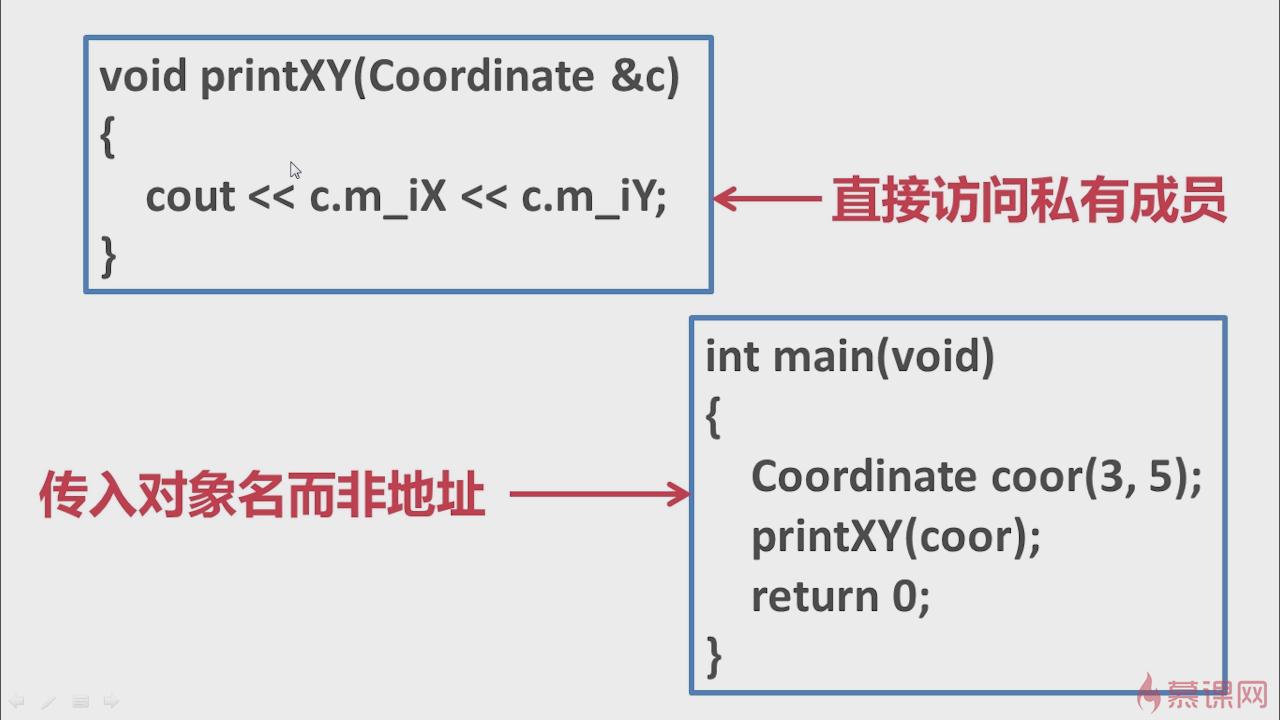
# 1 友元函数:友元全局函数和友元成员函数

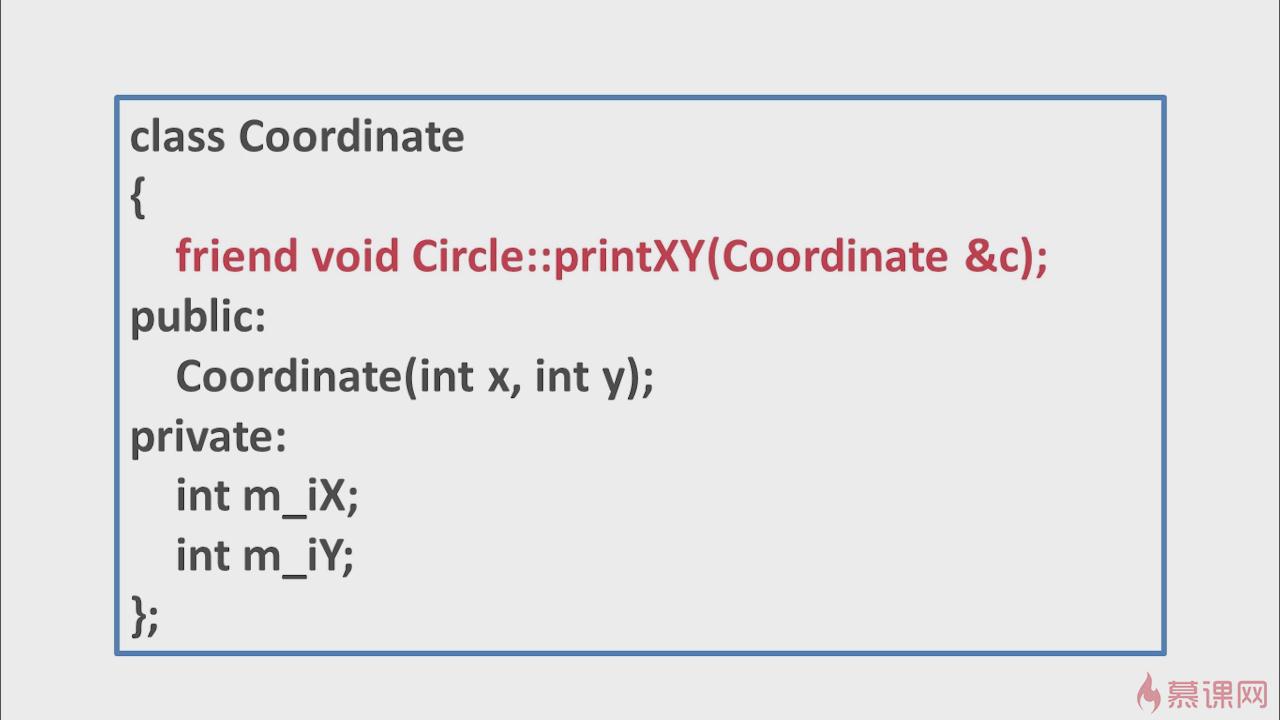
**友元函数关键字：friend**

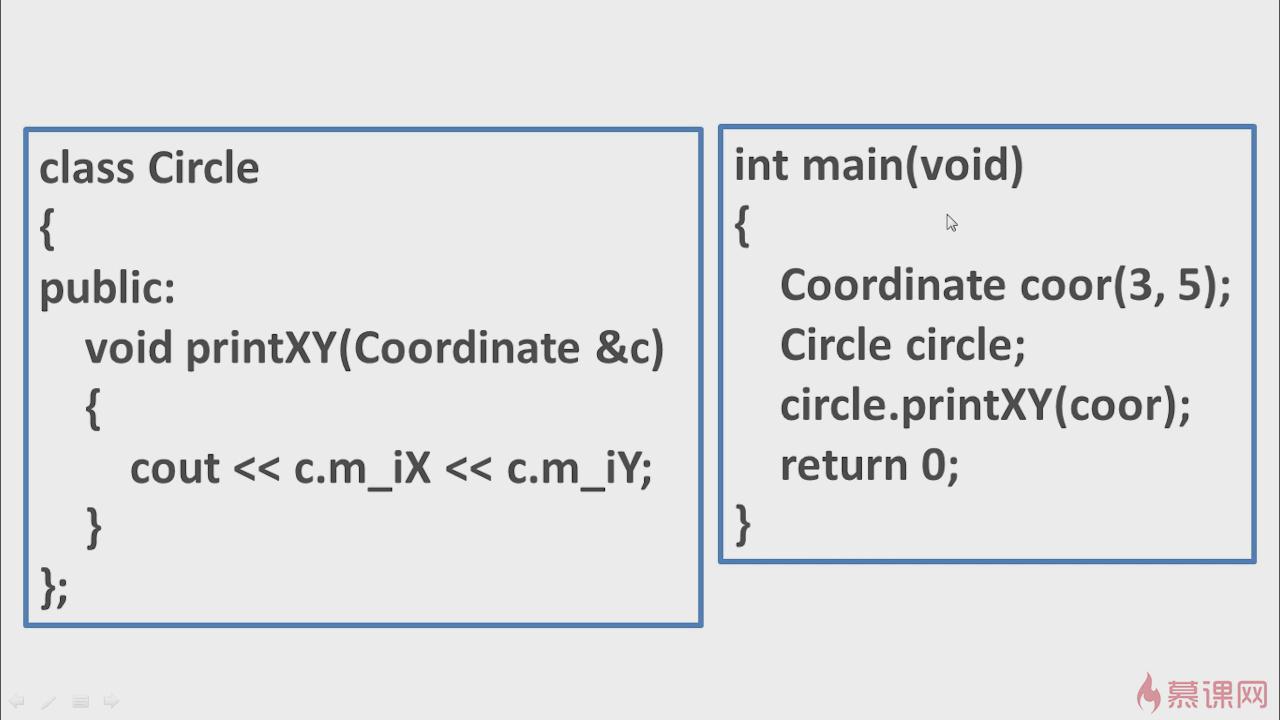
**友元全局函数：**

****

****

**友元函数的优点：可以直接访问私有的或者保护的数据成员**

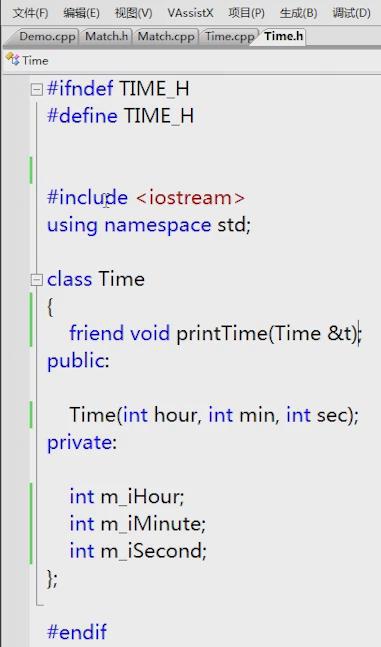
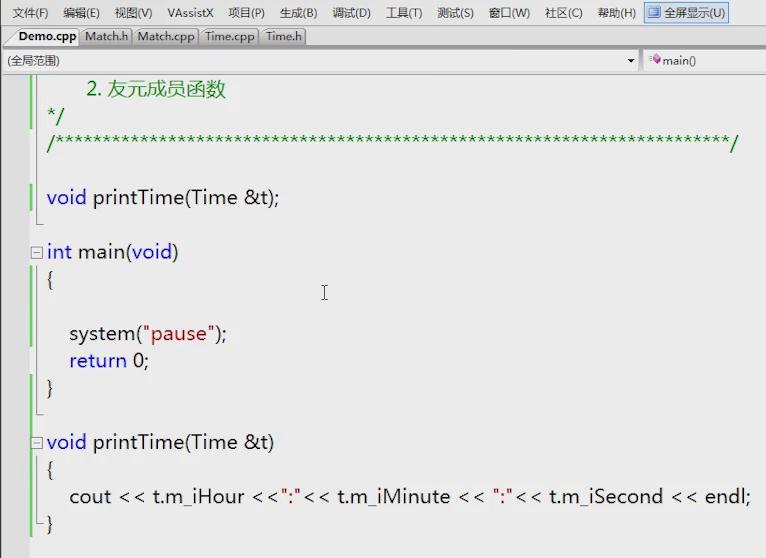
**友元成员函数：**

****

**友元函数的缺点：在直接访问类中的私有或者保护的数据成员的时候，会带来无意中破坏原来类的数据的风险，所以一般情况下不建议使用友元函数。**

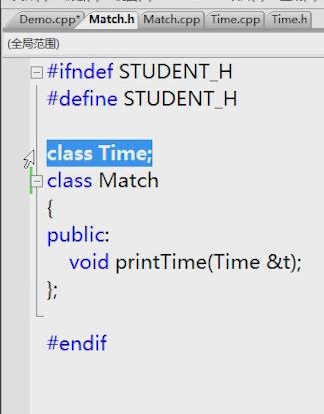
**友元函数实例：**

**如下图，这是一个友元全局函数的实例，我们在main件函数中定义了一个printTime的函数，想要打印出Time类中的“时分秒”成员，但是注意，Time类下的时分秒是在private访问限定符下的，如果不加任何处理，printTime函数直接使用私有成员“时分秒”肯定是非法的，所以，我们在Time类中声明了printTime是Time类的友元函数：（这是一个友元全局函数）**

****

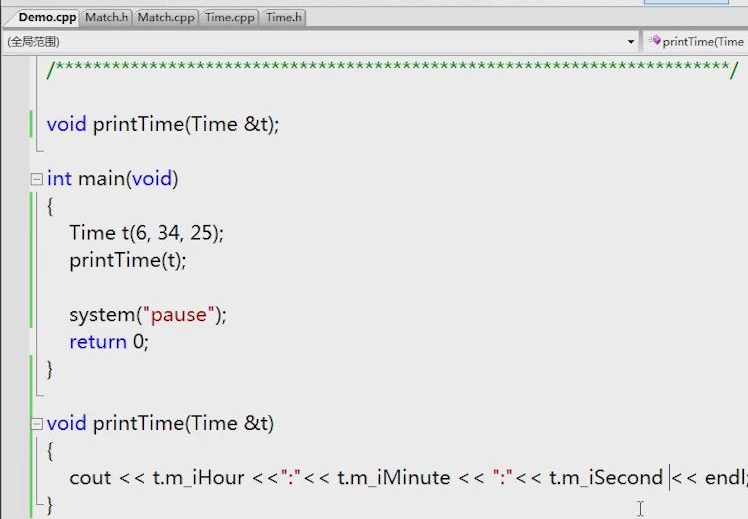
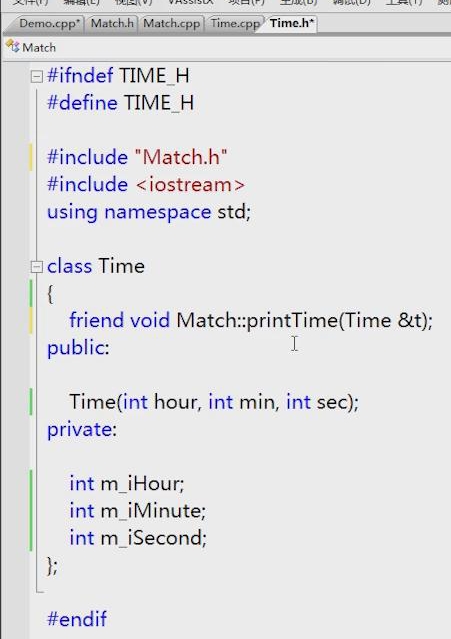
**注意，友元函数的声明在书写的时候可以写在目标类的任意位置，不会影响友元函数的使用效果。**

**下图是将上面的函数改写成为了友元成员函数，我们可以在Match类中定义一个关于Time的友元成员函数，在match类中对该友元函数进行声明的时候我们要特别注意match；类上面的class time，这是对time类的声明，目的是告诉编译系统，在编译的时候先当做已经有了time类，先让其通过编译，然后后面会对其进行定义（类似于先使用后定义）**

****

**如下图，在time中声明友元的时候我们要注意，在对应函数名前一定写清楚该友元成员函数属于谁。**

**和友元全局函数一样，该函数声明可以放在time类的任何位置，效果都是一样的。**

****

# 2 静态数据成员及静态成员函数：

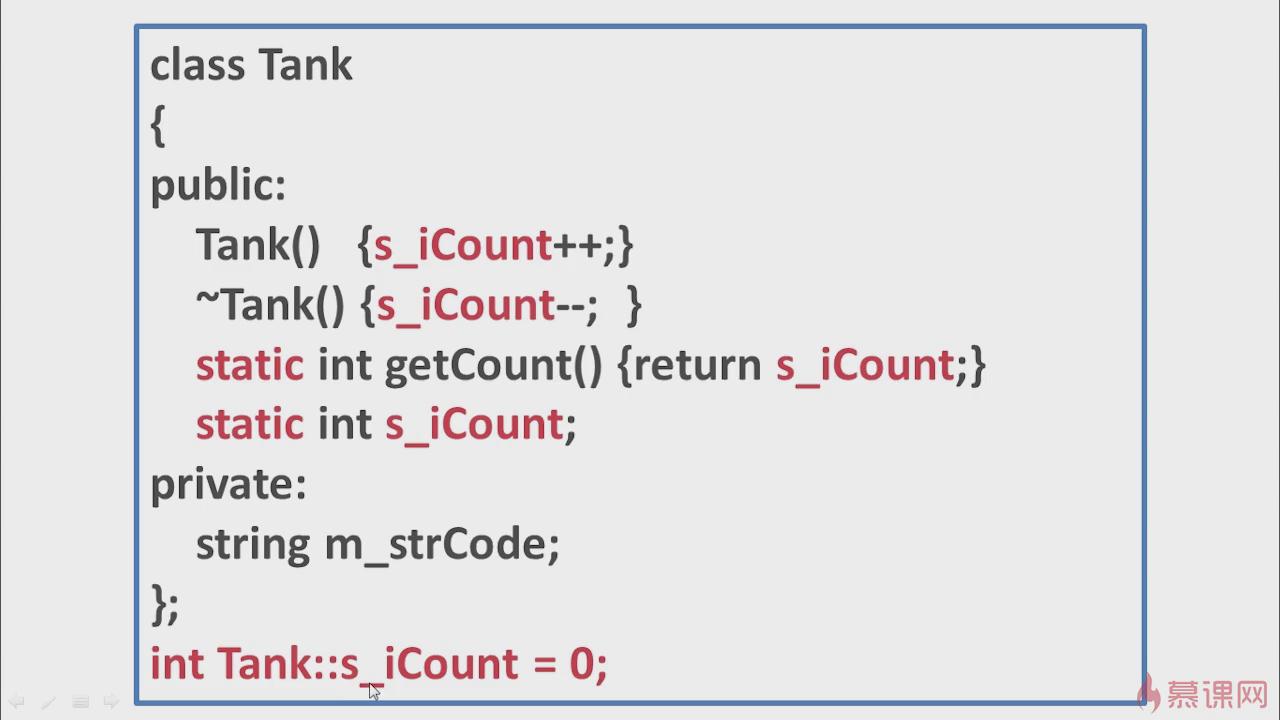
**区分常数据成员和常成员函数与静态成员函数及静态成员函数：**

**常数据成员：不能改变其值得变量即“常变量”**

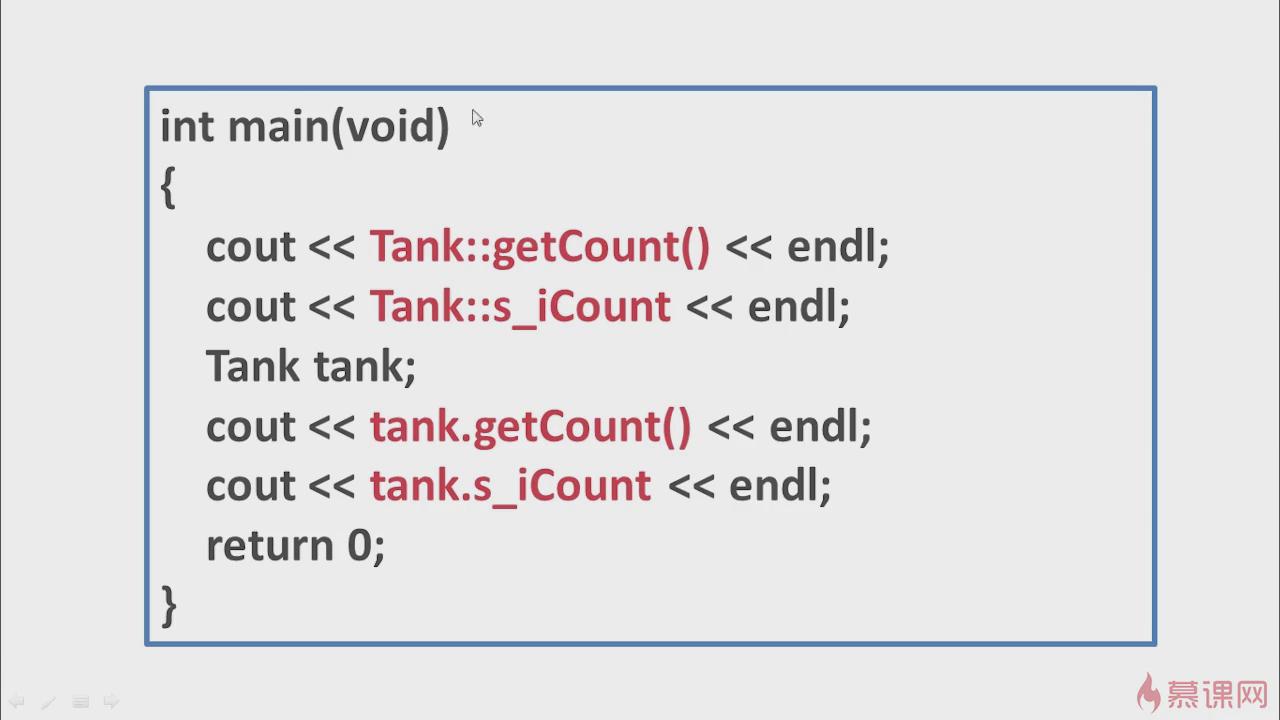
**常成员函数：指只能读取自己类中的数据成员但不能改变自己类中的数据成员的值**

