虚函数表解析

C++中的虚函数的作用主要是实现了多态的机制。

关于多态，简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”，这是一种泛型技术。

所谓泛型技术，说白了就是试图使用不变的代码来实现可变的算法。比如：模板技术，RTTI 技术，虚函数技术，要么是试图做到在编译时决议，要么试图做到运行时决议。

关于虚函数的使用方法，我在这里不做过多的阐述。大家可以看看相关的 C++的书籍。在这篇文章中，我只想从虚函数的实现机制上面为大家一个清晰的剖析。

当然，相同的文章在网上也出现过一些了，但我总感觉这些文章不是很容易阅读，大段大段的代码，没有图片，没有详细的说明，没有比较，没有举一反三。不利于学习和阅读，所以这是我想写下这篇文章的原因。也希望大家多给我提意见。

言归正传，让我们一起进入虚函数的世界。

虚函数表

对 C++ 了解的人都应该知道虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。简称为 V-Table。在这个表中，主是要一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。

这样，在有虚函数的类的实例中这个表被分配在了这个实例的内存中，所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

这里我们着重看一下这张虚函数表。

C++的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

听我扯了那么多，我可以感觉出来你现在可能比以前更加晕头转向了。没关系，下面就是实际的例子，相信聪明的你一看就明白了。

假设我们有这样的一个类：

```
1class Base {
2    public:
3        virtual void f() { cout << "Base::f" << endl; }
4        virtual void g() { cout << "Base::g" << endl; }
```

```
5           virtual void h() { cout << "Base::h" << endl; }
6
7};
```

按照上面的说法，我们可以通过 Base 的实例来得到虚函数表。 下面是实际例程：

```
1typedef void(*Fun)(void);
2
3Base b;
4
5Fun pFun = NULL;
6
7cout << "虚函数表地址：" << (int*)(&b) << endl;
8cout << "虚函数表 — 第一个函数地址：" << (int*)*(int*)(&b) << endl;
9
10// Invoke the first virtual function
11pFun = (Fun)*((int*)*(int*)(&b));
12pFun();
```

实际运行经果如下：(Windows XP+VS2003, Linux 2.6.22 + GCC 4.1.3)

虚函数表地址：0012FED4

虚函数表 — 第一个函数地址：0044F148

Base::f

通过这个示例，我们可以看到，我们可以通过强行把&b 转成 int *，取得虚函数表的地址，然后，再次取址就可以得到第一个虚函数的地址了，也就是 Base::f()，这在上面的程序中得到了验证（把 int* 强制转成了函数指针）。通过这个示例，我们就可以知道如果要调用 Base::g() 和 Base::h()，其代码如下：

```
1(Fun)*((int*)*(int*)(&b)+0);  // Base::f()
2(Fun)*((int*)*(int*)(&b)+1);  // Base::g()
3(Fun)*((int*)*(int*)(&b)+2);  // Base::h()
```

这个时候你应该懂了吧。什么？还是有点晕。也是，这样的代码看着太乱了。没问题，让我画个图解释一下。如下所示：

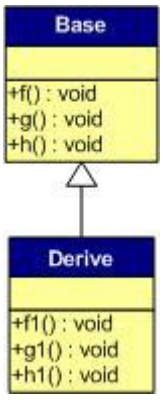
注意：在上面这个图中，我在虚函数表的最后多加了一个结点，这是虚函数表的结束结点，就像字符串的结束符“/0”一样，其标志了虚函数表的结束。这个结束标志的值在不同的编译器下是不同的。在

WinXP+VS2003 下，这个值是 NULL。而在 Ubuntu 7.10 + Linux 2.6.22 + GCC 4.1.3 下，这个值是如果 1，表示还有下一个虚函数表，如果值是 0，表示是最后一个虚函数表。

下面，我将分别说明“无覆盖”和“有覆盖”时的虚函数表的样子。没有覆盖父类的虚函数是毫无意义的。我之所以要讲述没有覆盖的情况，主要目的是为了给一个对比。在比较之下，我们可以更加清楚地知道其内部的具体实现。

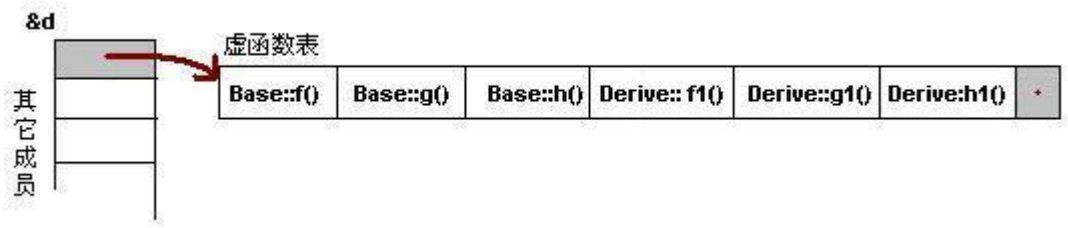
一般继承（无虚函数覆盖）

下面，再让我们来看看继承时的虚函数表是什么样的。假设有如下所示的一个继承关系：



请注意，在这个继承关系中，子类没有重载任何父类的函数。那么，在派生类的实例中，其虚函数表如下所示：

对于实例：Derive d; 的虚函数表如下：



我们可以看到下面几点：

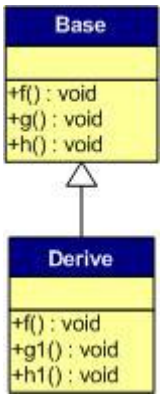
1）虚函数按照其声明顺序放于表中。

2）父类的虚函数在子类的虚函数前面。

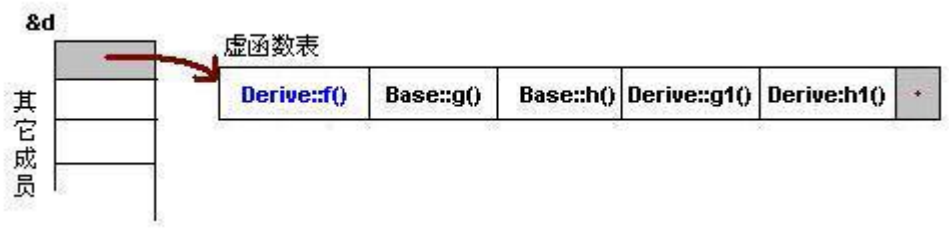
我相信聪明的你一定可以参考前面的那个程序，来编写一段程序来验证。

一般继承（有虚函数覆盖）

覆盖父类的虚函数是很显然的事情，不然，虚函数就变得毫无意义。下面，我们来看一下，如果子类中有虚函数重载了父类的虚函数，会是一个什么样子？假设，我们有下面这样的一个继承关系。



为了让大家看到被继承过后的效果，在这个类的设计中，我只覆盖了父类的一个函数：f()。那么，对于派生类的实例，其虚函数表会是下面的一个样子：



我们从表中可以看到下面几点，

1）覆盖的 f()函数被放到了虚表中原来父类虚函数的位置。

2）没有被覆盖的函数依旧。

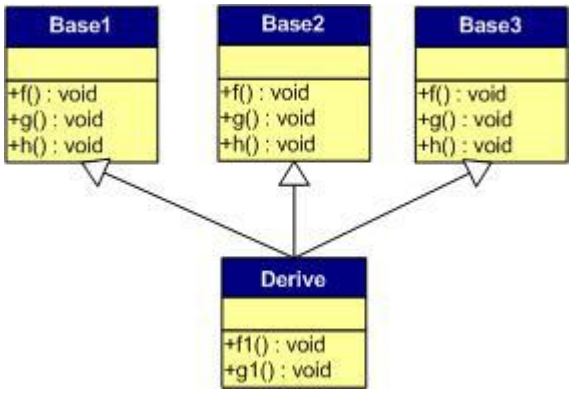
这样，我们就可以看到对于下面这样的程序，

```
1Base *b = new Derive();
2
3b->f();
```

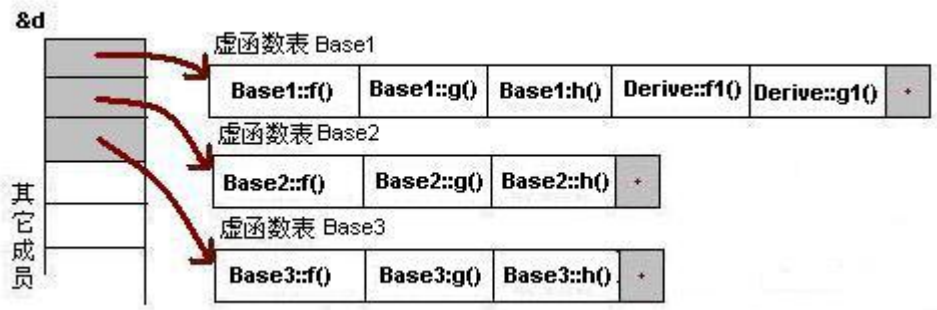
由 b 所指的内存中的虚函数表的 f()的位置已经被 Derive::f()函数地址所取代，于是在实际调用发生时，是 Derive::f()被调用了。这就实现了多态。

