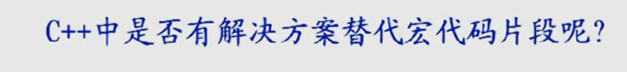


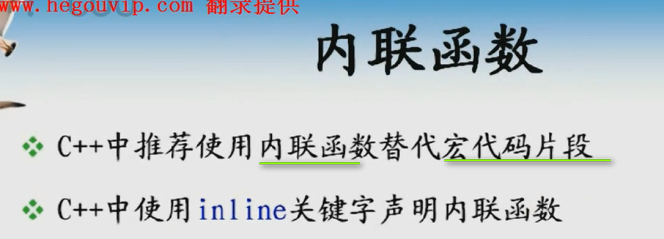
C++中的const常量是真正意义的常量 就可以替代宏定义

常量是编译器处理的 不是预处理器处理的 ---- 这样更加安全

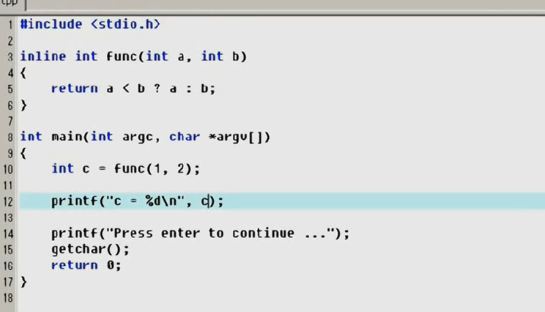
C++中 宏定义可以定义一个代码片段

 看起来是函数 但是 用起来又不是函数 ----- C中宏的副作用

------ C++中的内联函数

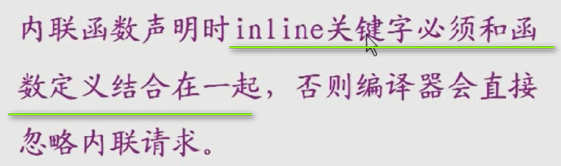


 内联函数可以替代宏代码片段了

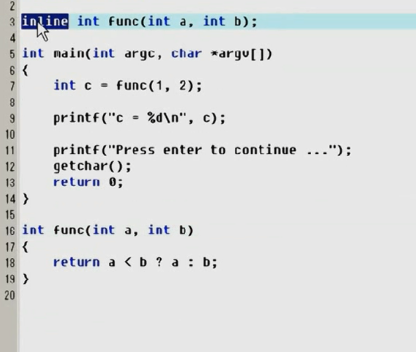


这里看起来 就是一个普通的函数调用

inline表象是普通的函数 意义是什么？



比如 这样

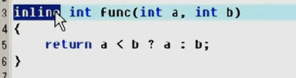
 如果仅仅在函数声明的时候 加上了关键字inline 那么这个函数是一定不能成为内联函数的

C++里面 inline关键字 如果放在函数声明时候 是不会成功的

内联函数和其他的函数有什么区别？

C++接受了内联请求 就会直接把函数体插入到调用的地方



 C++编译器编译到这块时候 就把这个func标记为内联定义

一般的函数调用 就是压栈 跳转 返回

但是 内联定义 ----- 就会把函数体 直接插入到对应的位置 直接进行替代



这样 也就是说 内联函数 比起普通函数来说 没有额外的函数调用的开销 ----- 这样内联函数没有这样的开销

---- 每一个函数都可以声明为内联呢？



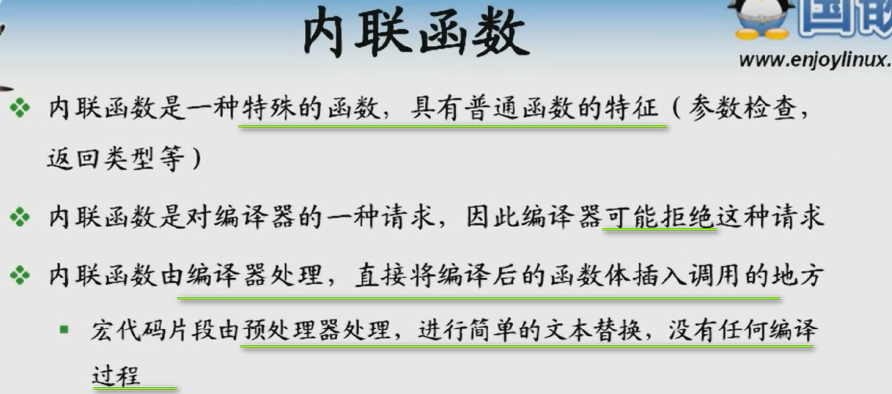
换句话说 inline和register都是向C++编译器请求 不一定最终真的可以

