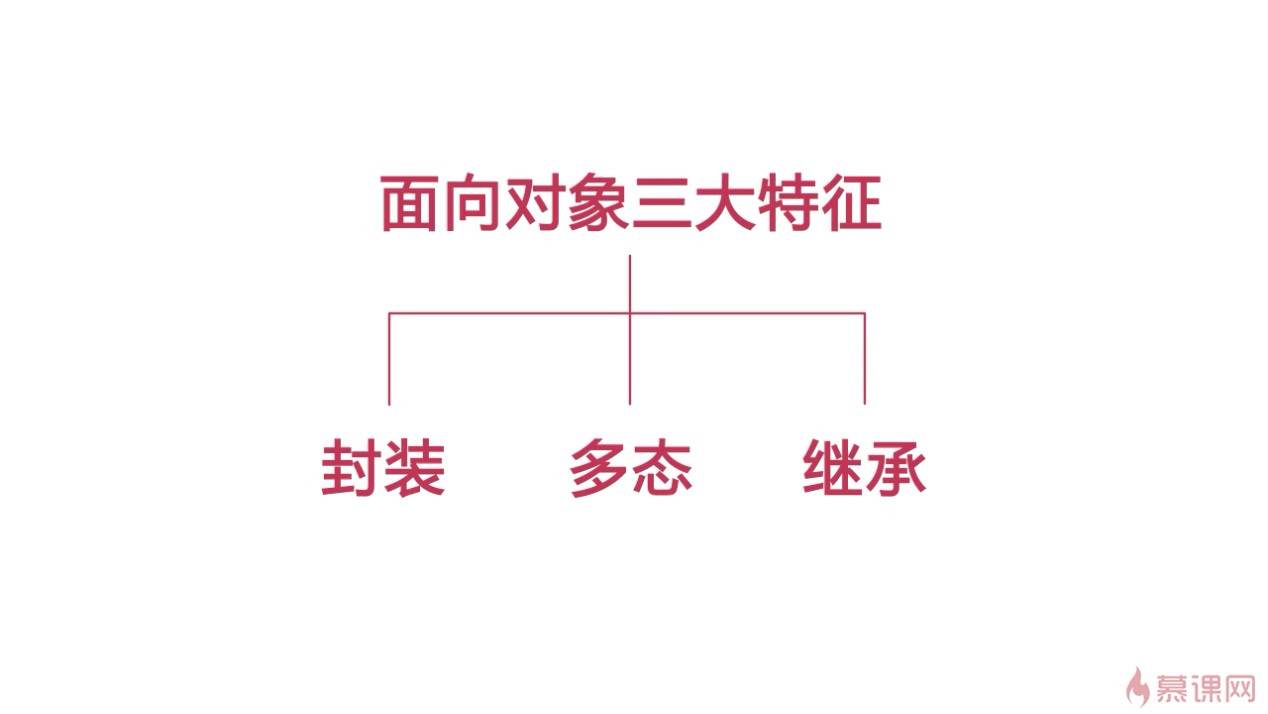
# 蛮重要的哦：

**面向对象的三大特征：**



**C，C++，C#三者区别**

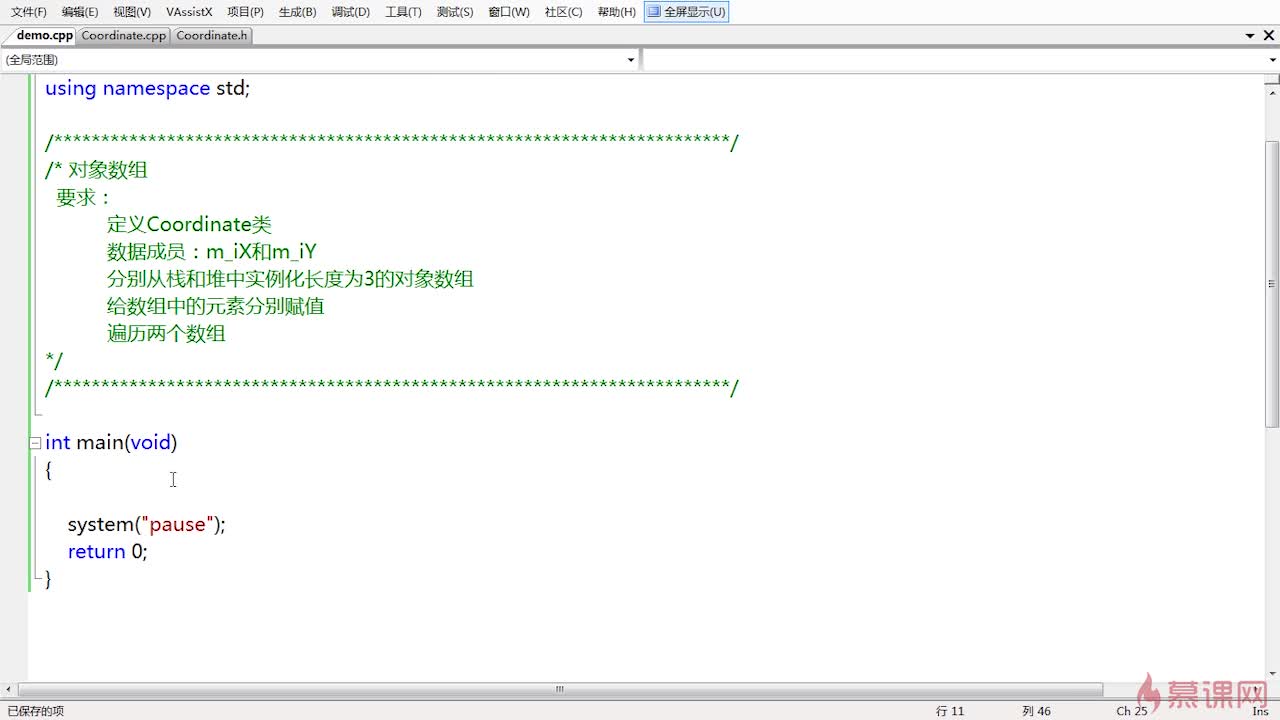
**继承关系是C->C++->C# C++完全向C兼容,C程序几乎不用修改即可在C++的编译器上运行.C++也称为带类的C,在C的基础上增加了许多面向对象的概念.虽然是C的扩展,但 并不意味着C功能不如C++,最牛的操作系统是用C写的(不是C++哦).**

**C#是微软弄的一个东东,继承了C和C++的许多东西,但和两者基本上 已完全不一样了.你可以把它当作一种全新的语言来学.**

**system就是调用从程序中调用系统命令(和shell命令)。**

**system("pause")就是从程序里调用“pause”命令;**

**而“pause”这个系统命令的功能很简单，就是在命令行上输出一行类似于“Press any key to exit”的字，等待用户按一个键，然后返回。**



内存的相关概念： **一个由c/C++编译的程序占用的内存分为以下几个部分   
1、栈区（stack）：由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。   
2、堆区（heap）： 一般由程序员分配释放（比如程序员用malloc和new开辟内存）， 若程序员不释放，程序结束时可能由**[**OS**](https://www.baidu.com/s?wd=OS&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4Phcvn1Ihm1ubnAwbPAmY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD)**回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。   
3、全局区（静态区）（static）：，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 —— 程序结束后由系统释放   
4、文字常量区：常量字符串就是放在这里的。 程序结束后由系统释放   
5、程序代码区：存放函数体的二进制代码。**

**堆和栈：**

**例：  
//main.cpp   
int a = 0; 全局初始化区   
char \*p1; 全局未初始化区   
main()   
{   
int b; 栈   
char s[] = "abc"; 栈   
char \***[**p2**](https://www.baidu.com/s?wd=p2&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4Phcvn1Ihm1ubnAwbPAmY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD)**; 栈   
char \***[**p3**](https://www.baidu.com/s?wd=p3&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4Phcvn1Ihm1ubnAwbPAmY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD)**= "123456"; 123456\0在常量区，p3在栈上。   
static int c =0； 全局（静态）初始化区   
p1 = (char \*)malloc(10); //堆  
p2 = (char \*)malloc(20); //堆  
分配得来得10和20字节的区域就在堆区。   
strcpy(p1, "123456"); 123456\0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。   
}**

为什么说c语言比c++更加高效？

**效率的高低不在于语言提供了什么特性或者你用了语言的什么特性，而在于你多做了多少**

**不必要的事情做得越多，效率就越低。当一门语言提供了一些特性让你可以非常容易和无痛地做不必要的事情的时候，程序员让语言做不必要事情的几率就会大大增加，因此语言的效率就会降低。C++ 给程序员提供了很多便利，自然也会导致程序员能够更容易地引入不必要的操作。**

**带参数的宏：**

**通过使用参数，可以创建外形和作用都和函数相似的类函数宏。宏的参数也用圆括号括起来，随后这些参数出现在替换部分。示例程序中下面语句就是类函数宏定义的示例：**

**#define PRINT(x) printf("x is:%d\n", x)**

**对比：**

**#define PRINT printf("x is %d\n", x)**

**两者不同的地方在于，PRINT是个常量字符串，它只打印名为x的变量，而使用带有参数的类函数宏PRINT(x)时，宏定义中的x由程序调用的符号代替。使用这个宏时既可以使用x，也可以自由的使用其他符号。**

**比如当使用PRINT(100)时，宏展开的结果是：**

|  |
| --- |
| **printf("x is %d\n", 100);** |

**如下是一个求平方的带参数类函数宏：**

|  |
| --- |
| **#define SQUARE(x) x\*x** |

**当x为4，我们尝试输出SQUARE(x)的值时，发现输出的是16，看起来好像是正确的。但是当x为4，输出SQUARE(x + 2)的值时，输出的竟然是14！在我们看来，SQUARE(x + 2)应该是6\*6，14甚至都不是一个整数的平方！**

**一个简单的原因是，预处理器不进行计算，只进行字符串替换。在出现x的地方，预处理器都用x+2进行替换。因此，在x等于4的情况下调用SQUARE(x+2)时，x\*x变成x+2\*x+2，因而得到14。如下修改宏定义可以让SQUARE(x + 2)的值为36：**

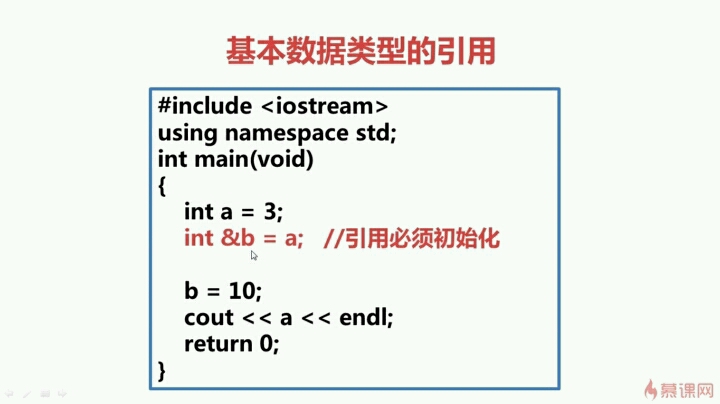
|  |
| --- |
| **#define SQUARE(x) (x)\*(x)** |

**从中得到的经验是：使用必需的足够的圆括号来保证以足够的顺序进行运算和结合。**

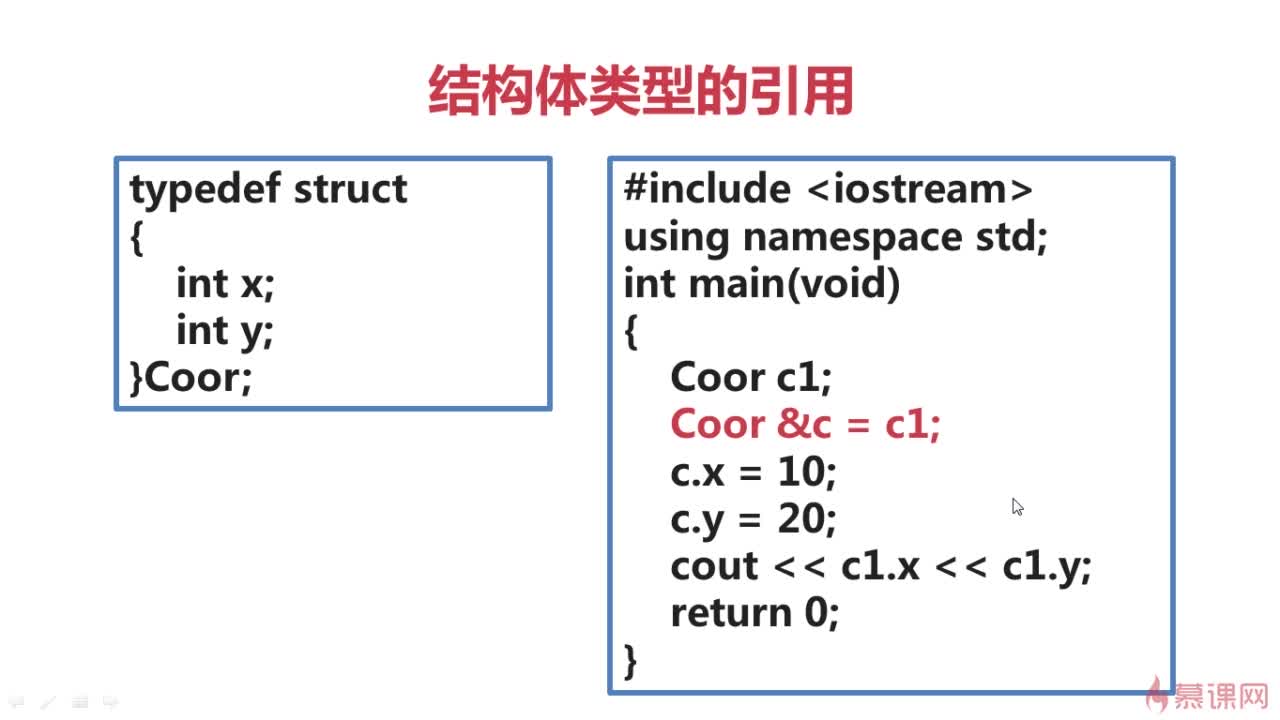
# 引用

（1）：引用概述**：引用就是变量的别名，引用必须初始化！！**

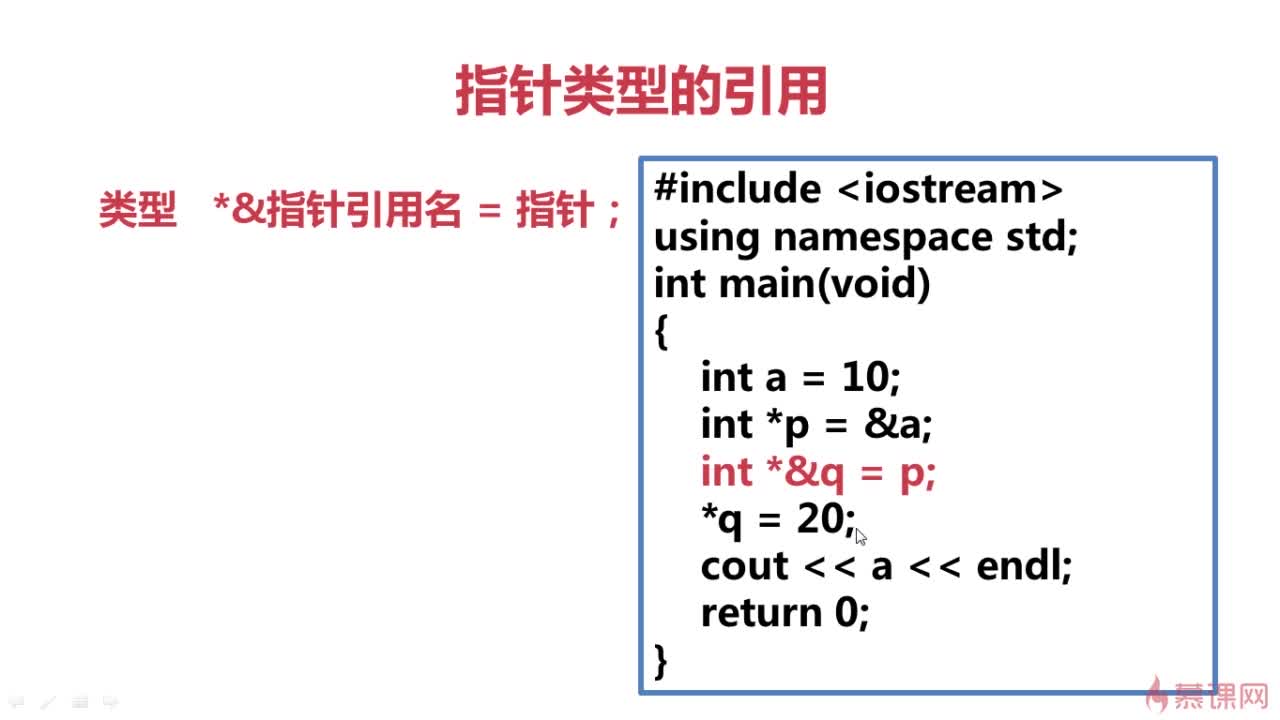
**整型的引用：**



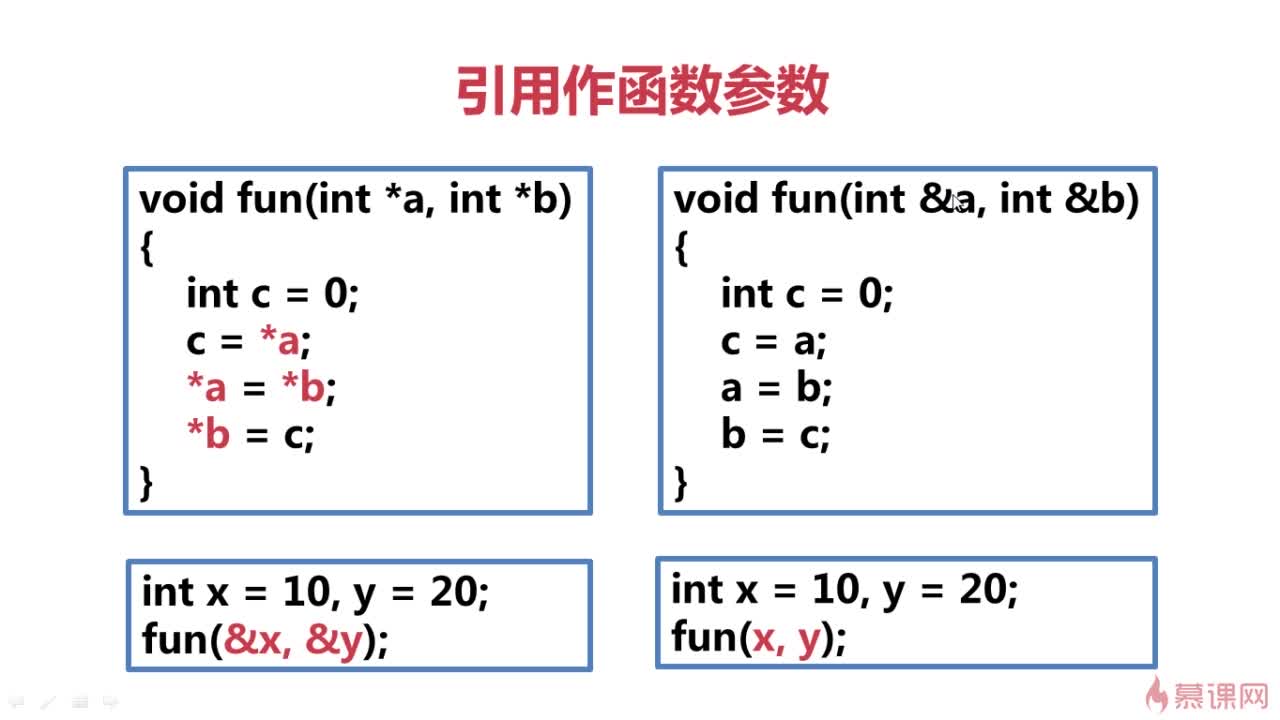
## 结构体类型的引用：



## 指针的引用：



## 引用做函数参数：



**上面一个引用做函数参数的意思是：void fun(int &a,int &b),意思是形参是两个引用值，比如调用时有：fun(x,y),即&a=x,&b=y;这就实现了引用，其中a,b是形参，即x，y的别名，这时相当于a，b**

**实际上，在虚实结合时是把实参x的地址传到形参a，使形参a的地址取实参x的地址，从而使a和x共享同一单元。同样，将实参y的地址传到形参b，使形参b的地址取实参y的地址，从而使b和y共享同一单元。这就是地址传递方式。为便于理解，可以通俗地说：把变量x的名字传给引用变量a，使a成为x的别名。这样就可以和指针一样改变主函数中的变量的值**

