# C++ 对象的内存布局(上)

陈皓

## 前言

07 年 12 月，我写了一篇《C++虚函数表解析》的文章，引起了大家的兴趣。有很多朋友对我的文章留了言，有鼓励我的，有批评我的，还有很多问问题的。我在这里一并对大家的留言表示感谢。这也是我为什么再写一篇续言的原因。因为，在上一篇文章中，我用了的示例都是非常简单的，主要是为了说明一些机理上的问题，也是为了图一些表达上方便和简单。不想，这篇文章成为了打开 C++对象模型内存布局的一个引子，引发了大家对 C++对象的更深层次的讨论。当然，我之前的文章还有很多方面没有涉及，从我个人感觉下来，在谈论虚函数表里，至少有以下这些内容没有涉及：

1）有成员变量的情况。

2）有重复继承的情况。

3）有虚拟继承的情况。

4）有钻石型虚拟继承的情况。

这些都是我本篇文章需要向大家说明的东西。所以，这篇文章将会是《C++虚函数表解析》的一个续篇，也是一篇高级进阶的文章。我希望大家在读这篇文章之前对 C++有一定的基础和了解，并能先读我的上一篇文章。因为这篇文章的深度可能会比较深，而且会比较杂乱，我希望你在读本篇文章时不会有大脑思维紊乱导致大脑死机的情况。;-)

# 对象的影响因素

简而言之，我们一个类可能会有如下的影响因素：

1）成员变量

2）虚函数（产生虚函数表）

3）单一继承（只继承于一个类）

4）多重继承（继承多个类）

5）重复继承（继承的多个父类中其父类有相同的超类）

6）虚拟继承（使用 virtual 方式继承，为了保证继承后父类的内存布局只会存在一份）

上述的东西通常是 C++这门语言在语义方面对对象内部的影响因素，当然，还会有编译器的影响（比如优化），还有字节对齐的影响。在这里我们都不讨论，我们只讨论 C++语言上的影响。

本篇文章着重讨论下述几个情况下的 C++对象的内存布局情况。

1）**单一的一般继承**（带成员变量、虚函数、虚函数覆盖）

2）**单一的虚拟继承**（带成员变量、虚函数、虚函数覆盖）

3）**多重继承**（带成员变量、虚函数、虚函数覆盖）

4）**重复多重继承**（带成员变量、虚函数、虚函数覆盖）

5）**钻石型的虚拟多重继承**（带成员变量、虚函数、虚函数覆盖）

我们的目标就是，让事情越来越复杂。

# 知识复习

我们简单地复习一下，我们可以通过对象的地址来取得虚函数表的地址，如：

```
typedef void(*Fun)(void);

Base b;

Fun pFun = NULL;

cout << "虚函数表地址:" << (int*)(&b) << endl;

cout << "虚函数表 — 第一个函数地址:" << (int*)*(int*)(&b) << endl;

// Invoke the first virtual function

pFun = (Fun)*((int*)*(int*)(&b));

pFun();
```

我们同样可以用这种方式来取得整个对象实例的内存布局。因为这些东西在内存中都是连续分布的，我们只需要使用适当的地址偏移量，我们就可以获得整个内存对象的布局。

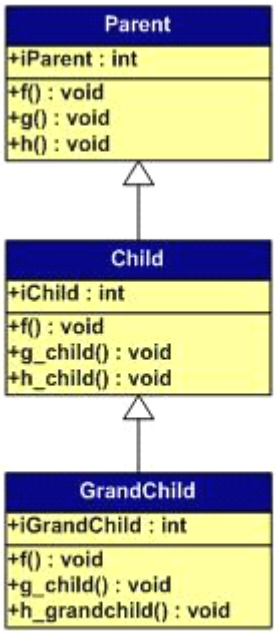
本篇文章中的例程或内存布局主要使用如下编译器和系统：

1）Windows XP 和 VC++ 2003

2）Cygwin 和 G++ 3.4.4

# 单一的一般继承

下面，我们假设有如下所示的一个继承关系：




单一继承

请注意，在这个继承关系中，父类，子类，孙子类都有自己的一个成员变量。而了类覆盖了父类的 f()方法，孙子类覆盖了子类的 g_child()及其超类的 f()。

我们的源程序如下所示：

```
class Parent {
public:
    int iparent;
    Parent ():iparent (10) {}
        virtual void f() { cout << " Parent::f()" << endl; }
        virtual void g() { cout << " Parent::g()" << endl; }
        virtual void h() { cout << " Parent::h()" << endl; }
```

```
};

class Child : public Parent {
public:
    int ichild;
    Child():ichild(100) {}
        virtual void f() { cout << "Child::f()" << endl; }
        virtual void g_child() { cout << "Child::g_child()" << endl; }
        virtual void h_child() { cout << "Child::h_child()" << endl; }
};

class GrandChild : public Child{
public:
    int igrandchild;
    GrandChild():igrandchild(1000) {}
        virtual void f() { cout << "GrandChild::f()" << endl; }
        virtual void g_child() { cout << "GrandChild::g_child()" << endl; }
        virtual void h_grandchild() { cout << "GrandChild::h_grandchild()" << endl; }
};
```

