C++多态解释与例子

类的多态特性是支持面向对象的语言最主要的特性，有过非面向对象语言开发经历的人，通常对这一章节的内容会觉得不习惯，因为很多人错误的认为，支持类的封装的语言就是支持面向对象的，其实不然，**Visual BASIC 6.0** 是典型的非面向对象的开发语言，但是它的确是支持类，支持类并不能说明就是支持面向对象，能够解决多态问题的语言，才是真正支持面向对象的开发的语言，所以务必提醒有过其它非面向对象语言基础的读者注意！  
  
　　多态的这个概念稍微有点模糊，如果想在一开始就想用清晰用语言描述它，让读者能够明白，似乎不太现实，所以我们先看如下代码：

//例程１    
#include <iostream>       
using namespace std;     
     
class Vehicle   
{     
public:     
    Vehicle(float speed,int total)   
    {   
        Vehicle::speed=speed;   
        Vehicle::total=total;   
    }   
    void ShowMember()   
    {   
        cout<<speed<<"|"<<total<<endl;   
    }   
protected:     
    float speed;   
    int total;   
};     
class Car:public Vehicle     
{     
public:     
    Car(int aird,float speed,int total):Vehicle(speed,total)     
    {     
        Car::aird=aird;     
    }   
    void ShowMember()   
    {   
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;   
    }   
protected:     
    int aird;   
};     
   
void main()     
{     
    Vehicle a(120,4);   
    a.ShowMember();   
    Car b(180,110,4);   
    b.ShowMember();   
    cin.get();   
}

在c++中是允许派生类重载基类成员函数的，对于类的重载来说，明确的，不同类的对象，调用其类的成员函数的时候，系统是知道如何找到其类的同名成员，上面代码中的a.ShowMember();，即调用的是Vehicle::ShowMember()，b.ShowMember();，即调用的是Car::ShowMemeber();

但是在实际工作中，很可能会碰到对象所属类不清的情况，下面我们来看一下派生类成员作为函数参数传递的例子，代码如下：

//例程２    
#include <iostream>       
using namespace std;     
     
class Vehicle   
{     
public:     
    Vehicle(float speed,int total)   
    {   
        Vehicle::speed=speed;   
        Vehicle::total=total;   
    }   
    void ShowMember()   
    {   
        cout<<speed<<"|"<<total<<endl;   
    }   
protected:     
    float speed;   
    int total;   
};     
class Car:public Vehicle     
{     
public:     
    Car(int aird,float speed,int total):Vehicle(speed,total)     
    {     
        Car::aird=aird;     
    }   
    void ShowMember()   
    {   
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;   
    }   
protected:     
    int aird;   
};     
   
void test(Vehicle &temp)   
{   
    temp.ShowMember();   
}   
   
void main()     
{   
    Vehicle a(120,4);   
    Car b(180,110,4);   
    test(a);   
    test(b);   
    cin.get();   
}

　　例子中，对象a与b分辨是基类和派生类的对象，而函数test的形参却只是Vehicle类的引用，按照类继承的特点，系统把Car类对象看做是一个Vehicle类对象，因为Car类的覆盖范围包含Vehicle类，所以test函数的定义并没有错误，我们想利用test函数达到的目的是，传递不同类对象的引用，分别调用不同类的,重载了的,ShowMember成员函数，但是程序的运行结果却出乎人们的意料，系统分不清楚传递过来的基类对象还是派生类对象，无论是基类对象还是派生类对象调用的都是基类的ShowMember成员函数。

为了要解决上述不能正确分辨对象类型的问题，c++提供了一种叫做**多态性**（polymorphism）的技术来解决问题，对于例程序１，这种能够在编译时就能够确定哪个重载的成员函数被调用的情况被称做**先期联编**（early binding），而在系统能够在运行时，能够根据其类型确定调用哪个重载的成员函数的能力，称为**多态性**，或叫**滞后联编**(late binding)，下面我们要看的例程３，就是滞后联编，滞后联编正是解决多态问题的方法。   
  
