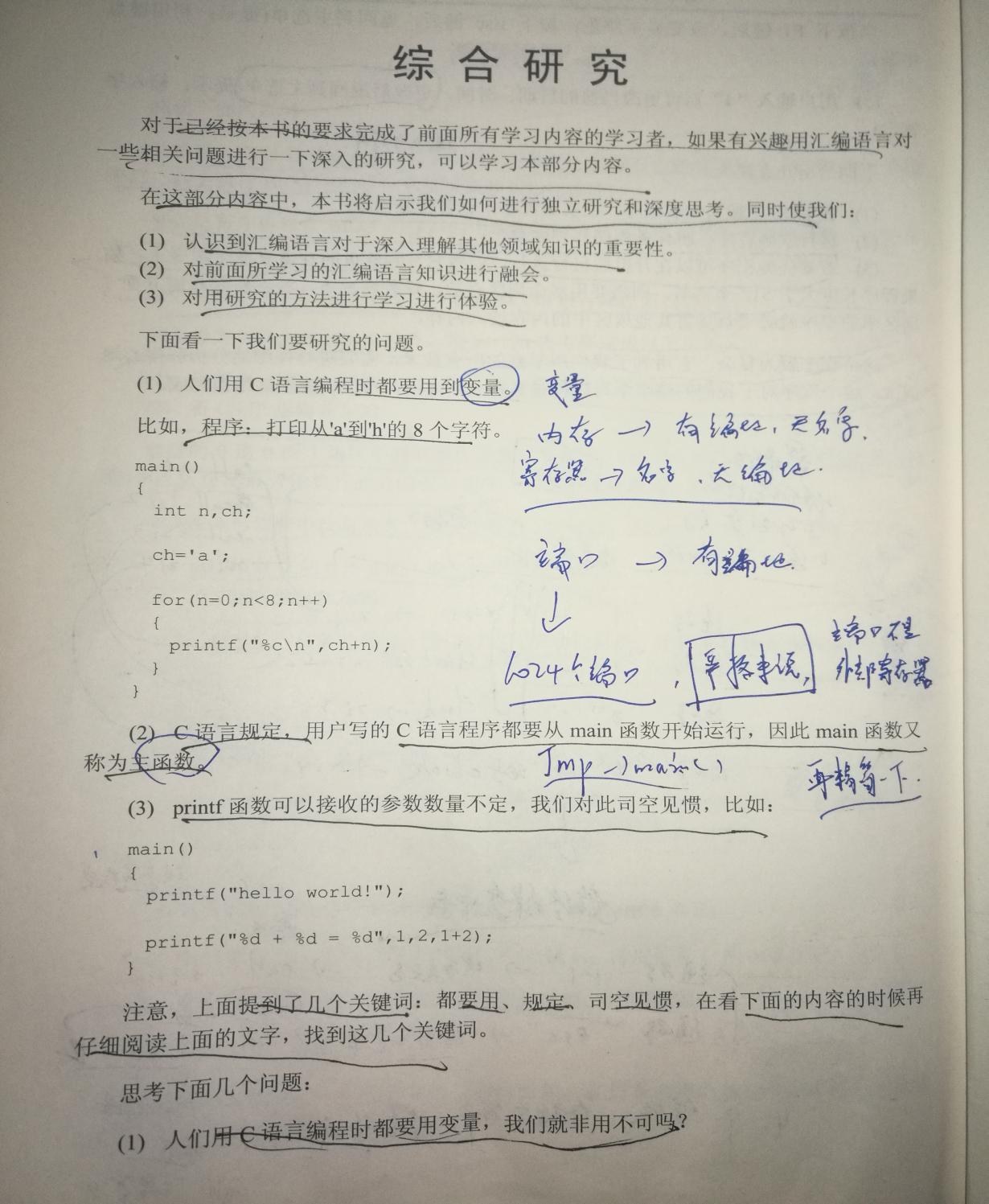