**静态关键字：static**

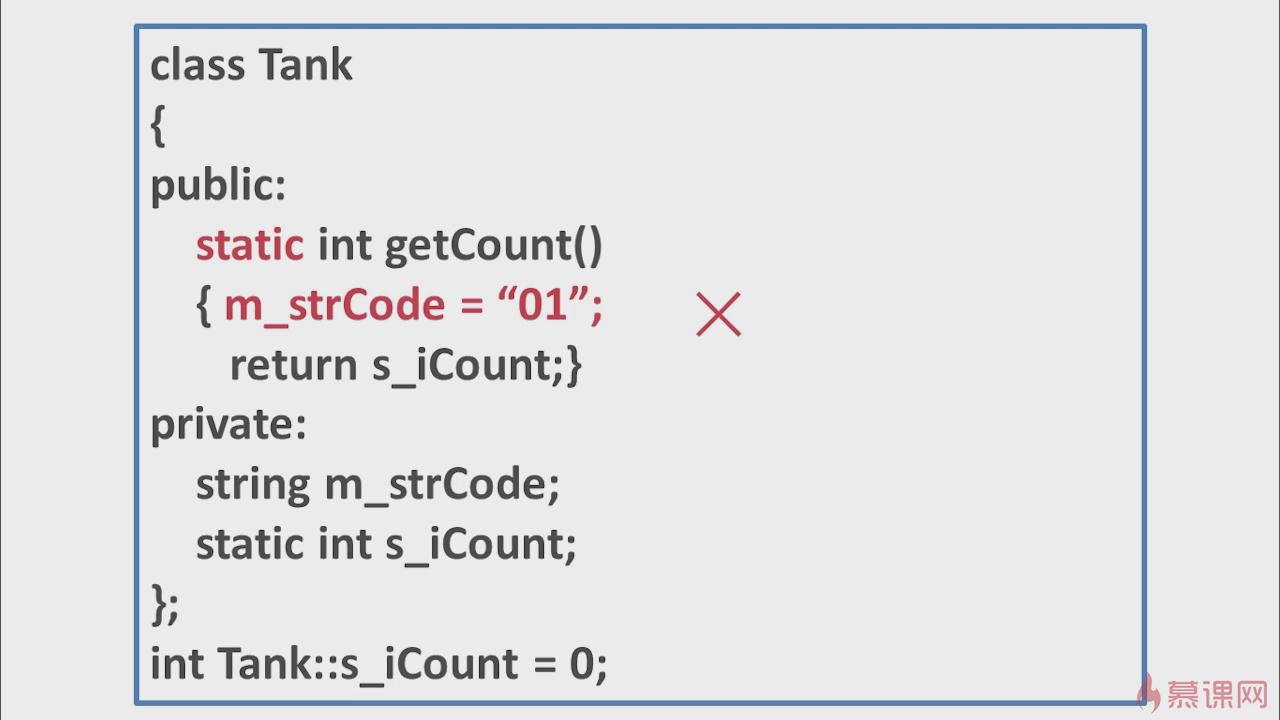
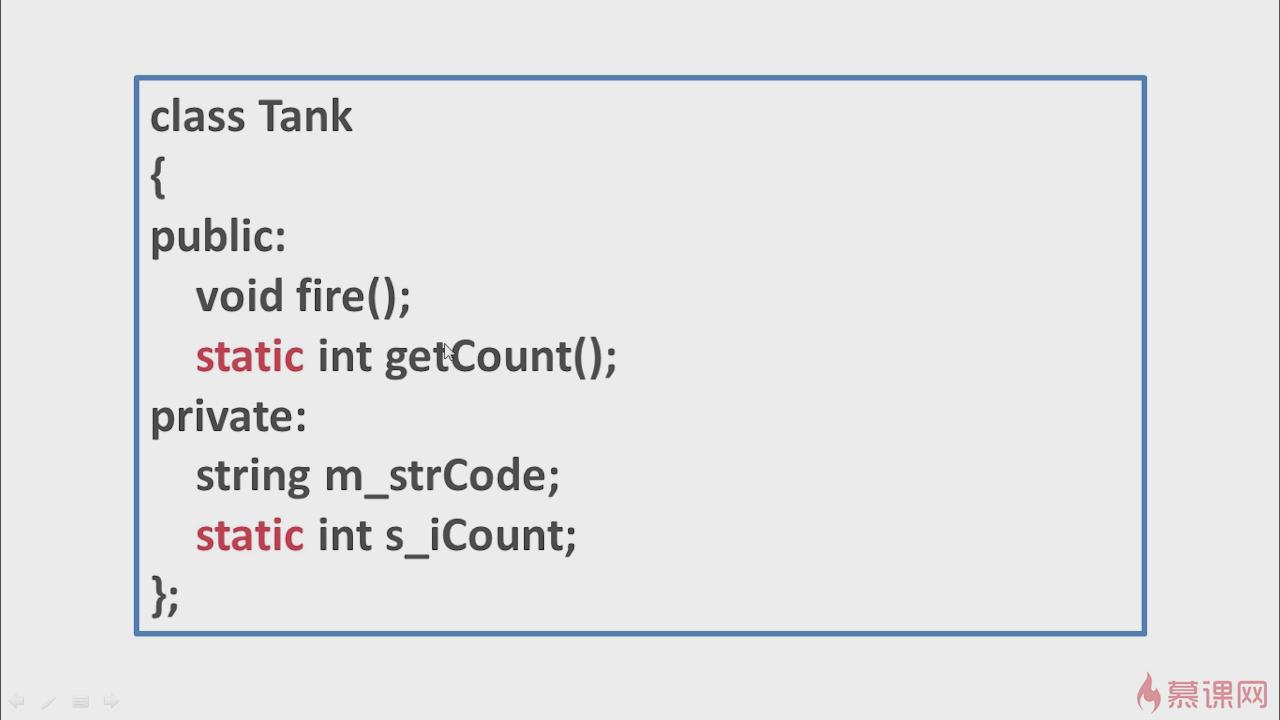
**例：注意，对于静态数据成员来说，必须在类外对其进行单独的初始化（一般是在.cpp文件的开头对该静态变量进行初始化），而不能在构造函数中对其进行初始化，因为静态数据成员是随着类的产生就产生了，而不是随着对象的产生而产生的，当定义了一个类但是还没有实例化对象的时候就已经存在静态数据成员和静态成员函数了，所以我们不能在构造函数中初始化静态数据成员，因为构造函数只有在实例化对象的时候才会被执行：**

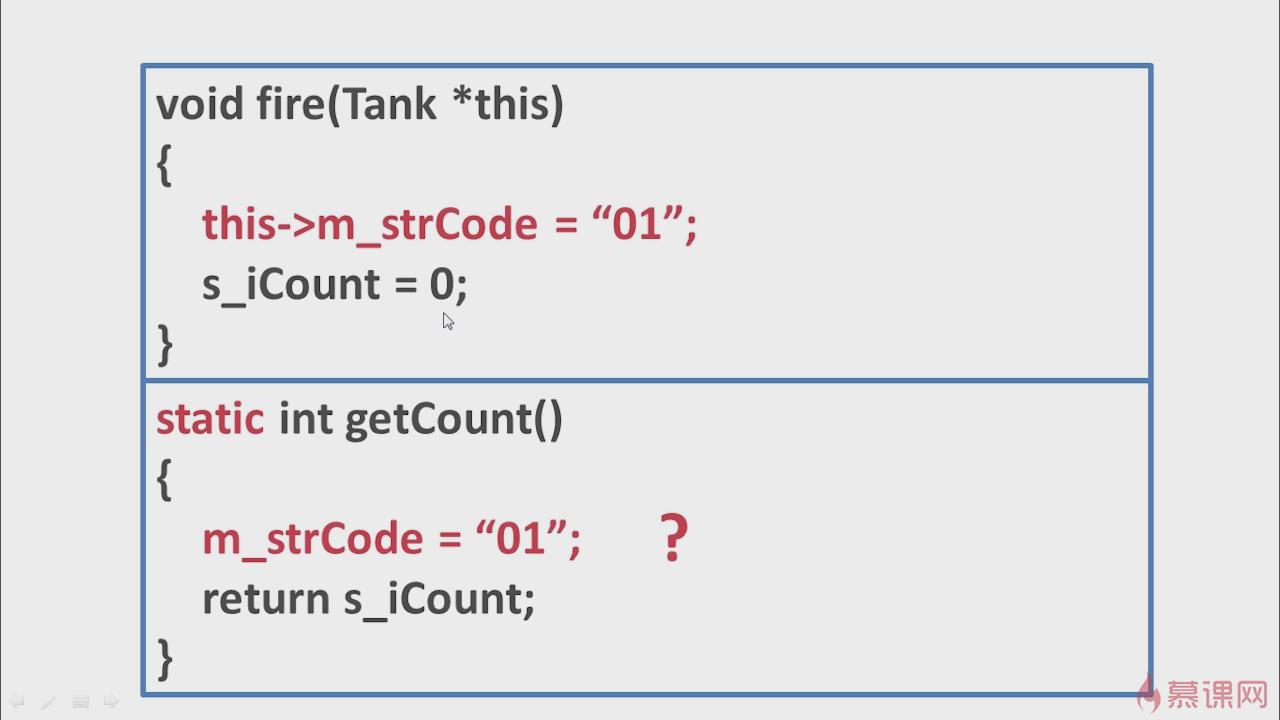
****

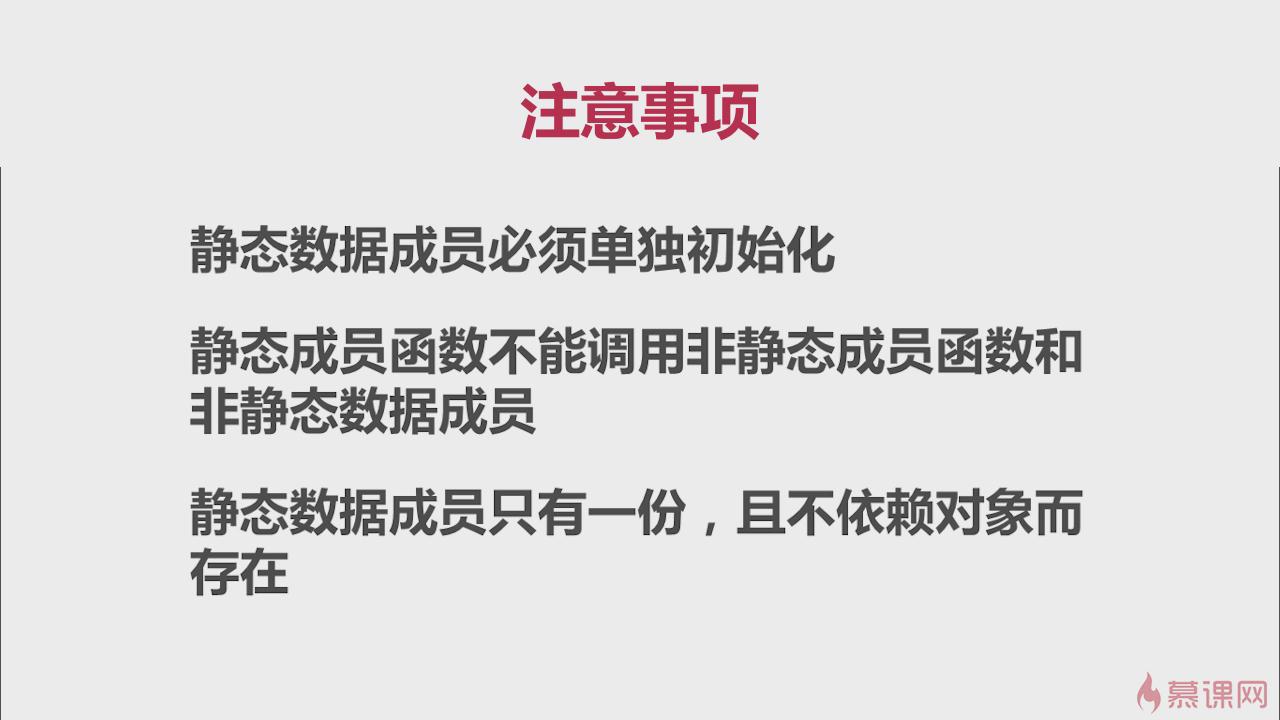
**如图，访问静态数据成员和静态成员函数的两种方法：**

****

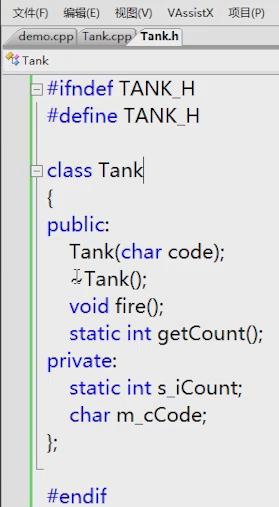
**注意，不能在静态成员函数中使用非静态的数据成员及调用非静态的成员函数：**

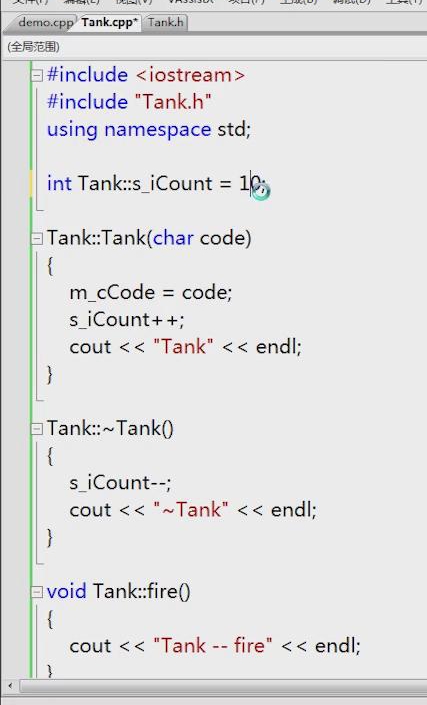
****

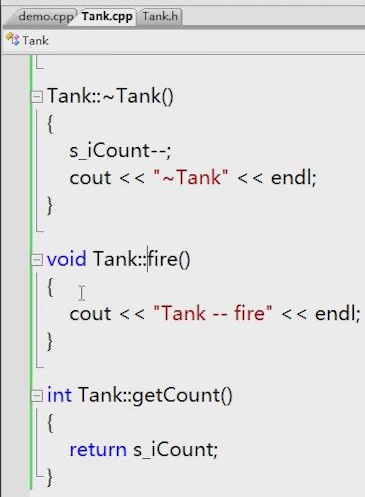
**原因：如图，在非静态成员函数中，当我们使用类的成员函数或者数据成员的时候会传入一个隐形的this指针，告诉编译系统该数据成员或者成员函数是对应类的，这样就可以直接使用。但是，在静态成员函数中由于它是在对象产生之前和类一起产生的，所以不会传入该类的this指针，这时候再使用非静态的数据成员或者成员函数的时候系统就会懵逼，就会报错。（其实就相当于静态数据成员和静态成员函数是全局变量和全局函数，当然不能在全局函数中使用类中的数据成员）**

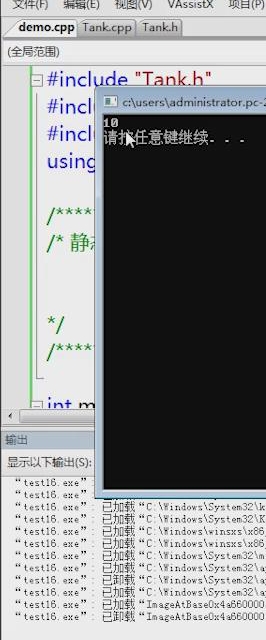
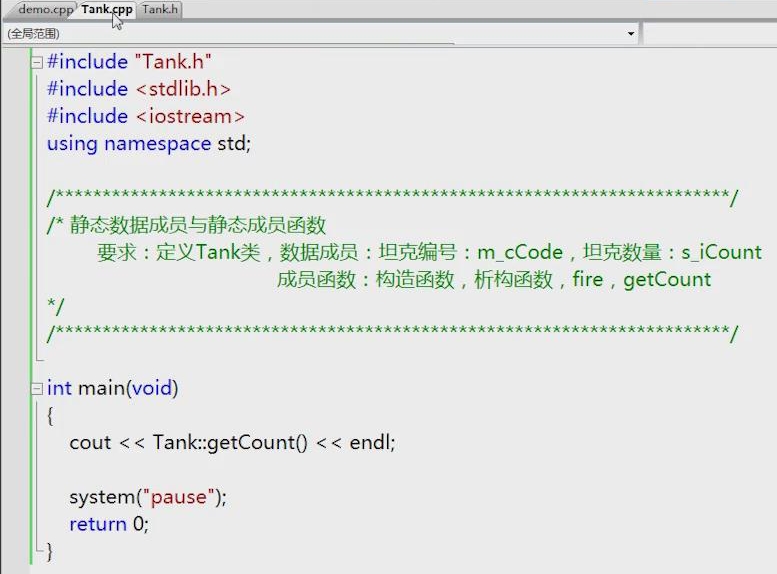
**静态数据成员和静态成员函数的注意事项：**

**静态数据成员和静态成员函数实例：**

****

****

****

****

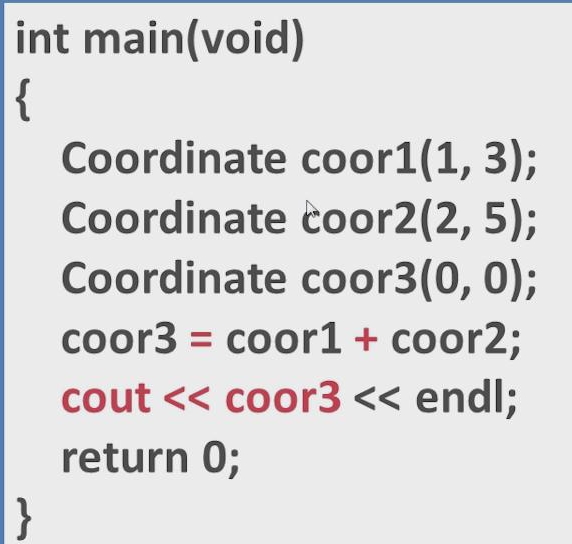
# 3：运算符重载：

**概念：给原有运算符赋新的功能**

**本质：函数重载**

**关键字：operator**

**例：**

****

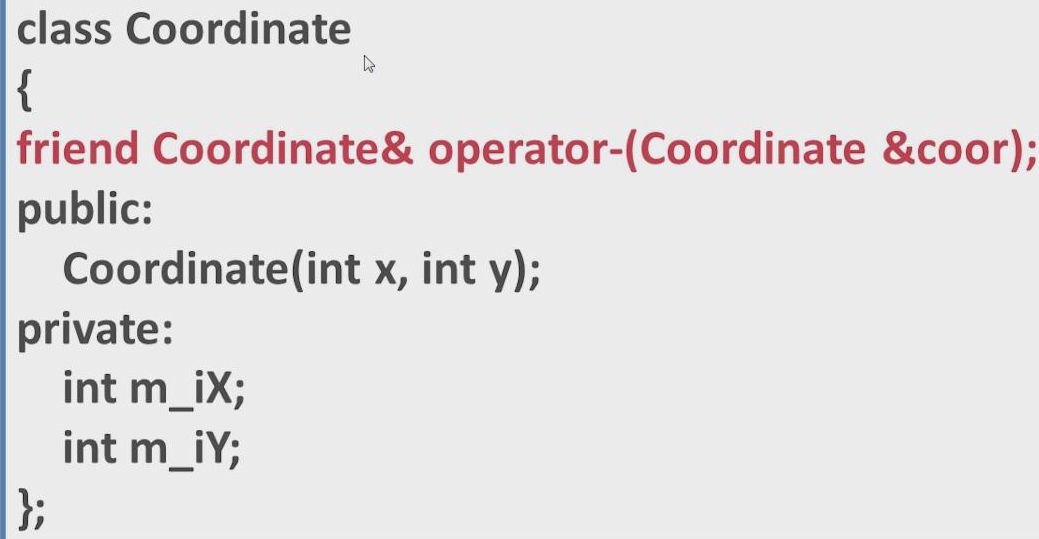
**在这个例子中coordinate是一个新定义的类，其包含了两个数据成员m\_iX和m\_iY，如果没有任何额外的定义声明，使用coordinate对象之间的等于（=）和加号（+）都会是非法的、计算机无法正确识别的，所以要想执行coor3=coor1+coor2，就必须要对=和+关于coordinate进行运算符重载，即赋予=和+关于coordinate新的定义。**

