转自：<http://blog.csdn.net/pathuang68/archive/2009/04/23/4102003.aspx>

**内存布局之一**

**内容概要：**

**满足下面2个条件时，**

**1. 父类有虚函数，子类无虚函数(即无虚函数重写或无虚函数覆盖)**

**2. 非虚继承**

**类对象之内存布局**

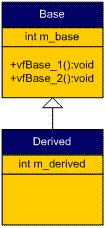
前述相关内容参考：

1. <http://blog.csdn.net/pathuang68/archive/2009/04/20/4096088.aspx>

2. <http://blog.csdn.net/pathuang68/archive/2009/04/21/4096429.aspx>

3. <http://blog.csdn.net/pathuang68/archive/2009/04/21/4096521.aspx>

Base类中有两个虚函数vfBase\_1()、vfBase\_2()和一个整形成员变量m\_base， Derived类中有一个整形成员变量m\_derived，二者的关系如下：



代码如下：

#include <iostream>

using namespace std;

class Base

{

public:

         int m\_base;

         inline virtual void vfBase\_1()

         {

                   cout << "This is in Base::vfBase\_1()" << endl;

         }

         inline virtual void vfBase\_2()

         {

                   cout << "This is in Base::vfBase\_2()" << endl;

         }

};

class Derived : public Base

{

public:

         int m\_derived;

};

typedef void (\*VFun)(void);

**//** **改为template形式，因为不能确定传进来的参数是Base类型的指针还是Derived类型的指针**

template<typename T>

VFun virtualFunctionPointer(T\* b, int i)

{

     return (VFun)(\*((int\*)(\*(int\*)b) + i));

}

int main(void)

{

         Derived d;

         cout << "The size of Base object = \t" << sizeof(Derived) << endl;

         cout << endl;

         int i = 0;

         while(virtualFunctionPointer(&d, i))

         {

                   VFun pVF = virtualFunctionPointer(&d, i++);

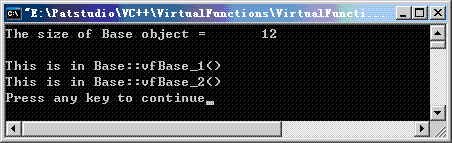
                   pVF();

         }

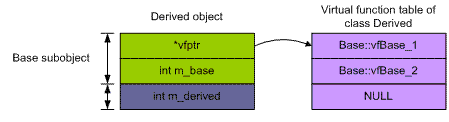
         return 0;

}

运行结果如下：



Derived对象的memory layout图解如下：



**内存布局之二**

**内容概要：**

**满足下面2个条件时，**

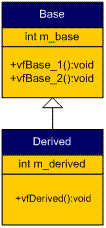
**1. 父类有虚函数，子类也有虚函数，但子类并没有重写或覆盖父类的虚函数**

**2. 非虚继承**

**类对象之内存布局**

续前篇：<http://blog.csdn.net/pathuang68/archive/2009/04/23/4101970.aspx>

如果在Derived类中增加一个下面的虚函数，会怎么样呢？Base类和Derived类之间的关系如下：



新加入的虚函数定义如下：

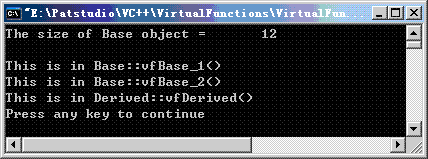
inline virtual void vfDerived()

{

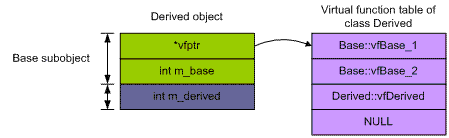
      cout << "This is in Derived::vfDerived()" << endl;

}

运行结果如下：



Derived对象的memory layout图解如下：



我们发现，Derived类本身并没有因为增加了一个虚函数，而增加一个vptr，这是因为编译器将Derived类中定义的虚函数加入到了其基类的虚函数表中，Derived因此共用了基类Base的vptr，就象是它自己的一样(其实当然也是Derived类自己的，因此Base subobject都被Derived包含了)。为什么说这个vptr是Base的呢？因为基类在派生类中必须保证其所谓的“原始的完整性”。

**内存布局之三**

**内容概要：**

**满足下面2个条件时，**

**1. 父类无虚函数，子类有虚函数**

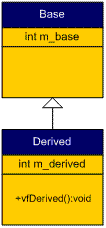
**2. 非虚继承**

**类对象之内存布局**

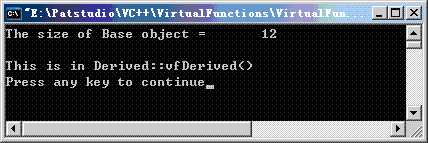
前篇：<http://blog.csdn.net/pathuang68/archive/2009/04/23/4101977.aspx>

如果将Base中的两个虚函数删除，情况有会怎么样呢？

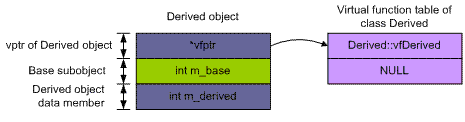
将Base中的两个虚函数删除，其他保持不变。Base类和Derived类两者之间的关系如下：



运行结果如下：



Derived对象的memory layout图解如下：



这次由于Base类中不存在虚函数，所以在Derived类对象中的Base subobject就不会有vptr了，但Derived类中有虚函数，所以它的对象中就有vptr了。

**内存布局之四**

**内容概要：**

**满足下面2个条件时，**

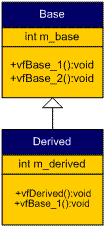
**1. 父类有虚函数，子类也有虚函数，且子类的虚函数重写或覆盖了父类的虚函数**

**2. 非虚继承**

**类对象之内存布局**

前篇：<http://blog.csdn.net/pathuang68/archive/2009/04/23/4101979.aspx>

在前面的例子中，恢复原来的两个虚函数vfBase\_1()和vfBase\_2()，同时在Derived类中重写基类的虚函数vfBase\_1()，Base类和Derived类之间的关系如下图：



整个代码如下：

#include <iostream>

using namespace std;

class Base

{

public:

         int m\_base;

         inline virtual void vfBase\_1()

         {

                   cout << "This is in Base::vfBase\_1()" << endl;

         }

         inline virtual void vfBase\_2()

         {

                   cout << "This is in Base::vfBase\_2()" << endl;

         }

};

class Derived : public Base

{

public:

         int m\_derived;

         inline virtual void vfDerived()

         {

                   cout << "This is in Derived::vfDerived()" << endl;

         }

         inline void vfBase\_1()

         {

                   cout << "This is in Derived::vfBase\_1()" << endl;

         }

};

typedef void (\*VFun)(void);

template<typename T>

VFun virtualFunctionPointer(T\* b, int i)

{

         return (VFun)(\*((int\*)(\*(int\*)b) + i));

}

int main(void)

{

         Derived d;

         cout << "The size of Base object = \t" << sizeof(Derived) << endl;

         cout << endl;

         int i = 0;

         while(virtualFunctionPointer(&d, i))

         {

                   VFun pVF = virtualFunctionPointer(&d, i++);

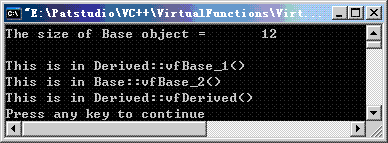
                   pVF();

