简单地说，虚函数是动态绑定的基础；动态绑定是实现运行时多态的基础。

要触发动态绑定，需满足两个条件：

(1)  只有虚函数才能进行动态绑定，非虚函数不进行动态绑定。

(2)  必须通过基类类型的引用或指针进行函数调用。

通过基类指针或基类引用做形参，当实参传入不同的派生类(或基类)的指针或引用，在函数内部触发动态绑定，从而来运行时实现多态的。

---------------------

动态类型与静态类型

静态类型

         是指不需要考虑表达式的执行期语义，仅分析程序文本而决定的表达式类型。静态类型仅依赖于包含表达式的程序文本的形式，而在程序运行时不会改变。通俗的讲，就是上下文无关，在编译时就可以确定其类型。

动态类型

        是指由一个左值表达式表示的左值所引用的最终派生对象的类型。例：如果一个静态类型为“类 B ”的指针p 指向一个继承于 B的类 D 的对象，则表达式 \*p 的动态类型为“D”。引用按照相似规则处理。一般地讲，基类的指针和基类引用有可能为动态类型，就是说在运行之前不能够确定其真实类型。通常我们说，“基类指针指向的对象的实际/真正类型”或“基类引用所引用的对象的实际/真正类型”，就是它们的动态类型。很显然，这个动态类型是  C++ 语言通过指针和引用实现运行时多态能力的核心概念。

动态绑定与静态绑定

静态绑定：编译时绑定，通过对象调用

动态绑定：运行时绑定，通过地址实现

只有采用“指针->函数()”或“引用变量.函数()”的方式调用C++类中的虚函数才会执行动态绑定。对于C++中的非虚函数，因为其不具备动态绑定的特征，所以不管采用什么样的方式调用，都不会执行动态绑定。

即所谓动态绑定，就是基类的指针或引用有可能指向不同的派生类对象，对于非虚函数，执行时实际调用该函数的对象类型即为该指针或引用的静态类型（基类类型）；而对于虚函数，执行时实际调用该函数的对象类型为该指针或引用所指对象的实际类型。比如下面代码：

class Base {

public:

void func() {

cout << "func() in Base." << endl;

}

virtual void test() {

cout << "test() in Base." << endl;

}

};

class Derived : public Base {

void func() {

cout << "func() in Derived." << endl;

}

virtual void test() {

cout << "test() in Derived." << endl;

}

};

int main() {

Base\* b;

b = new Derived();

b->func();

b->test();

}

由运行结果可以看到，b是一个基类指针，它指向了一个派生类对象，基类Base里面有两个函数，其中test为虚函数，func为非虚函数。因此，对于test就表现为动态绑定，实际调用的是派生类对象中的test，而func为非虚函数，因此它表现为静态绑定，也就是说指针类型是什么，就会调用该类型相应的函数。

虚函数、动态绑定、运行时多态之间的关系

简单地说，虚函数是动态绑定的基础；动态绑定是实现运行时多态的基础。

要触发动态绑定，需满足两个条件：

(1)  只有虚函数才能进行动态绑定，非虚函数不进行动态绑定。

(2)  必须通过基类类型的引用或指针进行函数调用。

通过基类指针或基类引用做形参，当实参传入不同的派生类(或基类)的指针或引用，在函数内部触发动态绑定，从而来运行时实现多态的。

下面通过实际例子才展示运行时多态的实现方式：

如下代码是一个Base基类和它的三个派生类Derived1，Derived2，Derived3。

class Base {

public:

void Print() {

cout << "Print() from Base." << endl;

}

virtual void Display() {

cout << "Display() from Base." << endl;

}

};

class Derived1 : public Base {

public:

void Print() {

cout << "Print() from Derived1." << endl;

}

void Display() {

cout << "Display() from Derived2." << endl;

}

};

class Derived2 : public Base {

public:

void Print() {

cout << "Print() from Derived2." << endl;

}

void Display() {

cout << "Display() from Derived2." << endl;

}

};

class Derived3 : public Base {

public:

void Print() {

cout << "Print() from Derived3." << endl;

}

void Display() {

cout << "Display() from Derived3." << endl;

}

};

下面两个全局函数分别以基类指针和基类引用作形参来实现运行时多态：

//通过基类引用作形参实现多态

void Polymorphic1(Base& b) {

b.Print();

b.Display();

}

//通过基类指针作形参实现多态

void Polymorphic2(Base\* b) {

b->Print();

b->Display();

}

下面是测试代码：

int main() {

Base b;

Derived1 d1;

Derived2 d2;

Derived3 d3;

vector<Base> base\_vec;

base\_vec.push\_back(b);

base\_vec.push\_back(d1);

base\_vec.push\_back(d2);

base\_vec.push\_back(d3);

vector<Base\*> base\_ptr\_vec;

base\_ptr\_vec.push\_back(&b);

base\_ptr\_vec.push\_back(&d1);

base\_ptr\_vec.push\_back(&d2);

base\_ptr\_vec.push\_back(&d3);

cout << endl << "对通过基类引用作形参实现多态进行测试" << endl;

//对通过基类引用作形参实现多态进行测试 （测试方式错误）

for (int i = 0; i != base\_vec.size(); ++i) {

Polymorphic1(base\_vec[i]);

}

cout << endl << "对通过基类指针作形参实现多态进行测试" << endl;

//对通过基类指针作形参实现多态进行测试

for (int i = 0; i != base\_vec.size(); ++i) {

Polymorphic2(base\_ptr\_vec[i]);

}

cout << endl << "对通过基类引用作形参实现多态进行测试" << endl;

//对通过基类引用作形参实现多态进行测试

Polymorphic1(b);

Polymorphic1(d1);

Polymorphic1(d2);

Polymorphic1(d3);

return 0;

}

测试结果如下图：

        我们看到第一组想对通过基类引用作形参实现多态进行测试，需要将不同派生类的对象作实参传过去。然而，把派生类放到基类的vector中存储的过程中，派生类对象被自动转换为基类对象了，因而实际存储的均为基类对象，所以再从vector中取出对象元素做实参传递的时候，传递的均为基类对象，所以测试失败。

        对这个比较挫的测试进行的分析和思考：由C++ STL的vector容器中存储的对象拷贝引起的对capacity属性 的理解

        而下面两组我们分别把不同派生类的指针和对象作实参进行测试，结果显示均实现了运行时多态：即传入不同的对象，就会调用该对象相应的Display函数，因为在基类中，Display为虚函数，所以这里它实现了对象的动态绑定，从而实现了运行时多态；与之做对比的Print函数在基类中为非虚构函数，因此对Print函数不会进行动态绑定，而是静态绑定：即基类指针只能调用基类中的Print函数。

参考文献：

1. http://blog.csdn.net/lynnboy/article/details/154894

2. http://www.cnblogs.com/chris12/archive/2012/10/28/2744131.html

3. http://blog.csdn.net/livelylittlefish/article/details/2171521

          原创文章，转载请注明： 转载自  IIcyZhao's Road

          本文链接地址:  http://blog.csdn.net/iicy266/article/details/11906509

---------------------