如果C++编译器发现你的函数不能内联编译 就会把这个函数变成普通的函数



内联函数 本质还是函数 就还具有普通函数的特征 ---特殊在没有额外的函数调用的开销 但是 对于编译器而言 还是函数 但是 宏代码是预处理器执行的文本替换

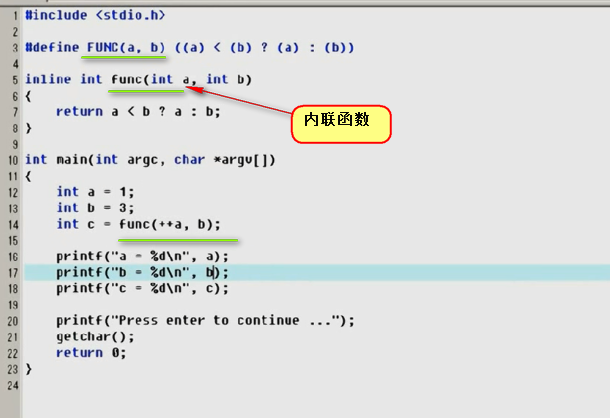
也就是 编译器看来 编译器不认识宏



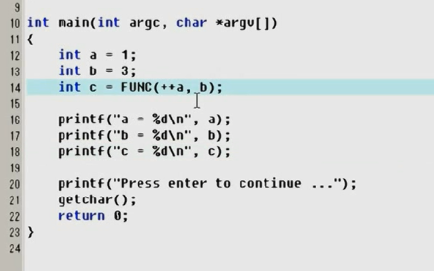


效率上 内联函数和宏代码片段一样 但是 内联函数更加安全

举例说明





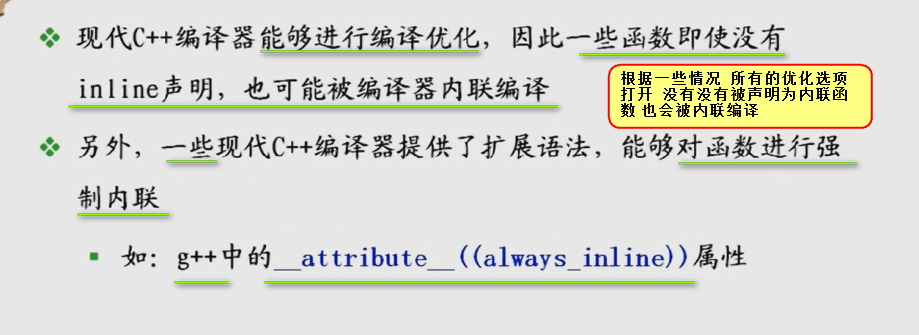


现在用的是宏

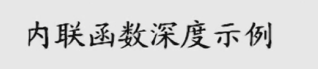


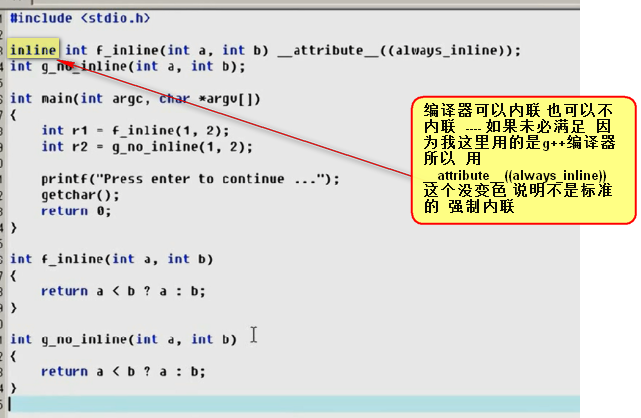
宏的副作用 ------- 之前的课程会知道这个问题 内联函数确实要比宏代码片段安全 没有副作用