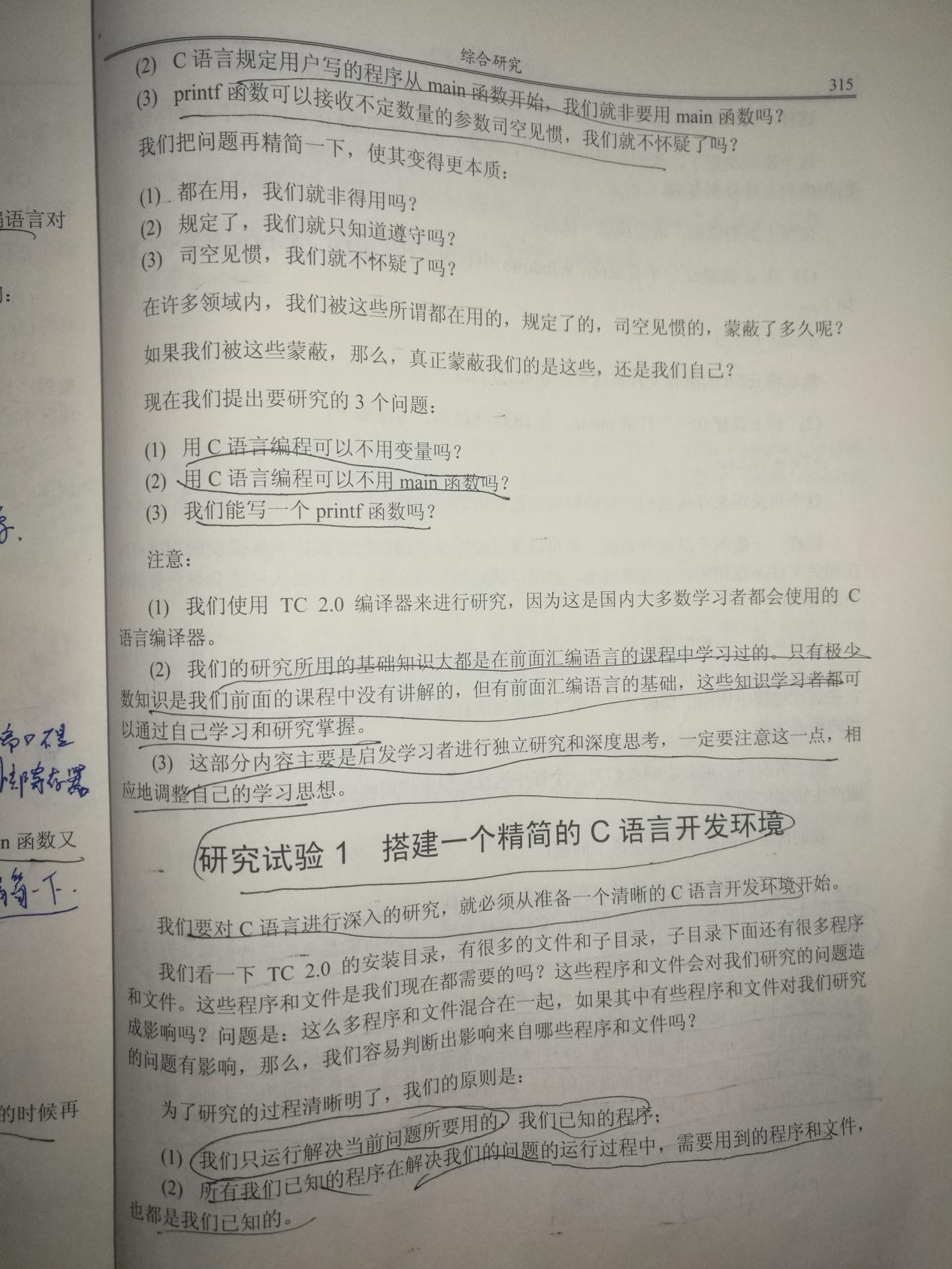
**第一次宣讲会综合研究报告**

