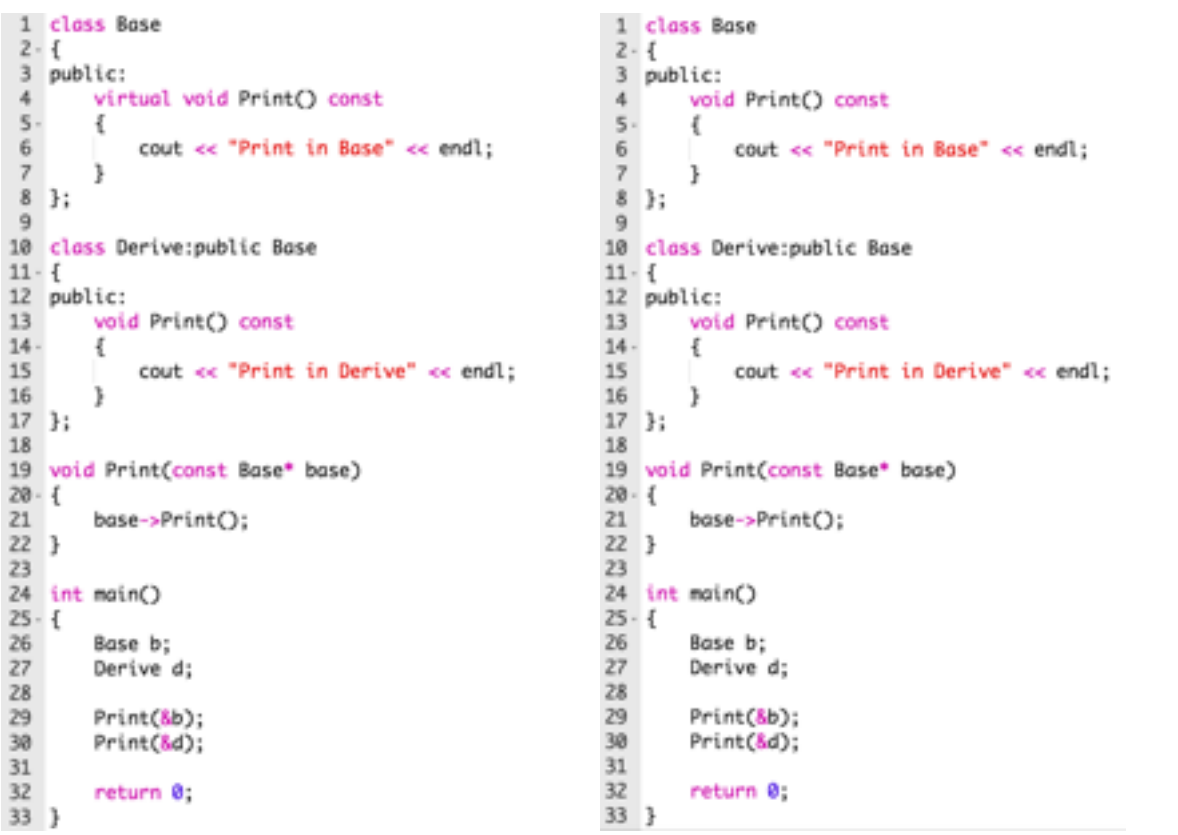
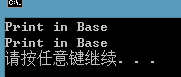
第三讲 C C++基础 （二）

1. 下面两段代码的输出分别是什么？



1.

2.

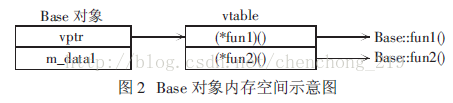
由于将基类定义为了虚函数。

2）简述C++虚函数作用及底层实现原理

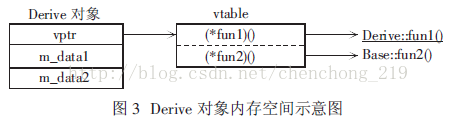
答：实现多态：C++在基类中声明一个带关键之Virtual的函数，这个函数叫虚函数；它可以在该基类的派生类中被重新定义并被赋予另外一种处理功能。通过指向指向派生类的基类指针或引用调用虚函数，编译器可以根据指向对象的类型在运行时决定调用的目标函数。这就实现了多态。

