# 虚继承，虚函数，虚析构函数。

**虚继承：为了不让子类中包含两份父类的数据。**

**虚函数：为了实现多态性。**

**虚析构函数：为了不内存泄露**

**多态具体到语法中是指，使用父类指针指向子类对象，并可以通过该指针调用子类的方法。**

**产生多态的基础是继承关系，没有继承就没有多态**

**多态的语法核心是virtual关键字，必须使用virtual才能使多个类间建立多态关系。**

**封装、继承、多态是面向对象的三大特性。**

# 多态：

## 1：多态概述：

**什么是多态？**

****

**以封装 和继承为基础，才能谈到动态多态即多态的实现是建立在类的继承和封装的基础上的。**

**多态分为静态多态和动态多态：**

### 静态多态：

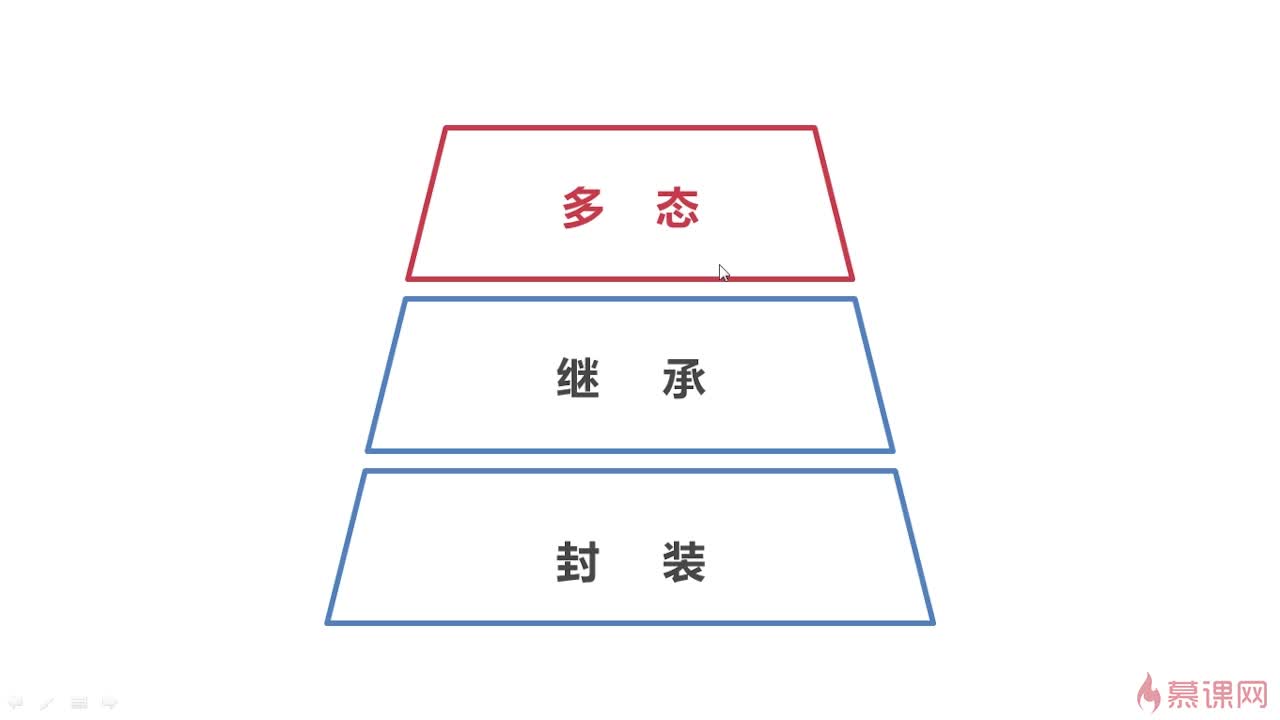
**也称为早绑定，例如一个类中互为重载的函数的情况（相同对象接收到不同消息**

**而产生不同的动作）**

### 动态多态：

**也称为晚绑定，其必须以继承，封装为基础，动态多态至少有两个类，一个是父类，一个是子类，有三个类的时候，动态多态才表现的更加明显（不同对象接收到相同消息产生了不同的动作）**

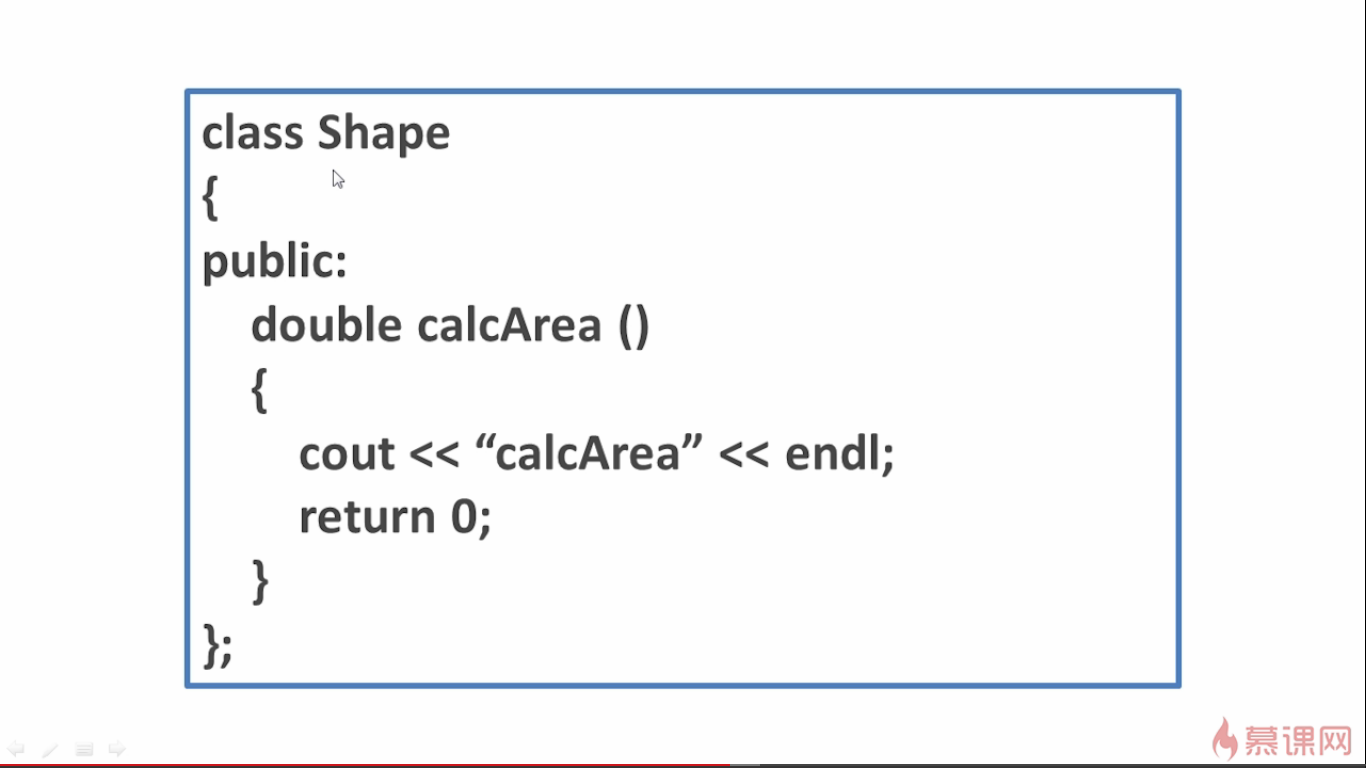
**封装-类；继承-子类关系**

****

# 虚函数及实现原理：

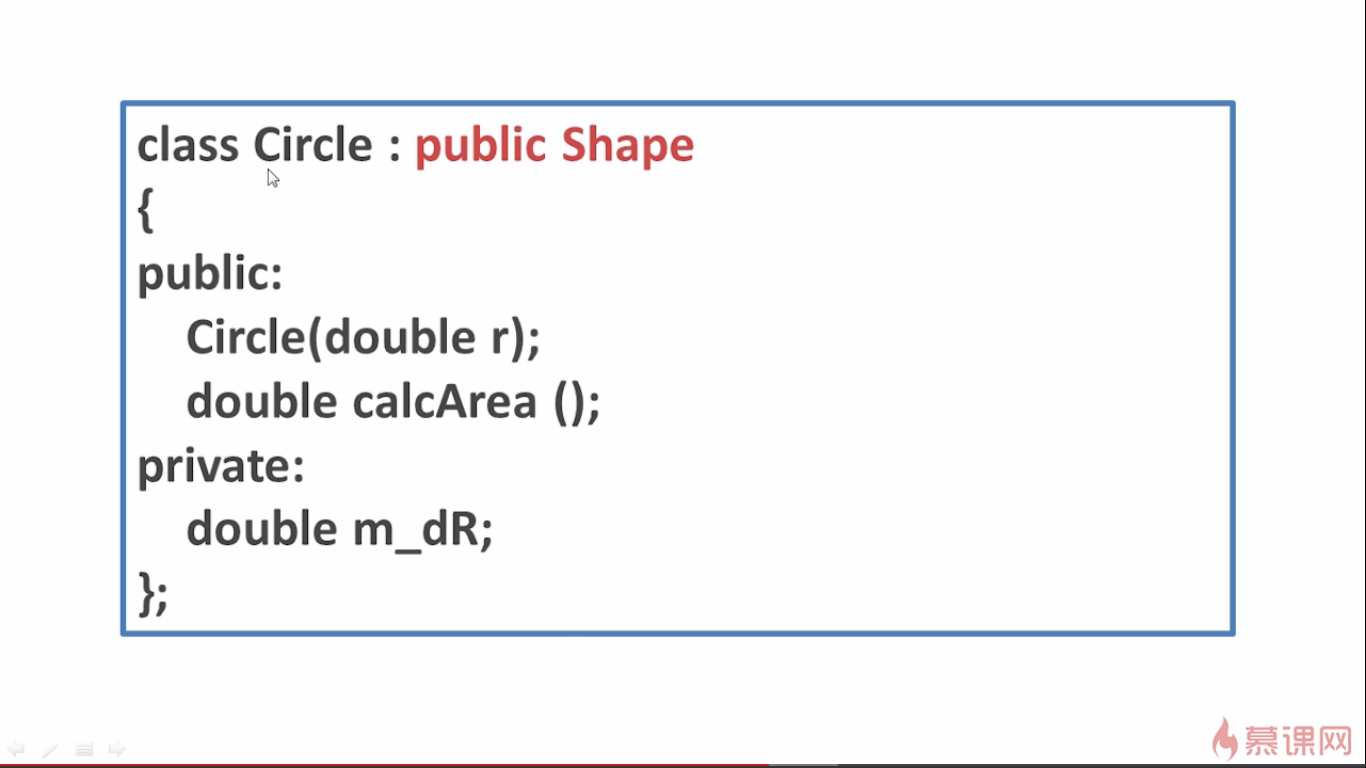
**例：**

**定义一个父类：**

****

**定义两个子类：**

**子类一：**

****

**子类一对应的.cpp文件：**

**#include”Circle.h”**

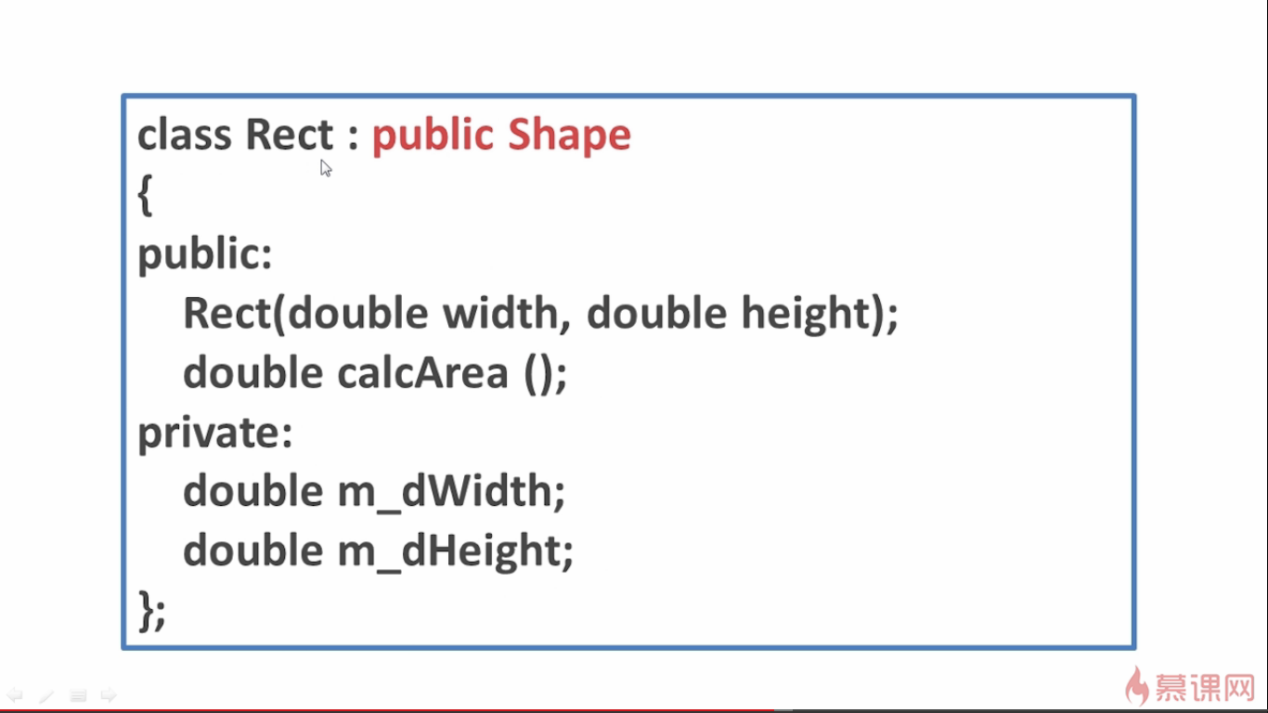
**Double Circle::calcArea()**

**{**

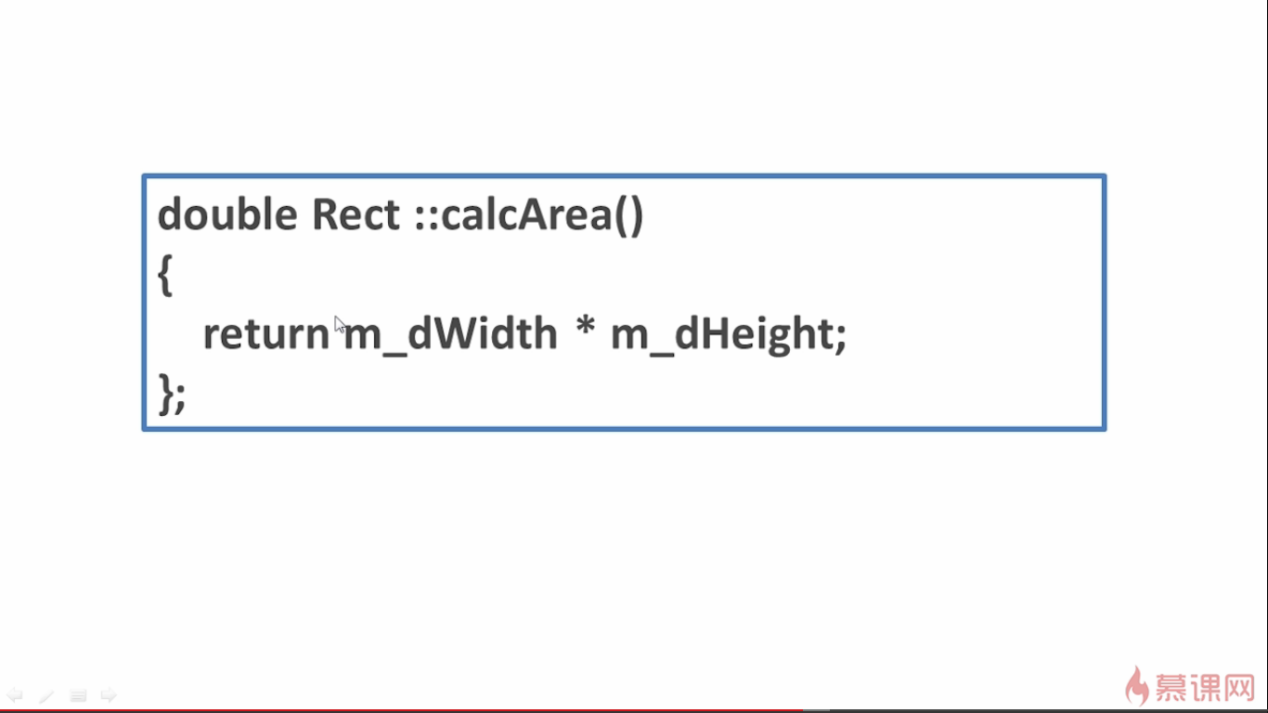
**Return 3.14\*m\_dR\*m\_Dr;**

**}**

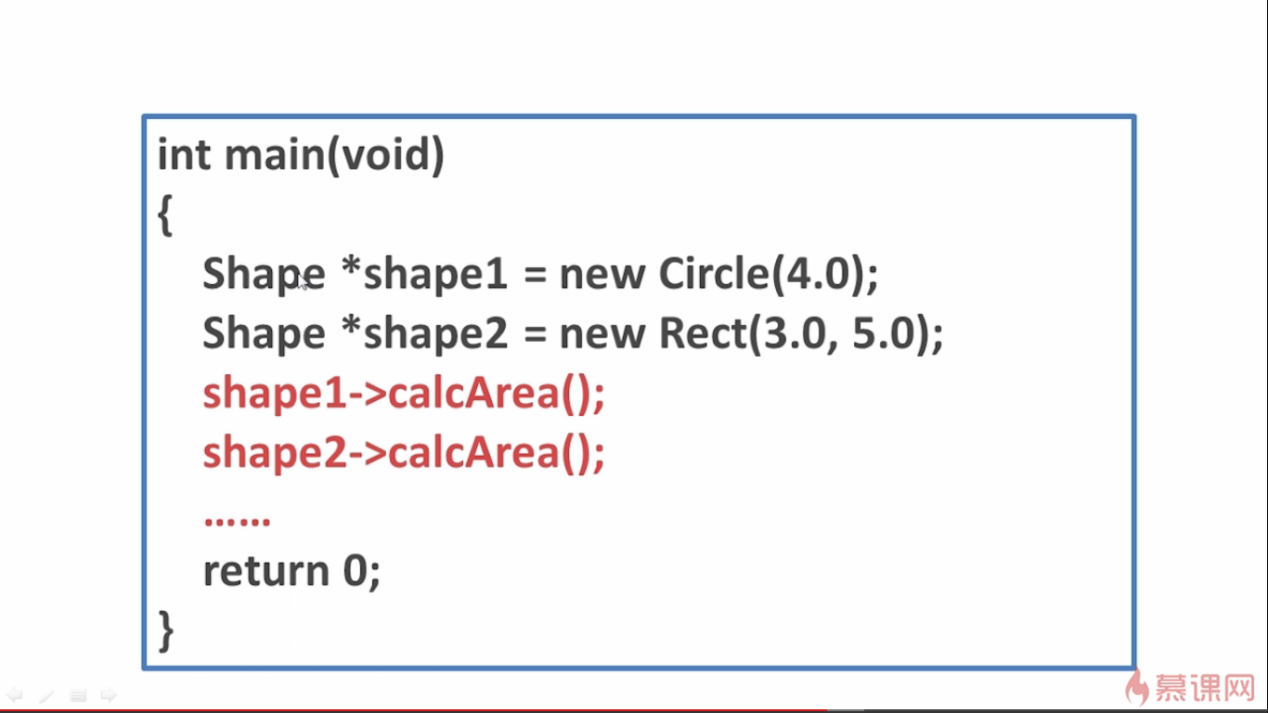
**子类二：**

****

**子类二对应的.cpp文件：**

****

**主函数：**

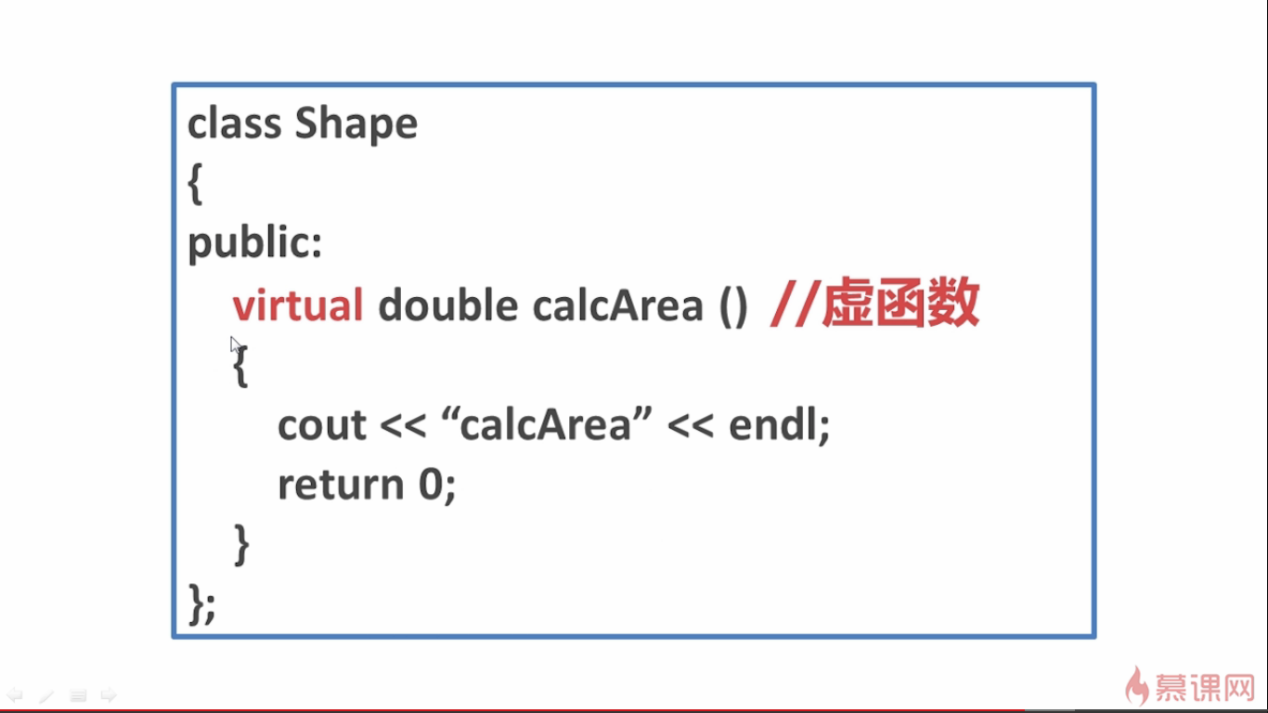
****

**主函数想要实现的是通过父类shape的指针分别调用Circle和Rect的计算面积的函数，但是，使用主函数的那种方法肯定是实现不了的，因为shape1/2->calcArea()会调用shape类中定义的calcArea（）函数即输出calcArea,不会按照我们的预想调用对应的子类的计算面积的函数，那么，我们应该怎样解决这个问题呢？这就引出了虚函数的概念：**

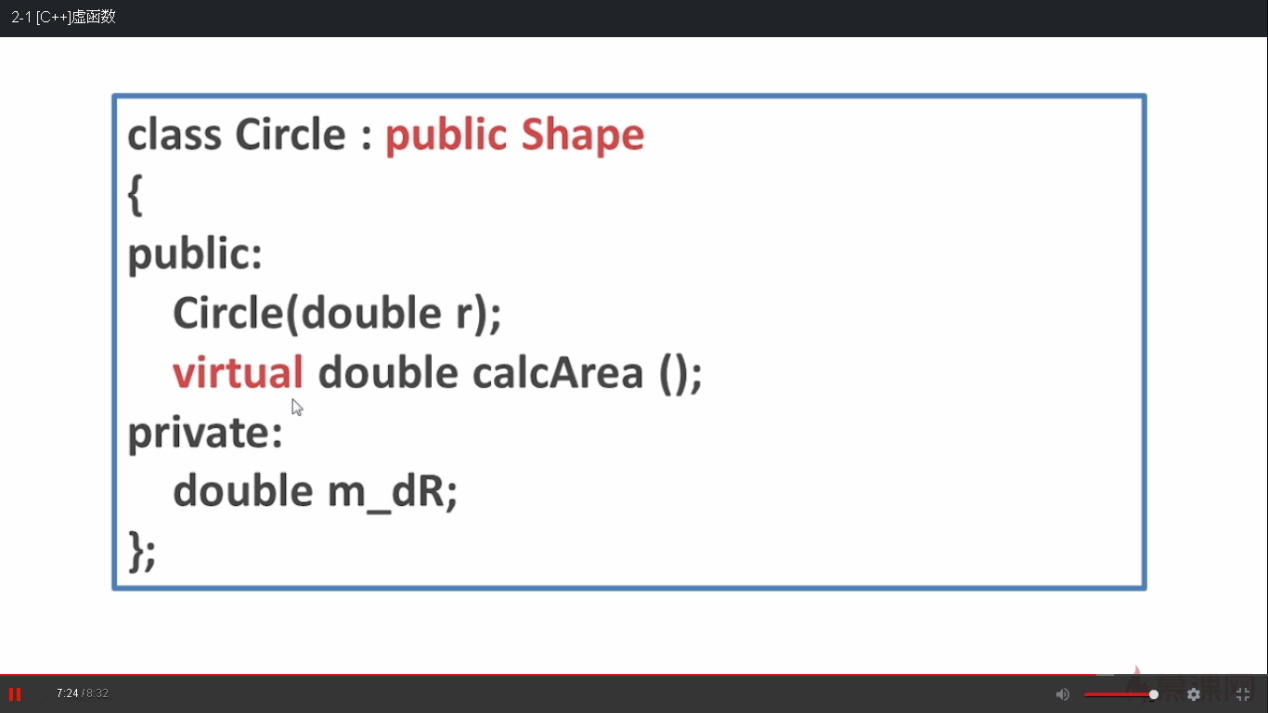
**用virtual修饰函数，该函数就变成了虚函数。虚函数为了解决动态多态问题而产生的**