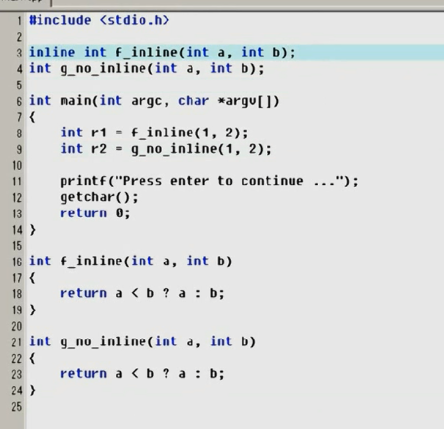
内联函数在效率上和宏代码片段效率是一致的 ---- 如何证明



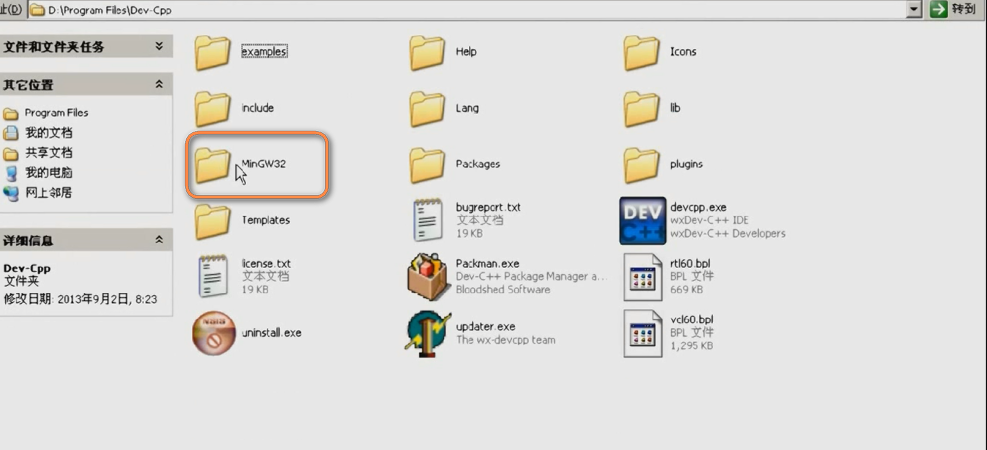
给函数定义一个属性 定义了一个always\_inline

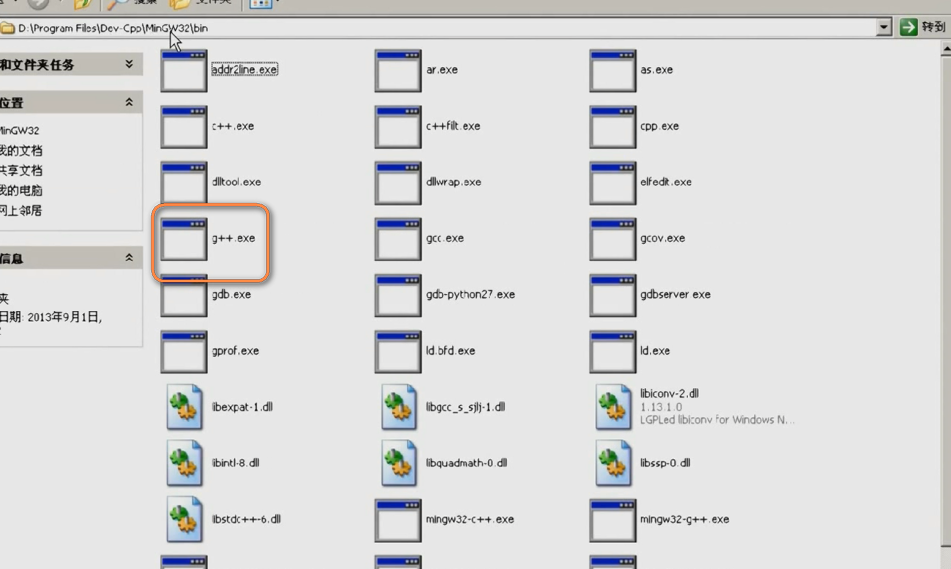


 先删去这个属性

 进行编译

在命令行使用g++ 在dev\_C++里面



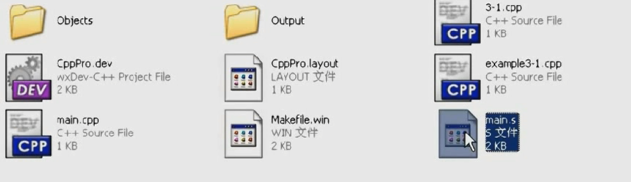


把这个路径添加到系统的环境变量里面

现在单步编译



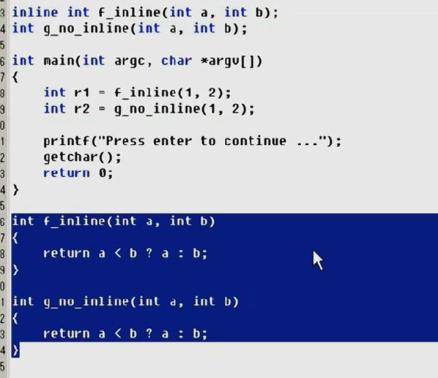
可以看到 里面生成了一个main.s



打开

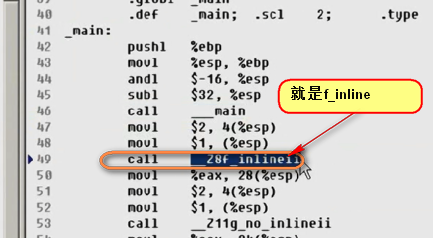




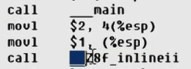


函数体一样 一个是内联编译 一个不是内联编译 ---- 现在到汇编代码来看

Main运行之后

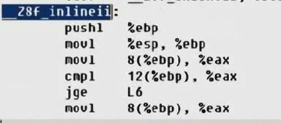


编译以后 得到的名称 编译器会对函数名进行处理之后

   
对比看

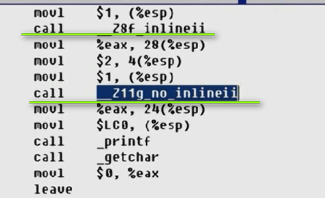
---- 可以看到调用f\_inline的时候 先把1 和 2都入栈 movl命令 然后调用call命令