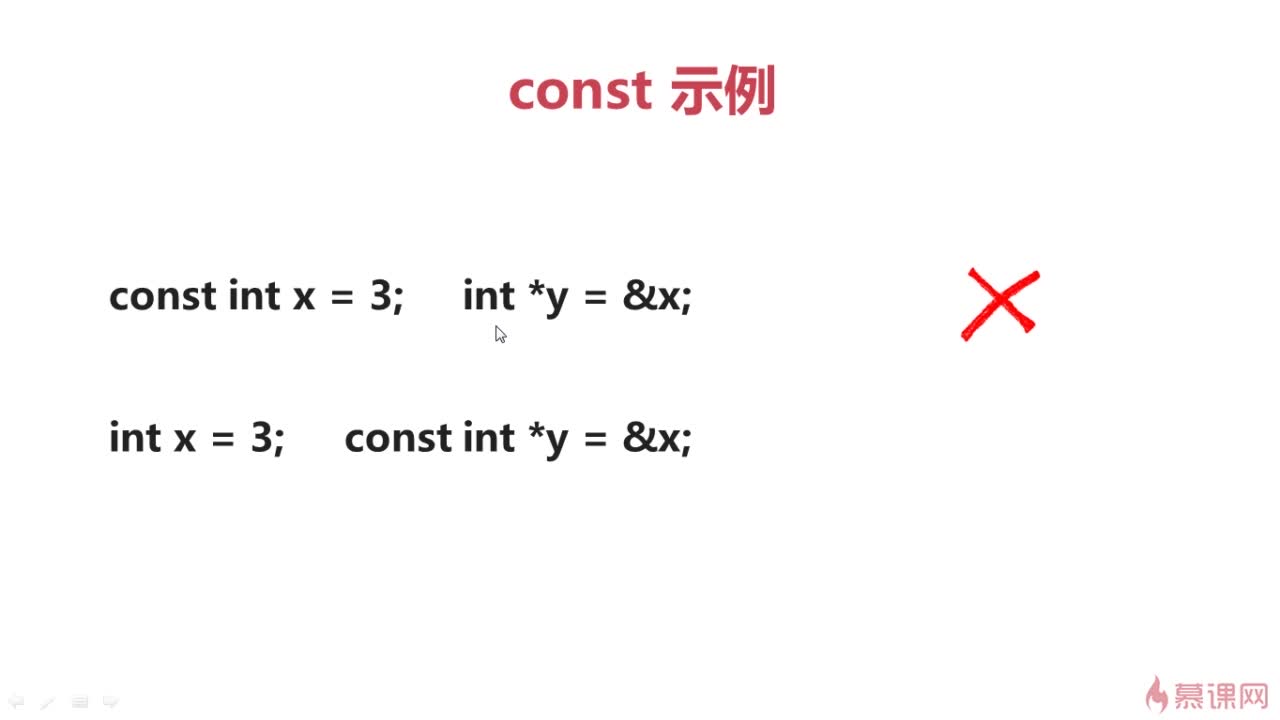
**但是，引用和指针是有区别的，区别如下：**

* **使用引用类型就不必在fun函数中声明形参是指针变量。指针变量要另外开辟内存单元，其内容是地址。而引用变量不是一个独立的变量，不单独占内存单元，在上例引用做函数参数的fun函数中引用变量a和b的值的数据类型与实参相同，都是整型。**
* **在main函数中调用fun函数时，实参不必用变量的地址(在变量名的前面加&)，而直接用变量名。系统向形参传送的是实参的地址而不是实参的值。**
* **这种传递方式相当于Pascal语言中的“变量形参”，显然，这种用法比使用指针变量简单､直观､方便。使用变量的引用，可以部分代替指针的操作。有些过去只能用指针来处理的问题，现在可以用引用来代替，从而降低了程序设计的难度。**

# （2）：const:



**注意这种写法：**



**上例中第一种写法：**

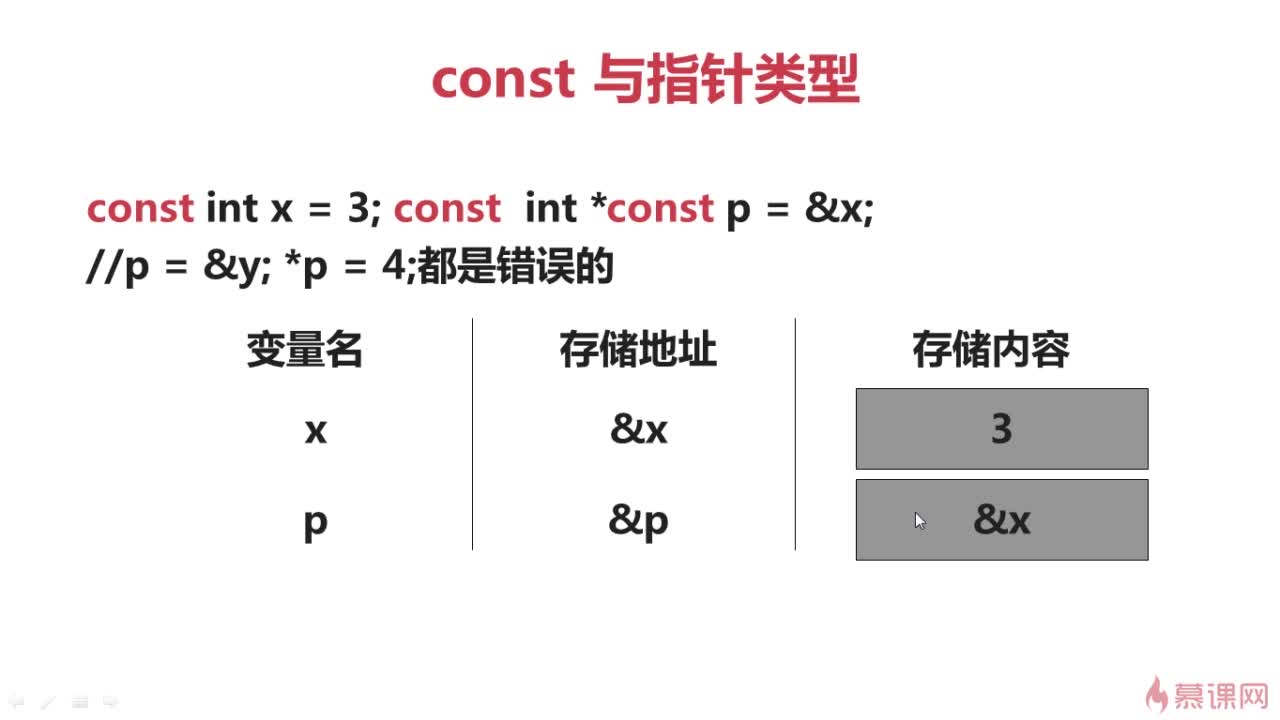
**const int x = 3;**

**int \*y = &x;**

**表示x是不可变的（只读，权限小） \*y是可变的（可读可写，权限大）**

**权限小的在前权限大的在后，有风险（不合理但是不违法，建议不要这么用）；**

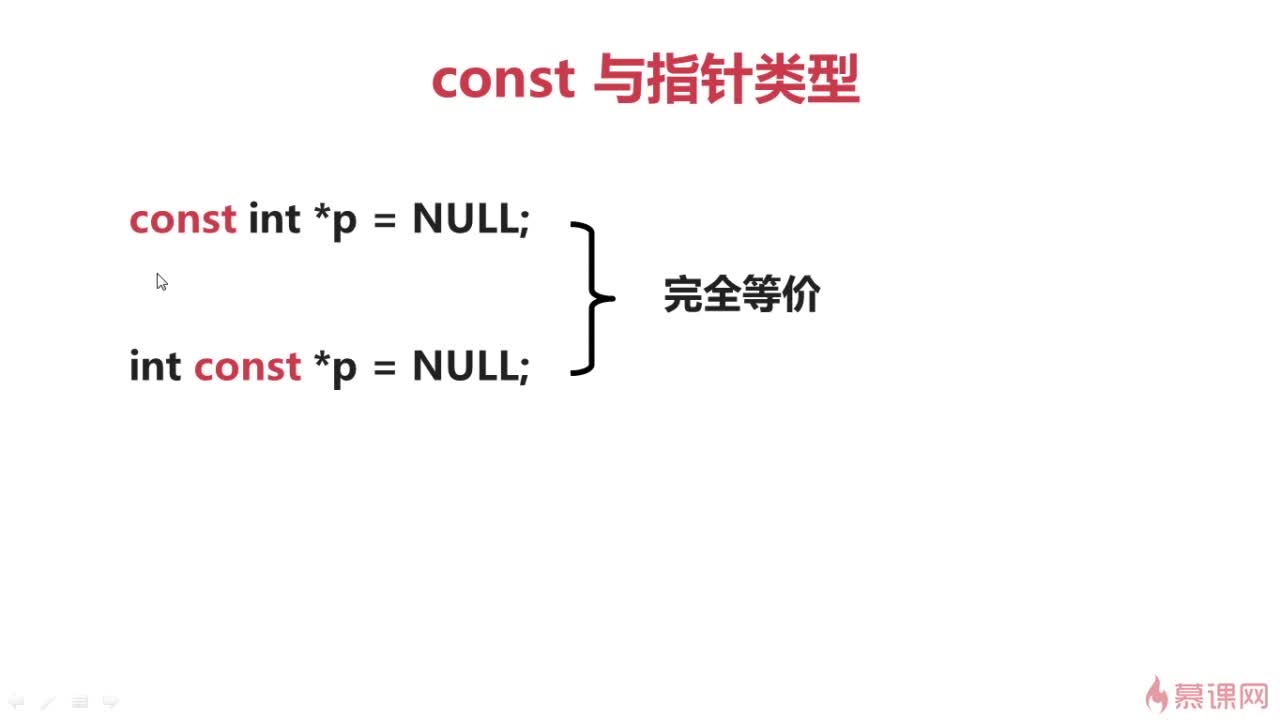
**第二种写法合理合法。**

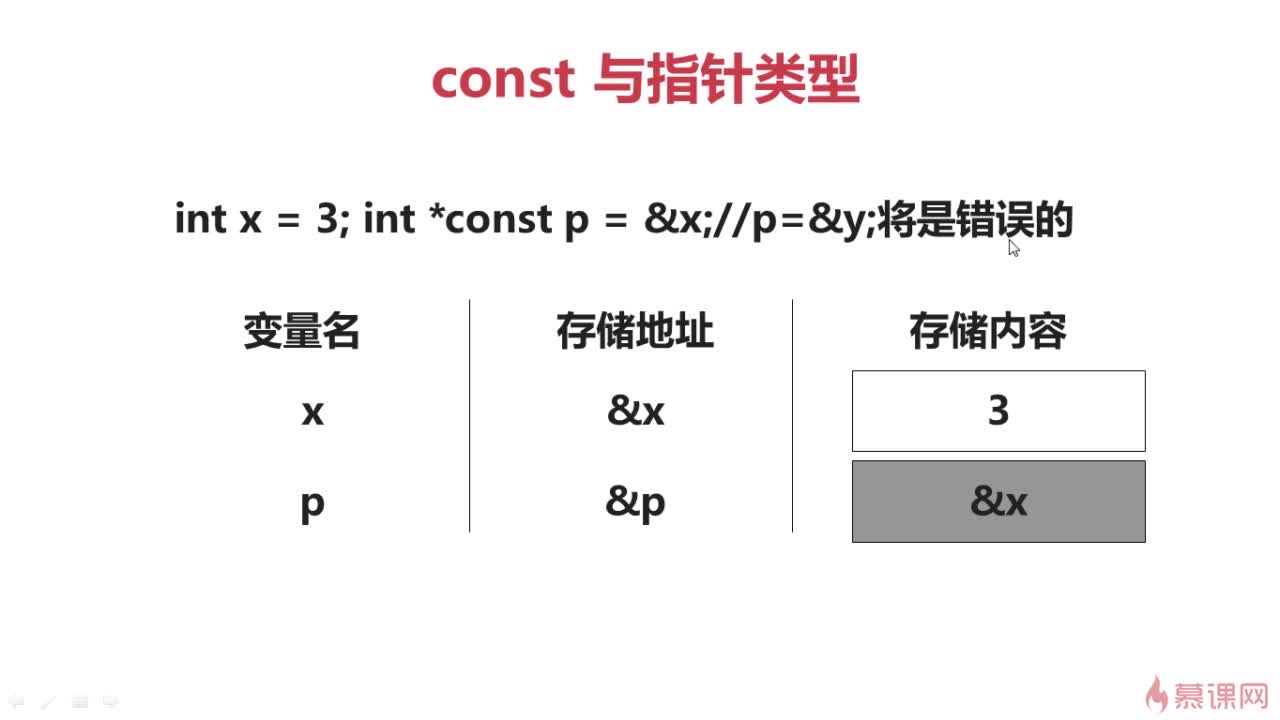


**int \*const p = &x;**

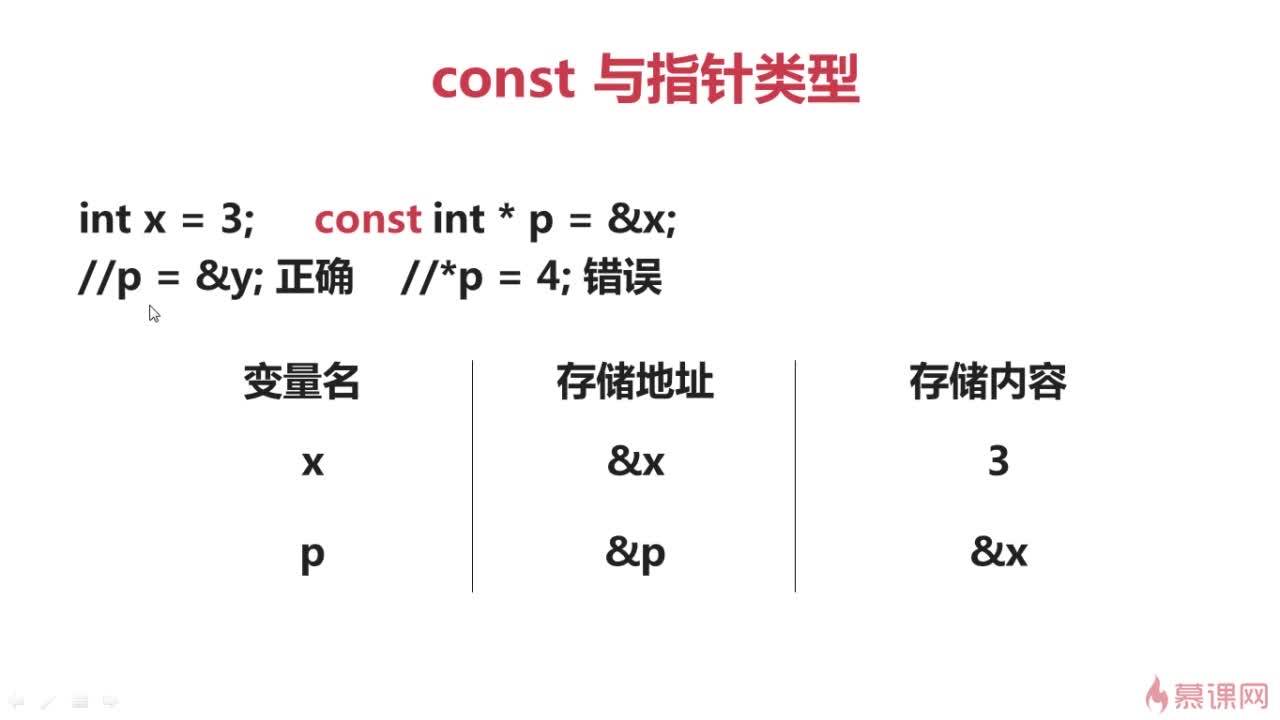
**此时const修饰的是指针，所指的是p变量的地址,因此再次指向一个地址是错误的：**

**以下两种用法是等价的：**





**Const int \*p:**



**此时const int\*p const指向的是\*p， const所指的不是p指向的地址，而是p指针指向的变量,所以可以使用p（常量）指向另外一个地址 p = &y，但是不可以令p已经指向的地址为某一数值 \*p = 3**

**小贴士：**

**说一个比较好记的方法来区分 int const \*p与 int\* const p，把\*读作pointer to然后从后往前读.  
第一个int const \*p就可以读作 p is a pointer to const int，p是指向常量的指针  
第二个int\* const p就可以读作 p is a const pointer to int，p是指向int型的常指针**

**3：典型例子  
①int x=3; const int &y=x;//x=10;正确//y=20;错误，不能改变y的值  
②const int x=3;x=5;错误  
③int x=3; const int y=x;y=5;错误  
④int x=3;const int \*y=&x;\*y=5;错误，y=&b,正确  
⑤int x=3,z=4;int \*const y=&x;y=&z;错误，\*y=2，正确  
⑥const int x=3; const int &y=x;y=5;错误**

**3：例：下面选型中用法错误的是（）**

**A int const a = 3; int \*p = &a;**

**B int a = 3; int const \*p = &a;**

**C int a = 3; int \* const p = &a;**

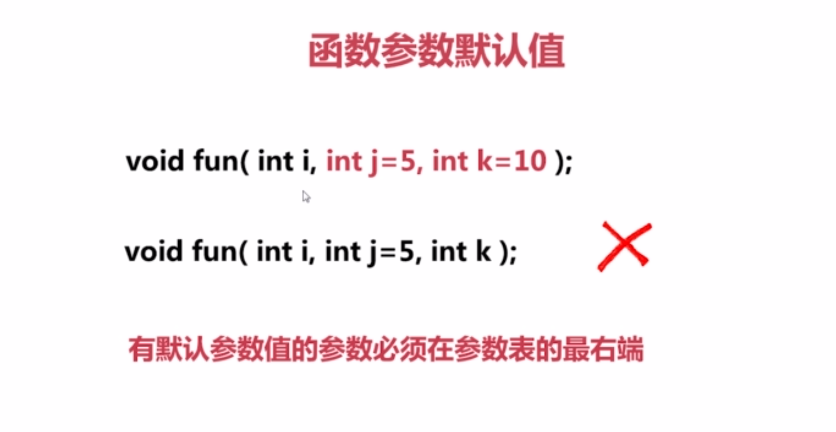
**D const int a = 3; int const &b = a;**