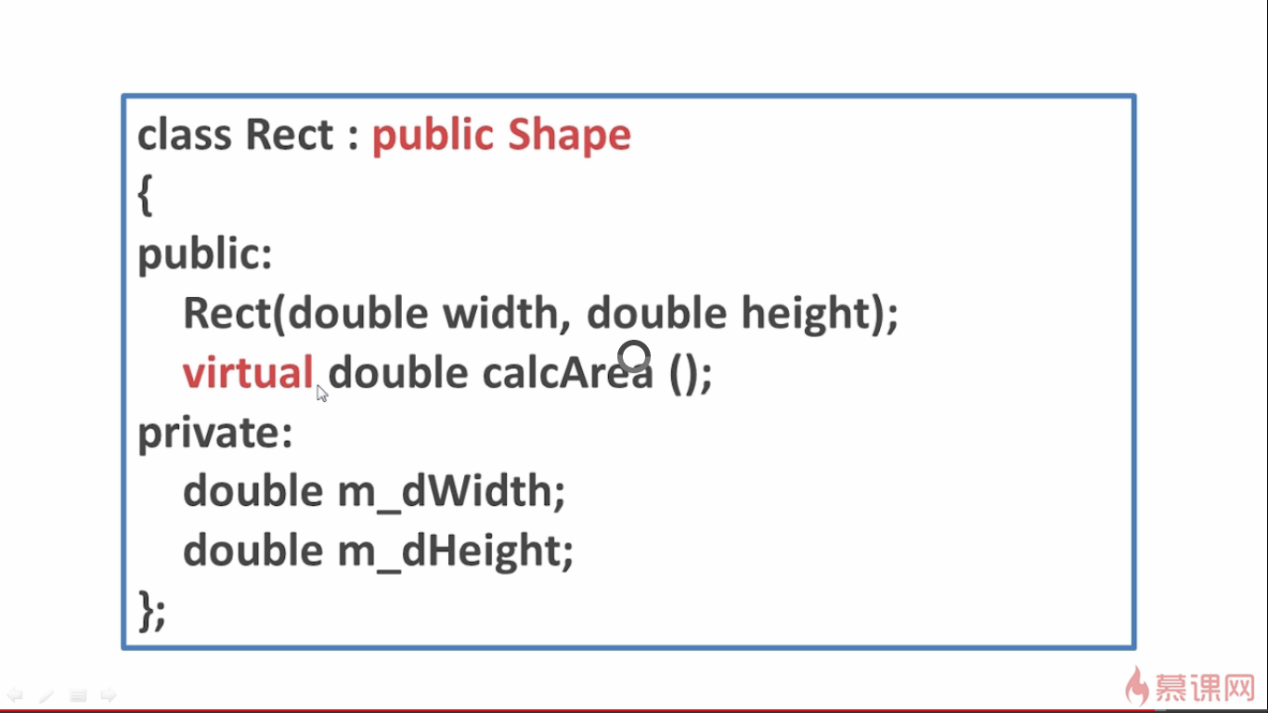
**还是上面那个例子，使用virtual关键字实现虚函数：**

**1：在shape中的想要实现虚函数的函数名前加virtual关键字**

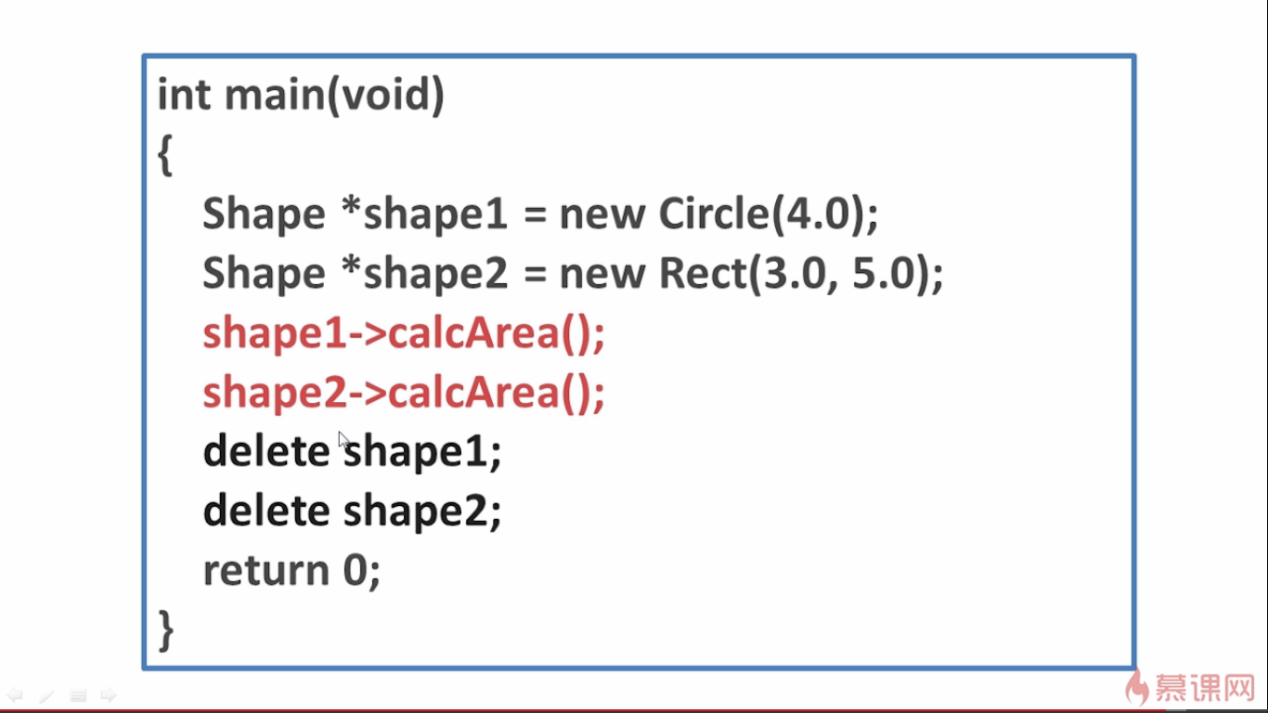
****

**2：在子类的对应函数的开头加virtual（这一步可以省略，如果不加，系统会自动加上，但是为了清晰可读，建议加上）**

****

****

**然后在主函数中：**

****

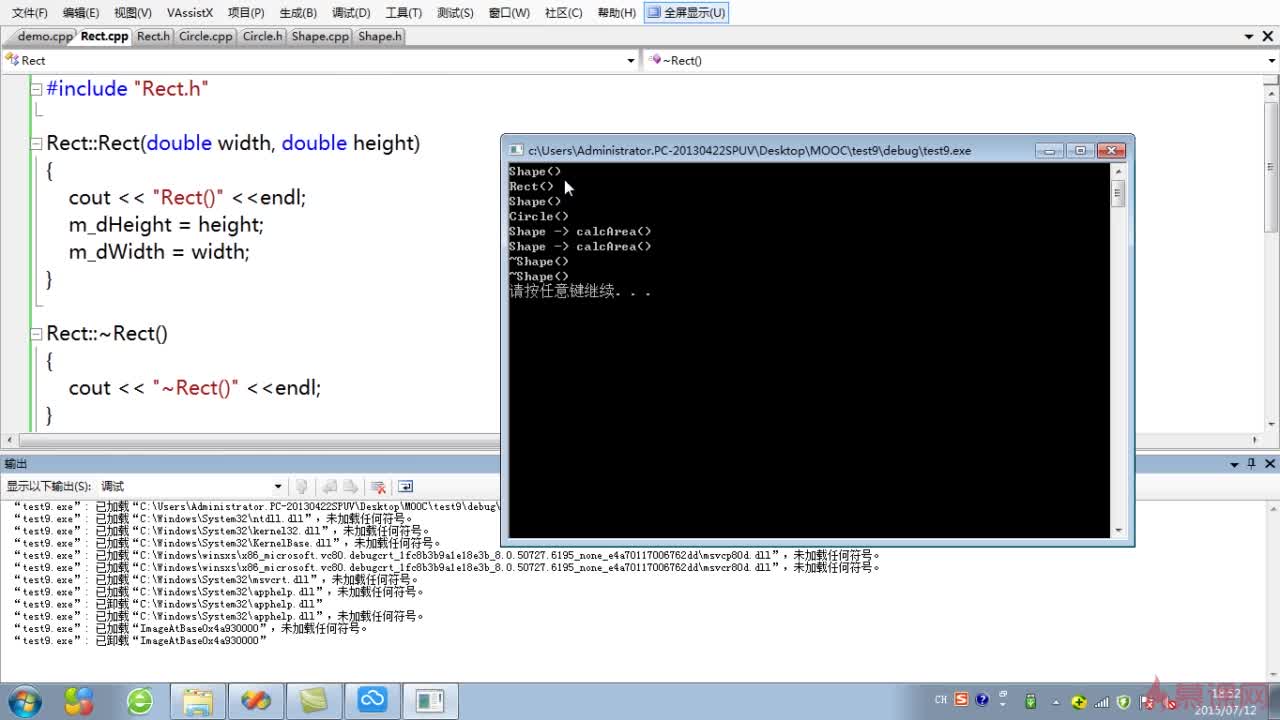
**这样就可以实现函数的虚化及使用相同的指令却实现不同的操作，即多态。**

**虚函数详解：**

**虚函数，是用子类实例化一个父类（共有继承）时，通过在共有函数前+virtual来表示调用该父类函数时可能在派生类们中有同名函数的不同实现。满足复制兼容规则。**

**Shape \*shape1 = new Reac(3，6);**

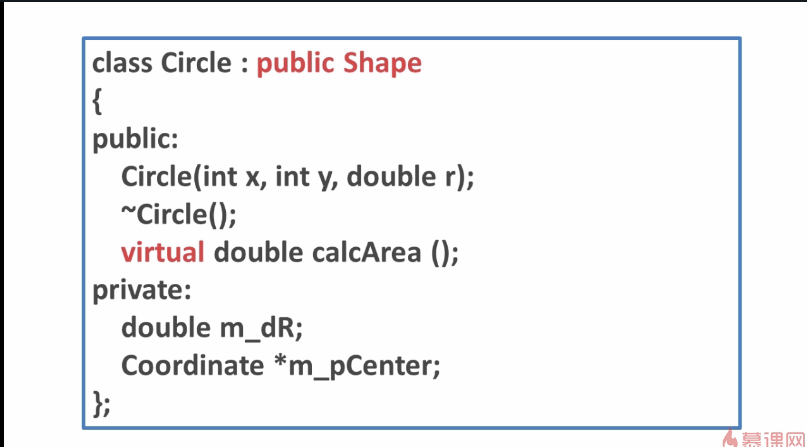
**Shape \*shape2 = new Circle(5);**

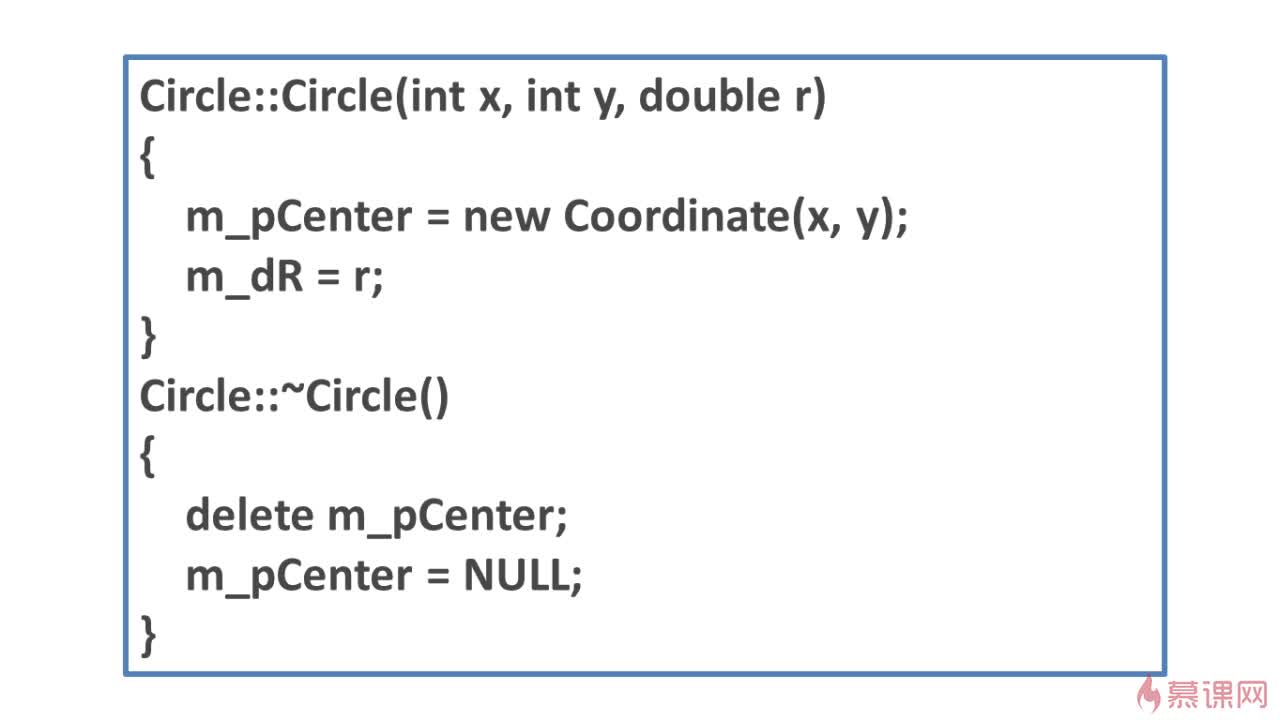
****

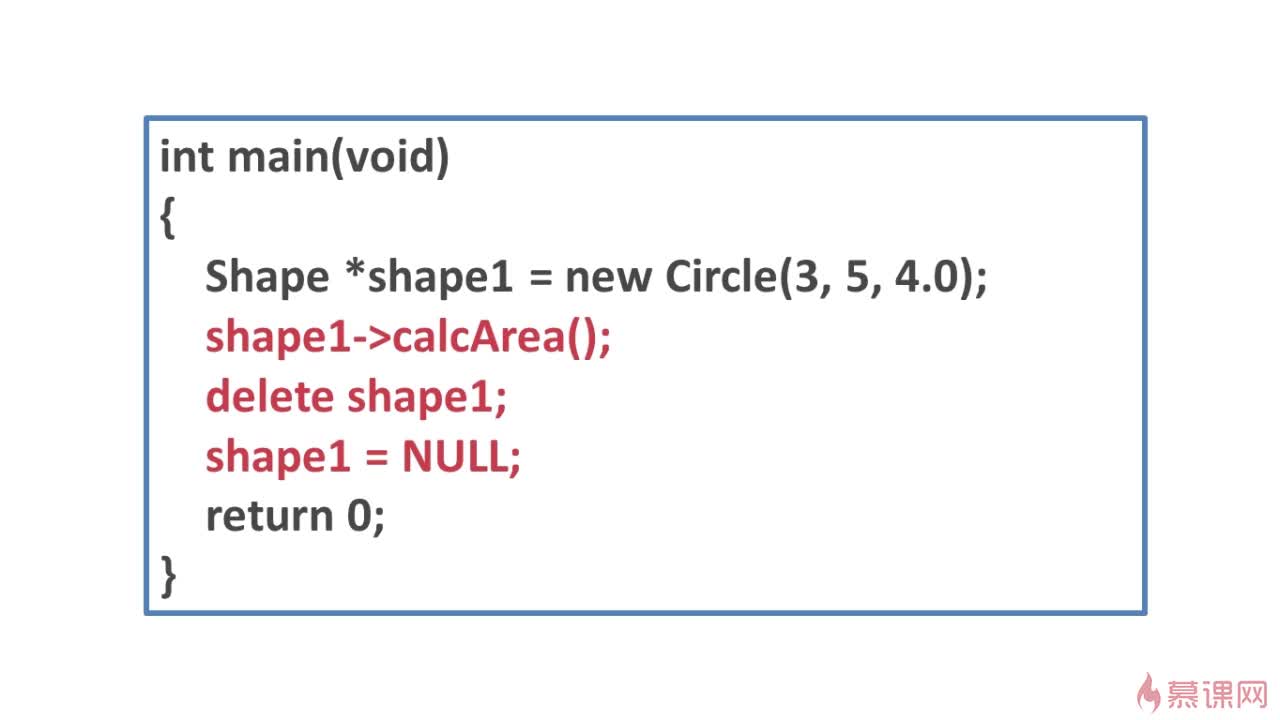
## 虚析构函数：

**（只有子类从堆中开辟内存的时候虚析构函数才有意义）**

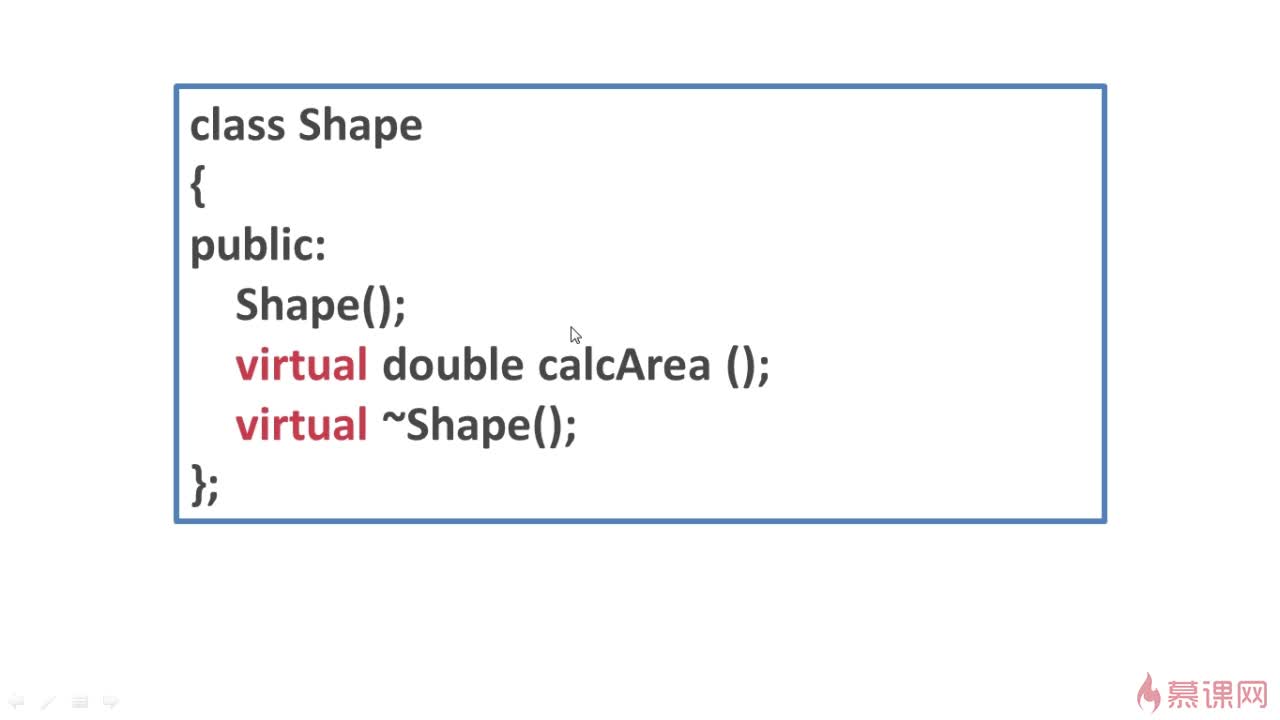
**例：**

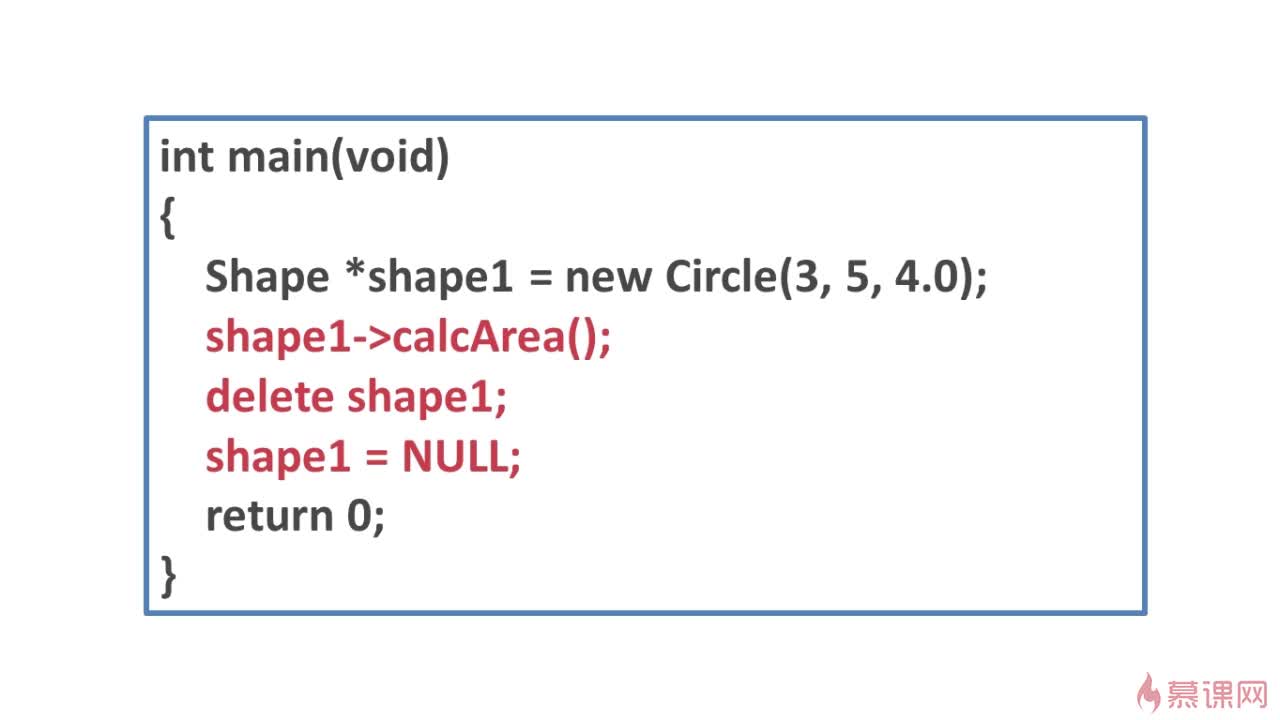
****

****

****

**在Circle这个类中，我们定义了一个Coordoninate类的指针m\_pCebter (Coordoninate类是提前定义好的点的类型，含有X、Y)，我们在circle的构造函数中使用new在堆中开辟了内存，并在析构函数中将开辟的内存释放掉，但是应该注意的是，在主函数中，如果使用父类的指针来意图执行circle类的析构函数的话，是达不到目的的，因为如果使用普通的指向子类的父类的指针，最后只会执行父类的析构函数，不会执行子类的析构函数，这就会造成内存的泄露，所以，我们必须把父类的析构函数定义成虚析构函数：**

****

****

**这样的话系统在最后执行完所有程序后才会先执行子类的虚构函数，再执行父类的析构函数。**

# Virtual关键字的使用限制：

## 1：普通函数不能是虚函数 ：

**析构函数--必须在类中，不能是全局函数**

****

## 2：静态成员函数不能是虚构函数

****

## 3：内联函数不能是虚函数

**--如果“inline virtual int eat(){}”**

**编译器会忽略掉 inline 使其失去内联的意义**

**下面这种使用方法是错误的：**

****

## 4：构造函数不能是虚函数：

****

**注意：只有虚析构函数，没有虚析构函数**

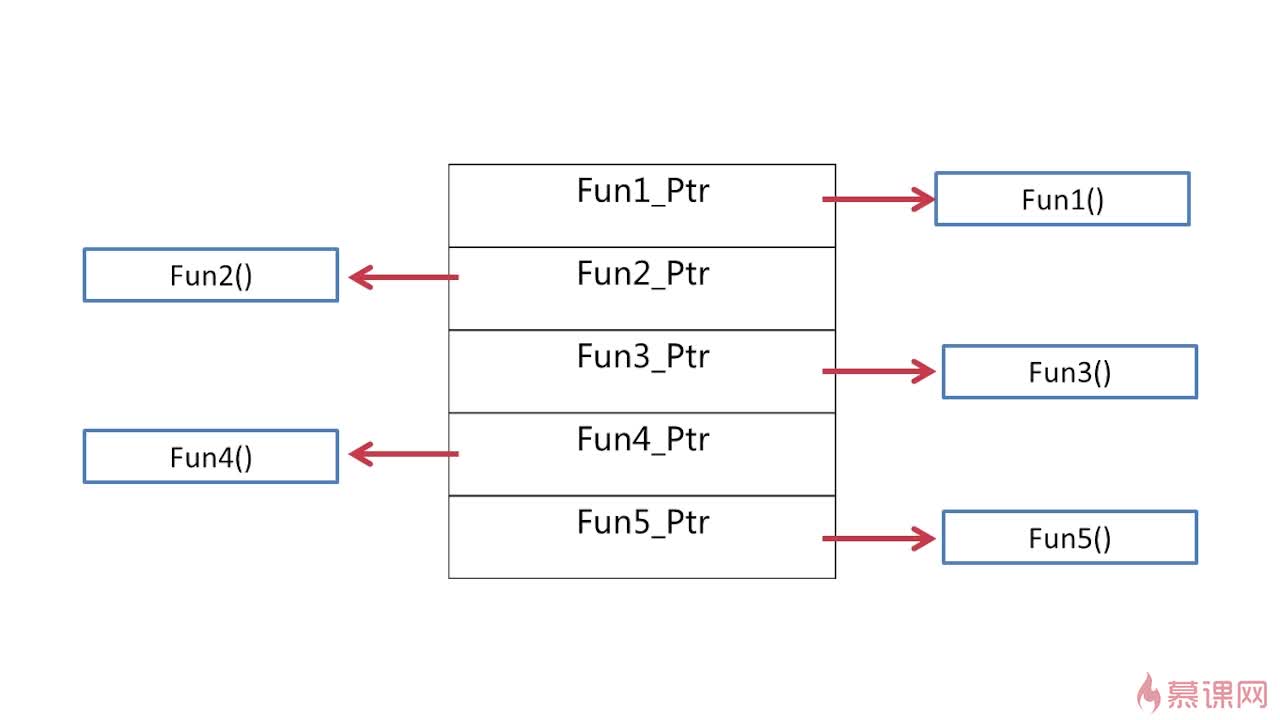
**虚函数使用virtual关键字定义，但使用virtual关键字时，并非全部是虚函数**

**虚函数特性可以被继承，当子类中定义的函数与父类中虚函数的声明相同时，该函数也是虚函数**

# 虚析构函数及虚函数的实现原理：

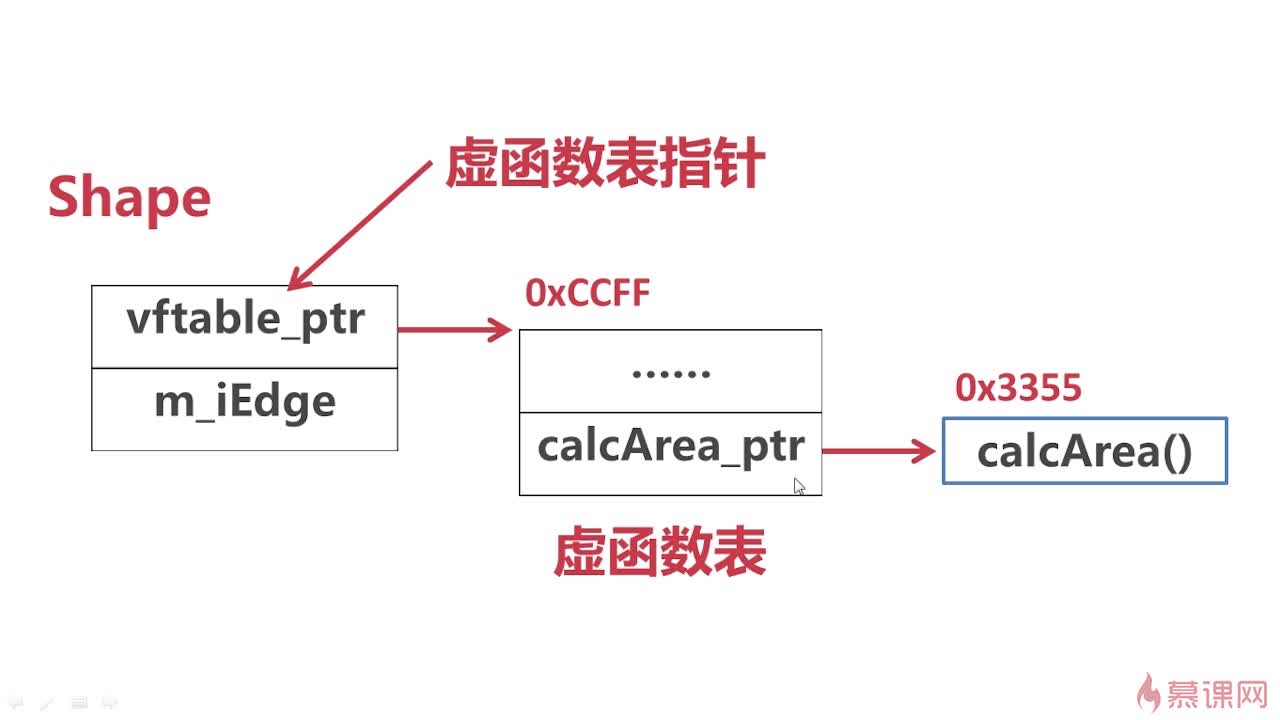
**C++中的多态是通过虚函数表来实现的**

**函数指针：**

****

**函数实际上就是一段储存在内存的二进制代码，可以通过指向该二进制码的开头的指针来调用他即函数指针。**

**虚函数实现原理：**

****

**解释：在实例化父类的时候如果父类中有虚函数系统会同时生成一个指向虚函数表的指针即虚函数表指针，在执行虚函数的时候先通过虚函数表指针找到对应的虚函数表，然后再通过虚函数表里的指向虚函数的指针即虚函数指针找到对应的存储虚函数的二进制代码的对应内存，即执行虚函数。**

