

C++ 新提供的关键字 意义和用法

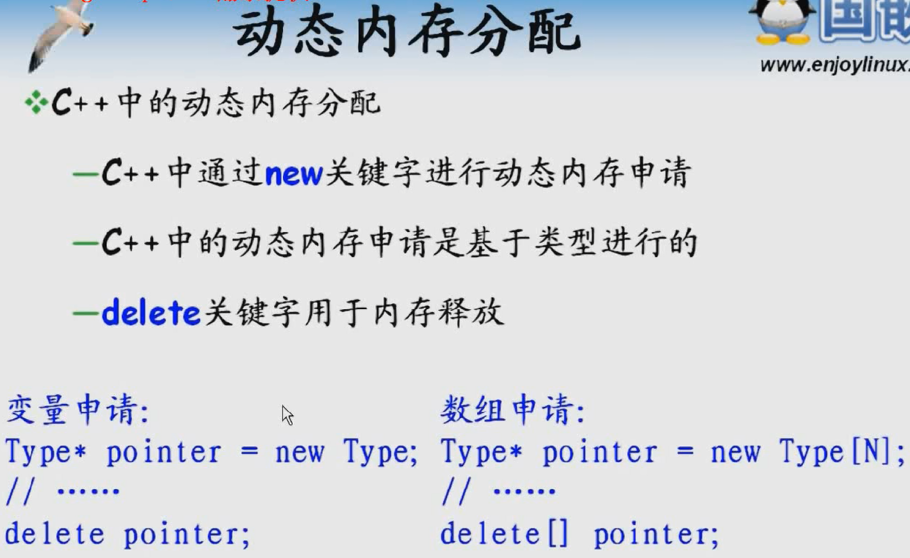
C语言中 如何动态内存的分配和释放呢？ 通过标准C库的malloc和free两个函数进行 --- 这个是标准库中的函数 所以 malloc和free 不是C语言的一部分 不属于C语言本身