多重继承（无虚函数覆盖）

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：子类并没有覆盖父类的函数。



对于子类实例中的虚函数表，是下面这个样子：



我们可以看到：

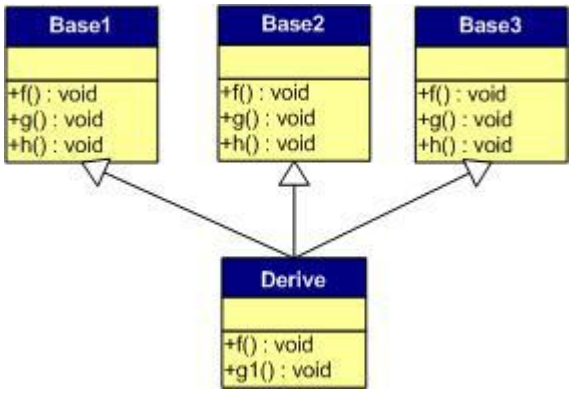
1） 每个父类都有自己的虚表。

2） 子类的成员函数被放到了第一个父类的表中。（所谓的第一个父类是按照声明顺序来判断的）

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。

多重继承（有虚函数覆盖）

下面我们再来看看，如果发生虚函数覆盖的情况。



下图中，我们在子类中覆盖了父类的 f()函数。

下面是对于子类实例中的虚函数表的图：

我们可以看见，三个父类虚函数表中的 f()的位置被替换成了子类的函数指针。这样，我们就可以任一静态类型的父类来指向子类，并调用子类的 f()了。如：

```
1Derive d;
2Base1 *b1 = &d;
3Base2 *b2 = &d;
4Base3 *b3 = &d;
5b1->f(); //Derive::f()
```

```
b2->f(); //Derive::f()
b3->f(); //Derive::f()

b1->g(); //Base1::g()
b2->g(); //Base2::g()
b3->g(); //Base3::g()
```

安全性

每次写 C++的文章，总免不了要批判一下 C++。这篇文章也不例外。通过上面的讲述，相信我们对虚函数表有一个比较细致的了解了。水可载舟，亦可覆舟。下面，让我们来看看我们可以用虚函数表来干点什么坏事吧。

一、通过父类型的指针访问子类自己的虚函数

我们知道，子类没有重载父类的虚函数是一件毫无意义的事情。因为多态也是要基于函数重载的。虽然在上面的图中我们可以看到 Base1 的虚表中有 Derive 的虚函数，但我们根本不可能使用下面的语句来调用子类的自有虚函数：

```
Base1 *b1 = new Derive();
b1->f1();  //编译出错
```

任何妄图使用父类指针想调用子类中的**未覆盖父类的成员函数**的行为都会被编译器视为非法，所以，这样的程序根本无法编译通过。但在运行时，我们可以通过指针的方式访问虚函数表来达到违反 C++语义的行为。（关于这方面的尝试，通过阅读后面附录的代码，相信你可以做到这一点）

二、访问 non-public 的虚函数

另外，如果父类的虚函数是 private 或是 protected 的，但这些非 public 的虚函数同样会存在于虚函数表中，所以，我们同样可以使用访问虚函数表的方式来访问这些 non-public 的虚函数，这是很容易做到的。

如：

```
1class Base {
2   private:
3           virtual void f() { cout << "Base::f" << endl; }
4
5};
6
7class Derive : public Base{
8
```

```
};

typedef void(*Fun)(void);

void main() {
    Derive d;
    Fun  pFun = (Fun)*((int*)*(int*)(&d)+0);
    pFun();
}
```

结束语

C++这门语言是一门 Magic 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，需要去了解 C++中那些危险的东西。不然，这是一种搬起石头砸自己脚的编程语言。

附录一：VC 中查看虚函数表

我们可以在 VC 的 IDE 环境中的 Debug 状态下展开类的实例就可以看到虚函数表了（并不是很完整的）

附录 二：例程

下面是一个关于多重继承的虚函数表访问的例程：

```
#include <iostream>
using namespace std;

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

};

class Base2 {
public:
        virtual void f() { cout << "Base2::f" << endl; }
        virtual void g() { cout << "Base2::g" << endl; }
        virtual void h() { cout << "Base2::h" << endl; }
};

class Base3 {
public:
```

```
21          virtual void f() { cout << "Base3::f" << endl; }
22          virtual void g() { cout << "Base3::g" << endl; }
23          virtual void h() { cout << "Base3::h" << endl; }
24};
25
26class Derive : public Base1, public Base2, public Base3 {
27public:
28          virtual void f() { cout << "Derive::f" << endl; }
29          virtual void g1() { cout << "Derive::g1" << endl; }
30};
31
32typedef void(*Fun)(void);
33
34int main()
35{
36          Fun pFun = NULL;
37
38          Derive d;
39          int** pVtab = (int**)&d;
40
41          //Base1's vtable
42          //pFun = (Fun)*((int*)*(int*)((int*)&d+0)+0);
```

```
43          pFun = (Fun)pVtab[0][0];
44          pFun();
45
46          //pFun = (Fun)*((int*)*(int*)((int*)&d+0)+1);
47          pFun = (Fun)pVtab[0][1];
48          pFun();
49
50          //pFun = (Fun)*((int*)*(int*)((int*)&d+0)+2);
51          pFun = (Fun)pVtab[0][2];
52          pFun();
53
54          //Derive's vtable
55          //pFun = (Fun)*((int*)*(int*)((int*)&d+0)+3);
56          pFun = (Fun)pVtab[0][3];
57          pFun();
58
59          //The tail of the vtable
60          pFun = (Fun)pVtab[0][4];
61          cout<<pFun<<endl;
62
63          //Base2's vtable
64          //pFun = (Fun)*((int*)*(int*)((int*)&d+1)+0);
```

```
65          pFun = (Fun)pVtab[1][0];
66          pFun();
67
68          //pFun = (Fun)*((int*)*(int*)((int*)&d+1)+1);
69          pFun = (Fun)pVtab[1][1];
70          pFun();
71
72          pFun = (Fun)pVtab[1][2];
73          pFun();
74
75          //The tail of the vtable
76          pFun = (Fun)pVtab[1][3];
77          cout<<pFun<<endl;
78
79          //Base3's vtable
80          //pFun = (Fun)*((int*)*(int*)((int*)&d+1)+0);
81          pFun = (Fun)pVtab[2][0];
82          pFun();
83
84          //pFun = (Fun)*((int*)*(int*)((int*)&d+1)+1);
85          pFun = (Fun)pVtab[2][1];
86          pFun();
```

```
87
88          pFun = (Fun)pVtab[2][2];
89          pFun();
90
91          //The tail of the vtable
92          pFun = (Fun)pVtab[2][3];
93          cout<<pFun<<endl;
94
95          return 0;
96}
```

注：本文年代久远，所有的示例都是在 32 位机上跑的。

虚函数有什么缺点或者析构函数声明为虚函数有什么缺点？

大体原因如下：如果某个类不包含虚函数，那一般是表示它将不作为一个基类来使用。当一个类不准备作为基类使用时，使**析构函数为虚**是个坏主意。因为它会为类增加一个虚函数表，对象增加一个虚指针，使得对象的体积增大。

所以基本的一条是：无故的声明虚析构函数和永远不去声明一样是错误的。实际上，很多人这样总结：当且仅当类里包含至少一个虚函数的时候才去声明虚析构函数。

```
class Point{

public:
```

```
    Point(int xCoord, int yCoord){//省略...}

    ~Point(){}

private:

    int x,y;

}
```

虚函数的实现细节是不重要的。重要的是如果 Point 类包含一个虚函数，这个类型的对象的大小就会增加。

在一个 32 位**架构**中，它们将从 64 位（相当于两个 int）长到 96 位（两个 int 加上 vptr 32）；

在一个 64 位架构中，他们可能从 64 位长到 128 位，因为在这样的架构中指针的大小是 64 位的。

为 Point 加上 vptr 将会使它的大小增长 50-100%！Point 对象不再能塞入一个 64bit 缓存器。而且，Point 对象在 C++ 和其他语言（比如 C）中，看起来不再具有相同的结构，因为其它语言缺乏 vptr 的对应物。因此也就不再可能传入其它语言写成的函数或从其中传出，除非你为 vptr 做出明确的补偿。

STL 里的容器都没有虚析构函数，全部的 STL 容器类型（例如，vector，list，set，tr1::unordered_map）

MFC 中的消息机制没有采用 C++中的虚函数机制，原因是消息太多，虚函数内存开销太大。在 Qt 中也没有采用 C++中的虚函数机制，原因与此相同，其实这里还有更深层次上的原因，大体说来，多态的底层实现机制只有两种：

1. 一种是按照名称查表
2. 一种是按照位置查表

两种方式各有利弊，而 C++的虚函数机制无条件的采用了后者，导致的**问题就是在子类很少覆盖基类函数实现的时候内存开销太大，再加上象界面编程这样子类众多的情况（大概是都不覆盖）**，基本上 C++的虚函数机制就废掉了，于是各家库的编写者就只好自谋生路了，说到底，这确实是 C++语言本身的缺陷。

