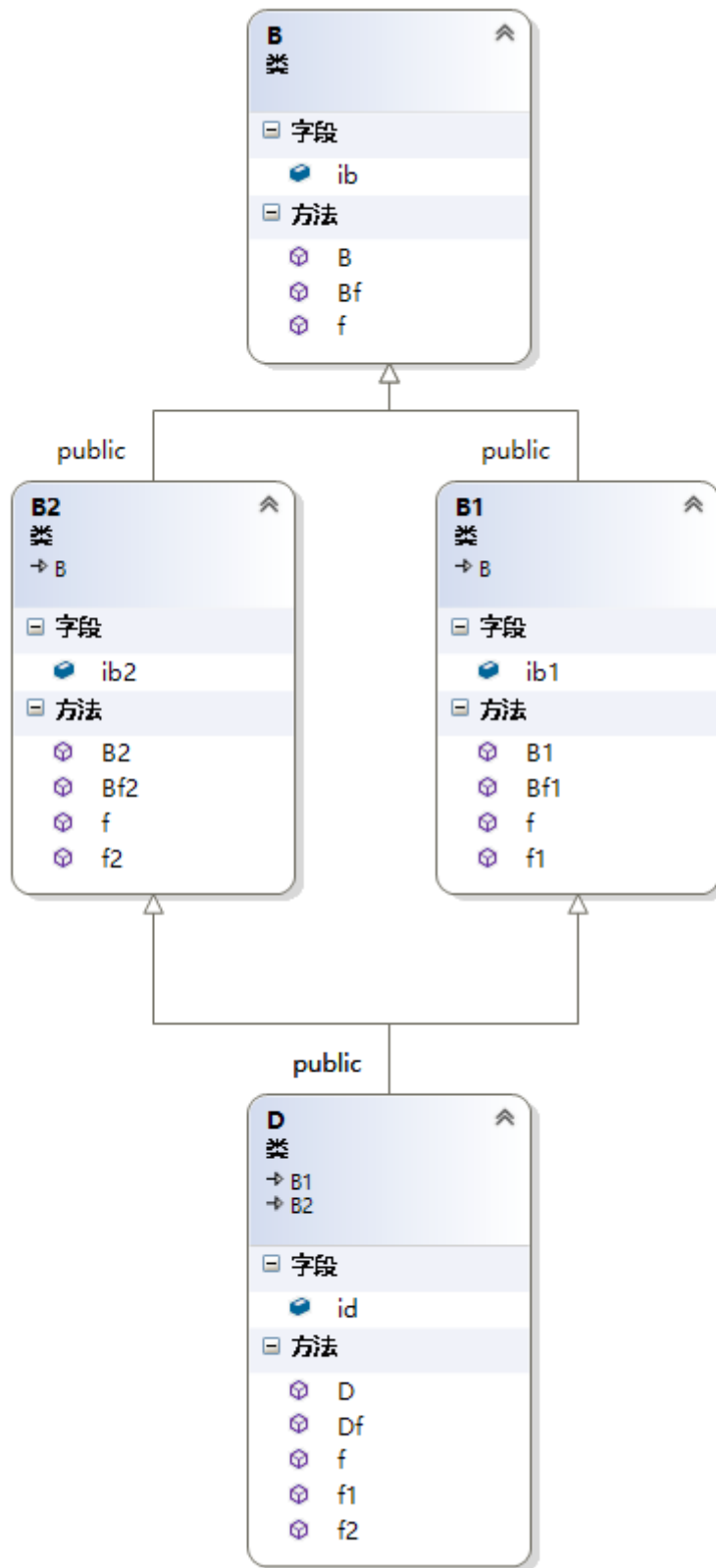
```

[0]Base::vptr  地址: 0022F890
[0]Derive::~Derive
[1]Drive_multyBase::print  地址: 14554383
[2]Drive_multyBase::Drive_print  地址: 14554358
[1]Base::baseI=1000  地址: 0022F894
[2]Base::vptr  地址: 0022F898
[0]Drive_multyBase::~Derive
[1]Drive_multyBase::print  地址: 14554383
[3]Base_2::base2I=2000  地址: 0022F89C
[4]Drive_multyBase::Drive_multyBaseI=3000  地址: 0022F8A0

```

5.2.2 菱形继承

菱形继承也称为钻石型继承或重复继承，它指的是基类被某个派生类简单重复继承了多次。这样，派生类对象中拥有多份基类实例（这会带来一些问题）。为了方便叙述，我们不使用上面的代码了，而重新写一个重复继承的继承层次：



```
1  class B
2
3  {
4
5  public:
6
7      int ib;
8
9  public:
10
```



