**研究14、15研讨会报告**

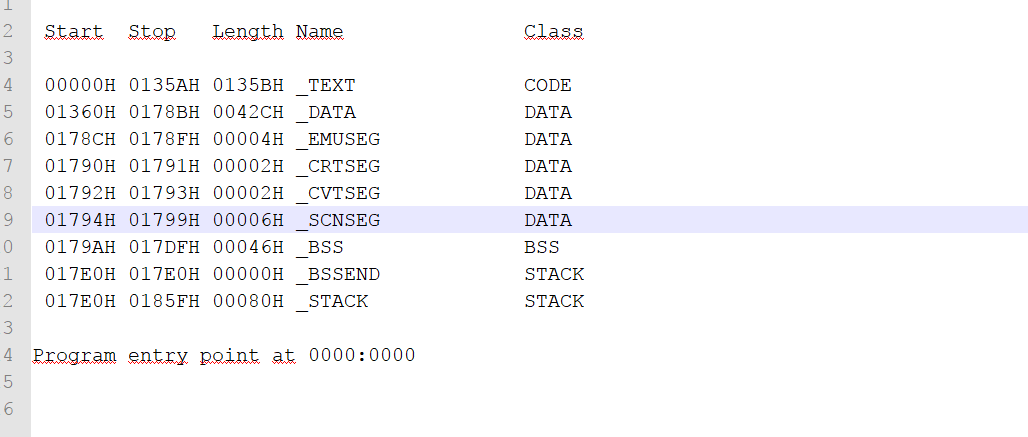
1. \*argv[]：argv是什么类型？

sizeof(argv)=2，其实是对一个指向指针的指针求sizeof的结果。

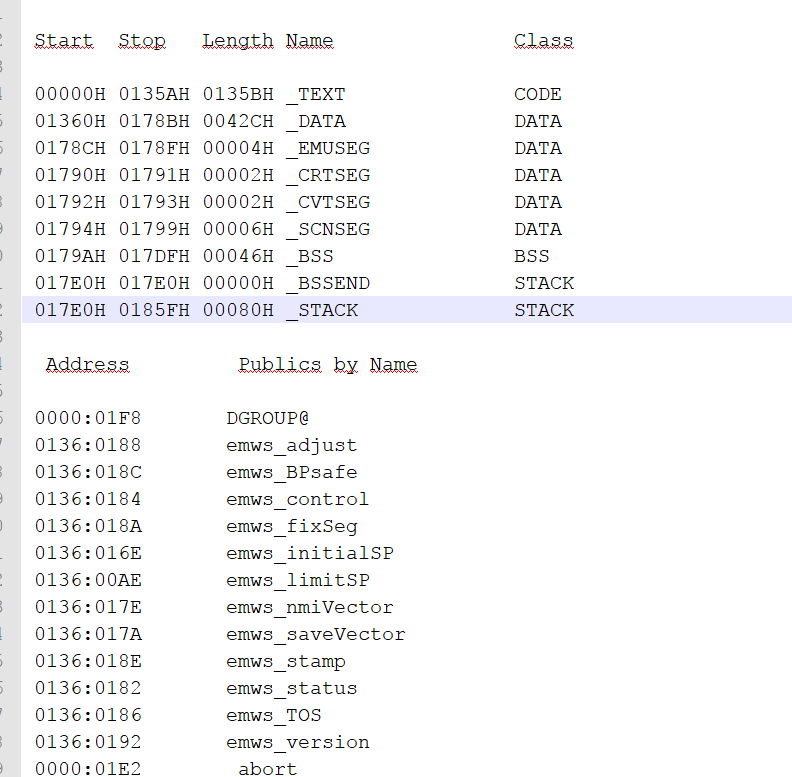
1. 生成各个被调符号的map文件，了解map文件的作用；