参考：http://blog.csdn.net/oowgsoo/article/details/1529411

感叹一句，还真的是自谋生路啊，而且是不约而同的，Delphi 就采用了 dynamic 机制，以减小子类的内存开销，而且大多数是不必要的虚函数开销。

在这些方法里：

1. VC++使用一组结构体的宏定义来实现消息映射，从而解决虚函数的缺陷。
2. Qt 的信号槽机制其实就是按照名称查表。
3. Delphi 则通过 dynamic 机制从编译器的角度来解决这个问题。这些方法里，自然是 Delphi 的方法最简洁方便，因为没有多余的宏，没有生成的临时代码。

以前一直不明白为什么说 Delphi 语法优美，现在有 2 点体会，好像是这么回事：

1. C++的宏定义表面上是个好东西，也确实是个好东西，因为可以做无穷无尽的扩展（QT 就是一个典型的例子），但用它就会把源代码搞的很复杂，你完全都不知道编译器/类库会把你的源代码替换成什么东西。VC++也是靠这套方法实现消息机制的，QT 更夸张，每个文件都要生成 moc 文件才能真正进行编译。而 Delphi 的源代码就是源代码，对所有人/所有库都平等，没有什么隐晦的代码。
2. Delphi 的编译器做了很多事情，Delphi 的库也提前把许多事情都做好了（比如 TPageControl 如果要在 VC++和 QT 中实现，使用起来烦不胜烦，谁用谁知道），使之手敲的代码看起来尽可能的少。

一、虚函数的工作原理

虚函数的实现要求对象携带额外的信息，这些信息用于在运行时确定该对象应该调用哪一个虚函数。典型情况下，这一信息具有一种被称为 vptr（virtual table pointer，虚函数表指针）的指针的形式。vptr 指向一个被称为 vtbl（virtual table，虚函数表）的函数指针数组，每一个包含虚函数的类都关联到 vtbl。当一个对象调用了虚函数，实际的被调用函数通过下面的步骤确定：找到对象的 vptr 指向的 vtbl，然后在 vtbl 中寻找合适的函数指针。

虚拟函数的地址翻译取决于对象的内存地址，而不取决于数据类型(编译器对函数调用的合法性检查取决于数据类型)。如果类定义了虚函数，该类及其派生类就要生成一张虚拟函数表，即 vtable。而在类的对象地址空间中存储一个该虚表的入口，占 4 个字节，这个

入口地址是在构造对象时由编译器写入的。所以，由于对象的内存空间包含了虚表入口，编译器能够由这个入口找到恰当的虚函数，这个函数的地址不再由数据类型决定了。故对于一个父类的对象指针，调用虚拟函数，如果给他赋父类对象的指针，那么他就调用父类中的函数，如果给他赋子类对象的指针，他就调用子类中的函数(取决于对象的内存地址)。

虚函数需要注意的大概就是这些个地方了，之前在 More effective C++上好像也有见过，不过这次在 Visual C++权威剖析这本书中有了更直白的认识，这本书名字很牛逼，看看内容也就那么回事，感觉名不副实，不过说起来也是有其独到之处的，否则也没必要出这种书了。

每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就会为这个类创建一个虚函数表（VTABLE）保存该类所有虚函数的地址，其实这个 VTABLE 的作用就是保存自己类中所有虚函数的地址，可以把 VTABLE 形象地看成一个函数指针数组，这个数组的每个元素存放的就是虚函数的地址。在每个带有虚函数的类 中，编译器秘密地置入一指针，称为ｖｐｏｉｎｔｅｒ（缩写为ＶＰＴＲ），指向这个对象的ＶＴＡＢＬＥ。 当构造该派生类对象时，其成员 VPTR 被初始化指向该派生类的 VTABLE。所以可以认为 VTABLE 是该类的所有对象共有的，在定义该类时被初始化；而 VPTR 则是每个类对象都有独立一份的，且在该类对象被构造时被初始化。

通过基类指针做虚函数调 用时（也就是做多态调用时），编译器静态地插入取得这个ＶＰＴＲ，并在ＶＴＡＢＬＥ表中查找函数地址的代码，这样就能调用正确的函数使晚捆绑发生。为每个类设置ＶＴＡＢＬＥ、初始化ＶＰＴＲ、为虚函数调用插入代码，所有这些都是自动发生的，所以我们不必担心这些。

[cpp] view plain copy

```cpp
#include<iostream>
using namespace std;

class A
{
public:
    virtual void fun1()
    {
        cout << "A::fun1()" << endl;
    }
    virtual void fun2()
    {
        cout << "A::fun2()" << endl;
    }
};

class B : public A
{
public:
    void fun1()
    {
        cout << "B::fun1()" << endl;
```

```
    }
    void fun2()
    {
        cout << "B::fun2()" << endl;
    }
};

int main()
{
    A *pa = new B;
    pa->fun1();
    delete pa;

    system("pause");
    return 0;
}
```

毫无疑问，调用了 B::fun1()，但是 B::fun1()不是像普通函数那样直接找到函数地址而执行的。真正的执行方式是：首先取出 pa 指针所指向的对象的 vptr 的值，这个值就是 vtbl 的地址，由于调用的函数 B::fun1()是第一个虚函数，所以取出 vtbl 第一个表项里的值，这个值就是 B::fun1()的地址了，最后调用这个函数。因此只要 vptr 不同，指向的 vtbl 就不同，而不同的 vtbl 里装着对应类的虚函数地址，所以这样虚函数就可以完成它的任务，多态就是这样实现的。

而对于 class A 和 class B 来说，他们的 vptr 指针存放在何处？其实这个指针就放在他们各自的实例对象里。由于 class A 和 class B 都没有数据成员，所以他们的实例对象里就只有一个 vptr 指针。

虚拟函数使用的缺点

优点讲了一大堆，现在谈一下缺点，虚函数最主要的缺点是执行效率较低，看一看虚拟函数引发的多态性的实现过程，你就能体会到其中的原因，另外就是由于要携带额外的信息（VPTR），所以导致类多占的内存空间也会比较大，对象也是一样的。

含有虚函数的对象在内存中的结构如下：

[cpp] view plain copy

```
class A
{
private:
    int a;
    int b;
public:
    virtual void fun0()
    {
        cout<<"A::fun0"<<endl;
    }
};
```

1、直接继承

那我们来看看编译器是怎么建立 VPTR 指向的这个虚函数表的，先看下面两个类：

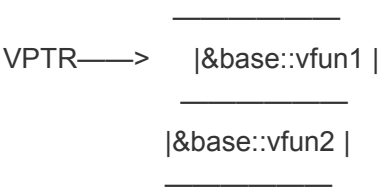
[cpp] view plain copy

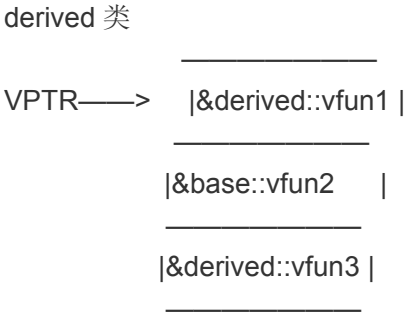
```cpp
class base
{
private:
    int a;
public:
    void bfun()
    {
    }
    virtual void vfun1()
    {
    }
    virtual void vfun2()
    {
    }
};

class derived : public base
{
private:
    int b;
public:
    void dfun()
    {
    }
    virtual void vfun1()
    {
    }
    virtual void vfun3()
    {
    }
};
```

两个类的 VPTR 指向的虚函数表（VTABLE）分别如下：

base 类





每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就为这个类创建一个 VTABLE，如上图所示。在这个表中，编译器放置了在这个类中或在它的基类中所有已声明为 virtual 的函数的地址。如果在这个派生类中没有对在基类中声明为 virtual 的函数进行重新定义，编译器就使用基类 的这个虚函数地址。（在 derived 的 VTABLE 中，vfun2 的入口就是这种情况。）然后编译器在这个类中放置 VPTR。**当使用简单继承时，对于每个对象只有一个 VPTR**。VPTR 必须被初始化为指向相应的 VTABLE，这在构造函数中发生。

一旦 VPTR 被初始化为指向相应的 VTABLE，对象就"知道"它自己是什么类型。但只有当虚函数被调用时这种自我认知才有用。

没有虚函数类对象的大小正好是数据成员的大小，包含有一个或者多个虚函数的类对象编译器向里面插入了一个 VPTR 指针(void *)，指向一个存放函数地址的表就是我们上面说的 VTABLE，这些都是编译器为我们做的我们完全可以不关心这些。所以有虚函数的类对象的大小是数据成员的大小加上一个 VPTR 指针(void *)的大小。

总结一下 VPTR 和 VTABLE 和类对象的关系:

每一个具有虚函数的类都有一个虚函数表 VTABLE，里面按在类中声明的虚函数的顺序存放着虚函数的地址，这个虚函数表 VTABLE 是这个类的所有对象所共有的，也就是说无论用户声明了多少个类对象，但是这个 VTABLE 虚函数表只有一个。