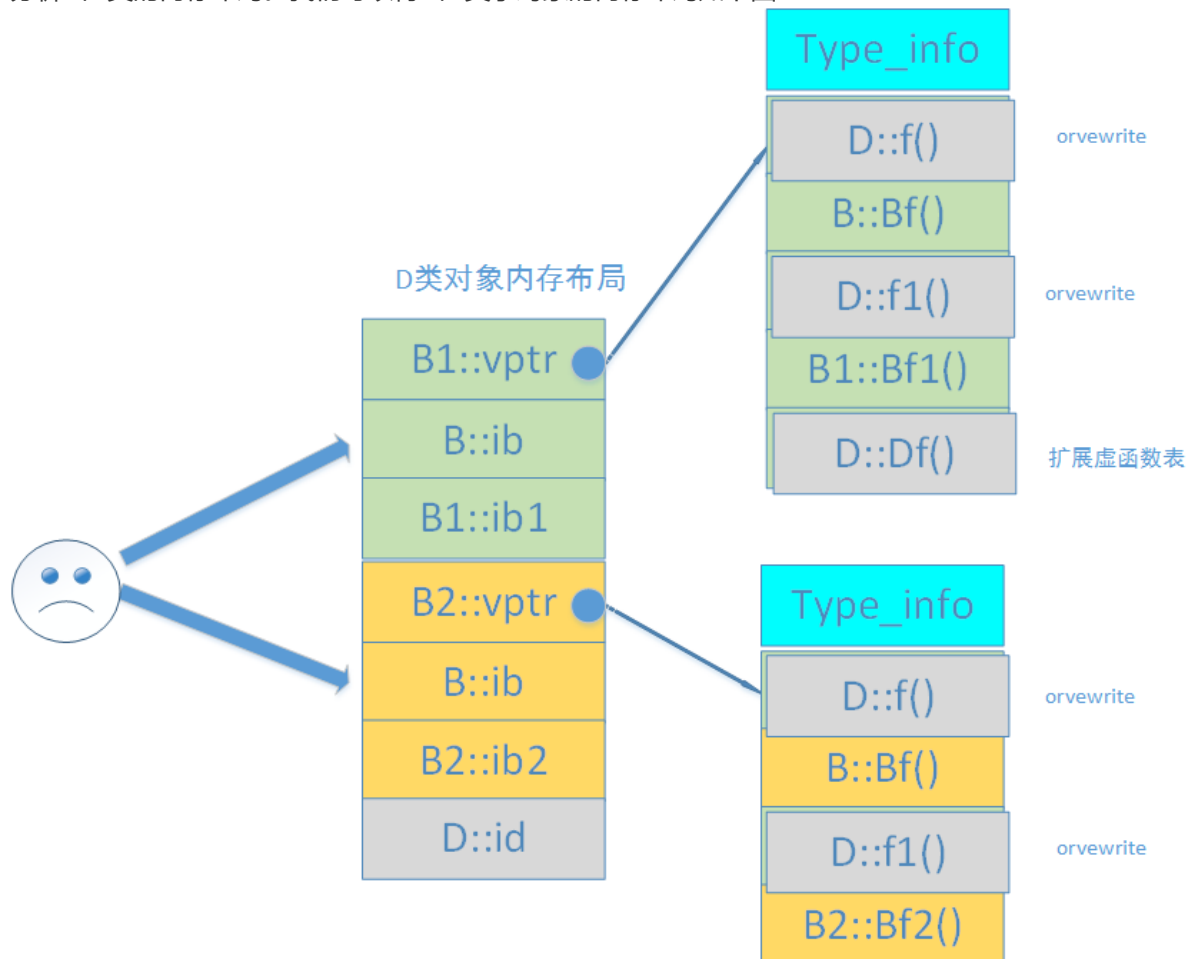
```
11     B(int i=1) :ib(i){}
12
13     virtual void f() { cout << "B::f()" << endl; }
14
15     virtual void Bf() { cout << "B::Bf()" << endl; }
16
17 };
18
19 class B1 : public B
20 {
21
22 public:
23
24     int ib1;
25
26 public:
27
28     B1(int i = 100 ) :ib1(i) {}
29
30     virtual void f() { cout << "B1::f()" << endl; }
31
32     virtual void f1() { cout << "B1::f1()" << endl; }
33
34     virtual void Bf1() { cout << "B1::Bf1()" << endl; }
35
36
37
38
39 };
40
41 class B2 : public B
42 {
43
44 public:
45
46     int ib2;
47
48 public:
49
50     B2(int i = 1000) :ib2(i) {}
51
52     virtual void f() { cout << "B2::f()" << endl; }
53
54     virtual void f2() { cout << "B2::f2()" << endl; }
55
56     virtual void Bf2() { cout << "B2::Bf2()" << endl; }
57
58
59 };
60
61
62 class D : public B1, public B2
63 {
64
65 public:
66
67     int id;
```

```

69
70
71
72 public:
73
74     D(int i= 10000) :id(i){}
75
76     virtual void f() { cout << "D::f()" << endl; }
77
78     virtual void f1() { cout << "D::f1()" << endl; }
79
80     virtual void f2() { cout << "D::f2()" << endl; }
81
82     virtual void Df() { cout << "D::Df()" << endl; }
83
84 };

```

这时，根据单继承，我们可以分析出B1，B2类继承于B类时的内存布局。又根据一般多继承，我们可以分析出D类的内存布局。我们可以得出D类子对象的内存布局如下图：



菱形继承下的C++对象模型

D类对象内存布局中，图中青色表示b1类子对象实例，黄色表示b2类子对象实例，灰色表示D类子对象实例。从图中可以看到，由于D类间接继承了B类两次，导致D类对象中含有两个B类的数据成员ib，一个属于来源B1类，一个来源B2类。这样不仅增大了空间，更重要的是引起了程序歧义：

```

1 D d;
2
3 d.ib = 1;           //二义性错误,调用的是B1的ib还是B2的ib?
4
5 d.B1::ib = 1;       //正确
6
7 d.B2::ib = 1;       //正确