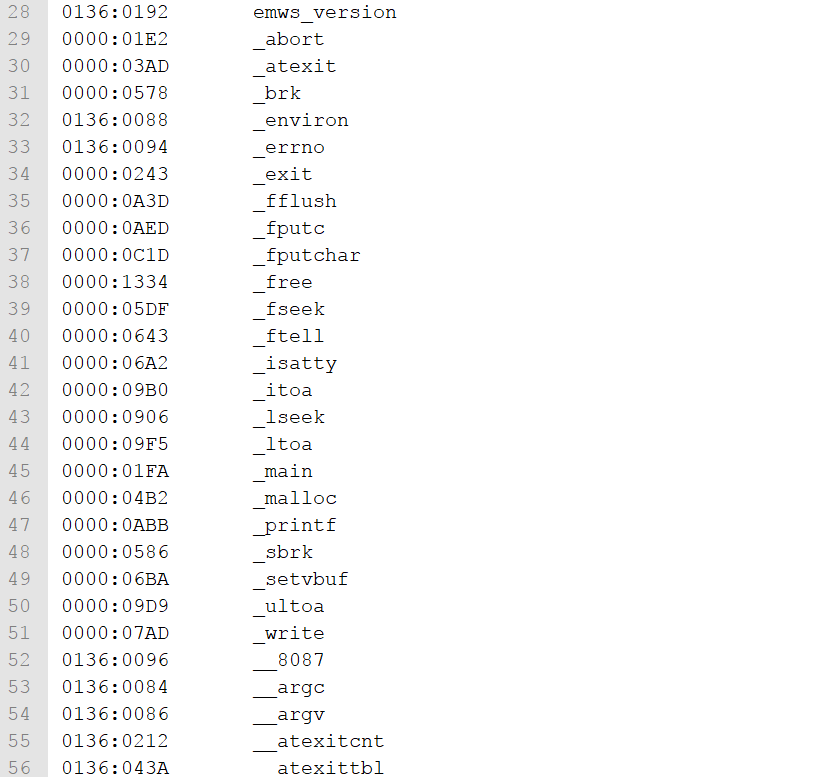
map三种生成格式segment、detailed、publics可以看看

Segment版map:

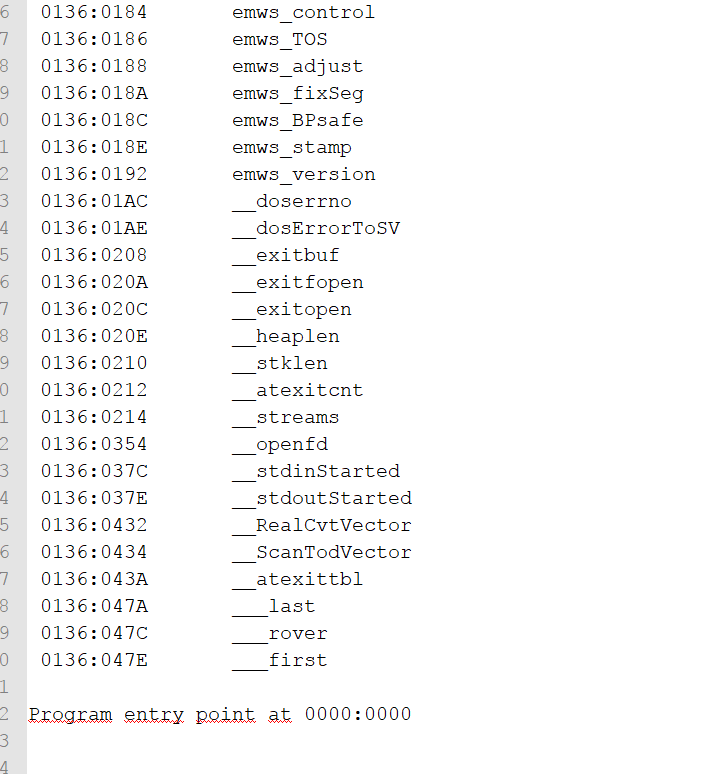


Publics版map:

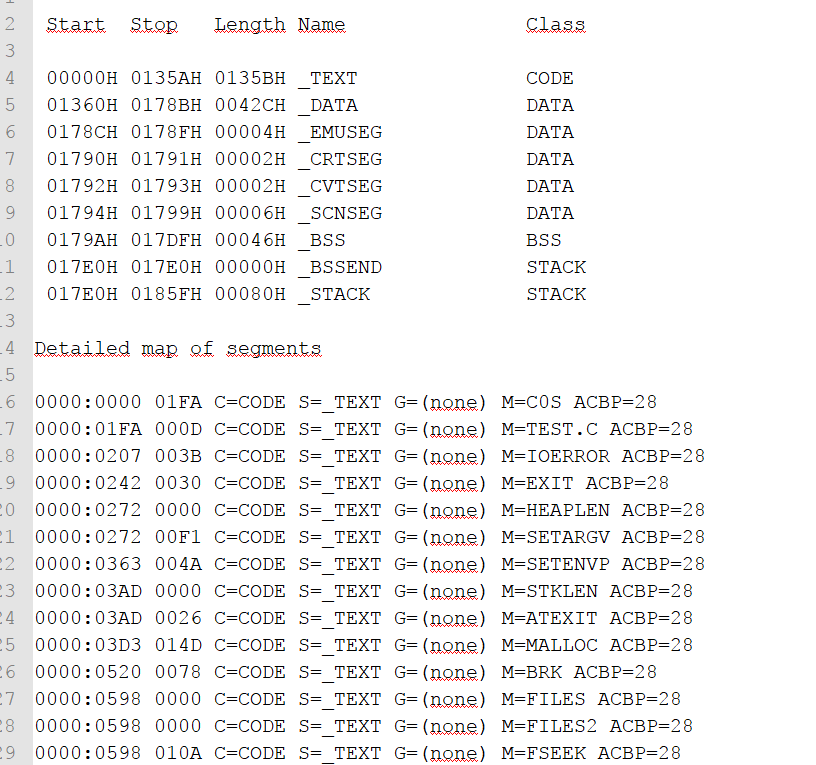




等等。。。。。。



detailed版map:



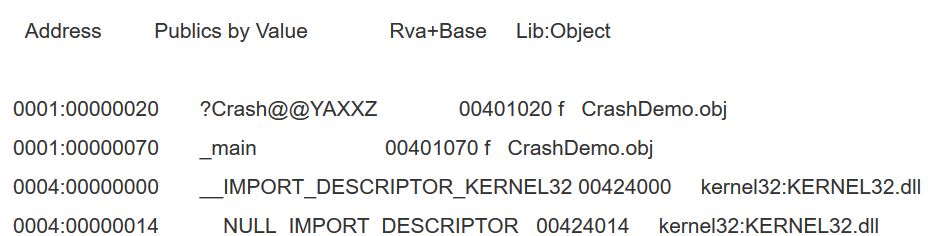
这个更大。

三个版本，内容依次在兼容前版本的基础上增多。

map文件的作用是，我们可以根据程序产生错误地址在map文件中找到相应的函数、变量地址。

比如程序有“除0错误”，在 Debug 方式下编译的话，运行时肯定会产生“非法操作”。好，让我们运行它，果然，“非法操作”对话框出现了，这时我们点击“详细信息”按钮，记录下产生崩溃的地址。比如：0x0040104a

再看看它的 MAP 文件：



仔细浏览 Rva+Base 这列，会发现第一个比崩溃地址 0x0040104a 大的函数地址是 0x00401070 ，所以说在 0x00401070 这个地址之前的那个入口就是产生崩溃的函数，也就是这行：  
  
0001:00000020  ?Crash@@YAXXZ   **00401020** f   CrashDemo.obj

发生崩溃的函数就是 ?Crash@@YAXXZ ，也就是 Crash() 这个子函数。

不仅如此，根据\*.map文件还可以找到具体的行号。不过得先生成代码行信息，类似这样：

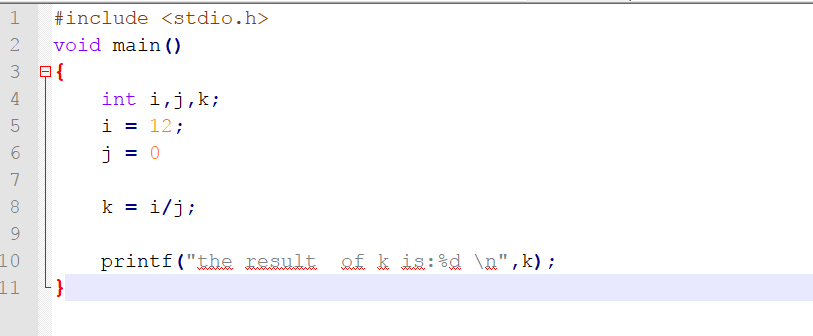


13 0001:00000020

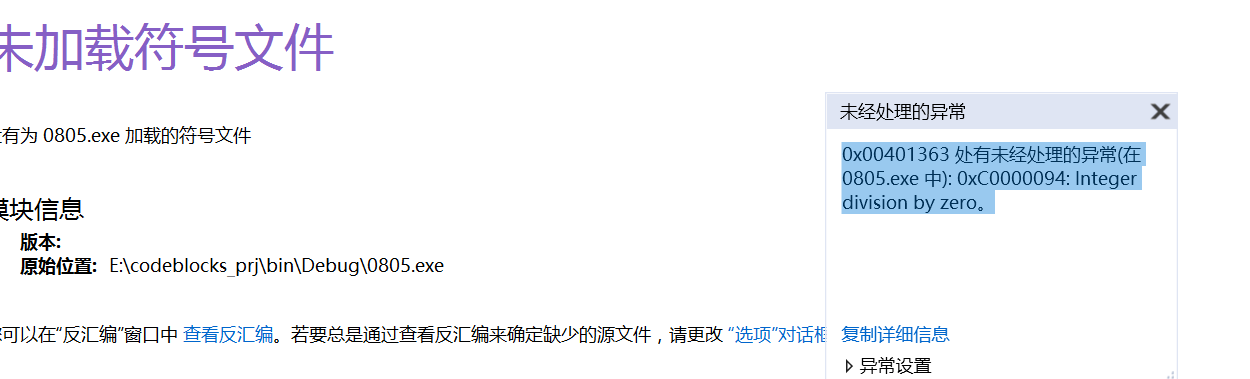
第一个数字代表在源代码中的代码行号，第二个数是该代码行在所属的代码段中的偏移量。  
  