在每个具有虚函数的类的对象里面都有一个 VPTR 虚函数指针，这个指针指向 VTABLE 的首地址，每个类的对象都有这么一种指针。

2、虚继承

这个是比较不好理解的，对于虚继承，若派生类有自己的虚函数，则它本身需要有一个虚指针，指向自己的虚表。另外，派生类虚继承父类时，首先要通过加入一个虚指针来指向父类，因此有可能会有两个虚指针。

二、(虚)继承类的内存占用大小

首先，平时所声明的类只是一种类型定义，它本身是没有大小可言的。 因此，如果用 sizeof 运算符对一个类型名操作，那得到的是具有该类型实体的大小。

计算一个类对象的大小时的规律：

1、空类、单一继承的空类、多重继承的空类所占空间大小为：1（字节，下同）；

2、一个类中，虚函数本身、成员函数（包括静态与非静态）和静态数据成员都是不占用类对象的存储空间的；

3、因此一个对象的大小≥所有非静态成员大小的总和；

4、当类中声明了虚函数（不管是 1 个还是多个），那么在实例化对象时，编译器会自动

在对象里安插一个指针 vPtr 指向虚函数表 VTable；

5、虚承继的情况：由于涉及到虚函数表和虚基表，会同时增加一个（多重虚继承下对应多个）vfPtr 指针指向虚函数表 vfTable 和一个 vbPtr 指针指向虚基表 vbTable，这两者所占的空间大小为：8（或 8 乘以多继承时父类的个数）；

6、在考虑以上内容所占空间的大小时，还要注意编译器下的“补齐”padding 的影响，即编译器会插入多余的字节补齐；

7、类对象的大小=各非静态数据成员（包括父类的非静态数据成员但都不包括所有的成员函数）的总和+ vfptr 指针(多继承下可能不止一个)+vbptr 指针(多继承下可能不止一个)+编译器额外增加的字节。

示例一：含有普通继承

[cpp] view plain copy

```cpp
class A
{
};

class B
{
    char ch;
    virtual void func0()  {  }
};

class C
{
    char ch1;
    char ch2;
    virtual void func()  {  }
    virtual void func1()  {  }
};

class D: public A, public C
{
    int d;
    virtual void func()  {  }
    virtual void func1()  {  }
};

class E: public B, public C
{
    int e;
    virtual void func0()  {  }
    virtual void func1()  {  }
};
```

```cpp
int main(void)
{
    cout<<"A="<<sizeof(A)<<endl;    //result=1
    cout<<"B="<<sizeof(B)<<endl;    //result=8
    cout<<"C="<<sizeof(C)<<endl;    //result=8
    cout<<"D="<<sizeof(D)<<endl;    //result=12
    cout<<"E="<<sizeof(E)<<endl;    //result=20
    return 0;
}
```

前面三个 A、B、C 类的内存占用空间大小就不需要解释了，注意一下内存对齐就可以理解了。

求 sizeof(D)的时候，需要明白，首先 VPTR 指向的虚函数表中保存的是类 D 中的两个虚函数的地址，然后存放基类 C 中的两个数据成员 ch1、ch2，注意内存对齐，然后存放数据成员 d，这样 4+4+4=12。

求 sizeof(E)的时候，首先是类 B 的虚函数地址，然后类 B 中的数据成员，再然后是类 C 的虚函数地址，然后类 C 中的数据成员，最后是类 E 中的数据成员 e，同样注意内存对齐，这样 4+4+4+4+4=20。

示例二：含有虚继承

[cpp] view plain copy

```cpp
class CommonBase
{
    int co;
};

class Base1: virtual public CommonBase
{
public:
    virtual void print1() {  }
    virtual void print2() {  }
private:
    int b1;
};

class Base2: virtual public CommonBase
{
public:
    virtual void dump1() {  }
    virtual void dump2() {  }
private:
    int b2;
};

class Derived: public Base1, public Base2
```

```cpp
{
public:
    void print2() {  }
    void dump2() {  }
private:
    int d;
};
```

sizeof(Derived)=32，其在内存中分布的情况如下：

[cpp] view plain copy

```
class Derived size(32):
      +---
      | +--- (base class Base1)
  | | {vfptr}
  | | {vbptr}
  | | b1
      | +---
      | +--- (base class Base2)
  | | {vfptr}
  | | {vbptr}
  | | b2
      | +---
  | d
      +---
      +--- (virtual base CommonBase)
  | co
      +---
```

示例 3：

[cpp] view plain copy

```cpp
class A
{
public:
    virtual void aa() {  }
    virtual void aa2() {  }
private:
    char ch[3];
};

class B: virtual public A
{
public:
    virtual void bb() {  }
```

```
14.     virtual void bb2() {  }
15. };
16.
17. int main(void)
18. {
19.     cout<<"A's size is "<<sizeof(A)<<endl;
20.     cout<<"B's size is "<<sizeof(B)<<endl;
21.     return 0;
22. }
```

执行结果：A's size is 8
B's size is 16

说明：对于虚继承，类 B 因为有自己的虚函数，所以它本身有一个虚指针，指向自己的虚表。另外，类 B 虚继承类 A 时，首先要通过加入一个虚指针来指向父类 A，然后还要包含父类 A 的所有内容。因此是 4+4+8=16。

两种多态实现机制及其优缺点

除了 c++的这种多态的实现机制之外，还有另外一种实现机制，也是查表，不过是按名称查表，是 smalltalk 等语言的实现机制。这两种方法的优缺点如下：

（1）、按照绝对位置查表，这种方法由于编译阶段已经做好了索引和表项(如上面的 call *(pa->vptr[1]))，所以运行速度比较快；缺点是：当 A 的 virtual 成员比较多（比如 1000 个），而 B 重写的成员比较少（比如 2 个），这种时候，B 的 vtable B 的剩下的 998 个表项都是放 A 中的 v i r t u a l 成员函数的指针，如果这个派生体系比较大的时候，就浪费了很多的空间。

比如：G U I 库，以M F C 库为例，M F C 有很多类，都是一个继承体系；而且很多时候每个类只是 1 ，2 个成员函数需要在派生类重写，如果用C ＋＋的虚函数机制，每个类有一个虚表，每个表里面有大量的重复，就会造成空间利用率不高。于是M F C 的消息映射机制不用虚函数，而用第二种方法来实现多态，那就是：

（ 2 ）、按照函数名称查表，这种方案可以避免如上的问题；但是由于要比较名称，有时候要遍历所有的继承结构，时间效率性能不是很高。（关于M F C 的消息映射的实现，看下一篇文章）

3 、总结：

如果继承体系的基类的 virtual 成员不多，而且在派生类要重写的部分占了其中的大多数时候，用C ＋＋的虚函数机制是比较好的；

但是如果继承体系的基类的 virtual 成员很多，或者是继承体系比较庞大的时候，而且派生类中需要重写的部分比较少，那就用名称查找表，这样效率会高一些，很多的G U I 库都是这样的，比如M F C ， Q T 。

P S ：其实，自从计算机出现之后，时间和空间就成了永恒的主题，因为两者在 98%的情况下都无法协调，此长彼消；这个就是计算机科学中的根本瓶颈之所在。软件科学和算法的发展，就看能不能突破这对时空权衡了。呵呵。。

何止计算机科学如此，整个宇宙又何尝不是如此呢？最基本的宇宙之谜，还是时间和空间。

C++如何不用虚函数实现多态

可以考虑使用函数指针来实现多态

[cpp] view plain copy

```cpp
#include<iostream>
using namespace std;

typedef void (*fVoid)();

class A
{
public:
    static void test()
    {
        printf("hello A\n");
    }

    fVoid print;

    A()
    {
        print = A::test;
    }
};

class B : public A
{
public:
    static void test()
    {
        printf("hello B\n");
    }

    B()
    {
        print = B::test;
    }
};


int main(void)
{
    A aa;
    aa.print();

    B b;
```

```
43.     A* a = &b;
44.     a->print();
45.
46.     return 0;
47. }
```

这样做的好处主要是绕过了 vtable。我们都知道虚函数表有时候会带来一些性能损失。

模板编程就是泛型编程。就好像虚函数编程就是面向对象编程一样。于是，只要程序中用到了模板声明和定义，就是属于泛型编程了。然而从广义上说，泛型编程更讲究目的：模板设计的目的就是为了能得到多种类型的有效通用。

虚函数表（virtual table）vtbls 指向虚函数表的指针（virtual table pointer）vptrs。

volatile 的语法和 const 是一样的，但是 volatile 的意思是“在编译器认识的范围外，这个数据是可以被改变”，不知何故，环境正在改变数据（可能通过多任务，多线程或者中断处理）所以，volatile 告诉编译器不要善自作出有关数据的任何假定。

如果编译器说："我已经报数据读进寄存器，而且再没有与寄存器接触"，一般情况下，它不需要再读这个数据，但是，如果数据是 volatile 修饰的，编译器就不能作出这样的假定，因为这个数据可能被其他进程改变了，它**必须重读这数据**而不是优化这个代码来消除通常情况下那些冗余的读写操作代码