```

尽管我们可以通过明确指明调用路径以消除二义性，但二义性的潜在性还没有消除，我们可以通过虚继承来使D类只拥有一个ib实体。

6.虚继承

虚继承解决了菱形继承中最派生类拥有多个间接父类实例的情况。虚继承的派生类的内存布局与普通继承很多不同，主要体现在：

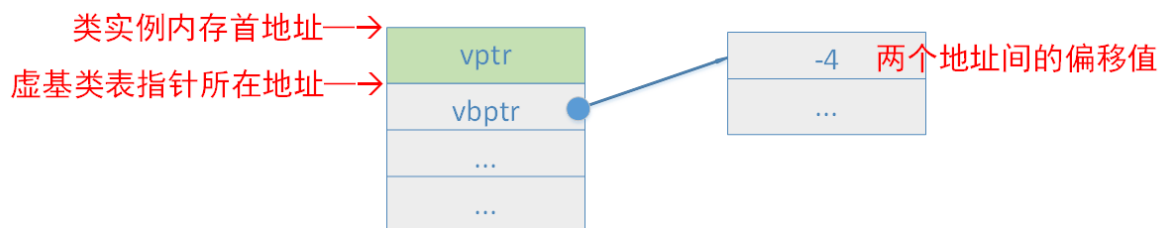
- 虚继承的子类，如果本身定义了新的虚函数，则编译器为其生成一个虚函数指针（vptr）以及一张虚函数表。该vptr位于对象内存最前面。
 - vs非虚继承：直接扩展父类虚函数表。
- 虚继承的子类也单独保留了父类的vptr与虚函数表。这部分内容接与子类内容以一个四字节的0来分界。
- 虚继承的子类对象中，含有四字节的虚表指针偏移值。

为了分析最后的菱形继承，我们还是先从单虚继承继承开始。

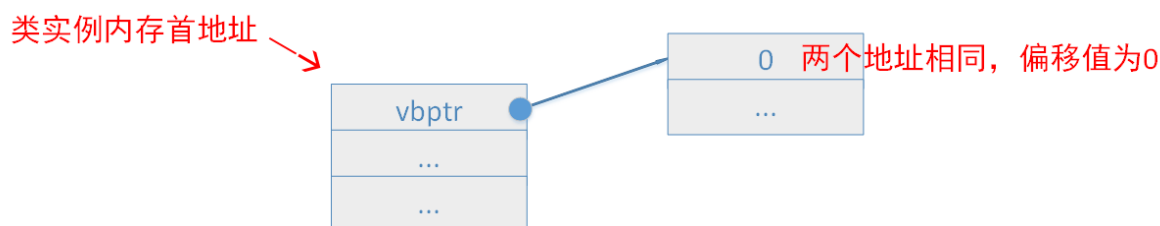
6.1.虚基类表解析

在C++对象模型中，虚继承而来的子类会生成一个隐藏的虚基类指针（vbptr），在Microsoft Visual C++中，**虚基类表指针总是在虚函数表指针之后**，因而，对某个类实例来说，如果它有虚基类指针，那么虚基类指针可能在实例的0字节偏移处（该类没有vptr时，vbptr就处于类实例内存布局的最前面，否则vptr处于类实例内存布局的最前面），也可能在类实例的4字节偏移处。

一个类的虚基类指针指向的虚基类表，与虚函数表一样，虚基类表也由多个条目组成，条目中存放的是**偏移值**。第一个条目存放虚基类表指针（vbptr）所在地址到该类内存首地址的偏移值，由第一段的分析我们知道，这个偏移值为0（类没有vptr）或者-4（类有虚函数，此时有vptr）。我们通过一张图来更好地理解。