**每一个类都有自己的虚函数表，该类的所有对象共用该虚函数表，来调用相应的虚函数**

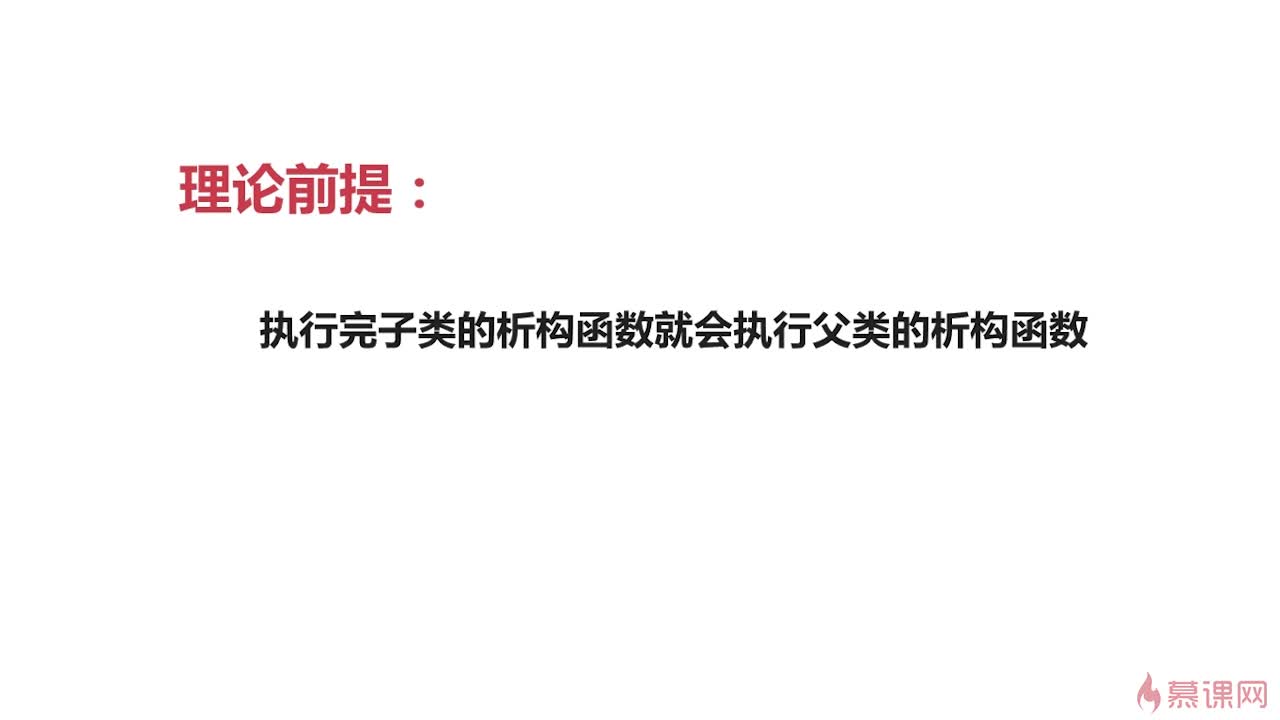
**两张虚函数表中的函数指针可能指向同一个函数。**

**关于多态的理解：**

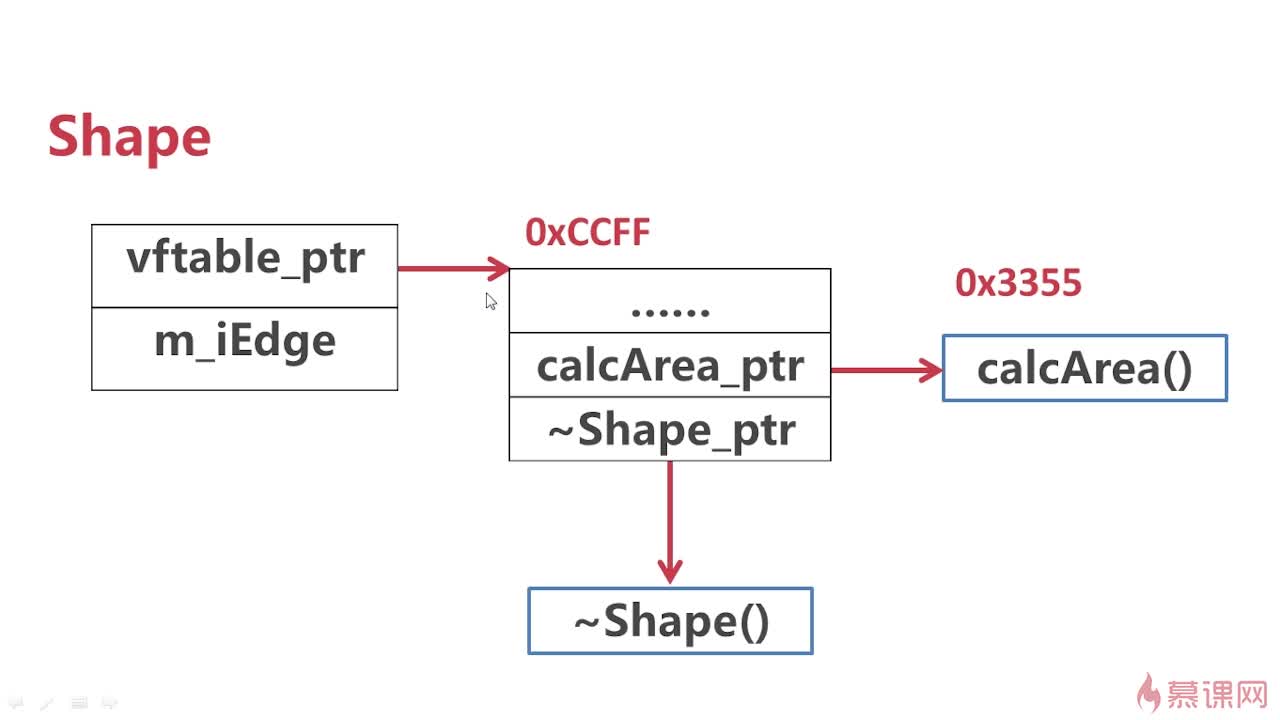
**当父类有虚函数时，编译系统会自动为该类配一个虚函数表，并将类中虚函数的地址写入这个表中，表的位置由类记录在隐含的虚函数表指针中，而且其每个实例化的对象均会共享这个表。其子类会继承该虚函数表，当然，子类的虚函数表的指针值是重新赋予的，与父类的不同。在子类的虚函数表中，既继承了父类的虚函数地址，也有子类自己的虚函数地址。此时子类中若是出现了与虚函数列表中父类函数同名的函数，则该子类函数会替换掉对应的父类函数（即函数的覆盖）。此时再通过父类指针调用同名函数时，就会调用到子类的函数。**

**这里有一点要注意到：虚函数只是将自己的地址复制到虚函数表中而已，其它则与别的成员函数并无二异。也就是说，虚函数有两个入口。而调用虚函数默认先从虚函数表中调用。所以这就说明了为什么父类指针可以调用到同名的子类函数：因为虚函数表中原来的父类函数地址被子类函数覆盖掉了。当然，同名的另一个父类函数还是存在的，只不过虚函数表中没有它的地址而已。可以通过另一个入口调用。**

**实现虚析构函数的理论前提：**

****

**析构函数的实现原理：**

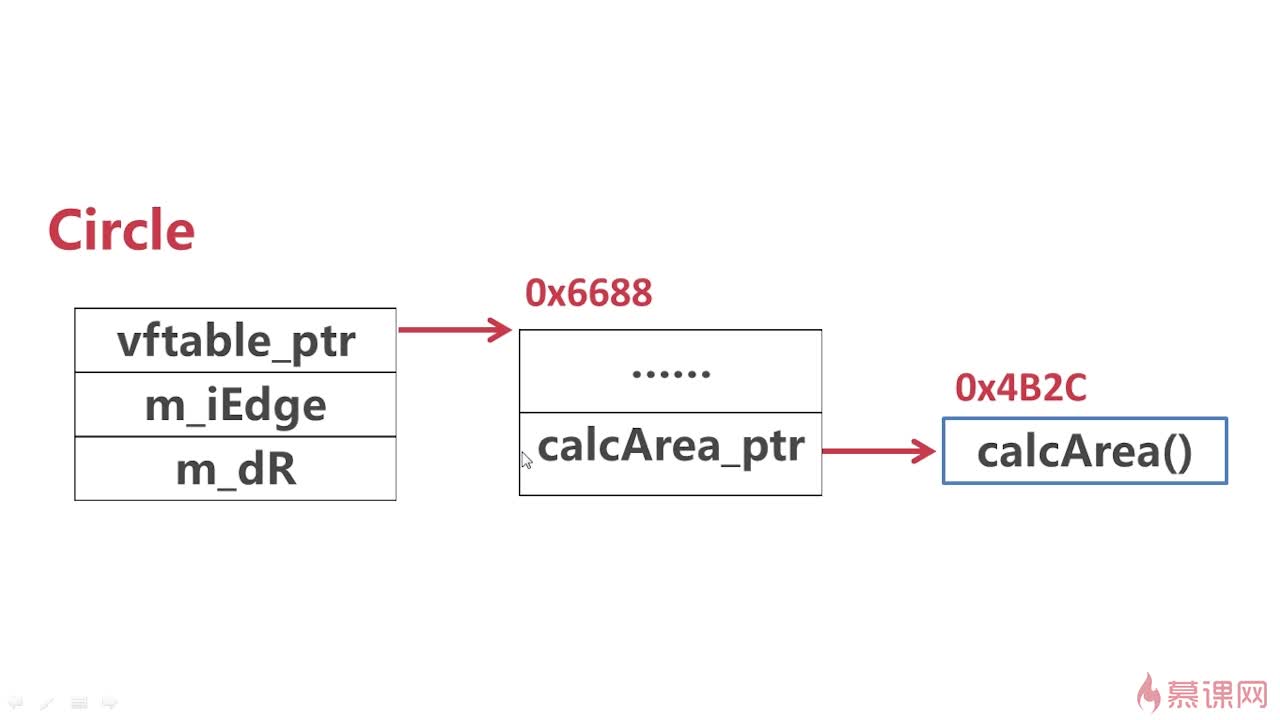
****

**首先，系统通过父类的虚函数表指针找到父类的虚函数表，然后在虚函数表中找到对应的指向子类的虚析构函数的函数指针，然后再通过虚析构函数指针找到子类的虚析构函数即执行子类的析构函数，执行完子类的虚析构函数之后系统会自动执行父类的虚析构函数。**

**这个面试的时候经常考哦：**

**函数的隐藏：在没有多态的父类与子类中，存在同名函数，则会出现函数的隐藏。**

**函数的覆盖：在有多态的父类与子类中，在子类中，同名的函数会覆盖掉从父类继承来的同名函数。**

****

# 几个概念：

## 对象的大小：

**在类实例化的对象中，数据成员所占据的内存大小（不包括成员函数）**

## 对象的地址：

**通过一个类，实例化一个对象，对象在内存中占有一定的内存单元，内存单元的第一个地址就是对象的地址**

## 对象成员的指针：

**用类实例化一个对象之后，对象当中可能有一个或多个数据成员，对象的每个数据成员的类型不同，占据的内存大小也有不同，地址也不同**

**虚函数表指针：具有虚函数的情况下，实例化对象时，对象的第一块内存中存储的是一个指针，为虚函数表指针，占据大小为4。**

**对象的大小：指的是数据对象所占内存的大小，不包括成员函数**

**虚函数表的指针在有虚函数的时候并且实例化对象的时候，这个对象的第一块内存当中所存储的是一个指针，就是虚函数表的指针**

**某类中一个对象成员都没有的时候，这个类也是可以实例化的，实例化之后，作为一个对象来说，它必须标明自己的存在，C++用一个内存单元去标记它（即该对象的大小就为1），即这个内存单元只为标记这个对象的存在。如果有数据成员，数据成员的内存就能够标记自己的存在，就不需要额外的内存单元去标记了。**

**即使没有数据成员，加入virtual之后，实例化对象，就会有虚函数表指针，该指针占4个基本内存单元，就不像之前只占一个基本内存单元了**

**虚函数表指针在对象当中的前4个基本内存单元**

**有虚函数时，先使用指针 p 指向对象，访问的虚函数表指针，（\*p）取得的是虚函数表地址，再使 p++ 就指向了数据成员，（\*p）就能取得数据成员的值。**

# 纯虚函数和抽象类：

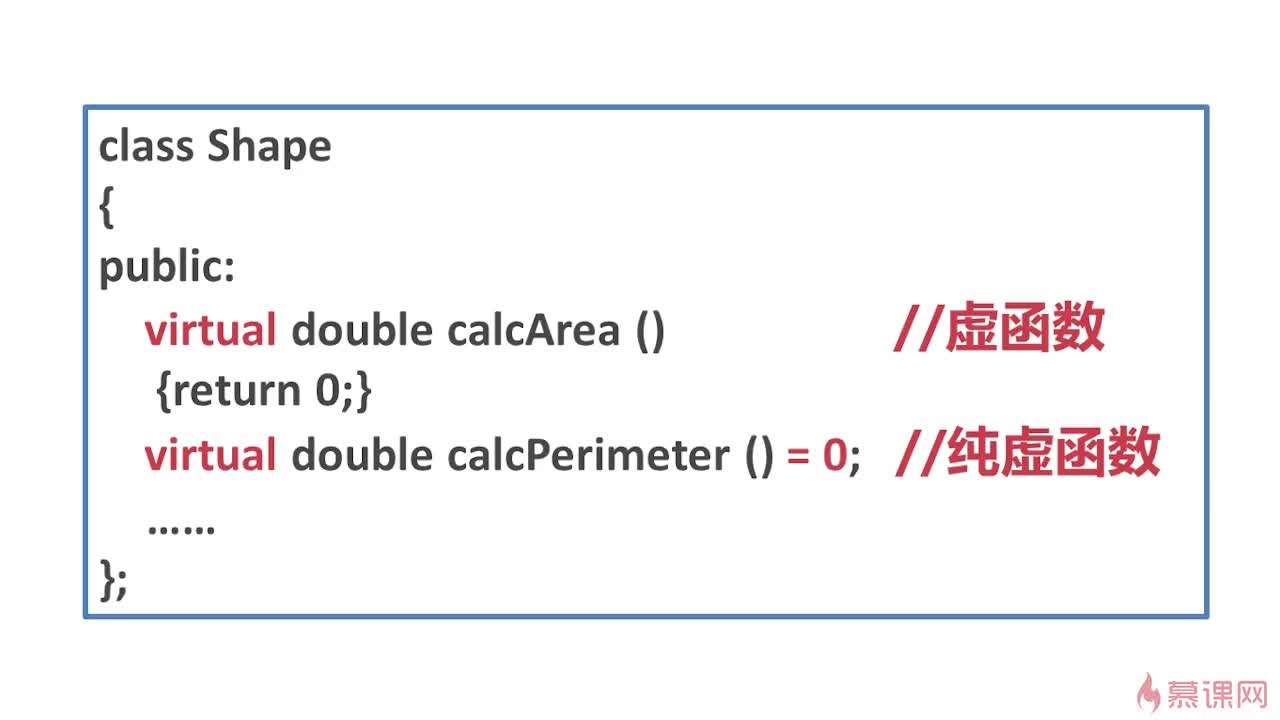
**为什么需要纯虚函数？**

**因为在有的父类当中的函数所要达到的目的比较抽象，实现起来比较难，比如：**

**定义一个person类，类中有一个work函数，用来实现对其工作的相关操作，但是，作为一个人来说他的工作很抽象啊，他的工作是不确定的，所以就没法直接定义，我们可以让一个worker类继承他，即worker:public person，这样在woeker中也会继承下来work，但是我们发现工人这个类的工作也比较抽象，那么我们可以再定义一个dustman即清洁工的类来继承worker这个类，这样清洁工的工作是确定的（扫地），我们就可以在清洁工这个类中对work函数进行具体的操作和修改，这样就实现了抽象类到具体类的转化。**