但是 到了C++中 新增了两个关键字 new和delete

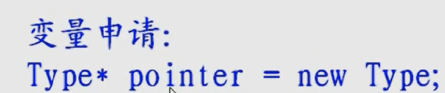
New是进行动态内存分配的专用关键字

Delete是动态内存释放的专用关键字

**所以 动态内存分配和释放是C++语言的一部分**

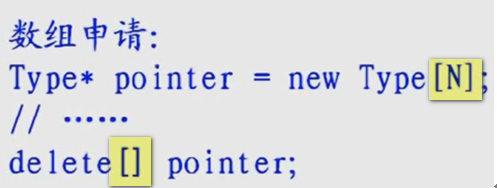


如果要动态在对空间申请变量

 这样比C语言中好理解

这个指针指向新的Type的变量

用完这个变量 直接delete就好了 【java就是new出来 就不管了 没有delete】

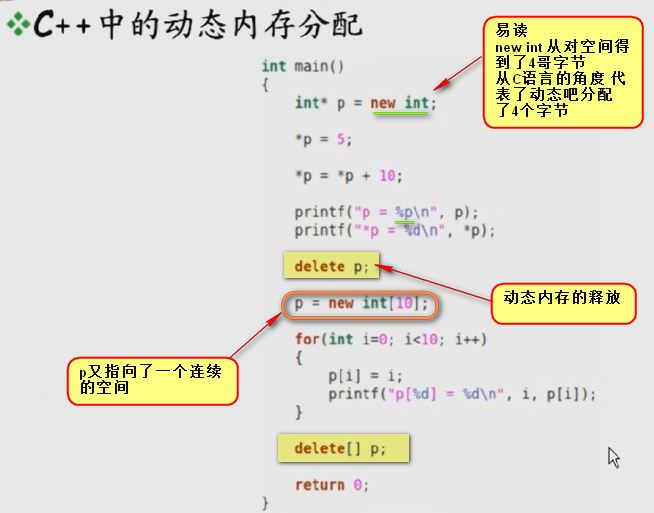


【无论是分配单个 还是数组 都是Type \* 但是 释放的时候 单个是delete 数组是delete[]】

C++中 []代表数组 ----- 这个point指向Type的数组 数组大小是N

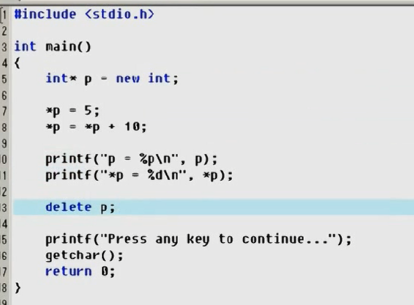
用完之后 动态释放就可以了

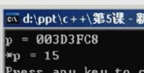
在C++如何进行动态内存的分配和释放



前后delete p; 这个是释放p指向的一个内存空间 但是 delete[] 是释放的是十个对象的连续空间

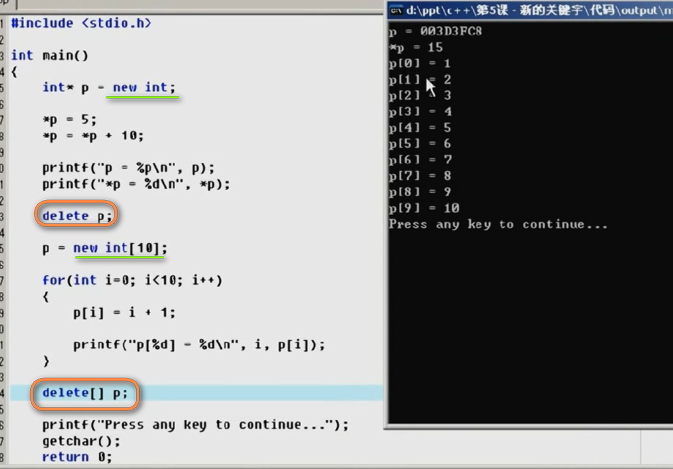
【java中是不可以写成new int这样的！】





这样p就指向了堆空间里面的变量 这个地方没有什么悬念

下面动态申请一个数组

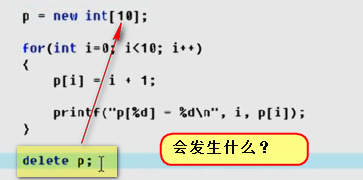


在C++里面 单个变量的动态申请 和 数组的申请 不太一样

数组的话 需要给出数组标识符 给出数组的大小

对于数组情况 如果 我们动态申请的是数组 但是 释放的时候 是delete p;. 而不是delete[] p;

会有什么情况呢？



按照C++编译器的习惯 这么写 delete p; 就认为p指向一个单独的类型变量 --- 这样仅仅释放一个变量而已 ---- 这样就会产生内存泄漏

 告诉编译器要申请10个int 但是  告诉编译**器仅仅释放了p[0] ---- 造成九个元素的内存泄漏 ---- 4\*9 就会造成36个字节的内存泄漏**

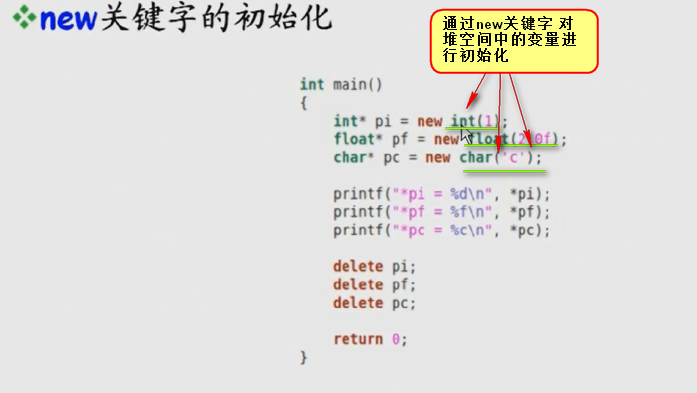
**在C语言中 都是free统一的方式来释放 但是 如果C++还是delete 这样就会造成内存泄漏 务必记住**