我们使用以下程序作为测试程序：（下面程序中，我使用了一个 int** pVtab 来作为遍历对象内存布局的指针，这样，我就可以方便地像使用数组一样来遍历所有的成员包括其虚函数表了，在后面的程序中，我也是用这样的方法的，请不必感到奇怪，）

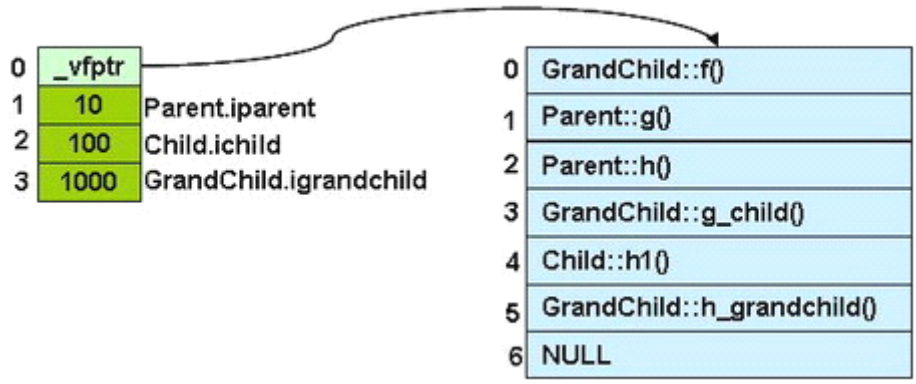
```
typedef void(*Fun)(void);
GrandChild gc;
int** pVtab = (int**)&gc;
cout << "[0] GrandChild::_vptr->" << endl;
for (int i=0; (Fun)pVtab[0][i]!=NULL; i++){
    pFun = (Fun)pVtab[0][i];
    cout << "    ["<<i<<"] ";
    pFun();
}
cout << "[1] Parent.iparent = " << (int)pVtab[1] << endl;
cout << "[2] Child.ichild = " << (int)pVtab[2] << endl;
cout << "[3] GrandChild.igrandchild = " << (int)pVtab[3] << endl;
```

其运行结果如下所示：（在 VC++ 2003 和 G++ 3.4.4 下）

```
[0] GrandChild::_vptr->
    [0] GrandChild::f()
    [1] Parent::g()
    [2] Parent::h()
    [3] GrandChild::g_child()
    [4] Child::h1()
    [5] GrandChild::h_grandchild()
[1] Parent.iparent = 10
[2] Child.ichild = 100
```

```
[3] GrandChild.igrandchild = 1000
```

使用图片表示如下：




可见以下几个方面：

1）虚函数表在最前面的位置。

2）成员变量根据其继承和声明顺序依次放在后面。

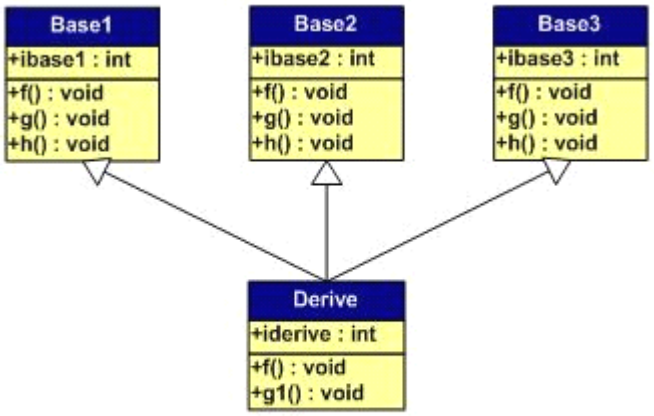
3）在单一的继承中，被 overwrite 的虚函数在虚函数表中得到了更新。

# 多重继承

下面，再让我们来看看多重继承中的情况，假设有下面这样一个类的继承关系。

注意：子类只 overwrite 了父类的 f()函数，而还有一个是自己的函数（我们这样做

的目的是为了用 g1()作为一个标记来标明子类的虚函数表）。而且每个类中都有一个自己的成员变量：




多重继承

我们的类继承的源代码如下所示：父类的成员初始为 10，20，30，子类的为 100

```
class Base1 {
public:
    int ibase1;
    Base1():ibase1(10) {}
        virtual void f() { cout << "Base1::f()" << endl; }
        virtual void g() { cout << "Base1::g()" << endl; }
        virtual void h() { cout << "Base1::h()" << endl; }

};

class Base2 {
```

```
public:
    int ibase2;
    Base2():ibase2(20) {}
        virtual void f() { cout << "Base2::f()" << endl; }
        virtual void g() { cout << "Base2::g()" << endl; }
        virtual void h() { cout << "Base2::h()" << endl; }
};

class Base3 {
public:
    int ibase3;
    Base3():ibase3(30) {}
        virtual void f() { cout << "Base3::f()" << endl; }
        virtual void g() { cout << "Base3::g()" << endl; }
        virtual void h() { cout << "Base3::h()" << endl; }
};


class Derive : public Base1, public Base2, public Base3 {
public:
    int iderive;
    Derive():iderive(100) {}
        virtual void f() { cout << "Derive::f()" << endl; }
        virtual void g1() { cout << "Derive::g1()" << endl; }
```

```
};
```