## 一元运算符重载：

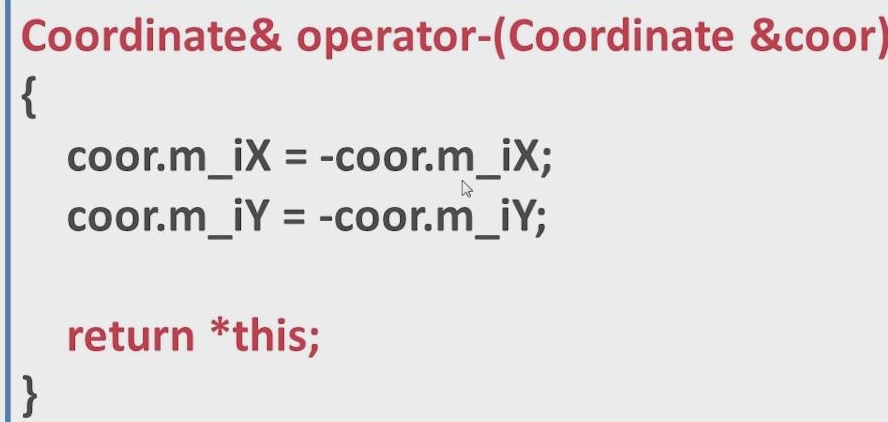
### 负号（-）运算符的重载

#### 友元函数重载

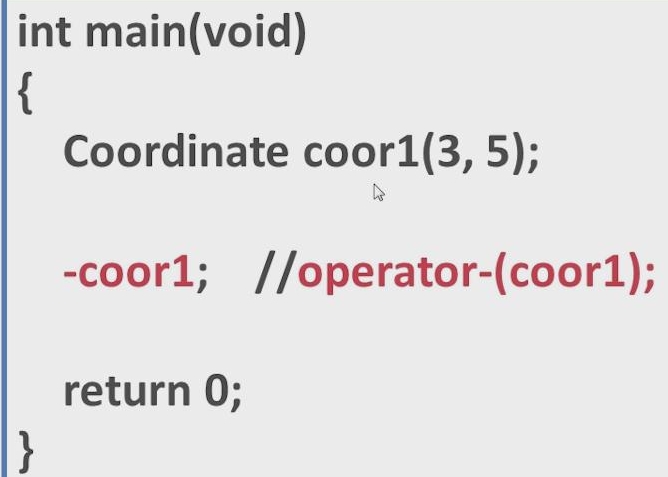
**注意一元运算符使用友元函数重载时的声明方法：**



**注意一元运算符使用友元函数重载时的定义方法：**

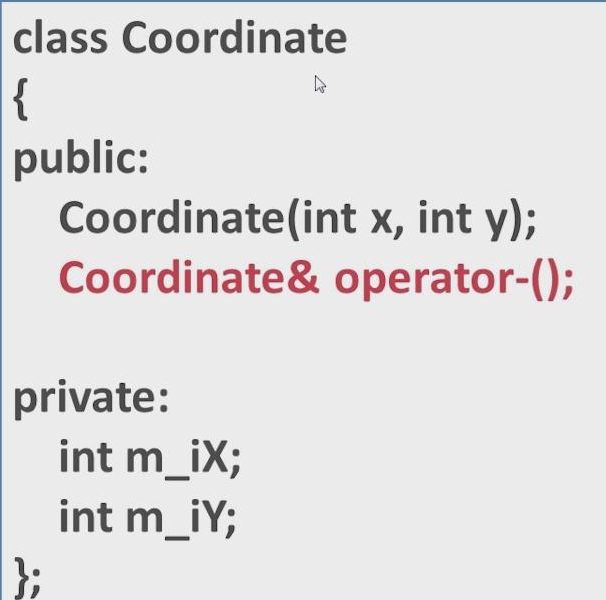


**注意一元运算符使用成员函数重载时的使用方法：**

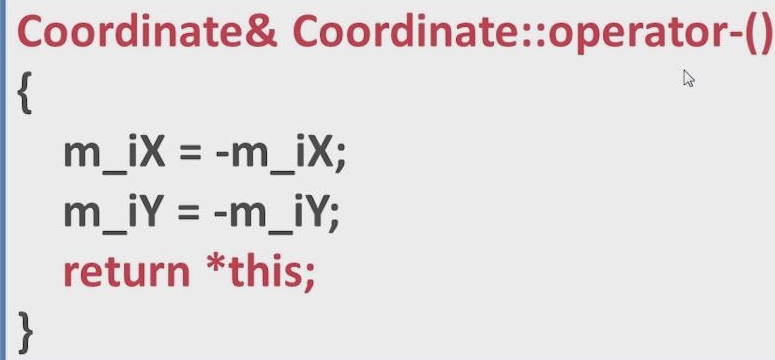


#### 成员函数重载

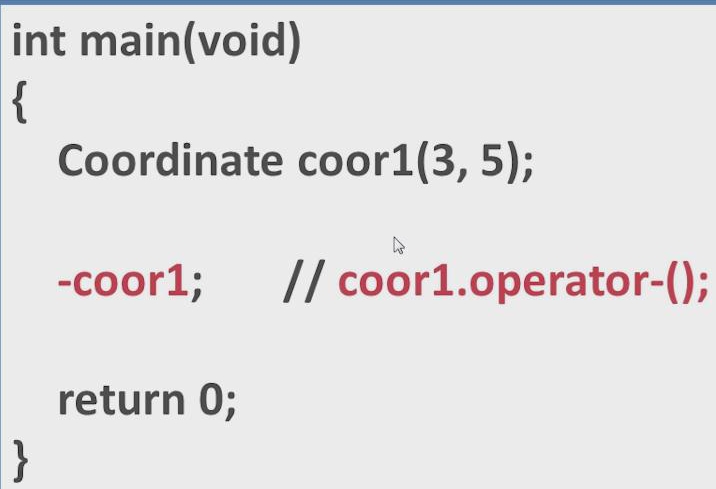
**注意一元运算符使用成员函数重载时的声明方法：**



**一元符号使用成员函数进行重载的定义方法：**



**一元运算符使用成员函数重载时的使用方法：**

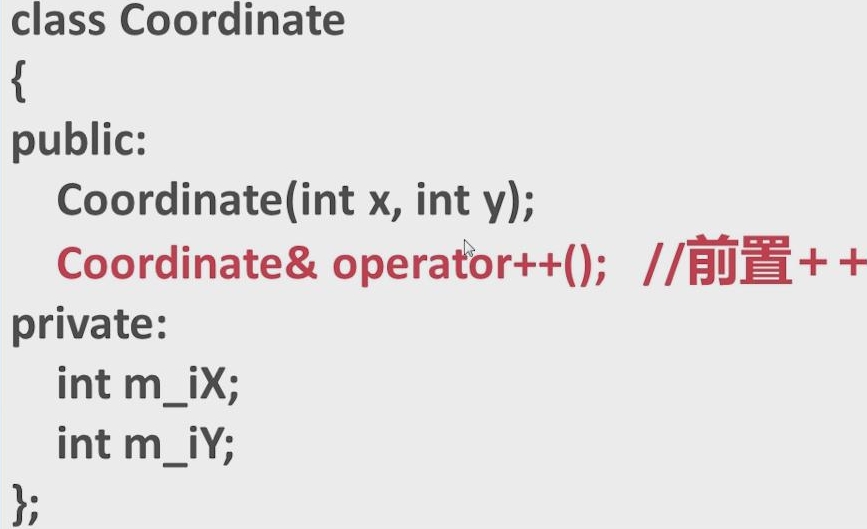
****

**所以可以得出一元符号不管是进行成员函数和重载还是进行友元函数重载，其使用方法都是一样的**

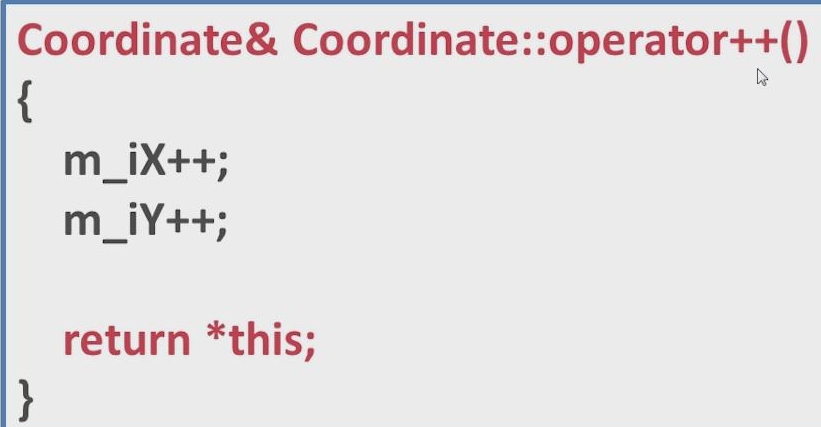
### ++运算符符号重载

#### 前置++符号重载

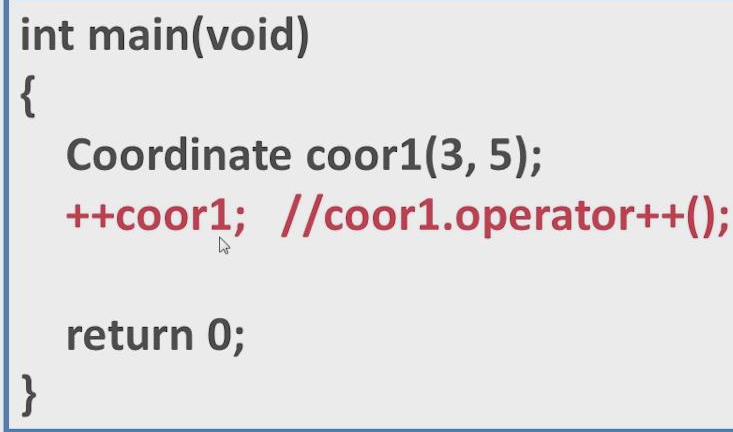
**声明方法：**

****

**定义方法：**

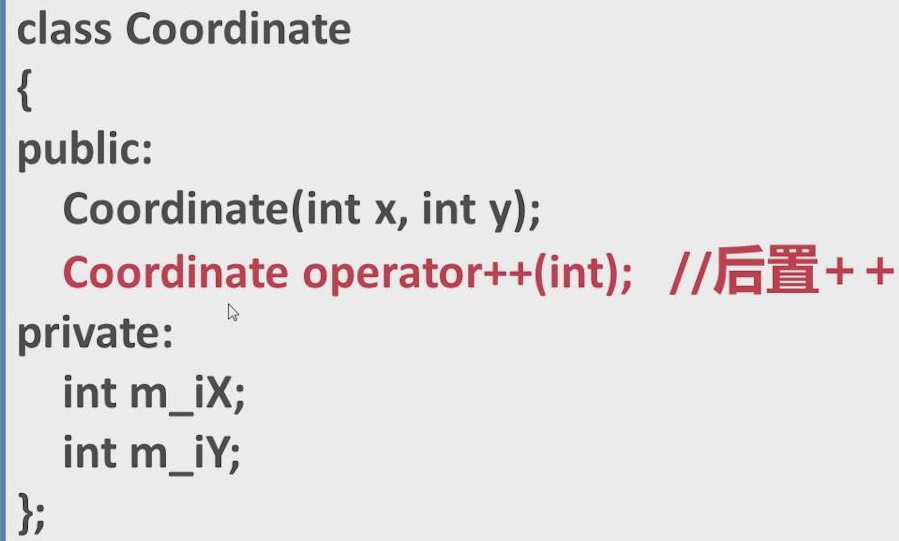


**使用方法：**

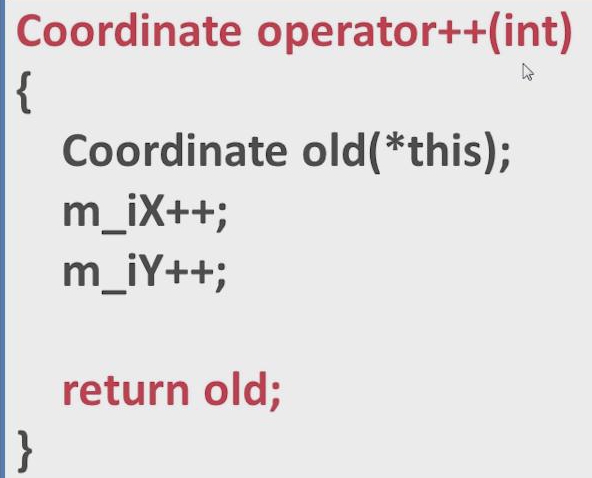


#### 后置++符号重载

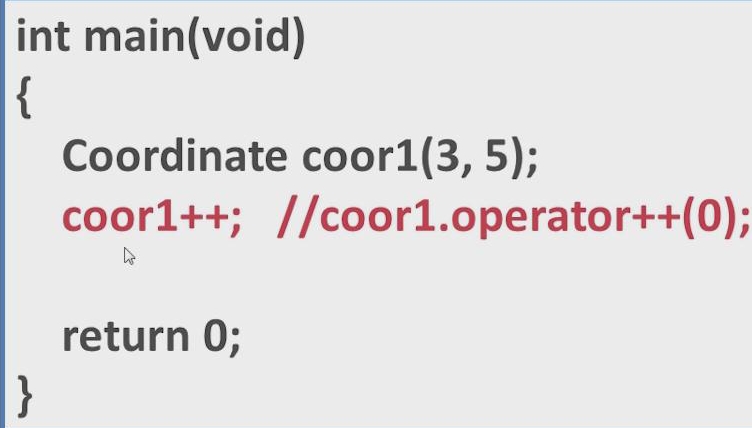
**声明方法：**



**定义方法：**



**使用方法**：



**对比前置++和后置++我们要注意的是后置++区别于前置++的声明方法在于后置++的函数声明里面要多加一个int，这个int在使用的时候没有任何作用，可以不用传入热河数值，只是系统为了区别前置和后置所要求的使用规则**

### 一元运算符重载总结：

**通过-运算符和++运算符的重载我们可以得出以下结论：**

**1：声明方法：**

**Coordinate& operator-（）；**

**friend Coordinate& operator-(Coordinate &coor);**

**Coordinate& operator++（）；//前置++声明**

**Coordinate& operator++（int）；//后置++声明**

**2：定义方法：**

**Coordinate& Coordinate：：operator-（）**

**Coordinate& operator-（Coordinate &coor）**

**Coordinate& Coordinate：：operator++（）**

**Coordinate operator++（int）**

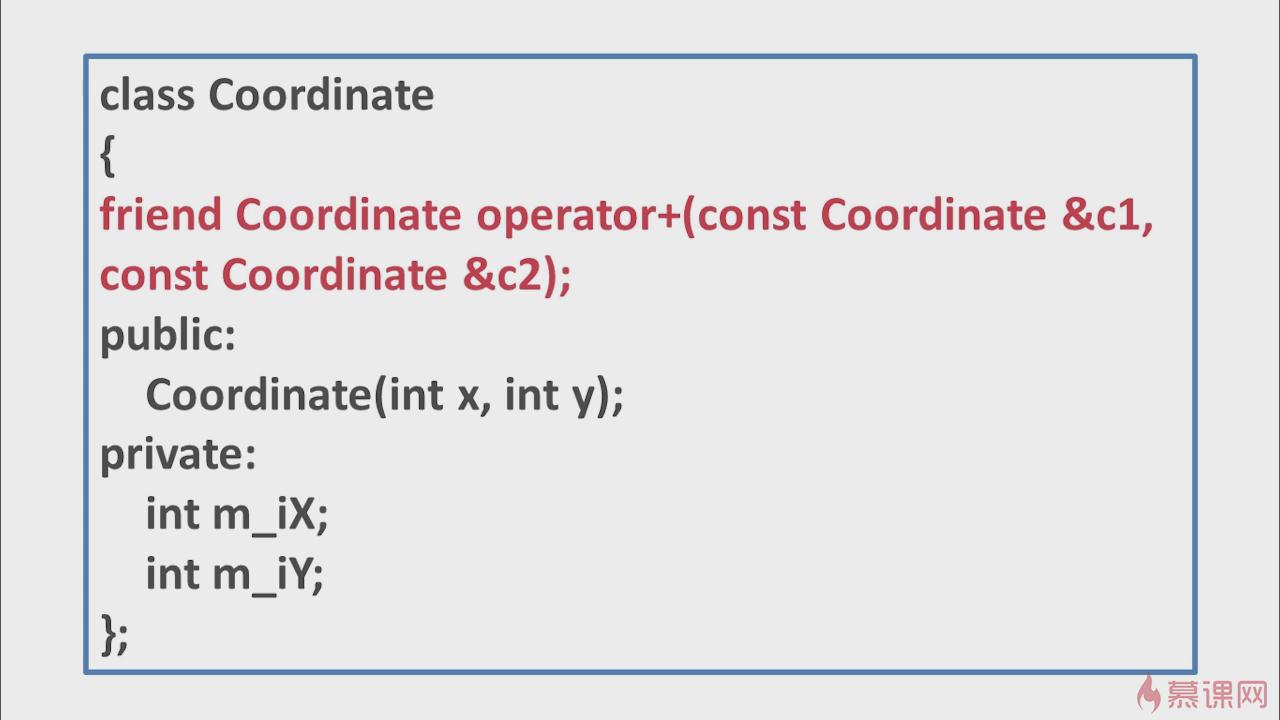
**可以发现：如果函数的返回类型是引用则需要返回this指针。**

## 二元运算符重载：

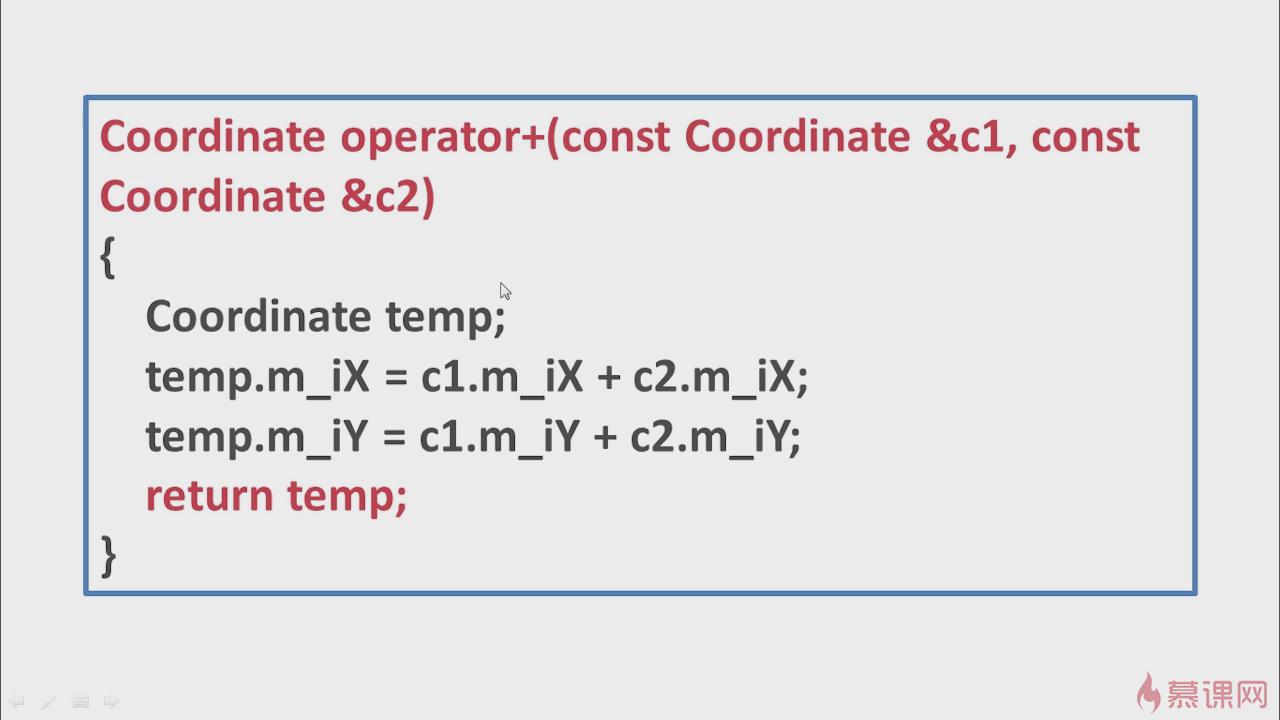
### +号运算符重载：

#### 友元函数重载

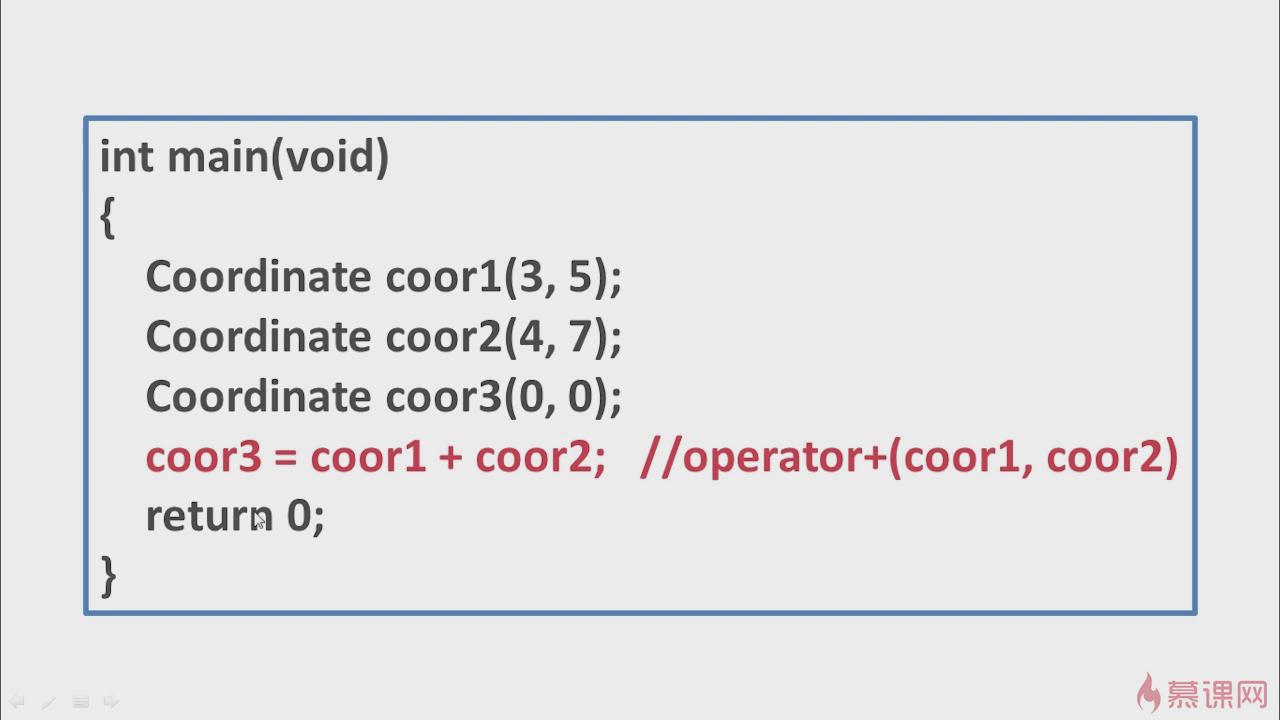
**声明方法：**



**定义方法：**

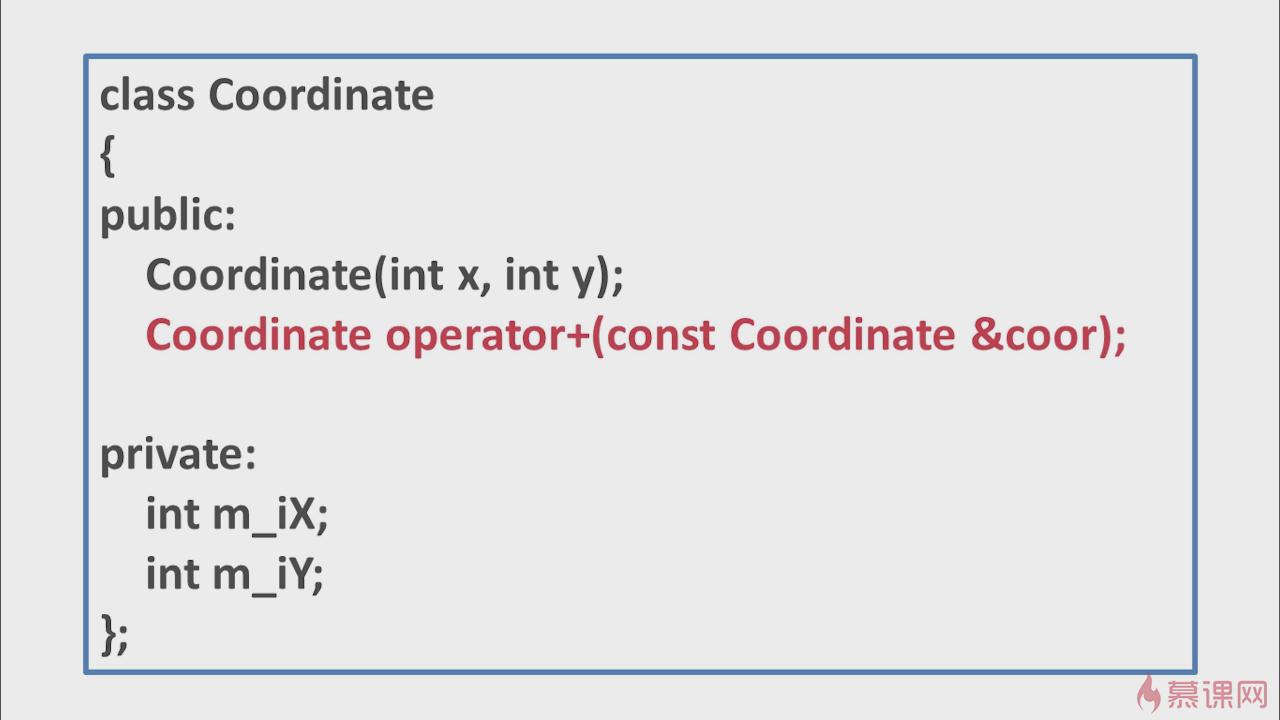


**使用方法：**

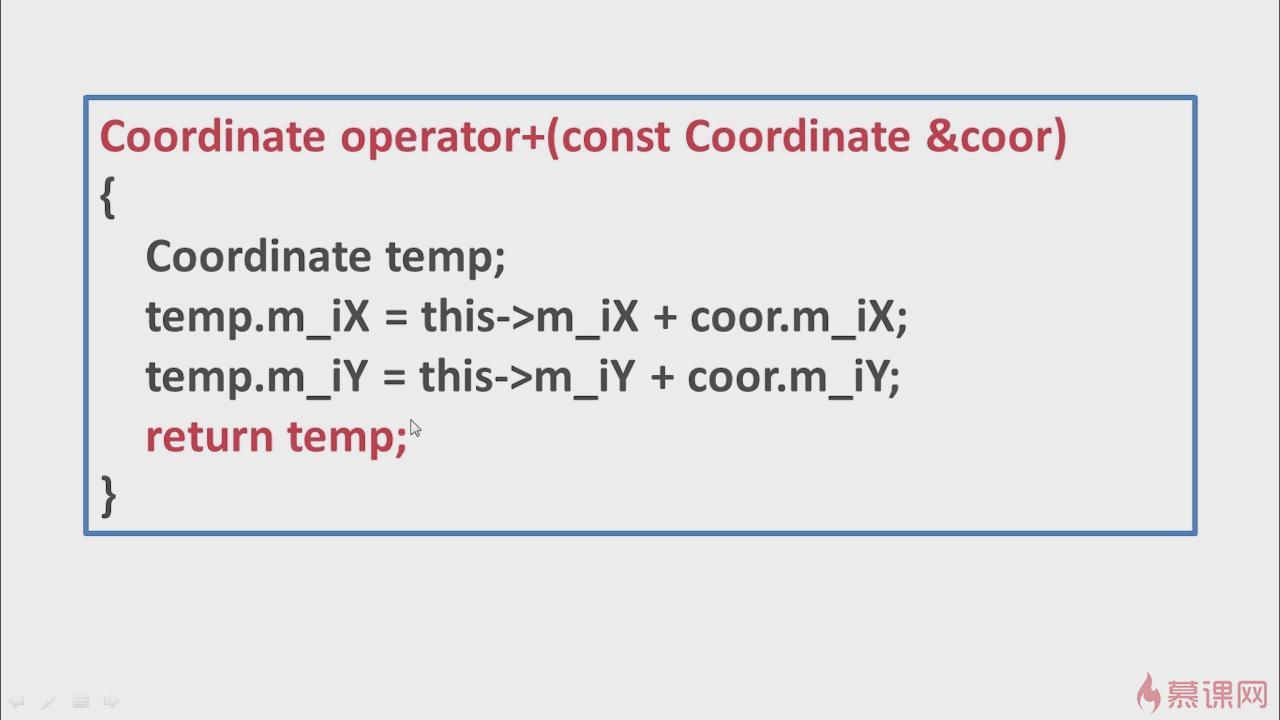


#### 成员函数重载

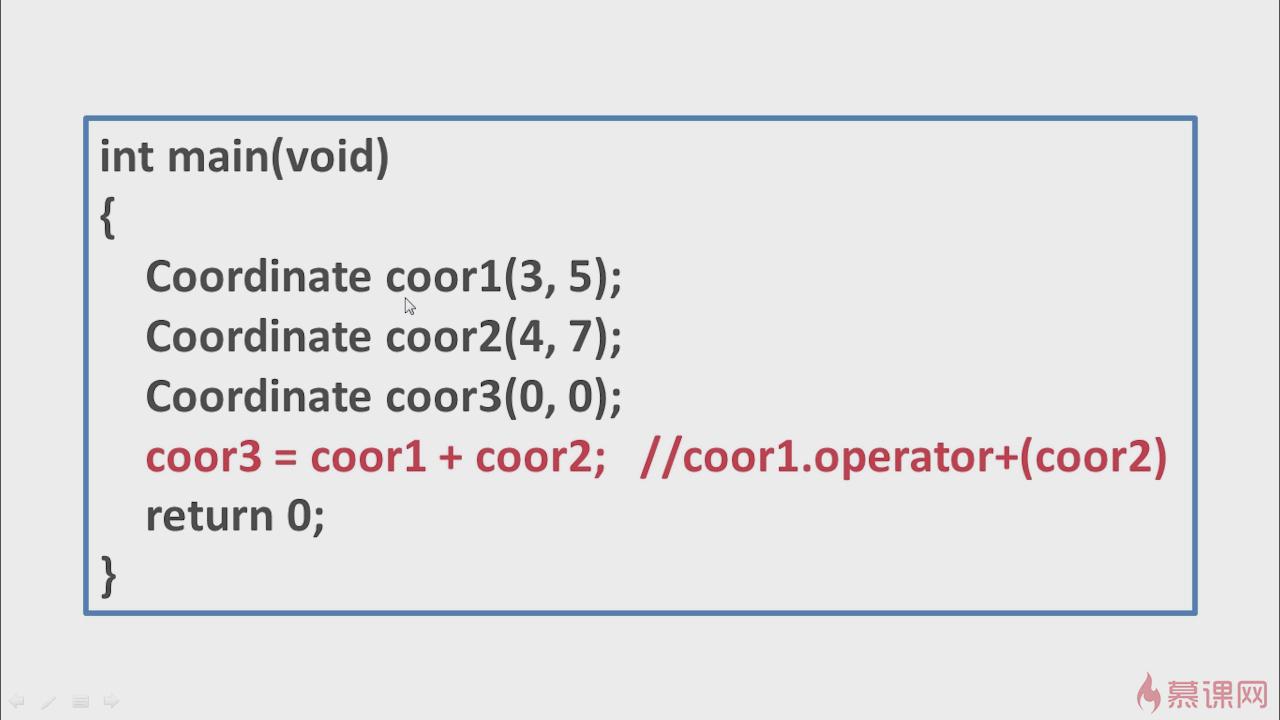
**声明方法：**



**定义方法：**

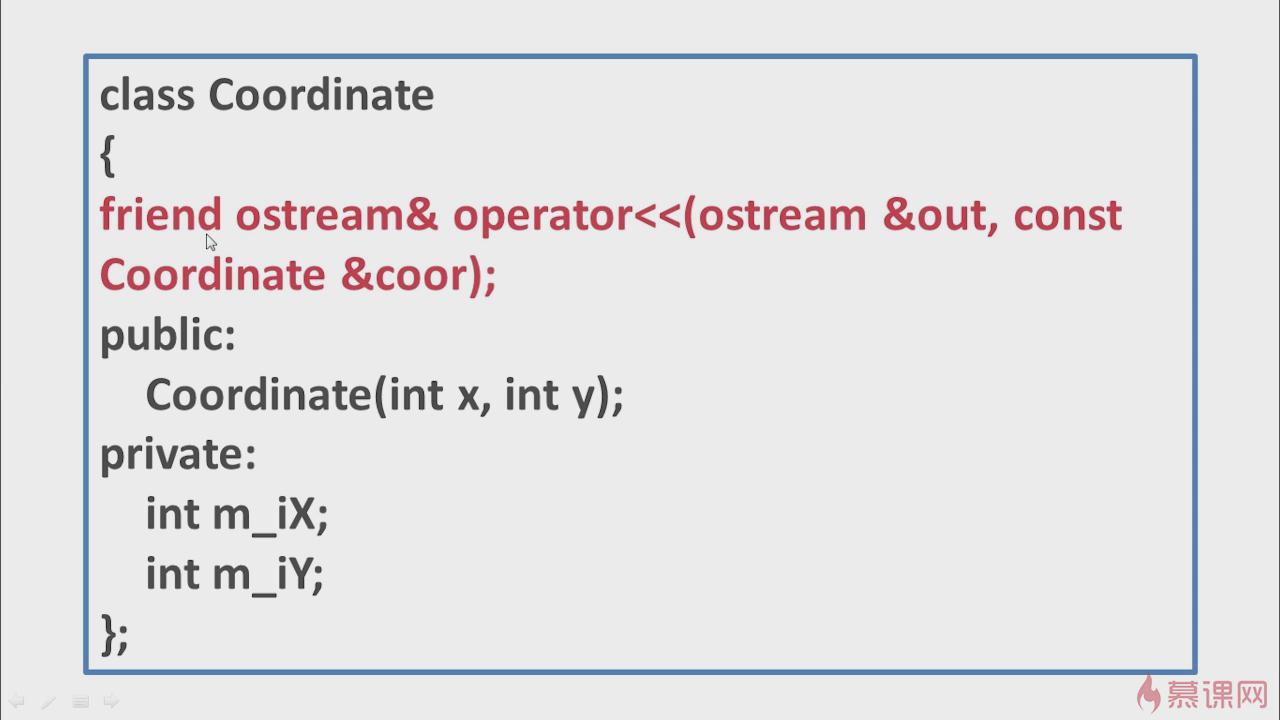


**使用方法：**

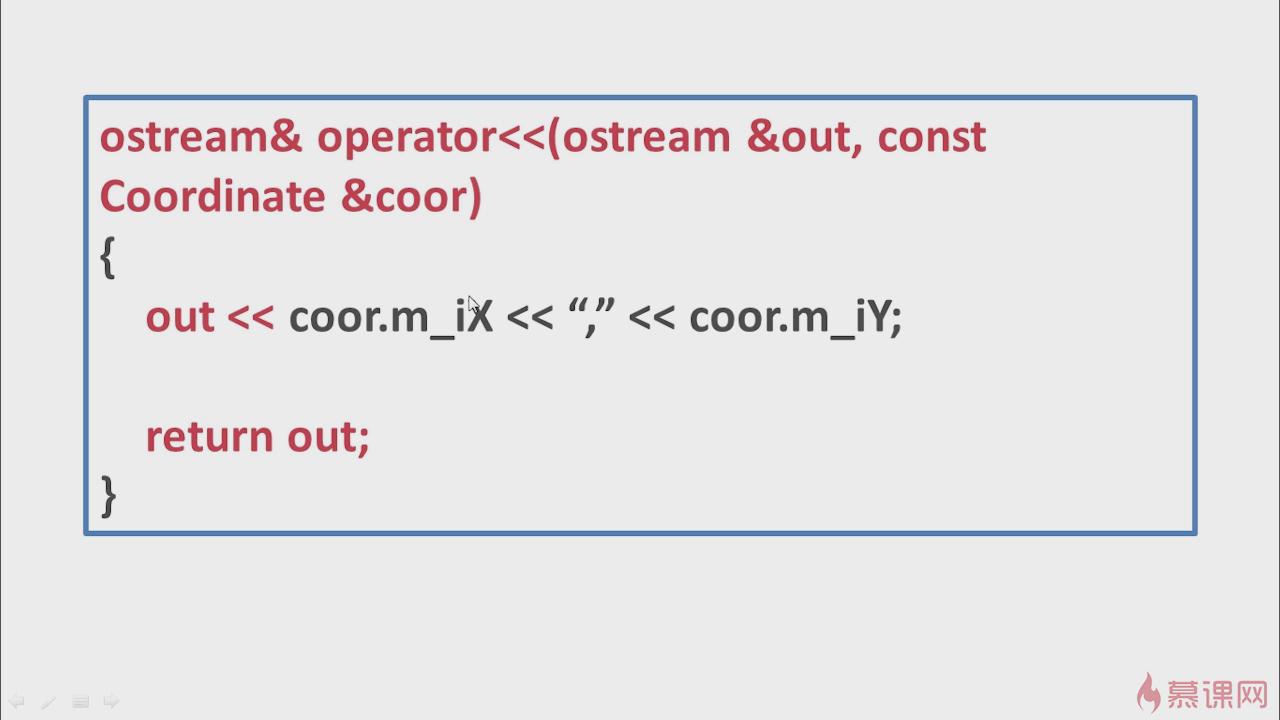


### <<（输出运算符重载）

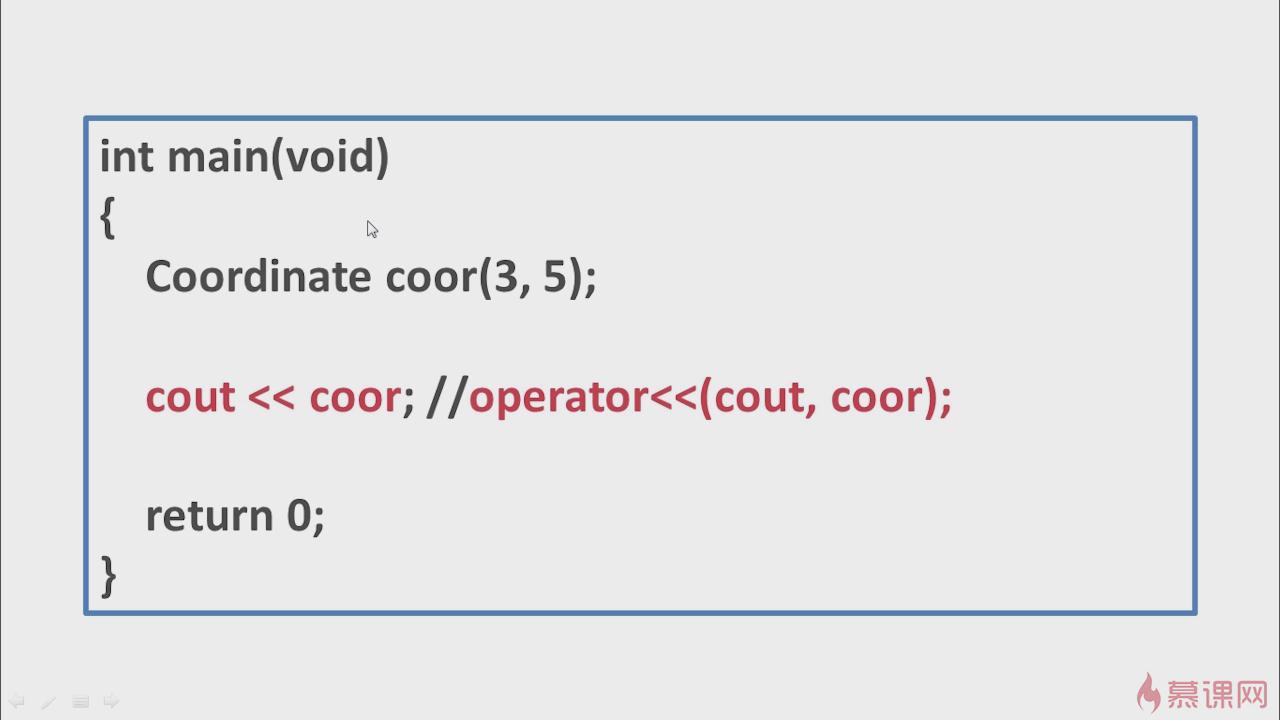
**声明方法：**



**定义方法**：

****

**使用方法：**



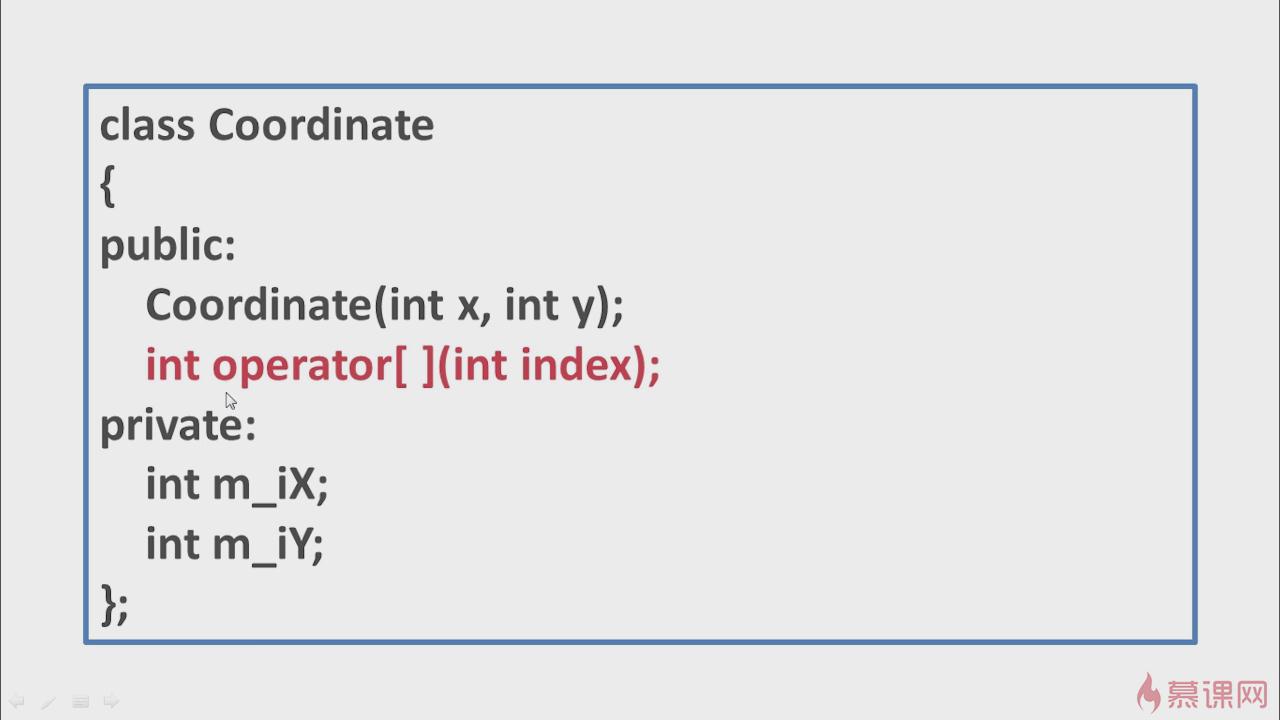
**思考：输出运算符可以采用成员函数重载吗？**

**答案是否定的，原因：**

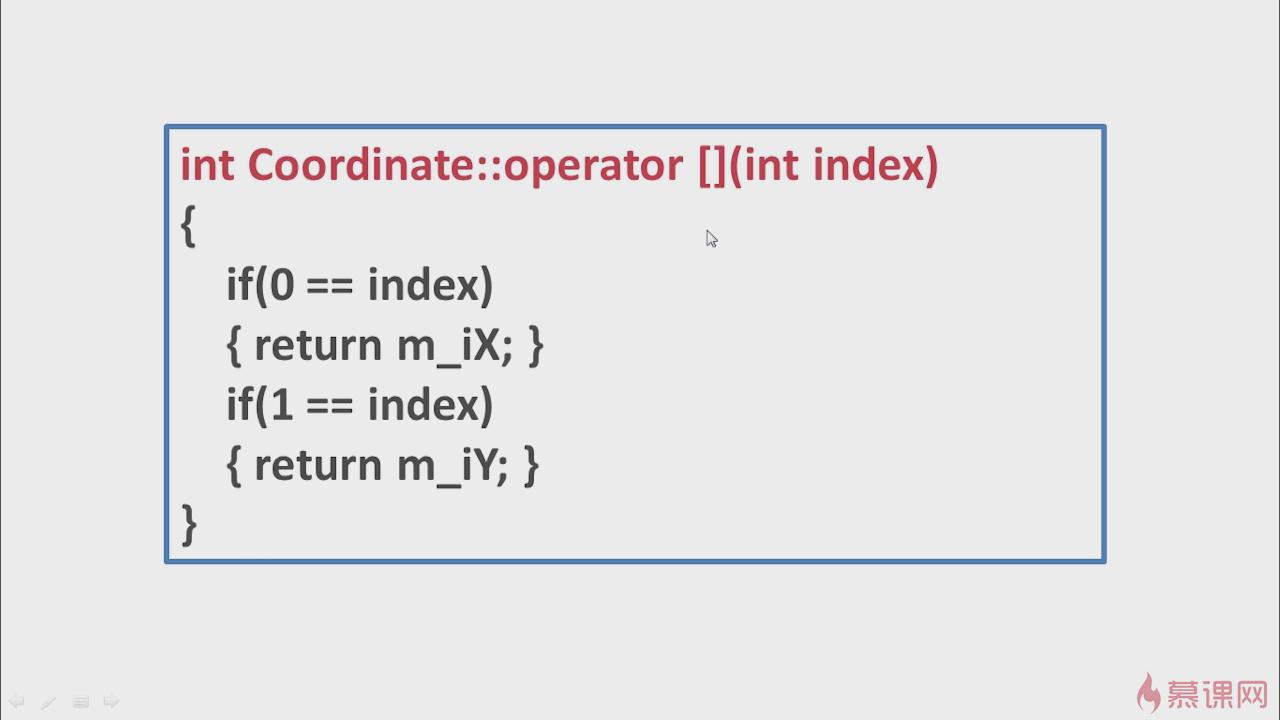
**对于<<运算符来说，其要求第一个参数必须是ostream类型的，而如果使用成员函数重载，第一个对象总会默认成当前对象的this指针，这样的话就会产生矛盾，所以，对于输出运算符<<来说只能使用友元函数进行重载而不能使用成员函数进行重载。**