讲了这个例子之后

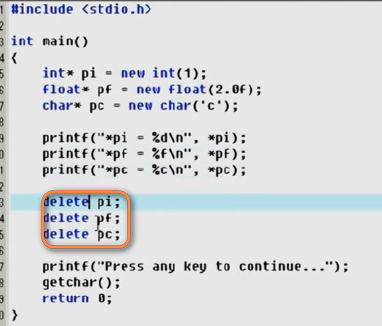
**Malloc函数和new关键字的区别**

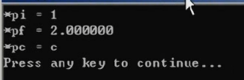
因为C++兼容C语言 所以 C++中用malloc是绝对没有问题的





如果不归还堆空间 delete 就会内存泄漏





通过这个例子 malloc函数和new关键字的区别 ---- malloc函数就非常麻烦 使用new更加高效

=============== C++的新的概念 命名空间 ===============================

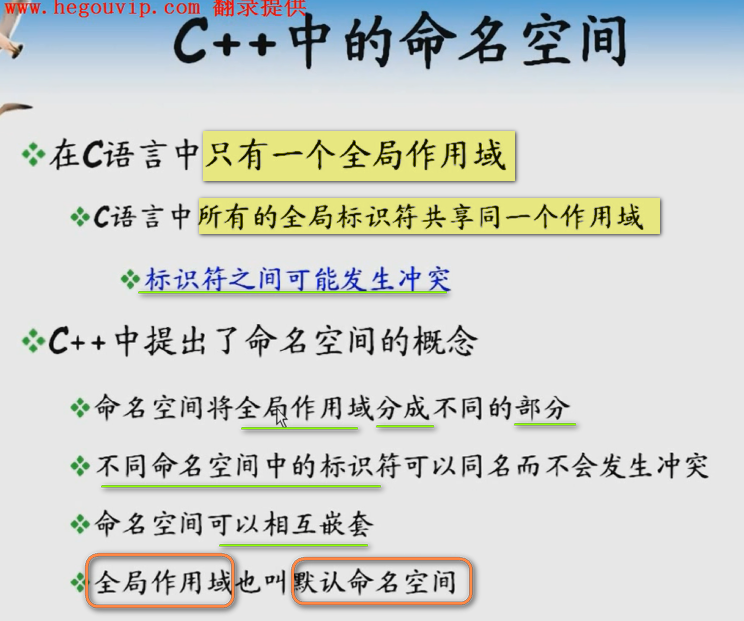
**C语言只有一个全局作用域**

C语言中所有的全局变量 全局数组 全局标识符 都共享一个全局作用域

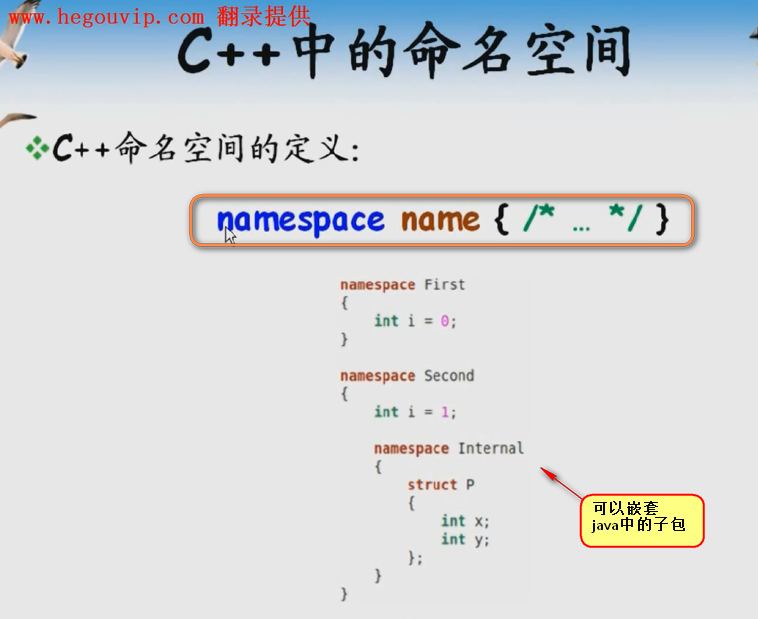
当我们的代码越来越多 标识符之间就可能发生能冲突 ---- 很多研究程序设计的学者 提出来 不用全局变量 或者少用 这种冲突就小了 但是 这仅仅是倡议 但是 无法解决标识符真正的冲突 倡议会有人遵守么？

**C++就提出了命名空间的概念 ------ C语言的全局作用域有多大？ 无限大**

从理论上是无限大 为什么不进行分类呢？



C++如何处理原来的C的全局作用域 ----- 为了兼容 就是C++中的命名空间

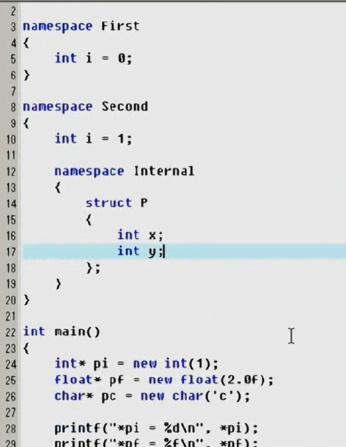


这两个i其实不会发生冲突 编译是可以通过的 因为编译器发现位于不同的命名空间里面

在Second这个ns中 又一次 声明了一个Internal这个ns

***命名空间的嵌套合理的原因***

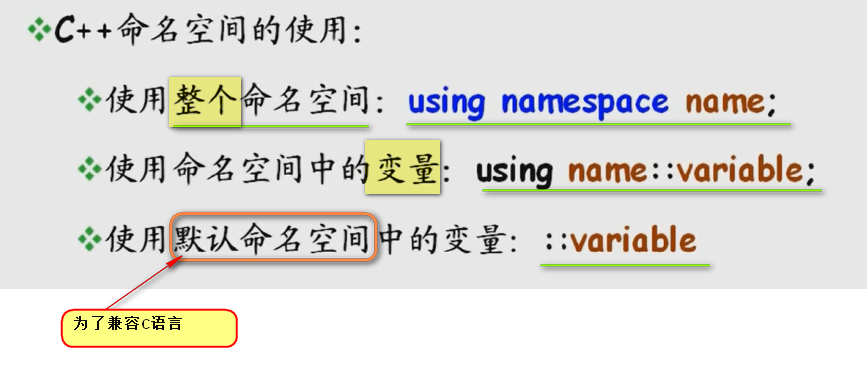
**命名空间就是把作用域中划分块的 所以 在一个已存在的命名空间继续划分块是合法的**