我们通过下面的程序来查看子类实例的内存布局：下面程序中，注意我使用了一个 s 变量，其中用到了 sizof(Base)来找下一个类的偏移量。（因为我声明的是 int 成员，所以是 4 个字节，所以没有对齐问题。关于内存的对齐问题，大家可以自行试验，我在这里就不多说了）

```
typedef void(*Fun)(void);


Derive d;

int** pVtab = (int**)&d;

cout << "[0] Base1::_vptr->" << endl;
pFun = (Fun)pVtab[0][0];
cout << "       [0] ";
pFun();

pFun = (Fun)pVtab[0][1];
cout << "       [1] ";pFun();

pFun = (Fun)pVtab[0][2];
cout << "       [2] ";pFun();
```

```
pFun = (Fun)pVtab[0][3];
cout << "       [3] ";          pFun();

pFun = (Fun)pVtab[0][4];
cout << "       [4] ";          cout<<pFun<<endl;

cout << "[1] Base1.ibase1 = " << (int)pVtab[1] << endl;

int s = sizeof(Base1)/4;

cout << "[" << s << "] Base2::_vptr->"<<endl;
pFun = (Fun)pVtab[s][0];
cout << "       [0] "; pFun();

Fun = (Fun)pVtab[s][1];
cout << "       [1] "; pFun();

pFun = (Fun)pVtab[s][2];
cout << "       [2] ";          pFun();

pFun = (Fun)pVtab[s][3];
out << "       [3] ";
```

```
cout<<pFun<<endl;

cout << "["<< s+1 <<"] Base2.ibase2 = " << (int)pVtab[s+1] << endl;

s = s + sizeof(Base2)/4;

cout << "[" << s << "] Base3::_vptr->"<<endl;
pFun = (Fun)pVtab[s][0];
cout << "      [0] "; pFun();

pFun = (Fun)pVtab[s][1];
cout << "      [1] ";       pFun();

pFun = (Fun)pVtab[s][2];
cout << "      [2] ";       pFun();

pFun = (Fun)pVtab[s][3];
 cout << "      [3] ";
cout<<pFun<<endl;

s++;
cout << "["<< s <<"] Base3.ibase3 = " << (int)pVtab[s] << endl;
s++;
```

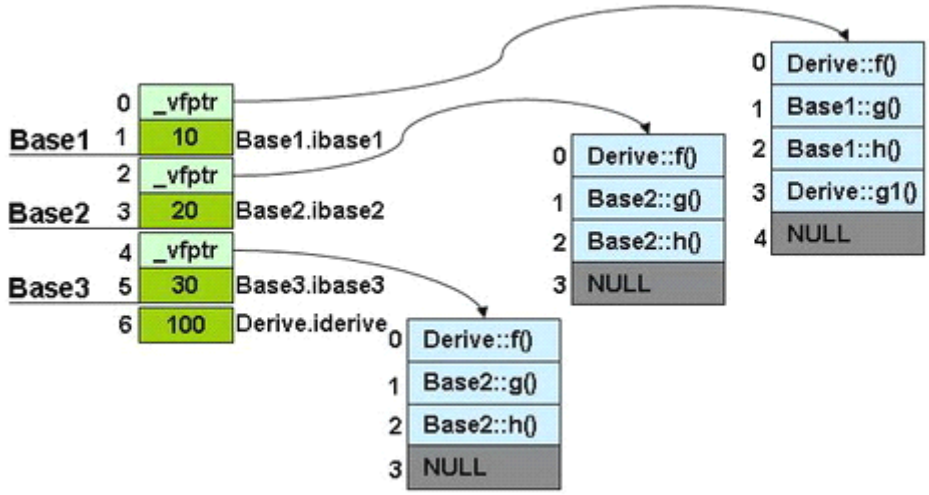
```
cout << "["<< s <<"] Derive.iderive = " << (int)pVtab[s] << endl;
```