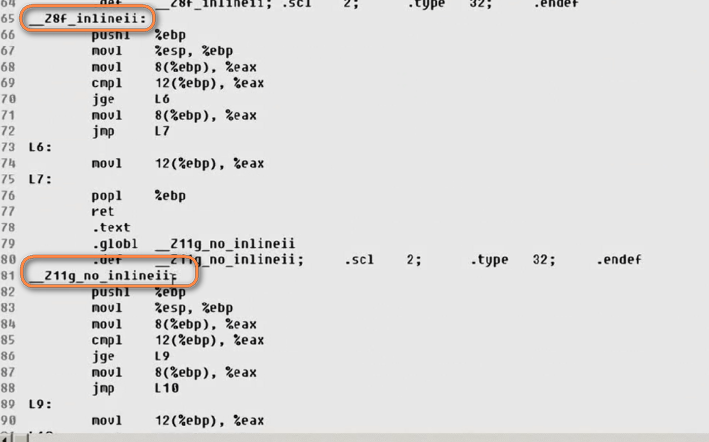
跳转到函数定义的地方来执行



这样就进行了函数的跳转

Main调用完了之后 发现

 这两个地方 做法非常类似 g的调用也是先把参数入栈 然后 call ---最后跳转到g函数定义的地方



这两个函数体是一样的 发现 会汇编里面的函数体也是基本一样

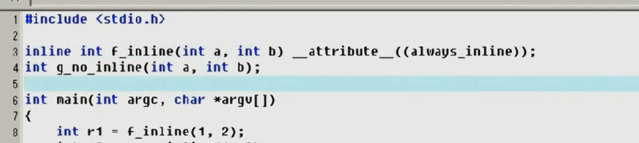
在main中

是不是和main中的逻辑一样

从汇编中可以看出 内联请求没有得到满足 【因为之前说过了 内联必须在函数定义才可能有效 如果不在函数定义 就一定无效】

因为汇编代码还是调用f\_line的语句

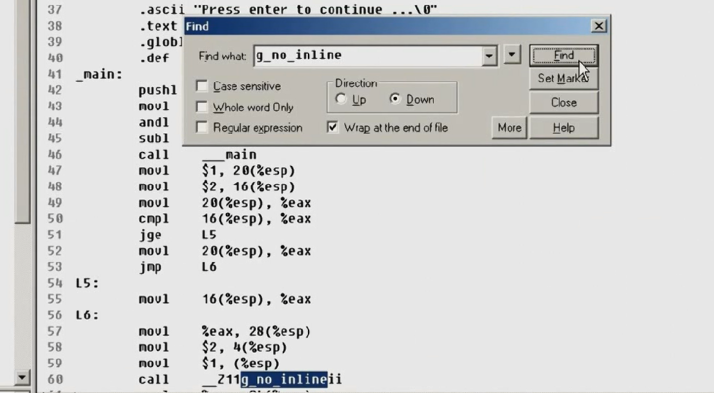
这时候 使用g++的扩展 强制成内联



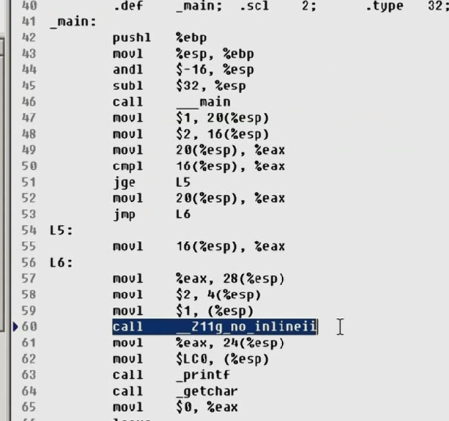
编译通过 然后单步编译



发现f\_line根本没找到 g\_no\_line就找到了

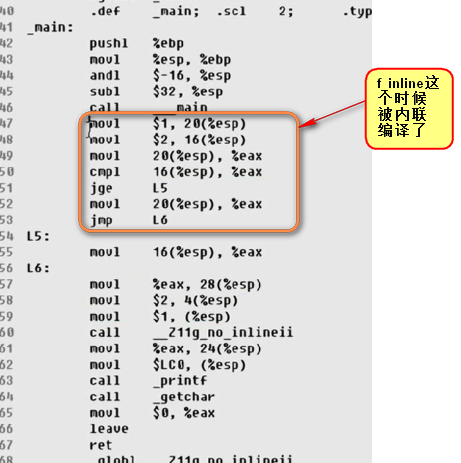


在main



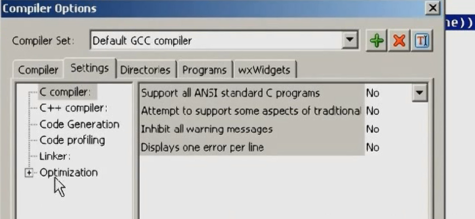
F\_inline没有了 在main开始之后 只有一个call g\_no\_inline