         }

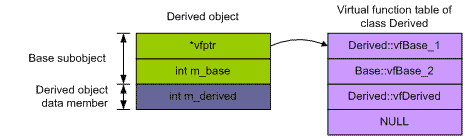
         return 0;

}

运行结果如下：



Derived 对象的memory layout图解如下：



因为Derived类中重写了虚函数vfBase\_1()，所以Derived::vfBase\_1()就取代了Base::vfBase\_1()的位置，位于虚函数表的开始处。而Base::vfBase\_1()就不会再在Derived的虚函数表中出现了。

**内存布局之五**

**内容概要：**

**满足下面3个条件时，**

**1. 父类有虚函数，子类也有虚函数，且子类的虚函数重写或覆盖了父类的虚函数**

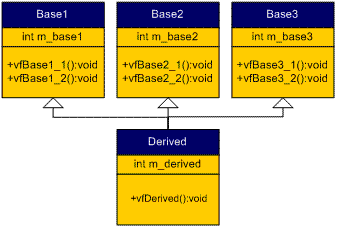
**2. 非虚继承**

**3. 多重继承**

**类对象之内存布局**

多重继承，派生类不重写基类中的虚函数。

假定各类之间的关系如下图：



代码如下：

#include <iostream>

using namespace std;

class Base1

{

public:

         int m\_base1;

         inline virtual void vfBase1\_1()

         {

                   cout << "This is in Base1::vfBase1\_1()" << endl;

         }

         inline virtual void vfBase1\_2()

         {

                   cout << "This is in Base1::vfBase1\_2()" << endl;

         }

};

class Base2

{

public:

         int m\_base2;

         inline virtual void vfBase2\_1()

         {

                   cout << "This is in Base2::vfBase2\_1()" << endl;

         }

         inline virtual void vfBase2\_2()

         {

                   cout << "This is in Base2::vfBase2\_2()" << endl;

         }

};

class Base3

{

public:

         int m\_Base3;

         inline virtual void vfBase3\_1()

         {

                   cout << "This is in Base3::vfBase3\_1()" << endl;

         }

         inline virtual void vfBase3\_2()

         {

                   cout << "This is in Base3::vfBase3\_2()" << endl;

         }

};

class Derived : public Base1, public Base2, public Base3

{

public:

         int m\_derived;

         inline virtual void fd()

         {

                   cout << "This is in Derived::fd()" << endl;

         }

};

typedef void (\*VFun)(void);

template<typename T>

VFun virtualFunctionPointer(T\* b, int i)

{

         return (VFun)(\*((int\*)(\*(int\*)b) + i));

}

int main(void)

{

         Derived d;

         cout << "The size of Derived object = \t" << sizeof(Derived) << endl;

         cout << endl;

         cout << "1st virtual function table: " << endl;

         int i = 0;

         while(virtualFunctionPointer(&d, i))

         {

                   VFun pVF = virtualFunctionPointer(&d, i++);

                   pVF();

         }

         cout << endl;

         cout << "2nd virtual function table: " << endl;

         i = 0;

         int\* tmp = ((int\*)&d) + 2;

         while(virtualFunctionPointer(tmp, i))

         {

                   VFun pVF = virtualFunctionPointer(tmp, i++);

                   pVF();

         }

         cout << endl;

         cout << "3rd virtual function table: " << endl;

         i = 0;

         tmp = ((int\*)&d) + 4;

         while(virtualFunctionPointer(tmp, i))

         {

                   VFun pVF = virtualFunctionPointer(tmp, i++);

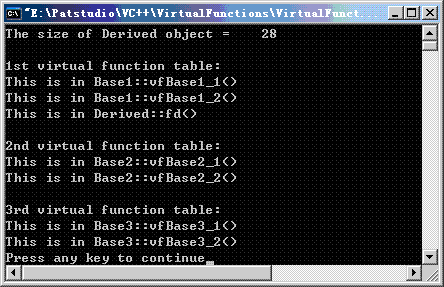
                   pVF();

         }

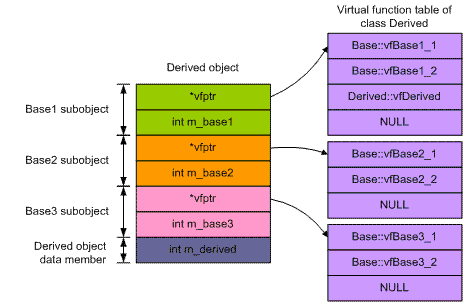
         return 0;

}

运行结果如下：



Derived对象之memory layout如下：



由上面的分析可知：

其一：有三个虚函数表

其二：在Derived类中定义的虚函数Derived::vfDerived()附加在一个虚函数表的最后

内存布局之六

如果在[对象内存布局 (5)](http://blog.csdn.net/pathuang68/archive/2009/04/23/4101999.aspx)的代码中，将Base1中的两个虚函数声明删除，同时将main函数中的下面代码注释掉(因为现在只有两张虚函数表了)：

        cout << "3rd virtual function table: " << endl;

         i = 0;

         tmp = ((int\*)&d) + 4;

         while(virtualFunctionPointer(tmp, i))

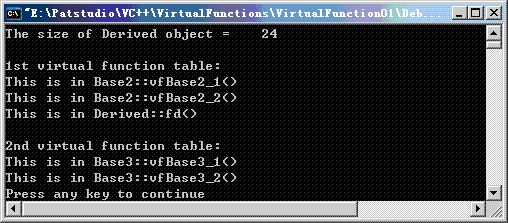
         {

                   VFun pVF = virtualFunctionPointer(tmp, i++);

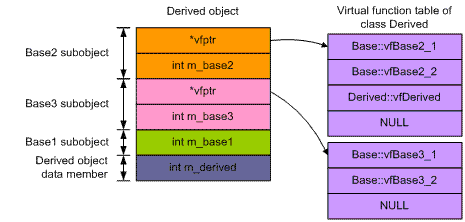
                   pVF();

         }

其他代码不变，运行结果如下：



Derived对象的memory layout图解如下：



由上面分析可知：

其一，Base1 subobject排列在Base2 subobject和Base3 subobject之后，尽管它在Derived类的继承列表中排列在首位，因为它已经没有虚函数表了，但仍然排列在Derived类的成员变量m\_derived之前。

其二，在Derived类中定义的虚函数Derived::vfDerived()附加在一个虚函数表的最后，这时第一张虚函数表是类Base2的。

**内存布局之七**

在[对象内存布局 (5)](http://blog.csdn.net/pathuang68/archive/2009/04/23/4101999.aspx)的代码中，在Derived类中覆盖Base1中声明的vfBase1\_1()，其他代码不变。修改后的Derived的定义如下：

class Derived : public Base1, public Base2, public Base3

{

public:

int m\_derived;

     inline virtual void fd()

     {

              cout << "This is in Derived::fd()" << endl;

     }

     inline void vfBase1\_1()

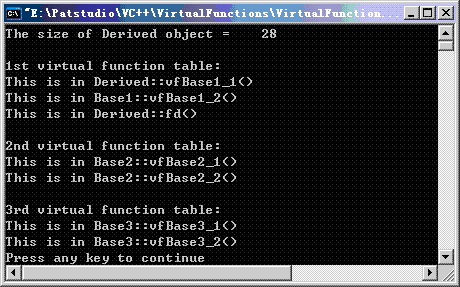
     {

              cout << "This is in Derived::vfBase1\_1()" << endl;