1. **研究过程展示**

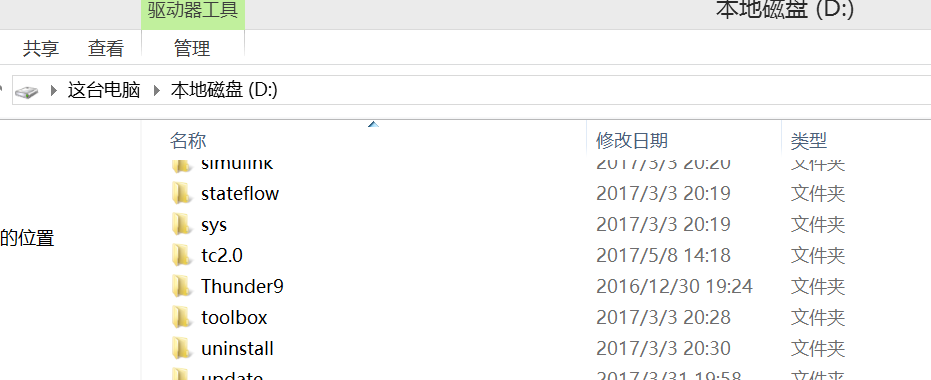
**综合研究前言：**

本次前言主要是读书思考的过程

****

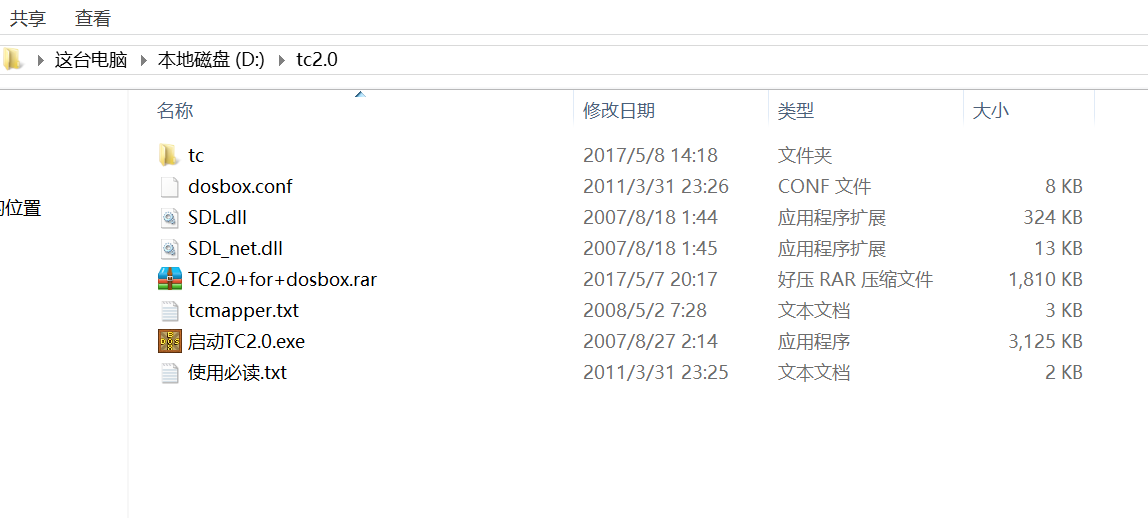
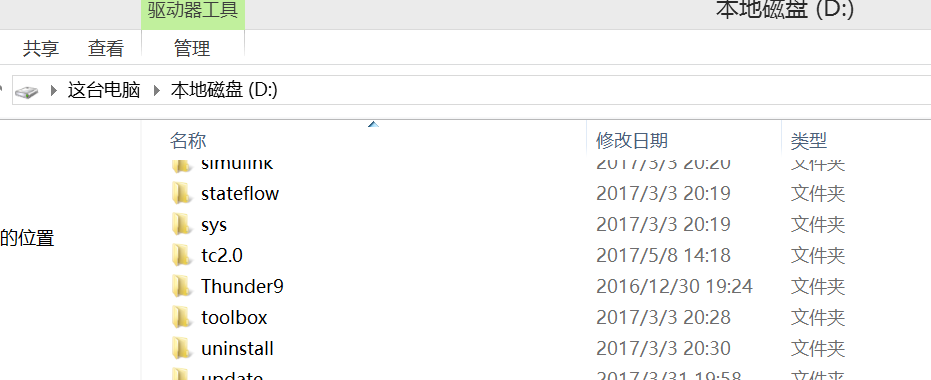
****