底层机制：

在每一个含有虚函数的类对象中，都含有一个VPTR，指向虚函数表。



派生类也会继承基类的虚函数，如果宅派生类中改写虚函数，虚函数表就会受到影响；表中元素所指向的地址不是基类的地址，而是派生类的函数地址。



当执行语句pBase->fun1()时，由于PBase指向的是派生类对象，于是就调用的Deriver::fun1()。

3）一个对象访问普通成员函数和虚函数哪个更快

答：速度是如果执行的次数少的话，两者相同；到达一定的次数之后执行速度是随机的。

4）在什么情况下，析构函数需要是虚函数

答：基类指针可以指向派生类的对象（多态性），如果删除该指针delete []p；就会调用该指针指向的派生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全。所以，将析构函数声明为虚函数是十分必要的。

5）内联函数、构造函数、静态成员函数可以是虚函数吗？

答：

inline, static, constructor三种函数都不能带有virtual关键字。

inline是编译时展开，必须有实体；

static属于class自己的，也必须有实体；

virtual函数基于vtable（内存空间），constructor函数如果是virtual的，调用时也需要根据vtable寻找，但是constructor是virtual的情况下是找不到的，因为constructor自己本身都不存在了，创建不到class的实例，没有实例，class的成员（除了public static/protected static for friend class/functions，其余无论是否virtual）都不能被访问了。

**C++函数中那些不可以被声明为虚函数**

常见的不不能声明为虚函数的有：普通函数（非成员函数）；静态成员函数；内联成员函数；构造函数；友元函数。

**1．为什么C++不支持普通函数为虚函数？**

普通函数（非成员函数）只能被overload，不能被override，声明为虚函数也没有什么意思，因此编译器会在编译时邦定函数。

**2．为什么C++不支持构造函数为虚函数？**

这个原因很简单，主要是从语义上考虑，所以不支持。因为构造函数本来就是为了明确初始化对象成员才产生的，然而virtual function主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用virtual函数来完成你想完成的动作。（这不就是典型的悖论）

**3．为什么C++不支持内联成员函数为虚函数？**

其实很简单，那内联函数就是为了在代码中直接展开，减少函数调用花费的代价，虚函数是为了在继承后对象能够准确的执行自己的动作，这是不可能统一的。（再说了，inline函数在编译时被展开，虚函数在运行时才能动态的邦定函数）

**4．为什么C++不支持静态成员函数为虚函数？**

这也很简单，静态成员函数对于每个类来说只有一份代码，所有的对象都共享这一份代码，他也没有要动态邦定的必要性。

**5．为什么C++不支持友元函数为虚函数？**

因为C++不支持友元函数的继承，对于没有继承特性的函数没有虚函数的说法**。**

6）构造函数中可以调用虚函数吗

答：

class Base

{

public:

    Base()

    {

        Fuction();

    }

    virtual void Fuction()

    {

        cout << "Base::Fuction" << endl;

    }

};

class A : public Base

{

public:

    A()

    {

        Fuction();

    }

    virtual void Fuction()

    {

        cout << "A::Fuction" << endl;

    }

};

// 这样定义一个A的对象，会输出什么？

A a;

 首先回答标题的问题，调用当然是没有问题的，但是获得的是你想要的结果吗？或者说你想要什么样的结果？

        有人说会输出：

A::Fuction

A::Fuction

  如果是这样，首先我们回顾下C++对象模型里面的构造顺序，在构造一个子类对象的时候，首先会构造它的基类，如果有多层继承关系，实际上会从最顶层的基类逐层往下构造（虚继承、多重继承这里不讨论），如果是按照上面的情形进行输出的话，那就是说在构造Base的时候，也就是在Base的构造函数中调用Fuction的时候，调用了子类A的Fuction，而实际上A还没有开始构造，这样函数的行为就是完全不可预测的，因此显然不是这样，实际的输出结果是：