### 索引运算符[]重载

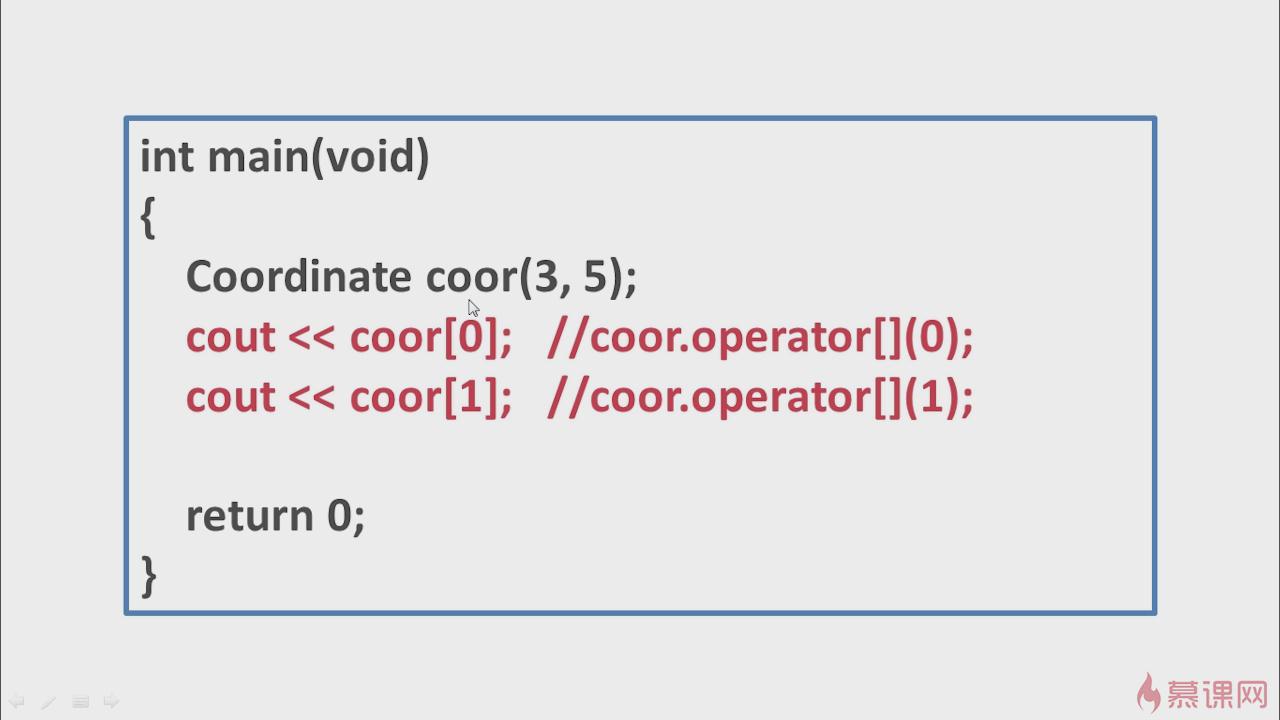
**声明方法：**



**定义方法：**



**使用方法：**



**思考，索引运算符可以采用友元函数重载吗？**

**答案是否定的**

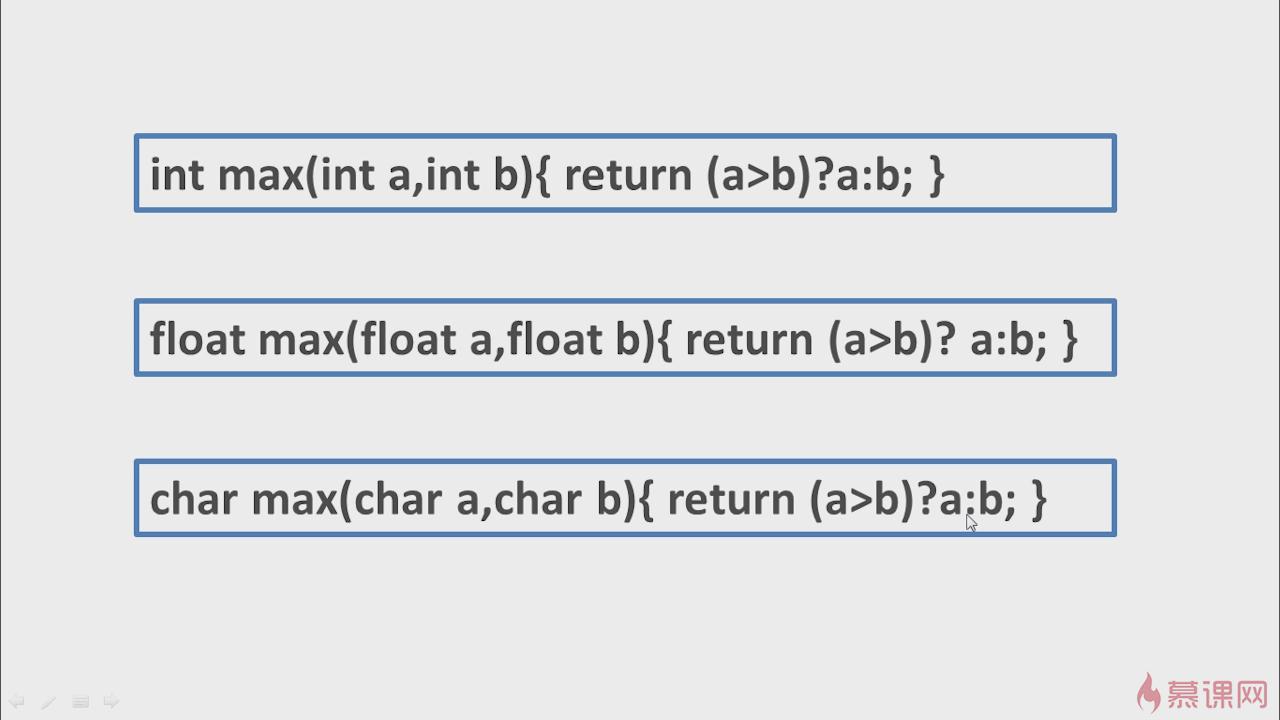
**原因：[]运算符要求第一个参数必须是this指针，而友元函数的第一个参数既可以是对应对象的this指针，也可以是其他的，所以不符合[]运算符要求参数类型的范围，所以不能用友元函数对[]运算符进行重载。**

# 4：模板：

## 函数模板：

**为什么要引入模板？**

**例：如下图：**

****

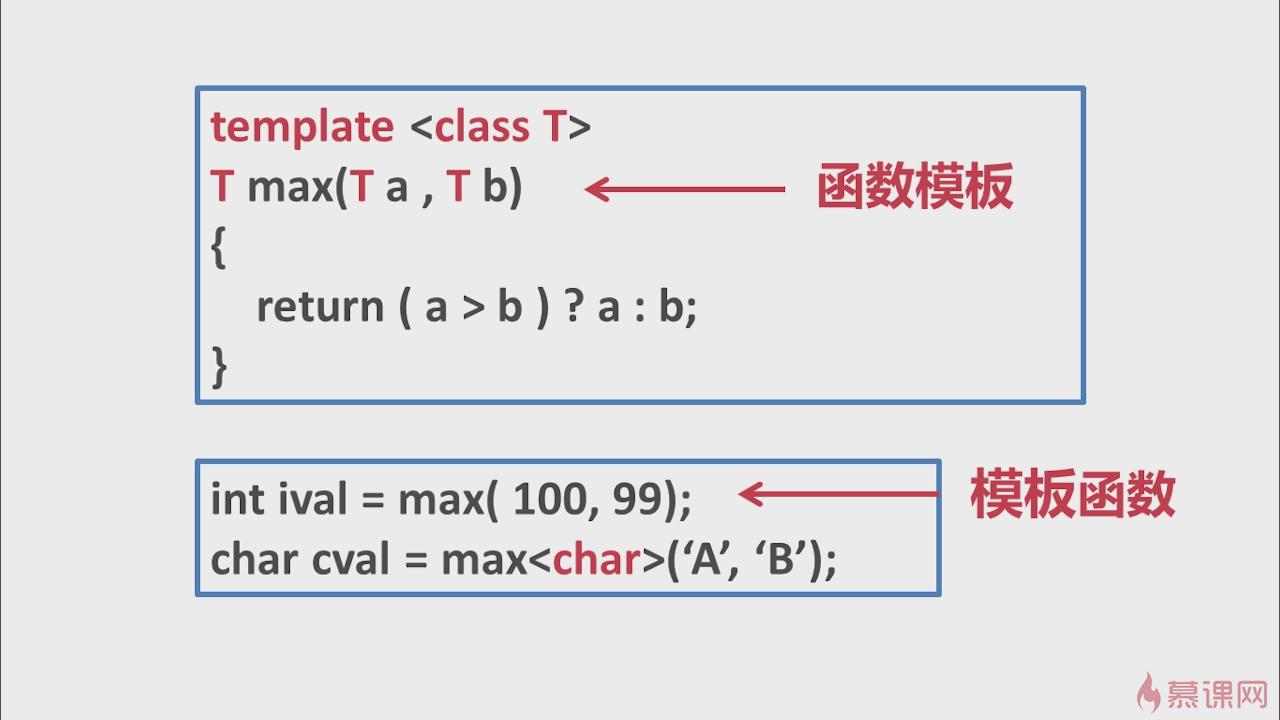
**在上图中，三个函数的目的是一样的，区别在于函数的参数类型不同，所以要引入函数模板：**

### 类型作为函数参数：

**关键字：template typename class**

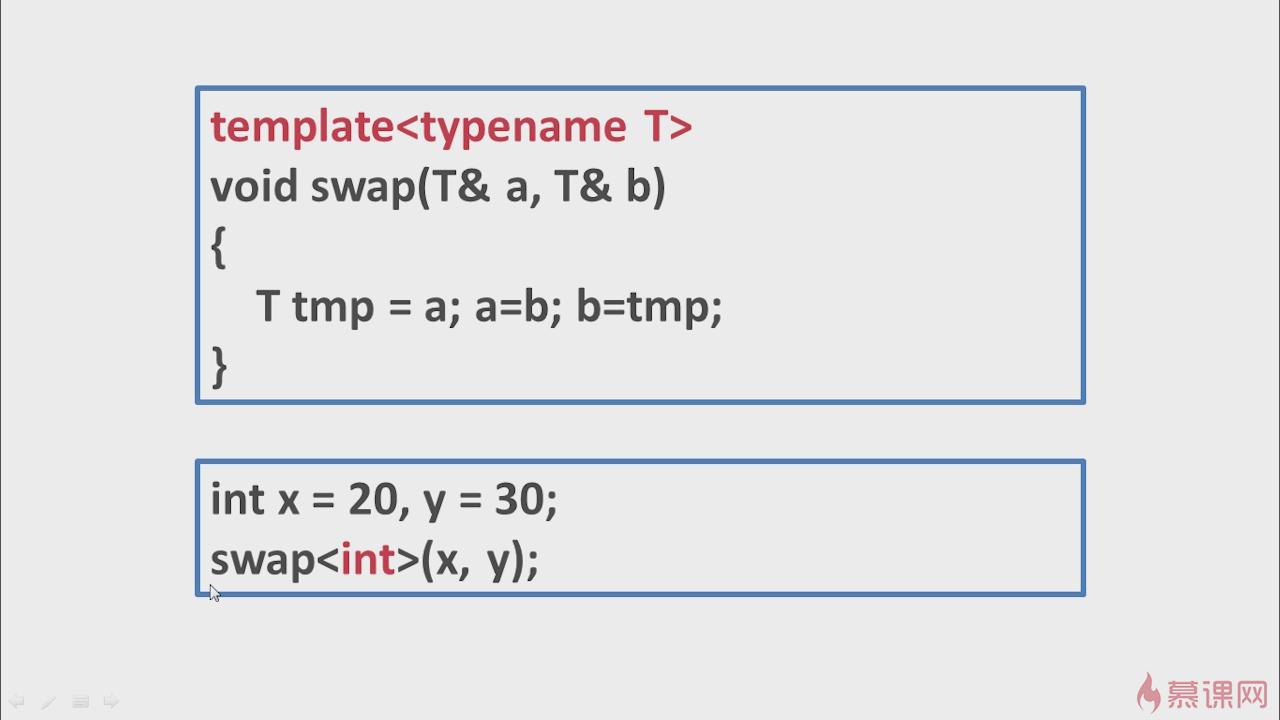


**例：**

****

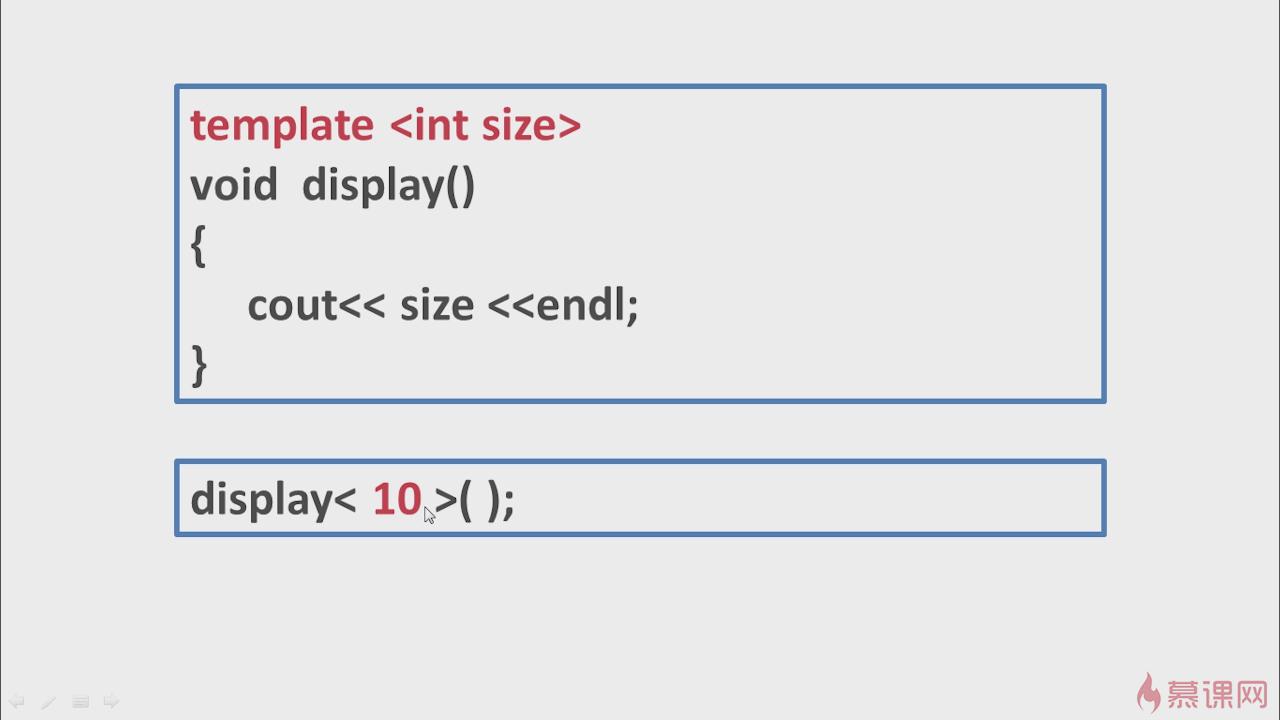
**注意函数模板和模板函数的区别：函数模板在确定好对应的参数类型后就成了模板函数。**