类实例含有vptr的情况



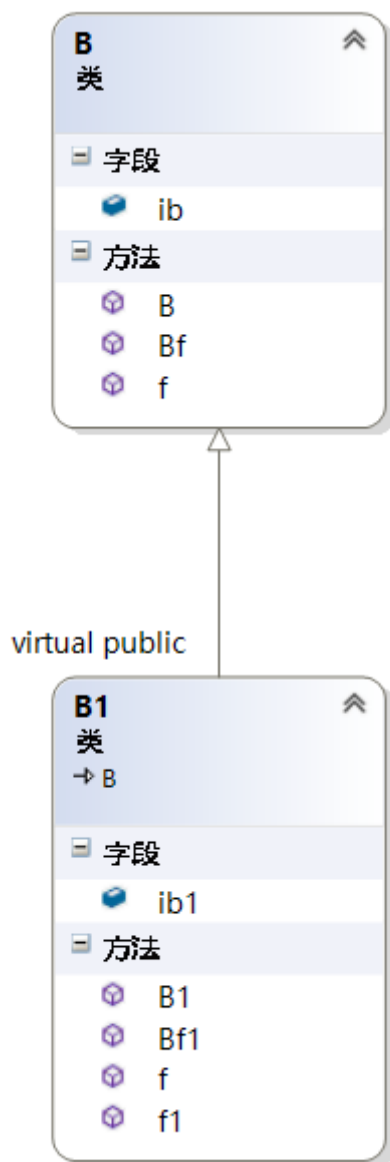
类实例不含vptr的情况

虚基类表的第二、第三...个条目依次为该类的最左虚继承父类、次左虚继承父类...的内存地址相对于虚基类表指针的偏移值，这点我们在下面会验证。

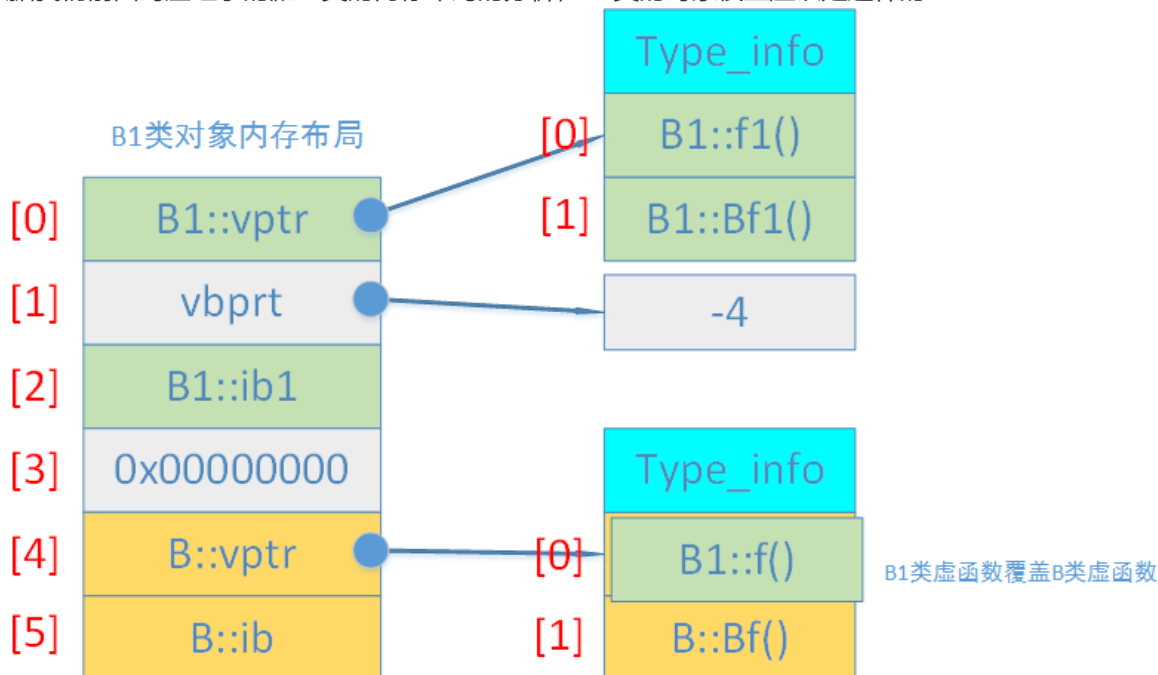
6.2.简单虚继承

如果我们的B1类虚继承于B类：

```
1 //类的内容与前面相同
2 class B{...}
3 class B1 : virtual public B
```



根据我们前面对虚继承的派生类的内存布局的分析，B1类的对象模型应该是这样的：



单虚拟继承下B1类对象的对象模型

我们通过指针访问B1类对象的内存，以验证上面的C++对象模型：

```
1  int main()
2  {
3      B1 a;
4      cout << "B1对象内存大小为: " << sizeof(a) << endl;
5
6      //取得B1的虚函数表
7      cout << "[0]B1::vptr";
8      cout << "\t地址: " << (int *)(&a) << endl;
9
10     //输出虚表B1::vptr中的函数
11     for (int i = 0; i < 2; ++ i)
12     {
13         cout << " [" << i << "]";
14         Fun fun1 = (Fun)*((int *)*(int *)(&a) + i);
15         fun1();
16         cout << "\t地址: \t" << *((int *)*(int *)(&a) + i) << endl;
17     }
18
19     //[1]
20     cout << "[1]vbprt " ;
21     cout << "\t地址: " << (int *)(&a) + 1 << endl; //虚表指针的地址
22     //输出虚基类指针条目所指的内容
23     for (int i = 0; i < 2; i++)
24     {
25         cout << " [" << i << "]";
26
27         cout << *((int *)*((int *)*(int *)(&a) + 1) + i);
28
29         cout << endl;
30     }
```

```

31
32
33 // [2]
34 cout << "[2]B1::ib1=" << *(int*)((int *)&a) + 2;
35 cout << "\t地址: " << (int *)&a + 2;
36 cout << endl;
37
38 // [3]
39 cout << "[3]值=" << *(int*)((int *)&a) + 3;
40 cout << "\t\t地址: " << (int *)&a + 3;
41 cout << endl;
42
43 // [4]
44 cout << "[4]B::vptr";
45 cout << "\t地址: " << (int *)&a + 3 << endl;
46
47 // 输出B::vptr中的虚函数
48 for (int i = 0; i < 2; ++i)
49 {
50     cout << "    [" << i << "]";
51     Fun fun1 = (Fun)*((int *)((int *)&a) + 4) + i;
52     fun1();
53     cout << "\t\t地址:\t" << *((int *)((int *)&a) + 4) + i << endl;
54 }
55
56 // [5]
57 cout << "[5]B::ib=" << *(int*)((int *)&a) + 5;
58 cout << "\t地址: " << (int *)&a + 5;
59 cout << endl;