**”**

## 纯虚函数：

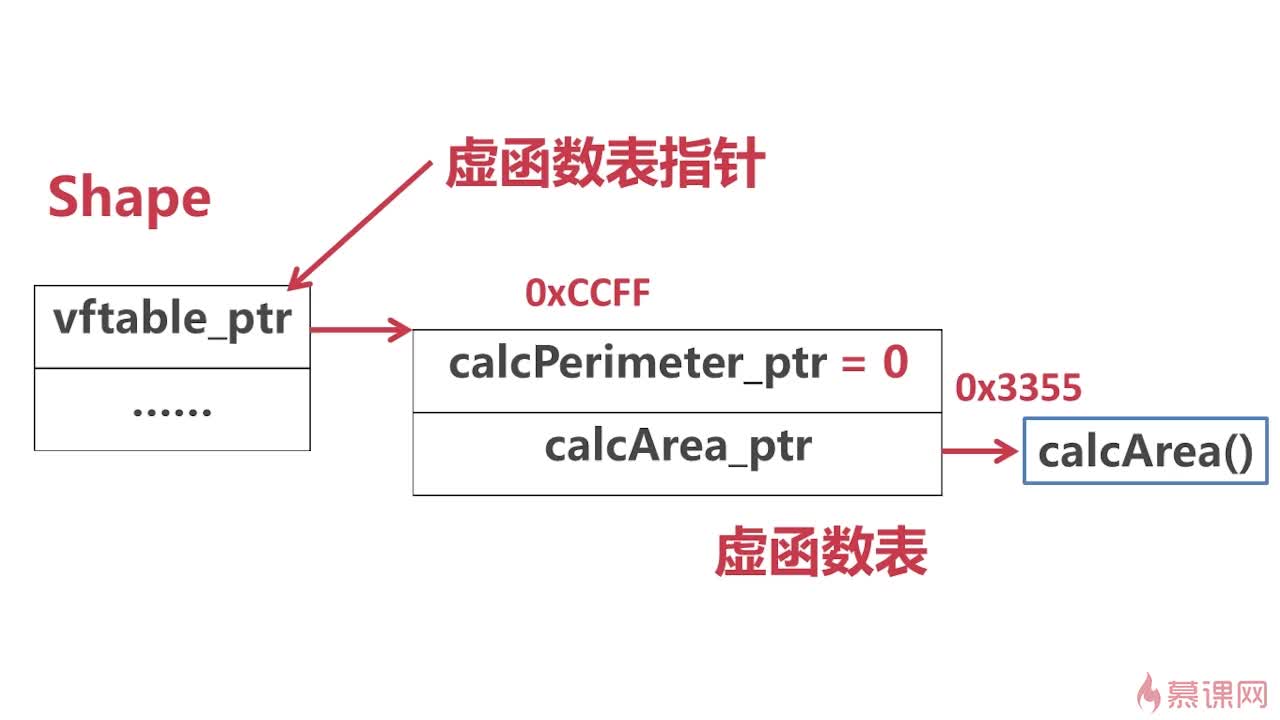
****

**纯虚函数的特点（用法）：**

**（1）：没有函数体；**

**（2）：函数声明后加“=0”；**

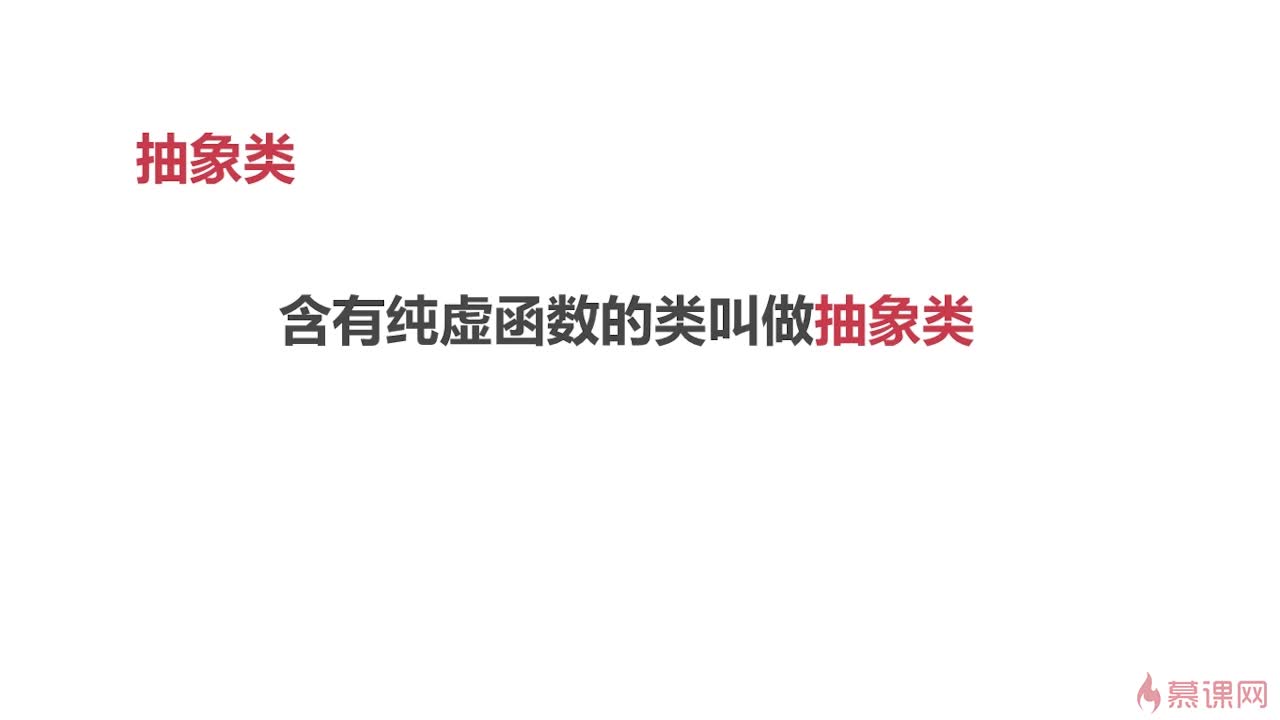
**纯虚函数的虚函数表指针：**

****

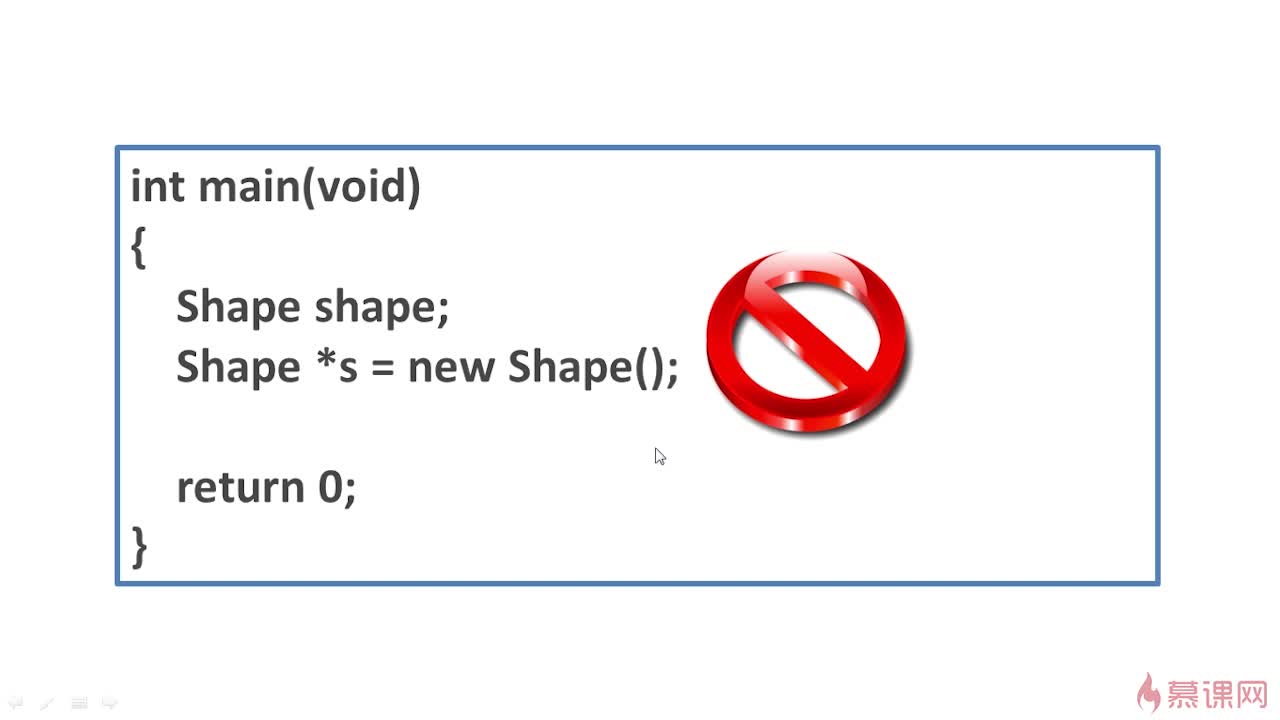
**可见，虚函数表里的纯虚函数得指针值为0即不指向任何数据**

## 抽象类：

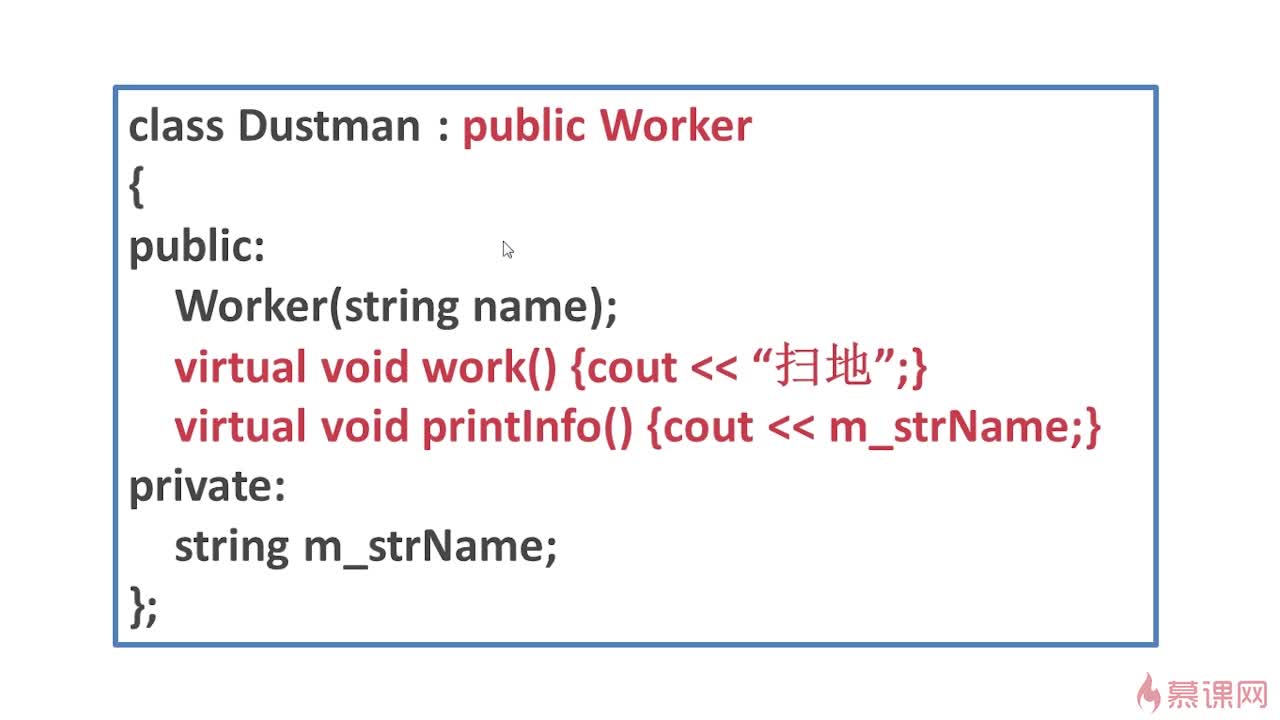
**含有纯虚函数的类叫做抽象类**

****

**抽象类不允许实例化对象：**

****

**在抽象类当中，抽象类的子类也有可能是抽象类，抽象类的子类只有把父类中的纯虚函数做了实现(即用非纯虚函数覆盖了从父类继承而来的纯虚函数）时，才是非抽象类，才能实例化对象。**

****

**纯虚函数无法调用，所以抽象类无法实例化对象**

**抽象基类是本类族的公共接口，例如可通过抽象基类的指针访问到任何子类的虚函数，并对这些函数下达同一命令，从而实现多态性。**

**若是子类并没有对父类中的纯虚函数进行定义，则该子类仍然是抽象基类。**

**父类是抽象类，且纯虚函数是公有的，子类公有继承下来的时候，若没有现实跟父类同名的纯虚函数，则也会继承父类的纯虚函数，此时的子类也是一个抽象类，也不能实例化对象。**

**只有函数声明没有函数定义的虚函数是纯虚函数。**

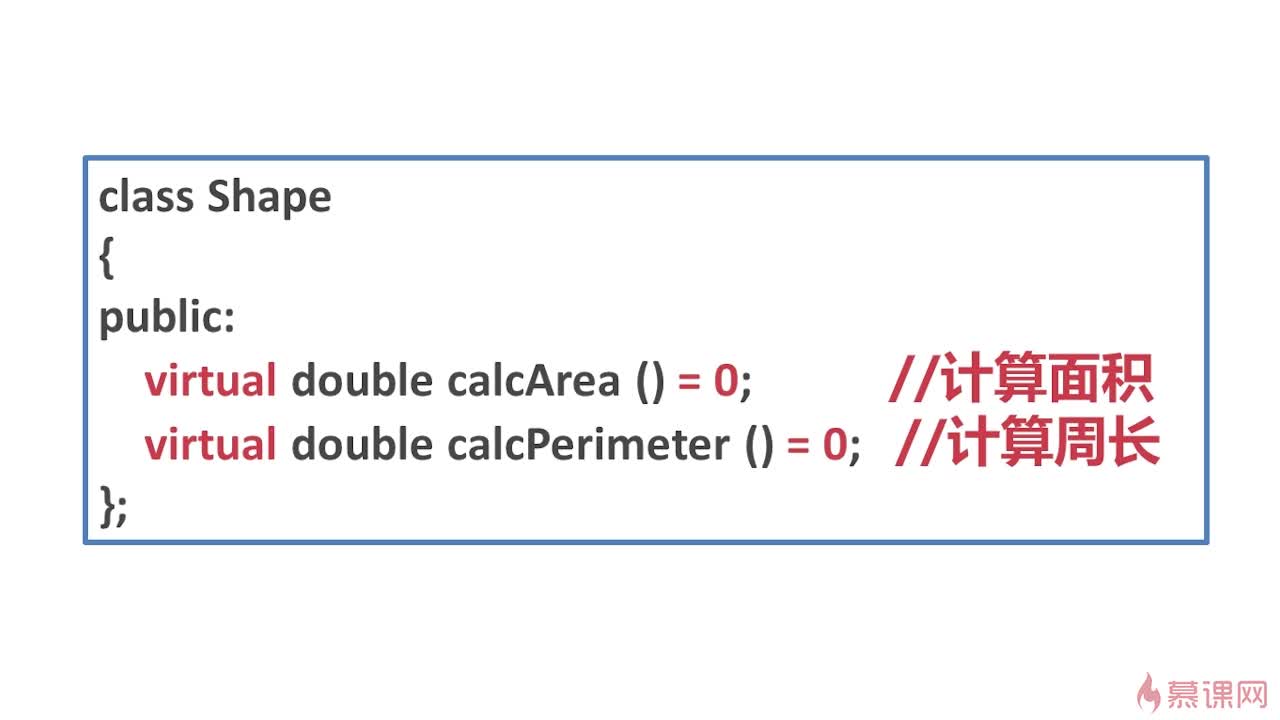
**含有纯虚函数的类叫做抽象类**

# 接口类

**接口类：在抽象类当中，仅含有纯虚函数（接口类不需要写构造函数和析构函数，只写纯虚函数，且不用cpp文件），而不含有别的东西，就叫做接口类**

****

**比如：**

****

**Shape就是一个接口类。**

**接口类更多的是飙到一种能力或协议（用接口类做函数形参，可以限制传入参数的类型，限制了传入的参数类型，只有拥有接口类定义的函数功能的对象才能作为实参，可以放心的调用接口类的所有成员函数。）**

**接口类实例：**

**#include<iostream>**

**#include<stdlib.h>**

**#include"FighterPlane.h"**

**using namespace std;**

**void flyMatch(Flyable \*f1,Flyable \*f2)**

**{**

**f1->takeoff ();**

**f1->land ();**

**f2->takeoff ();**

**f2->land ();**

**}**

**int main(void)**

**{**

**FighterPlane p1("001");**

**FighterPlane p2("002");**

**p1.printCode ();**

**p2.printCode ();**

**flyMatch(&p1,&p2);**

**return 0;**

**}**

**函数限制了传入的参数的参数体类型，并且在函数中可以放心的调用接口中定义的纯虚函数。这就是接口的最常用法**

**可以使用接口类指针指向其子类对象，并调用子类对象中实现的接口类中纯虚函数。**

**一个类可以继承一个接口类，也可以继承多个接口类。**

**一个类可以继承接口类的同时也继承非接口类。**

**接口类可以被继承，不能实例化**

**接口类也是抽象类**

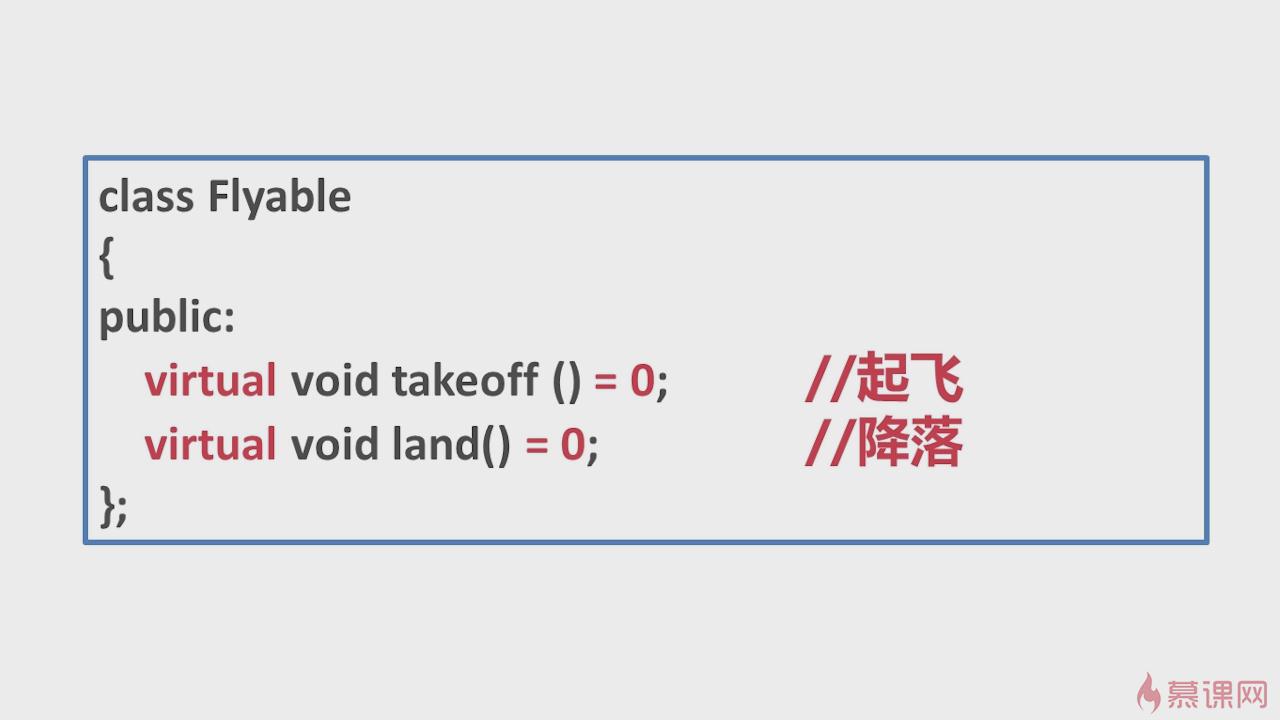
# 运行时类型识别（RTTI：Run\_Time Type Identification）