**注意上面的那个max函数的使用，当使用时没有指定参数类型的时候系统会根据使用的参数的值确定参数的类型，比如上面的使用int ival max（100,99）和 int ival max<int>(100,99)效果是一样的。**

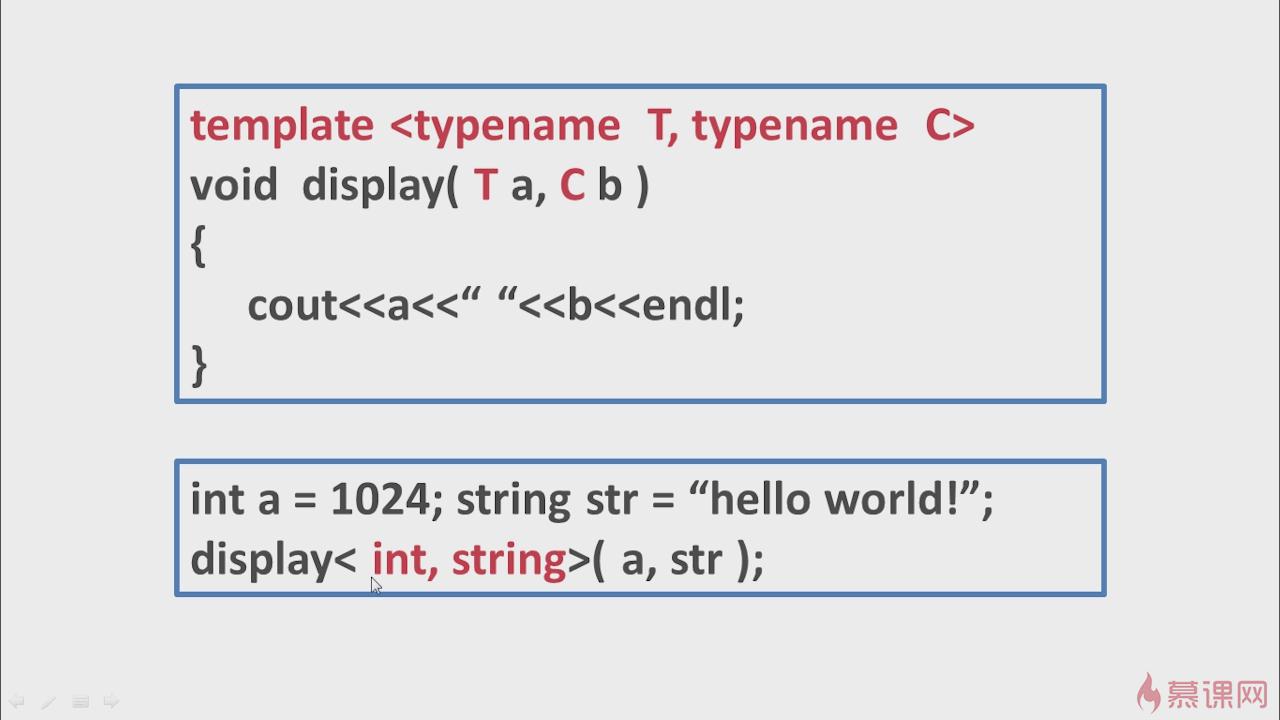
****

### 变量作为模板参数：

**例：**



### 多参数函数模板：

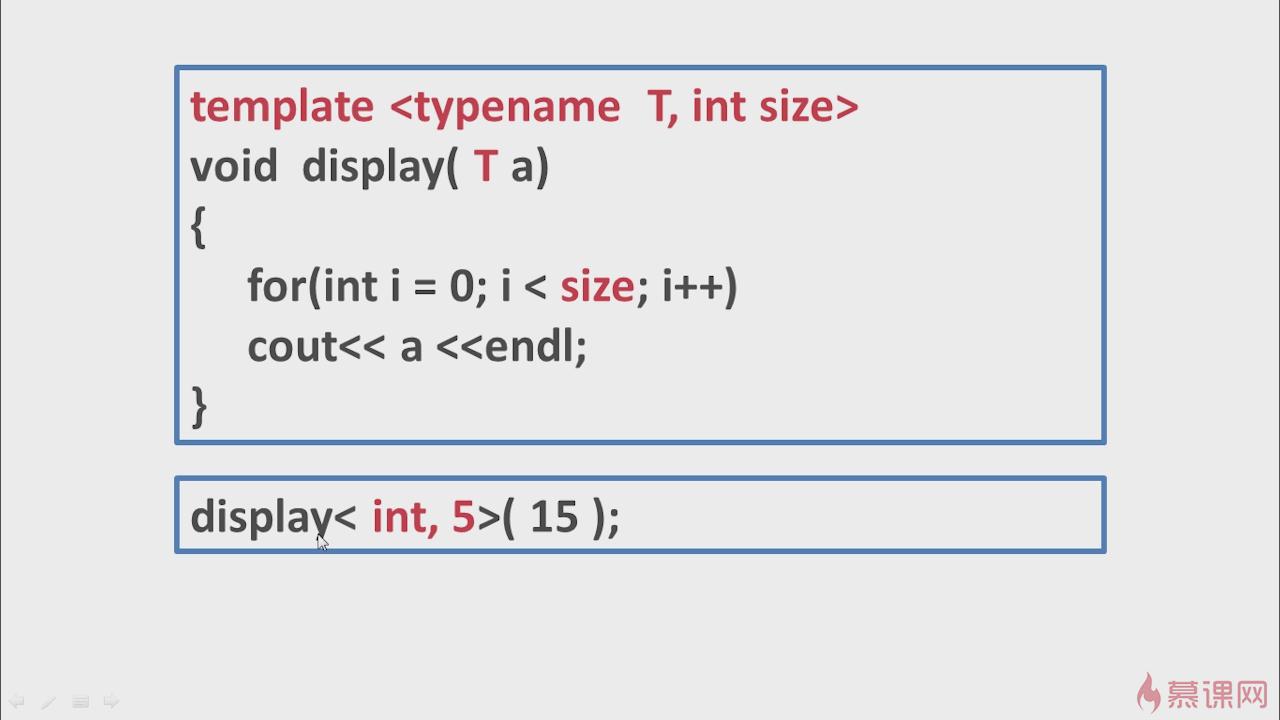


**注意，此时的typename不可省略**

**typename和class的作用是一模一样的：**

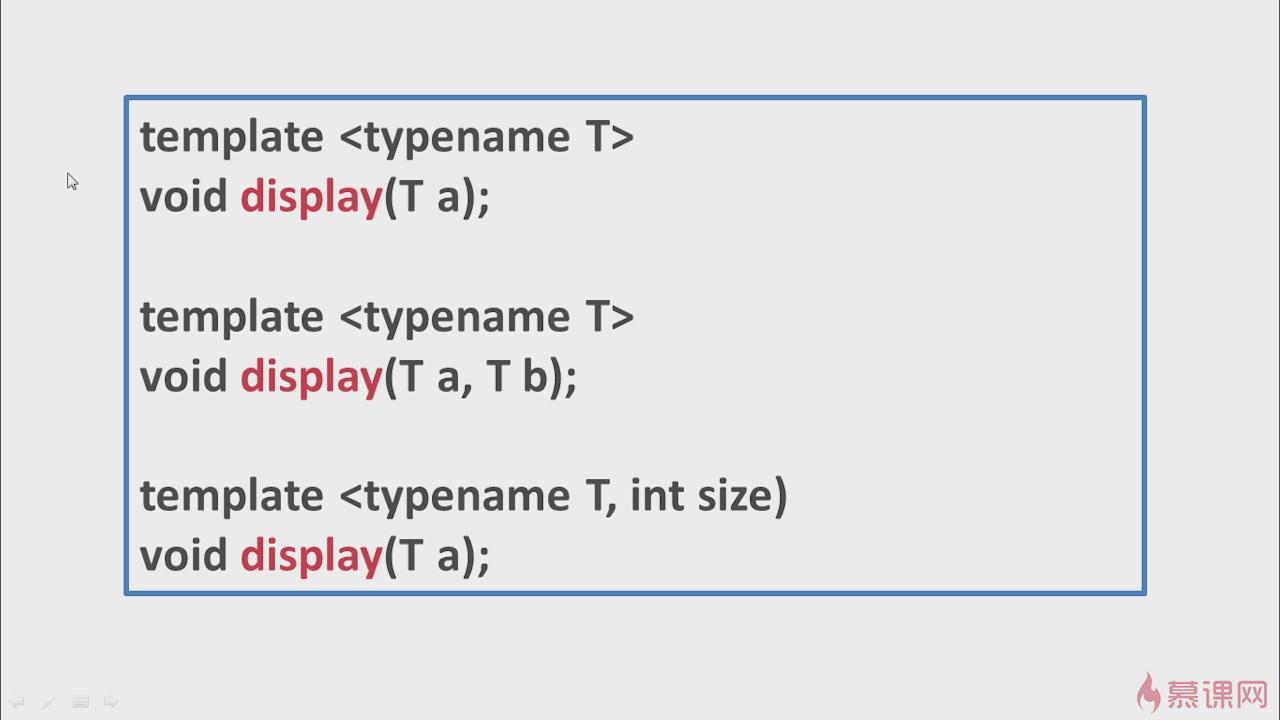
****

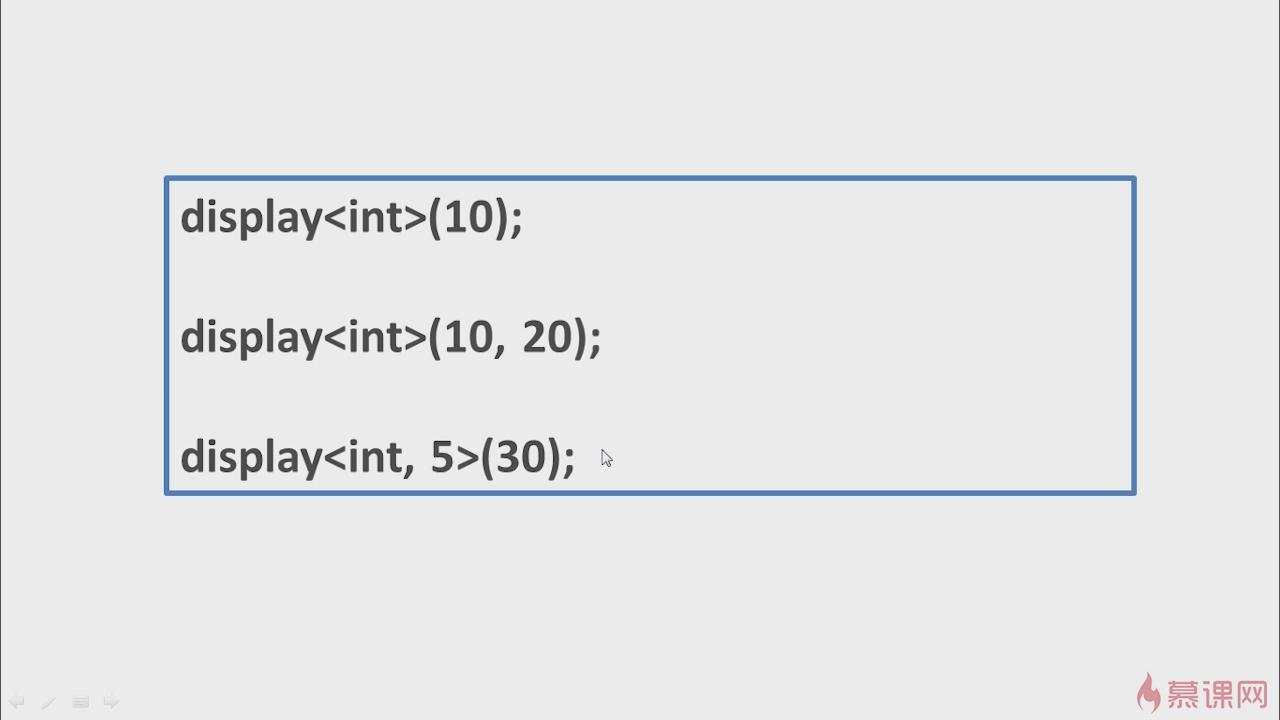
**类型作为模板参数和变量作为模板参数的混用：**

****

### 函数模板与重载：

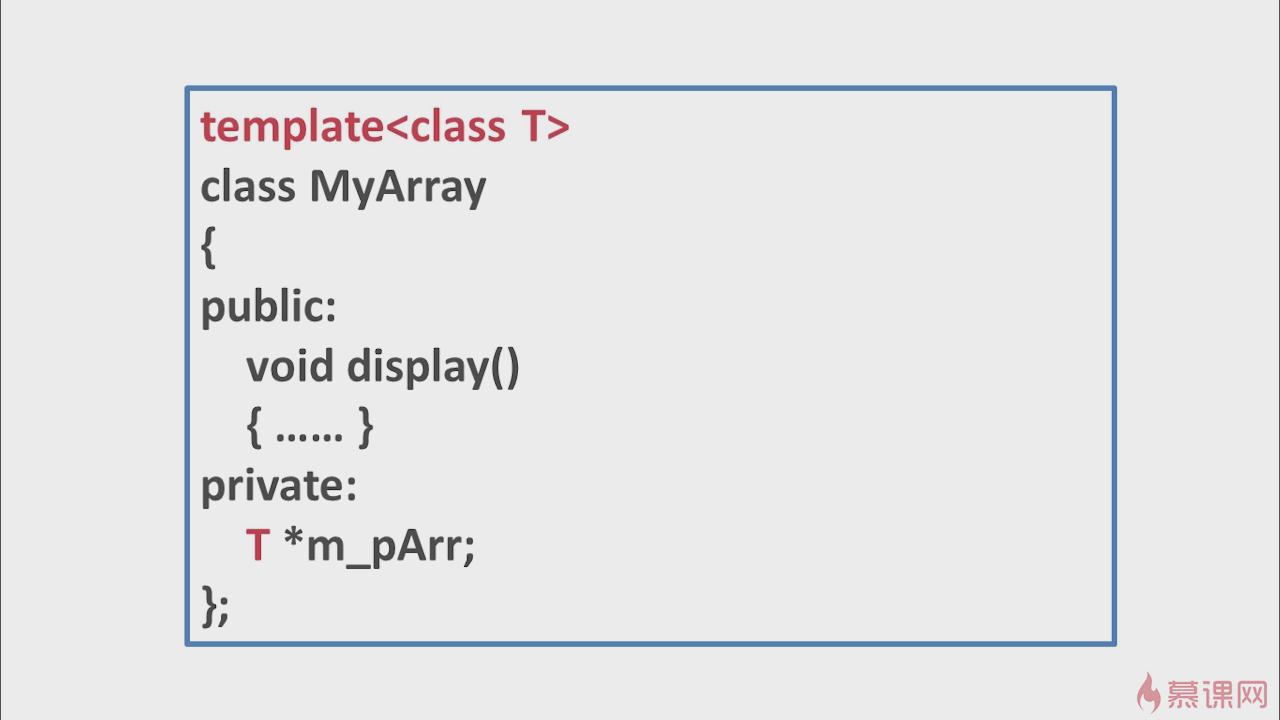
**定义：**



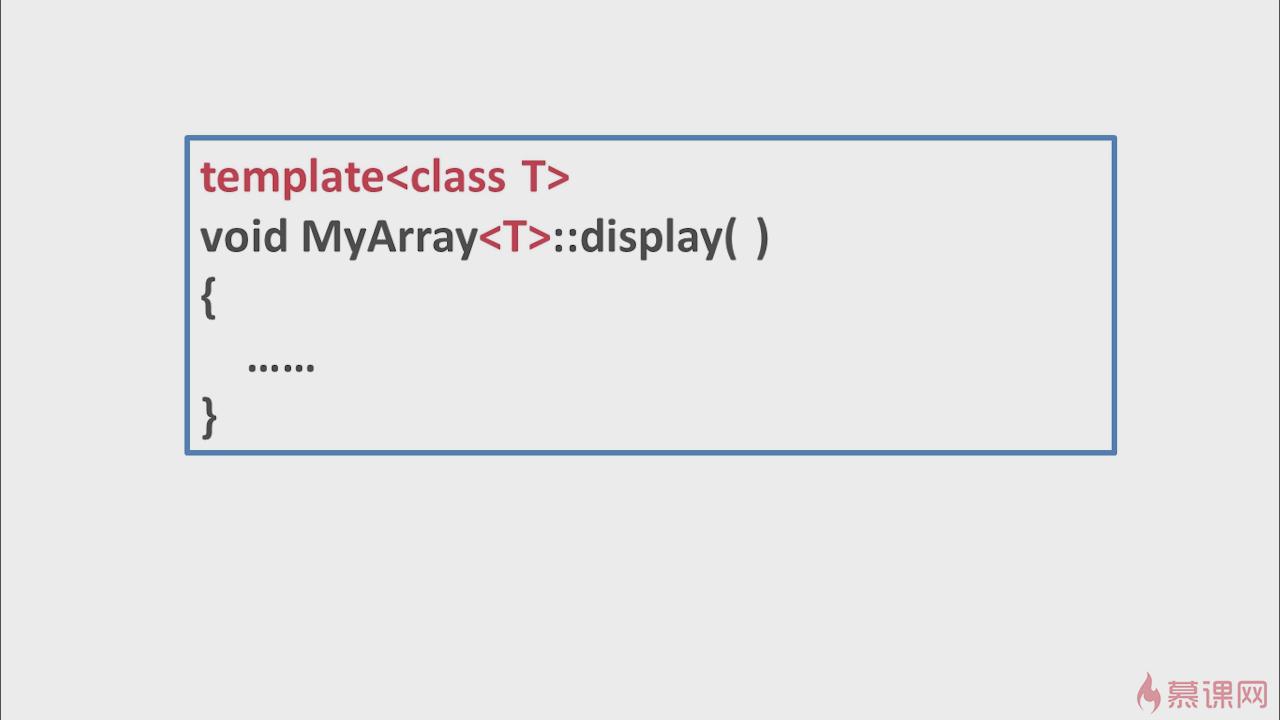
**用法：**

## 类模板

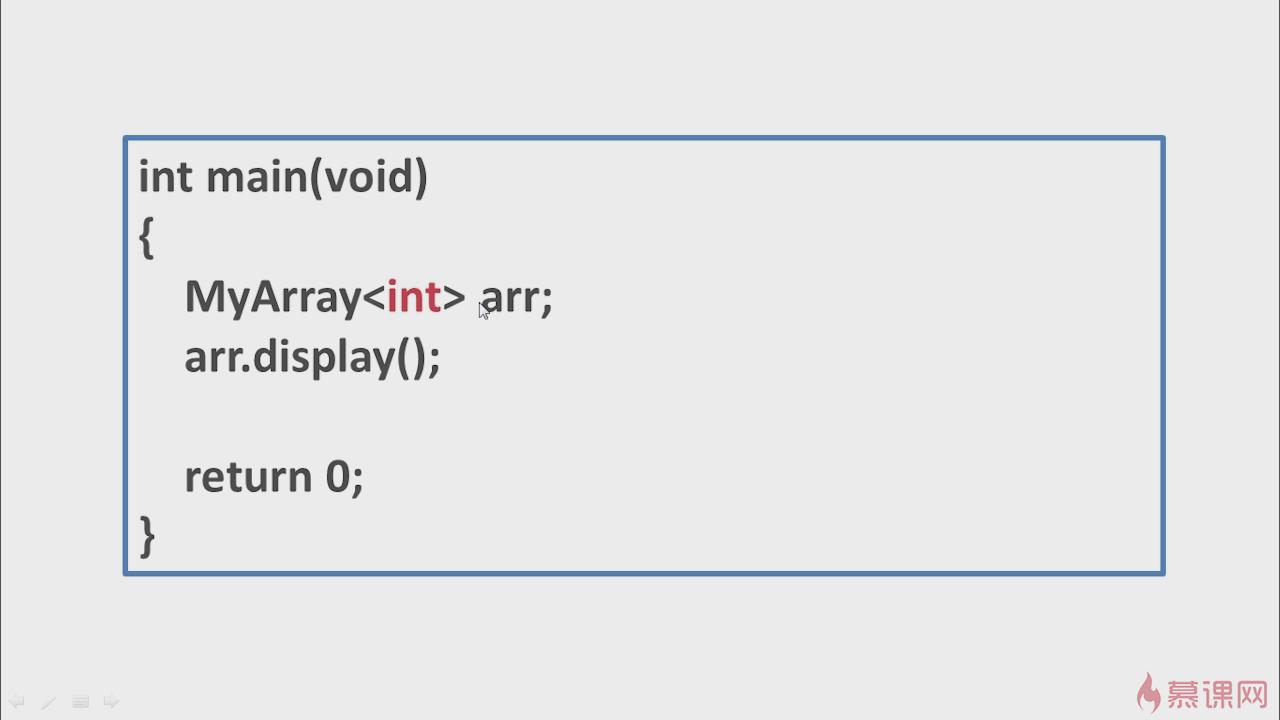
### 类型作为类模板参数的定义方法：



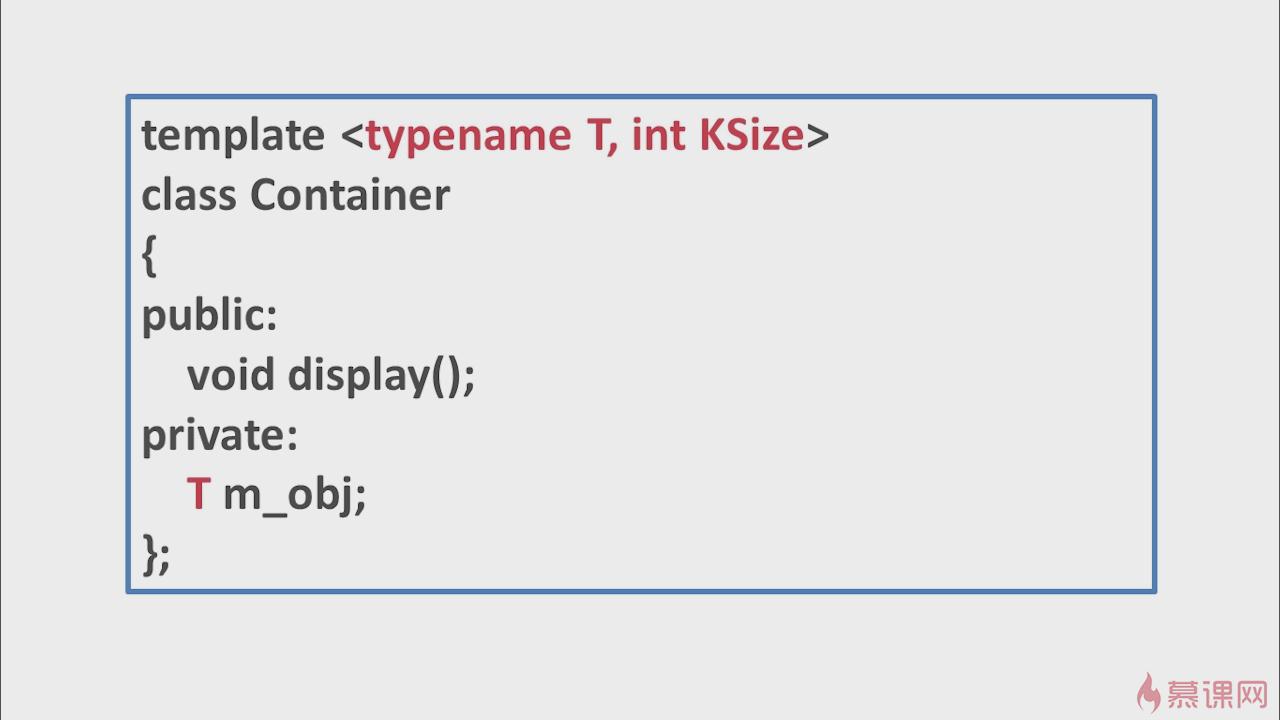
**注意：类模板中的成员函数在类内定义时和普通的类的定义方法没有什么区别，但是在类外定义时却有着很大的区别：**