其运行结果如下所示：（在 VC++ 2003 和 G++ 3.4.4 下）

```
[0] Base1::_vptr->
    [0] Derive::f()
    [1] Base1::g()
    [2] Base1::h()
    [3] Driver::g1()
[4] 00000000      ← 注意：在 GCC 下，这里是 1
[1] Base1.ibase1 = 10
[2] Base2::_vptr->
    [0] Derive::f()
    [1] Base2::g()
    [2] Base2::h()
[3] 00000000      ← 注意：在 GCC 下，这里是 1
[3] Base2.ibase2 = 20
[4] Base3::_vptr->
    [0] Derive::f()
    [1] Base3::g()
    [2] Base3::h()
    [3] 00000000
[5] Base3.ibase3 = 30
```

```
[6] Derive.iderive = 100
```

使用图片表示是下面这个样子：




我们可以看到：

1） 每个父类都有自己的虚表。

2） 子类的成员函数被放到了第一个父类的表中。

3） 内存布局中，其父类布局依次按声明顺序排列。

4） 每个父类的虚表中的 f()函数都被 overwrite 成了子类的 f()。这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。

# C++ 对象的内存布局(下)

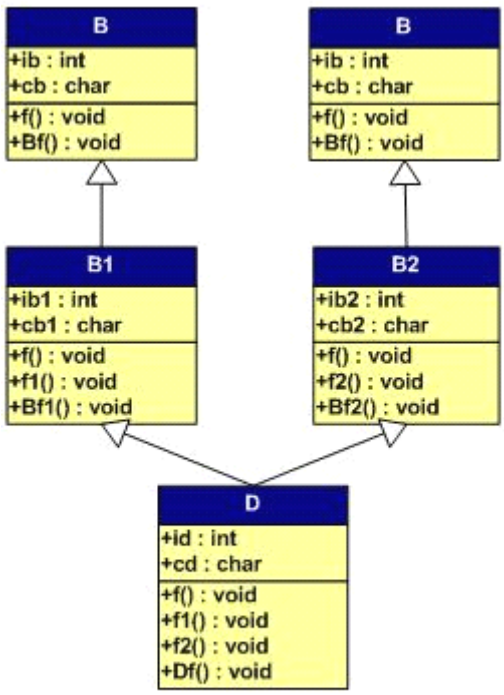
陈皓

http://blog.csdn.net/haoel

## 重复继承

下面我们再来看看，发生重复继承的情况。所谓重复继承，也就是某个基类被间接地重复继承了多次。

下图是一个继承图，我们重载了父类的 f()函数。




钻石型--重复继承

其类继承的源代码如下所示。其中，每个类都有两个变量，一个是整形（4 字节），一个是字符（1 字节），而且还有自己的虚函数，自己 overwrite 父类的虚函数。如子类 D 中，f()覆盖了超类的函数， f1() 和 f2() 覆盖了其父类的虚函数，Df()为自己的虚函数。

```
class B
{
    public:
        int ib;
        char cb;
    public:
        B():ib(0),cb('B') {}

        virtual void f() { cout << "B::f()" << endl;}
        virtual void Bf() { cout << "B::Bf()" << endl;}
};
class B1 :  public B
{
    public:
        int ib1;
        char cb1;
    public:
        B1():ib1(11),cb1('1') {}
```

```
        virtual void f() { cout << "B1::f()" << endl;}
        virtual void f1() { cout << "B1::f1()" << endl;}
        virtual void Bf1() { cout << "B1::Bf1()" << endl;}

};
class B2:  public B
{
    public:
        int ib2;
        char cb2;
    public:
        B2():ib2(12),cb2('2') {}

        virtual void f() { cout << "B2::f()" << endl;}
        virtual void f2() { cout << "B2::f2()" << endl;}
        virtual void Bf2() { cout << "B2::Bf2()" << endl;}

};

class D : public B1, public B2
{
    public:
        int id;
        char cd;
```

```
public:
    D():id(100),cd('D') {}

    virtual void f() { cout << "D::f()" << endl;}
    virtual void f1() { cout << "D::f1()" << endl;}
    virtual void f2() { cout << "D::f2()" << endl;}
    virtual void Df() { cout << "D::Df()" << endl;}

};
```

我们用来存取子类内存布局的代码如下所示：（在 VC++ 2003 和 G++ 3.4.4 下）

```
typedef void(*Fun)(void);
int** pVtab = NULL;
Fun pFun = NULL;

D d;
pVtab = (int**)&d;
cout << "[0] D::B1::_vptr->" << endl;
pFun = (Fun)pVtab[0][0];
cout << "      [0] ";     pFun();
pFun = (Fun)pVtab[0][1];
cout << "      [1] ";     pFun();
pFun = (Fun)pVtab[0][2];
cout << "      [2] ";     pFun();
pFun = (Fun)pVtab[0][3];
```

```
cout << "        [3] ";      pFun();
pFun = (Fun)pVtab[0][4];
cout << "        [4] ";      pFun();
pFun = (Fun)pVtab[0][5];
cout << "        [5] 0x" << pFun << endl;

cout << "[1] B::ib = " << (int)pVtab[1] << endl;
cout << "[2] B::cb = " << (char)pVtab[2] << endl;
cout << "[3] B1::ib1 = " << (int)pVtab[3] << endl;
cout << "[4] B1::cb1 = " << (char)pVtab[4] << endl;

cout << "[5] D::B2::_vptr->" << endl;
pFun = (Fun)pVtab[5][0];
cout << "        [0] ";      pFun();
pFun = (Fun)pVtab[5][1];
cout << "        [1] ";      pFun();
pFun = (Fun)pVtab[5][2];
cout << "        [2] ";      pFun();
pFun = (Fun)pVtab[5][3];
cout << "        [3] ";      pFun();
pFun = (Fun)pVtab[5][4];
cout << "        [4] 0x" << pFun << endl;

cout << "[6] B::ib = " << (int)pVtab[6] << endl;
```