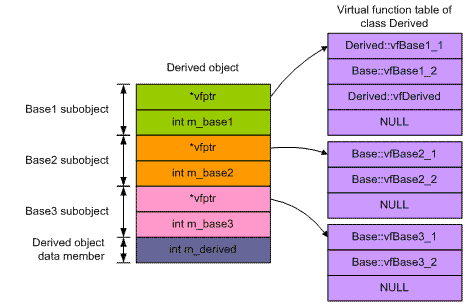
     }

};

运行结果如下：



Derived对象值memory layout图解如下：



我们可以看到，在Derived中overriden的虚函数Derived::vfBase1\_1排在第一个虚函数表的第一位。虚函数Base::vfBase1\_1()由于已经被overridden，所以在Derived对象的虚函数表中不再出现。

**内存布局之八**

在[内存对象布局 (5)](http://blog.csdn.net/pathuang68/archive/2009/04/23/4101999.aspx)的代码中，在Derived类中将三个基类中的虚函数分别覆盖一个，即分别覆盖Base1中声明的vfBase1\_1()，Base2中声明的vfBase2\_1()以及Base3中声明的vfBase3\_1()。保持其他代码不变，修改后的Derived代码如下：

class Derived : public Base1, public Base2, public Base3

{

public:

         int m\_derived;

         inline virtual void fd()

         {

                   cout << "This is in Derived::fd()" << endl;

         }

         inline void vfBase1\_1()

         {

                   cout << "This is in Derived::vfBase1\_1()" << endl;

         }

         inline void vfBase2\_1()

         {

                   cout << "This is in Derived::vfBase2\_1()" << endl;

         }

         inline void vfBase3\_1()

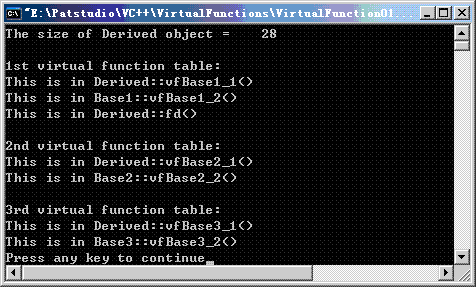
         {

                   cout << "This is in Derived::vfBase3\_1()" << endl;

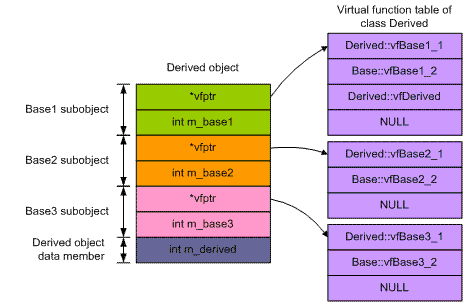
         }

};

运行结果如下：



Derived对象的memory layout图解如下：



Derived中覆盖的虚函数，分别出现在三个不同的虚函数表中，而且分别代替个基类的原虚函数的位置，即：

第一个虚函数表中，Derived::vfBase1\_1()代替了Base::vfBase1\_1()的位置，Base::vfBase1\_1()不再在虚函数表中出现；

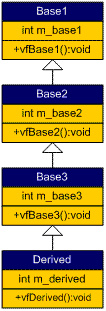
第二个虚函数表中，Derived::vfBase2\_1()代替了Base::vfBase2\_1()的位置，Base::vfBase2\_1()不再在虚函数表中出现；

第三个虚函数表中，Derived::vfBase3\_1()代替了Base::vfBase3\_1()的位置，Base::vfBase3\_1()不再在虚函数表中出现；

在Derived中自己定义的虚函数，总是处在第一个虚函数表的最后一项的位置。

**内存布局之九**

假定多层继承的各类之间的关系如下图。假定派生类不override基类的虚函数，即Base2不override Base1中声明的虚函数vfBase1()，Base3不override Base2中声明的虚函数vfBase2()，Derived不override Base3中声明的虚函数vfBase3()。



代码如下：

#include <iostream>

using namespace std;

class Base1

{

public:

         int m\_base1;

         inline virtual void vfBase1\_1()

         {

                   cout << "This is in Base1::vfBase1\_1()" << endl;

         }

};

class Base2 : public Base1

{

public:

         int m\_base2;

         inline virtual void vfBase2\_1()

         {

                   cout << "This is in Base2::vfBase2\_1()" << endl;

         }

};

class Base3 : public Base2

{

public:

         int m\_Base3;

         inline virtual void vfBase3\_1()

         {

                   cout << "This is in Base3::vfBase3\_1()" << endl;

         }

};

class Derived : public Base3

{

public:

         int m\_derived;

         inline virtual void fd()

         {

                   cout << "This is in Derived::fd()" << endl;

         }

};

typedef void (\*VFun)(void);

template<typename T>

VFun virtualFunctionPointer(T\* b, int i)

{

         return (VFun)(\*((int\*)(\*(int\*)b) + i));

}

int main(void)

{

         Derived d;

         cout << "The size of Derived object = \t" << sizeof(Derived) << endl;

         cout << endl;

         cout << "1st virtual function table: " << endl;

         int i = 0;

         while(virtualFunctionPointer(&d, i))

         {

                   VFun pVF = virtualFunctionPointer(&d, i++);

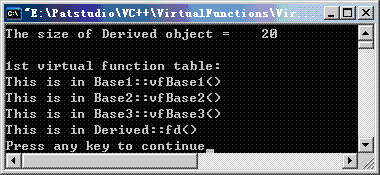
                   pVF();

         }

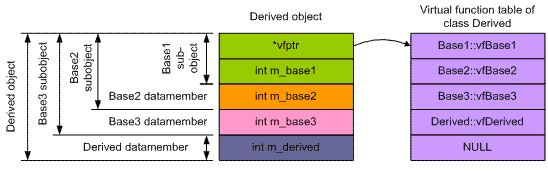
         return 0;

}

运行结果：

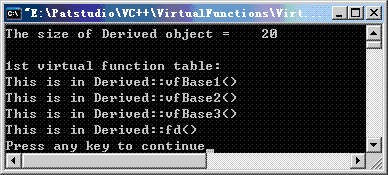


Derived对象的memory layout图解如下：

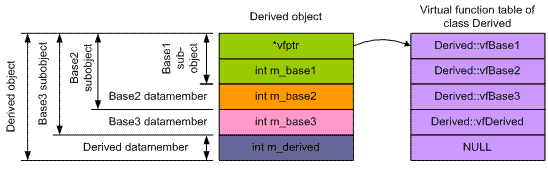


**内存布局之十**

在[对象内存布局 (9)](http://blog.csdn.net/pathuang68/archive/2009/04/23/4103581.aspx)基础上做些修改：派生类override基类的虚函数，即Base2 override Base1中声明的虚函数vfBase1()，Base3 override Base1中声明的虚函数vfBase1()和Base2中声明的虚函数vfBase2()， Derived override Base1中声明的虚函数vfBase1()、Base2中声明的虚函数vfBase2()和Base3中声明的虚函数vfBase3()。运行结果如下：



Derived对象的memory layout图解如下：



**内存布局之十一**

在C++中，一个类实例化得到的结果就是一个对象。一个类包含成员变量和成员函数，其中成员变量又分为nonstatic成员变量和static成员变量；成员函数又可以分为nonstatic成员函数、static成员函数以及virtual成员函数。一个对象包含可能存在的vfptr以及它声明的或基类继承而来的nonstatic成员变量，static成员变量、static成员函数、nonstatic成员函数以及virtual函数均存在于对象之外。