```

运行结果:

```

B1对象内存大小为: 24
[0]B1::vptr      地址: 007CFDF0
  [0]B1::f1()    地址: 12653789
  [1]B1::Bf1()   地址: 12653794
[1]vbptr        地址: 007CFDF4
  [0]-4
  [1]12
[2]B1::ib1=100  地址: 007CFDF8
[3]值=0         地址: 007CFDFC
[4]B::vptr      地址: 007CFDFC
  [0]B1::f()     地址: 12653804
  [1]B::Bf()     地址: 12653299
[5]B::ib=1      地址: 007CFE04

```

这个结果与我们的C++对象模型图完全符合。这时我们可以来分析一下虚表指针的第二个条目值12的具体来源了，回忆上文讲到的：

第二、第三...个条目依次为该类的最左虚继承父类、次左虚继承父类...的内存地址相对于虚基类表指针的偏移值。

在我们的例子中，也就是B类实例内存地址相对于vbptr的偏移值，也即是：[4]-[1]的偏移值，结果即为12，从地址上也可以计算出来：007CFDFC-007CFDF4结果的十进制数正是12。现在，我们对虚基类表的构成应该有了一个更好的理解。

6.3.虚拟菱形继承

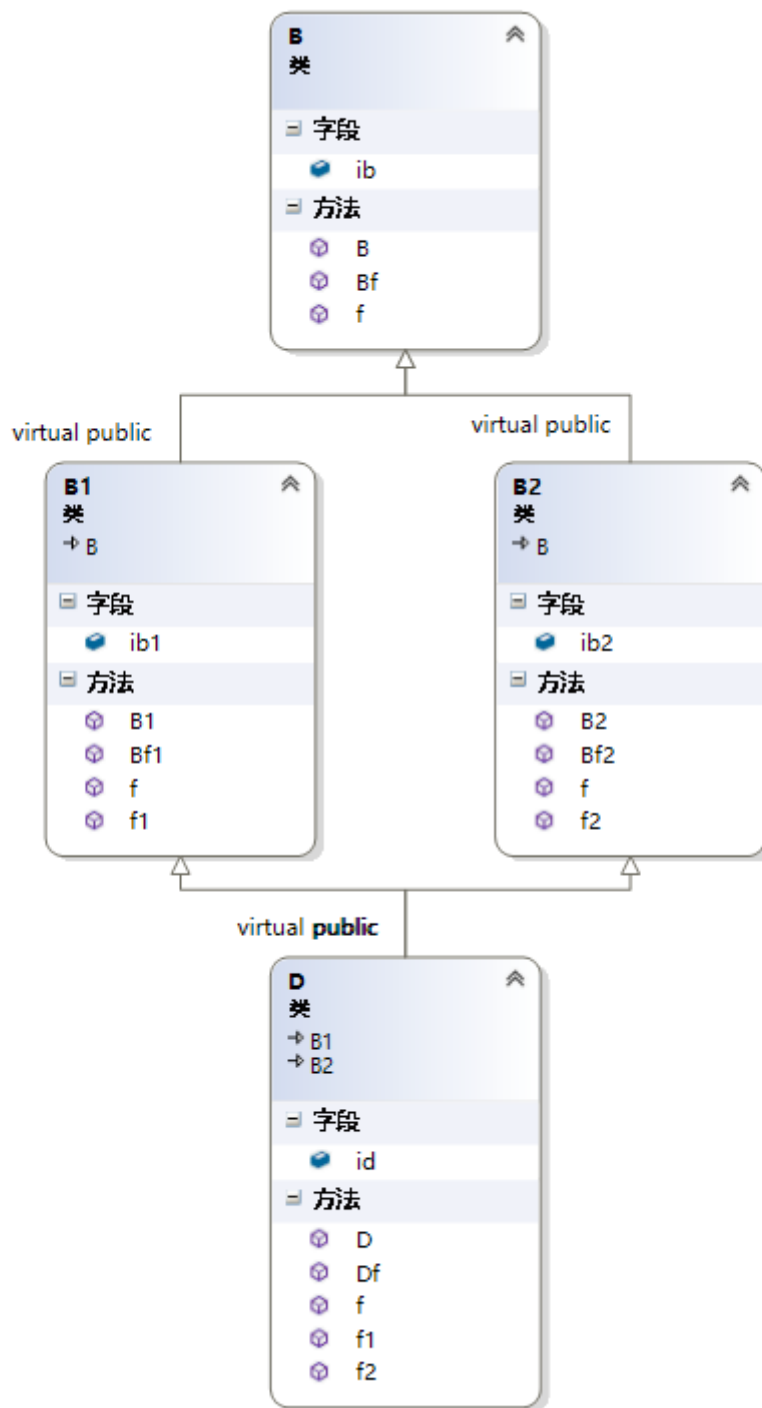
如果我们有如下继承层次：

```

1 class B{...}
2 class B1: virtual public B{...}
3 class B2: virtual public B{...}
4 class D : public B1,public B2{...}