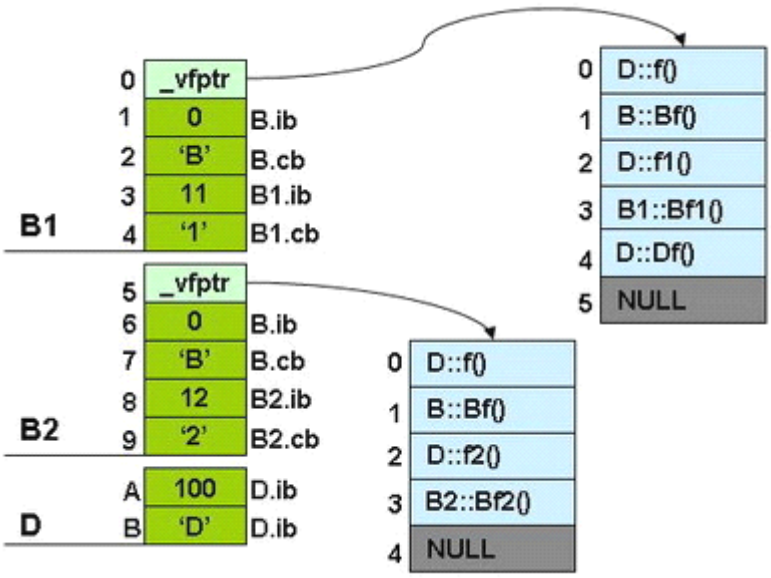
```
cout << "[7] B::cb = " << (char)pVtab[7] << endl;

cout << "[8] B2::ib2 = " << (int)pVtab[8] << endl;

cout << "[9] B2::cb2 = " << (char)pVtab[9] << endl;


cout << "[10] D::id = " << (int)pVtab[10] << endl;

cout << "[11] D::cd = " << (char)pVtab[11] << endl;
```

程序运行结果如下：

| GCC 3.4.4 | VC++ 2003 |
|---|---|
| [0] D::B1::_vptr-> | [0] D::B1::_vptr-> |
| [0] D::f() | [0] D::f() |
| [1] B::Bf() | [1] B::Bf() |
| [2] D::f1() | [2] D::f1() |
| [3] B1::Bf1() | [3] B1::Bf1() |
| [4] D::f2() | [4] D::Df() |
| [5] 0x1 | [5] 0x00000000 |
| [1] B::ib = 0 | [1] B::ib = 0 |
| [2] B::cb = B | [2] B::cb = B |
| [3] B1::ib1 = 11 | [3] B1::ib1 = 11 |
| [4] B1::cb1 = 1 | [4] B1::cb1 = 1 |
| [5] D::B2::_vptr-> | [5] D::B2::_vptr-> |
| [0] D::f() | [0] D::f() |

| | |
|---|---|
| [1] B::Bf() | [1] B::Bf() |
| [2] D::f2() | [2] D::f2() |
| [3] B2::Bf2() | [3] B2::Bf2() |
| [4] 0x0 | [4] 0x00000000 |
| [6] B::ib = 0 | [6] B::ib = 0 |
| [7] B::cb = B | [7] B::cb = B |
| [8] B2::ib2 = 12 | [8] B2::ib2 = 12 |
| [9] B2::cb2 = 2 | [9] B2::cb2 = 2 |
| [10] D::id = 100 | [10] D::id = 100 |
| [11] D::cd = D | [11] D::cd = D |

下面是对于子类实例中的虚函数表的图：




我们可以看见，最顶端的父类 B 其成员变量存在于 B1 和 B2 中，并被 D 给继承下去了。而在 D 中，其有 B1 和 B2 的实例，于是 B 的成员在 D 的实例中存在两份，