 编译一下 编译通过

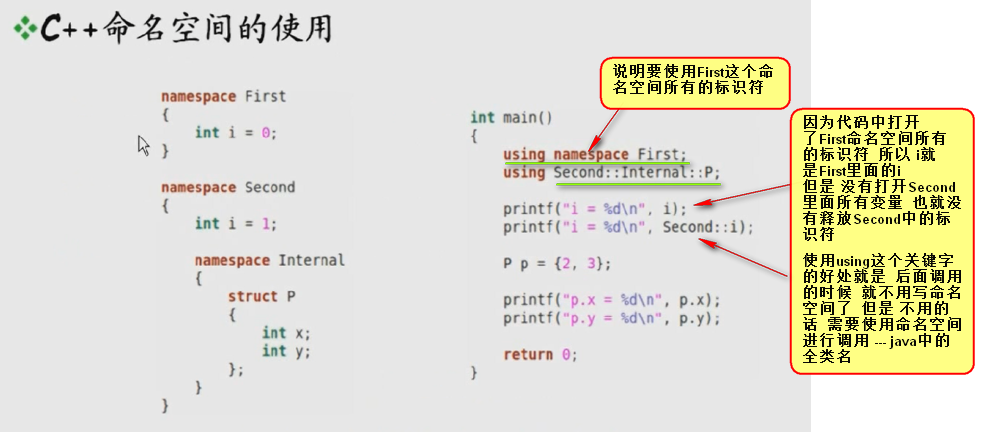
**意味着命名空间确实起作用了 并且不同ns的同名变量还不会冲突**

如何使用命名空间中的变量？

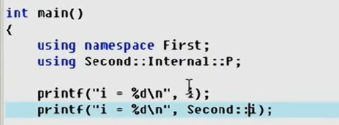
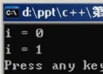
**【java中 实际上 对外访问最小单位是类和对象 所以 没有全局变量一说。。。。 所以 C++中对应的命名空间的变量 在Java中找不到目标可以类比 】**



默认情况下 为了兼容C 所以 默认情况下 默认命名空间 是可以使用的 可以不用using这个关键字

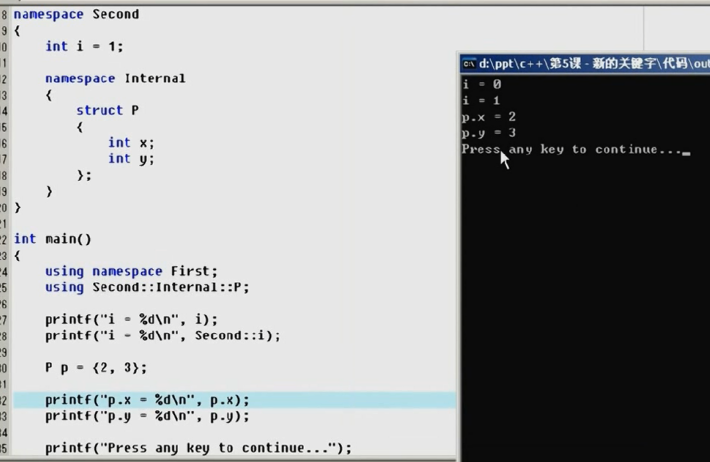


因为using Second::Internal::P; 已经使用了 所以 打开了 所以 后面使用的时候 直接使用 P p = {2, 3};