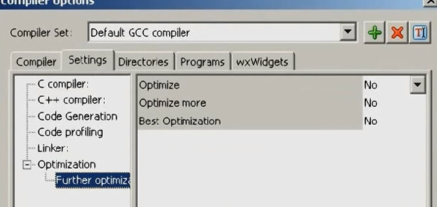
汇编代码中 没有f\_line的定义



设置编译器

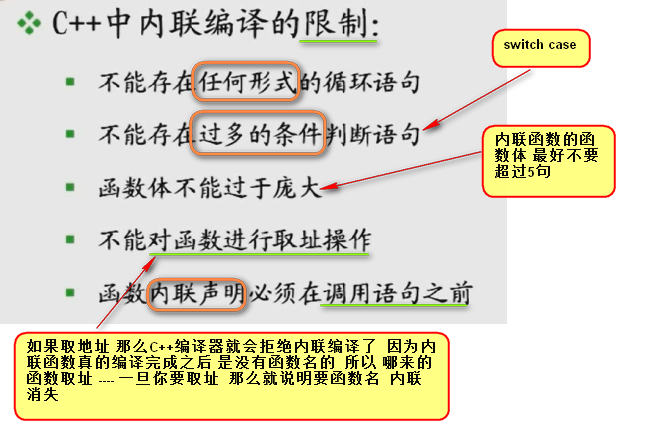


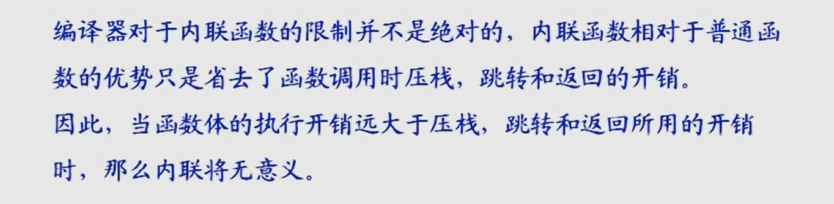




所有的优化选项都没有开 所以g++非常有可能对于内联编译的请求就拒绝了

为了代码的一致性 最好不要使用编译器自身扩展的属性 不要想着 任何函数都可以做成内联函数





内联函数相对于普通函数 就是省去了函数调用的压栈 跳转 和 返回的开销 --- 当一个函数里面 有三个for 5个switch ---这样 执行函数体的开销远大于压栈等的开销

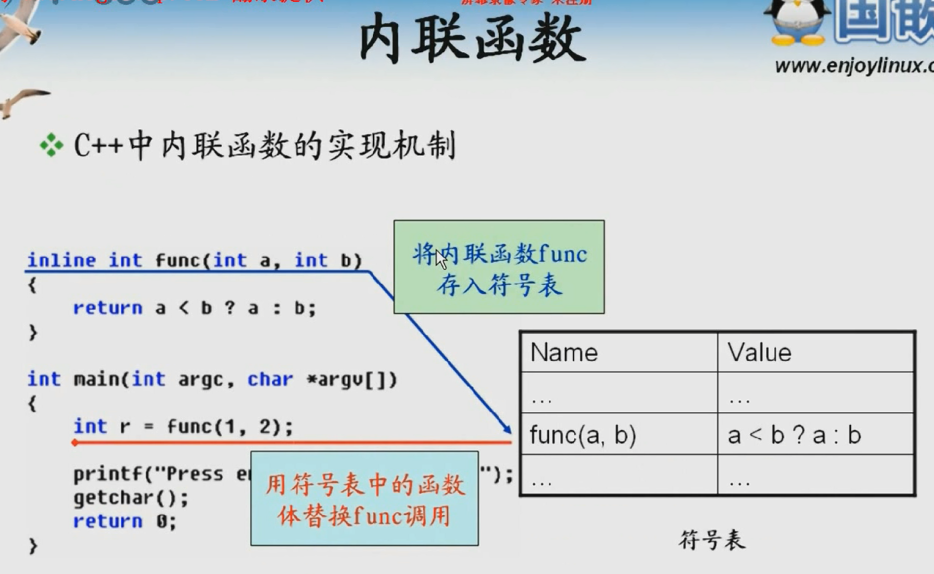
执行一个函数500ms 但是 压栈等等需要不到1ms 那么 此时变成内联函数 没有任何意义