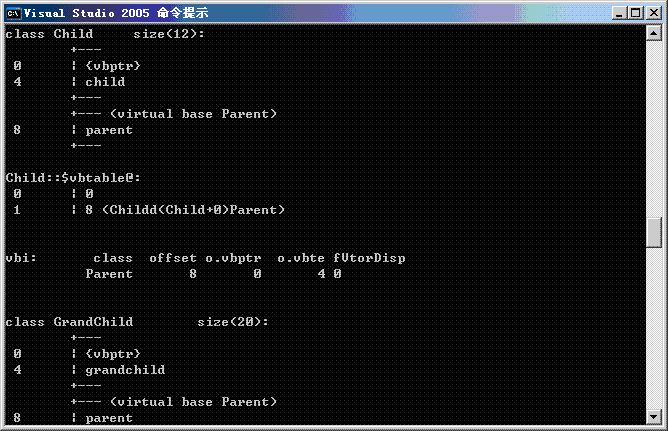
VC2005中有一个非常重要的编译选项：

对于查看类的对象的内存布局，微软内部在VC2005中(要先进入Microsoft Visual Studio -> Visual Studio Tools -> Visual Studio 2005命令提示)提供了一个非常重要的编译选项：/d1reportSingleClassLayout

   比如，如果向查看文件Polymorphism06.cpp中的类Child的对象在内存中的分布情况，先进入cmd命令窗口，改变目录到Polymorphism06.cpp所在的目录，然后键入如下命令：

         cl Polymorphism06.cpp /d1reportSingleClassLayoutChild

回车得到如下结果：



(注意：除上面关于内存布局的信息外还有很多其他信息)

有时我们会发现输出中会出现vtordisp，它到底是是什么意思呢，请看下面MSDN的解释：

The vtordisp pragma is applicable only to code that uses virtual bases. If a derived class overrides a virtual function that it inherits from a virtual base class, and if a constructor or destructor for the derived class calls that function using a pointer to the virtual base class, the compiler may introduce additional hidden "vtordisp" fields into classes with virtual bases.

The vtordisp pragma affects the layout of classes that follow it. The /vd0 and /vd1 options specify the same behavior for complete modules. Specifying off suppresses the hidden vtordisp members. Specifying on, the default, enables them where they are necessary. Turn off vtordisp only if there is no possibility that the class's constructors and destructors call virtual functions on the object pointed to by the this pointer.

vtordisp is now deprecated and will be removed in a future release of Visual C++.

/vd编译器选项会影响全局编译模式。使用vtordisp编译指示可以在基于类方式上打开或禁止vtordisp域:

#pragma vtordisp(off)

class GetReal:virtual public{...};

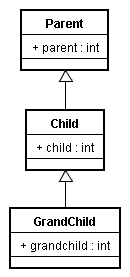
#pragma vtordisp(on)

这是一个过时的东西，以后的VC编译器也不会在支持了。

**内存布局之十二**

注意：关于内存对齐(memory alignment)，请看[关于内存对齐问题](http://blog.csdn.net/pathuang68/archive/2009/04/18/4088824.aspx)，后面将会用到。

下面我们进行在普通继承(即非虚继承)时，派生类的指针转换到基类指针的情形研究。假定各类之间的关系如下图：



代码如下：

#include <iostream>

using namespace std;

#pragma vtordisp(off)

class Parent

{

public:

     int parent;

};

class Child : public Parent

{

public:

     int child;

};

class GrandChild : public Child

{

public:

     int grandchild;

};

int main(void)

{

     Child\* pc = new Child();

     GrandChild\* pgc = new GrandChild();

     cout << "1. The address of Child object:\t\t";

     cout << (unsigned long\*)pc << endl;

     cout << "2. The address of GrandChild object:\t";

     cout << (unsigned long\*)pgc << endl;

    // 将类Child对象的指针pc，向上转型(upcast)到其父类型指针，即Parent\*

     Parent\* pp = (Parent\*)pc;

     cout << "3. Child\* casted to Parent\*:\t\t";

     cout << (unsigned long\*)pp << endl;

    // 将类GrandChild对象的指针pgc，向上转型(upcast)到其父类型指针，即Child\*

     Child\* pc2 = (Child\*)pgc;

     cout << "4. GrandChild\* casted to Child\*:\t";

     cout << (unsigned long\*)pc2 << endl;

    // 再将上面通过转型得到的类Child对象的指针pc2，向上转型(upcast)到其父类型指针，即Parent\*

     Parent\* pp2 = (Parent\*)pc2;

     cout << "5. Child\* casted to Parent\*:\t\t";

     cout << (unsigned long\*)pp2 << endl;

    // 将类GrandChild对象的指针pgc，向上转型(upcast)到其祖类型指针，即Parent\*

     Parent\* pp3 = (Parent\*)pgc;

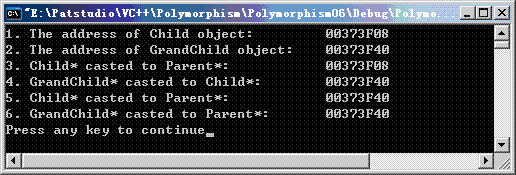
     cout << "6. GrandChild\* casted to Parent\*:\t";

     cout << (unsigned long\*)pp3 << endl;

     return 0;

}

得到如下结果：



我们发现在普通继承的情况下，将派生类对象的指针upcast为基类指针时，指针的值并不会发生改变。

比如上面输出中的1和3是一样的。

         Child\* pc = new Child();             ->               pc = 0x00373F08

         Parent\* pp = (Parent\*)pc;           ->               pp = 0x00373F08

还有上面的2和4的输出也是一样的：

         GrandChild\* pgc = new GrandChild();       ->          pgc = 0x00373F40

         Child\* pc2 = (Child\*)pgc;                              ->          pc2 = 0x00373F40

保持整个程序其他部分代码不做任何变动，我们将Child改为从Parent虚继承，改后的Child代码如下：

class Child : public virtual Parent

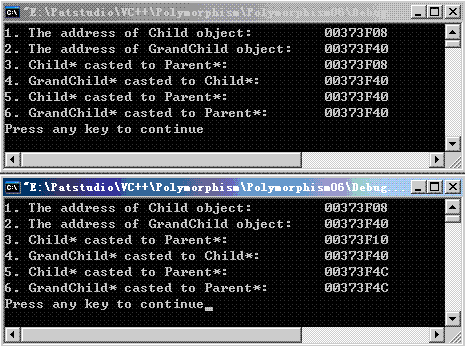
{

public:

     int child;

};

运行结果为下图值下半部分：



现在来一一比较两者之间的不同：

第1条，两者相同；

第2条，两者相同；

第3条，由0x00373F08变成了0x00373F10了，也就是说经过Parent\* pp = (Parent\*)pc;后，即由pc = 0x00373F08得到了pp = 0x00373F10。很奇怪！

第4条，两者相同；

第5条，由0x00373F40变成了0x00373F4C了，也就是说经过Parent\* pp2 = (Parent\*)pc2;后，即由pc2 = 0x00373F40得到了pp2 = 0x00373F4C。很奇怪！