Base::Fuction

A::Fuction

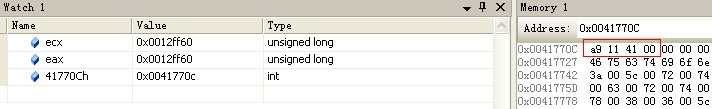
   据说在**[Java](http://lib.csdn.net/base/17" \o "Java EE知识库" \t "_blank)**中是上一种输出（感觉有点匪夷所思）。

        我们来单步看一下到底发生了什么？在A的构造函数里面首先会去调用Base的构造函数，Base的构造函数如下：

class Base  
{  
public:  
 Base()  
00411600  push        ebp    
00411601  mov         ebp,esp   
00411603  sub         esp,0CCh   
00411609  push        ebx    
0041160A  push        esi    
0041160B  push        edi    
0041160C  push        ecx    
0041160D  lea         edi,[ebp-0CCh]   
00411613  mov         ecx,33h   
00411618  mov         eax,0CCCCCCCCh   
0041161D  rep stos    dword ptr es:[edi]   
**0041161F  pop         ecx    
00411620  mov         dword ptr [ebp-8],ecx   
00411623  mov         eax,dword ptr [this]   
00411626  mov         dword ptr [eax],offset Base::`vftable' (41770Ch)** {  
  Fuction();  
**0041162C  mov         ecx,dword ptr [this]   
0041162F  call        Base::Fuction (4111A9h)**

 }  
00411634  mov         eax,dword ptr [this]   
00411637  pop         edi    
00411638  pop         esi    
00411639  pop         ebx    
0041163A  add         esp,0CCh   
00411640  cmp         ebp,esp   
00411642  call        @ILT+460(\_\_RTC\_CheckEsp) (4111D1h)   
00411647  mov         esp,ebp   
00411649  pop         ebp    
0041164A  ret

        从单步跟踪来看，注意黑色加粗的那部分汇编代码，ecx中存放的是对象的地址（0x0012ff60，我的机器上的情况看下图，有图有真相），首先是设置vtable的地址到对象的前四个字节（不同的编译器可能不同），然后就直接调用了Base::Fuction函数，并没有走虚机制，而我们此时看虚表中的状态，虚表已经填充的是0x4111a9，注意虚表的地址0x0041770c，而此时对象地址0x0012FF60前四个字节存放的正是0x0041770c。



    继续跟踪，流程又回到A的构造函数中，再次注意加粗部分的代码，从基类Base的构造函数返回后，在A的构造函数中，重设了虚表指针，现在的虚表指针是（0x417700h），同样调用Fuction的时候直接调用了A::Fuction函数，并没有使用虚机制，而且此时虚表0x417700h指向的位置存放的0x41110e正是A::Fuction的地址。



class A : public Base  
{  
public:  
 A()  
00411590  push        ebp    
00411591  mov         ebp,esp   
00411593  sub         esp,0CCh   
00411599  push        ebx    
0041159A  push        esi    
0041159B  push        edi    
0041159C  push        ecx    
0041159D  lea         edi,[ebp-0CCh]   
004115A3  mov         ecx,33h   
004115A8  mov         eax,0CCCCCCCCh   
004115AD  rep stos    dword ptr es:[edi]   
004115AF  pop         ecx    
004115B0  mov         dword ptr [ebp-8],ecx   
004115B3  mov         ecx,dword ptr [this]   
**004115B6  call        Base::Base (411140h)   
004115BB  mov         eax,dword ptr [this]   
004115BE  mov         dword ptr [eax],offset A::`vftable' (417700h)** {  
  Fuction();  
**004115C4  mov         ecx,dword ptr [this]   
004115C7  call        A::Fuction (41110Eh)** }  
004115CC  mov         eax,dword ptr [this]   
004115CF  pop         edi    
004115D0  pop         esi    
004115D1  pop         ebx    
004115D2  add         esp,0CCh   
004115D8  cmp         ebp,esp   
004115DA  call        @ILT+460(\_\_RTC\_CheckEsp) (4111D1h)   
004115DF  mov         esp,ebp   
004115E1  pop         ebp    
004115E2  ret

        其实事情就是这么简单。