C++中虚拟继承的概念

为了解决从不同途径继承来的同名的数据成员在内存中有不同的拷贝造成数据不一致问题，将共同基类设置为虚基类。这时从不同的路径继承过来的同名数据成员在内存中就只有一个拷贝，同一个函数名也只有一个映射。这样不仅就解决了二义性问题，也节省了内存，避免了数据不一致的问题。

class 派生类名：virtual 继承方式 基类名

virtual 是关键字，声明该基类为派生类的虚基类。

在多继承情况下，虚基类关键字的作用范围和继承方式关键字相同，只对紧跟其后的基类起作用。

声明了虚基类之后，虚基类在进一步派生过程中始终和派生类一起，维护同一个基类子对象的拷贝。

C++虚拟继承

◇概念：

C++使用虚拟继承（Virtual Inheritance），解决从不同途径继承来的同名的数据成员在内存中有不同的拷贝造成数据不一致问题，将共同基类设置为虚基类。这时从不同的路径继承过来的同名数据成员在内存中就只有一个拷贝，同一个函数名也只有一个映射。

◇解决问题：

解决了二义性问题，也节省了内存，避免了数据不一致的问题。

◇同义词：

虚基类（把一个动词当成一个名词而已）

当在多条继承路径上有一个公共的基类，在这些路径中的某几条汇合处，这个公共的基类就会产生多个实例(或多个副本)，若只想保存这个基类的一个实例，可以将这个公共基类说明为虚基类。

◇语法：

class 派生类: virtual 基类 1，virtual 基类 2，...，virtual 基类 n

{

...//派生类成员声明

};

◇执行顺序

首先执行虚基类的构造函数，多个虚基类的构造函数按照被继承的顺序构造；

执行基类的构造函数，多个基类的构造函数按照被继承的顺序构造；

执行成员对象的构造函数，多个成员对象的构造函数按照申明的顺序构造；

执行派生类自己的构造函数；

析构以与构造相反的顺序执行；

mark

从虚基类直接或间接派生的派生类中的构造函数的成员初始化列表中都要列出对虚基类构造函数的调用。但只有用于建立对象的最派生类的构造函数调用虚基类的构造函数，而该派生类的所有基类中列出的对虚基类的构造函数的调用在执行中被忽略，从而保证对虚基类子对象只初始化一次。

在一个成员初始化列表中同时出现对虚基类和非虚基类构造函数的调用时，虚基类的构造函数先于非虚基类的构造函数执行。

◇因果：

多重继承->二义性->虚拟继承解决

◇二义性：

```
1: //-----------------------------------------------------
2: //名称：blog_virtual_inherit.cpp
3: //说明：C++虚拟继承学习演示
4: //环境：VS2005
5: //blog：pppboy.blog.163.com
6: //-----------------------------------------------------
7: #include "stdafx.h"
8: #include <iostream>
9: using namespace std;
10:
11: //Base
12: class Base
13: {
14: public:
15: Base(){cout << "Base called..."<< endl;}
16: void print(){cout << "Base print..." <<endl;}
17: private:
18: };
19:
20: //Sub
21: class Sub //定义一个类 Sub
22: {
23: public:
24: Sub(){cout << "Sub called..." << endl;}
25: void print(){cout << "Sub print..." << endl;}
26: private:
27: };
28:
29: //Child
30: class Child : public Base , public Sub //定义一个类 Child 分别继承
自 Base ， Sub
31: {
32: public:
```

```
33: Child(){cout << "Child called..." << endl;}
34: private:
35: };
36:
37: int main(int argc, char* argv[])
38: {
39: Child c;
40:
41: //不能这样使用，会产生二意性，VC 下 error C2385
42: //c.print();
43:
44: //只能这样使用
45: c.Base::print();
46: c.Sub::print();
47:
48: system("pause");
49: return 0;
50: }
```

◇多重继承：

```
1: //-----------------------------------------------------
2: //名称：blog_virtual_inherit.cpp

3: //说明：C++虚拟继承学习演示
4: //环境：VS2005
5: //blog：pppboy.blog.163.com
6: //-----------------------------------------------------
7: #include "stdafx.h"
8: #include <iostream>
9: using namespace std;
10:
11: int gFlag = 0;
12:
13: class Base
```

```
14: {
15: public:
16: Base(){cout << "Base called : " << gFlag++ << endl;}
17: void print(){cout << "Base print" <<endl;}
18: };
19:
20: class Mid1 : public Base
21: {
22: public:
23: Mid1(){cout << "Mid1 called" << endl;}
24: private:
25: };
26:
27: class Mid2 : public Base
28: {
29: public:
30: Mid2(){cout << "Mid2 called" << endl;}
31: };
32:
33: class Child:public Mid1, public Mid2
34: {
35: public:
36: Child(){cout << "Child called" << endl;}
37: };
38:
39: int main(int argc, char* argv[])
40: {
41: Child d;
42:
43: //不能这样使用，会产生二意性
   //d.print();
45:
46: //只能这样使用
47: d.Mid1::print();
48: d.Mid2::print();
```

```
49:
50: system("pause");
51: return 0;
52: }
53:
```

//output

```
Base called : 0
Mid1 called
Base called : 1
Mid2 called
Child called
Base print
Base print
```

◇虚拟继承

在派生类继承基类时，加上一个 virtual 关键词则为虚拟继承

```
1: //-----------------------------------------------------
2: //名称：blog_virtual_inherit.cpp

3: //说明：C++虚拟继承学习演示
4: //环境：VS2005
5: //blog：pppboy.blog.163.com
6: //-----------------------------------------------------
7: #include "stdafx.h"
8: #include <iostream>
9: using namespace std;
10:
11: int gFlag = 0;
12:
13: class Base
14: {
15: public:
16: Base(){cout << "Base called : " << gFlag++ << endl;}
```

```
17: void print(){cout << "Base print" <<endl;}
18: };
19:
20: class Mid1 : virtual public Base
21: {
22: public:
23: Mid1(){cout << "Mid1 called" << endl;}
24: private:
25: };
26:
27: class Mid2 : virtual public Base
28: {
29: public:
30: Mid2(){cout << "Mid2 called" << endl;}
31: };
32:
33: class Child:public Mid1, public Mid2
34: {
35: public:
36: Child(){cout << "Child called" << endl;}
37: };
38:
39: int main(int argc, char* argv[])
40: {
41: Child d;
42:
43: //这里可以这样使用
44: d.print();
45:
46: //也可以这样使用
47: d.Mid1::print();
48: d.Mid2::print();
49:
50: system("pause");
51: return 0;
```

```
52: }
53:
```

//output

```
1: Base called : 0
2: Mid1 called
3: Mid2 called
4: Child called
5: Base print
6: Base print
7: Base print
8: 请按任意键继续. . .
```

◇通过输出的比较

1. 在多继承情况下，虚基类关键字的作用范围和继承方式关键字相同，只对紧跟其后的基类起作用。

2. 声明了虚基类之后，虚基类在进一步派生过程中始终和派生类一起，维护同一个基类子对象的拷贝。

3. 观察类构造函数的构造顺序，拷贝也只有一份。

◇与虚函数关系

虚拟继承与虚函数有一定相似的地方，但他们之间是绝对没有任何联系的。

再想一次：虚拟继承，虚基类，虚函数。

1、虚基类的作用从上面的介绍可知:如果一个派生类有多个直接基类，而这些直接基类又有一个共同的基类，则在最终的派生类中会保留该间接共同基类数据成员的多份同名成员。

在引用这些同名的成员时，必须在派生类对象名后增加直接基类名，以避免产生二义性，使其惟一地标识一个成员，如

c1.A::display( )。

在一个类中保留间接共同基类的多份同名成员，这种现象是人们不希望出现的。C++提供虚基类(virtual base class
)的方法，使得在继承间接共同基类时只保留一份成员。

现在，将类 A 声明为虚基类，方法如下:

```
class
A//声明基类 A
{…};
class B :virtual public
A//声明类 B 是类 A 的公用派生类，A 是 B 的虚基类
{…};
class C :virtual public
A//声明类 C 是类 A 的公用派生类，A 是 C 的虚基类
{…};
```

注意：
虚基类并不是在声明基类时声明的，而是在声明派生类时，指定继承方式时声明的。因为一个基类可以在生成一个派生类时作为虚基类，而在生成另一个派生类时不作为虚基类。

声明虚基类的一般形式为
class 派生类名: virtual 继承方式
基类名
经过这样的声明后，当基类通过多条派生路径被一个派生类继承时，该派生类只继承该基类一次。

需要注意：为了保证虚基类在派生类中只继承一次，应当在该基类的所有直接派生类中声明为虚基类。否则仍然会出现对基类的多次继承。

如果在派生类 B 和 C 中将类 A 声明为虚基类，而在派生类 D 中没有将类 A 声明为虚基类，则在派生类 E 中，虽然从类 B 和 C 路径派生的部分只保留一份基类成员，但从类 D 路径派生的部分还保留一份基类成员。