```

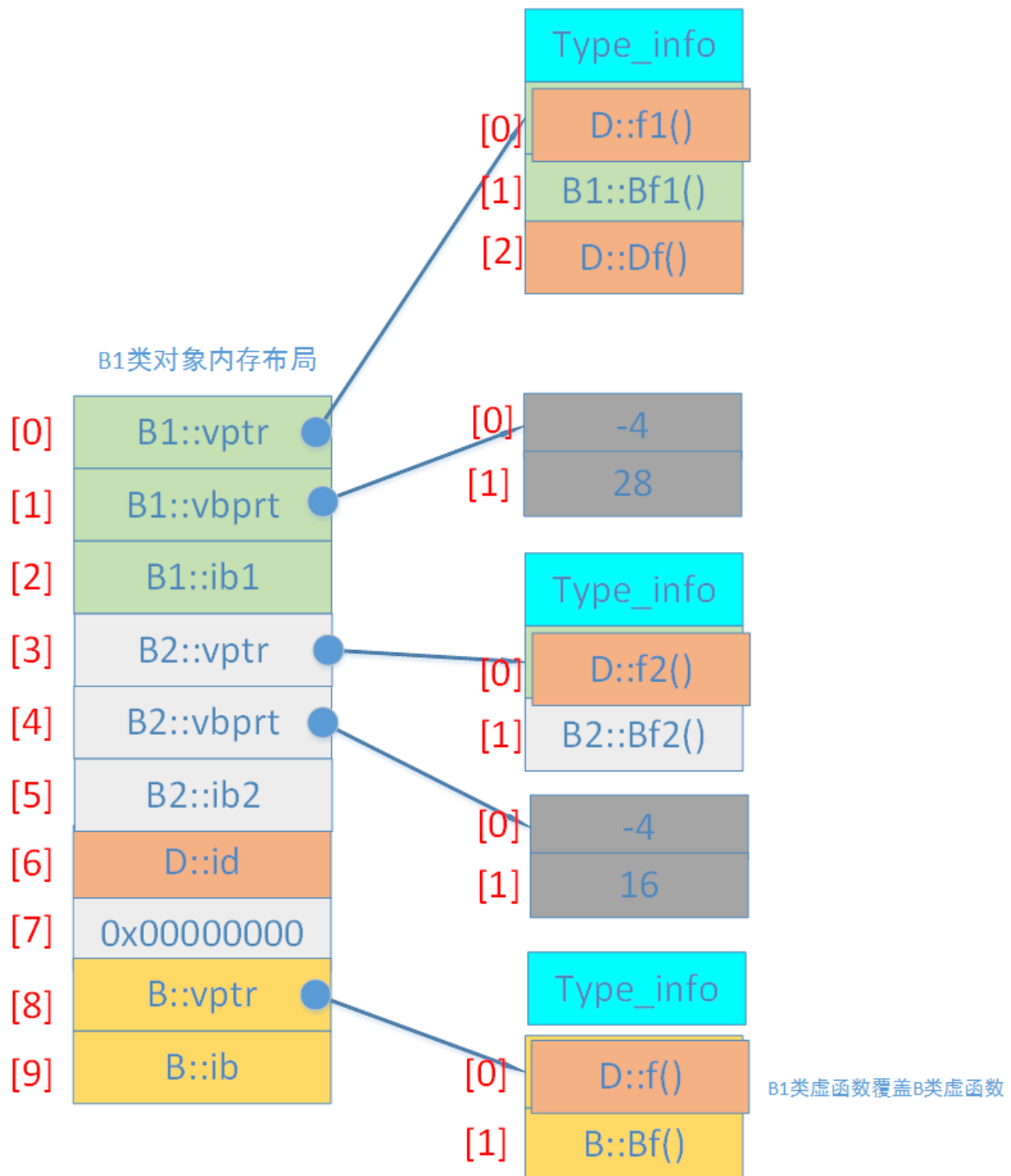
类图如下所示：



菱形虚拟继承下，最派生类D类的对象模型又有不同的构成了。在D类对象的内存构成上，有以下几点：

- 在D类对象内存中，基类出现的顺序是：先是B1（最左父类），然后是B2（次左父类），最后是B（虚祖父类）
- D类对象的数据成员`id`放在B类前面，两部分数据依旧以0来分隔。
- 编译器没有为D类生成一个它自己的`vpitr`，而是覆盖并扩展了最左父类的虚基类表，与简单继承的对象模型相同。
- 超类B的内容放到了D类对象内存布局的最后。