打印出来0和1

使用p这个标识符 定义一个变量 并初始化2 3



【可以看到using 可以放在程序中的任意位置 但是java中的import是不可以随便放的】

这个就是C++提供的解决原来C语言中全局作用域可能产生命名冲突的问题

ns简单 但是大型C++项目中是非常有用的

=========== C++强制类型转换 ================

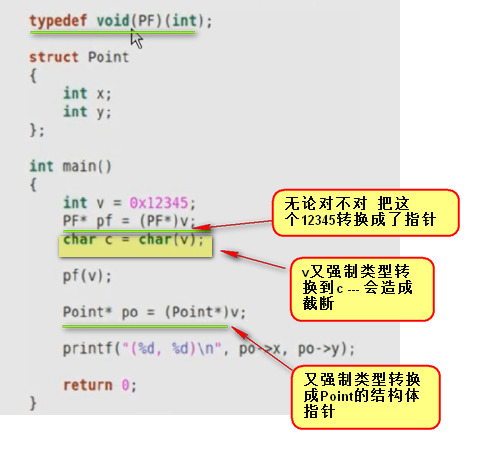


C语言中的强制类型转换非常好用 ----- 想怎么转换就怎么转换

有没有什么问题？

**C语言中 强制类型转换 (Type)(Expression) 或者 Type(Expression)**

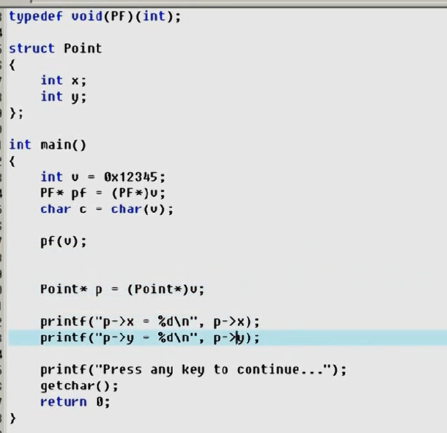
第二种看上去类似于函数的调用 两者没有本质的区别



体验一下C语言中的强制类型转换

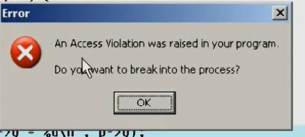
可以看到 这个int的v被胡乱强制类型转换成任意的类型

 然后 pf(v); ---- 请问0x12345这个地方真的有函数么？不一定有



编译一下 C++编译器 也没有认为报错

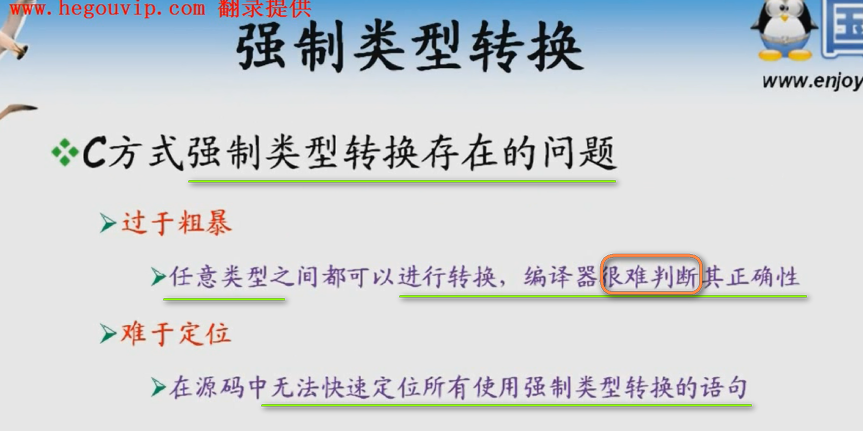
运行的时候



从这个整个程序的逻辑 就是瞎搞 特别是 

但是 为什么会有这样的pf(v); 因为C语言的强制类型转换特别灵活好用 ----让你的程序有了bug

通过刚才的烂程序 C语言这种强制类型转换 太过简单灵活 却给程序带来了问题 --- 说的好听 就是灵活 难听 就是过于粗暴



现在的gcc已经尽力判断强制类型是否正确

**产品中 难找的bug----- 很有可能的来源：**

**运算符优先级（位运算 逻辑运算 和 数学运算的混合运算）**

**多线程编程 各个线程的交互**

**强制类型转换**

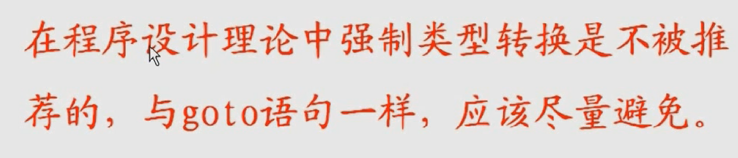
**【卫士的那个bug是不是这里面呢？】**

C的代码有5000行 能否快速判断 5000行源码是否使用强制类型转换？哪些语句使用了强制类型转换？查找 查找什么关键字来判断是否有强制类型转换？ 很难 但是 你不能说没有强制类型转换