2、虚基类的初始化如果在虚基类中定义了带参数的构造函数，而且没有定义默认构造函数，则在其所有派生类(包括直接派生或间接派生的派生类)中，通过构造函数的初始化表对虚基类进行初始化。例如

```
class
A//定义基类 A
{
   A(int i){ } //基类构造函数，有一个参数};
class B :virtual public A
//A 作为 B 的虚基类
{
   B(int n):A(n){ } //B 类构造函数，在初始化表中对虚基类初始化
};
class C
:virtual public A //A 作为 C 的虚基类
{
   C(int n):A(n){ }
//C 类构造函数，在初始化表中对虚基类初始化
```

```
};
class D :public B,public C
//类 D 的构造函数，在初始化表中对所有基类初始化
{
   D(int n):A(n),B(n),C(n){ }
};
```

注意：

在定义类 D 的构造函数时，与以往使用的方法有所不同。规定：

在最后的派生类中不仅要负责对其直接基类进行初始化，还要负责对虚基类初始化。C++编译系统只执行最后的派生类对虚基类的构造函数的调用，而忽略虚基类的其他派生类(如类 B 和类 C)

对虚基类的构造函数的调用，这就保证了虚基类的数据成员不会被多次初始化。

温馨提示：使用多重继承时要十分小心，经常会出现二义性问题。许多专业人员认为：不要提倡在程序中使用多重继承，只有在比较简单和不易出现二义性的情况或实在必要时才使用多重继承，能用单一继承解决的问题就不要使用多重继承。也是由于这个原因，有些面向对象的程序设计语言(如 Java，Smalltalk)并不支持多重继承。

虚继承与虚基类的本质

虚继承和虚基类的定义是非常的简单的，同时也是非常容易判断一个继承是否是虚继承的，虽然这两个概念的定义是非常的简单明确的，但是在 C++语言中虚继承作为一个比较生

僻的但是又是绝对必要的组成部份而存在着，并且其行为和模型均表现出和一般的继承体系之间的巨大的差异（包括访问性能上的差异），现在我们就来彻底的从语言、模型、性能和应用等多个方面对虚继承和虚基类进行研究。

首先还是先给出虚继承和虚基类的定义。

虚继承：在继承定义中包含了 virtual 关键字的继承关系；

虚基类：在虚继承体系中的通过 virtual 继承而来的基类，需要注意的是：

struct CSubClass : public virtual CBase {}; 其中 CBase 称之为 CSubClass 的虚基类，而不是说 CBase 就是个虚基类，因为 CBase 还可以不不是虚继承体系中的基类。

有了上面的定义后，就可以开始虚继承和虚基类的本质研究了，下面按照语法、语义、模型、性能和应用五个方面进行全面的描述。

1. 语法

语法有语言的本身的定义所决定，总体上来说非常的简单，如下：

struct CSubClass : public virtual CBaseClass {};

其中可以采用 public、protected、private 三种不同的继承关键字进行修饰，只要确保包含 virtual 就可以了，这样一来就形成了虚继承体系，同时 CBaseClass 就成为

了 CSubClass 的虚基类了。

其实并没有那么的简单，如果出现虚继承体系的进一步继承会出现什么样的状况呢？

如下所示：

[cpp] view plain copy

```cpp
/*
 * 带有数据成员的基类
 */
struct CBaseClass1
{
    CBaseClass1( size_t i ) : m_val( i ) {}

    size_t m_val;
};
/*
 * 虚拟继承体系
 */
struct CSubClassV1 : public virtual CBaseClass1
{
    CSubClassV1( size_t i ) : CBaseClass1( i ) {}
};
struct CSubClassV2 : public virtual CBaseClass1
{
    CSubClassV2( size_t i ) : CBaseClass1( i ) {}
};
struct CDiamondClass1 : public CSubClassV1, public CSubClassV2
{
    CDiamondClass1( size_t i ) : CBaseClass1( i ), CSubClassV1( i ), CSubClassV2( i ) {}
};
struct CDiamondSubClass1 : public CDiamondClass1
{
    CDiamondSubClass1( size_t i ) : CBaseClass1( i ), CDiamondClass1( i ) {}
};
```

注意上面代码中的CDiamondClass1和CDiamondSubClass1两个类的构造函数初始化列

表中的内容。可以发现其中均包含了虚基类CBaseClass1的初始化工作，如果没有这

个初始化语句就会导致编译时错误，为什么会这样呢？一般情况下不是只要在

CSubClassV1 和 CSubClassV2 中包含初始化就可以了么？要解释该问题必须要明白虚
继承的语义特征，所以参看下面语义部分的解释。

2. 语义

从语义上来讲什么是虚继承和虚基类呢？上面仅仅是从如何在 C++语言中书写合法的
虚继承类定义而已。首先来了解一下 virtual 这个关键字在 C++中的公共含义，在 C++
语言中仅仅有两个地方可以使用 virtual 这个关键字，一个就是类成员虚函数和这里
所讨论的虚继承。不要看这两种应用场合好像没什么关系，其实他们在背景语义上
具有 virtual 这个词所代表的共同的含义，所以才会在这两种场合使用相同的关键字。
那么 virtual 这个词的含义是什么呢？
virtual 在《美国传统词典[双解]》中是这样定义的：

adj.（形容词）
1. Existing or resulting in essence or effect though not in actual
fact, form, or name:
实质上的，实际上的：虽然没有实际的事实、形式或名义，但在实际上或效
果上存在或产生的；
2. Existing in the mind, especially as a product of the imagination.
Used in literary criticism of text.
虚的，内心的：在头脑中存在的，尤指意想的产物。用于文学批评中。

我们采用第一个定义，也就是说被 virtual 所修饰的事物或现象在本质上是存在的，
但是没有直观的形式表现，无法直接描述或定义，需要通过其他的间接方式或手段
才能够体现出其实际上的效果。
那么在 C++中就是采用了这个词意，不可以在语言模型中直接调用或体现的，但是确
实是存在可以被间接的方式进行调用或体现的。比如：虚函数必须要通过一种间接的
运行时（而不是编译时）机制才能够激活（调用）的函数，而虚继承也是必须在运行
时才能够进行定位访问的一种体制。存在，但间接。其中关键就在于存在、间接和共
享这三种特征。
对于虚函数而言，这三个特征是很好理解的，间接性表明了他必

须在运行时根据实际

的对象来完成函数寻址，共享性表象在基类会共享被子类重载后的虚函数，其实指向

相同的函数入口。

对于虚继承而言，这三个特征如何理解呢？存在即表示虚继承体系和虚基类确实存在，

间接性表明了在访问虚基类的成员时同样也必须通过某种间接机制来完成（下面模型

中会讲到），共享性表象在虚基类会在虚继承体系中被共享，而不会出现多份拷贝。

那现在可以解释语法小节中留下来的那个问题了，“为什么一旦出现了虚基类，就必

须在没有一个（这里疑似笔误，应为“每一个”）继承类中都必须包含虚基类的初始化语句”。由上面的分析可以知道，

虚基类是被共享的，也就是在继承体系中无论被继承多少次，对象内存模型中均只会

出现一个虚基类的子对象（这和多继承是完全不同的），这样一来既然是共享的那么

每一个子类都不会独占，但是总还是必须要有一个类来完成基类的初始化过程（因为

所有的对象都必须被初始化，哪怕是默认的），同时还不能够重复进行初始化，那到

底谁应该负责完成初始化呢？C++标准中（也是很自然的）选择在每一次继承子类中

都必须书写初始化语句（因为每一次继承子类可能都会用来定义对象），而在最下层

继承子类中实际执行初始化过程。所以上面在每一个继承类中都要书写初始化语句，

但是在创建对象时，而仅仅会在创建对象用的类构造函数中实际的执行初始化语句，

其他的初始化语句都会被压制不调用。

```
[cpp] view plain copy
```

1. <作者的意思是：一个继承体系中，A->B->C->D,构造一个 D 对象的话，A 在内存中只有一份真实的存在，而不是 4 份，这就存在着谁来初始化的问题。当然是 D，因为生成的对象是 D 的对象！>

3. 模型

为了实现上面所说的三种语义含义，在考虑对象的实现模型（也就是内存模型）时就很自然了。在 C++中对象实际上就是一个连续的地址空间的语义代表，我们来分析虚继承下的内存模型。

3.1. 存在

也就是说在对象内存中必须要包含虚基类的完整子对象，以便能够完成通过地址完成对象的标识。那么至于虚基类的子对象会存放在对象的那个位置（头、中间、尾部）则由各个编译器选择，没有差别。（在 VC8 中无论虚基类被声明在什么位置，虚基类的子对象都会被放置在对象内存的尾部）