所以 C++编译器有可能拒绝内联请求 --- 可以智能判断要不要内联

有时候 就想把500行的代码变成内联函数 – 强制变成内联函数 但是没有意义

还有 就是内联函数 编译器也是 插入和替换 ---- 和预处理器的插入和替换有什么区别？C++编译器如何进行内联函数如何调用？

讲讲内联函数在编译器的实现机制



C++编译器 编译到func这个函数的时候 有内联声明 我们的程序请求C++编译器 要把func变成内联函数 ----- 发现符合做成内联函数 此时 C++编译器就把这个内联函数放到了符号表中 ---- 函数名是Key 函数体因为比较小 就变成了value

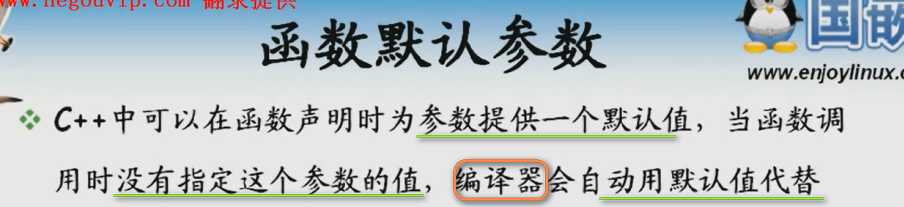
------ 当编译到int r=func(1,2)的时候 就发现这个func是在我的符号表中可以找到 ---- 编译器就做了先对这个func两个参数 进行类型检查 但是预处理器绝对不会做这件事 绝对不管你的参数类型对不对 就是进行简单的文本替换 ---- 但是编译器不一样 对两个参数进行类型检查 发现匹配 这样编译器到符号表中拿出来这个func(a,b)的函数体 进行替换