**typeid**

**dynamic\_cast**

**例：**

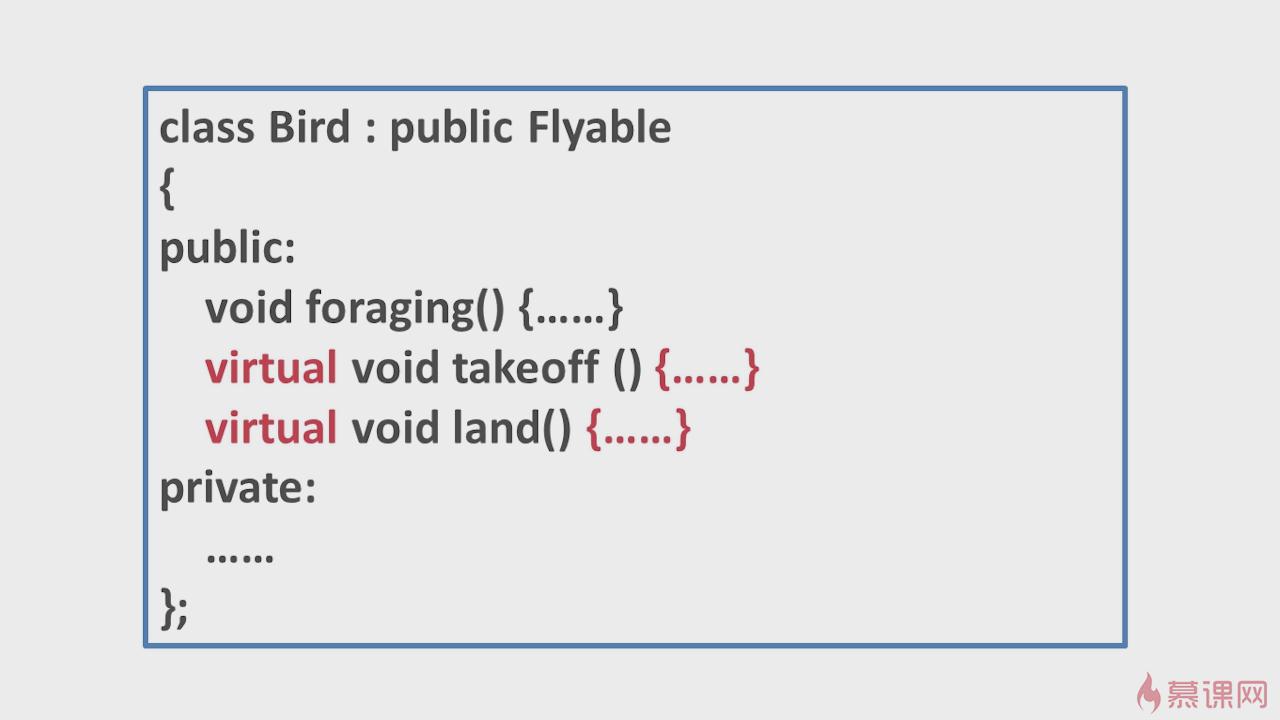
**定义一个基类：**

****

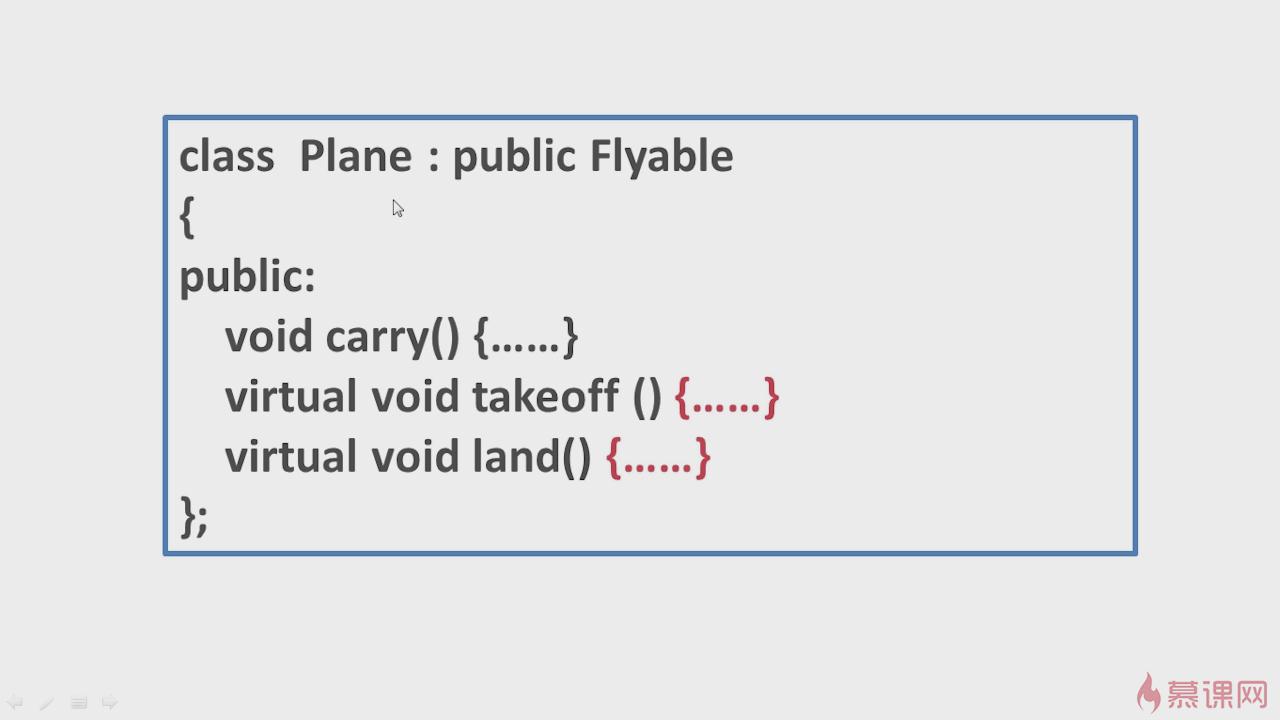
**基类（能飞）有两个函数：起飞和降落**

**定义两个派生类鸟和飞机：**

**对于鸟来说，它除了能飞之外还有一个特定的属性“觅食”：**

****

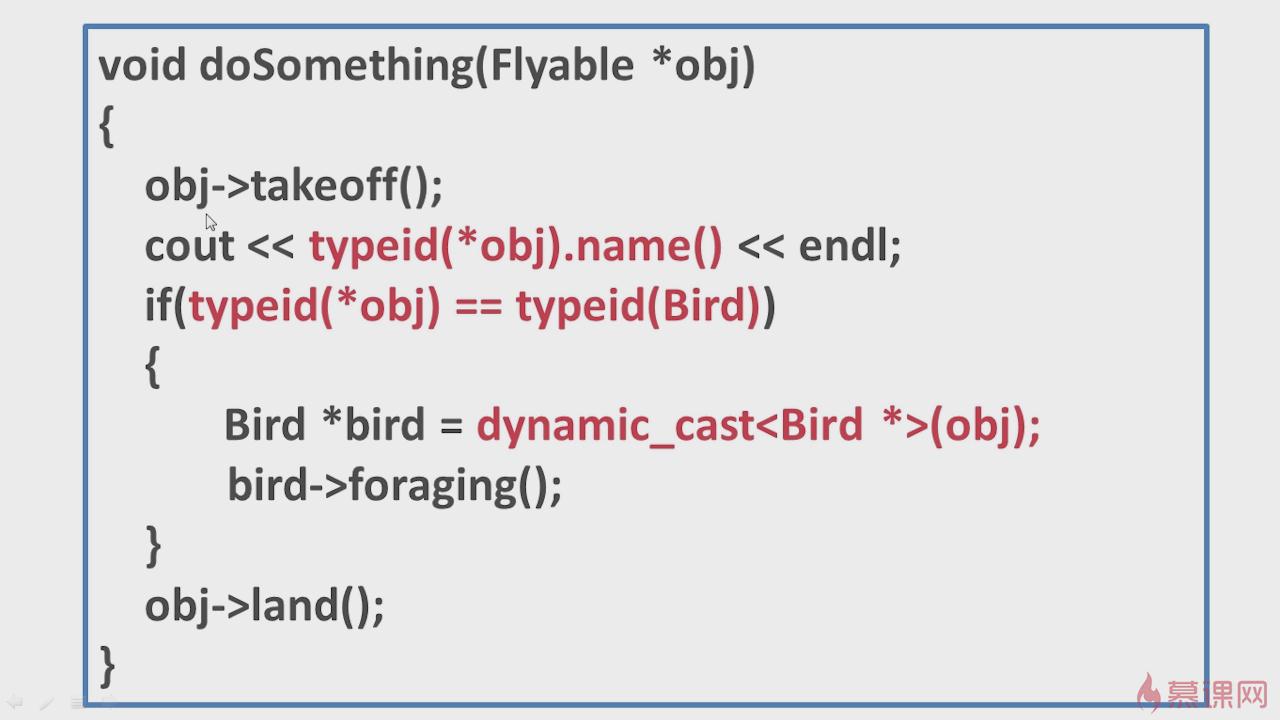
**对于飞机来说除了能飞之外还有一个特定的属性“运输”：**

****

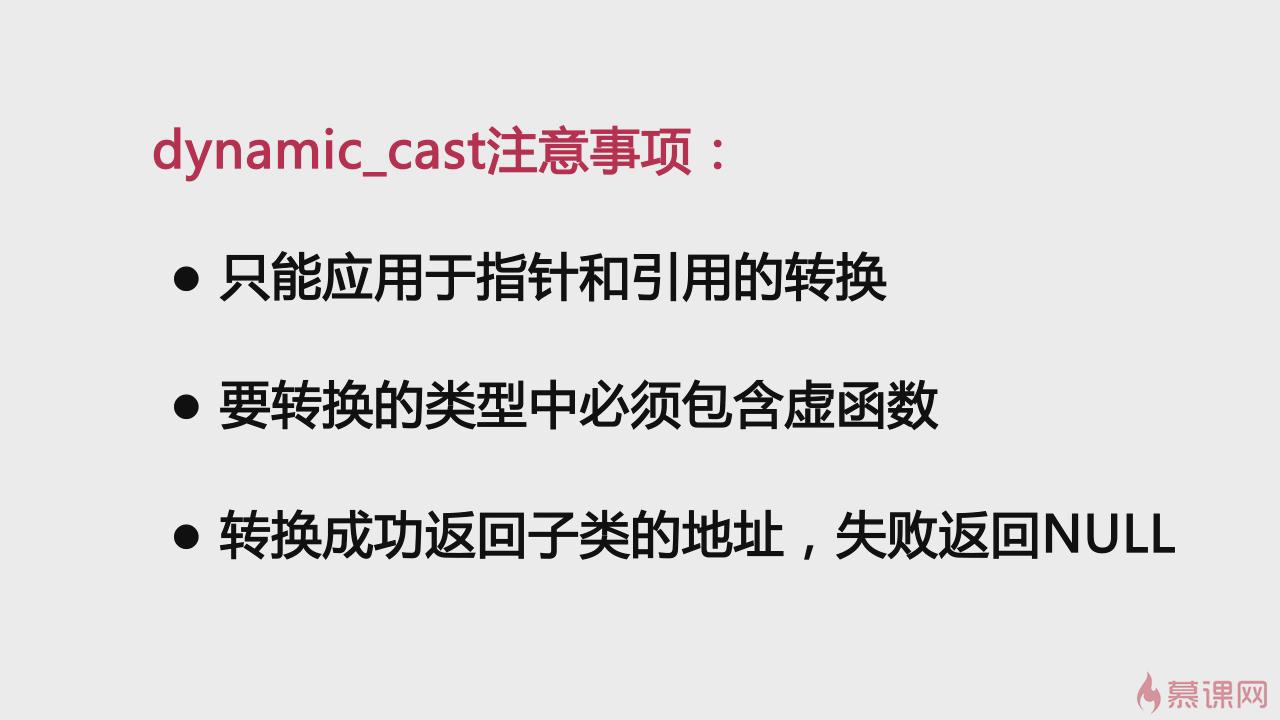
**现在我们定义一个函数：在这个函数中我们想传入一个Flyable的指针，首先要分别执行传入指针的起飞和降落函数，其次，我们希望如果传入的指针是鸟这个类型的指针，那么就调用鸟的特定函数觅食，如果是飞机这个类型那就要调用飞机的运输函数：**

**所以我们要使用typeid和dynamic\_cast来进行实现，这两个运算符的作用是：**

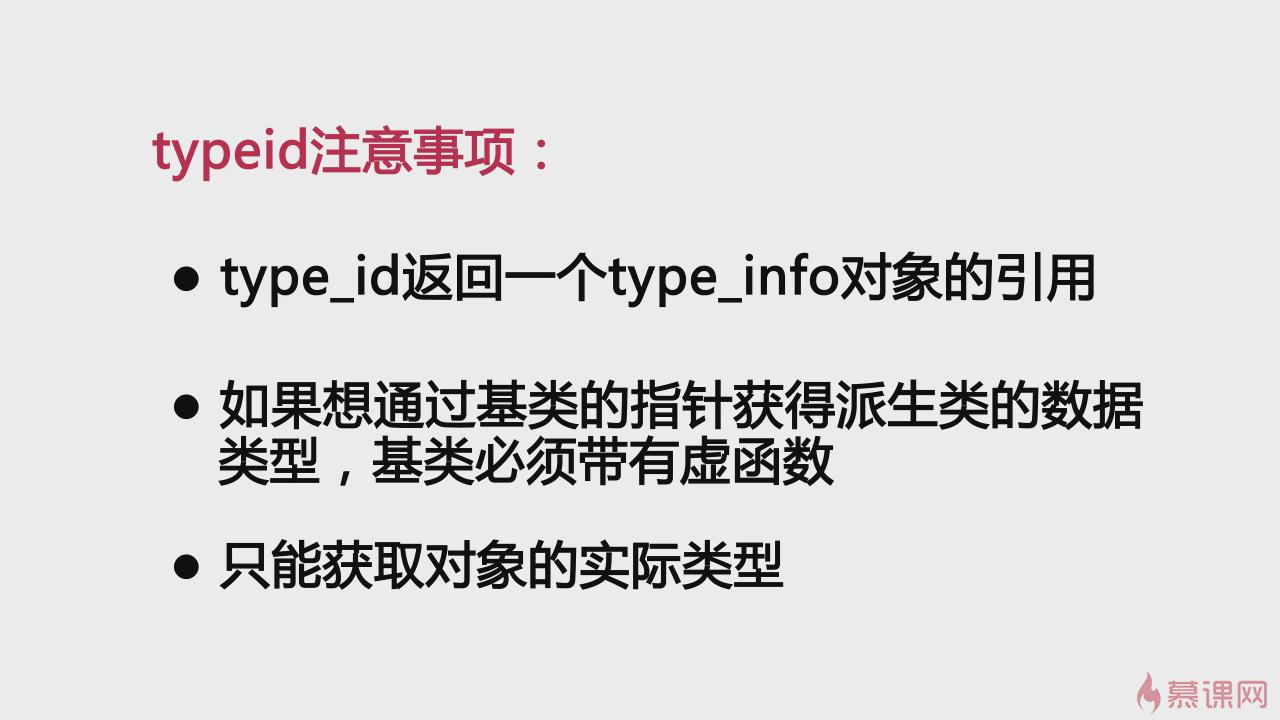
**typeid用来获取（判断）指针或者引用的类型，dynamic\_cast用来将基类的指针或引用转化为对应的派生类的指针或引用：**