如果要查找代码行号，需要使用下面的公式做一些十六进制的减法运算：  
  
崩溃行偏移 = 崩溃地址（Crash Address）-基地址（ImageBase Address） - 0x1000   
  
为什么要这样做呢？细心的朋友可能会留意到 Rva+Base 这栏了，我们得到的崩溃地址都是由 偏移地址（Rva）+ 基地址（Base） 得来的，所以在计算行号的时候要把基地址减去，一般情况下，基地址的值是 0x00400000 。另外，由于一般的 PE 文件的代码段都是从 0x1000 偏移开始的，所以也必须减去 0x1000 。  
  
好了，明白了这点，我们就可以来进行小学减法计算了：  
  
崩溃行偏移 = 0x0040104a - 0x00400000 - 0x1000 = 0x4a   
  
如果浏览 MAP 文件的代码行信息，会看到不超过计算结果，但却最接近的数是 CrashDemo.cpp 文件中的：  
  
16 0001:00000046   
  
这里又和之前查找出问题函数的思想一致了，找第一个比他大的数的之前那组数据的行号，也就是在源代码中的第 16 行

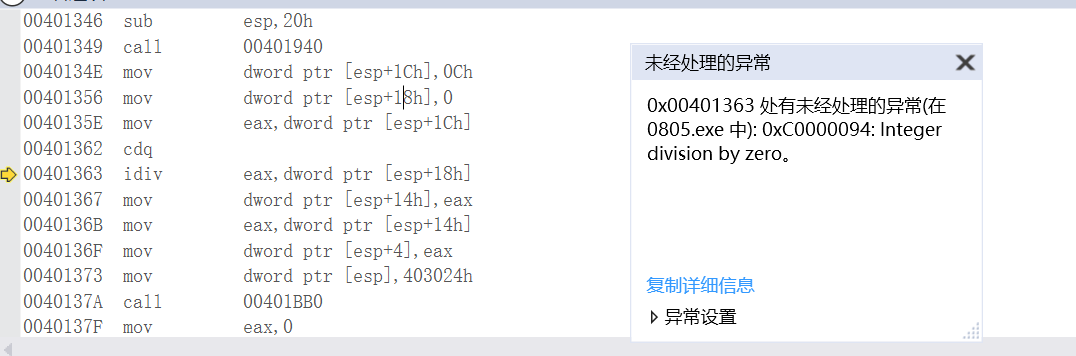
好吧，我想亲自操作一下，巩固印象。编一个故意的问题程序：

Test.c:



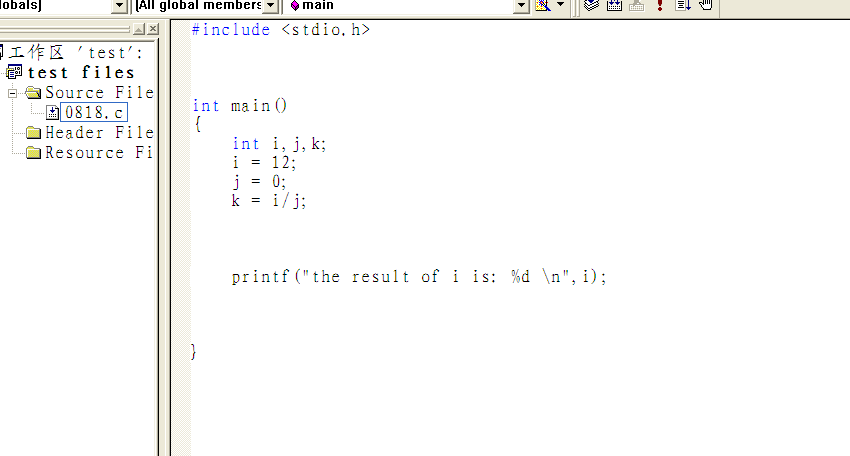
可以看到第8行中，除数j是0，这里故意出错，看我们可不可以找到。



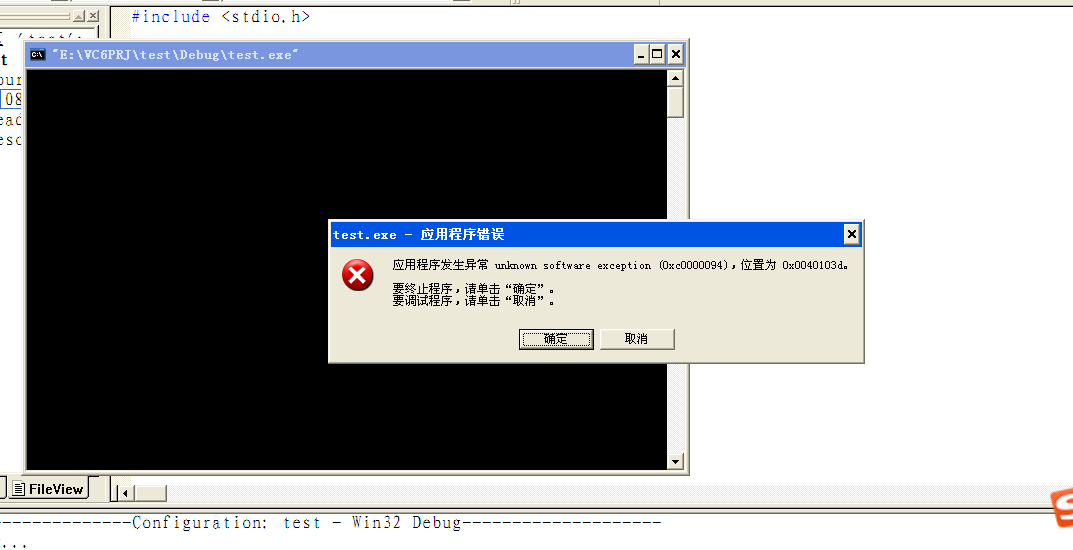


这里反汇编也可以看到出错的位置。

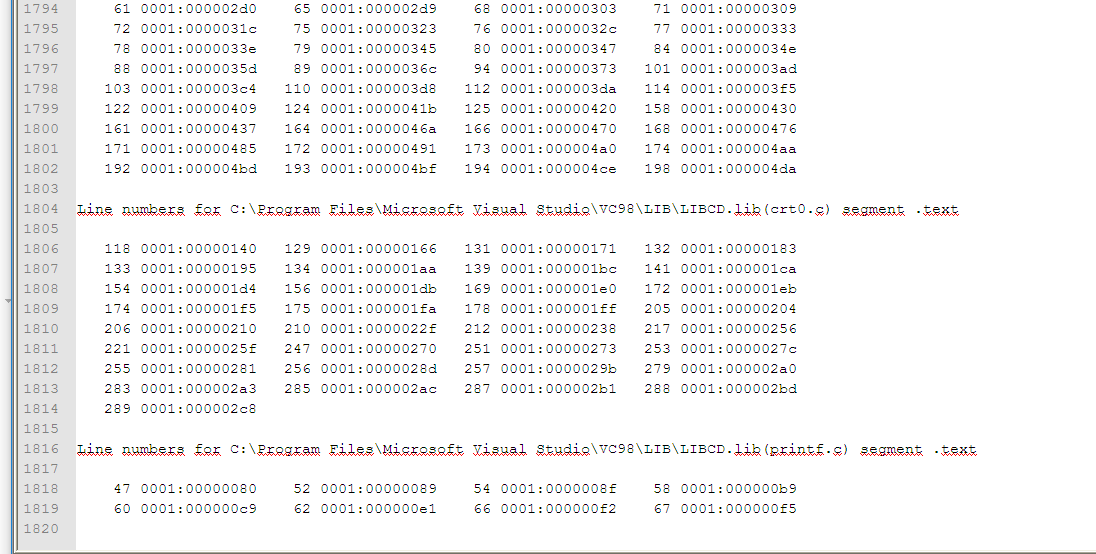
但是，听说高版本的vs生成的map文件没有行信息，我决定用一下虚拟机里的VC++6.0:



这个程序j为0，作了除数。运行奔溃：

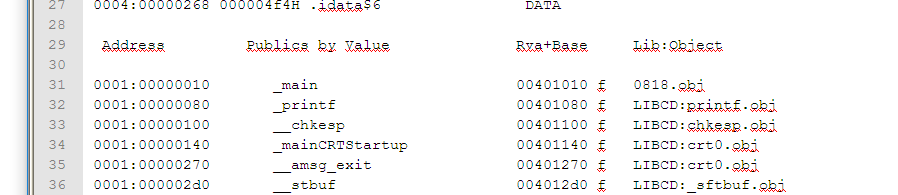


错误位置是0x0040103d



可以看到生成的map文件是多么巨大。

找一下第一个比0x0040103d大的数的之前一个数

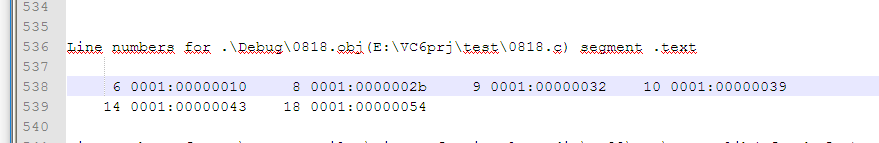


可以看到，是0x00401010,对应的函数是main函数。

计算崩溃行偏移= 0x0040103d-0x00400000-0x1000 =0x3d.