**综合研究一：**

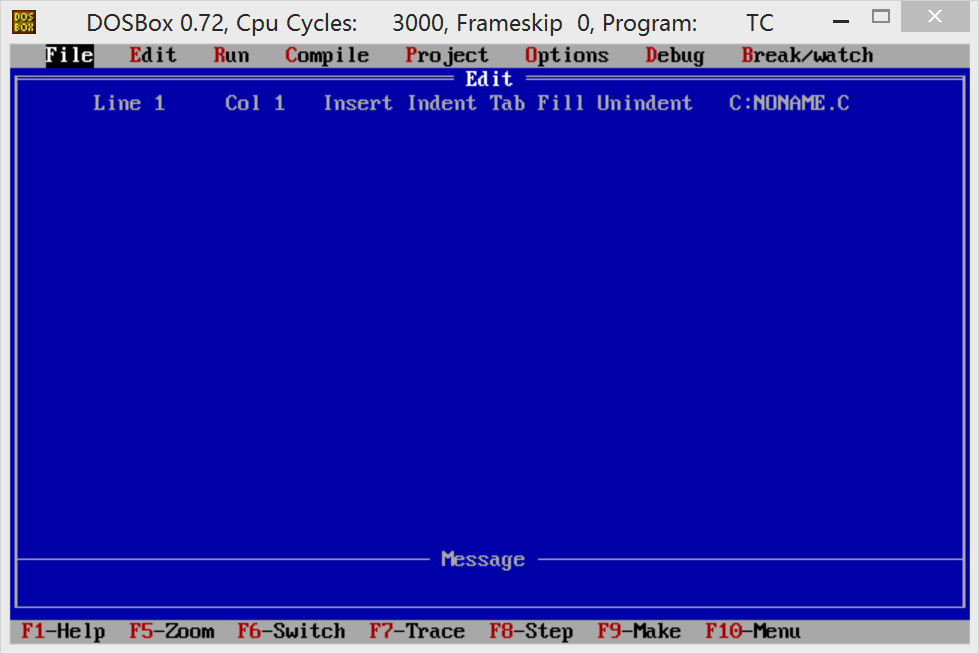
****

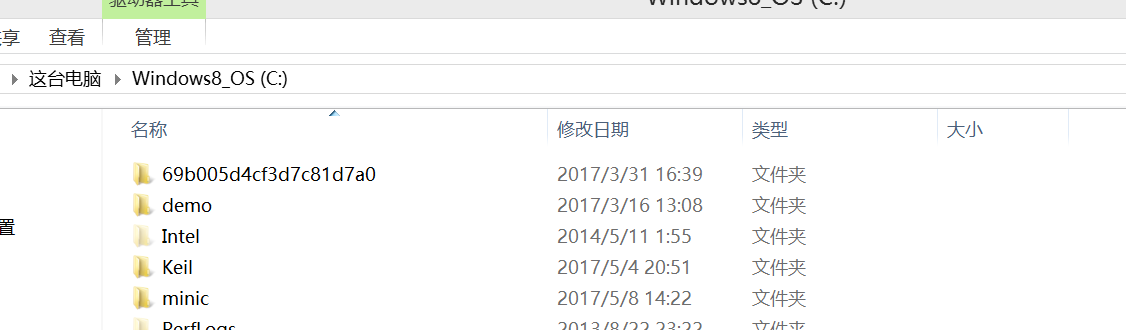
**如图在D盘新建：D:\tc2.0**

双击打开如下图：

****

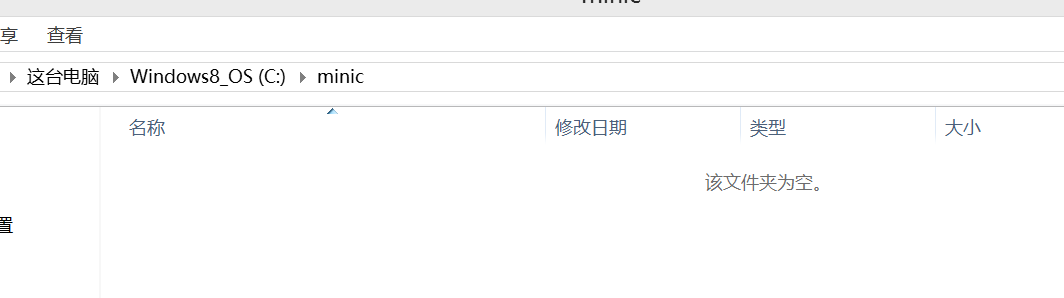