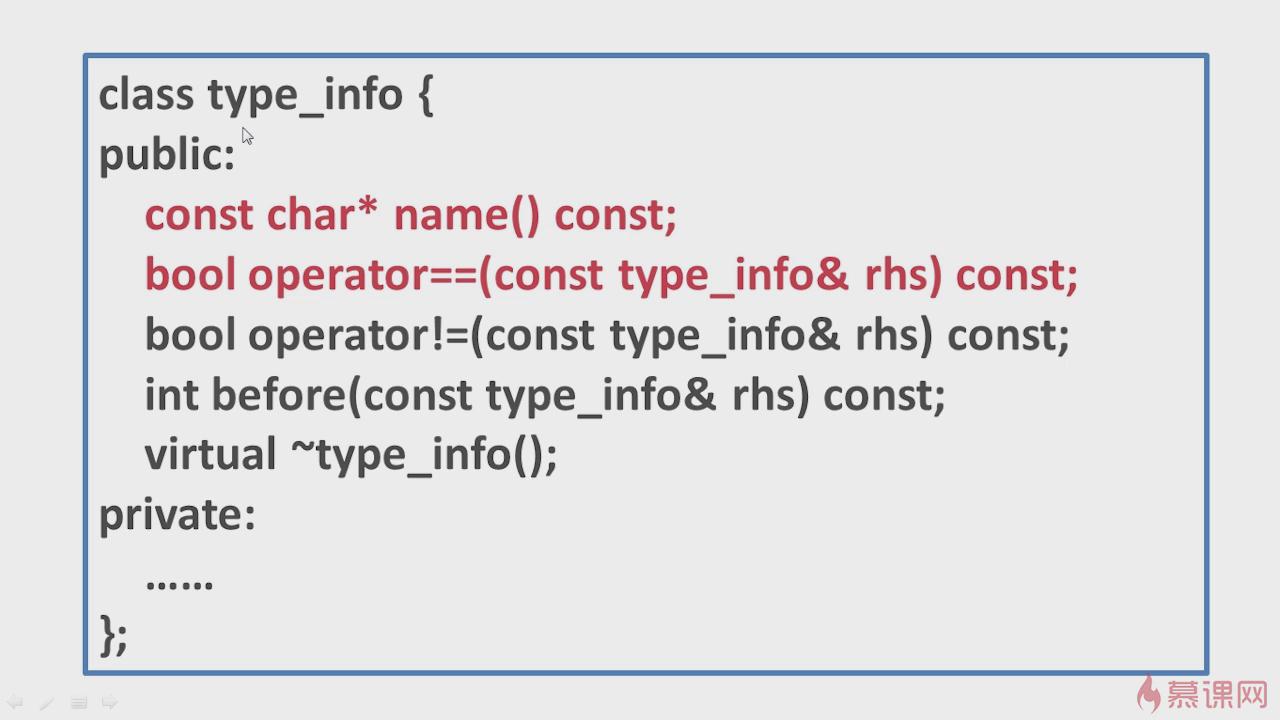
****

## Dynamic-cast的注意事项：

****

## Typeid的注意事项：

**使用typeid必须包含头文件typeinfo**

**type\_id返回的type\_info的解释：**

**其实typeinfo就是一个类，类中实现了typeid的目的，最重要的代码是红色的那两行，第一行定义了一个name指针用来获取对应指针或引用的类型，第二行用来判断指针类型之间是否相等。**

# 异常处理：

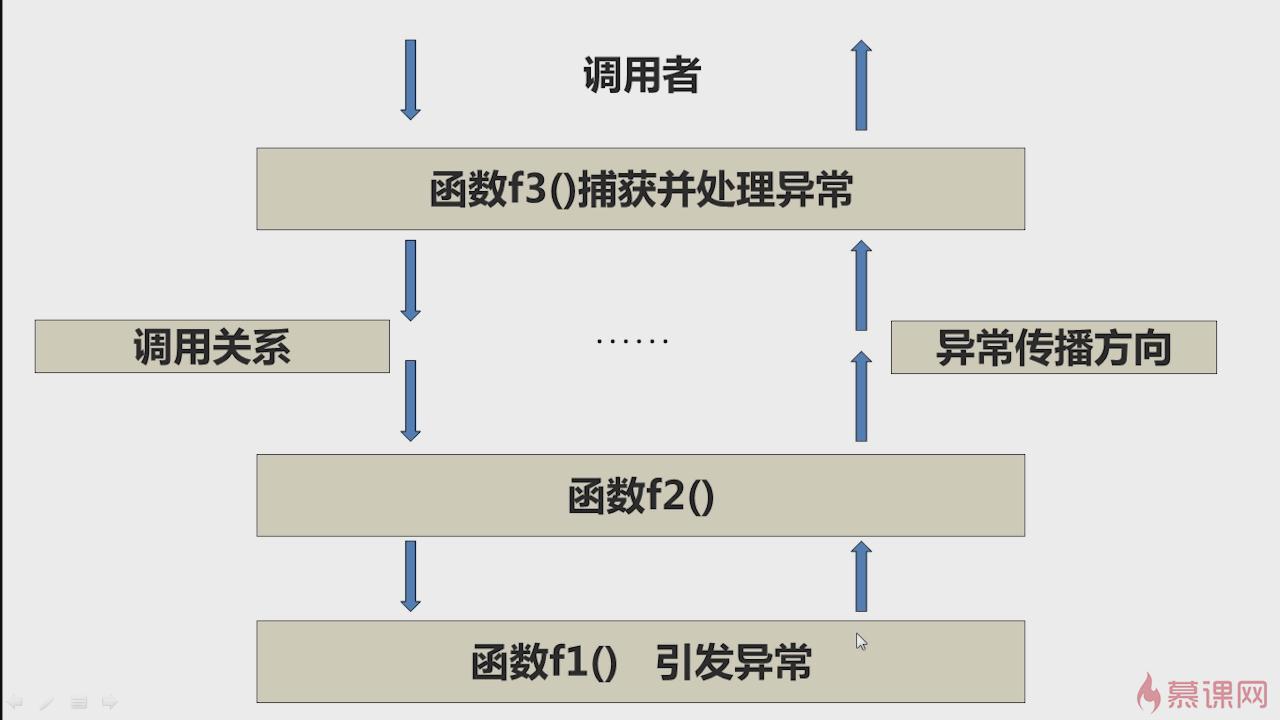
**异常：程序运行期出现的错误**

**异常处理：对有可能发生异常的地方作出可见性的安排**

**异常处理的基本思想：主逻辑与异常处理分离**

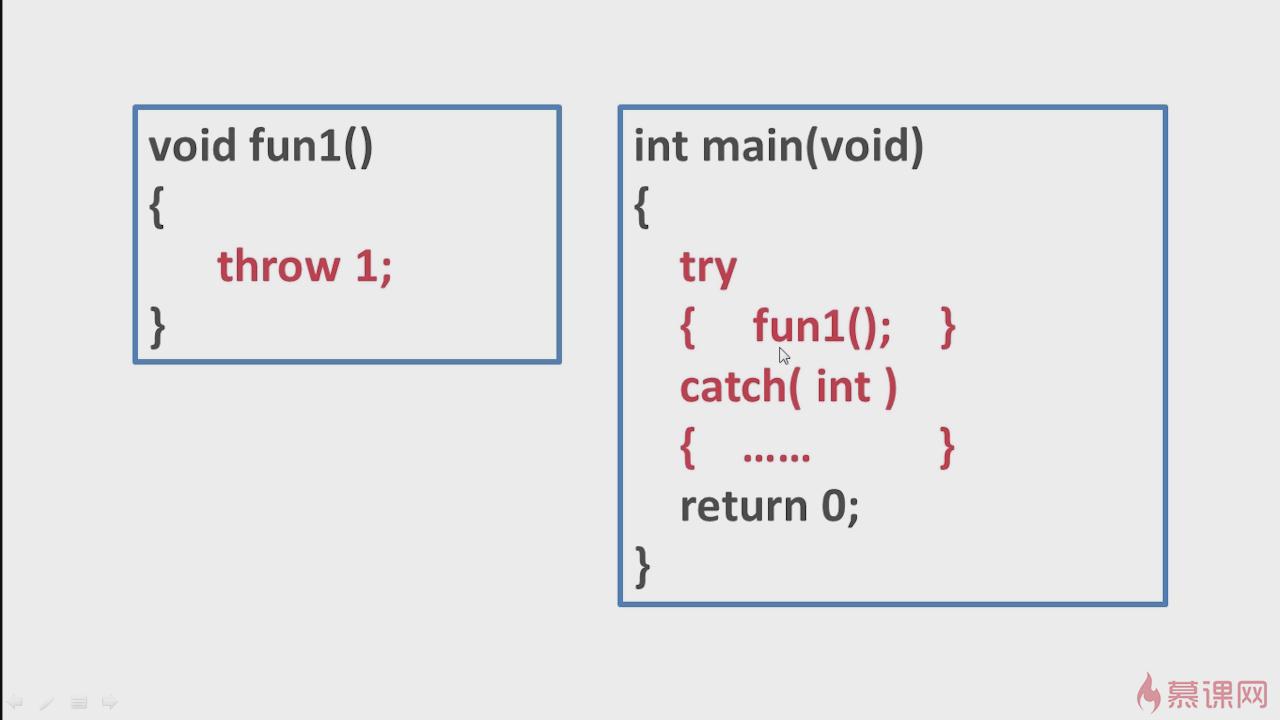
**关键字：**

****

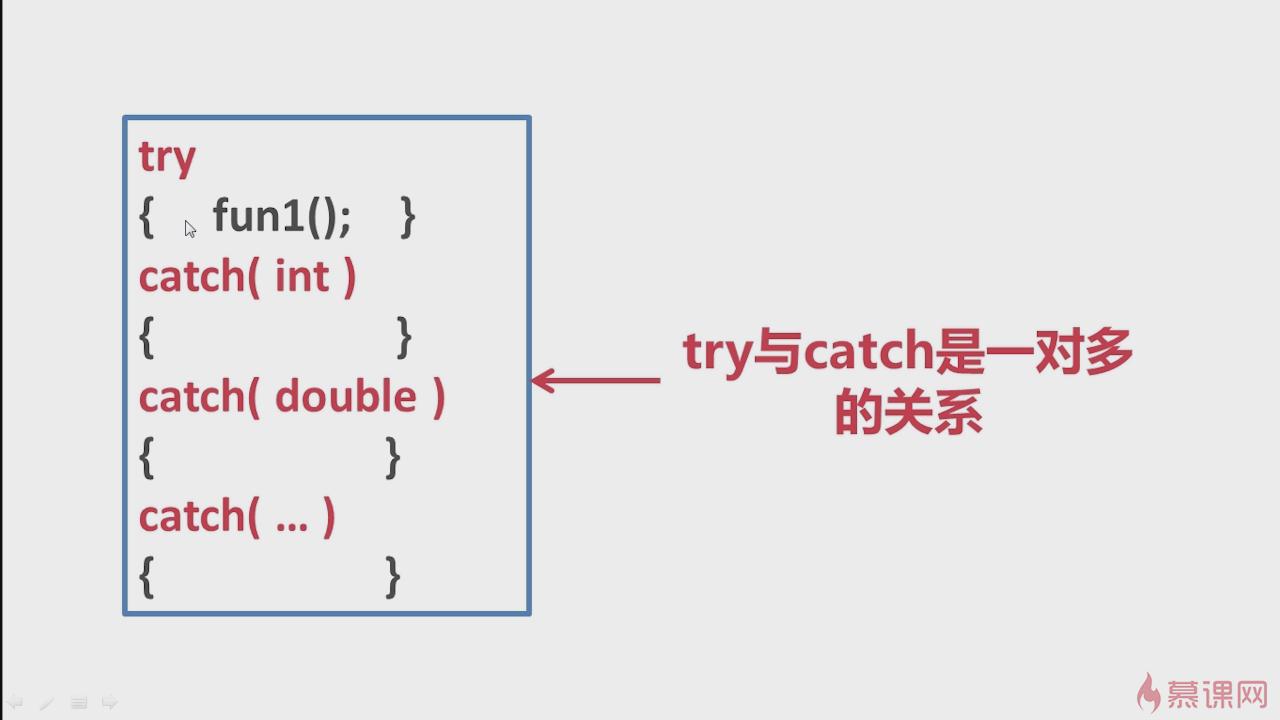
****

**如上图，f1函数发生异常之后他就会把异常信息反馈给调用他的f2函数，如说f2可以处理该异常，则f2对该异常进行处理，如果f2不能进行处理，那么就再将该异常信息反馈给更高级的f3函数，以此类推，如果反馈到最高级的函数后仍然无法进行处理，那么该错误就会抛给系统，系统就会崩溃。**

**例：**

****

**try和catch可以是一对多的关系：**

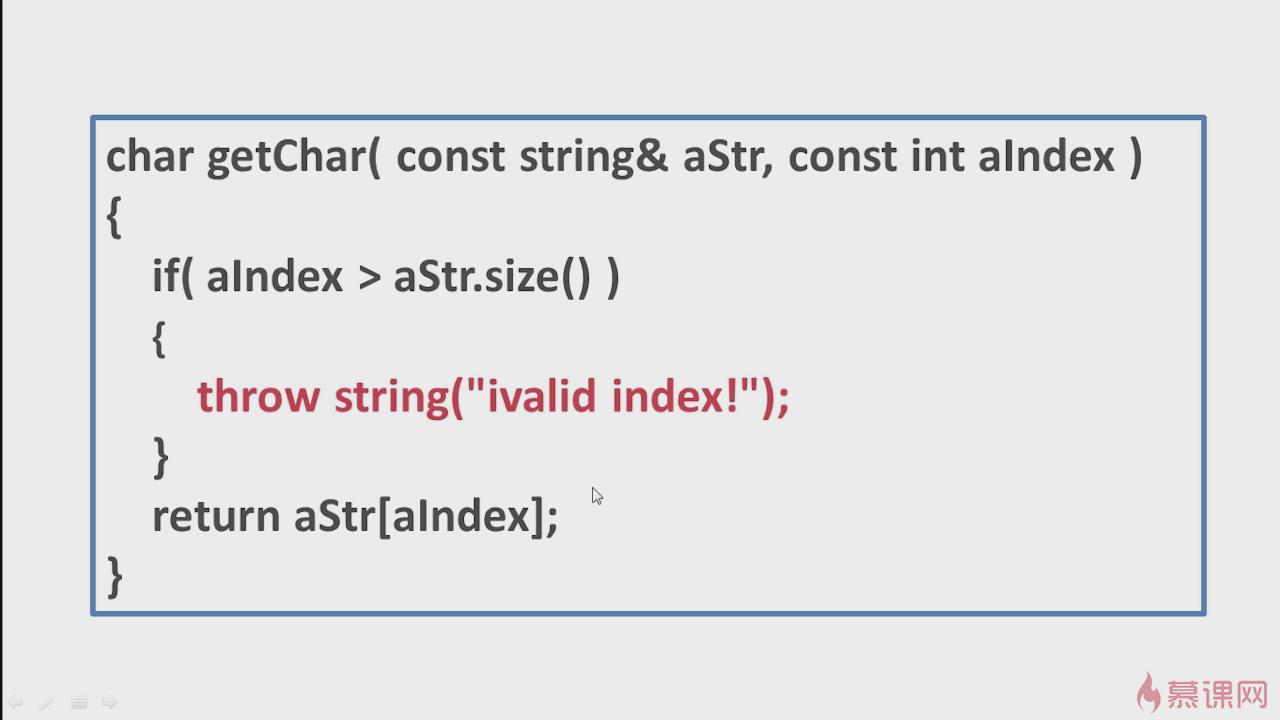
****

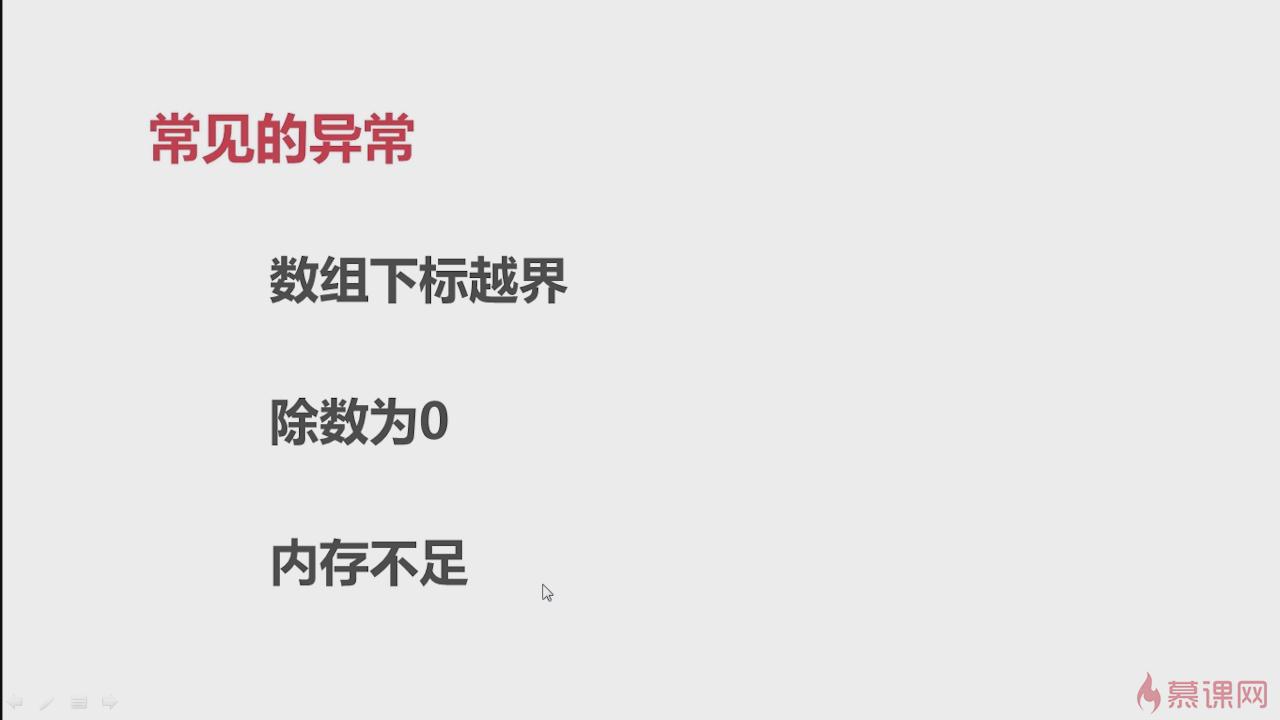
**注意这个例子中的最后一个catch的写法：**

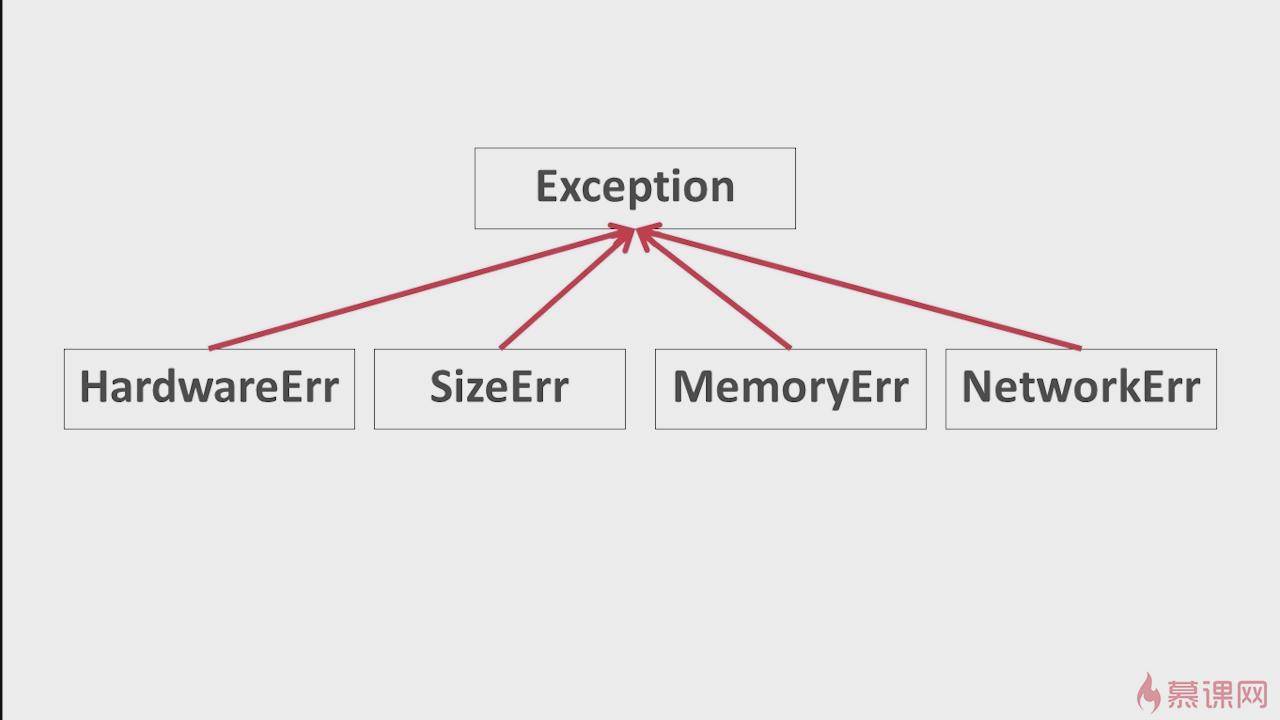
**catch(…){}**

**这个语句的作用是当上面所有catch都没有捕获到throw的值的话，就把该异常当做最后一类一场进行处理即执行最后一个catch里面的函数语句，注意这个语句中catch的括号里面写的是三个点“.”。**

**例：下面这个函数是要返回给定数组的特定字符，但是当给定一个数组比如a【10】，却要求获得a【299】，这样就会出现数组越界，这样的话就让getchar函数抛出“ivalid index！ ”提示错误，而在主函数中用catch捕获上面【】抛出的“ivalid index！ ”然后将它打印出来，即提示用户错误原因：**