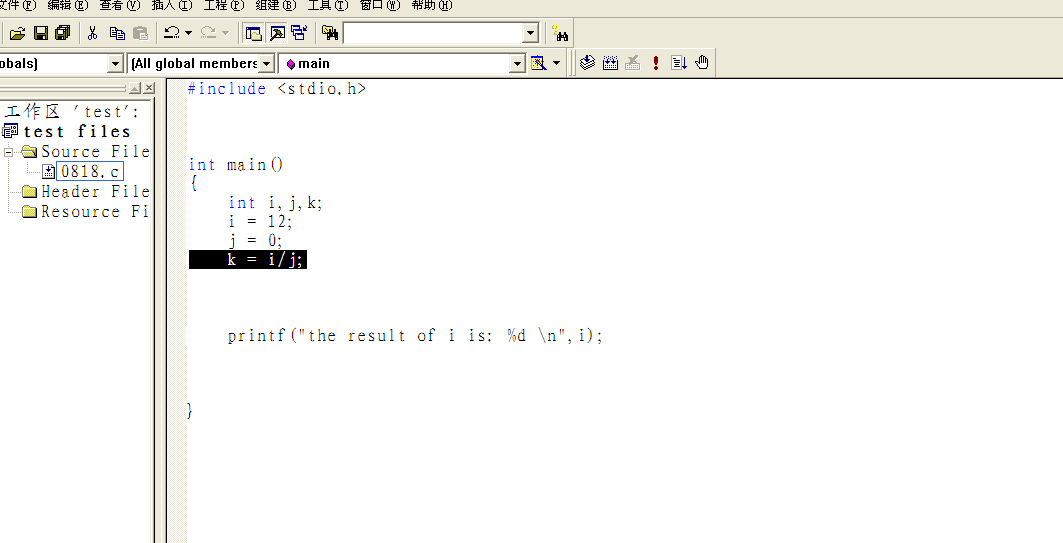
再根据行信息对应出行号。

同样找出第一个大于0x3d的数的前一个数：



这里，也就是0x39满足条件，所以应该是第10行。

Vc++6.0本身不带显示行号功能，但是，数一下，就可以知道黑色标出的“k=i/j;”就是第10行。可以，找到了问题，不过，高版本的vs生成的map文件会没有行信息了呢？



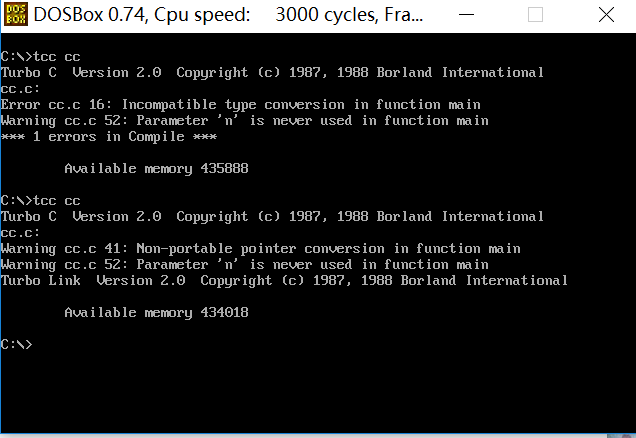
3、将双引号改为单引号的问题。

在原来自己编写的cc.c程序中进行修改：

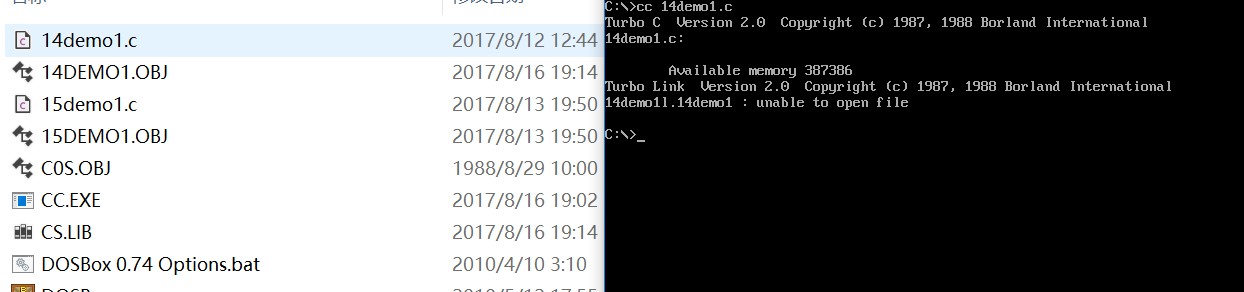
将原来的“ char c[5] = ","; ”改为“ char c = ','; ”；