双击打开“启动TC2.0.exe”效果如下图：

****

****

**如图在C盘新建：C:\minic**

双击打开如下图：

****

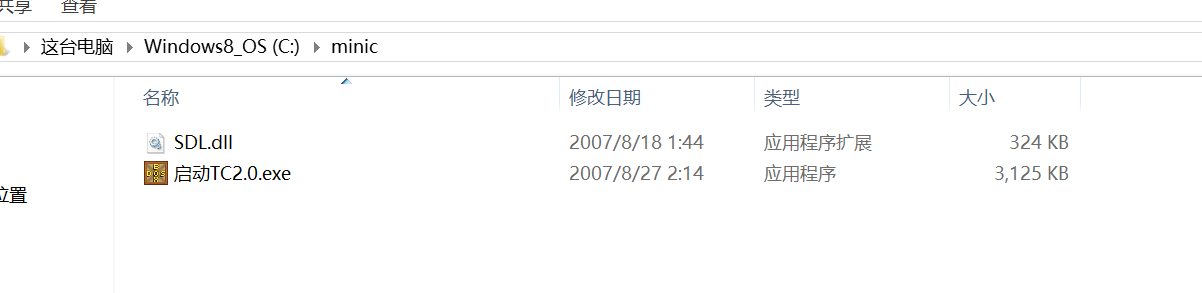
依次拷贝文件效果如下图：

****

双击之后：

****

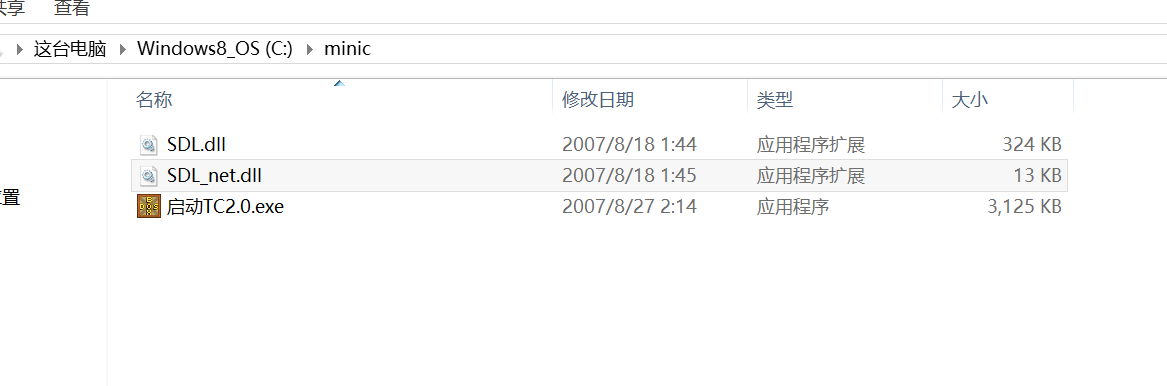