**解析：答案：A指针指向const修饰的变量时，应该是const int \*p=&a！！！！;**

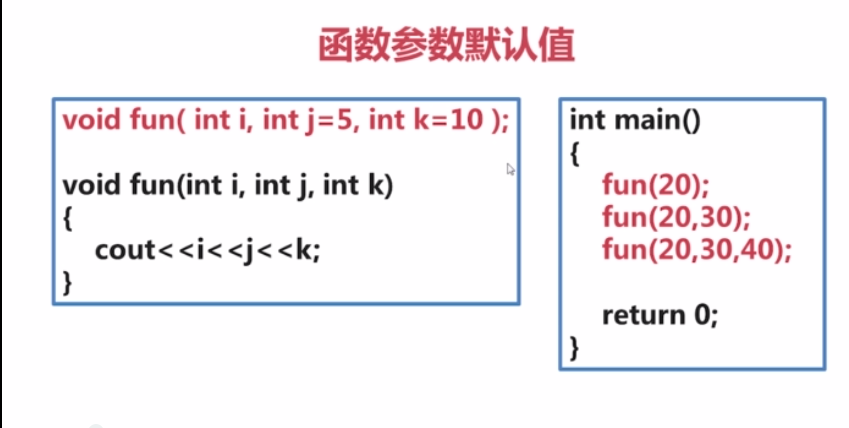
# 函数

## 给函数参数赋默认值

**有默认参数值得参数必须位于参数表的最右端**

****

**注意：若要给函数的参数赋默认值的时候，应该在函数声明中赋值，而不应该在函数定义时赋值，因为如果在函数定义时赋值，有的编译器可能会通不过。**

****

**在这种情况下即函数的形式参数可以与实际参数数量不一致，比如若用fun(20),则实际传入的参数应该是：20，5，10；即无实参的时候使用默认值，否则实参覆盖默认值，比如fun(20,30,40)就会覆盖原来的5，10.**

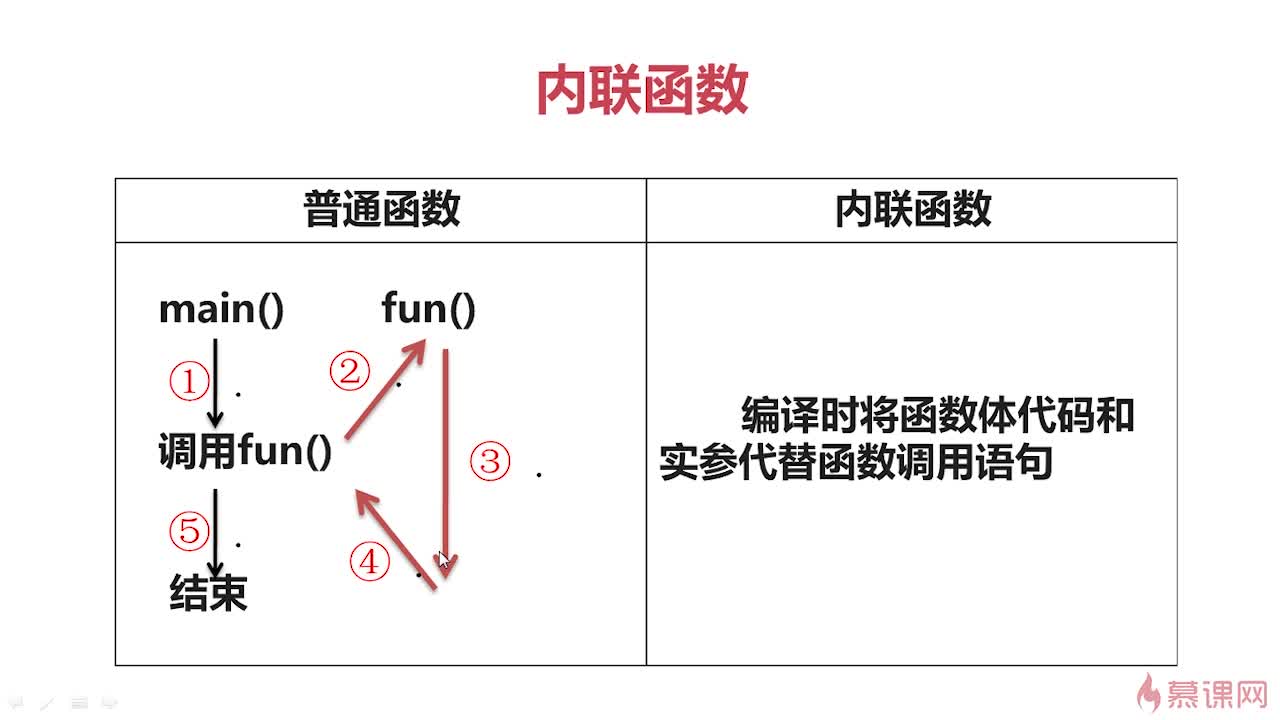
## 内联函数：

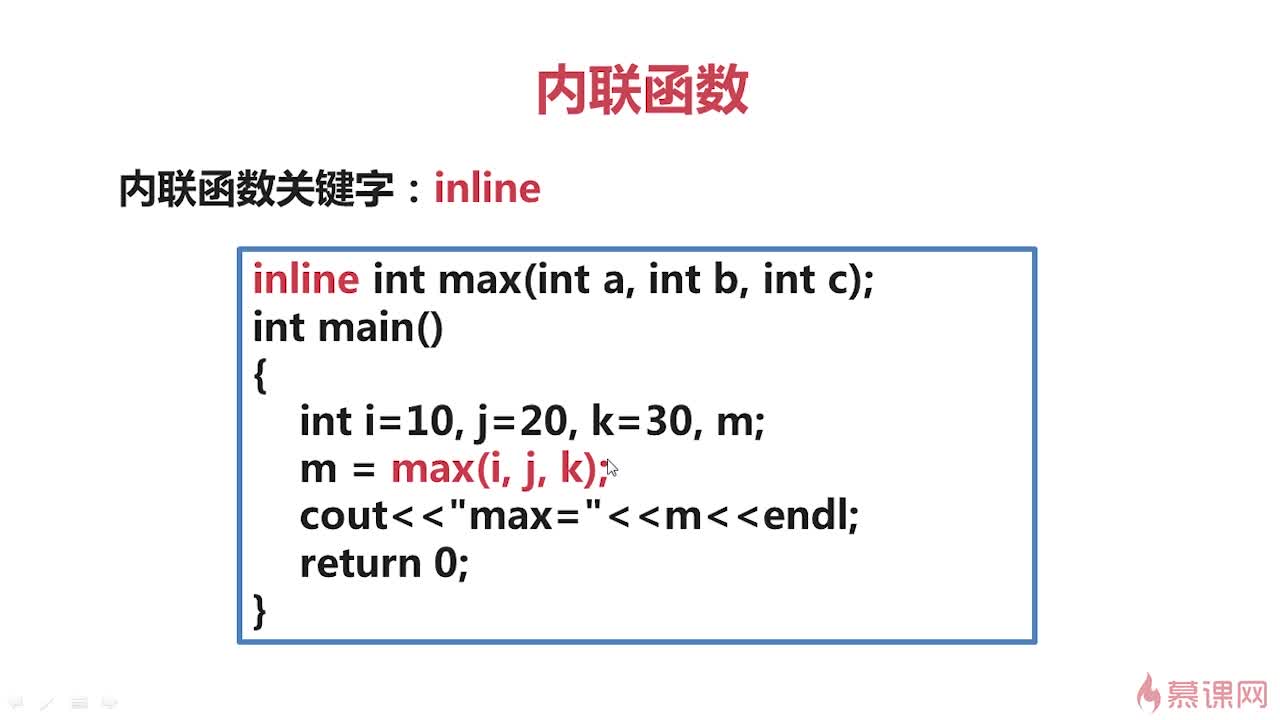
**使用内联函数后，相当于将内联函数中的代码加到调用函数内部。**

**内联函数使用关键字 inline**

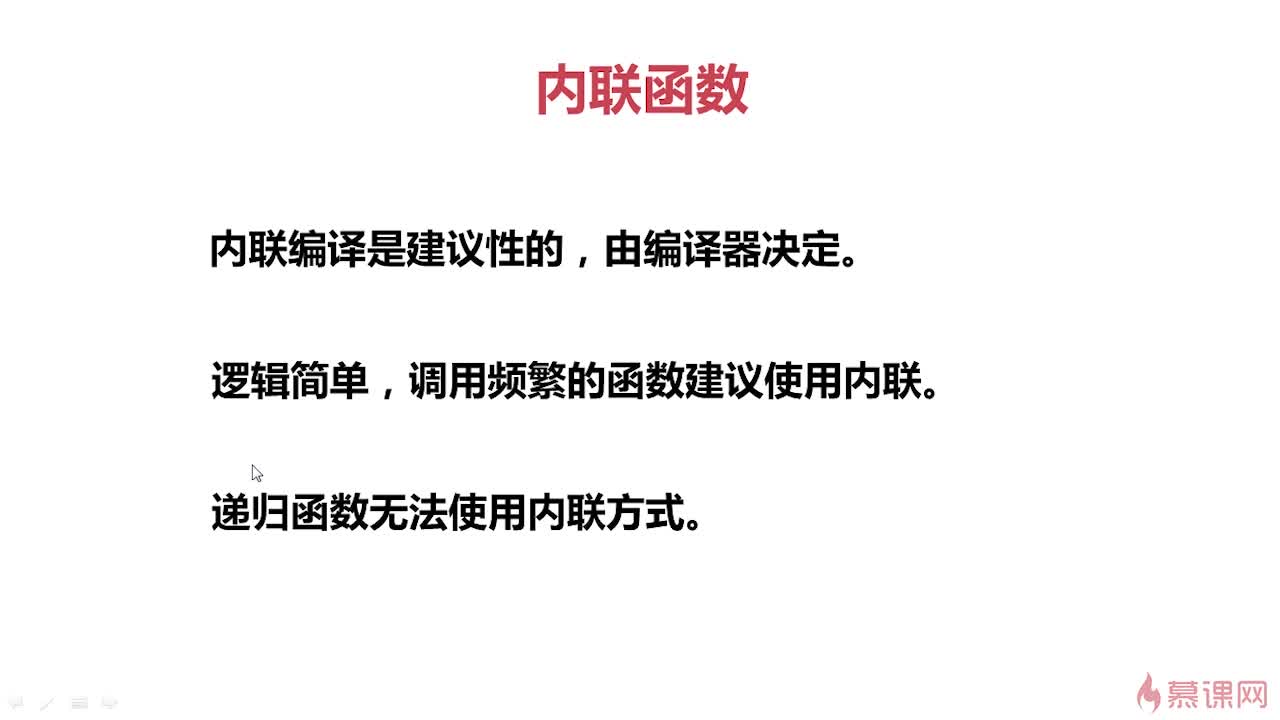
**使用内联函数和使用普通函数一样。**

**使用内联函数后，相当于将内联函数中的代码加到调用函数内部。**

****

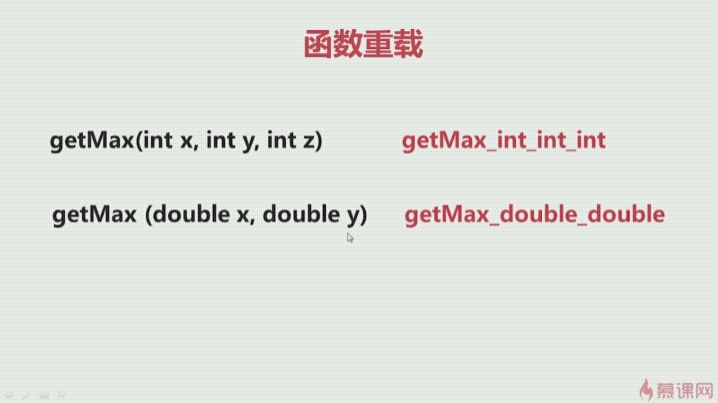
****

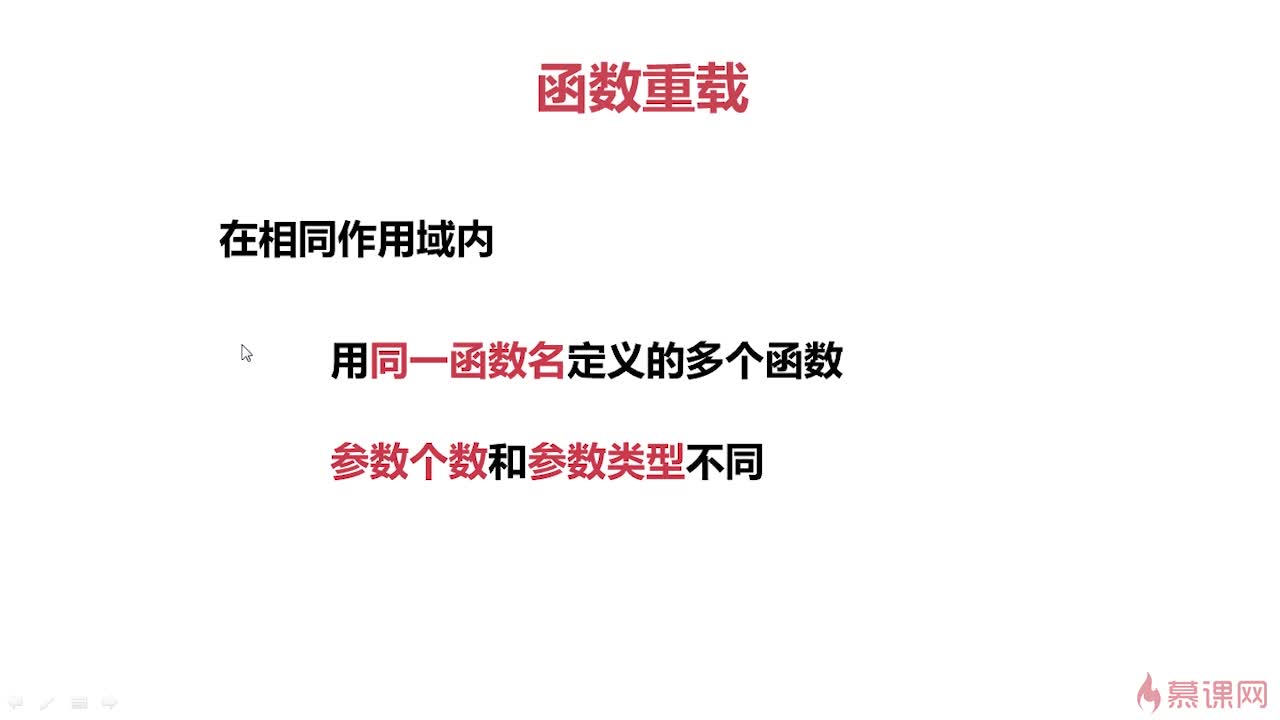
**既然内联函数这么实用，为什么不将所有的函数用作内联函数：**

****

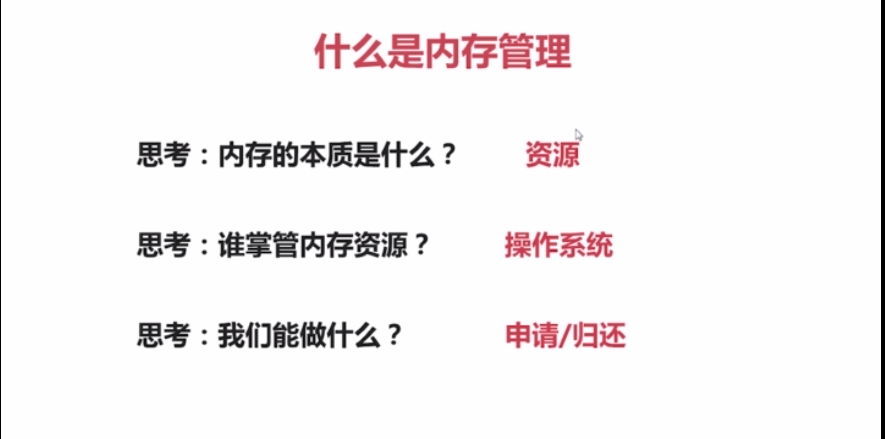
## 函数重载

**在定义同名函数后，系统会以 函数名＋参数类型和个数 来区别同名函数（C++的重载的两个函数参数数量可以相同也可以不同，当参数数量相同时，只需要对应参数类型不同即称为重载）**

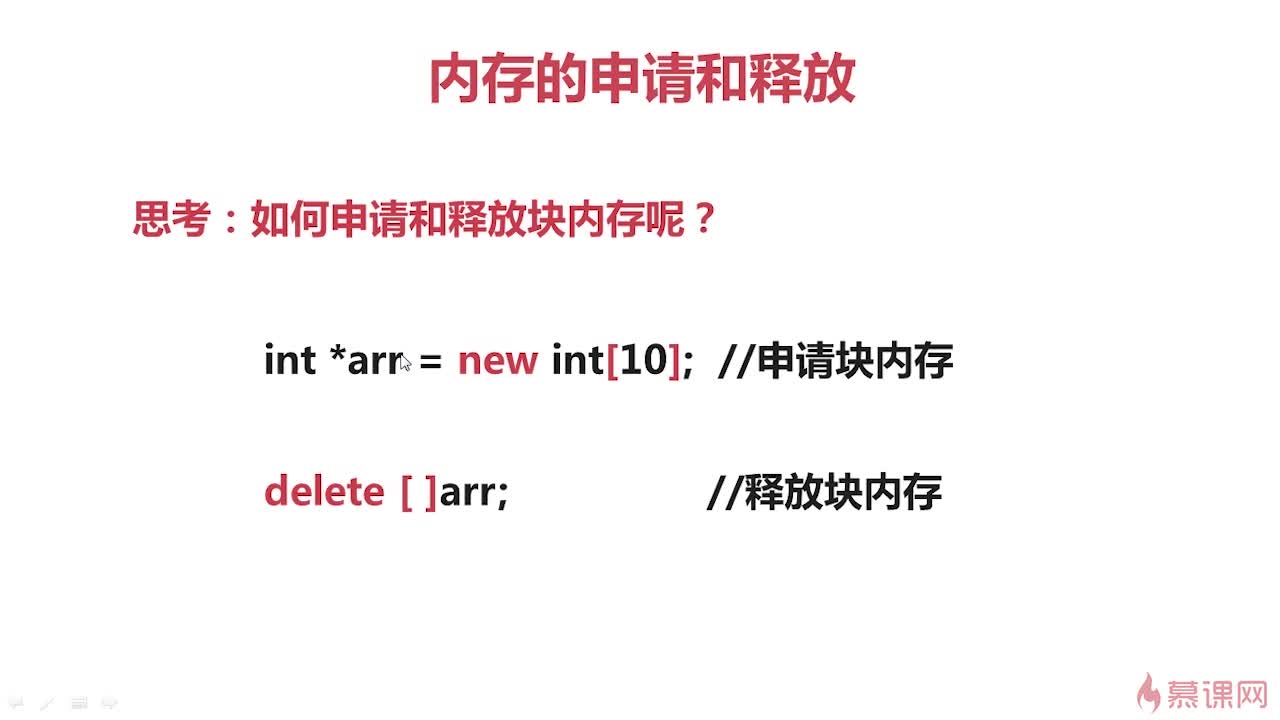
****

****

# 内存管理

****

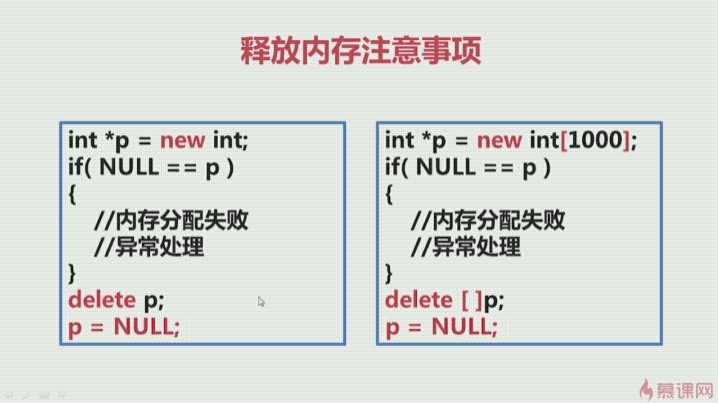
**申请/归还内存的过程就叫内存管理**

****

**注意：在释放了内存以后一定要让对应的指针指向空**

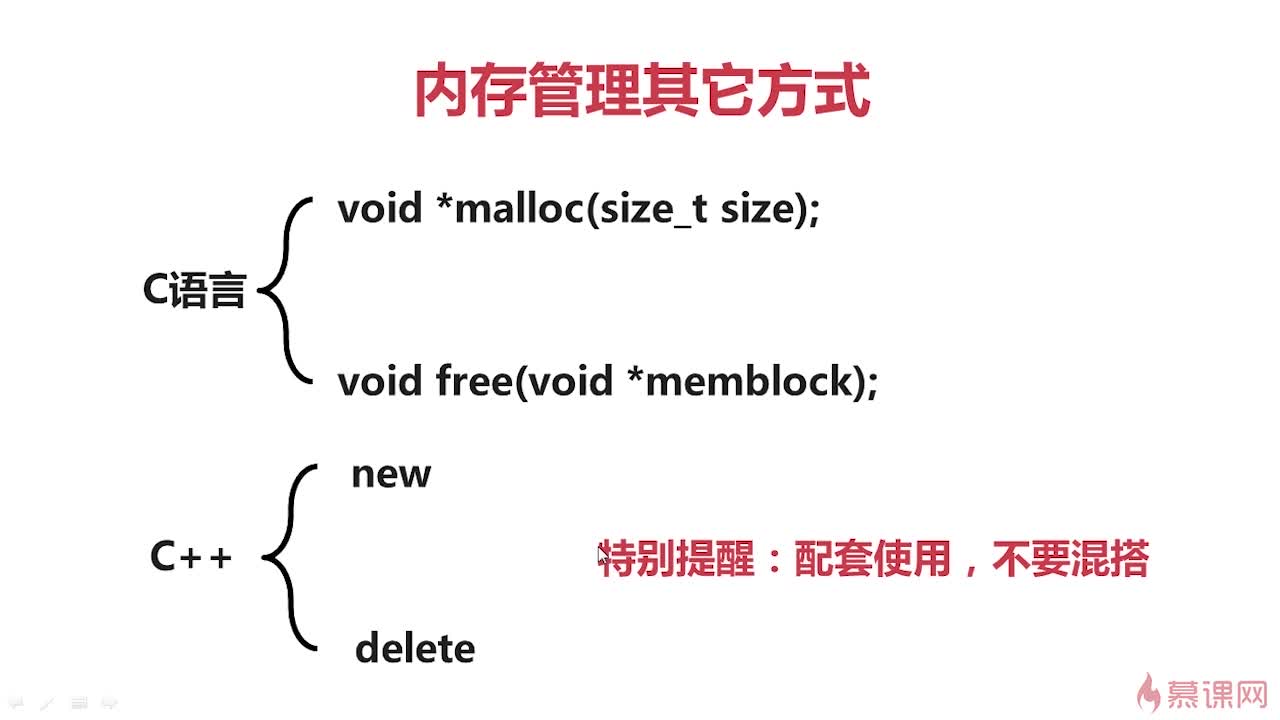
**在申请内存时可以进行初始化，如：**

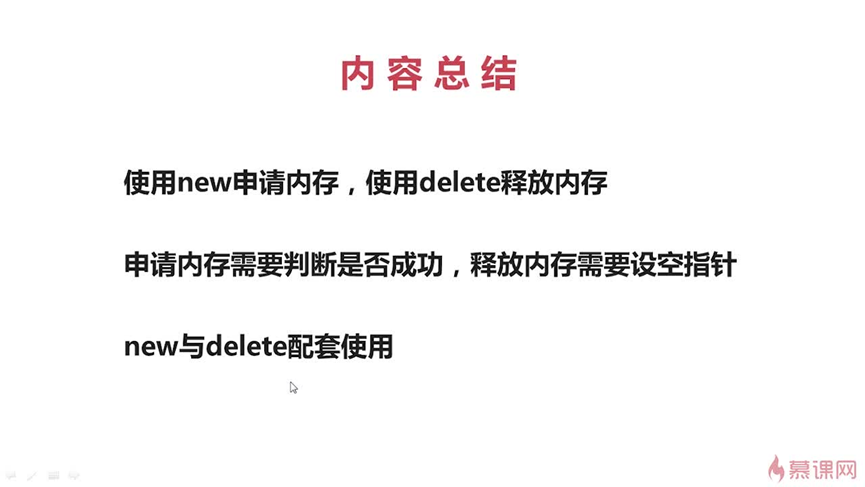
**int \*p=new int(20);//让p指向一个申请的int型的内存单元，并在这个内存单元内赋值20；**