3.2. 间接

间接性表明了在直接虚基承子类中一定包含了某种指针（偏移或表格）来完成通过子类访问虚基类子对象（或成员）的间接手段（因为虚基类子对象是共享的，没有确定关系），至于采用何种手段由编译器选择。（在 VC8 中在子类中放置了一个虚基类指针 vbc，该指针指向虚函数表中的一个 slot，该 slot 中存放则虚基类子对象的偏移量的负值，实际上就是个以补码表示的 int 类型的值，在计算虚基类子对象首地址时，需要将该偏移量取绝对值相加，这个主要是为了和虚表中只能存放虚函数地址这一要求相区别，因为地址是原码表示的无符号 int 类型的值）

3.3. 共享

共享表明了在对象的内存空间中仅仅能够包含一份虚基类的子对象，并且通过某种间接的机制来完成共享的引用关系。在介绍完整个内容后会附上测试代码，体现这些内容。

4. 性能

由于有了间接性和共享性两个特征，所以决定了虚继承体系下的对象在访问时必然会在时间和空间上与一般情况有较大不同。

4.1. 时间

在通过继承类对象访问虚基类对象中的成员（包括数据成员和函数成员）时，都必须通过某种间接引用来完成，这样会增加引用寻址时间（就和虚函数一样），其实就是调整 this 指针以指向虚基类对象，只不过这个调整是运行时间接完成的。（在 VC8 中通过打开汇编输出，可以查看*.cod 文件中的内容，在访问虚基类对象成员时会形成三条 mov 间接寻址语句，而在访问一般继承类对象时仅仅只有一条 mov

常量直接寻址语句）

4.2. 空间

由于共享所以不同在对象内存中保存多份虚基类子对象的拷贝，这样较之多继承节省空间。

5. 应用

谈了那么多语言特性和内容，那么在什么情况下需要使用虚继承，而一般应该如何使用呢？

这个问题其实很难有答案，一般情况下如果你确性出现多继承没有必要，必须要共享基类子对象的时候可以考虑采用虚继承关系（C++标准 iOS 体系就是这样的）。由于每

一个继承类都必须包含初始化语句而又仅仅只在最底层子类中调用，这样可能就会使得某些上层子类得到的虚基类子对象的状态不是自己所期望的（因为自己的初始化语句被压制了），所以一般建议不要在虚基类中包含任何数据成员（不要有状态），只可以作为接口类来提供。

附录：测试代码

[c-sharp] view plain copy

```
#include <ctime>
#include <iostream>
/*
 * 带有数据成员的基类
 */
struct CBaseClass1
{
    CBaseClass1( size_t i ) : m_val( i ) {}
    size_t m_val;
};
/*
 * 虚拟继承体系
 */
struct CSubClassV1 : public virtual CBaseClass1
{
    CSubClassV1( size_t i ) : CBaseClass1( i ) {}
};
struct CSubClassV2 : public virtual CBaseClass1
{
    CSubClassV2( size_t i ) : CBaseClass1( i ) {}
};
struct CDiamondClass1 : public CSubClassV1, public CSubClassV2
{
    CDiamondClass1( size_t i ) : CBaseClass1( i ), CSubClassV1( i ), CSubClassV2( i ) {}
};
struct CDiamondSubClass1 : public CDiamondClass1
{
    CDiamondSubClass1( size_t i ) : CBaseClass1( i ), CDiamondClass1( i ) {}
```

```
};
/*
 * 正常继承体系
 */
struct CSubClassN1 : public CBaseClass1
{
    CSubClassN1( size_t i ) : CBaseClass1( i ) {}
};
struct CSubClassN2 : public CBaseClass1
{
    CSubClassN2( size_t i ) : CBaseClass1( i ) {}
};
struct CMultiClass1 : public CSubClassN1, public CSubClassN2
{
    CMultiClass1( size_t i ) : CSubClassN1( i ), CSubClassN2( i ) {}
};
struct CMultiSubClass1 : public CMultiClass1
{
    CMultiSubClass1( size_t i ) : CMultiClass1( i ) {}
};
/*
 * 不带有数据成员的接口基类
 */
struct CBaseClass2
{
    virtual void func() {};
    virtual ~CBaseClass2() {}
};
/*
 * 虚拟继承体系
 */
// struct CBaseClassX { CBaseClassX() {i1 = i2 = 0xFFFFFFFF;} size_t i1, i2;}
;
struct CSubClassV3 : public virtual CBaseClass2
{
};
struct CSubClassV4 : public virtual CBaseClass2
{
};
struct CDiamondClass2 : public CSubClassV3, public CSubClassV4
{
};
struct CDiamondSubClass2 : public CDiamondClass2
{
```

```
72. };
73. /*
74.  * 正常继承体系
75.  */
76. struct CSubClassN3 : public CBaseClass2
77. {
78. };
79. struct CSubClassN4 : public CBaseClass2
80. {
81. };
82. struct CMultiClass2 : public CSubClassN3, public CSubClassN4
83. {
84. };
85. struct CMultiSubClass2 : public CMultiClass2
86. {
87. };
88. /*
89.  * 内存布局用类声明.
90.  */
91. struct CLayoutBase1
92. {
93.     CLayoutBase1() : m_val1( 0 ), m_val2( 1 ) {}
94.     size_t m_val1, m_val2;
95. };
96. struct CLayoutBase2
97. {
98.     CLayoutBase2() : m_val1( 3 ) {}
99.     size_t m_val1;
100. };
101. struct CLayoutSubClass1 : public virtual CBaseClass1, public CLayoutBase1,
    public CLayoutBase2
102. {
103.     CLayoutSubClass1() : CBaseClass1( 2 ) {}
104. };
105.
106. #define MAX_TEST_COUNT 1000 * 1000 * 16
107. #define TIME_ELAPSE() ( std::clock() - start * 1.0 ) / CLOCKS_PER_SEC
108. int main( int argc, char *argv[] )
109. {
110.     /*
111.      * 类体系中的尺寸.
112.      */
113.     std::cout << "================================ sizeof ================
    ===============" << std::endl;
```

```
114.     std::cout << "    ----------------------------------------------------
    -----------" << std::endl;
115.     std::cout << "sizeof( CBaseClass1 )       = " << sizeof( CBaseClass1 )
    << std::endl;
116.     std::cout << std::endl;
117.     std::cout << "sizeof( CSubClassV1 )       = " << sizeof( CSubClassV1 )
    << std::endl;
118.     std::cout << "sizeof( CSubClassV2 )       = " << sizeof( CSubClassV2 )
    << std::endl;
119.     std::cout << "sizeof( CDiamondClass1 )    = " << sizeof( CDiamondClass1
     ) << std::endl;
120.     std::cout << "sizeof( CDiamondSubClass1 ) = " << sizeof( CDiamondSubCla
    ss1 ) << std::endl;
121.     std::cout << std::endl;
122.     std::cout << "sizeof( CSubClassN1 )       = " << sizeof( CSubClassN1 )
    << std::endl;
123.     std::cout << "sizeof( CSubClassN2 )       = " << sizeof( CSubClassN2 )
    << std::endl;
124.     std::cout << "sizeof( CMultiClass1 )      = " << sizeof( CMultiClass1 )
     << std::endl;
125.     std::cout << "sizeof( CMultiSubClass1 )   = " << sizeof( CMultiSubClass
    1 ) << std::endl;
126.     std::cout << "    ----------------------------------------------------
    -----------" << std::endl;
127.     std::cout << "sizeof( CBaseClass2 )       = " << sizeof( CBaseClass2 )
    << std::endl;
128.     std::cout << std::endl;
129.     std::cout << "sizeof( CSubClassV3 )       = " << sizeof( CSubClassV3 )
    << std::endl;
130.     std::cout << "sizeof( CSubClassV4 )       = " << sizeof( CSubClassV4 )
    << std::endl;
131.     std::cout << "sizeof( CDiamondClass2 )    = " << sizeof( CDiamondClass2
     ) << std::endl;
132.     std::cout << "sizeof( CDiamondSubClass2 ) = " << sizeof( CDiamondSubCla
    ss2 ) << std::endl;
133.     std::cout << std::endl;
134.     std::cout << "sizeof( CSubClassN3 )       = " << sizeof( CSubClassN3 )
    << std::endl;
135.     std::cout << "sizeof( CSubClassN4 )       = " << sizeof( CSubClassN4 )
    << std::endl;
136.     std::cout << "sizeof( CMultiClass2 )      = " << sizeof( CMultiClass2 )
     << std::endl;
137.     std::cout << "sizeof( CMultiSubClass2 )   = " << sizeof( CMultiSubClass
    2 ) << std::endl;
```

```
138.     /*
139.      * 对象内存布局
140.      */
141.     std::cout << "================================ layout ================
    ===============" << std::endl;
142.     std::cout << "    --------------------------------MI--------------------
    -----------" << std::endl;
143.     CLayoutSubClass1 *lsc = new CLayoutSubClass1;
144.     std::cout << "sizeof( CLayoutSubClass1 )   = " << sizeof( CLayoutSubCla
    ss1 ) << std::endl;
145.     std::cout << "CLayoutBase1 offset of CLayoutSubClass1 is " << (char*)(C
    LayoutBase1 *)lsc - (char*)lsc << std::endl;
146.     std::cout << "CBaseClass1  offset of CLayoutSubClass1 is " << (char*)(C
    BaseClass1  *)lsc - (char*)lsc << std::endl;
147.     std::cout << "CLayoutBase2 offset of CLayoutSubClass1 is " << (char*)(C
    LayoutBase2 *)lsc - (char*)lsc << std::endl;
148.     int *ptr = (int*)lsc;
149.     std::cout << "vbc in CLayoutSubClass1 is " << *(int*)ptr[3] << std::end
    l;
150.     delete lsc;
151.     std::cout << "    --------------------------------SI--------------------
    -----------" << std::endl;
152.     CSubClassV1 *scv1 = new CSubClassV1( 1 );
153.     std::cout << "sizeof( CSubClassV1 )   = " << sizeof( CSubClassV1 ) << s
    td::endl;
154.     std::cout << "CBaseClass1 offset of CSubClassV1 is " << (char*)(CBaseCl
    ass1 *)scv1 - (char*)scv1 << std::endl;
155.     ptr = (int*)scv1;
156.     std::cout << "vbc in CSubClassV1 is " << *(int*)ptr[0] << std::endl;
157.     delete scv1;
158.     /*
159.      * 性能测试
160.      */
161.     std::cout << "================================ Performance ============
    ====================" << std::endl;
162.     double times[4];
163.     size_t idx = 0;
164.     CSubClassV1 *ptr1 = new CDiamondClass1( 1 );
165.     std::clock_t start = std::clock();
166.     {
167.         for ( size_t i = 0; i < MAX_TEST_COUNT; ++i )
168.             ptr1->m_val = i;
169.     }
170.     times[idx++] = TIME_ELAPSE();
```

```
171.     delete static_cast<CDiamondClass1*>( ptr1 );
172.     CSubClassN1 *ptr2 = new CMultiClass1( 0 );
173.     start = std::clock();
174.     {
175.         for ( size_t i = 0; i < MAX_TEST_COUNT; ++i )
176.             ptr2->m_val = i;
177.     }
178.     times[idx++] = TIME_ELAPSE();
179.     delete static_cast<CMultiClass1*>( ptr2 );
180.     std::cout << "CSubClassV1::ptr1->m_val " << times[0] << " s" << std::en
    dl;
181.     std::cout << "CSubClassN1::ptr2->m_val " << times[1] << " s" << std::en
    dl;
182.     return 0;
183. }
```