菱形虚拟继承下的C++对象模型为：



菱形虚拟继承下D类对象的对象模型

下面使用代码加以验证：

```
1  int main()
2  {
3      D d;
4      cout << "D对象内存大小为: " << sizeof(d) << endl;
5
6      //取得B1的虚函数表
7      cout << "[0]B1::vptr";
8      cout << "\t地址: " << (int *)&d << endl;
9
10     //输出虚表B1::vptr中的函数
11     for (int i = 0; i<3; ++i)
```



```

12     {
13         cout << "    [" << i << "]\n";
14         Fun fun1 = (Fun)*((int *)*(int *)&d + i);
15         fun1();
16         cout << "\t地址: \t" << *((int *)*(int *)&d + i) << endl;
17     }
18
19     //[1]
20     cout << "[1]B1::vbptr ";
21     cout << "\t地址: " << (int *)&d + 1 << endl;    //虚表指针的地址
22     //输出虚基类指针条目所指的内容
23     for (int i = 0; i < 2; i++)
24     {
25         cout << "    [" << i << "]\n";
26
27         cout << *((int *)*((int *)*(int *)&d + 1) + i);
28
29         cout << endl;
30     }
31
32
33     //[2]
34     cout << "[2]B1::ib1=" << *(int*)((int *)&d + 2);
35     cout << "\t地址: " << (int *)&d + 2;
36     cout << endl;
37
38     //[3]
39     cout << "[3]B2::vptra";
40     cout << "\t地址: " << (int *)&d + 3 << endl;
41
42     //输出B2::vptra中的虚函数
43     for (int i = 0; i < 2; ++i)
44     {
45         cout << "    [" << i << "]\n";
46         Fun fun1 = (Fun)*((int *)*((int *)&d + 3) + i);
47         fun1();
48         cout << "\t地址:\t" << *((int *)*((int *)&d + 3) + i) << endl;
49     }
50
51     //[4]
52     cout << "[4]B2::vbptr ";
53     cout << "\t地址: " << (int *)&d + 4 << endl;    //虚表指针的地址
54     //输出虚基类指针条目所指的内容
55     for (int i = 0; i < 2; i++)
56     {
57         cout << "    [" << i << "]\n";
58
59         cout << *((int *)*((int *)*(int *)&d + 4) + i);
60
61         cout << endl;
62     }
63
64     //[5]
65     cout << "[5]B2::ib2=" << *(int*)((int *)&d + 5);
66     cout << "\t地址: " << (int *)&d + 5;
67     cout << endl;
68
69     //[6]

```

```

70     cout << "[6]D::id=" << *(int*)((int *)&d) + 6);
71     cout << "\t地址: " << (int *)&d + 6;
72     cout << endl;
73
74     //[7]
75     cout << "[7]值=" << *(int*)((int *)&d) + 7);
76     cout << "\t\t地址: " << (int *)&d + 7;
77     cout << endl;
78
79     //间接父类
80     //[8]
81     cout << "[8]B::vptr";
82     cout << "\t地址: " << (int *)&d + 8 << endl;
83
84     //输出B::vptr中的虚函数
85     for (int i = 0; i < 2; ++i)
86     {
87         cout << "  [" << i << "]";
88         Fun fun1 = (Fun)*((int *)((int *)&d) + 8) + i);
89         fun1();
90         cout << "\t地址:\t" << *((int *)((int *)&d) + 8) + i) << endl;
91     }
92
93     //[9]
94     cout << "[9]B::id=" << *(int*)((int *)&d) + 9);
95     cout << "\t地址: " << (int *)&d + 9;
96     cout << endl;
97
98     getchar();
99 }
100

```

查看运行结果:

```

D对象内存大小为: 40
[0]B1::vptr    地址: 0041FB70
[0]D::f1()     地址: 8459545
[1]B1::Bf1()   地址: 8459490
[2]D::Df()     地址: 8459525
[1]B1::vbptr   地址: 0041FB74
[0]-4
[1]28
[2]B1::ib1=100 地址: 0041FB78
[3]B2::vptr    地址: 0041FB7C
[0]D::f2()     地址: 8459505
[1]B2::Bf2()   地址: 8459540
[4]B2::vbptr   地址: 0041FB80
[0]-4
[1]16
[5]B2::ib2=1000 地址: 0041FB84
[6]D::id=10000 地址: 0041FB88
[7]值=0        地址: 0041FB8C
[8]B::vptr     地址: 0041FB90
[0]D::f()      地址: 8459520
[1]B::Bf()     地址: 8458995
[9]B::id=1     地址: 0041FB94

```

7. 一些问题解答

7.1.C++封装带来的布局成本是多大？

在C语言中，“数据”和“处理数据的操作（函数）”是分开来声明的，也就是说，语言本身并没有支持“数据和函数”之间的关联性。

在C++中，我们通过类来将属性与操作绑定在一起，称为ADT，抽象数据结构。

C语言中使用struct（结构体）来封装数据，使用函数来处理数据。举个例子，如果我们定义了一个struct Point3如下：

```
1 typedef struct Point3
2 {
3     float x;
4     float y;
5     float z;
6 } Point3;
```

为了打印这个Point3d，我们可以定义一个函数：

```
1 void Point3d_print(const Point3d *pd)
2 {
3     printf("(%f,%f,%f)",pd->x,pd->y,pd->z);
4 }
```