除了char类型的强制类型转换 还有什么其他类型的？

如果使用C类的强制类型转换 很难找到

C类型的强制类型转换 问题非常大 --- 对于建立稳定可靠的软件产品是有害的



为什么要进行强制类型转换？为什么不在设计的时候 想好呢？



刚才的烂程序还可以用C++进行强制类型转换 --- 实际工程中还是可能使用强制类型转换

如何使得强制类型转换更可靠 而不是抛弃他？



这四种不同的强制类型转换 是C++推荐的

<Type>是最终要转换的目标

Expression是要转换的变量的值或者表达式

这四种不同的类型是C++提倡的

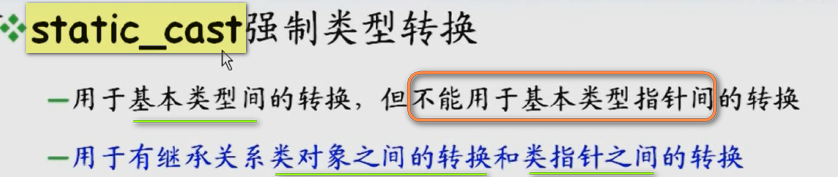
**C++把这四个词汇变成自身支持的关键字**

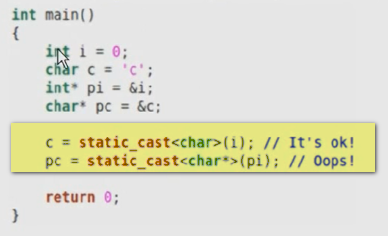
C++提倡大家使用这四个关键字进行强制类型转换 不要使用C语言的强制类型转换

如何使用？

static\_cast

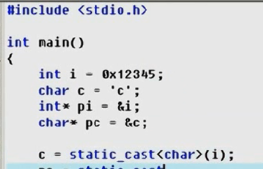
int转换成char 或者char转换成Int



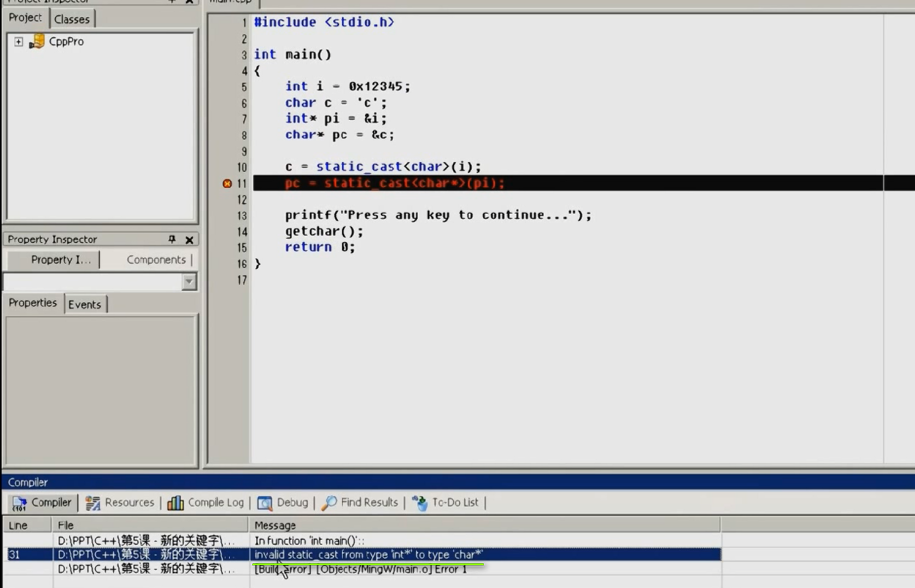


因为i和c是基本数据类型 想转换 可以使用static\_cast

但是 指向基本数据类型的指针之前不可以通过static\_cast进行强制类型转换

 编译通过

现在转换指针

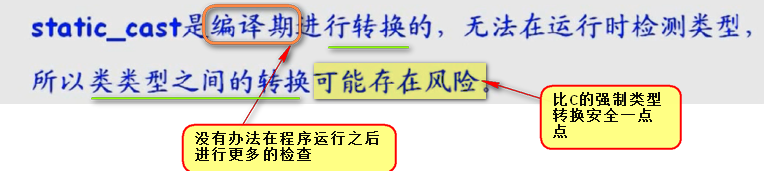


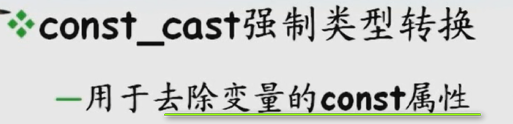
static\_cast有自己的使用范围

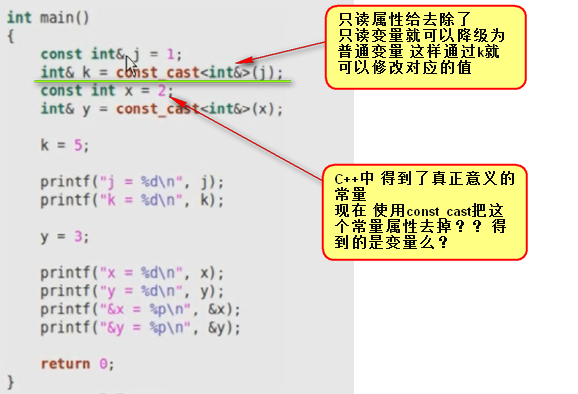
为什么要用static\_cast?

使用C++的强制类型转换 就要使用哪一个关键字的了解 可以避免很多错误

使用了这种强制类型转换 可以很容易通过强制类型转换的关键字定位问题







由于对x常量取引用了 ---- 引用实际上就是常指针 ----- 强制类型转换 const\_cast<int&> 实际上就是对x这个常量获取了地址 --- 常量的地址 就会导致编译器为这个常量分配一段内存【之前的课程说过】 ---- 这个时候y就是x这个常量的内存地址的别名