第6条，由0x00373F40变成了0x00373F4C了，也就是说经过Parent\* pp3 = (Parent\*)pgc;后，即由pgc = 0x00373F40得到了pp3 = 0x00373F4C。很奇怪！

为什么会这样呢？通过上述分析发现，出现这种指针发生变化的情况，均发生在将Child\*或者GrandChild\*转换到Parent\*的各行。Parent是Child的虚基类，Child又是GrandChild的基类。在上面的第4条中，我们通过Child\* pc2 = (Child\*)pgc;，试图将GrandChild\*转换为Child\*，事实上也转换成功了，同时指针的值并没有发生改变。GrandChild是普通继承于Child的，而非虚拟继承，换言之，Child不是GrandChild的虚基类，所以指针转换时，目标指针的值和赋给它的值保持一致。通过这样的分析我们似乎可以得出下面的结论：

当一个派生类对象的指针转换到虚基类指针时(不管两者之间是否有其他中间类，而且也不管这些中间类是否是派生类的普通基类还是虚基类)，指针的值就会发生变化。

为了验证上述结论，在上面的基础上，我们将GrandChild改为虚拟继承Child，修改后的GrandChild代码如下：

class GrandChild : public virtual Child

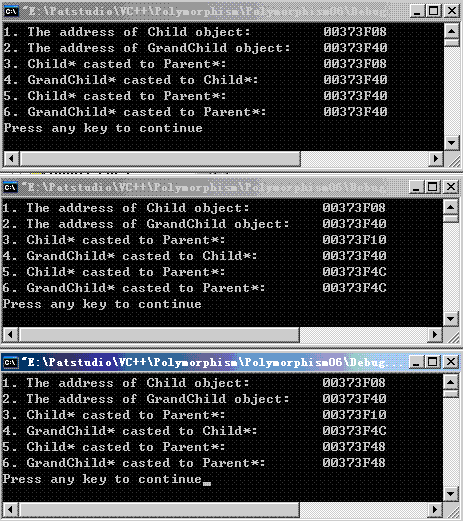
{

public:

         int grandchild;

};

运行程序，得到如下结果(下图的最下面一部分)：



我们看到，现在第4条也发生了变化。因此原来的结论是成立的。再次总结一下这条非常重要的结论：

如果没有虚拟继承，当将派生类对象的指针转换到基类时(即使基类中有虚函数)，指针的值不会发生变化；但当一个派生类对象的指针转换到虚基类指针时(不管两者之间是否有其他中间类，而且也不管这些中间类是否是派生类的普通基类还是虚基类)，指针的值就会发生变化。

**这个结论对后面的了解含有虚基类的对象内存布局有着非同一般的意义。**对于这个结论，我们还剩下一个问题，那就是为什么会这样呢？ 前面我们可以看到赋值后的指针的值并不等于赋给它的对象地址值。也就是说在这个赋值过程中编译器进行了额外的工作，即调整了指针的值。我们看看上面程序中Parent\* pp = (Parent\*)pc; (向上类型转换，即up-casting) 这行对应的汇编代码(在VC中，进行debug时，按Alt 8，即可查看到汇编代码)，看看编译器究竟做了些什么？

38:       Parent\* pp = (Parent\*)pc;

00401691   cmp         dword ptr [ebp-10h],0

00401695   jne          main+120h (004016a0)

00401697   mov         dword ptr [ebp-40h],0

0040169E   jmp         main+12Eh (004016ae)

004016A0   mov         eax,dword ptr [ebp-10h]

004016A3   mov         ecx,dword ptr [eax]                           // 6

004016A5   mov         edx,dword ptr [ebp-10h]                   // 7

004016A8   add         edx,dword ptr [ecx+4]                       // 8

004016AB   mov         dword ptr [ebp-40h],edx

004016AE   mov         eax,dword ptr [ebp-40h]

004016B1   mov         dword ptr [ebp-18h],eax

重要的是第6、7、8行代码，它们通过偏移值指针找到偏移值，并以此来调整指针的位置，让目的指针最终指向对象中的基类部分的数据成员。

至此，我们解释清楚了上面的问题。因为这部分讨论的结果太重要了，我们不妨再次总结如下：

如果没有虚拟继承，当将派生类对象的指针转换到基类时(即使基类中有虚函数)，指针的值不会发生变化；但当一个派生类对象的指针转换到虚基类指针时(不管两者之间是否有其他中间类，而且也不管这些中间类是否是派生类的普通基类还是虚基类)，**指针(目的指针)的值就会发生变化，目的指针最终指向对象中的基类部分(或曰基类的实例)。**

**内存布局之十三**

下面来看看虚基类对对象内存布局的影响。虚基类的主要作用就是在所有的派生类中，保留且仅保留一份虚基类的suboject。

**a. 一个虚基类的情况**

#include <iostream>

using namespace std;

class Base

{

public:

         int base\_member;

};

class Derived : public virtual Base {};

int main(void)

{

         Base b;

         Derived d;

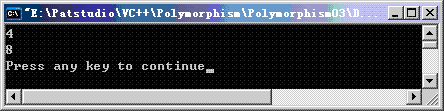
         cout << sizeof(b) << endl;

         cout << sizeof(d) << endl;

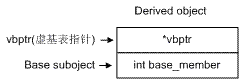
         return 0;

}

运行结果：



注意使用了虚继承，即class Derived : public **virtual** Base。这次Derived的对象大小为什么为8 bytes呢？这是因为编译器会给Derived对象安插一个虚基类表指针vbptr，下面给出Derived对象的memory layout：



虚基类表指针vbptr指向Derived类的virtual bass class table(虚基类表)，虚基类表中存放的是Derived类的虚基类表指针到虚基类实例指针的偏移量。

如果Derived有两个虚基类，那么这两个虚基类的实例指针，分别位于虚基类表的第2项和第3项，最后一项为0，意味着虚基类表的结束。虚基类表中的第1项是什么？目前不清楚，但是它的值大部分时候为0(请牛人指教，编译器是VC6)。下面我们来检验之。

在main函数的return前，增加如下语句：

         cout << "Derived object d's vbptr = " << (unsigned long\*)(&d) << endl;

         cout << "Address of virtual base class table = " << (unsigned long\*)\*(unsigned long\*)(&d) << endl;

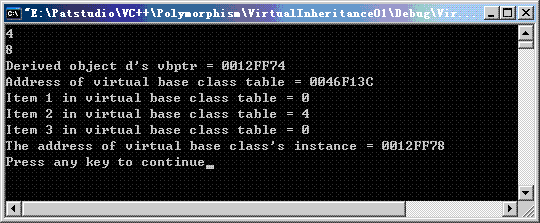
         cout << "Item 1 in virtual base class table = " << \*(unsigned long\*)\*(unsigned long\*)(&d) << endl;

         cout << "Item 2 in virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&d) + 1) << endl;

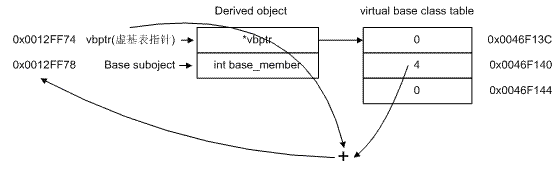
         cout << "Item 3 in virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&d) + 2) << endl;

         cout << "The address of virtual base class Base = " << (Base\*)(&d) << endl;