一份是 B1 继承而来的，另一份是 B2 继承而来的。所以，如果我们使用以下语句，则会产生二义性编译错误：

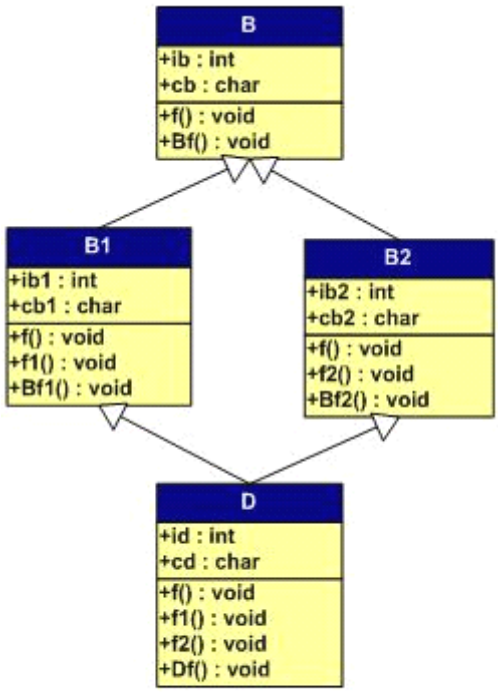
```
D d;
d.ib = 0;          //二义性错误
d.B1::ib = 1;      //正确
d.B2::ib = 2;      //正确
```

注意，上面例程中的最后两条语句存取的是两个变量。虽然我们消除了二义性的编译错误，但 B 类在 D 中还是有两个实例，这种继承造成了数据的重复，我们叫这种继承为重复继承。重复的基类数据成员可能并不是我们想要的。所以，C++引入了虚基类的概念。

## 钻石型多重虚拟继承

虚拟继承的出现就是为了解决重复继承中多个间接父类的问题的。钻石型的结构是其最经典的结构。也是我们在这里要讨论的结构：

上述的“重复继承”只需要把 B1 和 B2 继承 B 的语法中加上 virtual 关键，就成了虚拟继承，其继承图如下所示：




钻石型虚拟多重继承

上图和前面的“重复继承”中的类的内部数据和接口都是完全一样的，只是我们采用了虚拟继承：其省略后的源码如下所示：

```
class B {……};
class B1 : virtual public B{……};
class B2: virtual public B{……};
class D : public B1, public B2{ …… };
```

在查看 D 之前，我们先看一看**单一虚拟继承**的情况。下面是一段在 VC++2003 下的测试程序：（因为 VC++和 GCC 的内存而局上有一些细节上的不同，所以这里只给出 VC++的程序，GCC 下的程序大家可以根据我给出的程序自己仿照着写一个去试一试）：

```
int** pVtab = NULL;
Fun pFun = NULL;

B1 bb1;

pVtab = (int**)&bb1;
cout << "[0] B1::_vptr->" << endl;
pFun = (Fun)pVtab[0][0];
cout << "        [0] ";
pFun(); //B1::f1();
cout << "        [1] ";
pFun = (Fun)pVtab[0][1];
pFun(); //B1::bf1();
cout << "        [2] ";
cout << pVtab[0][2] << endl;

cout << "[1] = 0x";
cout << (int*)*((int*)(&bb1)+1) <<endl; //B1::ib1
cout << "[2] B1::ib1 = ";
cout << (int)*((int*)(&bb1)+2) <<endl; //B1::ib1
cout << "[3] B1::cb1 = ";
cout << (char)*((int*)(&bb1)+3) << endl; //B1::cb1

cout << "[4] = 0x";
```

```
cout << (int*)*((int*)(&bb1)+4) << endl; //NULL


cout << "[5] B::_vptr->" << endl;
pFun = (Fun)pVtab[5][0];
cout << "       [0] ";
pFun(); //B1::f();
pFun = (Fun)pVtab[5][1];
cout << "       [1] ";
pFun(); //B::Bf();
cout << "       [2] ";
cout << "0x" << (Fun)pVtab[5][2] << endl;


cout << "[6] B::ib = ";
cout << (int)*((int*)(&bb1)+6) <<endl; //B::ib
cout << "[7] B::cb = ";
```

其运行结果如下（我结出了 GCC 的和 VC++2003 的对比）：

| GCC 3.4.4 | VC++ 2003 |
|---|---|
| [0] B1::_vptr -> | [0] B1::_vptr-> |
| [0] : B1::f() | [0] B1::f1() |
| [1] : B1::f1() | [1] B1::Bf1() |
| [2] : B1::Bf1() | [2] 0 |
| [3] : 0 | [1] = 0x00454310 ←该地址取值后是-4 |

| | |
|---|---|
| [1] B1::ib1 : 11 | [2] B1::ib1 = 11 |
| [2] B1::cb1 : 1 | [3] B1::cb1 = 1 |
| [3] B::_vptr -> | [4] = 0x00000000 |
| [0] : B1::f() | [5] B::_vptr-> |
| [1] : B::Bf() | [0] B1::f() |
| [2] : 0 | [1] B::Bf() |
| [4] B::ib : 0 | [2] 0x00000000 |
| [5] B::cb : B | [6] B::ib = 0 |
| [6] NULL : 0 | [7] B::cb = B |