---- 这个就是C++编译器级别对内联函数的处理

这个内联函数 和C语言中宏代码片段的区别

**符号表就是C++编译器在编译程序的过程中 生成的一张表 存放变量名的 这个是C++自己使用的内容 --- 所以符号表不会进入最终的可执行程序**

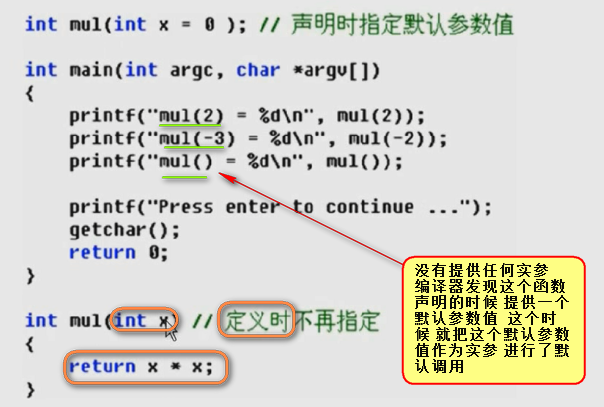
C++的函数默认参数

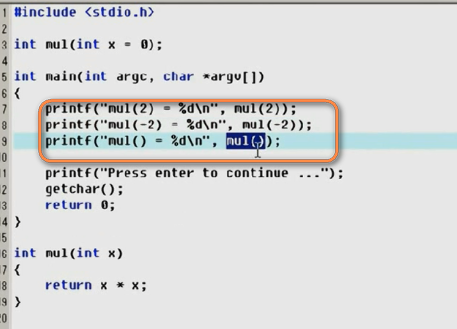


【php可以 但是 java不可以】



这种写法在C语言中是没有的

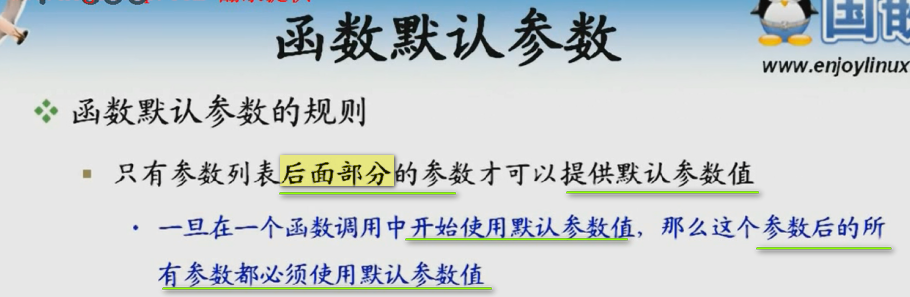


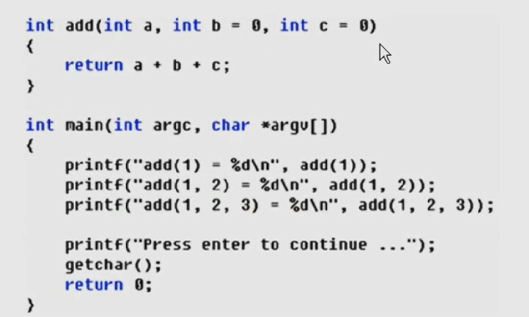


C++中 函数的默认参数是非常简单的东西

=====

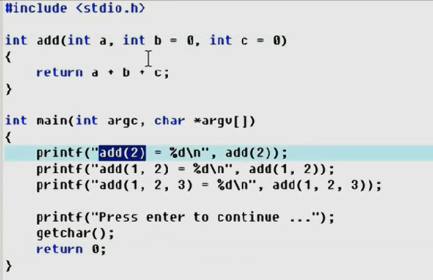
定义一个函数 不止一个参数 如果函数的参数过多的时候，默认的参数值有没有什么限制？



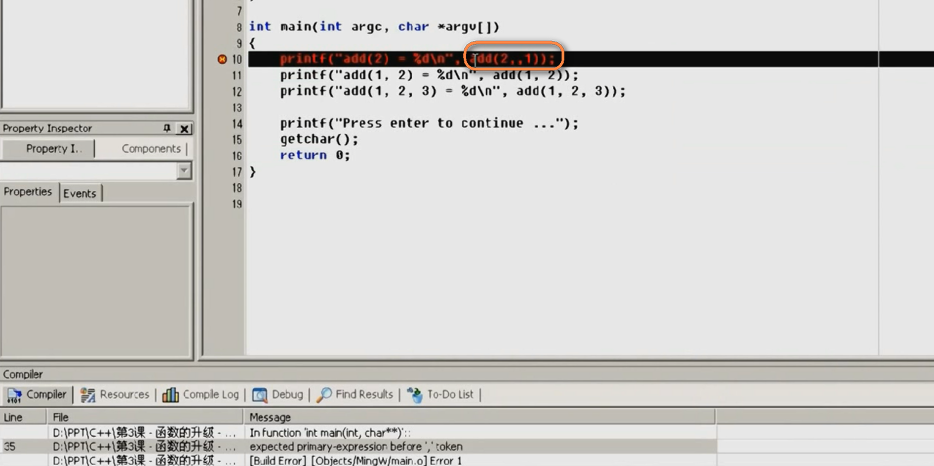


以b作为起点 提供默认参数值 那么 后面都要提供默认参数值

如果以a为起点 提供默认参数值 那么后面都要提供默认参数值



add(a) 意味着从b开始默认参数值 那么b之后 也必须是默认参数值



所以 当定义了一个函数 从某一个参数开始 提供了默认的参数值 在函数调用的时候 一旦某个参数使用了默认参数值 那么这个参数后面所有的参数都应该使用默认参数值

 从c开始使用默认参数值 那么c之后的参数也要使用默认参数值

**只需要记住这个规则就好**

======================

C++函数占位参数

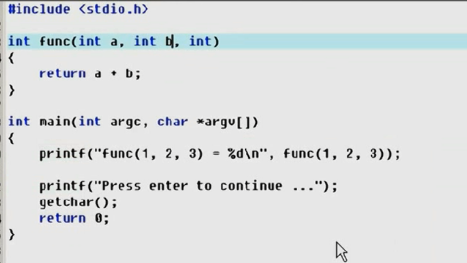
----- 可以给函数提供占位参数

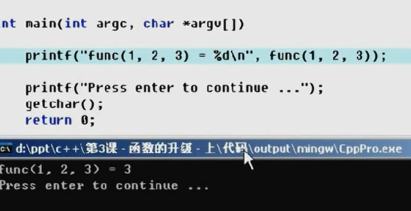


----- 因为没有名字 所以 函数体内部无法使用这个占位参数

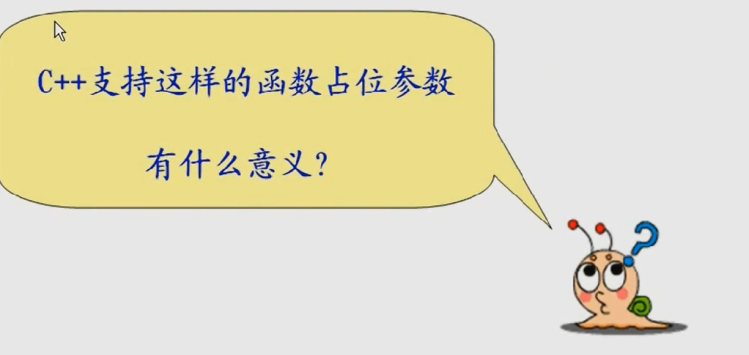


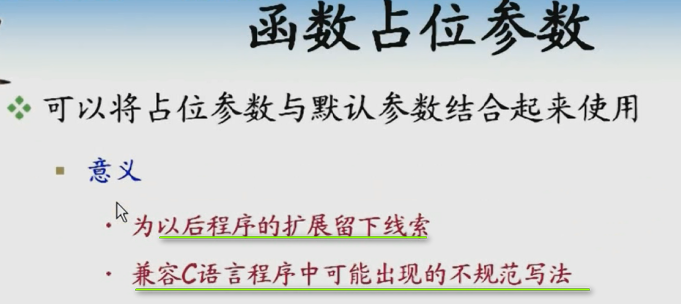
声明的时候 最后一个参数没有名字 ------ 把 这样的参数叫做函数占位参数 本质上也是一个参数 所以 调用的时候 必须提供第三个参数