编译后运行结果：



不难发现，虚基类示例地址 = vbptr + offset，即0x0012FF78 = 0x0012FF74 + 4。图示如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=5f5fff010100cwpj&url=http://static12.photo.sina.com.cn/orignal/5f5fff01g683b79bcf18b)

**b. 我们来看看Derived有两个虚基类的情况：**

#include <iostream>

using namespace std;

class Base1

{

public:

         int base1\_member;

};

class Base2

{

public:

         int base2\_member;

};

class Derived : public virtual Base1, public virtual Base2 {};

int main(void)

{

         Base1 b1;

         Base2 b2;

         Derived d;

         cout << sizeof(b1) << endl;

         cout << sizeof(b2) << endl;

         cout << sizeof(d) << endl;

         cout << "Derived object d's vbptr = " << (unsigned long\*)(&d) << endl;

         cout << "Address of virtual base class table = " << (unsigned long\*)\*(unsigned long\*)(&d) << endl;

         cout << "Item 1 in virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&d) + 0) << endl;

         cout << "Item 2 in virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&d) + 1) << endl;

         cout << "Item 3 in virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&d) + 2) << endl;

         cout << "Item 3 in virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&d) + 3) << endl;

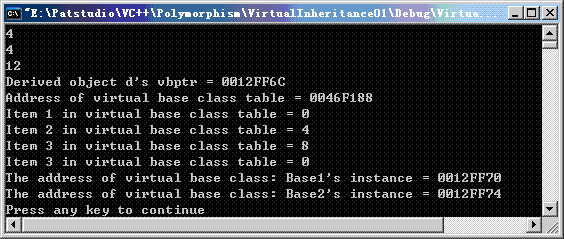
         cout << "The address of virtual base class: Base1's instance = " << (Base1\*)(&d) << endl;

         cout << "The address of virtual base class: Base2's instance = " << (Base2\*)(&d) << endl;

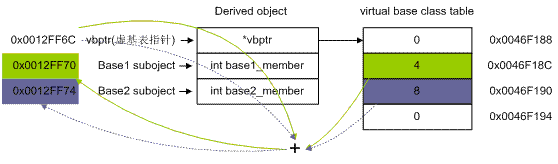
         return 0;

}

编译运行结果如下：



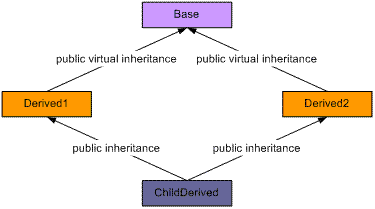
Derived对象的memory layout图解如下：



不管Derived类有多少个虚基类，它只有一个vbptr和一个virtual base class table。

**内存布局之十四**

继续探讨虚基类对对象内存布局的影响。几个类的继承关系如下图，这是虚基类最为常见的用法之一：



代码如下：

#include <iostream>

using namespace std;

class Base

{

public:

         int base\_member;

};

class Derived1 : public virtual Base

{

public:

         int derived1\_member;

};

class Derived2 : public virtual Base

{

public:

         int derived2\_member;

};

class ChildDerived : public Derived1, public Derived2

{

public:

         int childderived\_member;

};

int main(void)

{

         ChildDerived cd;

         cout << "sizeof(Base) = \t\t" << sizeof(Base) << endl;

         cout << "sizeof(Derived1) = \t" << sizeof(Derived1) << endl;

         cout << "sizeof(Derived2) = \t" << sizeof(Derived2) << endl;

         cout << "sizeof(ChildDerived) = \t" << sizeof(ChildDerived) << endl;

         cout << endl;

         cout << "Derived1 subobject's vbptr = " << (unsigned long\*)&cd << endl;

         cout << "Derived2 subobject's vbptr = " << (unsigned long\*)&cd + 2 << endl;

         cout << endl;

         cout << "Address of virtual base class table of Derived1 object = " << (unsigned long\*)(\*((unsigned long\*)&cd)) << endl;

         cout << "Item 1 in Derived1's virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&cd) + 0) << endl;

         cout << "Item 2 in Derived1's virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&cd) + 1) << endl;

         cout << "Item 3 in Derived1's virtual base class table = " << \*((unsigned long\*)\*(unsigned long\*)(&cd) + 2) << endl;

         cout << endl;

         cout << "Address of virtual base class table of Derived2 object = " << (unsigned long\*)\*((unsigned long\*)&cd + 2) << endl;

         cout << "Item 1 in Derived2's virtual base class table = " << \*((unsigned long\*)\*((unsigned long\*)&cd + 2) + 0) << endl;

         cout << "Item 2 in Derived2's virtual base class table = " << \*((unsigned long\*)\*((unsigned long\*)&cd + 2) + 1) << endl;

         cout << "Item 3 in Derived2's virtual base class table = " << \*((unsigned long\*)\*((unsigned long\*)&cd + 2) + 2) << endl;

         cout << endl;

         cout << "The address of base class Derived1's instance = " << (Derived1\*)&cd << endl;

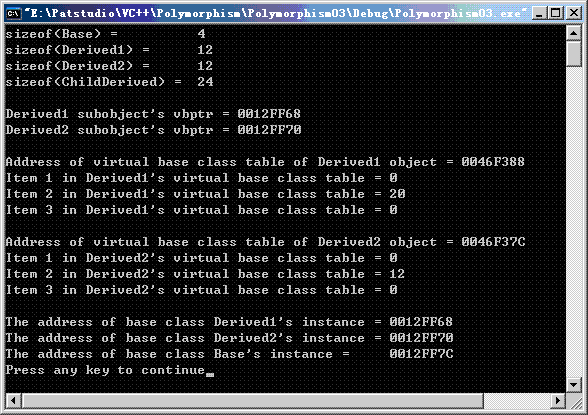
         cout << "The address of base class Derived2's instance = " << (Derived2\*)&cd << endl;

         cout << "The address of base class Base's instance = \t" << (Base\*)&cd << endl;

         return 0;

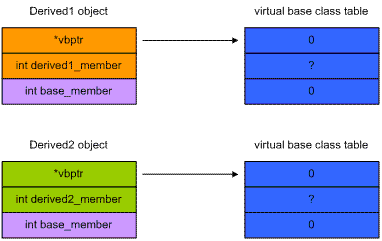
}

运行结果如下：



sizeof(Base) = 4 bytes，这个是很好解释的，因为Base中由一个类型为int的成员变量。

sizeof(Derived1) = 12 bytes和sizeofDerived2) = 12 bytes，图解如下：



两个“?”到底是多少我们且不去管它，等我们讨论ChildDerived对象的memory layout时再进行详细说明。从上图可以看到，Base subobject在最后。

现在我们来看看ChildDerived对象的memory layout的情况：

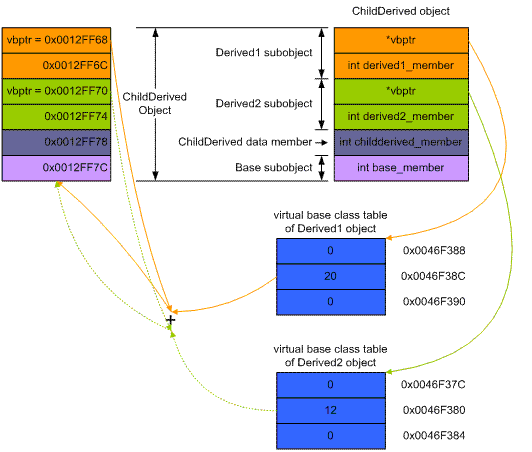
由于Derived1和Derived2都是从Base虚拟继承而来，因而Base是它们的虚基类，编译器在编译的时候会分别给他们安插vbptr指针；Derived1和Derived2同时被ChildDerived普通继承(非虚继承)，根据C++标准的要求，基类在在派生类中保证其原始的完整性，因此两个vbptr被继承到了ChildDerived类；由于Base被虚继承，可以看到Base suboject只有一份拷贝，而且放在最后。