这个时候y就可以当成一个普通的变量来处理

 j被强制类型转换 变成了普通变量

 指代了就是这个x的常量的内存的别名

改变y的值 打印输出 是3

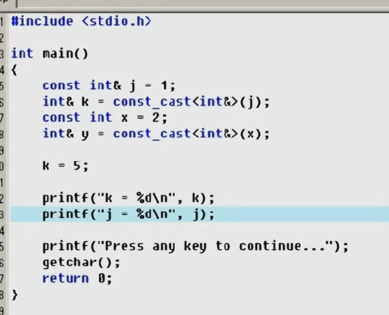
但是 x是常量进入了符号表 这个时候 x打印还是2 --- 编译器做的事情

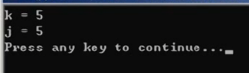
但是 y被赋值为3 就是3

但是 x和y的地址是一样的



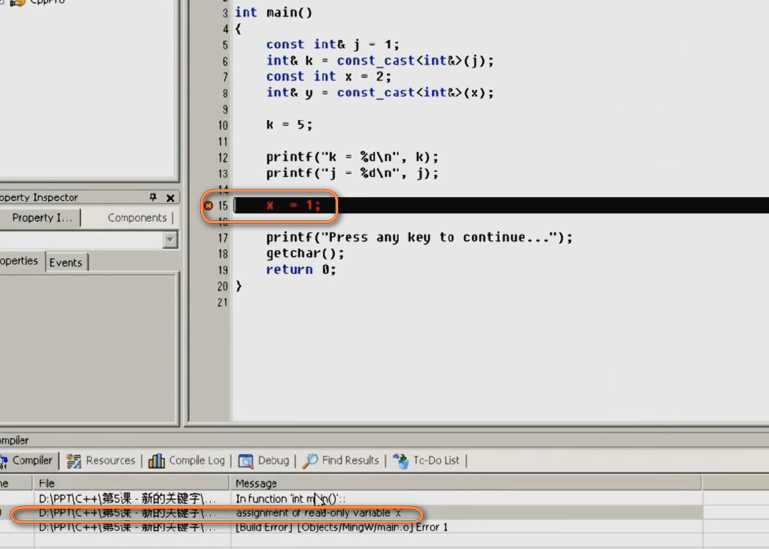
这样j就降级为普通变量了 只读属性被去掉了





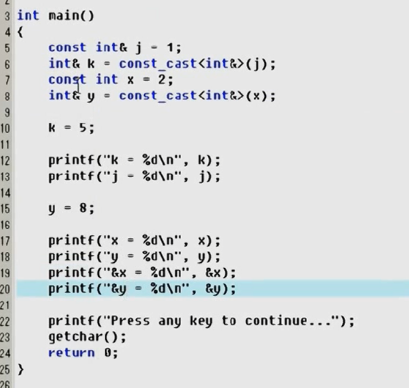
k和j都被修改成了5 ---- 这一行代码 把j这个只读变量降级成了普通变量 这样 改变k就改变了j

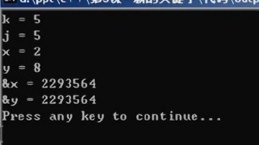
现在操作x



修改这个常量的引用

因为这个常量的内存空间变成了普通变量了





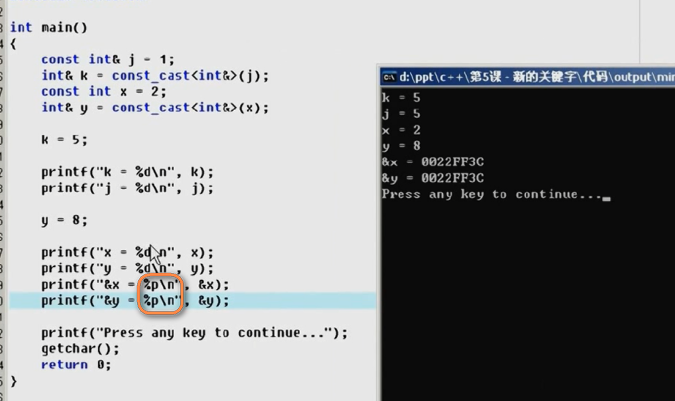
X还是2 但是y的值变成了8

我们这个常量在强制类型转换成引用之后 为这个常量分配了空间 这个空间的别名就是y

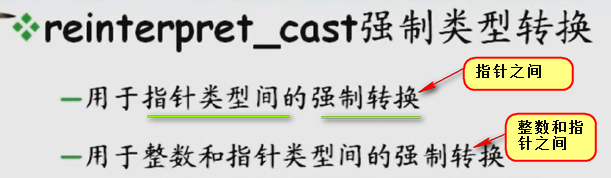
因为C++对const进行了优化 编译到17行的时候 访问x的值 直接从符号表中拿出来 替换成了2