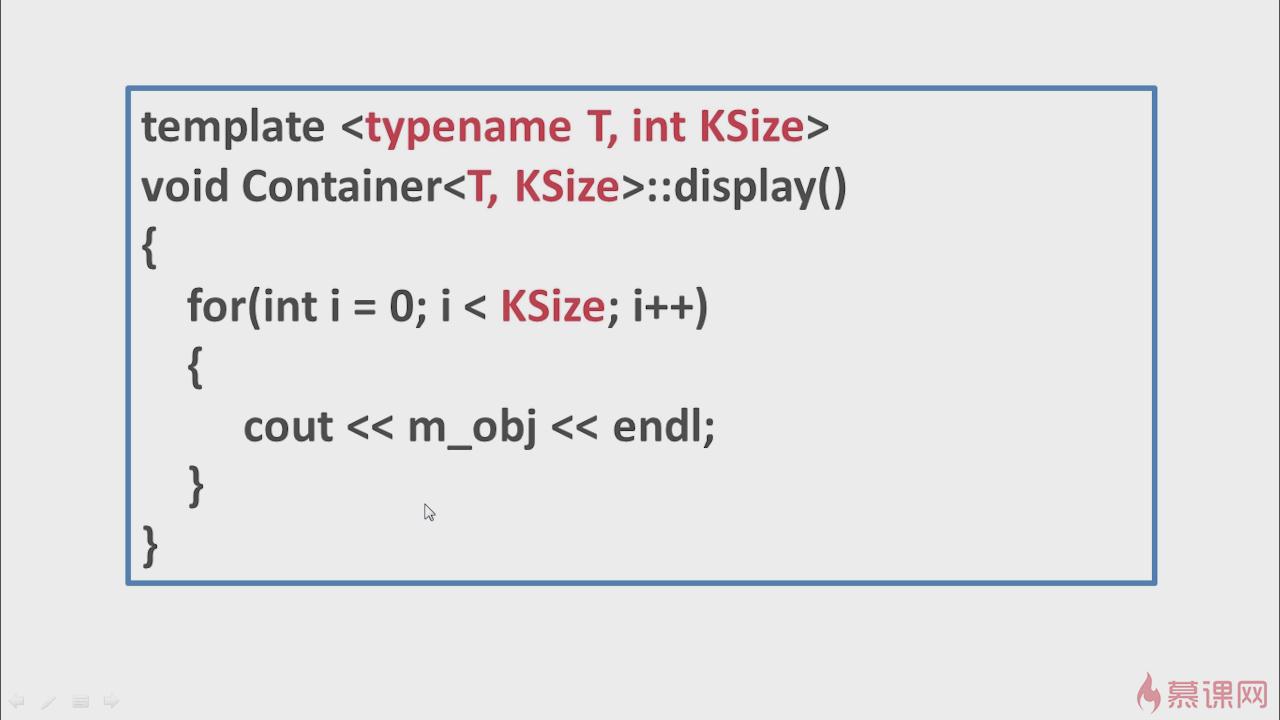
**使用方法：**

****

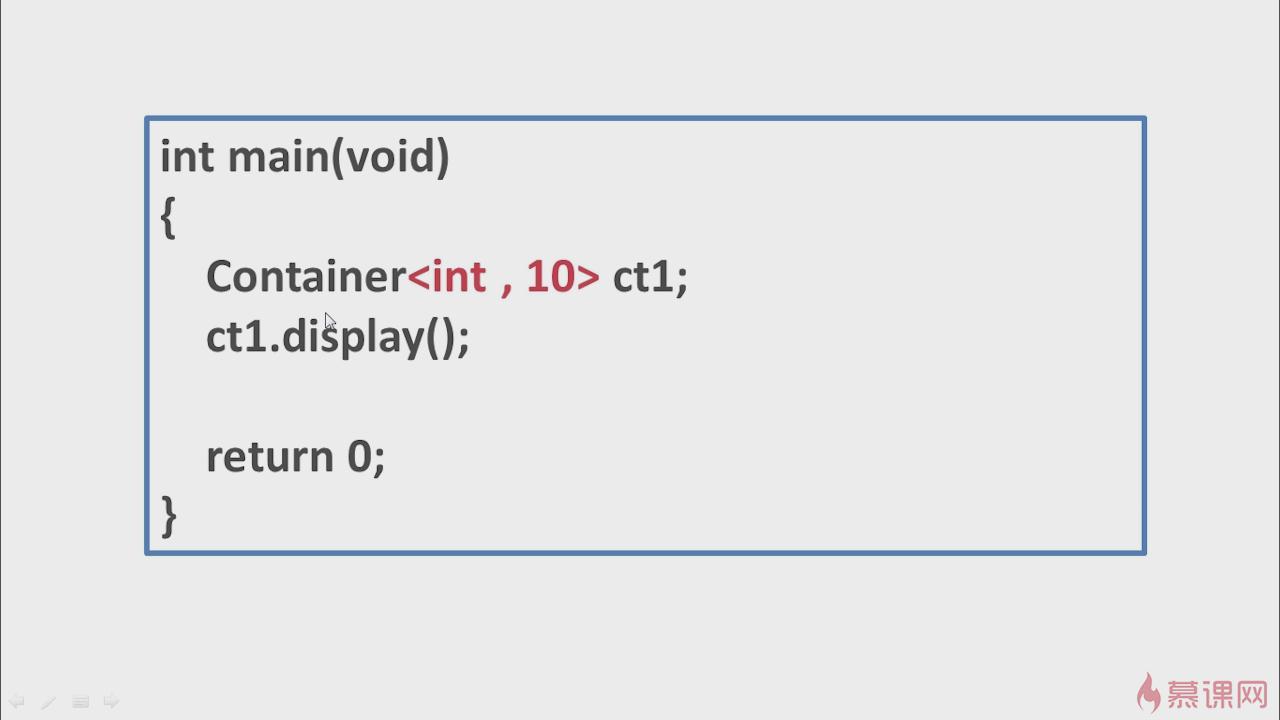
### 参数类型和变量一起作为类参数的定义方法：

****

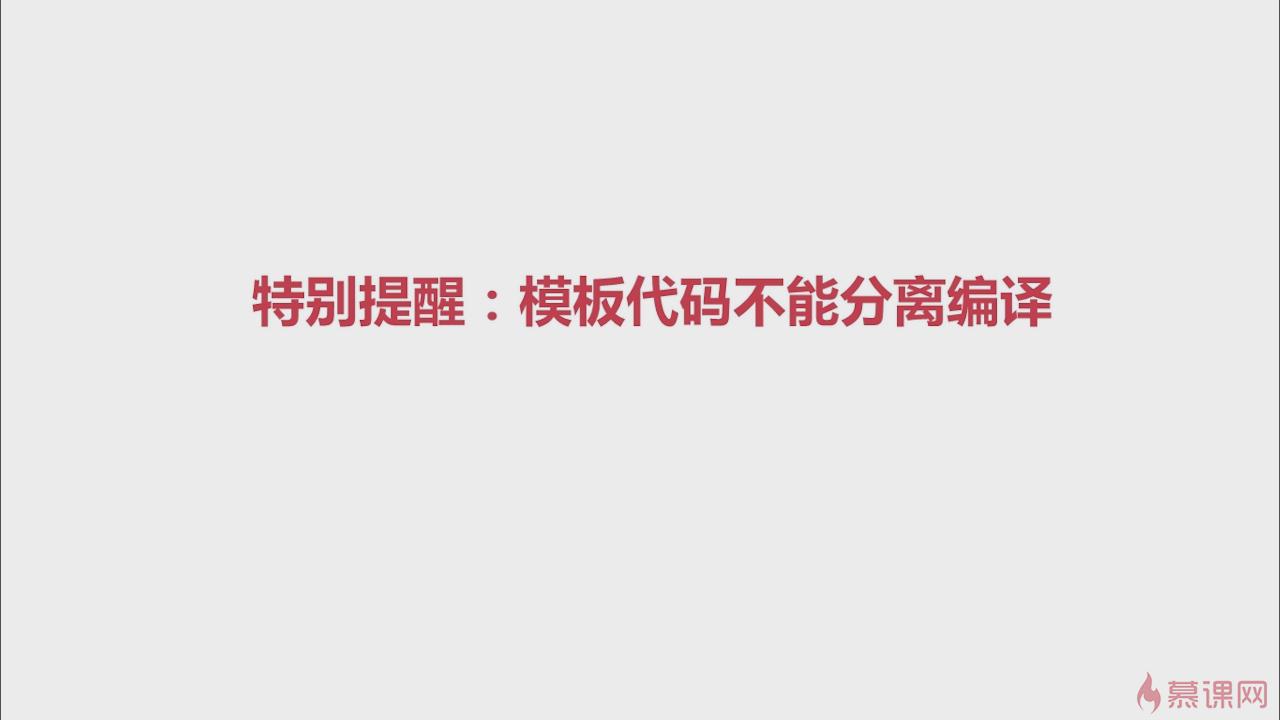
**成员函数的定义方法：**

****

**主函数中的使用方法：**

****

### 重点：

****

**这句话的意思是对于类模板要把类声明及定义写在一个文件（.h文件中），即就是原来的.h和.cpp文件的代码都写在一个文件.h中。**

**在vs、cs、dev编译器中都是这么要求的。**

# C++标准模板：

**STL:标准模板库**

**STL：Standard Template Lib**

## vector向量

**（使用时注意包含头文件<vector>）**

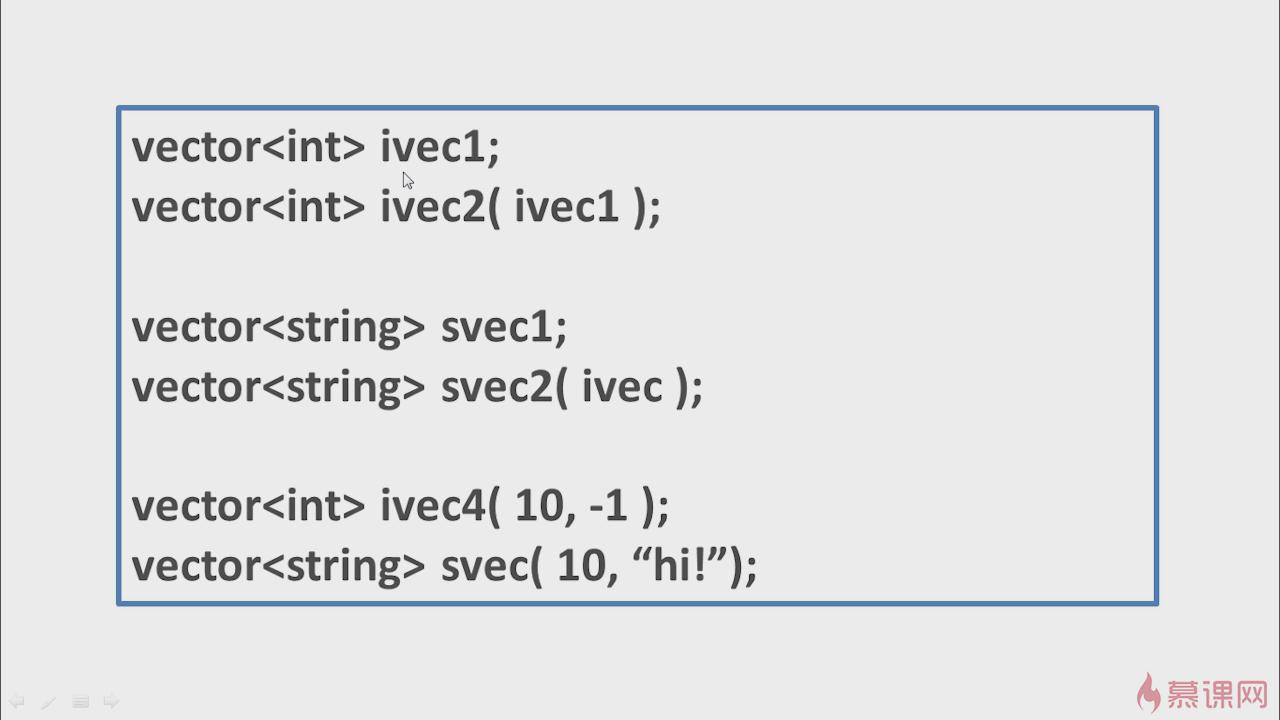
**向量的本质：对数组的封装**

**向量的特点：读取能在常数时间内完成**

**初始化vector对象的方法：**

****

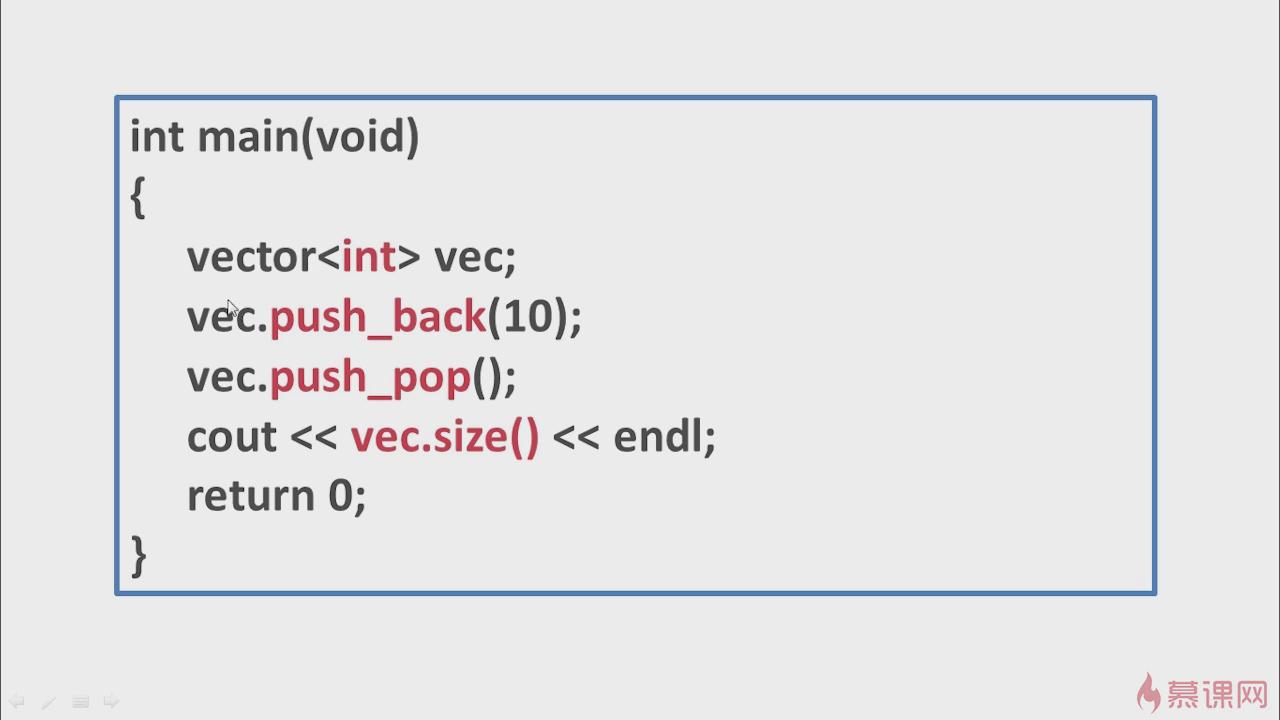
**初始化实例：**

****

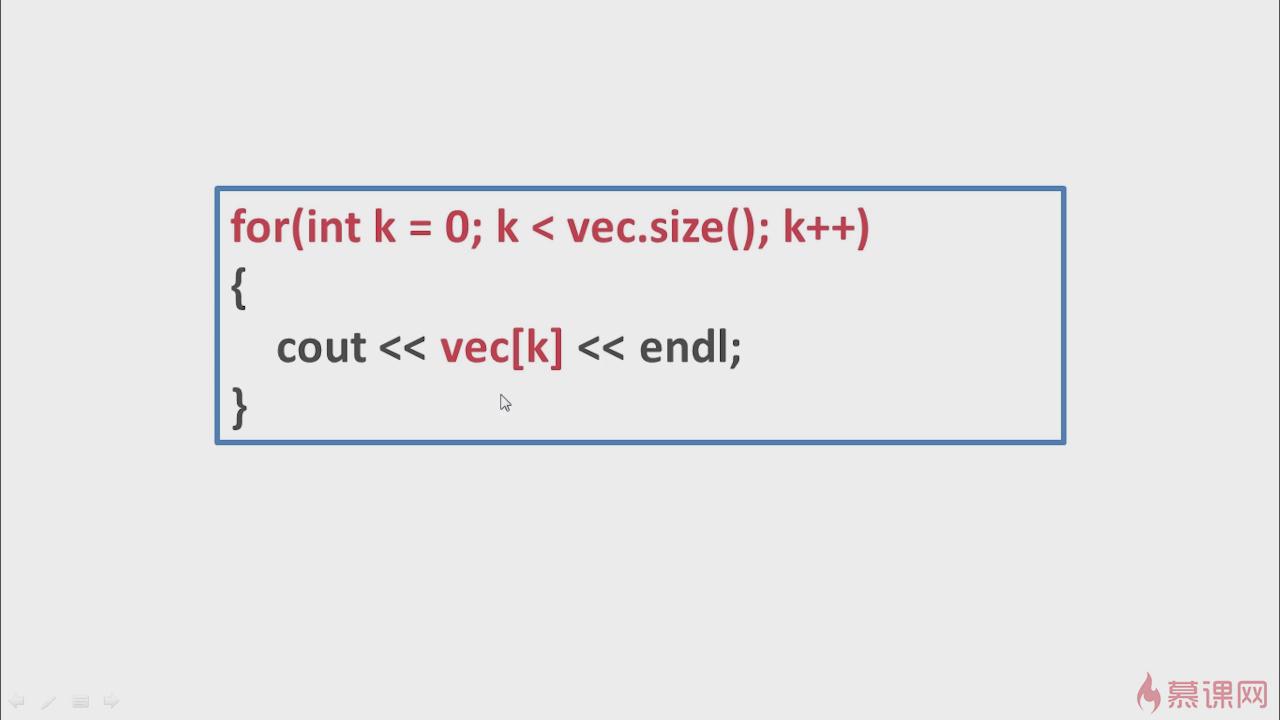
**vector常用函数：**

****

**Vector常用函数示例：**

****

**遍历vector向量示例：**

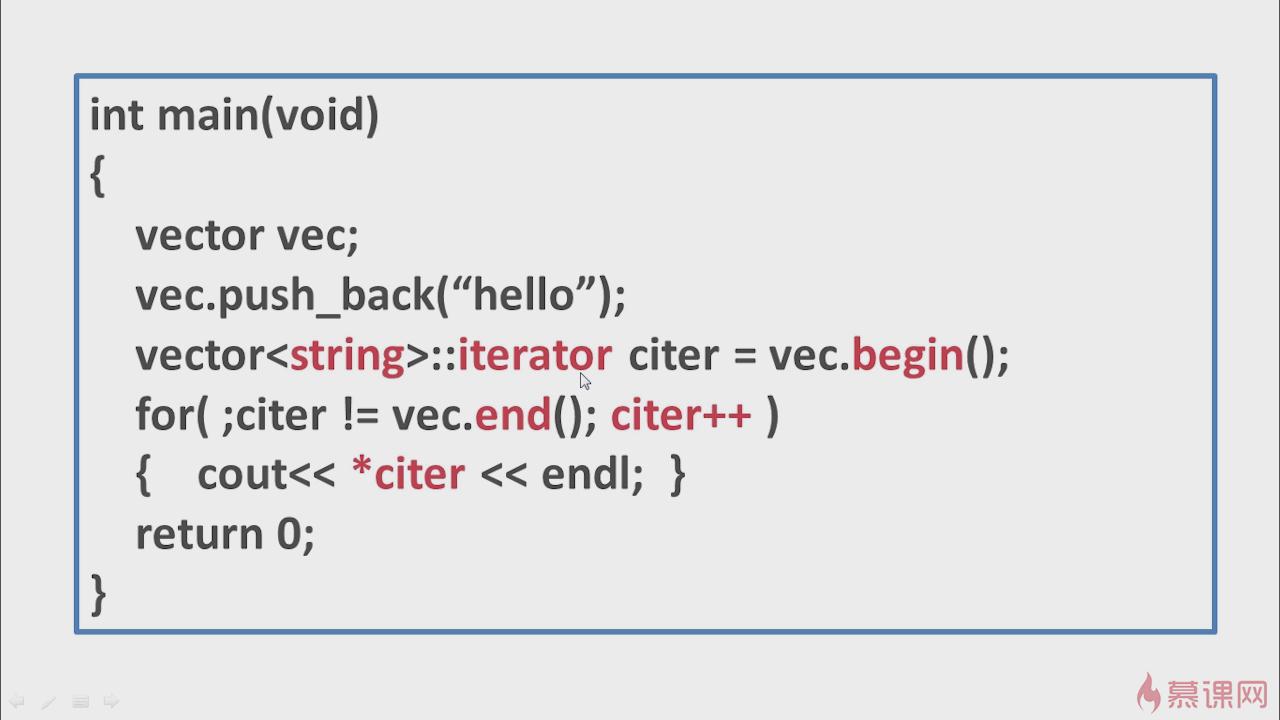
****

**要想遍历向量。除了上面那种方法之外还可以使用迭代器：**

## 迭代器：

**关键字：iterator**

**示例：**

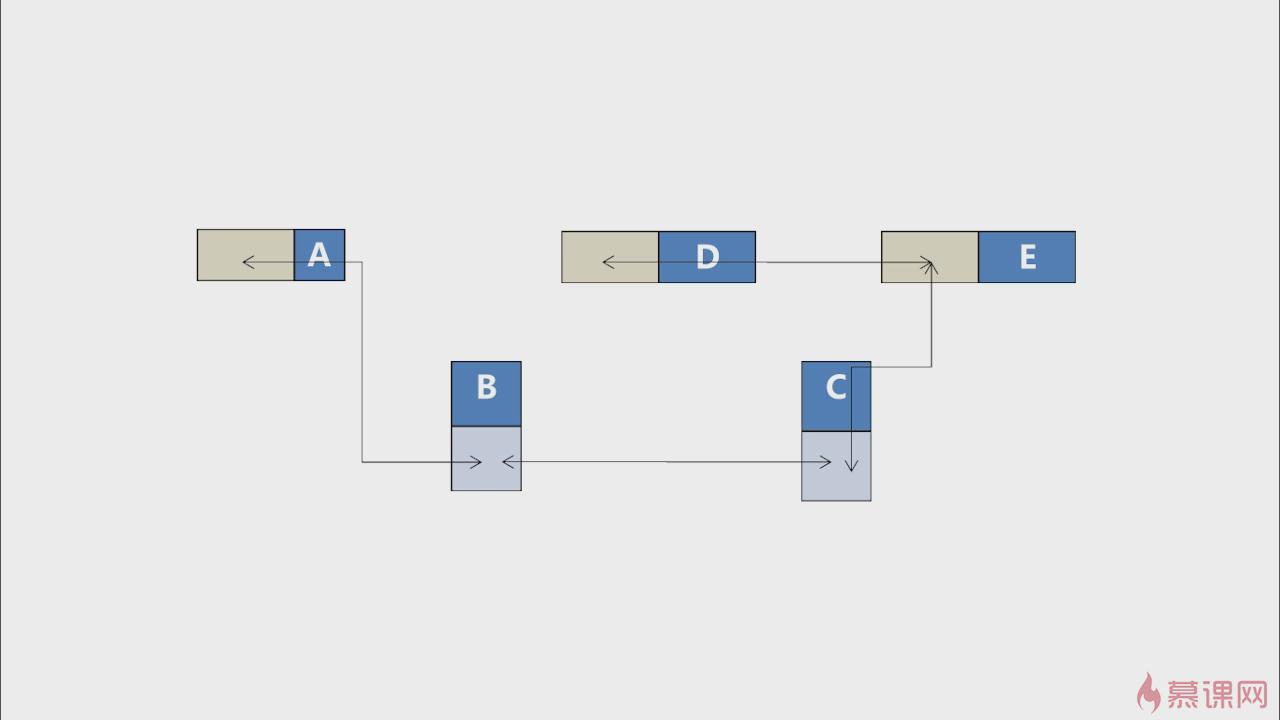


## 链表

**（使用时注意包含头文件<list>）**

**特点：插入数据速度快**

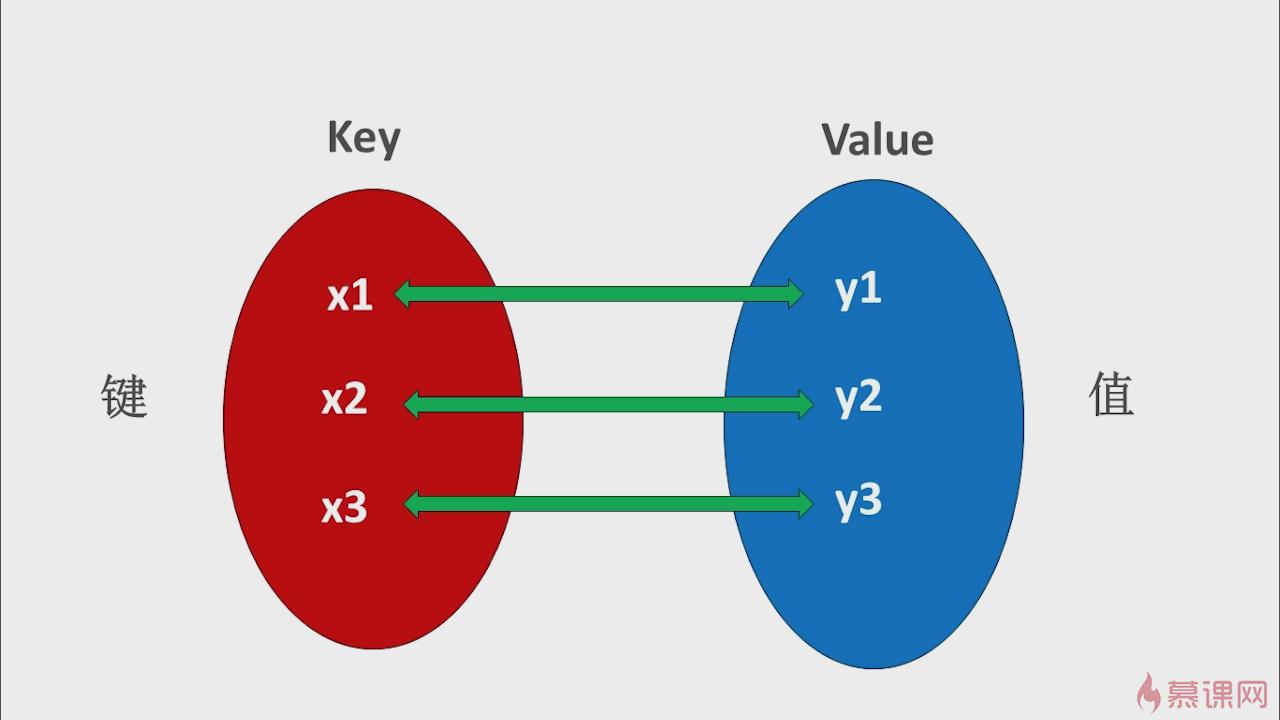