****

**注意要配套使用内存申请和释放（注意：对c语言来说，maclloc和free是两个函数，而c++中的new和delete不是函数，而是运算符）**

**New和malloc的区别：malloc只是简单地进行内存的分配，而new不光会分配内存，还会自动调用该变量（对象）的构造函数。**

****

****

# 类和对象

**（类-选择性暴露-封装-访问限定符）**

## 概述

**对象：具体的事物；**

**类：从具体的事物中抽象出来的东西；**

**以狗为例：狗就是一个具体的事物，即对象，而狗的属性，比如叫、跑等就叫从狗身上抽象出来的东西，就叫类。**

**类的定义：数据成员 成员函数**

**在类中，也有很多我们不想暴露的属性，比如电视，看电视时我们只需要声音和图形就可以了，不需要暴露他的电路工作信息等，这时我们就需要把这些属性隐藏起来，这就是所谓的“封装”。**

## 封装

**定义：将实现细节隐藏起来，只提供用户关心的部分**

**定义类要用关键字class：**

**class 类名**

**{**

**//访问限定符**

**//数据成员**

**//成员函数(方法)，直接在成员函数的后面写他的具体代码**

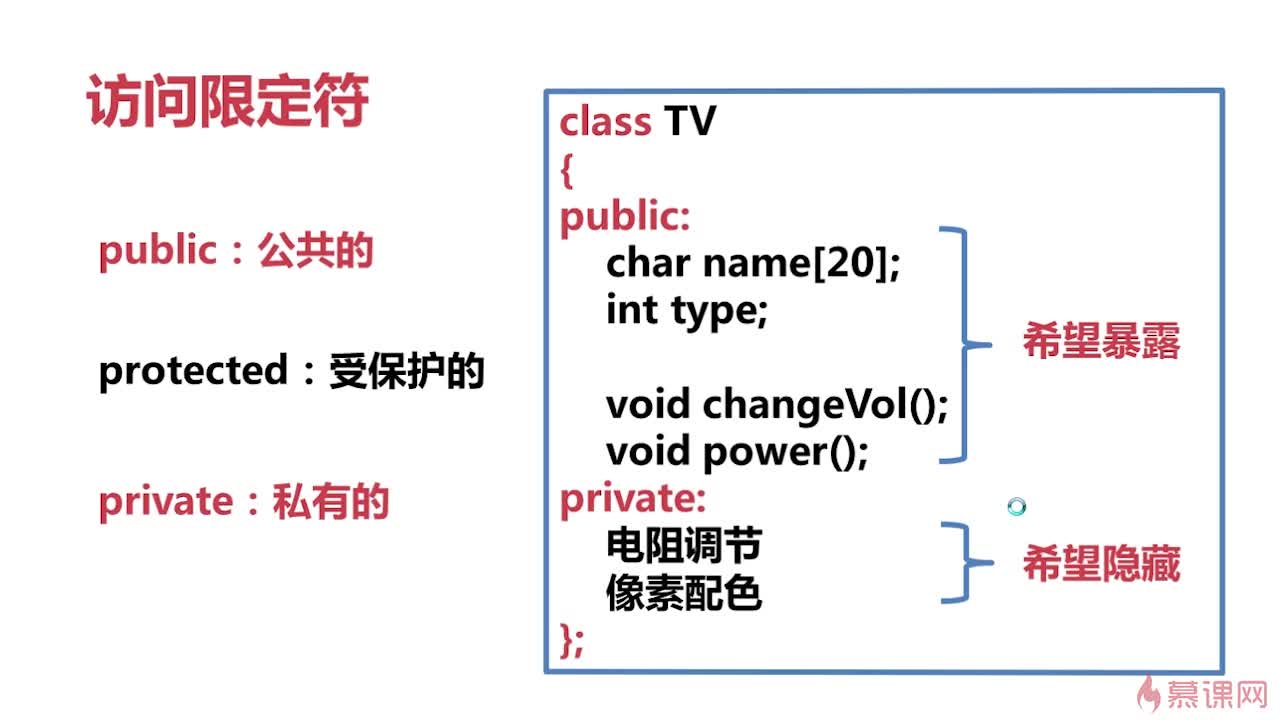
**}；**

**在类中用访问限定符来决定是否暴露，通过**

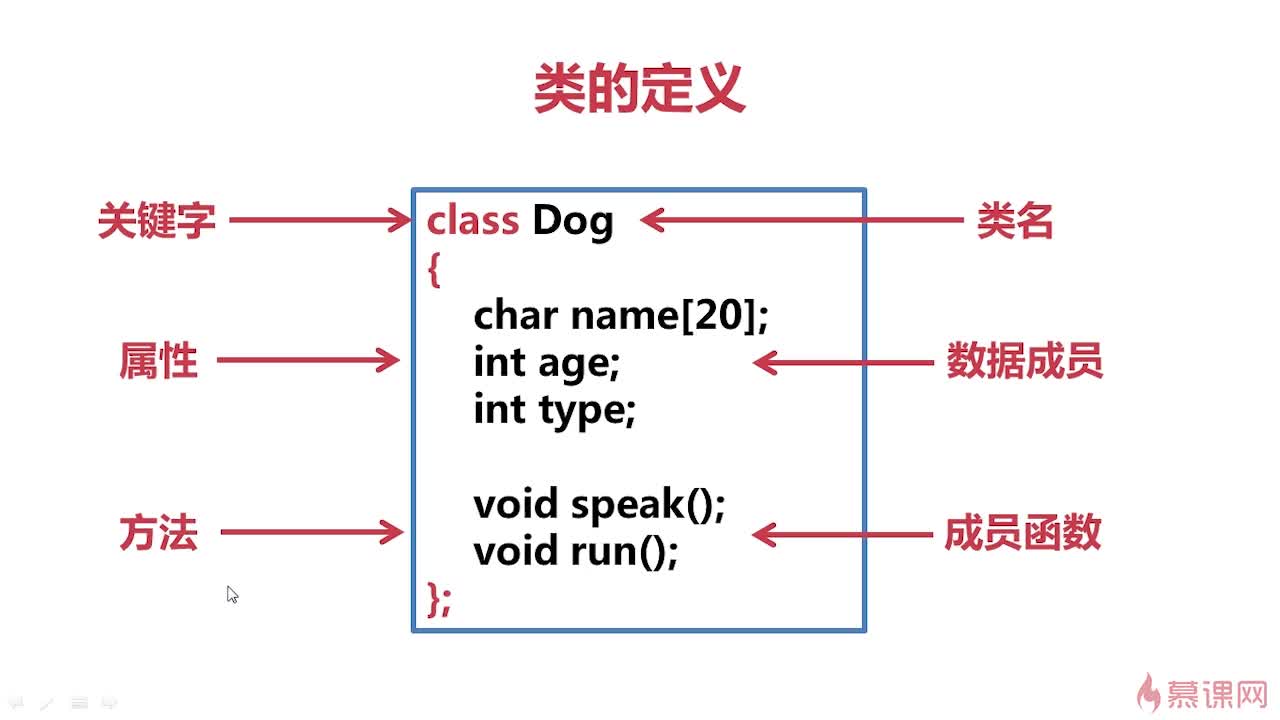
**public：公共的**

**protected:受保护的**

**private：私有的**

****

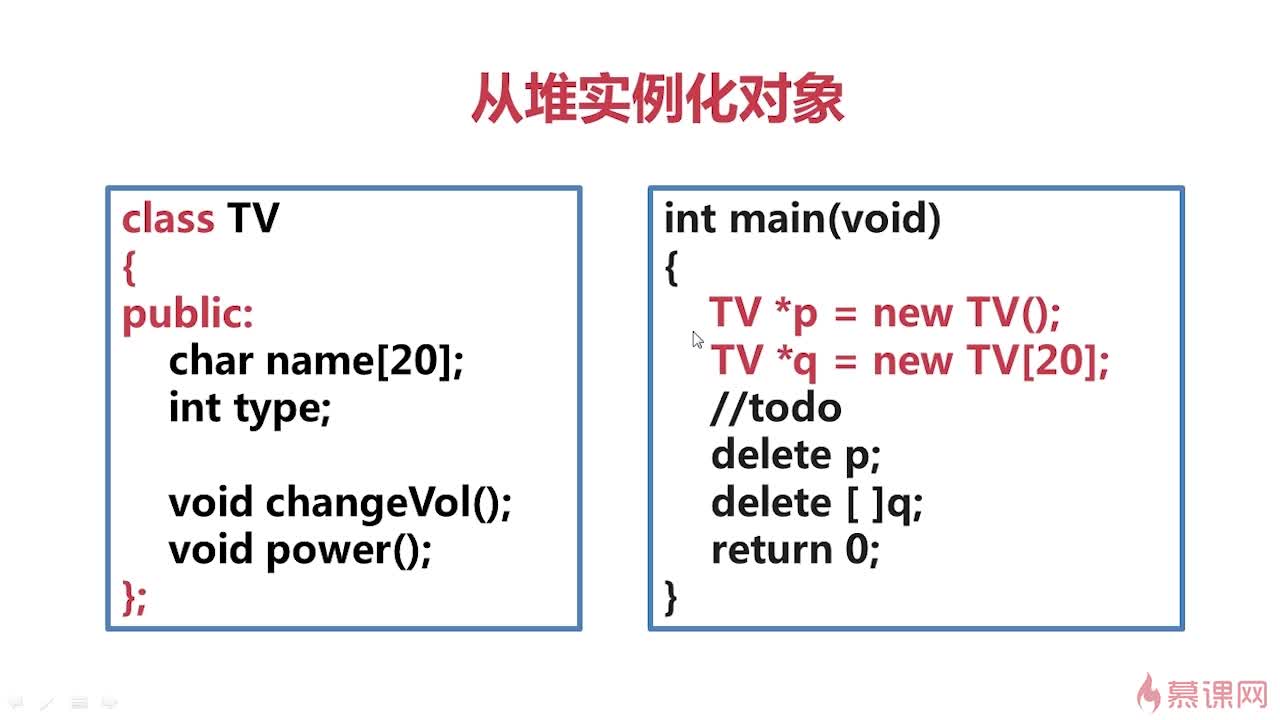
**隐藏的部分就体现了“封装”**

****

## 实例化

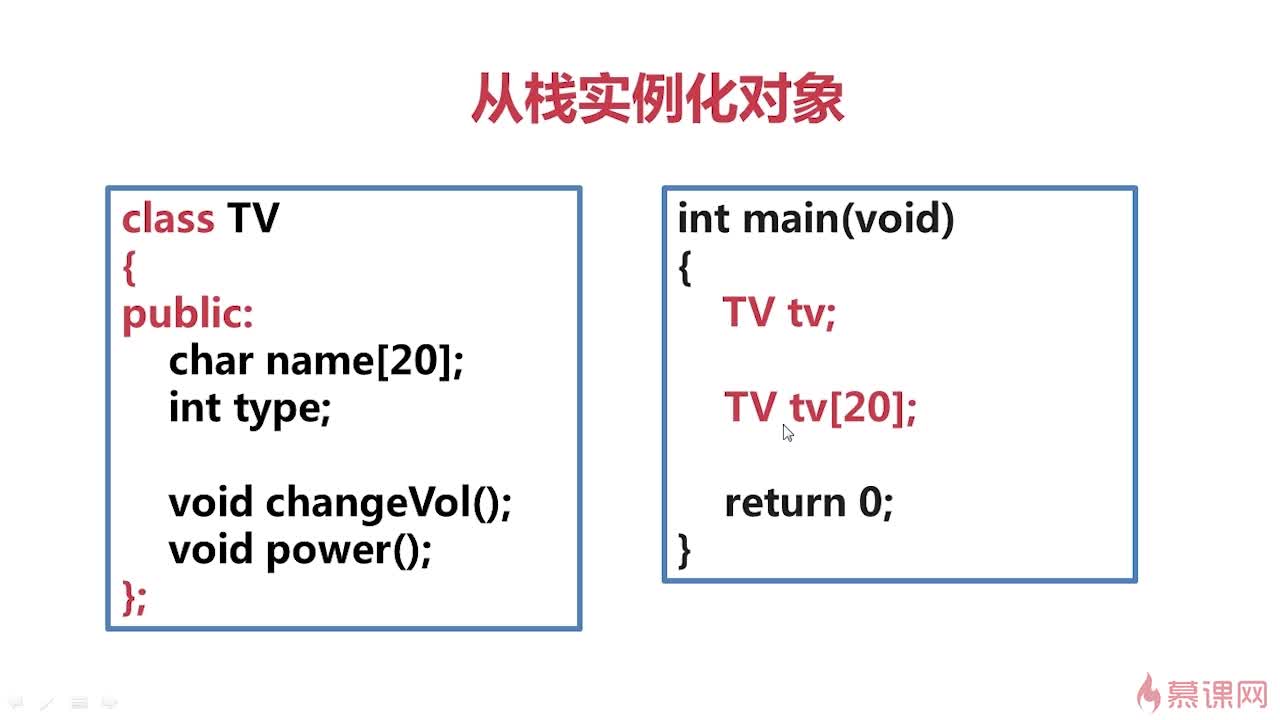
**类实例化的两种方式：一种是从栈实例化，一种是从堆实例化**

### 从堆实例化对象

****

**注意：从堆实例化对象，一定要释放内存，并且最后将指针指向空（NULL）；**

### 从栈实例化对象

****

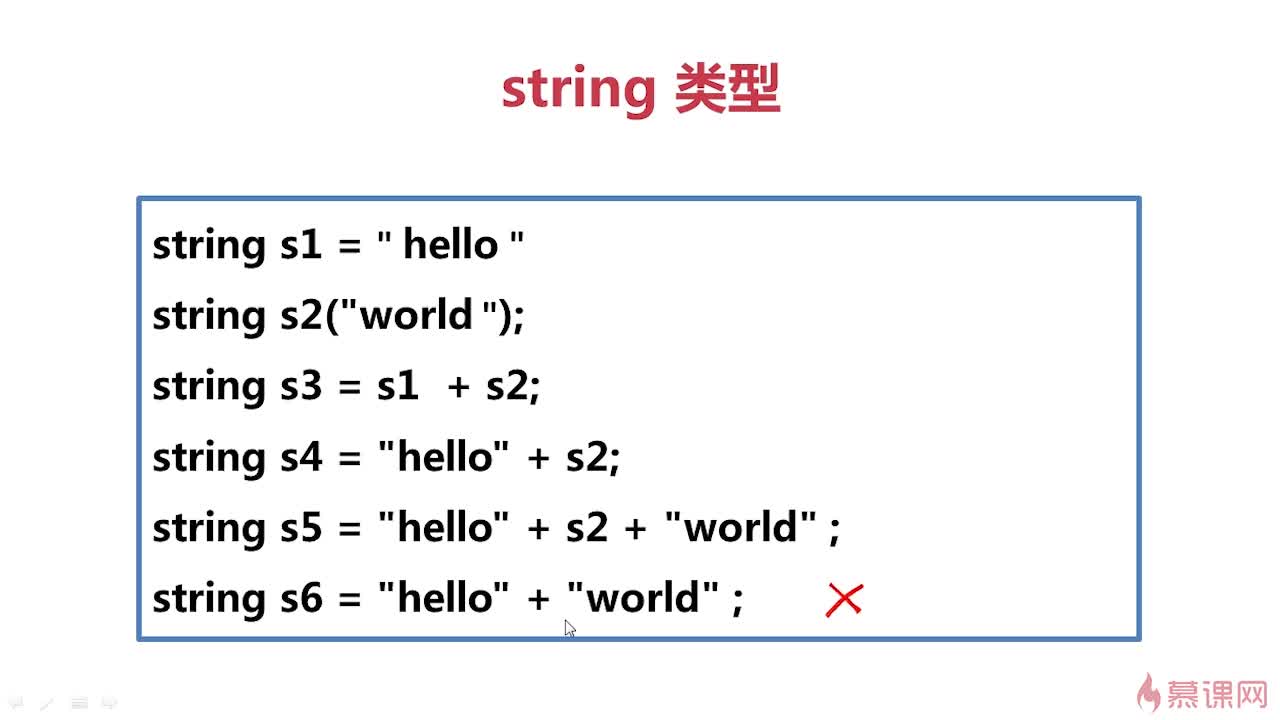
# c++中字符串的使用

**首先，c++中要对字符串进行操作，必须包含头文件#include<string>**

**字符串的相关操作:**

****

**连接+用法的注意：string变量和双引号引起来的字符串连接的才允许，两个双引号的不允许**

****

**String的初始化方法：**

****

**getline(cin,name)：cin读取一个单词，以空格、tab、回车作为判定分隔符；**

**getline读取一行，以回车作为分隔符。**

**具体的的含义：若用户直接输入回车，则给name为空，否则给name用户输入的字符串**

# 数据的封装

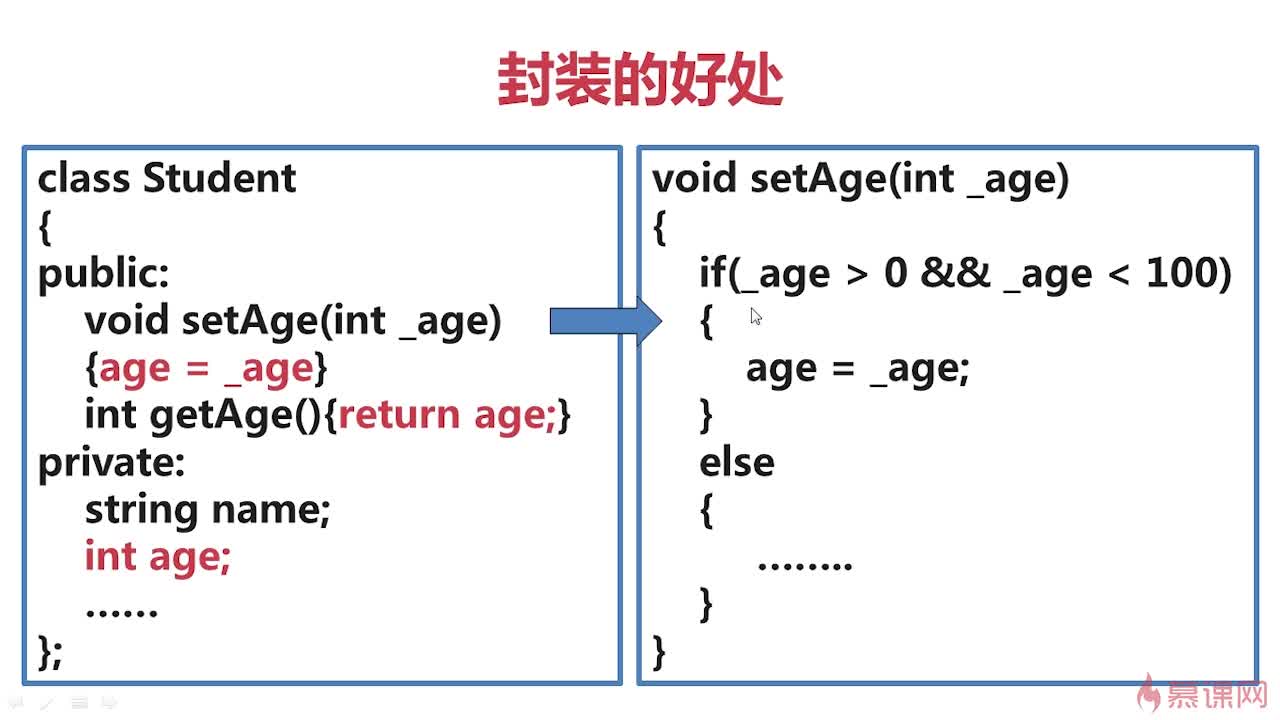
**在main函数中看不到具体的变量，而是通过直接为函数赋参数，来改变。也就是说，那些数据真实存在，但是只能通过调用函数来改变他们的值；**