在tcc编译链接的时候，提示“strcat(b,c);”语句有问题，没有错误，但是有警告：

Non-portable pointer conversion — 不可移动的指针（地址常数）转换



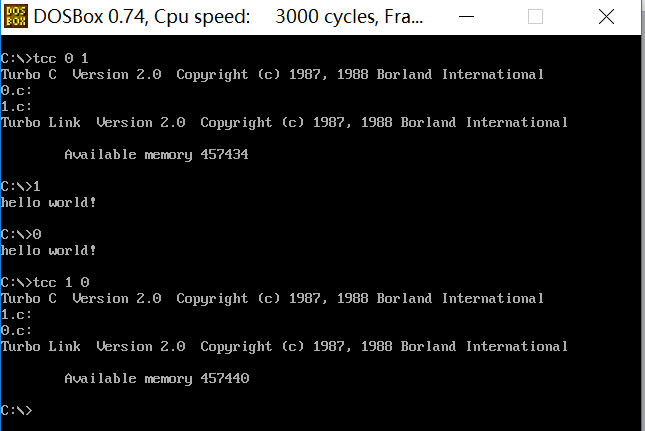
用重新生成的cc.exe替换掉原来cc.exe，看新生成的cc能否正常运用 ：

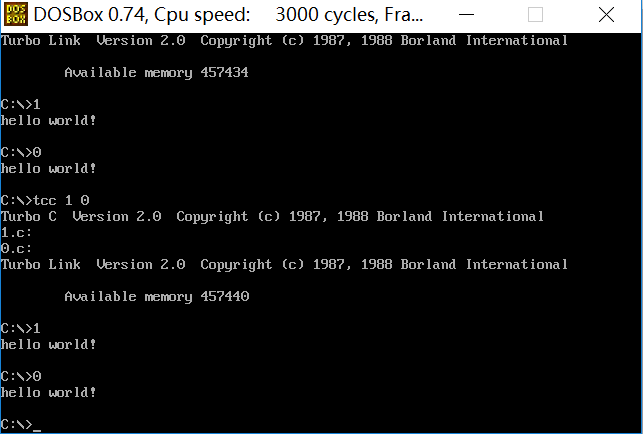


结果链接过程出现问题，无法生成对应的\*.exe文件。

4、Tcc 0.c 1.c在0.c中定义main函数调用f函数，在1.c中定义f。是否报错，如果不报错，为什么？（保证被调用文件在后面，避免声明）

运行结构不报错，可以正常运行：





1. c:

main()

{

f();

}

1. c:

f()

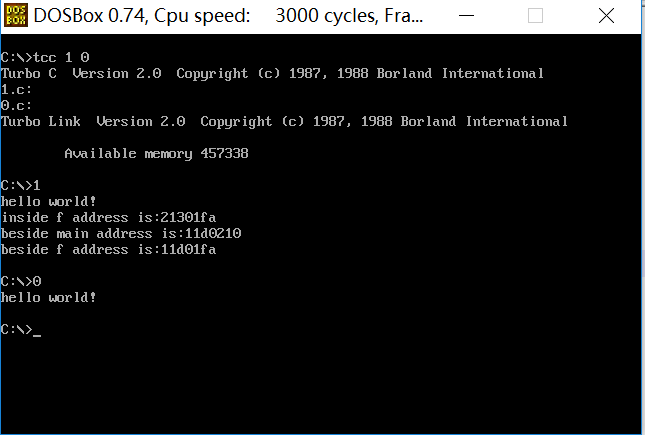
{

printf("hello world!\n");

