**1、C++ 关键字virtual**

Virtual是C++ OO机制中很重要的一个关键字。只要是学过C++的人都知道在类Base中加了Virtual关键字的函数就是虚拟函数（例如函数print），于是在Base的派生类Derived中就可以通过重写虚拟函数来实现对基类虚拟函数的覆盖。当基类Base的指针point指向派生类Derived的对象时，对point的print函数的调用实际上是调用了Derived的print函数而不是Base的print函数。这是面向对象中的多态性的体现。

这也许会使人联想到函数的重载，但稍加对比就会发现两者是完全不同的：

（1） 重载的几个函数必须在同一个类中；

覆盖的函数必须在有继承关系的不同的类中

（2） 覆盖的几个函数必须函数名、参数、返回值都相同；

重载的函数必须函数名相同，参数不同。参数不同的目的就是为了在函数调用的时候编译器能够通过参数来判断程序是在调用的哪个函数。这也就很自然地解释了为什么函数不能通过返回值不同来重载，因为程序在调用函数时很有可能不关心返回值，编译器就无法从代码中看出程序在调用的是哪个函数了。

（3） 覆盖的函数前必须加关键字Virtual；

重载和Virtual没有任何瓜葛，加不加都不影响重载的运作。

关于C++的隐藏规则：

（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）。

#include <iostream.h>

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; }

void g(int x){ cout << "Derived::g(int) " << x << endl; }

void h(float x){ cout << "Derived::h(float) " << x << endl; }

};

void 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 (surprise!)

// Bad : behavior depends on type of the pointer

pb->h(3.14f); // Base::h(float) 3.14 (surprise!)

pd->h(3.14f); // Derived::h(float) 3.14

}

bp 和dp 指向同一地址，按理说运行结果应该是相同的，而事实上运行结果不同，所以他把原因归结为C++的隐藏规则，其实这一观点是错的。决定bp和dp调用函数运行结果的不是他们指向的地址，而是他们的指针类型。“只有在通过基类指针或引用间接指向派生类子类型时多态性才会起作用”(C++ Primer 3rd Edition)。pb是基类指针，pd是派生类指针，pd的所有函数调用都只是调用自己的函数，和多态性无关，所以pd的所有函数调用的结果都输出Derived::是完全正常的；pb的函数调用如果有virtual则根据多态性调用派生类的，如果没有virtual则是正常的静态函数调用，还是调用基类的，所以有virtual的f函数调用输出Derived::，其它两个没有virtual则还是输出Base::很正常啊，nothing surprise!

所以并没有所谓的隐藏规则，虽然《高质量C++/C 编程指南》是本很不错的书，可大家不要迷信哦。记住“只有在通过基类指针或引用间接指向派生类子类型时多态性才会起作用”。

纯虚函数：

C++语言为我们提供了一种语法结构，通过它可以指明，一个虚拟函数只是提供了一个可被子类型改写的接口。但是，它本身并不能通过虚拟机制被调用。这就是纯虚拟函数（purevirtual function）。 纯虚拟函数的声明如下所示：

class Query {

public:

// 声明纯虚拟函数

virtual ostream& print( ostream&=cout ) const = 0;

// ...

};

这里函数声明后面紧跟赋值0。

包含（或继承）一个或多个纯虚拟函数的类被编译器识别为抽象基类。试图创建一个抽象基类的独立类对象会导致编译时刻错误。（类似地通过虚拟机制调用纯虚拟函数也是错误的例如）

// Query 声明了纯虚拟函数

// 所以, 程序员不能创建独立的 Query 类对象

// ok: NameQuery 中的 Query 子对象

Query \*pq = new NameQuery( "Nostromo" );

// 错误: new 表达式分配 Query 对象

Query \*pq2 = new Query;

抽象基类只能作为子对象出现在后续的派生类中。

如果只知道virtual加在函数前，那对virtual只了解了一半，virtual还有一个重要用法是virtual public，就是虚拟继承。虚拟继承在C++ Primer中有详细的描述，下面稍作修改的阐释一下：

在缺省情况下C++中的继承是“按值组合”的一种特殊情况。当我们写

class Bear : public ZooAnimal { ... };

每个Bear 类对象都含有其ZooAnimal 基类子对象的所有非静态数据成员以及在Bear中声明的非静态数据成员类似地当派生类自己也作为一个基类对象时如：

class PolarBear : public Bear { ... };

则PolarBear 类对象含有在PolarBear 中声明的所有非静态数据成员以及其Bear 子对象的所有非静态数据成员和ZooAnimal 子对象的所有非静态数据成员。在单继承下这种由继承支持的特殊形式的按值组合提供了最有效的最紧凑的对象表示。在多继承下当一个基类在派生层次中出现多次时就会有问题最主要的实际例子是iostream 类层次结构。ostream 和istream 类都从抽象ios 基类派生而来，而iostream 类又是从ostream 和istream 派生

class iostream :public istream, public ostream { ... };

缺省情况下，每个iostream 类对象含有两个ios 子对象：在istream 子对象中的实例以及在ostream 子对象中的实例。这为什么不好？从效率上而言，存储ios 子对象的两个复本，浪费了存储区，因为iostream 只需要一个实例。而且，ios 构造函数被调用了两次每个子对象一次。更严重的问题是由于两个实例引起的二义性。例如，任何未限定修饰地访问ios 的成员都将导致编译时刻错误：到底访问哪个实例？如果ostream 和istream 对其ios 子对象的初始化稍稍不同，会怎样呢？怎样通过iostream 类保证这一对ios 值的一致性？在缺省的按值组合机制下，真的没有好办法可以保证这一点。

C++语言的解决方案是，提供另一种可替代按“引用组合”的继承机制虚拟继承（virtual inheritance ）在虚拟继承下只有一个共享的基类子对象被继承而无论该基类在派生层次

中出现多少次共享的基类子对象被称为虚拟基类。

通过用关键字virtual 修政一个基类的声明可以将它指定为被虚拟派生。例如，下列声明使得ZooAnimal 成为Bear 和Raccoon 的虚拟基类：

// 关键字 public 和 virtual

// 的顺序不重要

class Bear : public virtual ZooAnimal { ... };

class Raccoon : virtual public ZooAnimal { ... };

虚拟派生不是基类本身的一个显式特性，而是它与派生类的关系如前面所说明的，虚拟继承提供了“按引用组合”。也就是说，对于子对象及其非静态成员的访问是间接进行的。这使得在多继承情况下，把多个虚拟基类子对象组合成派生类中的一个共享实例，从而提供了必要的灵活性。同时，即使一个基类是虚拟的，我们仍然可以通过该基类类型的指针或引用，来操纵派生类的对象。