**具体的就是在类的private部分定义成员，在类的public部分定义成员的读/写函数（按照需要进行读/写函数的定义），当想对类进行操作（即读写的时候）通过调用public里面的函数来实现。**

**面向对象的基本思想：以谁做什么来表达程序的逻辑，以对象为中心，对象的所有行为都通过调用自己的函数来完成。体现在代码层面就是数据成员调用自己的函数以及通过函数来对自己进行操作.**

**封装的好处：**

1. **可以在成员函数中做一些必要的限制**

****

**有时候确实不允许用户改变数据成员（只读性，只有get函数，无set函数）**

****

**对数据的封装一般用set来表示数赋值 get函数用来表示数据成员（只是一种建议的书写代码的习惯，这样可以使代码更加清晰可读）**

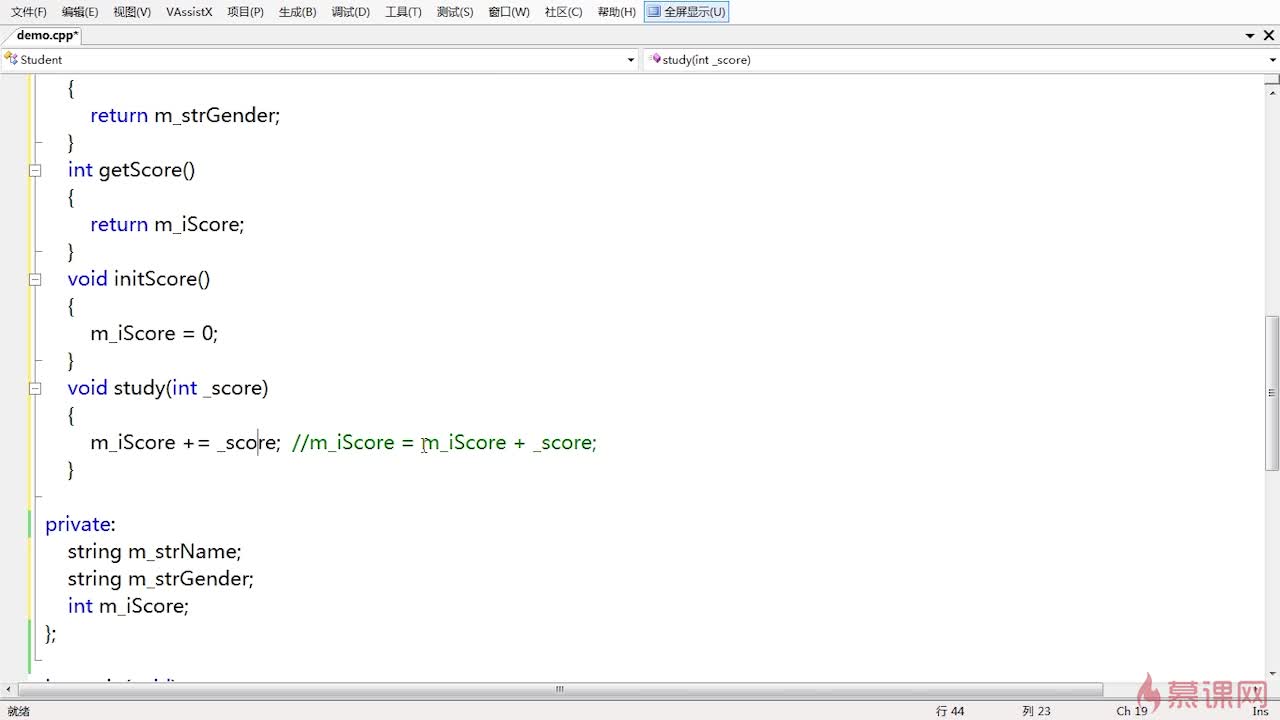
**建议：**

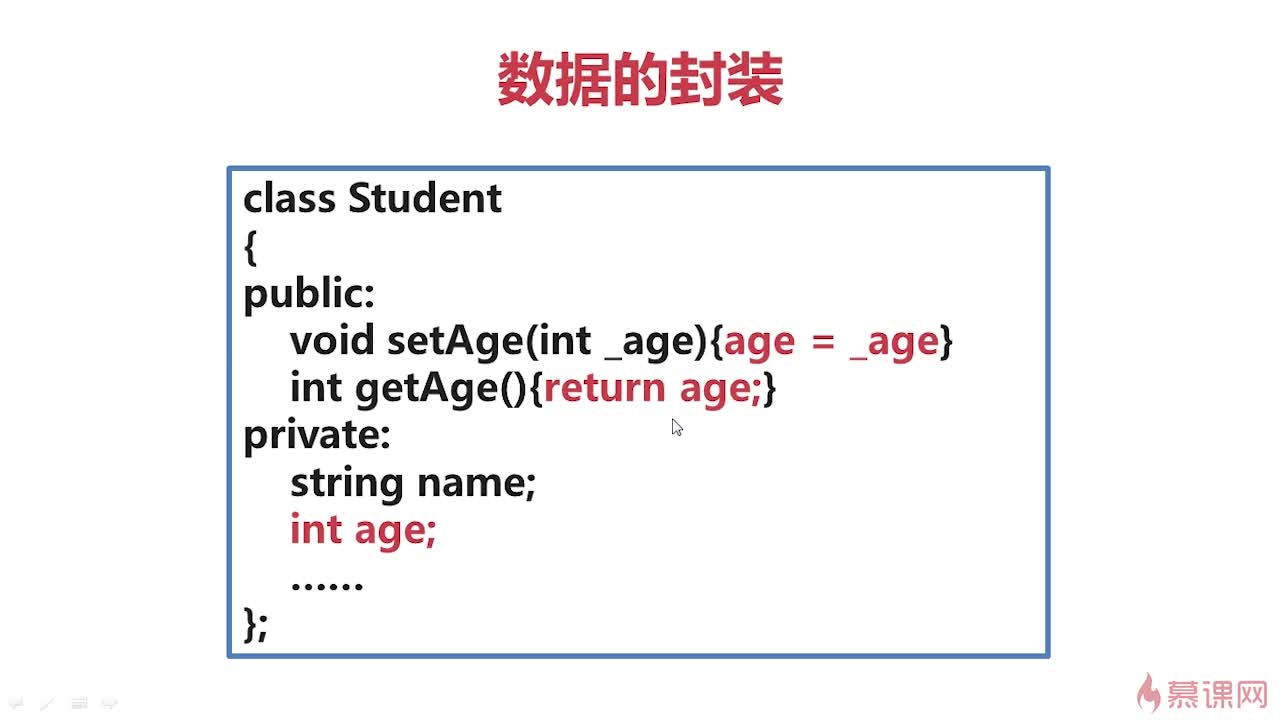
**1.类的写法一般是public写在前面，private写在后面**

**2.要求所有的数据成员时命名为 m\_数据类型+成员名（字符型写str,整形写i），在写相关函数进行封装的时候，把参数写成数据类型+\_+参数名，这样做也是为了程序更加清晰。**

**如：string m\_strName 以便区分数据成员，知道数据成员是定义在类里面的**

**形参是\_+成员名，如：void getName(string \_name)**

****

****

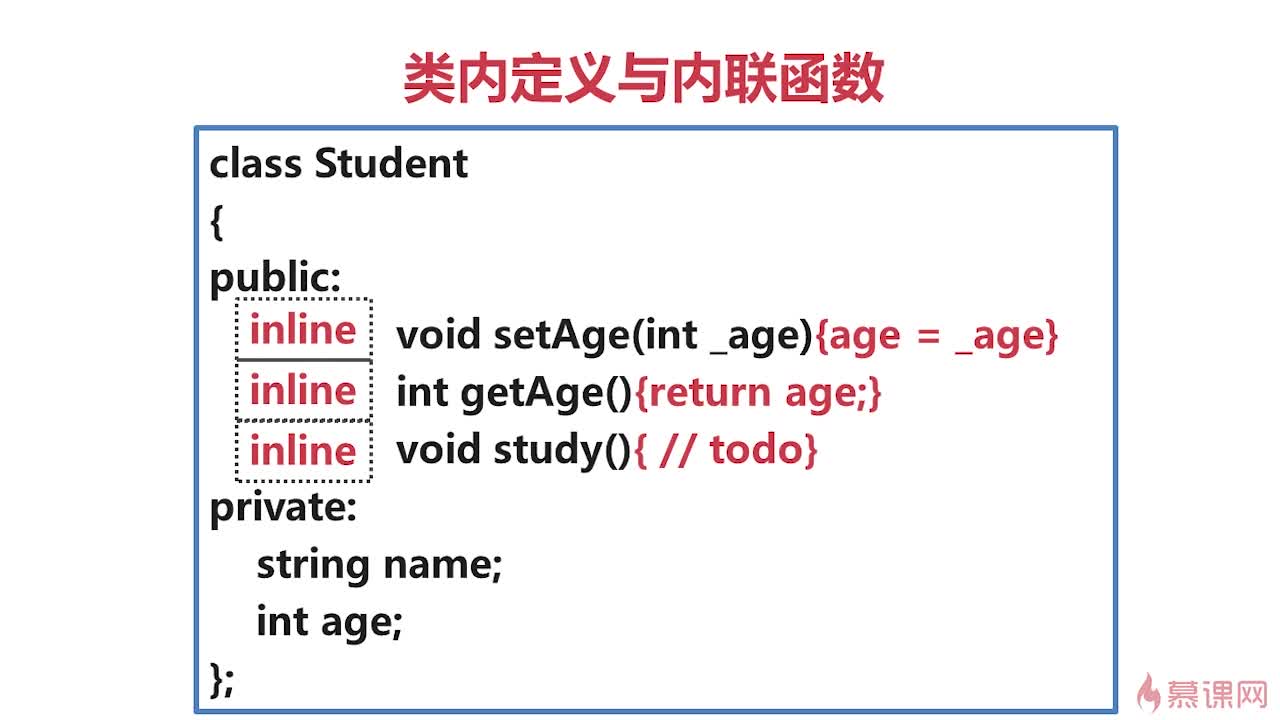
**将想要封装的数据写入private中，在public编写调用函数来改变封装的数据。**

# 函数的定义

**内联函数：关键字inline，相当于该函数在编译时原封不动地放在调用函数的地方进行编译；**

**类内定义：在类内定义的函数且函数体也在类内；**

**内联函数与类内定义的联系：类内定义默认是内联函数（当类内定义的函数比较简单的时候，复杂的函数编译器会拒绝将其转换成内联函数即会正常编译）**

****

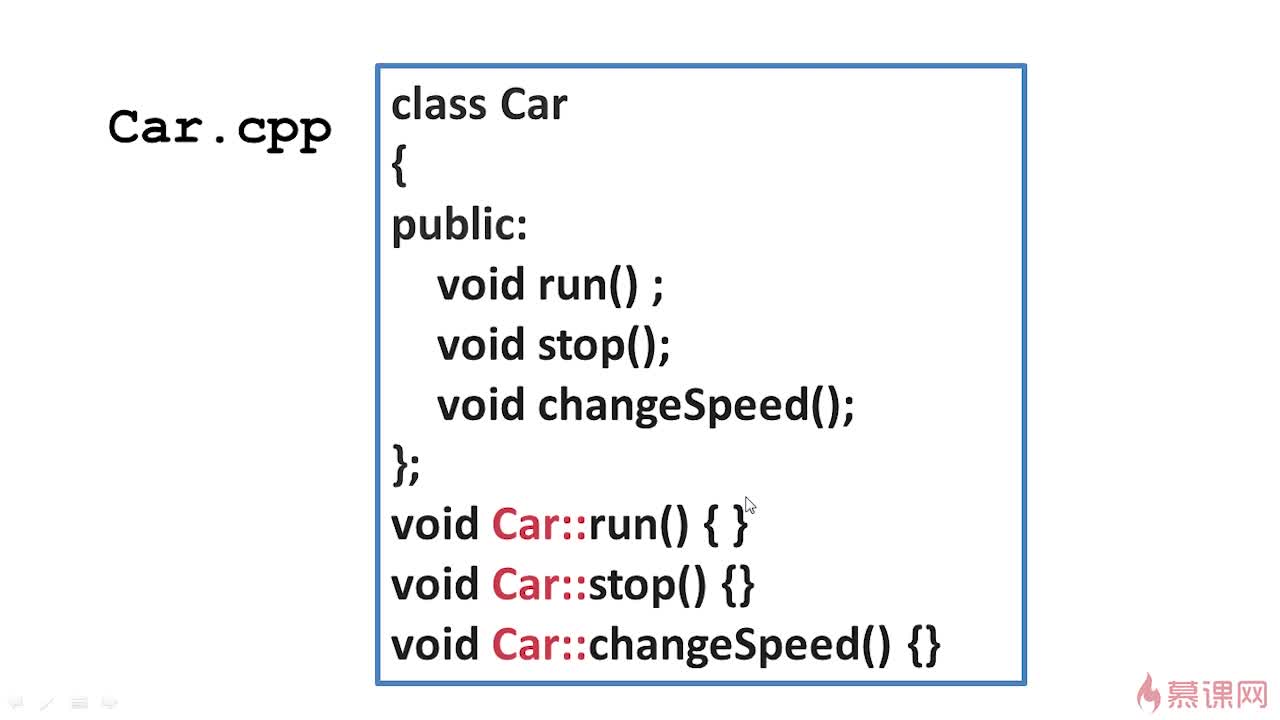
**有类内定义就有类外定义：**

**类内定义： 函数体在类内**

**类外定义：函数体在类外**

**同文件类外定义： 函数体在类外，在同文件内**

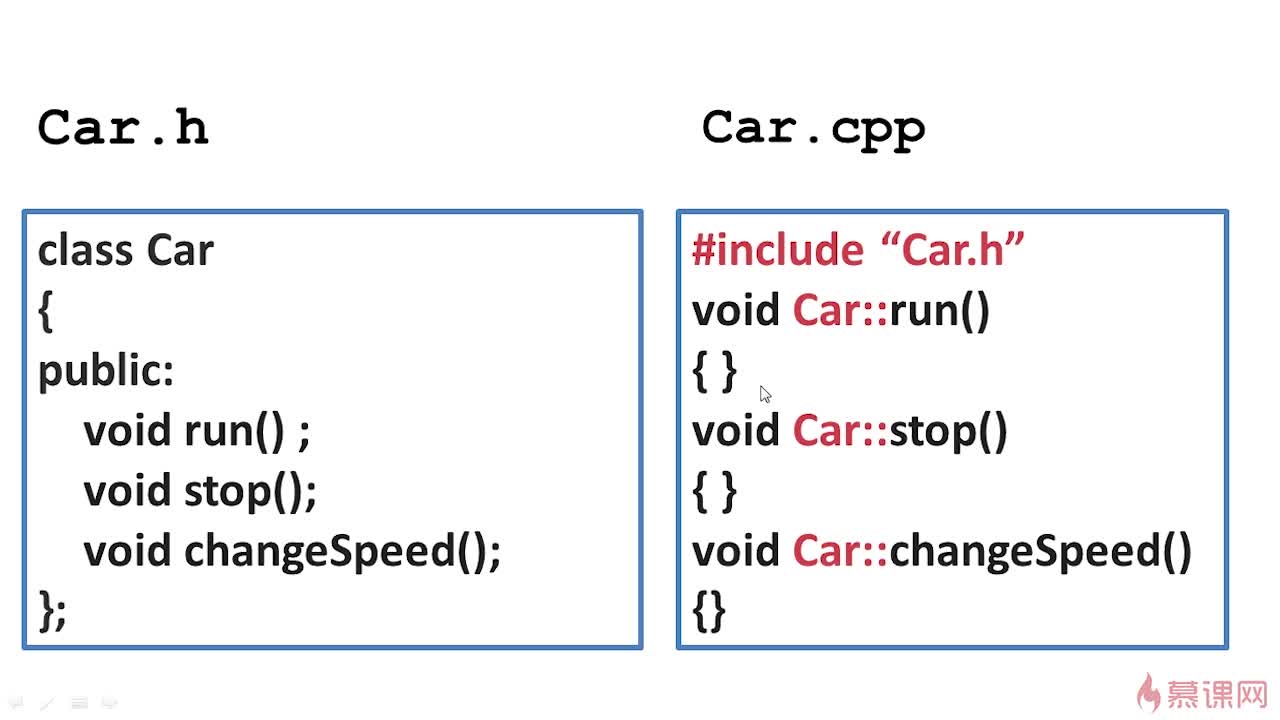
**同文件类外定义实例：**

****

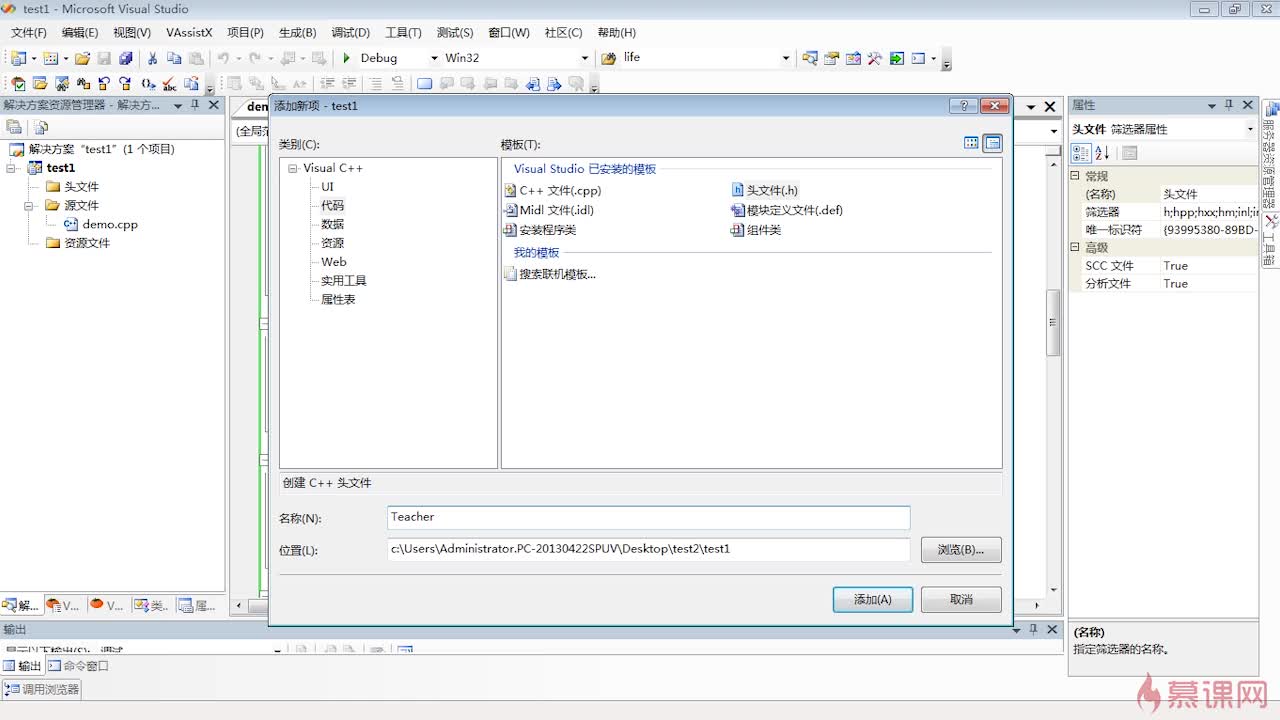
**分文件类外定义： 函数体在不同文件（最常用的是分文件类外定义）**

**分文件类外定义的意思就是：在.h文件中先声明类；然后在.cpp中定义类，且此文件中必须包含.h文件；以上两个文件中也应包含所需要的文件名，如iostream等等。定义完成后，就可以在主函数所在的文件中调用类的成员及其函数了,在该文件中也必须包含.h文件。**