这里，大家可以自己对比一下。关于细节上，我会在后面一并再说。

下面的测试程序是看子类 D 的内存布局，同样是 VC++ 2003 的（因为 VC++和 GC C 的内存布局上有一些细节上的不同，而 VC++的相对要清楚很多，所以这里只给出 VC++的程序，GCC 下的程序大家可以根据我给出的程序自己仿照着写一个去试一试）：

```
D d;

pVtab = (int**)&d;
cout << "[0] D::B1::_vptr->" << endl;
pFun = (Fun)pVtab[0][0];
cout << "      [0] ";      pFun(); //D::f1();
```

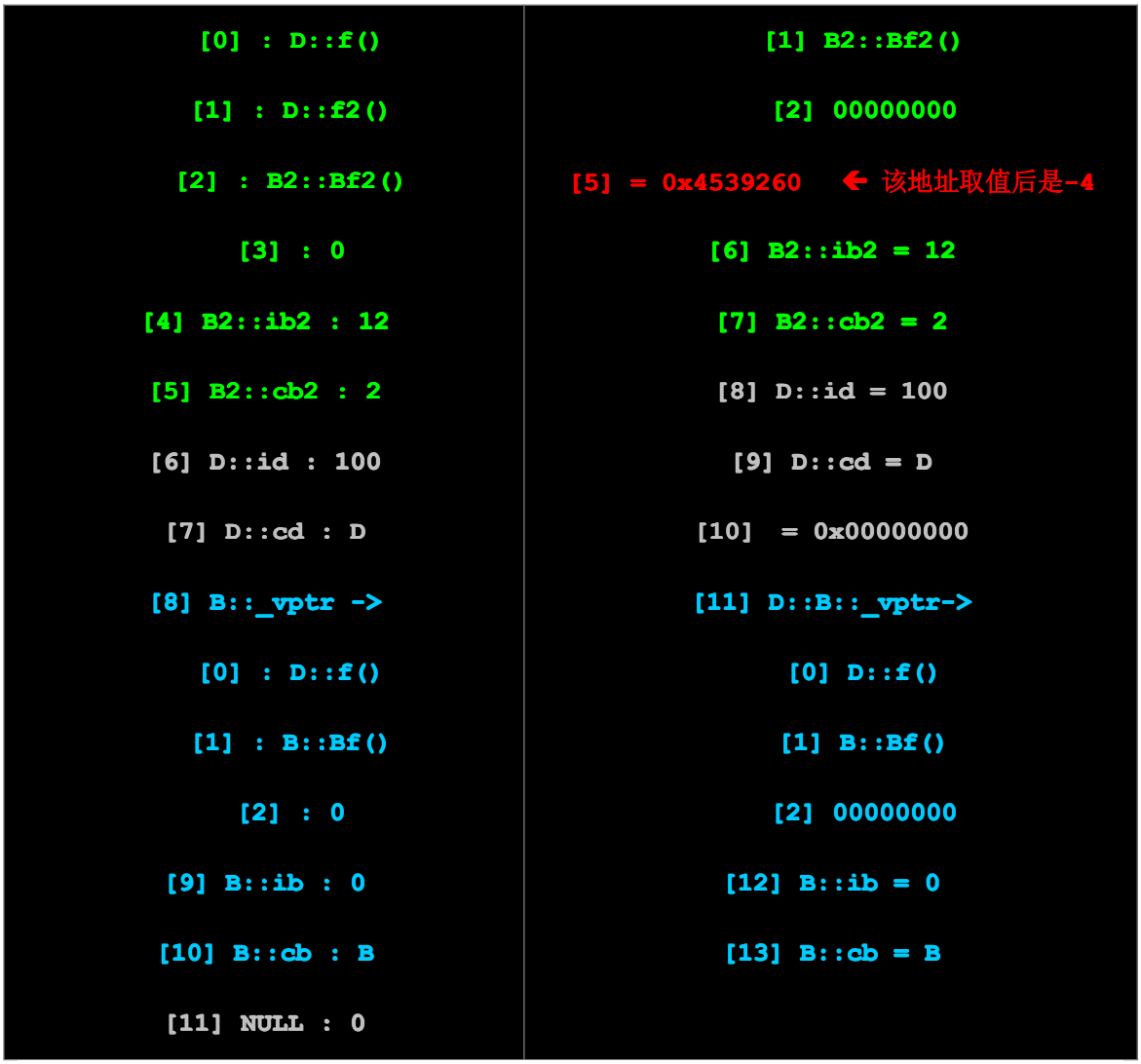
```
pFun = (Fun)pVtab[0][1];
cout << "      [1] ";      pFun(); //B1::Bf1();
pFun = (Fun)pVtab[0][2];
cout << "      [2] ";      pFun(); //D::Df();
pFun = (Fun)pVtab[0][3];
cout << "      [3] ";
cout << pFun << endl;

//cout << pVtab[4][2] << endl;
cout << "[1] = 0x";
cout <<  (int*)((&dd)+1) <<endl; //????

cout << "[2] B1::ib1 = ";
cout << *((int*)(&dd)+2) <<endl; //B1::ib1
cout << "[3] B1::cb1 = ";
cout << (char)*((int*)(&dd)+3) << endl; //B1::cb1

//----------------------
cout << "[4] D::B2::_vptr->" << endl;
pFun = (Fun)pVtab[4][0];
cout << "      [0] ";      pFun(); //D::f2();
pFun = (Fun)pVtab[4][1];
cout << "      [1] ";      pFun(); //B2::Bf2();
pFun = (Fun)pVtab[4][2];
```

```
cout << "        [2] ";
cout << pFun << endl;

cout << "[5] = 0x";
cout << *((int*)(&dd)+5) << endl; // ???

cout << "[6] B2::ib2 = ";
cout << (int)*((int*)(&dd)+6) <<endl; //B2::ib2
cout << "[7] B2::cb2 = ";
cout << (char)*((int*)(&dd)+7) << endl; //B2::cb2

cout << "[8] D::id = ";
cout << *((int*)(&dd)+8) << endl; //D::id
cout << "[9] D::cd = ";
cout << (char)*((int*)(&dd)+9) << endl;//D::cd

cout << "[10]   = 0x";
cout << (int*)*((int*)(&dd)+10) << endl;
//----------------------
cout << "[11] D::B::_vptr->" << endl;
pFun = (Fun)pVtab[11][0];
cout << "        [0] ";      pFun(); //D::f();
pFun = (Fun)pVtab[11][1];
cout << "        [1] ";      pFun(); //B::Bf();
```

```
pFun = (Fun)pVtab[11][2];
cout << "       [2] ";
cout << pFun << endl;

cout << "[12] B::ib = ";
cout << *((int*)(&dd)+12) << endl; //B::ib
cout << "[13] B::cb = ";
cout << (char)*((int*)(&dd)+13) <<endl;//B::cb
```