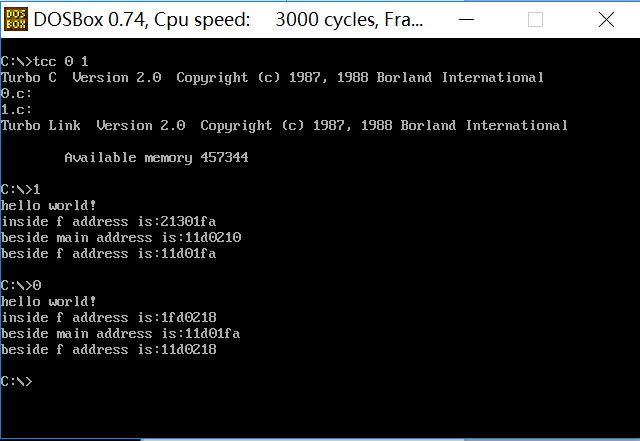
}

但是可以看到上面不论是“tcc 0 1”还是“tcc 1 0”,之后运行0或1都可以得到正确的程序执行结果。

加几条打印地址语句：

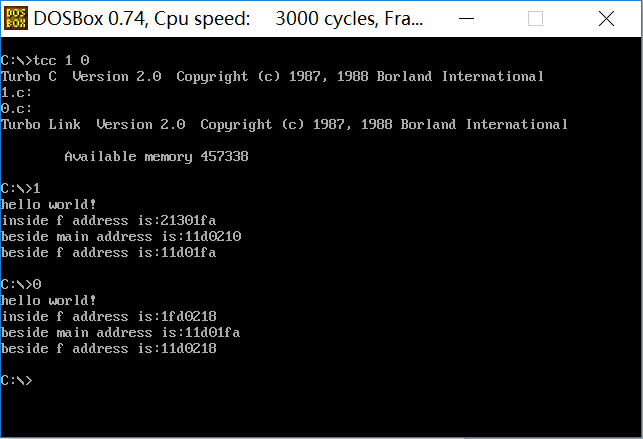


对比一下-------------------------------------------------------------------



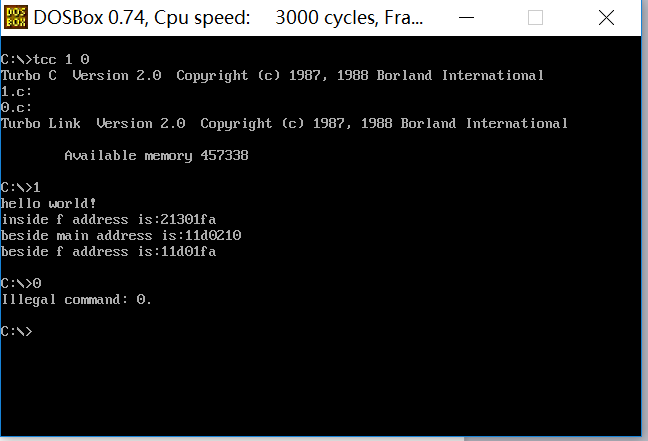
两者有很明显的区别。也就是tcc 0 1 的顺序不同导致了这种明显不同，而且，第二张图中可以看到地址发生的变化。

重新“tcc 1 0”确认一下第一张图出现的情况还会不会出现：

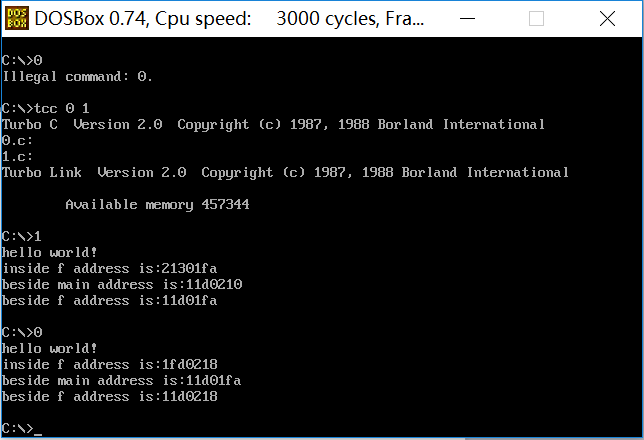


这里就奇怪了，怎么又出现了，时有时无的？

删掉生成的0.exe和1.exe，重新构建一下：



这是什么情况？



重复几次，还是这样。

这里研究乱了。是不是该对编译方面的知识了解的不多，找到一本《自己动手写编译器链接器的书》，看内容有些复杂，又要分析词法、语法方面的东西，但是听评价，有照搬tcc源码的嫌疑，有时间我还要再看看，跟着tcc源码学习一下。

5、Void (\*\*f)(int,int)

Void (\*f[4](int,int)={add，sub，div，mul};

为什么不对，为什么main的二参不会出错？

之前以为是函数们的大小不一，函数指针数组可以找到，指向函数指针的指针是不是不够智能，但是，后来自己写的先在Void (\*f1[4](int,int)={add，add，add，add};这样赋值，然后再Void (\*\*f)(int,int) = f1;最后该咋运行还是咋运行。而且把数组赋给指针的时候，会发生指针退化，指针变量只会包含数组的地址信息，而对数组的长度一无所知。所以我觉得可以排除了。这里一看初始化变量只是指向函数指针的指针，但是他的那些函数指针们呢？是不存在的。在那里，和数组不一样，是直接拷贝到栈里，函数名字也是地址。