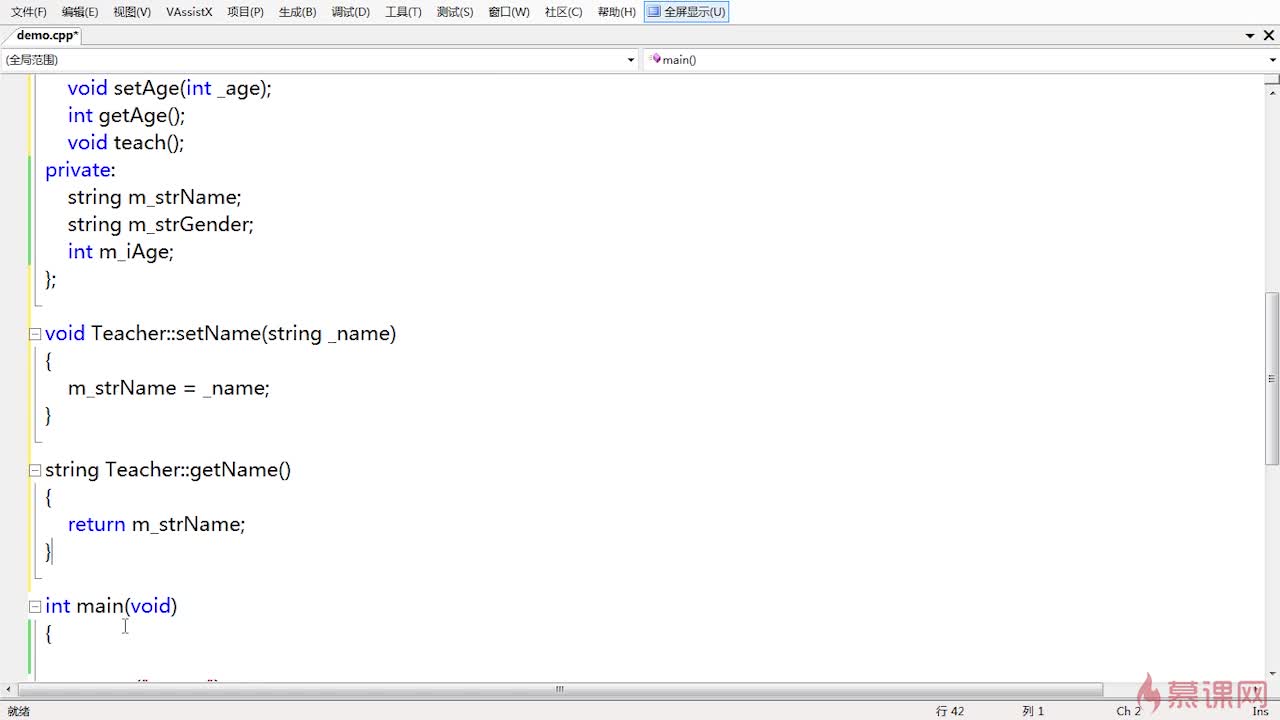
**分文件定义实例：**

****

**分文件定义的时候，记得点击头文件，选择头文件选项，然后起名字记得和命名的类名一致：**

****

**声明成员函数的时候加上分号，定义成员函数的时候不需要添加分号：**

****

# 函数

## 1：构造函数：

**使用构造函数的原因：防止忘记对类中的元素进行初始化，防止在只需要一个初始化的时候却进行了多次初始化。**

**（构造函数没有任何返回值，甚至不用写void，直接写函数名就可以）；**

**在实例化对象的时候会自动调用构造函数**

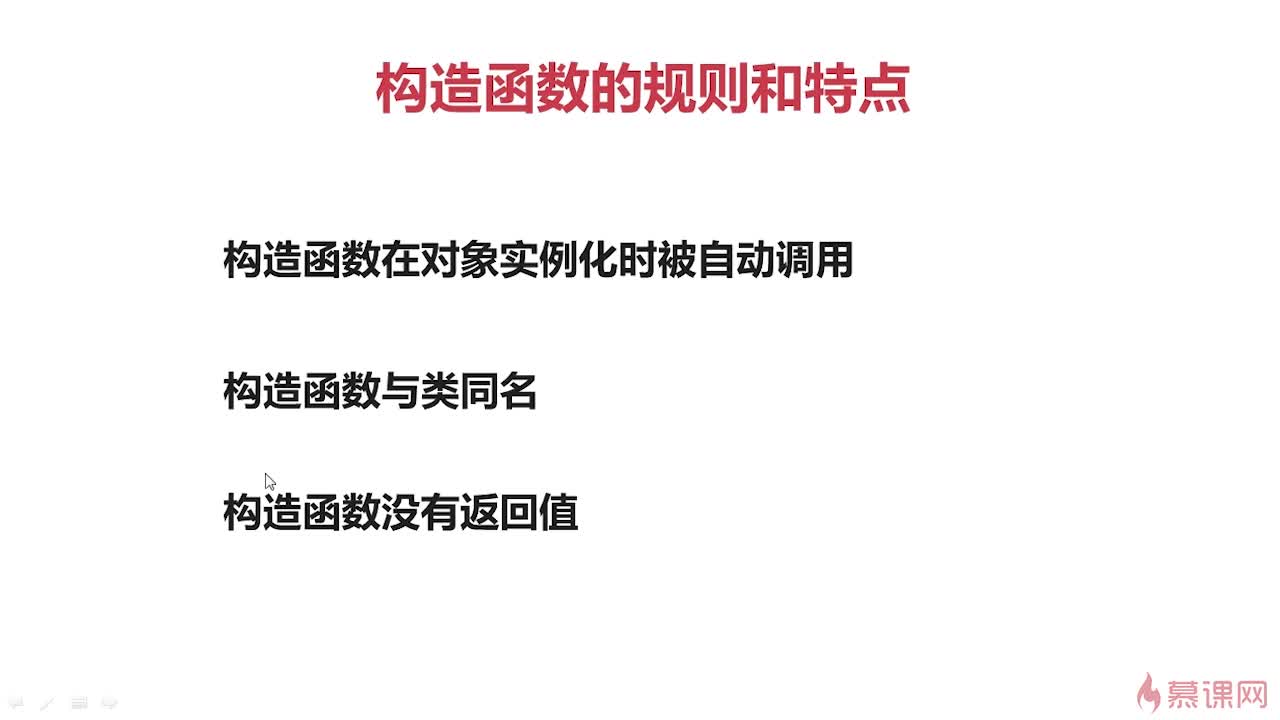
**在实例化对象的时候不需要传递参数的构造函数称之为默认构造函数（可以是无参构造函数，也可以是有参构造函数但是参数全部被赋了默认值的构造函数注意这是赋默认值应该在类里的函数声明里面赋默认值）；**

**构造函数是对象实例化时被自动调用。**

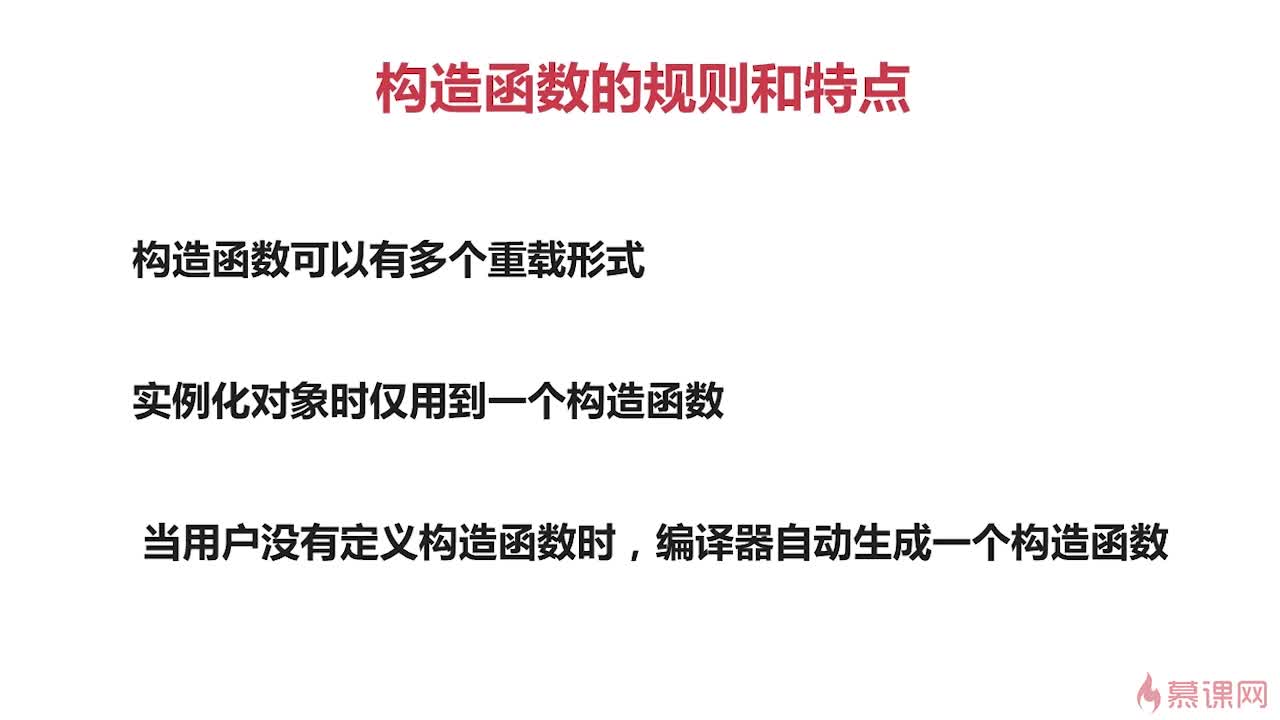
**内存分区：**

****

**构造函数的规则和特点：**

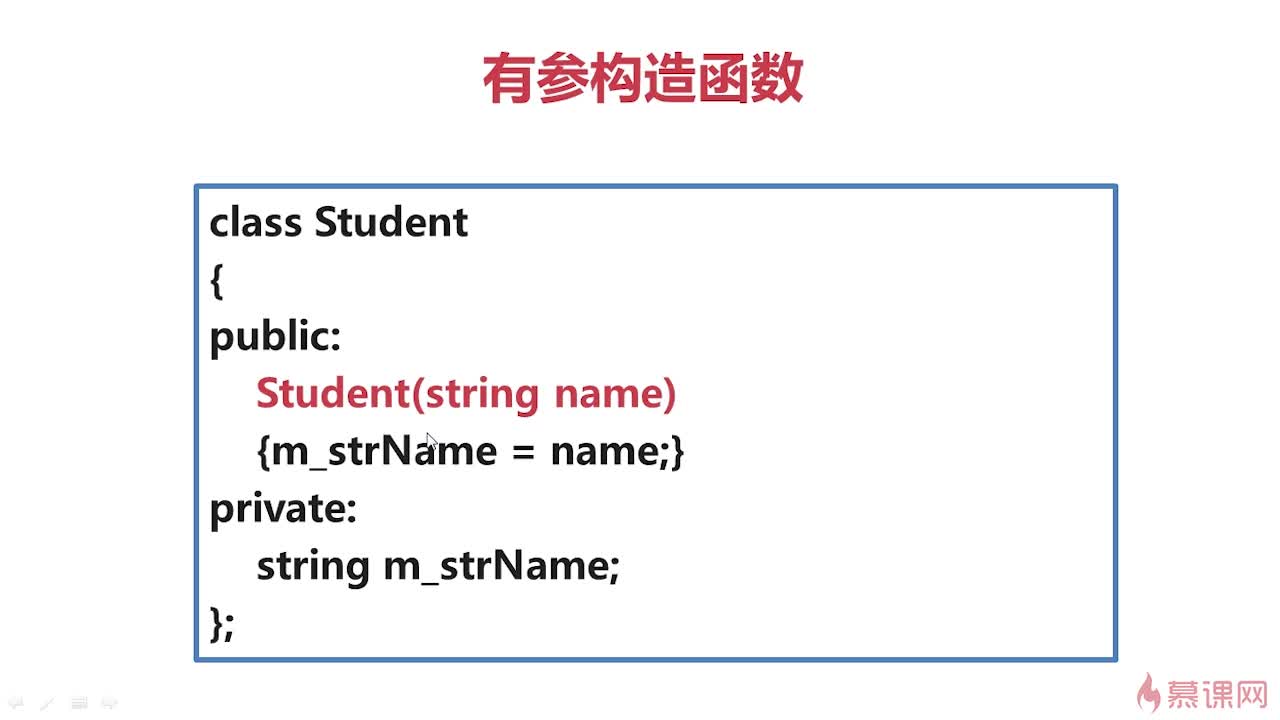
****

**由于所有的构造函数都必须与类同名，这就导致构造函数中必须出现函数重载以实现功能类似而所处理的数据类型不同的问题。**

****

**构造函数可为有参函数和无参函数：**

### （1）：有参构造函数：

****

### （2）：无参构造函数：

****

**注意：**

**无参构造函数和有参数但是参数都有默认值的构造函数不能同时存在，因为在调用时系统不知道到底应该调用哪个，比如：**

**Teacher();**

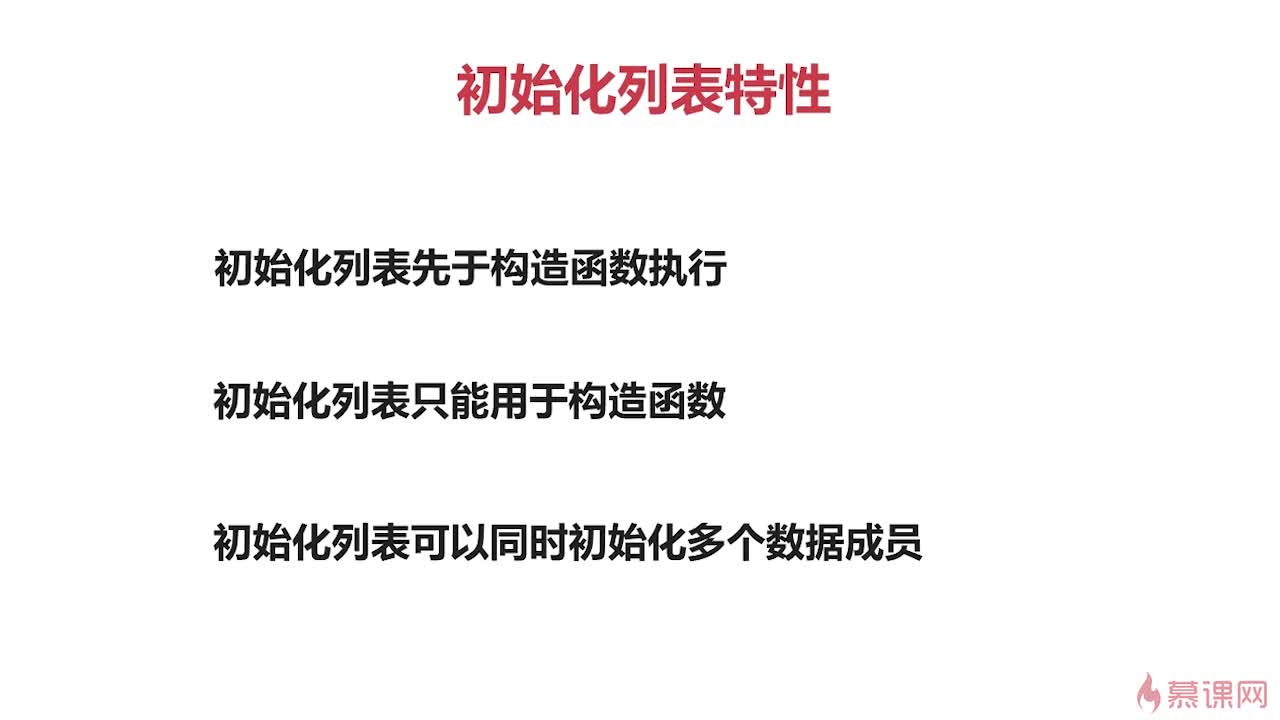
**Teacher(string strName="Lucy",int strAge=20);**

**上面是两个构造函数，在调用的时候如果写：**

**Teacher()；**

**这个调用方式是调用第一个构造函数的唯一方法，要想调用第一个构造函数就必须要这么调用，但是，这个调用方法照样可以调用第二个构造函数，因为第二个构造函数的参数全部被初始化了，在调用的时候可以没有实参，这样，编译系统在编译的时候就不知道到底该调用哪个构造函数，所以就会出现编译不能通过的错误。**