下面给出运行后的结果（分 VC++和 GCC 两部份）

| GCC 3.4.4 | VC++ 2003 |
|---|---|
| [0] B1::_vptr -> | [0] D::B1::_vptr-> |
| [0] : D::f() | [0] D::f1() |
| [1] : D::f1() | [1] B1::Bf1() |
| [2] : B1::Bf1() | [2] D::Df() |
| [3] : D::f2() | [3] 00000000 |
| [4] : D::Df() | [1] = 0x0013FDC4 ← 该地址取值后是-4 |
| [5] : 1 | [2] B1::ib1 = 11 |
| [1] B1::ib1 : 11 | [3] B1::cb1 = 1 |
| [2] B1::cb1 : 1 | [4] D::B2::_vptr-> |
| [3] B2::_vptr -> | [0] D::f2() |




关于虚拟继承的运行结果我就不画图了（前面的作图已经让我产生了很严重的厌倦感，所以就偷个懒了，大家见谅了）

在上面的输出结果中，我用不同的颜色做了一些标明。我们可以看到如下的几点：

1）无论是 GCC 还是 VC++，除了一些细节上的不同，其大体上的对象布局是一样的。也就是说，先是 B1（黄色），然后是 B2（绿色），接着是 D（灰

色），而 B 这个超类（青蓝色）的实例都放在最后的位置。

2）关于虚函数表，尤其是第一个虚表，GCC 和 VC++有很重大的不一样。但仔细看下来，还是 VC++的虚表比较清晰和有逻辑性。

3）VC++和 GCC 都把 B 这个超类放到了最后，而 VC++有一个 NULL 分隔符把 B 和 B1 和 B2 的布局分开。GCC 则没有。

4）VC++中的内存布局有两个地址我有些不是很明白，在其中我用红色标出了。取其内容是-4。接道理来说，这个指针应该是指向 B 类实例的内存地址（这个做法就是为了保证重复的父类只有一个实例的技术）。但取值后却不是。这点我目前还并不太清楚，还向大家请教。

5）GCC 的内存布局中在 B1 和 B2 中则没有指向 B 的指针。这点可以理解，编译器可以通过计算 B1 和 B2 的 size 而得出 B 的偏移量。

# 结束语

C++这门语言是一门比较复杂的语言，对于程序员来说，我们似乎永远摸不清楚这门语言背着我们在干了什么。需要熟悉这门语言，我们就必需要了解 C++里面的那些东西，需要我们去了解他后面的内存对象。这样我们才能真正的了解 C++，从而能够更好的使用 C++这门最难的编程语言。

在文章束之前还是介绍一下自己吧。我从事软件研发有十个年头了，目前是软件开发技术主管，技术方面，主攻 Unix/C/C++，比较喜欢网络上的技术，比如分布式计算，网格计算，P2P，Ajax 等一切和互联网相关的东西。管理方面比较擅长于团队建设，技术趋势分析，项目管理。欢迎大家和我交流，我的 MSN 和 Email 是：haoel@hotmail.com