从下面图解可以很清楚地发现：

sizeof(ChildDerived) = 24 bytes；

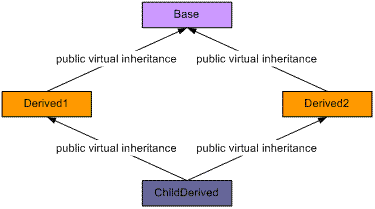
虚基类实例地址(0x0012FF7C) = Derived1的vbptr (0x0012FF68) + Derived1的vbptr到虚基类实例地址的偏移量(24)；

虚基类实例地址(0x0012FF7C) = Derived2的vbptr (0x0012FF70) + Derived1的vbptr到虚基类实例地址的偏移量(12)



**内存布局之十五**

和[对象内存布局 (14)](http://blog.csdn.net/pathuang68/archive/2009/04/24/4105851.aspx)差不多，只是改动了两个继承关系。几个类的继承关系如下图，这种继承方式不是很有实际意义，在此纯粹是探究类的memory layout。



代码如下：

#include <iostream>

using namespace std;

class Base

{

public:

         int base\_member;

};

class Derived1 : public virtual Base

{

public:

         int derived1\_member;

};

class Derived2 : public virtual Base

{

public:

         int derived2\_member;

};

class ChildDerived : public virtual Derived1, public virtual Derived2

{

public:

         int childderived\_member;

};

int main(void)

{

         ChildDerived cd;

         cout << "sizeof(Base) = \t\t" << sizeof(Base) << endl;

         cout << "sizeof(Derived1) = \t" << sizeof(Derived1) << endl;

         cout << "sizeof(Derived2) = \t" << sizeof(Derived2) << endl;

         cout << "sizeof(ChildDerived) = \t" << sizeof(ChildDerived) << endl;

         cout << endl;

         cout << "ChildDerived object's vbptr = " << (int\*)&cd << endl;

         cout << endl;

         cout << "Address of virtual base class table of ChildDerived object = " << (int\*)(\*((int\*)&cd)) << endl;

         cout << "Item 1 in ChildDerived's virtual base class table = " << \*((int\*)\*(int\*)(&cd) + 0) << endl;

         cout << "Item 2 in ChildDerived's virtual base class table = " << \*((int\*)\*(int\*)(&cd) + 1) << endl;

         cout << "Item 3 in ChildDerived's virtual base class table = " << \*((int\*)\*(int\*)(&cd) + 2) << endl;

         cout << "Item 4 in ChildDerived's virtual base class table = " << \*((int\*)\*(int\*)(&cd) + 3) << endl;

         cout << "Item 4 in ChildDerived's virtual base class table = " << \*((int\*)\*(int\*)(&cd) + 4) << endl;

         cout << endl;

         cout << "Address of virtual base class table of Derived1 object = " << (int\*)\*((int\*)&cd + 3) << endl;

         cout << "Item 1 in Derived1's virtual base class table = " << \*((int\*)\*((int\*)&cd + 3) + 0) << endl;

         cout << "Item 2 in Derived1's virtual base class table = " << \*((int\*)\*((int\*)&cd + 3) + 1) << endl;

         cout << "Item 3 in Derived1's virtual base class table = " << \*((int\*)\*((int\*)&cd + 3) + 2) << endl;

         cout << endl;

         cout << "Address of virtual base class table of Derived2 object = " << (int\*)\*((int\*)&cd + 5) << endl;

         cout << "Item 1 in Derived2's virtual base class table = " << \*((int\*)\*((int\*)&cd + 5) + 0) << endl;

         cout << "Item 2 in Derived2's virtual base class table = " << \*((int\*)\*((int\*)&cd + 5) + 1) << endl;

         cout << "Item 3 in Derived2's virtual base class table = " << \*((int\*)\*((int\*)&cd + 5) + 2) << endl;

         cout << endl;

         cout << "The address of base class Derived1's instance = " << (Derived1\*)&cd << endl;

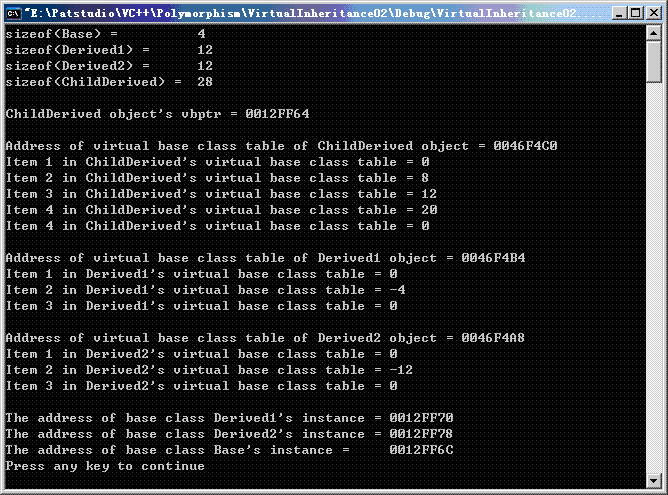
         cout << "The address of base class Derived2's instance = " << (Derived2\*)&cd << endl;

         cout << "The address of base class Base's instance = \t" << (Base\*)&cd << endl;

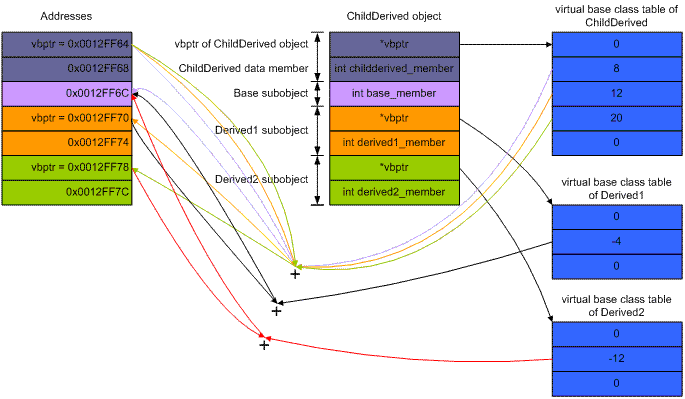
         return 0;

}

注意：因为可能出现负数，所以在上面的程序中，将原来的unsigned long\* 改为了int\*。运行结果：



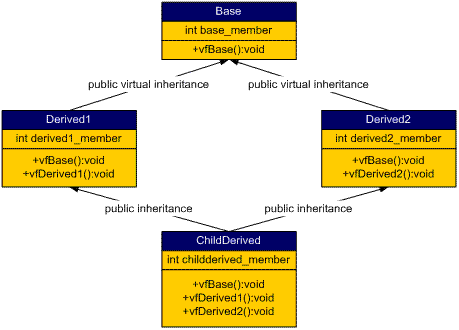
图解如下：



**内存布局之十六**

下面讨论虚基类和虚函数同时存在的时候，对对象内存布局的影响。

假定各个类之间的关系如下图：



Base中声明了一个虚函数vfBase()和一个整形成员变量；

Derived1 override了Base中声明的虚函数vfBase()，声明了一个虚函数vfDerived1()，另有一个整形成员变量derived1\_member；