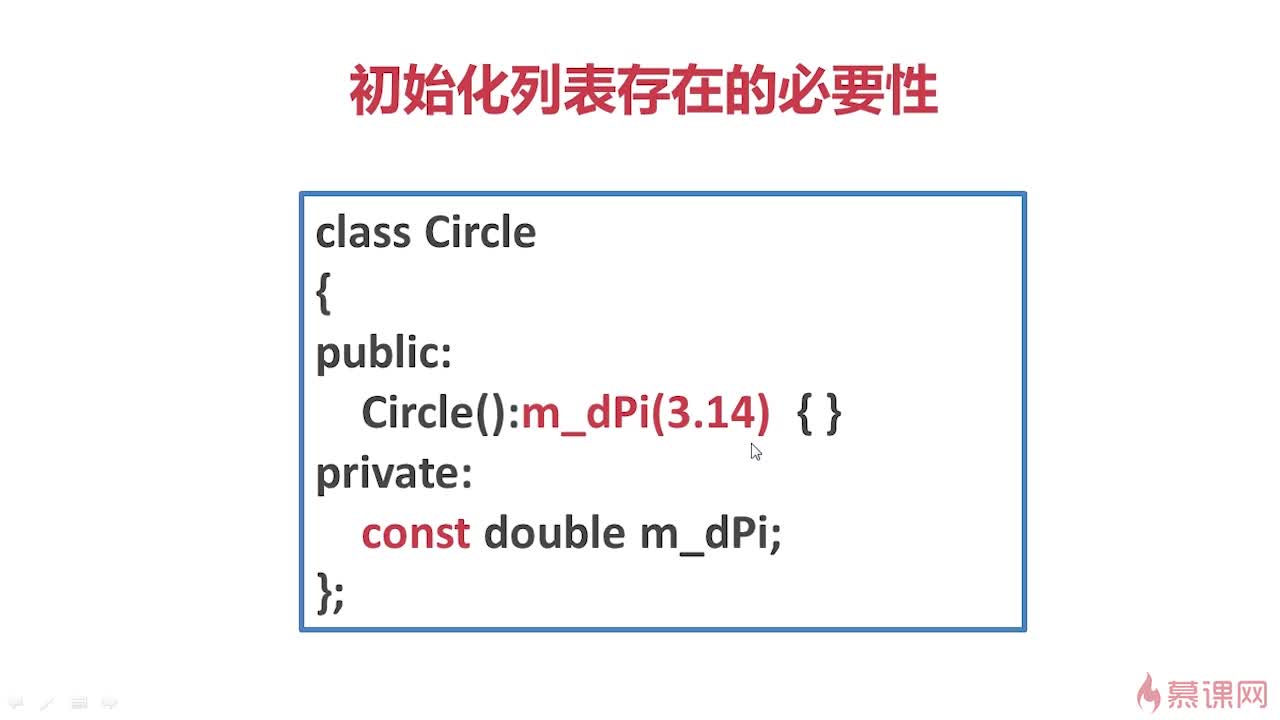
### 2：构造函数中的初始化列表：

****

**举例：**

****

**初始化列表的与构造函数相比的优势：初始化列表可以对const修饰的常量赋值**

****

**如果写成Circle(){m\_dPi=3.14}**

**编译时就会出错，因为m\_dPi前面有const修饰，所以不能用构造函数来进行初始化，必须用到初始化列表，这就是初始化列表与构造函数对比起来的优越性。**

### 3：拷贝构造函数：

**拷贝调用函数存在的意义：在拿实例化后的对象作为实参传递给调用函数的形参或者把已经实例化的对象的值拷贝给新定义的一个对象的时候会调用拷贝构造函数**

**比如：**

**Teacher t1;**

**Teacher t2=t1;**

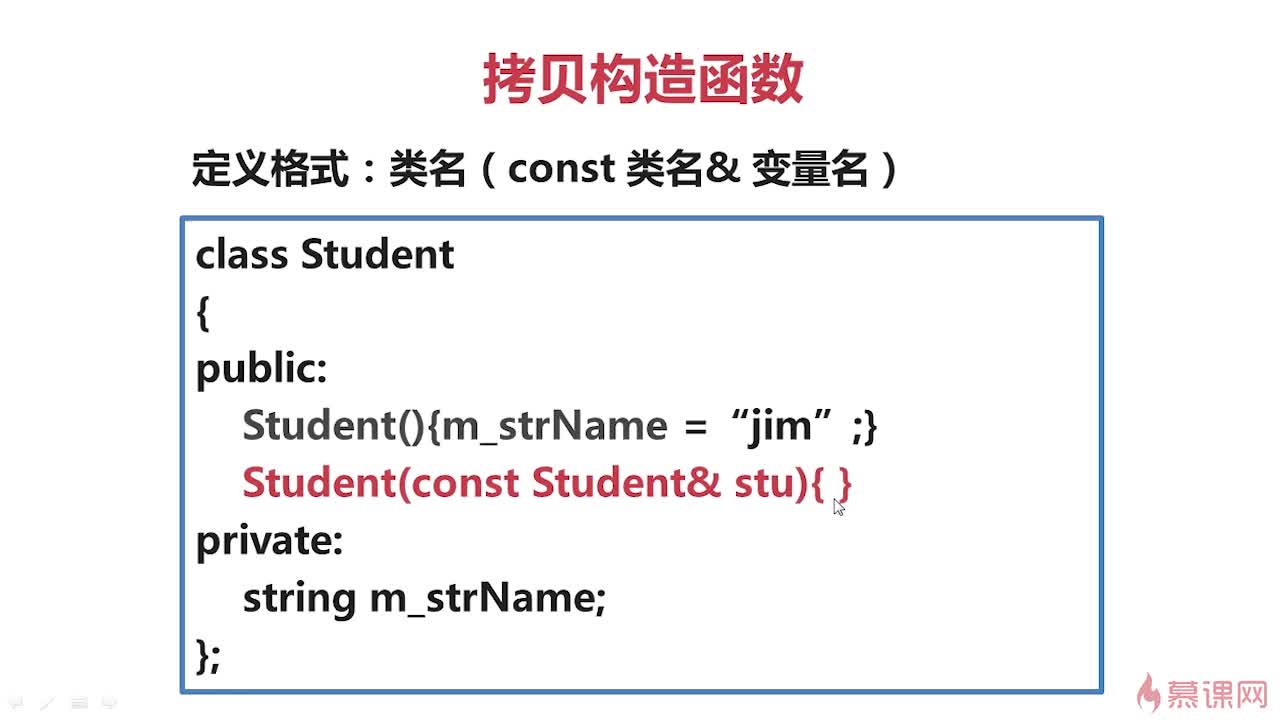
**Teacher t3(t2);会出发拷贝构造函数的调用**

**void test (Teacher t){}**

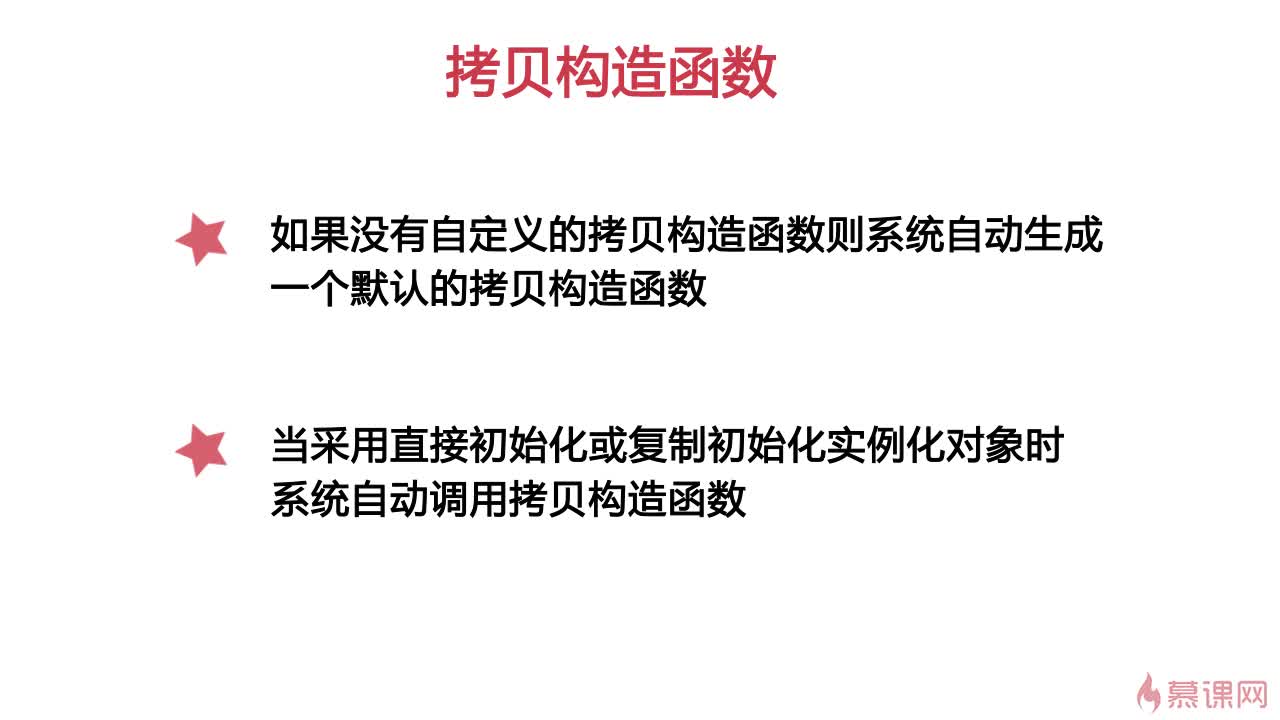
**Teacher t1;**

**test(t1);参数传递会出发拷贝构造函数**

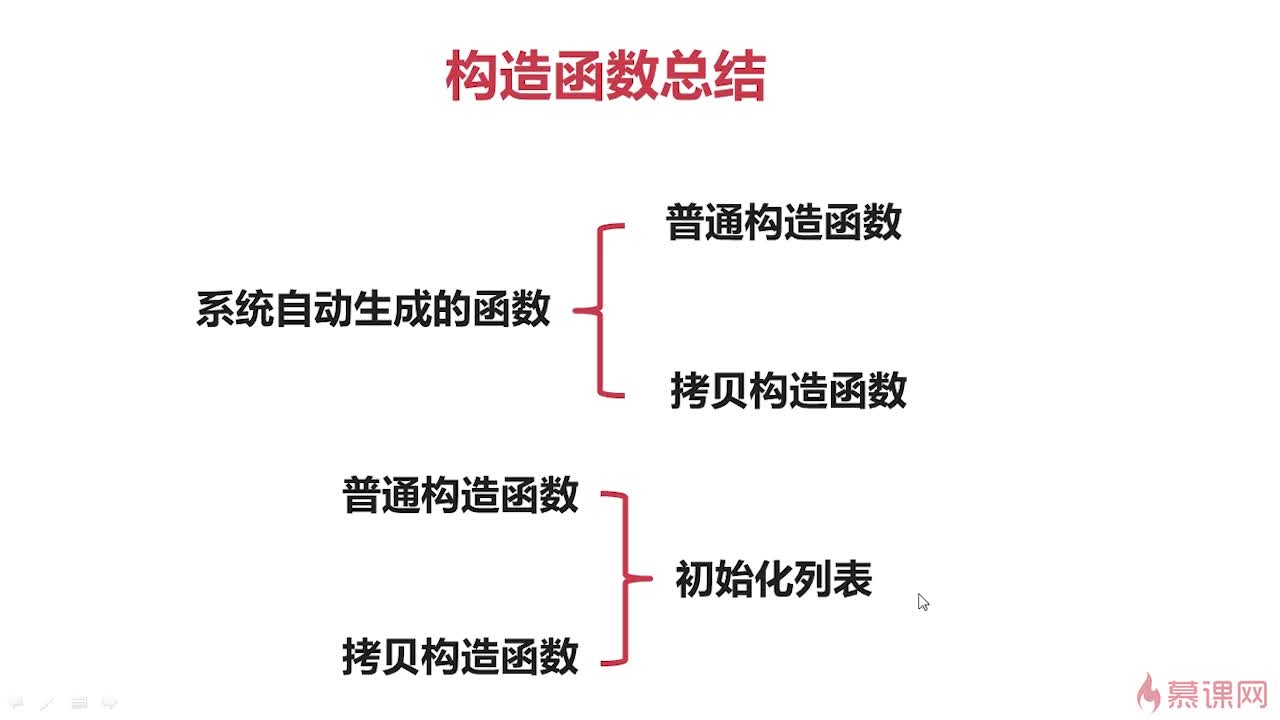
**拷贝构造函数格式：**

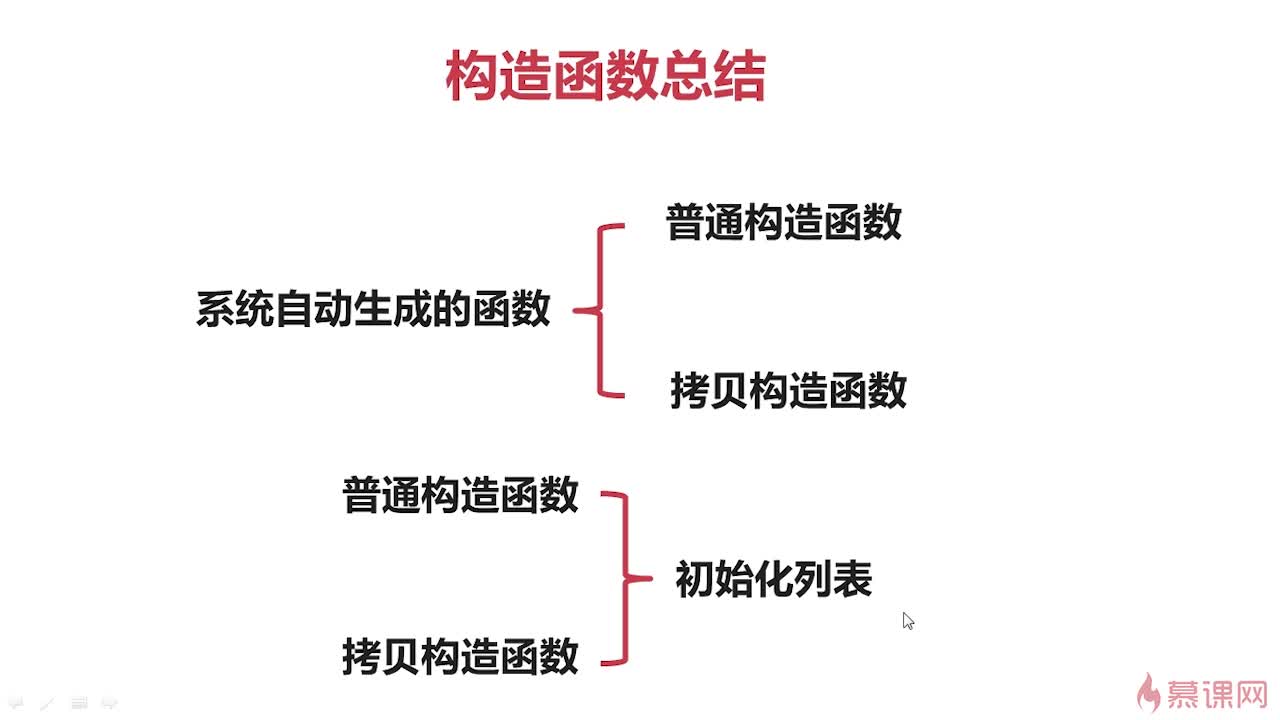
****

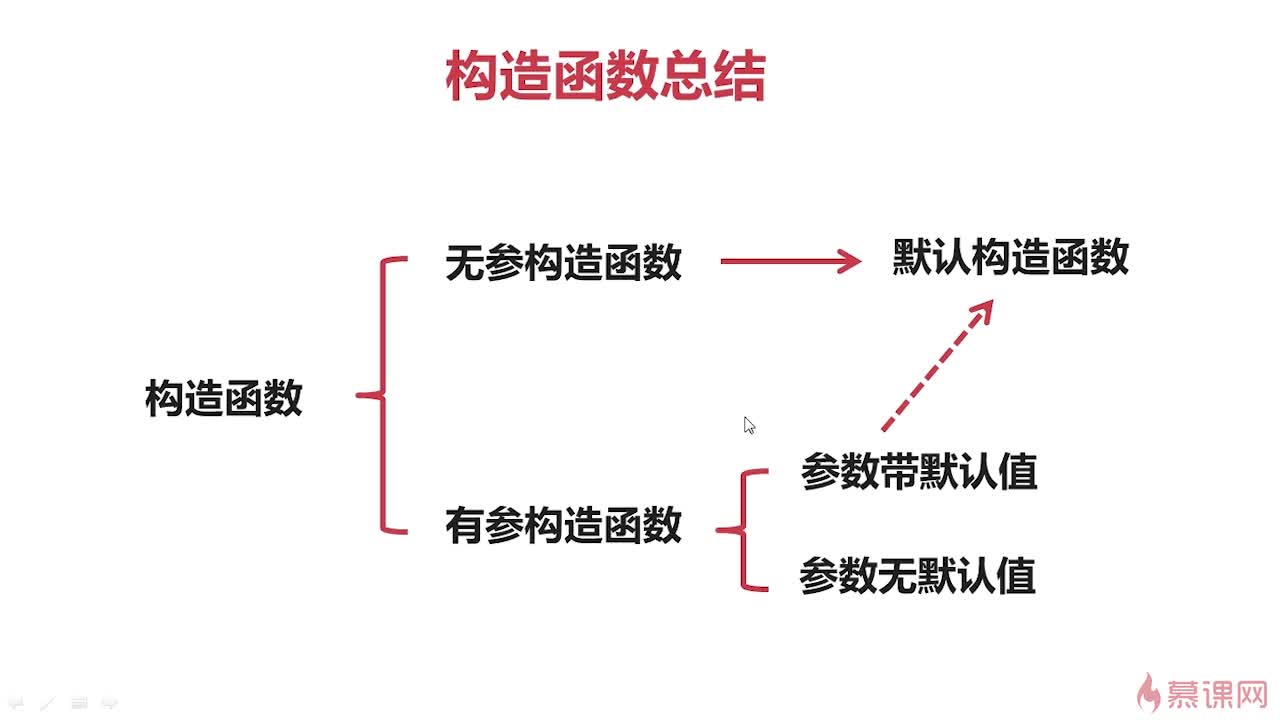
**注意：拷贝构造函数的参数是确定的，因此，不存在拷贝构造函数的重载。**

****

### 构造函数总结：

****

****

****

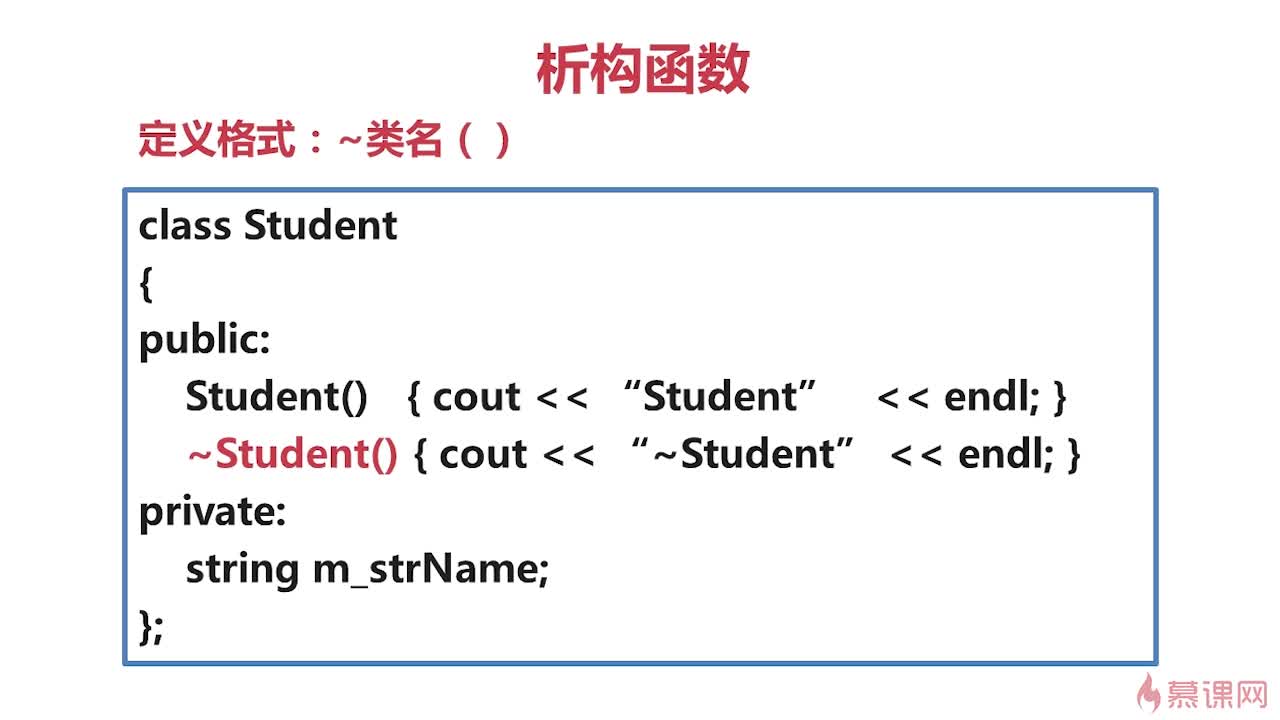
## 析构函数

**在对象被销毁之前自动调用析构函数，主要作用是释放内存；**

**析构函数使用方法：**

**用堆实例化对象，析构函数会在所有进程结束也就是窗口关闭后自动调用；而用栈实例化的对象，在用delete语句销毁时，自动调用析构函数；调用拷贝构造函数的对象也是同堆实例化的对象一样在窗口关闭之时自动调用析构函数**

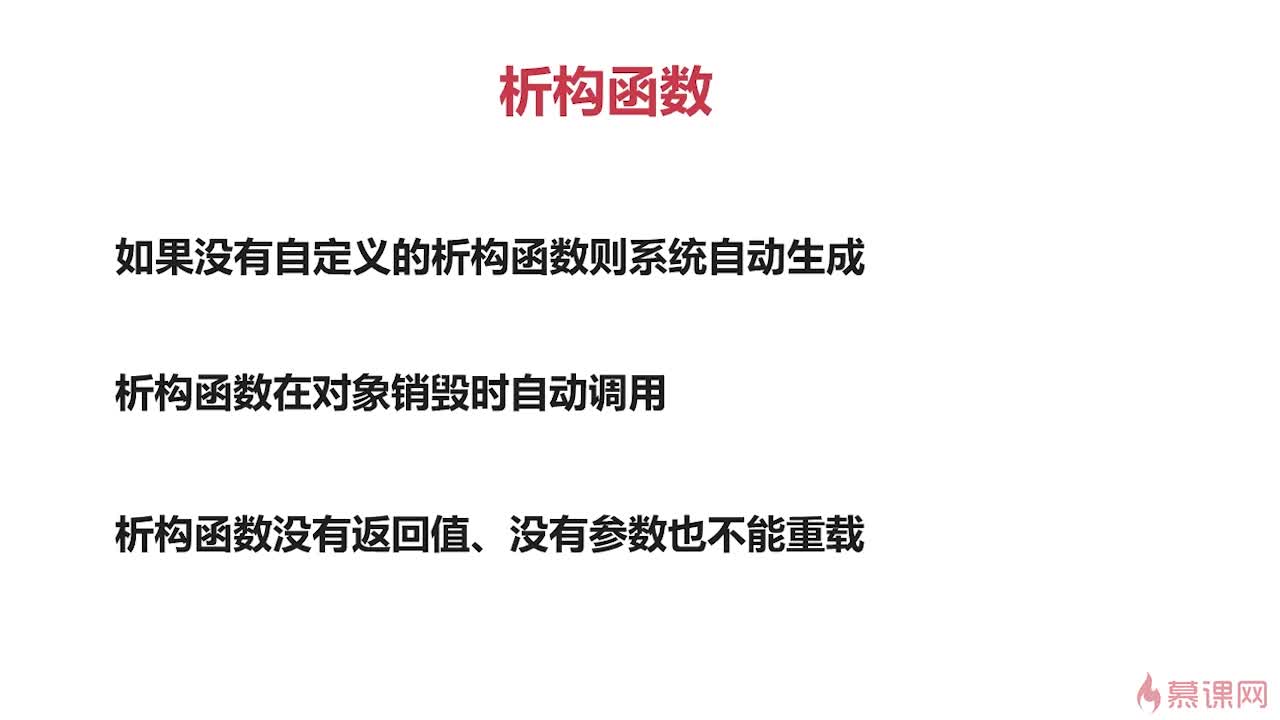
**析构函数在栈、堆中都能被调用，但是在堆中不会自动调用，必须手动释放内存，否则会造成内存泄漏**