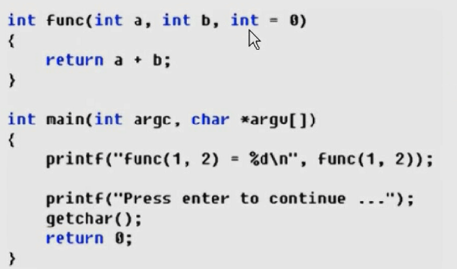




这个3是实参 赋值给函数占位参数 ---- 但是没有名字 函数体内无法使用



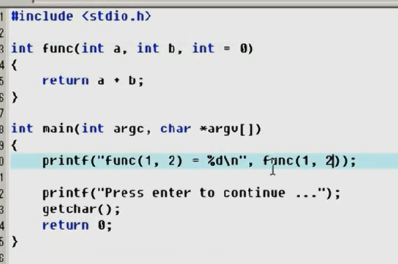




占位参数指定了一个默认值 这个默认值就是0 这样 只提供a b

**\*\***这样在我们的工程中 看到了 就可以明白 这样写是为了在扩展程序的时候 有一个线索

看到这样的写法 原来的作者告诉维护者 这个函数是非常有可能和扩展的 但是 目前是够用了 如果以后真的改变 那么例子中的第三个参数 就可能会变成有名字的普通参数

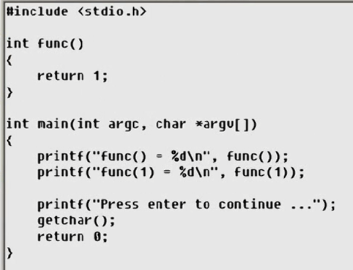


【占位就是 觉得可能有用 但是 还不清楚怎么实现 所以 先占一个位置 为后面做扩展】

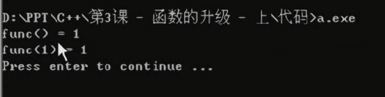
\*\*另一个目的 兼容C语言中的不规范的写法

占位参数+默认参数结合起来 就是为了解决C语言中的不规范写法

构建一个c程序

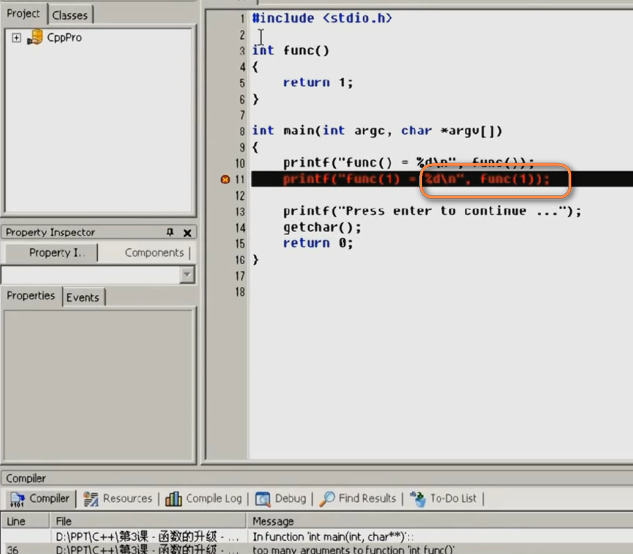
 【之前讲过这两种都是合法的 c中() 就可以是任意多个参数】



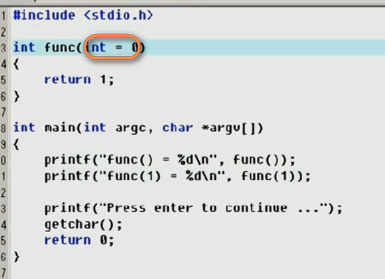


虽然代码中 没有给func这个函数给任何参数 但是c这个参数 没有参数定义 就可以以任意的参数进行调用

但是 放到C++中 无法通过



在C++中 如果出现了这样的错误 为了兼容 仅仅修改函数定义的地方 添加一个函数占位参数



这样程序编译通过了

Java中的内联函数

<http://blog.csdn.net/rflyee/article/details/51067301>

<https://www.2cto.com/kf/201211/165952.html>