测试环境：

软件环境：Visual Studio2005 Pro + SP1, boost1.34.0

硬件环境：PentiumD 3.0GHz, 4G RAM

测试数据：

```
================================ sizeof
================================
        ------------------------------------------------------------------
--
sizeof( CBaseClass1 )             = 4

sizeof( CSubClassV1 )             = 8
sizeof( CSubClassV2 )             = 8
sizeof( CDiamondClass1 )        = 12
sizeof( CDiamondSubClass1 ) = 12

sizeof( CSubClassN1 )             = 4
sizeof( CSubClassN2 )             = 4
sizeof( CMultiClass1 )           = 8
sizeof( CMultiSubClass1 )     = 8
        ------------------------------------------------------------------
--
sizeof( CBaseClass2 )             = 4

sizeof( CSubClassV3 )             = 8
sizeof( CSubClassV4 )             = 8
sizeof( CDiamondClass2 )        = 12
sizeof( CDiamondSubClass2 ) = 12
```

```
sizeof( CSubClassN3 )              = 4
sizeof( CSubClassN4 )              = 4
sizeof( CMultiClass2 )            = 8
sizeof( CMultiSubClass2 )     = 8
================================ layout
================================
       ------------------------------MI--------------------------
--
sizeof( CLayoutSubClass1 )     = 20
CLayoutBase1 offset of CLayoutSubClass1 is 0
CBaseClass1   offset of CLayoutSubClass1 is 16
CLayoutBase2 offset of CLayoutSubClass1 is 8
vbc in CLayoutSubClass1 is -12
       ------------------------------SI--------------------------
--
sizeof( CSubClassV1 )     = 8
CBaseClass1 offset of CSubClassV1 is 4
vbc in CSubClassV1 is 0
================================ Performance
================================
CSubClassV1::ptr1->m_val 0.062 s
CSubClassN1::ptr2->m_val 0.016 s
```

结果分析：

1. 由于虚继承引入的间接性指针所以导致了虚继承类的尺寸会增加 4 个字节；

2. 由 Layout 输出可以看出，虚基类子对象被放在了对象的尾部（偏移为 16），并且 vbc 指针必须紧紧的接在虚基类子对象的前面，所以 vbc 指针所指向的内容为“偏移 - 4”；

3. 由于 VC8 将偏移放在了虚函数表中，所以为了区分函数地址和偏移，所以偏移是用补码 int 表示的负值；

4. 间接性可以通过性能来看出，在虚继承体系同通过指针访问成员时的时间一般是一般类访问情况下的 4 倍左右，符合汇编语言输出文件中的汇编语句的安排。

那么，为什么要使用虚继承？？

为什么要引入虚拟继承？

虚拟继承在一般的应用中很少用到，所以也往往被忽视，这也主要是因为在C++中，多重继承是不推荐的，也并不常用，而一旦离开了多重继承，虚拟继承就完全失去了存在的必要（因为这样只会降低效率和占用更多的空间，关于这一点，我自己还没有太多深刻的理解，有兴趣的可以看网络上白杨的作品《RTTI、虚函数和虚基类的开销分析及使用指导》，说实话我目前还没看得很明白，高人可以指点下我）。

一个例子

以下面的一个例子为例：

[cpp] view plain copy

```cpp
#include <iostream.h>
    #include <memory.h>
    class CA
    {
    int k; //如果基类没有数据成员，则在这里多重继承编译不会出现二义性
    public:
    void f() {cout << "CA::f" << endl;}
    };
    class CB : public CA
    {
    };
    class CC : public CA
    {
    };
    class CD : public CB, public CC
    {
    };
    void main()
    {
    CD d;
    d.f();
```

```
22.     }
```

当编译上述代码时，我们会收到如下的错误提示：

error C2385: 'CD::f' is ambiguous

即编译器无法确定你在 d.f()中要调用的函数 f 到底是哪一个。这里可能会让人觉得有些奇怪，命名只定义了一个 CA::f，既然大家都派生自 CA，那自然就是调用的 CA::f，为什么还无法确定呢？

这是因为编译器在进行编译的时候，需要确定子类的函数定义，如 CA::f 是确定的，那么在编译 CB、CC 时还需要在编译器的语法树中生成 CB::f，CC::f 等标识，那么，在编译 CD 的时候，由于 CB、CC 都有一个函数 f，此时，编译器将试图生成这两个 CD::f 标识，显然这时就要报错了。（当我们不使用 CD::f 的时候，以上标识都不会生成，所以，如果去掉 d.f()一句，程序将顺利通过编译）

要解决这个问题，有两个方法：

1、重载函数 f()：此时由于我们明确定义了 CD::f，编译器检查到 CD::f()调用时就无需再像上面一样去逐级生成 CD::f 标识了；

此时 CD 的元素结构如下：

|CB(CA)|

|CC(CA)|

故此时的 sizeof(CD) = 8;（CB、CC 各有一个元素 k）

2、使用虚拟继承：虚拟继承又称作共享继承，这种共享其实也是编译期间实现的，当使用虚拟继承时，上面的程序将变成下面的形式：

[c-sharp] view plain copy

```
1.  #include <iostream.h>
2.      #include <memory.h>
3.      class CA
4.      {
5.      int k;
6.      public:
7.      void f() {cout << "CA::f" << endl;}
8.      };
9.      class CB : virtual public CA //也有一种写法是
    class CB : public virtual CA
10.     { //实际上这两种方法都可以
11.     };
12.     class CC : virtual public CA
13.     {
14.     };
```

```
15.    class CD : public CB, public CC
16.    {
17.    };
18.    void main()
19.    {
20.    CD d;
21.    d.f();
22.    }
```

此时，当编译器确定 d.f()调用的具体含义时，将生成如下的 CD 结构：

|CB|

|CC|

|CA|

同时，在 CB、CC 中都分别包含了一个指向 CA 的虚基类指针列表 vbptr（virtual base table pointer），其中记录的是从 CB、CC 的元素到 CA 的元素之间的偏移量。此时，不会生成各子类的函数 f 标识，除非子类重载了该函数，从而达到“共享”的目的（这里的具体内存布局，可以参看钻石型继承内存布局，在白杨的那篇文章中也有）。

也正因此，此时的 sizeof(CD) = 12（两个 vbptr + sizoef(int)）；

另注：

如果 CB，CC 中各定义一个 int 型变量，则 sizeof(CD)就变成 20(两个 vbptr + 3 个 sizoef(int)

如果 CA 中添加一个 virtual void f1(){}，sizeof(CD) = 16（两个 vbptr + sizoef(int)+vptr）;

再添加 virtual void f2(){}，sizeof(CD) = 16 不变。原因如下所示：带有虚函数的类，其内存布局上包含一个指向虚函数列表的指针（vptr），这跟有几个虚函数无关。