****

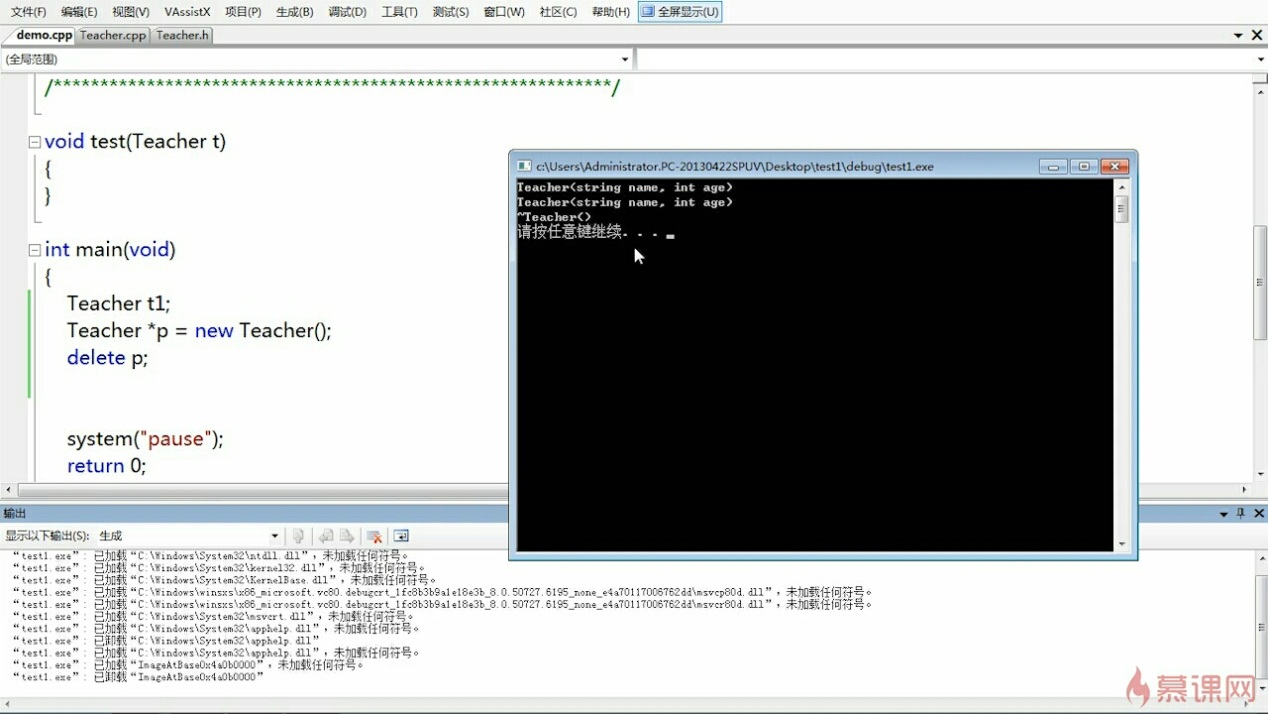
**析构函数实例：**

****

**析构函数特点：**

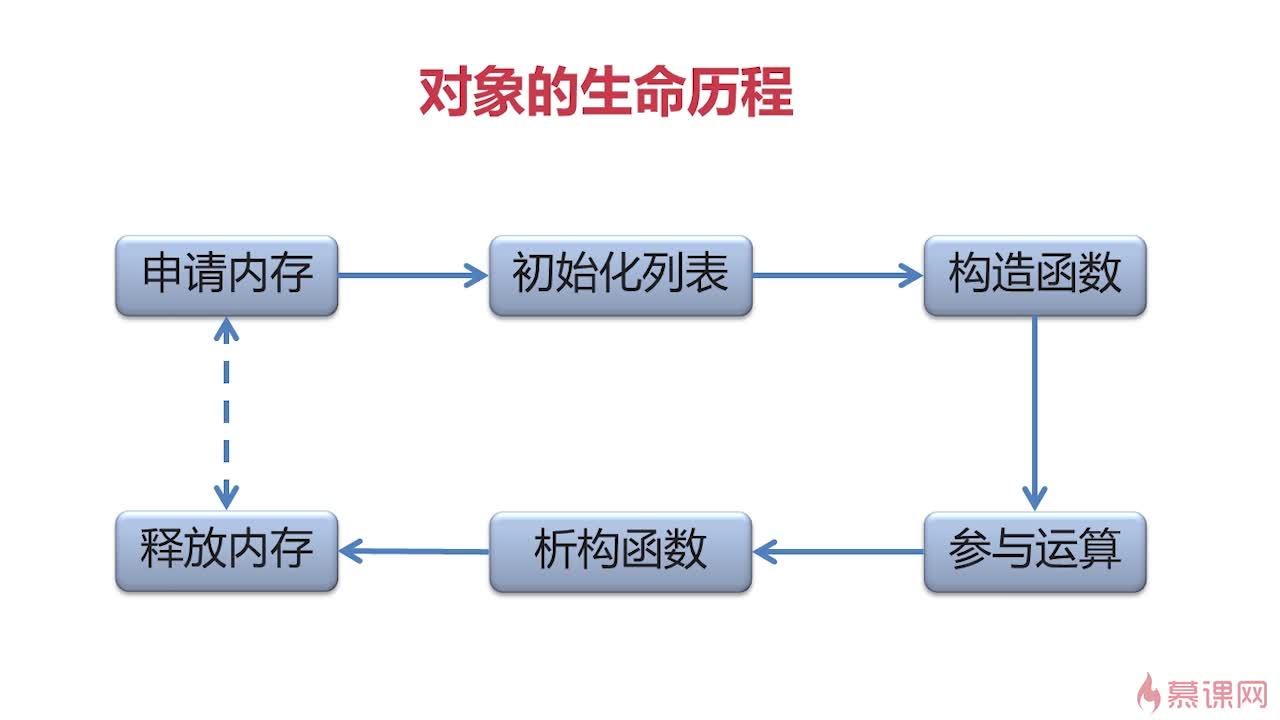
****

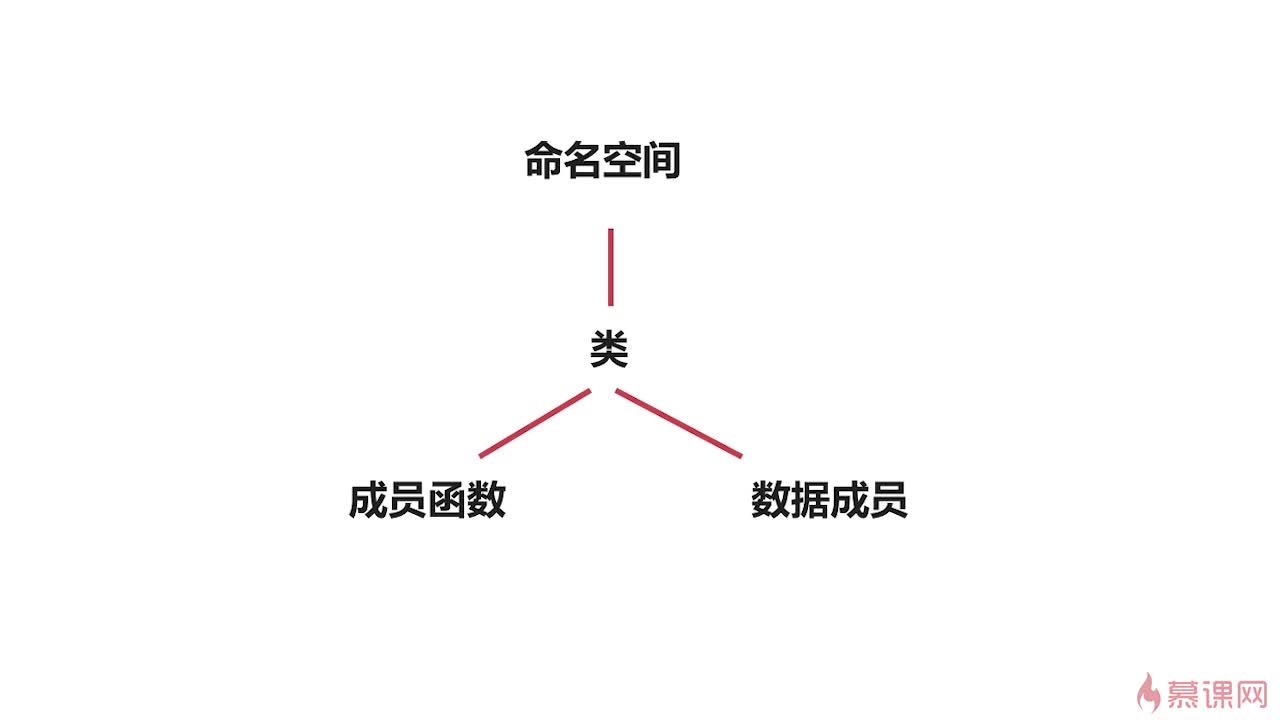
**调用delete会调用析构函数：**

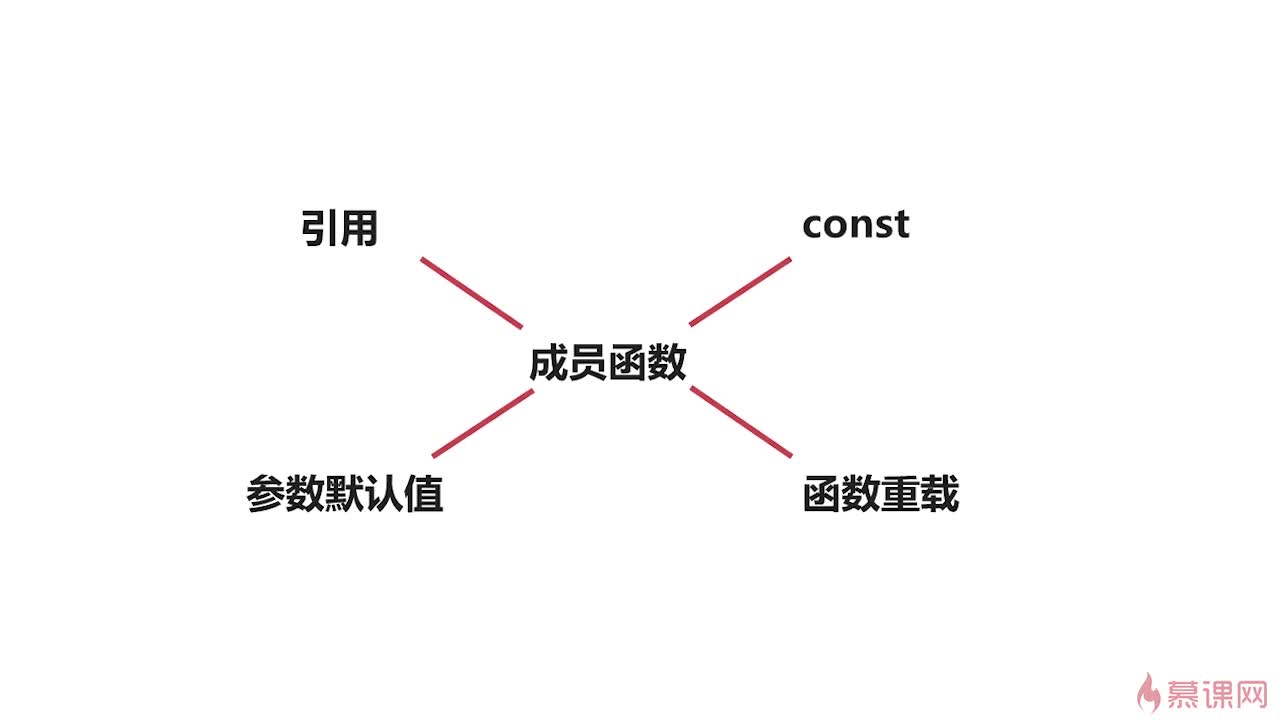
****

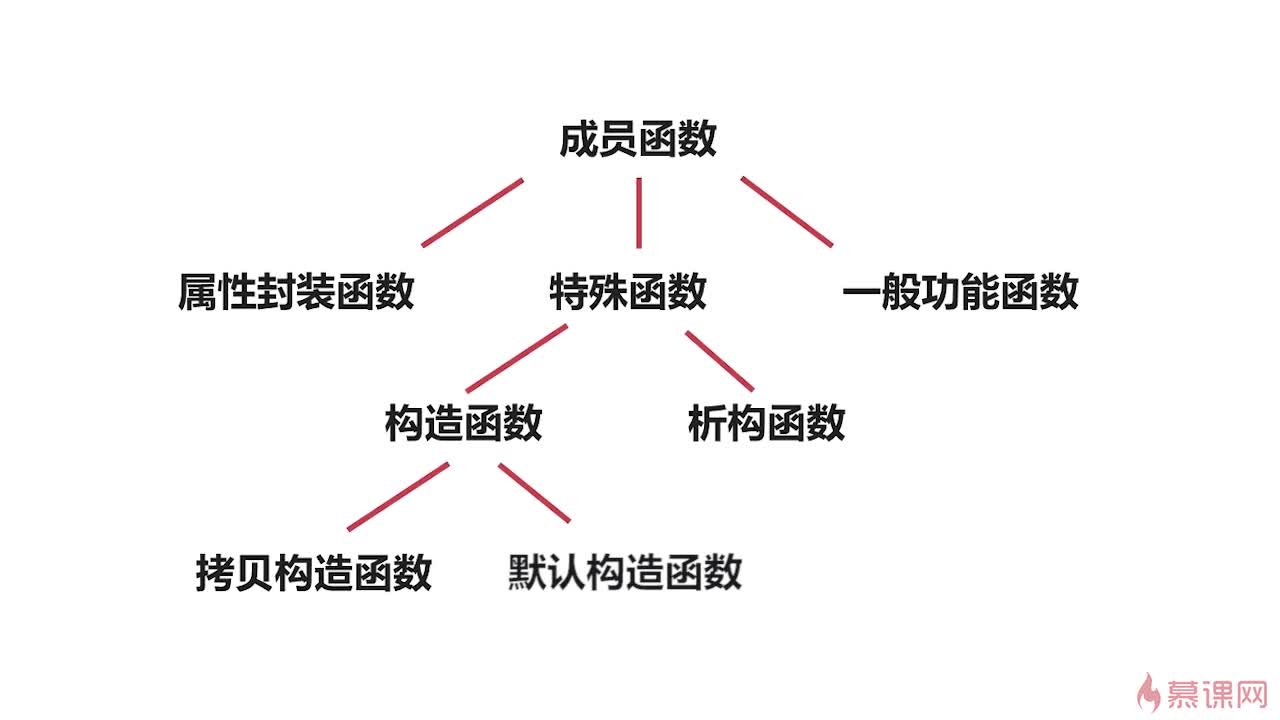
# 总结

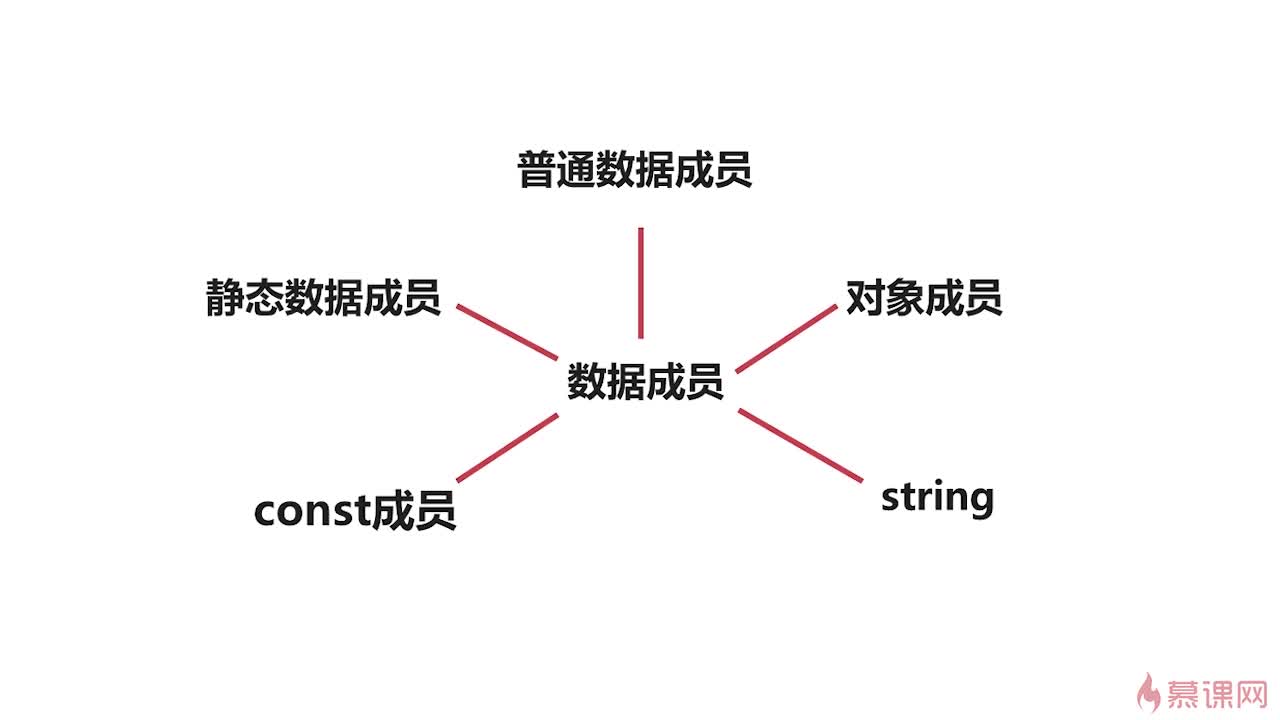
**对象的生命历程：**

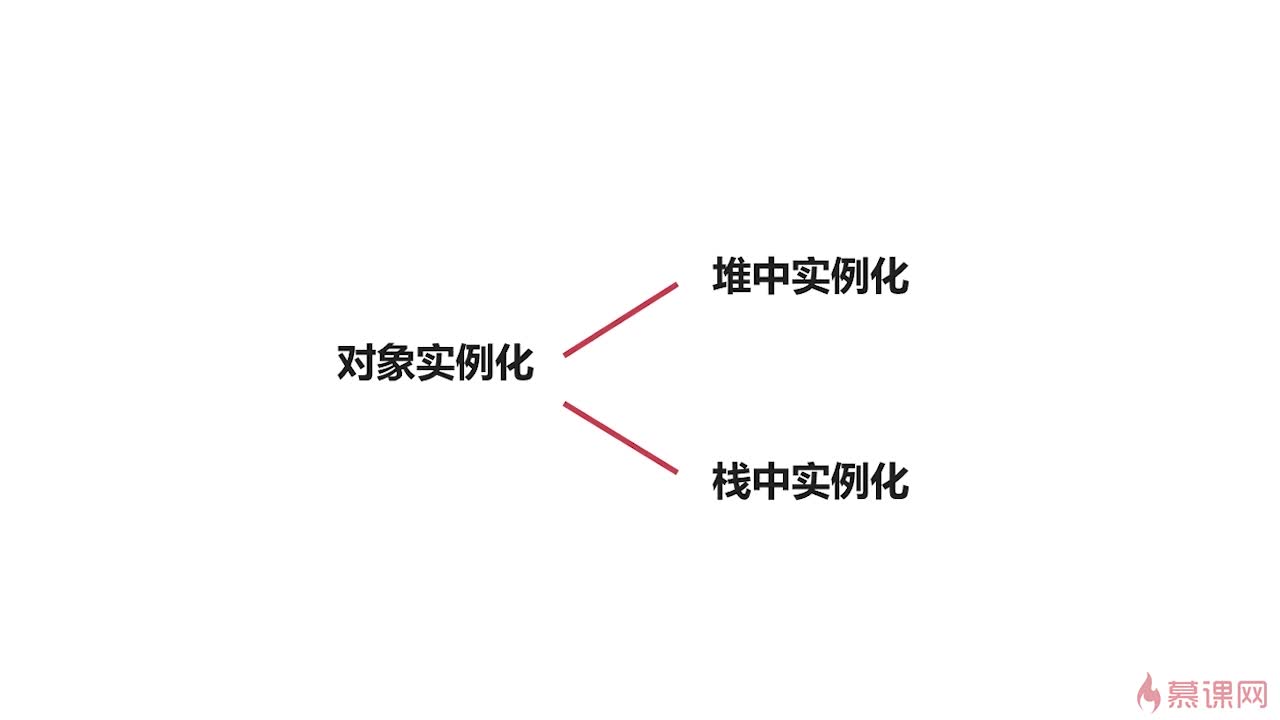
****

****

****

****

****

****