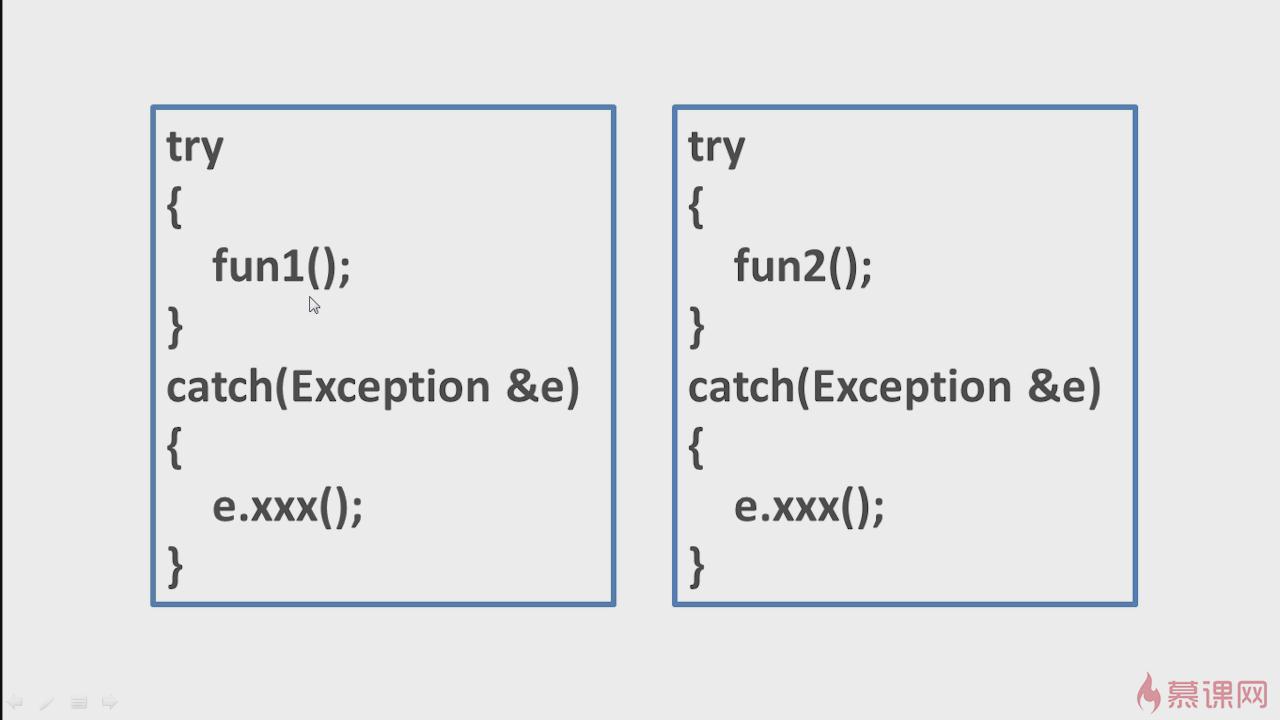
****

常见的异常：****异常处理与多态的关系：

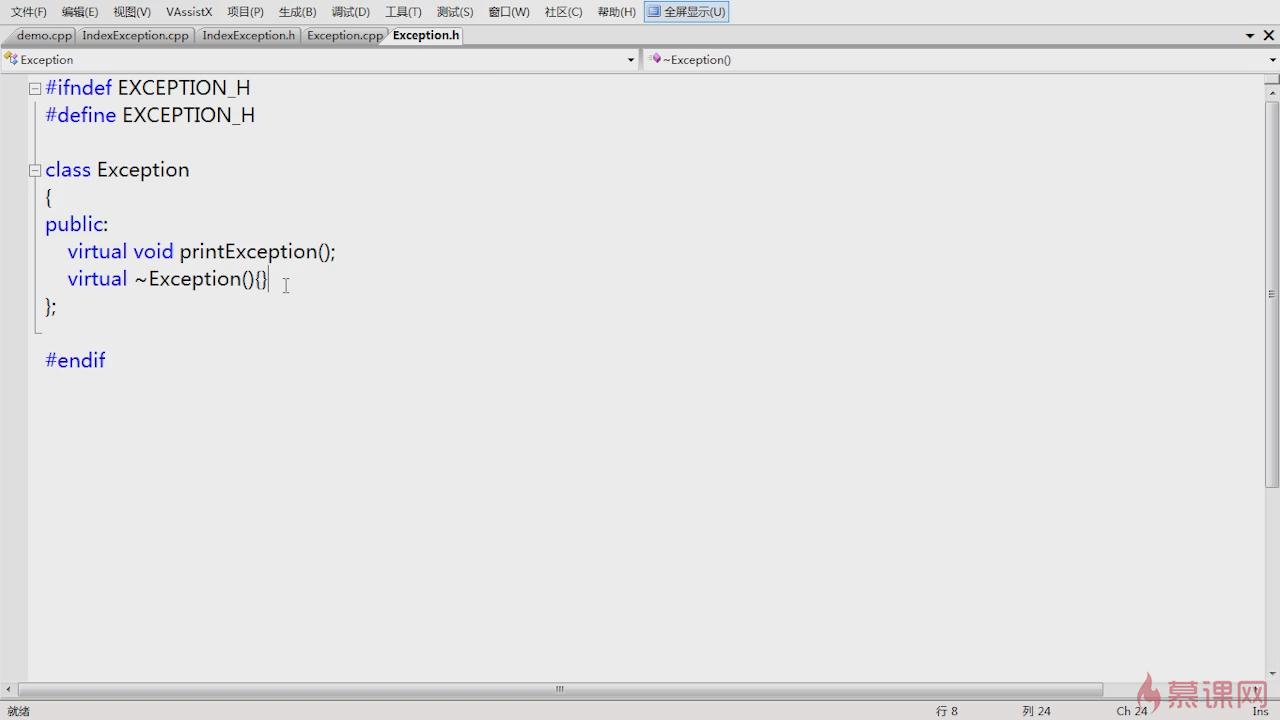
**如下图，我们定义一个异常处理的类，我们把它定义成为一个接口类，然后定义他的子类，在这些子类中打印不同的错误信息，当我们抛出这些子类的对象的时候，就能只用一个Exception父类来捕获所有的被抛出子类**

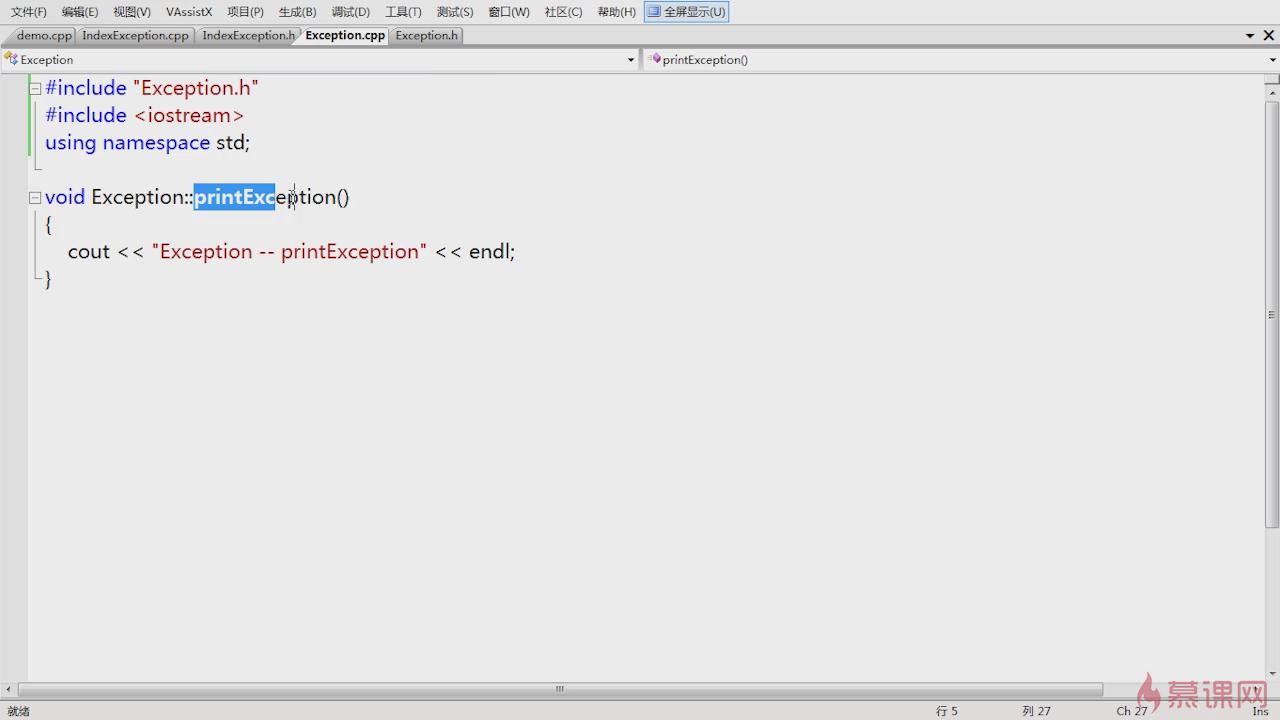
**例：如下图，在fun1中，我们抛出一个SizeErr()的处理错误的类，在fun2中，我们抛出一个MemoryErr（）的处理错误的类**

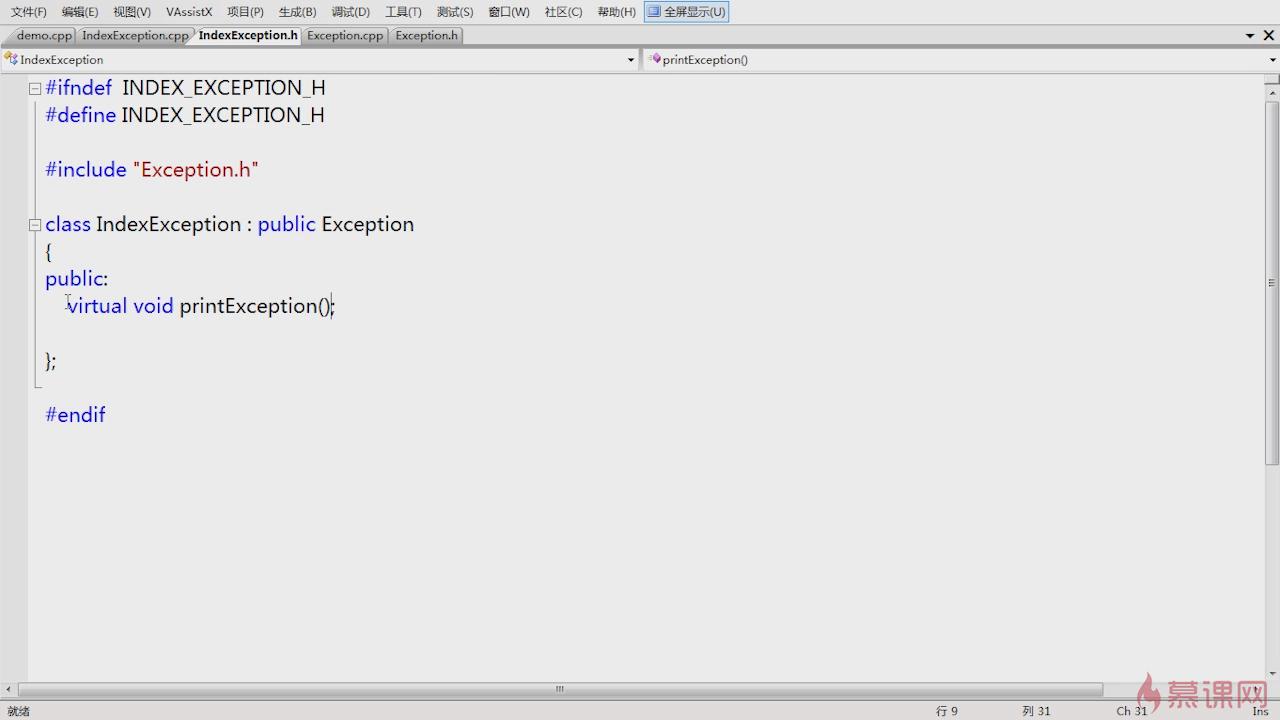
**如下图，不管是对fun1还是对fun2进行异常处理，我们都只需要用被抛出类的父类来捕获抛出的类，并且通过子类对象带相应的虚函数：**

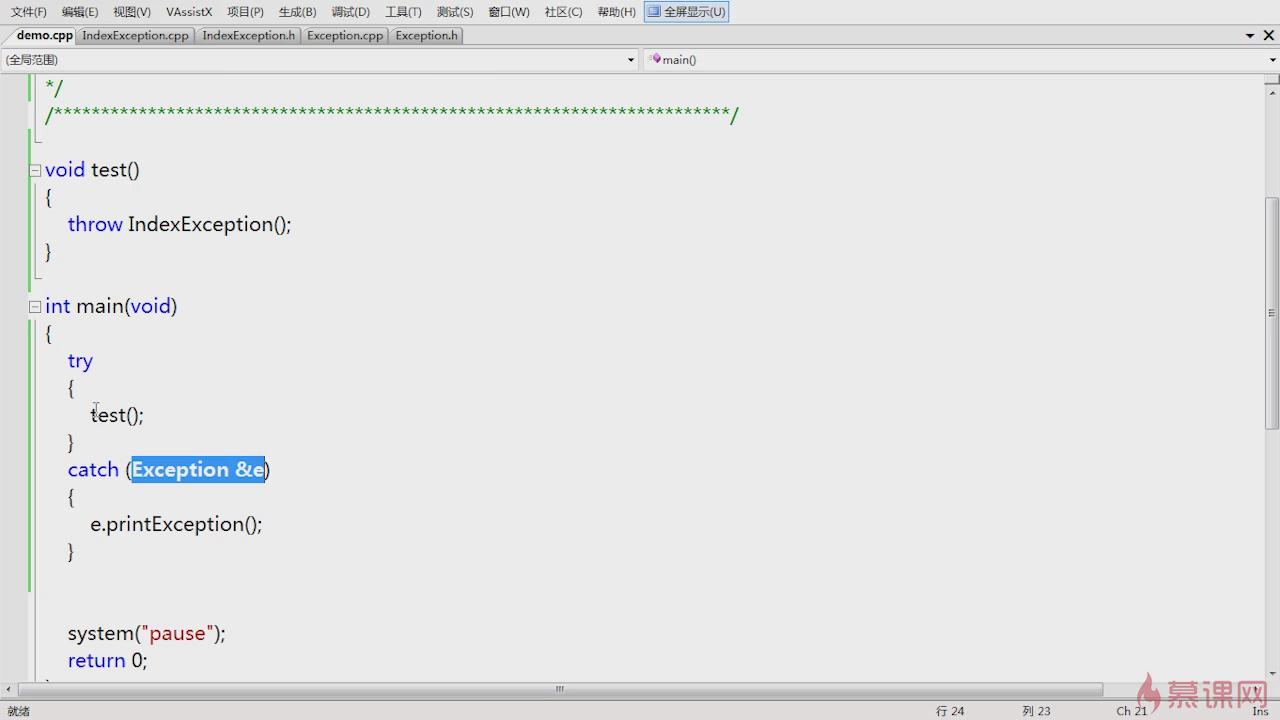
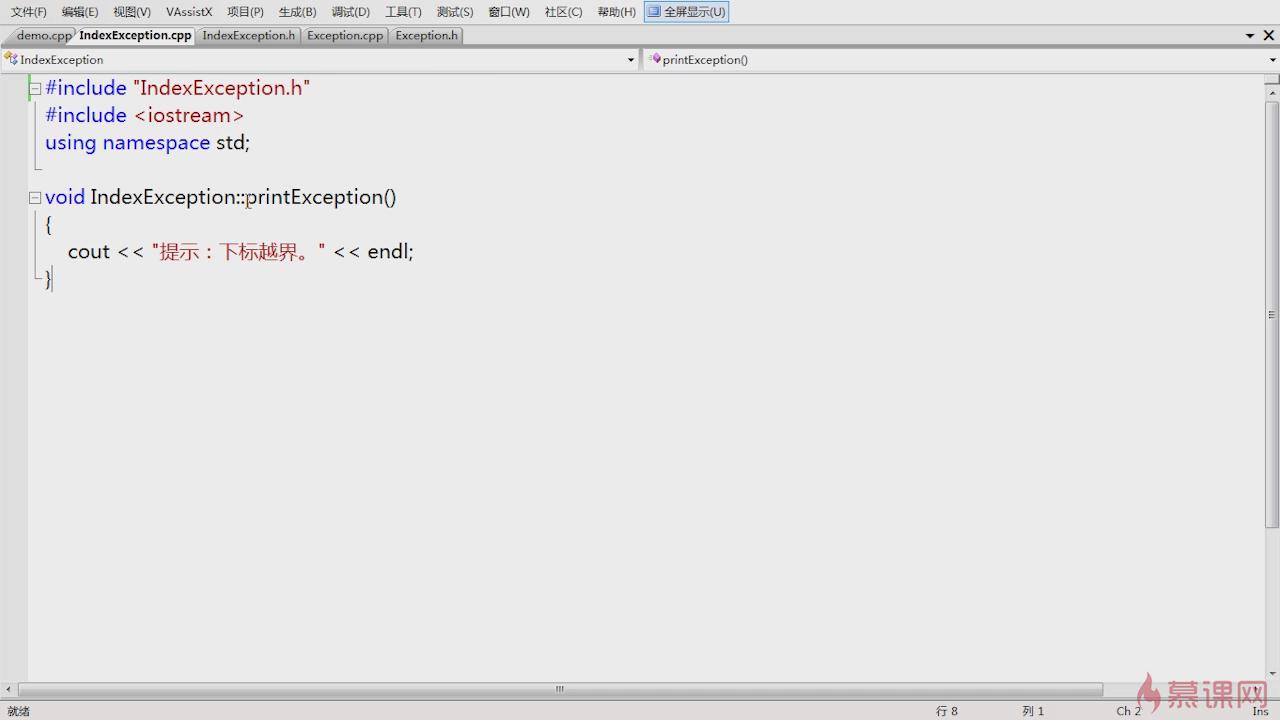
****

**实例：**

****

****

****

****