浅谈C++多态性

C++编程语言是一款应用广泛，支持多种程序设计的计算机编程语言。我们今天就会为大家详细介绍其中C++多态性的一些基本知识，以方便大家在学习过程中对此能够有一个充分的掌握。　　多态性可以简单地概括为“一个接口，多种方法”，程序在运行时才决定调用的函数，它是面向对象编程领域的核心概念。多态(polymorphism)，字面意思多种形状。　　C++多态性是通过虚函数来实现的，虚函数允许子类重新定义成员函数，而子类重新定义父类的做法称为覆盖(override)，或者称为重写。（这里我觉得要补充，重写的话可以有两种，直接重写成员函数和重写虚函数，只有重写了虚函数的才能算作是体现了C++多态性）而重载则是允许有多个同名的函数，而这些函数的参数列表不同，允许参数个数不同，参数类型不同，或者两者都不同。编译器会根据这些函数的不同列表，将同名的函数的名称做修饰，从而生成一些不同名称的预处理函数，来实现同名函数调用时的重载问题。但这并没有体现多态性。　　多态与非多态的实质区别就是函数地址是早绑定还是晚绑定。如果函数的调用，在编译器编译期间就可以确定函数的调用地址，并生产代码，是静态的，就是说地址是早绑定的。而如果函数调用的地址不能在编译器期间确定，需要在运行时才确定，这就属于晚绑定。　　那么多态的作用是什么呢，封装可以使得代码模块化，继承可以扩展已存在的代码，他们的目的都是为了代码重用。而多态的目的则是为了接口重用。也就是说，不论传递过来的究竟是那个类的对象，函数都能够通过同一个接口调用到适应各自对象的实现方法。

　　最常见的用法就是声明基类的指针，利用该指针指向任意一个子类对象，调用相应的虚函数，可以根据指向的子类的不同而实现不同的方法。如果没有使用虚函数的话，即没有利用C++多态性，则利用基类指针调用相应的函数的时候，将总被限制在基类函数本身，而无法调用到子类中被重写过的函数。因为没有多态性，函数调用的地址将是一定的，而固定的地址将始终调用到同一个函数，这就无法实现一个接口，多种方法的目的了。

笔试题目：

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

class A
{
public:
    void foo()
    {
        printf("1\n");
    }
    virtual void fun()
    {
        printf("2\n");
    }
};
class B : public A
{
public:
    void foo()
    {
```

```
        printf("3\n");
    }
    void fun()
    {
        printf("4\n");
    }
};
int main(void)
{
    A a;
    B b;
    A *p = &a;
    p->foo();
    p->fun();
    p = &b;
    p->foo();
    p->fun();
    return 0;
}
```

第一个p->foo()和p->fuu()都很好理解，本身是基类指针，指向的又是基类对象，调用的都是基类本身的函数，因此输出结果就是1、2。　　第二个输出结果就是1、4。p->foo()和p->fuu()则是基类指针指向子类对象，正式体现多态的用法，p->foo()由于指针是个基类指针，指向是一个固定偏移量的函数，因此此时指向的就只能是基类的foo()函数的代码了，因此输出的结果还是1。而p->fun()指针是基类指针，指向的fun是一个虚函数，由于每个虚函数都有一个虚函数列表，此时p调用fun()并不是直接调用函数，而是通过虚函数列表找到相应的函数的地址，因此根据指向的对象不同，函数地址也将不同，这里将找到对应的子类的fun()函数的地址，因此输出的结果也会是子类的结果4。　　笔试的题目中还有一个另类测试方法。即

B *ptr = (B *)&a; ptr->foo(); ptr->fun();　　问这两调用的输出结果。这是一个用子类的指针去指向一个强制转换为子类地址的基类对象。结果，这两句调用的输出结果是3，2。　　并不是很理解这种用法，从原理上来解释，由于B是子类指针，虽然被赋予了基类对象地址，但是ptr->foo()在调用的时候，由于地址偏移量固定，偏移量是子类对象的偏移量，于是即使在指向了一个基类对象的情况下，还是调用到了子类的函数，虽然可能从始到终都没有子类对象的实例化出现。　　而ptr->fun()的调用，可能还是因为C++多态性的原因，由于指向的是一个基类对象，通过虚函数列表的引用，找到了基类中fun()函数的地址，因此调用了基类的函数。由此可见多态性的强大，可以适应各种变化，不论指针是基类的还是子类的，都能找到正确的实现方法。

```
//小结：1、有virtual才可能发生多态现象
// 2、不发生多态（无virtual）调用就按原类型调用
#include<iostream>
using namespace std;

class Base
```

```
{
public:
    virtual void f(float x)
    {
        cout<<"Base::f(float)"<< x <<endl;
    }
    void g(float x)
    {
        cout<<"Base::g(float)"<< x <<endl;
    }
    void h(float x)
    {
        cout<<"Base::h(float)"<< x <<endl;
    }
};
class Derived : public Base
{
public:
    virtual void f(float x)
    {
        cout<<"Derived::f(float)"<< x <<endl;    //多态、覆盖
    }
    void g(int x)
    {
        cout<<"Derived::g(int)"<< x <<endl;      //隐藏
    }
    void h(float x)
    {
        cout<<"Derived::h(float)"<< x <<endl;    //隐藏
    }
};
int main(void)
{
    Derived d;
    Base *pb = &d;
    Derived *pd = &d;
    // Good : behavior depends solely on type of the object
    pb->f(3.14f);   // Derived::f(float) 3.14
    pd->f(3.14f);   // Derived::f(float) 3.14

    // Bad : behavior depends on type of the pointer
    pb->g(3.14f);   // Base::g(float)  3.14
    pd->g(3.14f);   // Derived::g(int) 3

    // Bad : behavior depends on type of the pointer
    pb->h(3.14f);   // Base::h(float) 3.14
    pd->h(3.14f);   // Derived::h(float) 3.14
    return 0;
}
```

令人迷惑的隐藏规则 本来仅仅区别重载与覆盖并不算困难，但是C++的隐藏规则使问题复杂性陡然增加。 这里“隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下： （1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。 （2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。 上面的程序中： （1）函数Derived::f(float)覆盖了Base::f(float)。 （2）函数Derived::g(int)隐藏了Base::g(float)，而不是重载。 （3）函数Derived::h(float)隐藏了Base::h(float)，而不是覆盖。 C++纯虚函数 一、定义 纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0” virtual void funtion()=0 二、引入原因 1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。 2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。 为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;），则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。 三、相似概念 1、多态性 指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。 **a、编译时多态性：通过重载函数实现 b、运行时多态性：通过虚函数实现。** 2、虚函数 虚函数是在基类中被声明为virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态覆盖（Override） 3、抽象类 包含纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对象。