Derived2 override了Base中声明的虚函数vfBase()，声明了一个虚函数vfDerived2()，另有一个整形成员变量derived2\_member；

ChildDerived分别override了Base、Derived1和Derived2中声明的虚函数vfBase()、vfDerived1() 和vfDerived2()，另外有一个整形成员变量childderived\_member；

代码如下：

#include <iostream>

using namespace std;

typedef void (\*VFun)(void);

template<typename T>

VFun virtualFunctionPointer(T\* b, int i)

{

         return (VFun)(\*((int\*)(\*(int\*)b) + i));

}

template<typename T>

int virtualBaseTableOffset(T\* b, int i)

{

         return (int)\*((int\*)\*(int\*)b + i);

}

class Base

{

public:

         int base\_member;

         inline virtual void vfBase()

         {

                   cout << "This is in Base::vfBase()" << endl;

         }

};

class Derived1 : public virtual Base

{

public:

         int derived1\_member;

         inline void vfBase()

         {

                   cout << "This is in Derived1::vfBase()" << endl;

         }

         inline virtual void vfDerived1()

         {

                   cout << "This is in Derived1::vfDerived1()" << endl;

         }

};

class Derived2 : public virtual Base

{

public:

         int derived2\_member;

         inline void vfBase()

         {

                   cout << "This is in Derived2::vfBase()" << endl;

         }

         inline virtual void vfDerived2()

         {

                   cout << "This is in Derived2::vfDerived2()" << endl;

         }

};

class ChildDerived : public Derived1, public Derived2

{

public:

         int childderived\_member;

         inline void vfBase()

         {

                   cout << "This is in ChildDerived::vfBase()" << endl;

         }

         inline void vfDerived1()

         {

                   cout << "This is in ChildDerived::vfDerived1()" << endl;

         }

         inline void vfDerived2()

         {

                   cout << "This is in ChildDerived::vfDerived2()" << endl;

         }

};

int main(void)

{

         ChildDerived cd;

         VFun pVF;

         int\* tmp;

         cout << "sizeof(Base) = \t\t" << sizeof(Base) << endl;

         cout << "sizeof(Derived1) = \t" << sizeof(Derived1) << endl;

         cout << "sizeof(Derived2) = \t" << sizeof(Derived2) << endl;

         cout << "sizeof(ChildDerived) = \t" << sizeof(ChildDerived) << endl;

cout << endl;

         cout << "address of ChildDerived object:" << endl;

         cout << "address = " << (int\*)&cd << endl;

cout << endl;

         cout << "1st virtual function table: " << endl;

         pVF = virtualFunctionPointer(&cd, 0);

         pVF();

         cout << endl;

         cout << "1st virtual base table: " << endl;

         tmp = (int\*)((int\*)&cd + 1);

         cout << "address = " << tmp << endl;

         cout << virtualBaseTableOffset(tmp, 0) << "\t<- not sure yet, but it doesn't matter here." << endl;

         cout << virtualBaseTableOffset(tmp, 1) << "\t<- offset from Derived1 subobject's vbptr to Base subobject." << endl;

         cout << virtualBaseTableOffset(tmp, 2) << "\t<- means the end of this virtual base table."  << endl;

         cout << endl;

         tmp = ((int\*)&cd) + 3;

         cout << "2nd virtual function table: " << endl;

         pVF = virtualFunctionPointer(tmp, 0);

         pVF();

         cout << endl;

         cout << "2nd virtual base table: " << endl;

         tmp = (int\*)((int\*)&cd + 4);

         cout << "address = " << tmp << endl;

         cout << virtualBaseTableOffset(tmp, 0) << "\t<- not sure yet, but it doesn't matter here." << endl;

         cout << virtualBaseTableOffset(tmp, 1) << "\t<- offset from Derived2 subobject's vbptr to Base subobject."  << endl;

         cout << virtualBaseTableOffset(tmp, 2) << "\t<- means the end of this virtual base table."  << endl;

         cout << endl;

         tmp = ((int\*)&cd) + 7;

         cout << "3rd virtual function table: " << endl;

         pVF = virtualFunctionPointer(tmp, 0);

         pVF();

         cout << endl;

         cout << "Derived1 subobject address = \t" << (Derived1\*)&cd << endl;

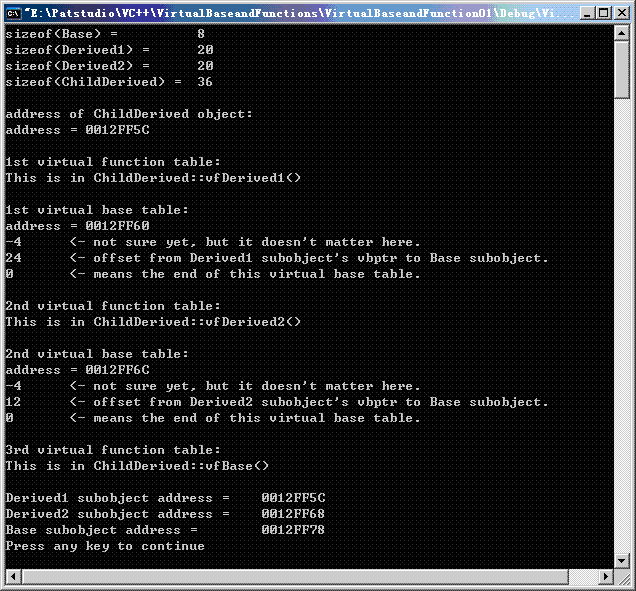
         cout << "Derived2 subobject address = \t" << (Derived2\*)&cd << endl;

         cout << "Base subobject address = \t" << (Base\*)&cd << endl;

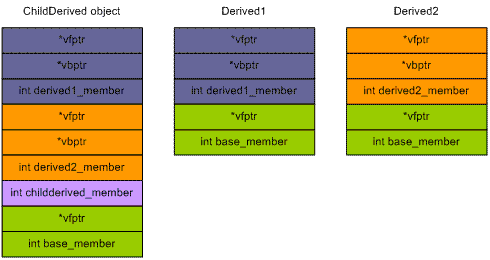
         return 0;

}

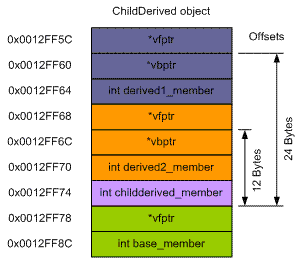
运行结果如下：



ChildDerived、Derived1和Derived2对象memory layout分别图解如下：



两个虚基类偏移量图解如下(虚函数表和虚基类表略)：



结论：

其一，只要涉及到虚基类，一切问题就变得复杂起来；

其二，如果同时存在vfptr和vbptr，vfptr居前，vbptr居后；

其三，普通基类居前，虚基类总是尽可能地排列在layout的最后；

其四，两个同一层次的虚基类subobject，先声明者居前，后声明者居后，这点和普通基类是一样的；

其五，两个不同层次的虚基类subobject，层次高者居前，层次低者居后；

其六，Stan Lippman建议，不要在一个virtual base class中声明nonstatic data member，理由是这样做会是问题变得非常复杂。

**内存布局之十七**