X和y这两个地址是完全一样的

换成16进制打印

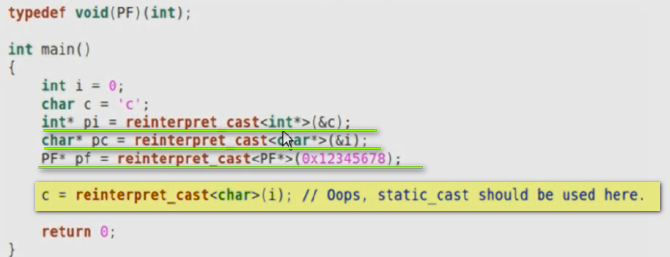


const\_cast功能非常单一 就是把只读变量的属性给去掉 除此之外没有其他功能

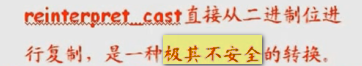


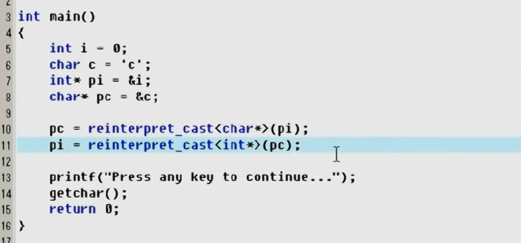
Reinterpret\_cast就是直接将二进制位进行复制 ---- 将字符c的地址 二进制数 ---直接复制到pi指针中去

Reinterpret\_cast比较强大 但是 不安全

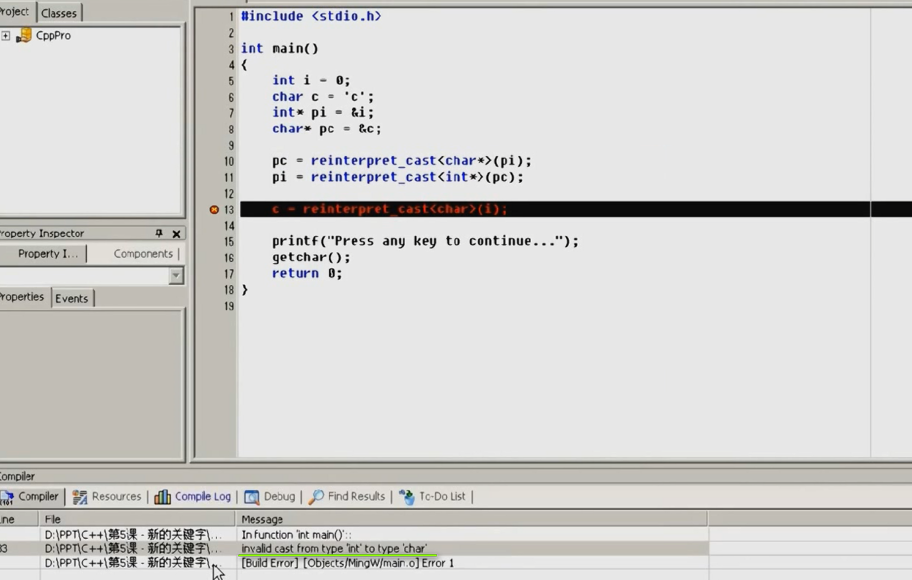


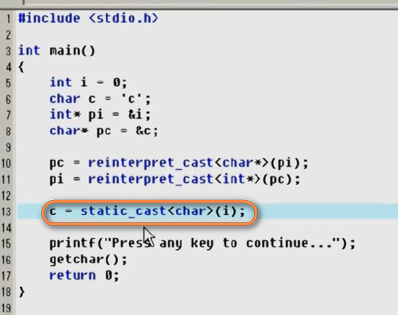
虽然可以做到 但是 不安全 没有意义 【字符就是整数 所以 可以看成】



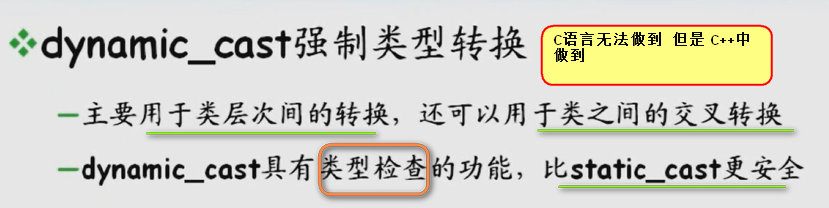


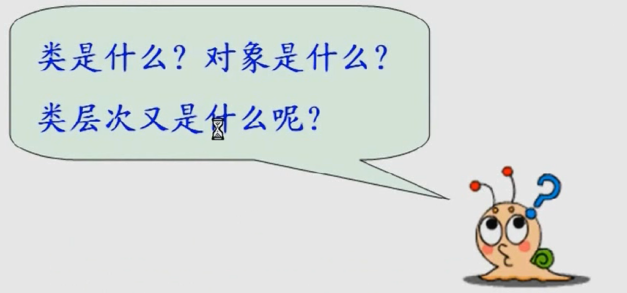
这种编译通过





reinterpret\_cast就两个 指针之间 或者整数和指针之间





--- dynamic\_cast是C++非常有用的 可以进行类型检查功能 后面提到类对象的时候 讲解这个

