**C++ 多态详解**

**多态现在一般的用法，就是拿一个父类的指针去调用子类中被重写的方法。但我搞不懂为什么要那么做，我们直接在子类中写一个同名的成员函数，从而隐藏父类的函数不就行了么？**

***回答：***

**将父类比喻为电脑的外设接口，子类比喻为外设，现在我有移动硬盘、U盘以及MP3，它们3个都是可以作为存储但是也各不相同。如果我在写驱动的时候，我用个父类表示外设接口，然后在子类中重写父类那个读取设备的虚函数，那这样电脑的外设接口只需要一个。但如果我不是这样做，而是用每个子类表示一个外设接口，那么我的电脑就必须有3个接口分别来读取移动硬盘、U盘以及MP3。若以后我还有SD卡读卡器，那我岂不是要将电脑拆了，焊个SD卡读卡器的接口上去？**

**所以，用父类的指针指向子类，是为了面向接口编程。大家都遵循这个接口，弄成一样的，到哪里都可以用，准确说就是“一个接口，多种实现“。**

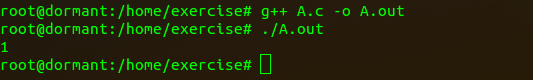
**这个解答很有意思！！！**

**忽然感觉学了很长时间的C++只是知道有多态这样一个重要的概念，编程时怎么用，却真的没有仔细想一下它的机制，也就是它是怎么实现的。于是看了些资料，做一下总结。**

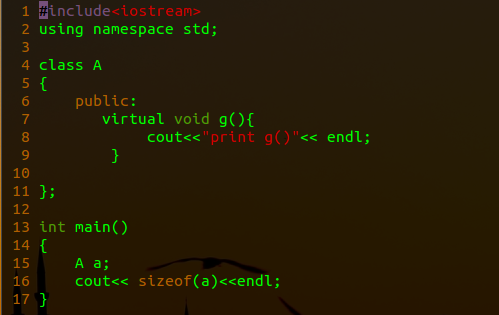
**先看一个例子：**

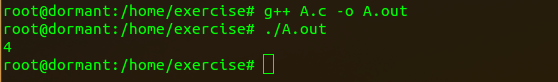


**执行完之后的结果是这样的：**



**这个很好理解，但当我们将函数g()加上virtual之后再看结果会看到**





**变成了4。这是因为在后者中变成了虚函数了。**

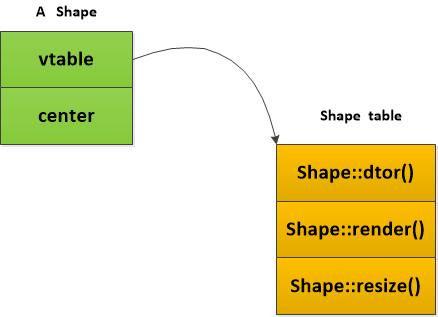
**virtual是让子类与父类之间的同名函数有联系，这就是多态性，实现动态绑定。**

**任何类若是有虚函数就会比比正常的类大一点，所有有virtual的类的对象里面最头上会自动加上一个隐藏的，不让我知道的指针，它指向一张表，这张表叫做vtable，vtable里是所有virtual函数的地址。**

**下边来看这样两段代码：**

1. class Shape {
2. public:
3. Shape();
4. virtual ~Shape();
5. virtual void render();
6. void move(const pos&);
7. virtual void resize();
8. protected:
9. pos center;
10. };

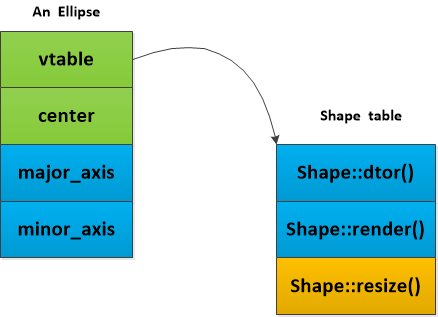
**这个类的内存分布是这样的：**



**就是在成员变量前有一个vtable的指针，它会指向一个table,这个table叫做虚函数表。**

1. class Ellipse : public Shape{
2. public:
3. Ellipse (float majr, float minr);
4. virtual void render();
6. protected:
7. float major\_axis;
8. float minor\_axis;
10. };

**Ellipse继承与Shape，看一下它的内存分布：**



**这里的vtable不是对象的，而是属于类的，这就是多态的实现机制。**

**如果需要了解更多关于虚表的知识，链接：**

[**http://blog.csdn.net/haoel/article/details/1948051/**](http://blog.csdn.net/haoel/article/details/1948051/)

**这样由上面的解释我们来详细讲解一下多态的概念和实现：**

**多态（Polymorphism）按字面的意思就是“多种状态”。在面向对象语言中，接口的多种不同的实现方式即为多态。引用Charlie Calverts对多态的描述——多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。其实我看到过一句话：调用同名函数却会因上下文的不同而有不同的实现。我觉得这样更加贴切，还加入了多态三要素：（1）相同函数名  （2）依据上下文  （3）实现却不同；**

**来看这个例子：**

1. #include<iostream>
2. using namespace std;
4. class A {
5. public:
6. A() : i(10）{}
7. virtual void f() { cout<< "A::f()"<<i<<endl;}
9. int i;
10. };
12. class B : public A{
13. public:
14. B() : j(20) {}
15. virtual void f() { cout << "B::f()" << j <<endl;}
17. int j;
18. };
20. int main()
21. {
23. A a;
24. B b;
25. A \*p = &b;
26. p->f();
27. return 0;
28. }

**这时我们执行这个程序，b的f()函数会执行，执行结果是"B::f() 20"，这里就是多态中的动态绑定，本来是基类型的指针赋给了子类型的对象地址，这样当运行时才能知道执行哪个f()函数。**

**之后修改main函数：**

1. int main()
2. {
3. A a;
4. B b;
5. A \*p = &b;
6. p->f();
7. a = b;
8. a.f();
9. return 0;
10. }

**执行结果是：**

**B::f() 20**

**A::f() 10**

**可见a.f()的结果输出是不同的。有很多理由说明这个，其一就是通过指针或引用才是动态绑定，通过点运算是不可以的。**

**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。**

**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。**

**同时需要了解多态的特性的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！！！**