**结构图（原理示例图）：**



## 映射（map）：

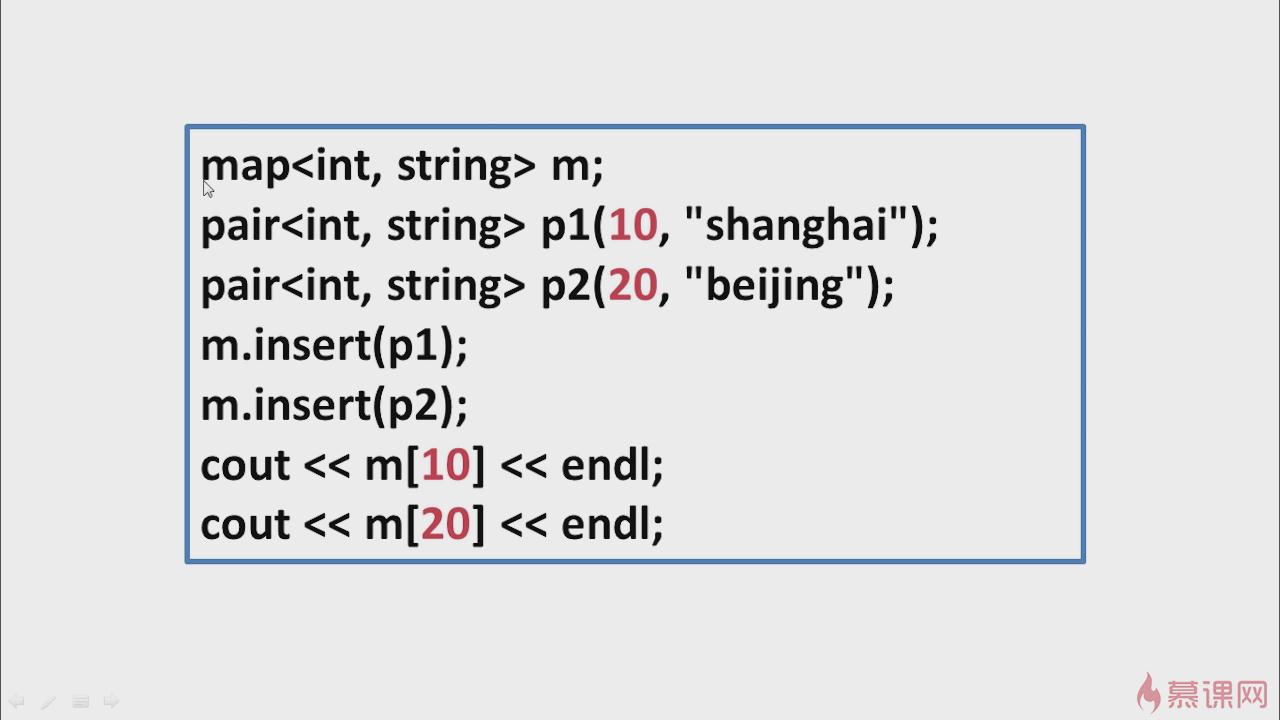
**（使用时注意包含头文件<map>）**

**原理示例图：**

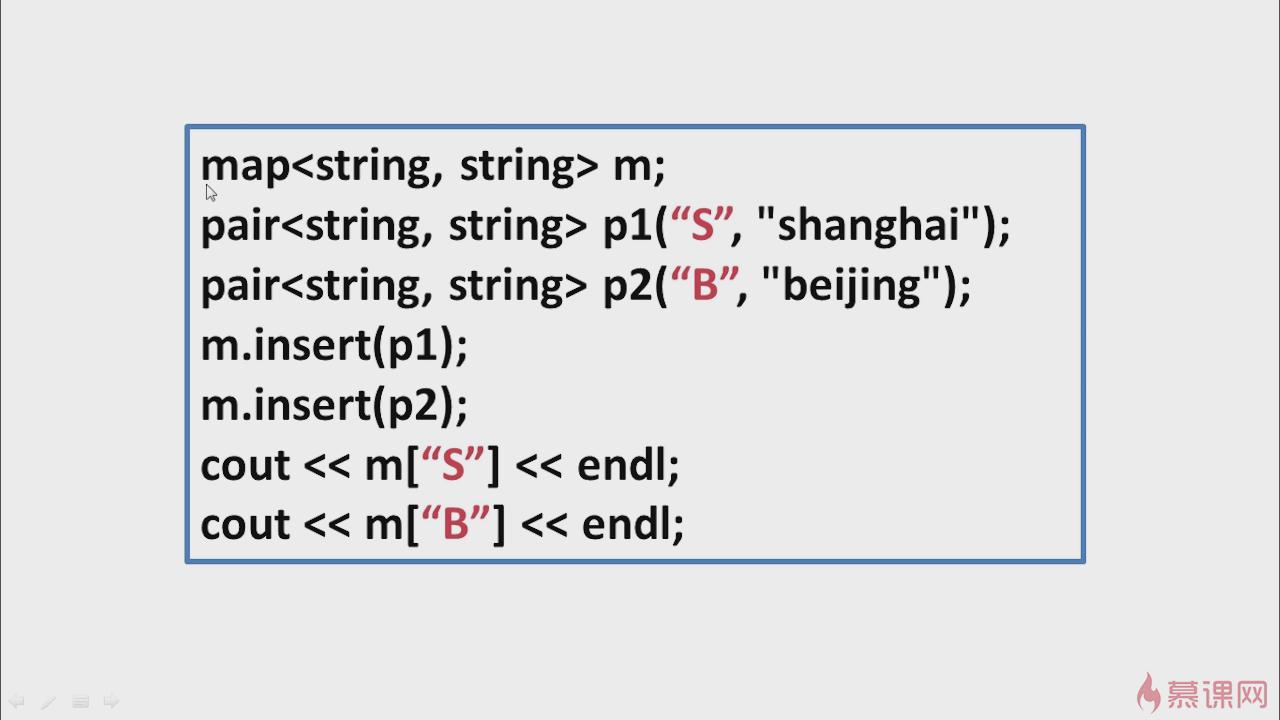


**映射的两种定义/使用方法：**

**方法1：（使用的时候类似于数组）**

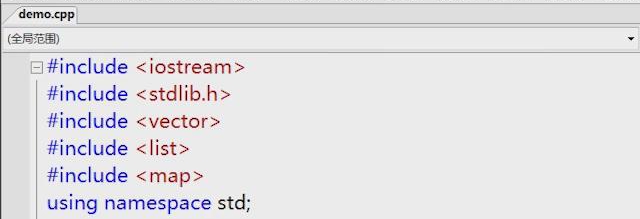


**方法二：（使用时与数组的区别）：**



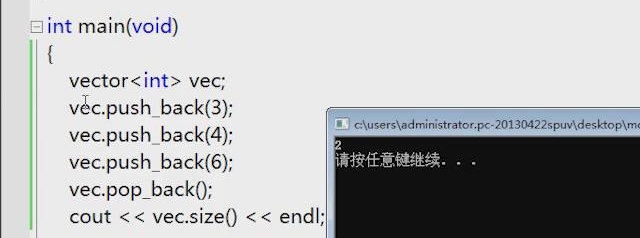
# C++标准模板示例：

**所有例子都是在包含以下这些头文件（使用字符时还要包含<string>）的前提下进行的**



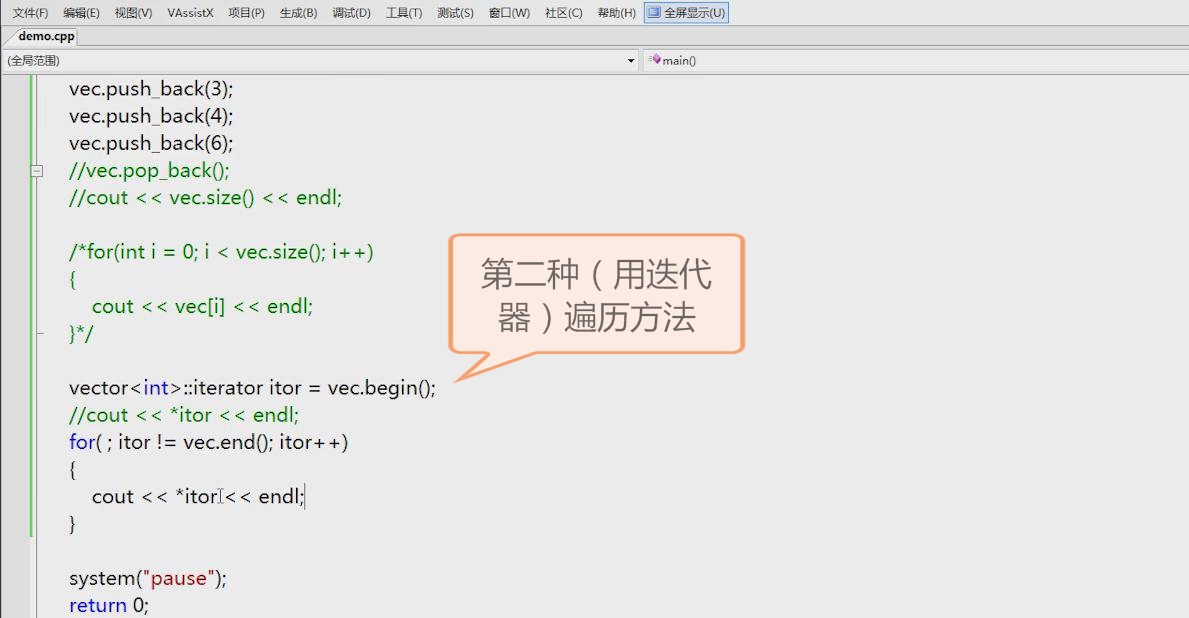
## 向量：

**定义及初始化方法：**

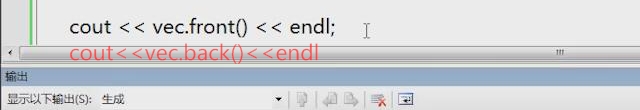


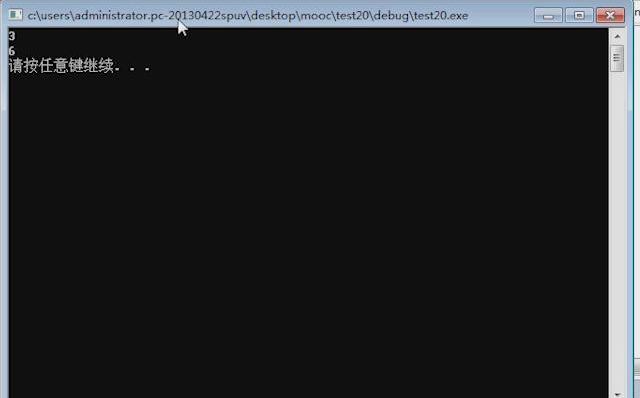
**遍历向量的方法：**





**两个函数的演示：**



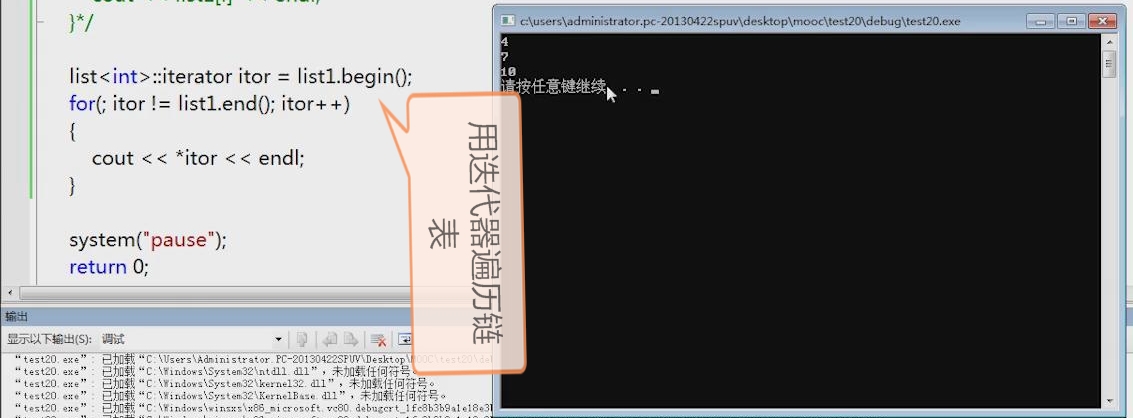


## 链表：

**定义及初始化方法：**

****

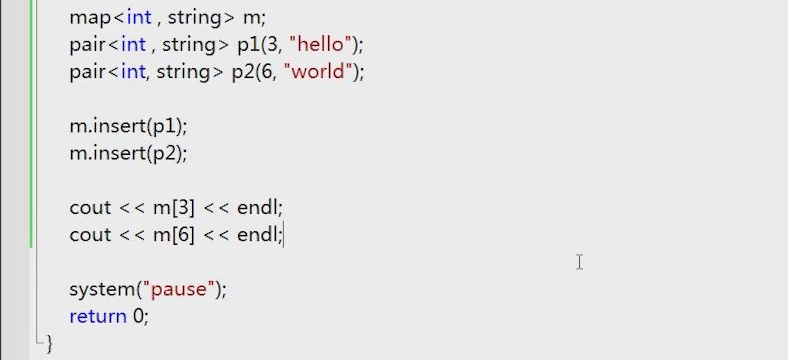
**链表只能用迭代器进行遍历：**



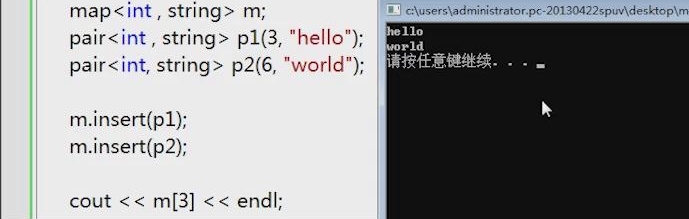
## 映射：

**定义及使用方法：**

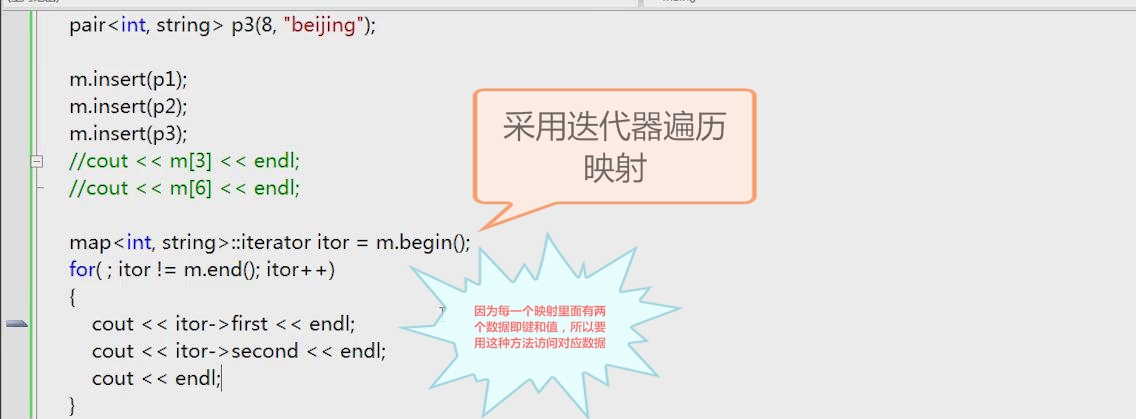
**首先定义“对”，关键字pair，然后用insert把定义的“对”放进映射中去：**



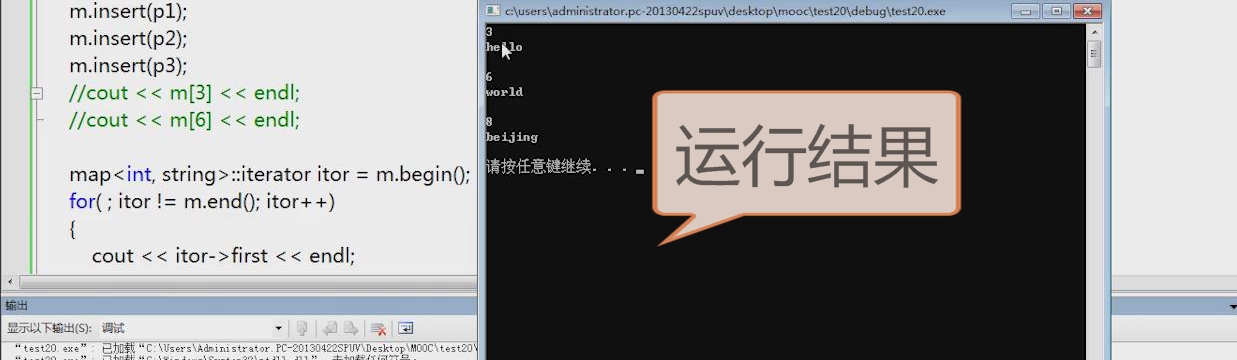
**运行结果：**



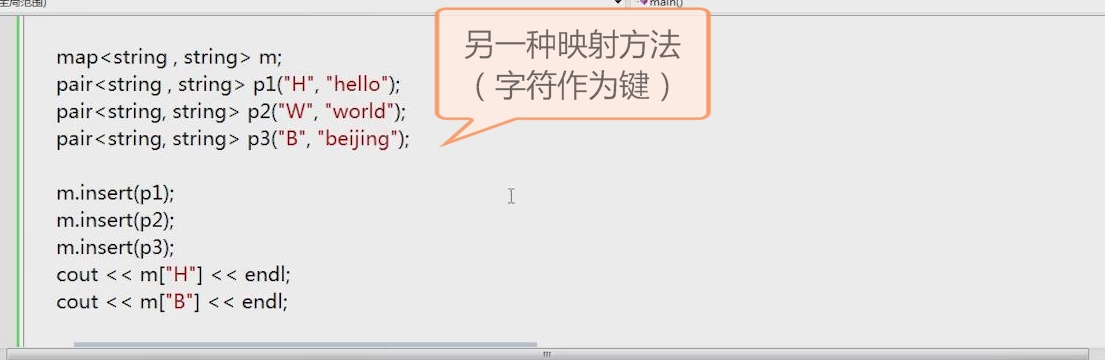
**遍历方法：**



**运行结果：**



**字符作为键的情况：**


**运行结果：**