　　代码如下：

//例程３    
#include <iostream>       
using namespace std;     
     
class Vehicle   
{     
public:     
    Vehicle(float speed,int total)   
    {   
        Vehicle::speed = speed;   
        Vehicle::total = total;   
    }   
    virtual void ShowMember()//虚函数    
    {   
        cout<<speed<<"|"<<total<<endl;   
    }   
protected:     
    float speed;   
    int total;   
};     
class Car:public Vehicle     
{     
public:     
    Car(int aird,float speed,int total):Vehicle(speed,total)     
    {     
        Car::aird = aird;     
    }   
    virtual void ShowMember()//虚函数,在派生类中，由于继承的关系，这里的virtual也可以不加    
    {   
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;   
    }   
public:     
    int aird;   
};   
   
void test(Vehicle &temp)   
{   
    temp.ShowMember();   
}   
   
int main()     
{     
    Vehicle a(120,4);   
    Car b(180,110,4);   
    test(a);   
    test(b);   
    cin.get();   
}

　　多态特性的工作依赖虚函数的定义，在需要解决多态问题的重载成员函数前，加上**virtual**关键字，那么该成员函数就变成了虚函数，从上例代码运行的结果看，系统成功的分辨出了对象的真实类型，成功的调用了各自的重载成员函数。

　　多态特性让程序员省去了细节的考虑，提高了开发效率，使代码大大的简化，当然虚函数的定义也是有缺陷的，因为多态特性增加了一些数据存储和执行指令的开销，所以能不用多态最好不用。

**虚函数的定义要遵循以下重要规则：**  
  
**1.如果虚函数在基类与派生类中出现，仅仅是名字相同，而形式参数不同，或者是返回类型不同，那么即使加上了virtual关键字，也是不会进行滞后联编的。   
  
　　2.只有类的成员函数才能说明为虚函数，因为虚函数仅适合用与有继承关系的类对象，所以普通函数不能说明为虚函数。   
  
　　3.静态成员函数不能是虚函数,因为静态成员函数的特点是不受限制于某个对象。   
  
　　4.内联(inline)函数不能是虚函数，因为内联函数不能在运行中动态确定位置。即使虚函数在类的内部定义定义，但是在编译的时候系统仍然将它看做是非内联的。   
  
　　5.构造函数不能是虚函数，因为构造的时候，对象还是一片位定型的空间，只有构造完成后，对象才是具体类的实例。   
  
　　6.析构函数可以是虚函数,而且通常声名为虚函数。**   
  
　　说明一下，虽然我们说使用虚函数会降低效率，但是在处理器速度越来越快的今天，将一个类中的所有成员函数都定义成为virtual总是有好处的，它除了会增加一些额外的开销是没有其它坏处的，对于保证类的封装特性是有好处的。

　　对于上面虚函数使用的重要规则6，我们有必要用实例说明一下，**为什么具备多态特性的类的析构函数，有必要声明为virtual**。   
  
　　代码如下：

#include <iostream>       
using namespace std;     
     
class Vehicle   
{     
public:    
    Vehicle(float speed,int total)   
    {   
        Vehicle::speed=speed;   
        Vehicle::total=total;   
    }   
    virtual void ShowMember()   
    {   
        cout<<speed<<"|"<<total<<endl;   
    }   
    virtual ~Vehicle()   
    {   
        cout<<"载入Vehicle基类析构函数"<<endl;   
        cin.get();   
    }   
protected:     
    float speed;   
    int total;   
};     
class Car:public Vehicle     
{     
public:     
    Car(int aird,float speed,int total):Vehicle(speed,total)     
    {     
        Car::aird=aird;     
    }   
    virtual void ShowMember()   
    {   
        cout<<speed<<"|"<<total<<"|"<<aird<<endl;   
    }   
    virtual ~Car()   
    {   
        cout<<"载入Car派生类析构函数"<<endl;   
        cin.get();   
    }   
protected:     
    int aird;   
};     
   
void test(Vehicle &temp)   
{   
    temp.ShowMember();   
}   
void DelPN(Vehicle \*temp)   
{   
    delete temp;   
}   
void main()   
{     
    Car \*a=new Car(100,1,1);   
    a->ShowMember();   
    DelPN(a);   
    cin.get();   
}

　　从上例代码的运行结果来看，当调用DelPN(a);后，在析构的时候，系统成功的确定了先调用Car类的析构函数，而如果将析构函数的virtual修饰去掉，再观察结果，会发现析构的时候，始终只调用了基类的析构函数，由此我们发现，多态的特性的virtual修饰，不单单对基类和派生类的普通成员函数有必要，而且对于基类和派生类的析构函数同样重要。