而在C++中，我们更倾向于定义一个Point3d类，以ADT来实现上面的操作：

```
1 class Point3d
2 {
3     public:
4         point3d (float x = 0.0,float y = 0.0,float z = 0.0)
5             : _x(x), _y(y), _z(z){}
6
7         float x() const {return _x;}
8         float y() const {return _y;}
9         float z() const {return _z;}
10
11     private:
12         float _x;
13         float _y;
14         float _z;
15 };
16
17 inline ostream&
18 operator<<(ostream &os, const Point3d &pt)
19 {
20     os<<"("<<pr.x()<<","<<
21         <<pt.y()<<","<<pt.z()<<")";
22 }
```

看到这段代码，很多人第一个疑问可能是：加上了封装，布局成本增加了多少？答案是class Point3d并没有增加成本。学过了C++对象模型，我们知道，Point3d类对象的内存中，只有三个数据成员。

上面的类声明中，三个数据成员直接内含在每一个Point3d对象中，而成员函数虽然在类中声明，却不出现在类对象（object）之中，这些函数(non-inline)属于类而不属于类对象，只会为类产生唯一的函数实例。

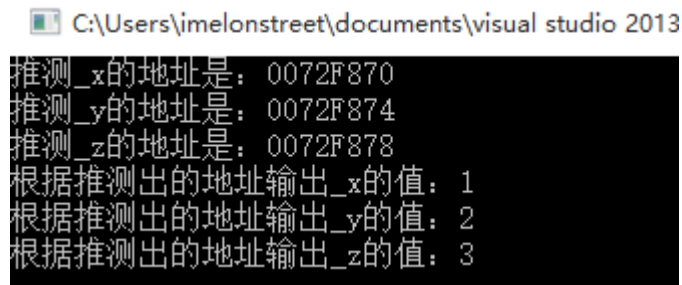
所以，**Point3d的封装并没有带来任何空间或执行期的效率影响**。而在下面这种情况下，C++的封装额外成本才会显示出来：

- 虚函数机制（virtual function），用以支持执行期绑定，实现多态。
- 虚基类（virtual base class），虚继承关系产生虚基类，用于在多重继承下保证基类在子类中拥有唯一实例。

不仅如此，Point3d类数据成员的内存布局与c语言的结构体Point3d**成员内存布局是相同的**。C++中处在同一个访问标识符（指public、private、protected）下的声明的数据成员，在内存中必定保证以其声明顺序出现。而处于不同访问标识符声明下的成员则无此规定。对于Point3类来说，它的三个数据成员都处于private下，在内存中一起声明顺序出现。我们可以做下实验：

```
1 void TestPoint3Member(const Point3d& p)
2 {
3
4     cout << "推测_x的地址是：" << (float *)(&p) << endl;
5     cout << "推测_y的地址是：" << (float *)(&p) + 1 << endl;
6     cout << "推测_z的地址是：" << (float *)(&p) + 2 << endl;
7
8     cout << "根据推测出的地址输出_x的值：" << *((float *)(&p)) << endl;
9     cout << "根据推测出的地址输出_y的值：" << *((float *)(&p)+1) << endl;
10    cout << "根据推测出的地址输出_z的值：" << *((float *)(&p)+2) << endl;
11
12 }
13 //测试代码
14 Point3d a(1,2,3);
15 TestPoint3Member(a);
```

运行结果：



C:\Users\jmelonstreet\documents\visual studio 2013

```
推测_x的地址是：0072F870
推测_y的地址是：0072F874
推测_z的地址是：0072F878
根据推测出的地址输出_x的值：1
根据推测出的地址输出_y的值：2
根据推测出的地址输出_z的值：3
```

从结果可以看到，x,y,z三个数据成员在内存中紧挨着。

总结一下：

不考虑虚函数与虚继承，当数据都在同一个访问标识符下，C++的类与C语言的结构体在对象大小和内存布局上是一致的，C++的封装并没有带来空间时间上的影响。

7.2.下面这个空类构成的继承层次中，每个类的大小是多少？

今有类如下：

```
1 class B{};
2 class B1:public virtual B{};
3 class B2:public virtual B{};
4 class D : public B1, public B2{};
5
6 int main()
7 {
```

```
8      B b;  
9      B1 b1;  
10     B2 b2;  
11     D d;  
12     cout << "sizeof(b)=" << sizeof(b)<<endl;  
13     cout << "sizeof(b1)=" << sizeof(b1) << endl;  
14     cout << "sizeof(b2)=" << sizeof(b2) << endl;  
15     cout << "sizeof(d)=" << sizeof(d) << endl;  
16     getchar();  
17 }
```

输出结果是：

```
sizeof(b)=1  
sizeof(b1)=4  
sizeof(b2)=4  
sizeof(d)=8
```

解析：

- 编译器为空类安插1字节的char，以使该类对象在内存得以配置一个地址。
- b1虚继承于b，编译器为其安插一个4字节的虚基类表指针（32为机器），此时b1已不为空，编译器不再为其安插1字节的char（优化）。
- b2同理。
- d含有来自b1与b2两个父类的两个虚基类表指针。大小为8字节。

完

转载请注明原出处：<http://www.cnblogs.com/QG-whz/p/4909359.html>