按要求拷贝**D:\tc2.0\SDL.dll**文件：



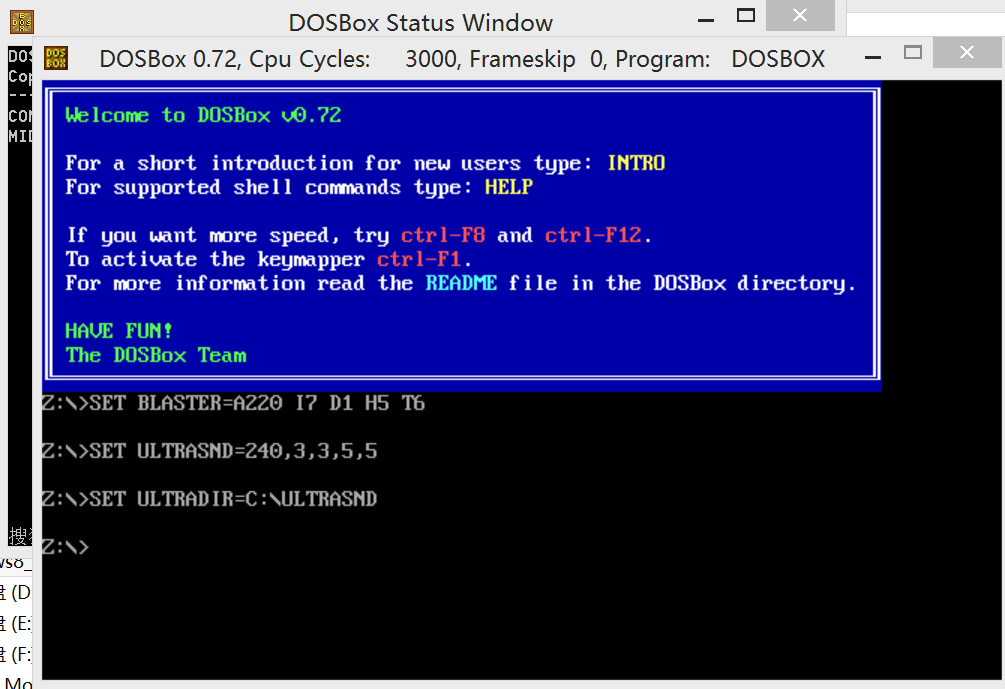
再次双击之后：

****

按要求拷贝**D:\tc2.0\SDL\_net.dll**文件：

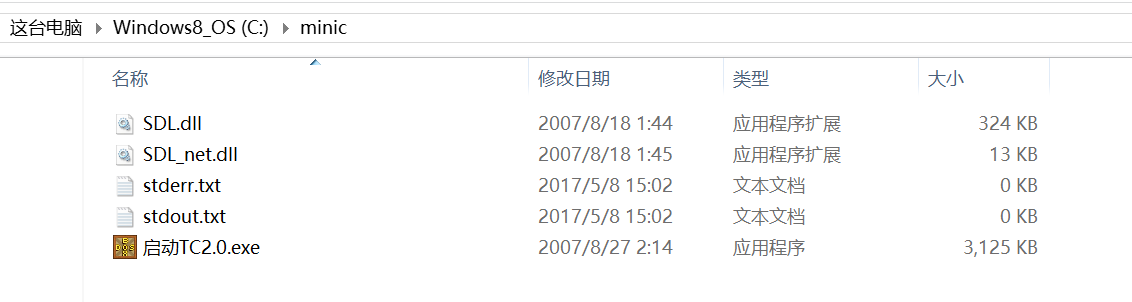
****

第三次双击之后：

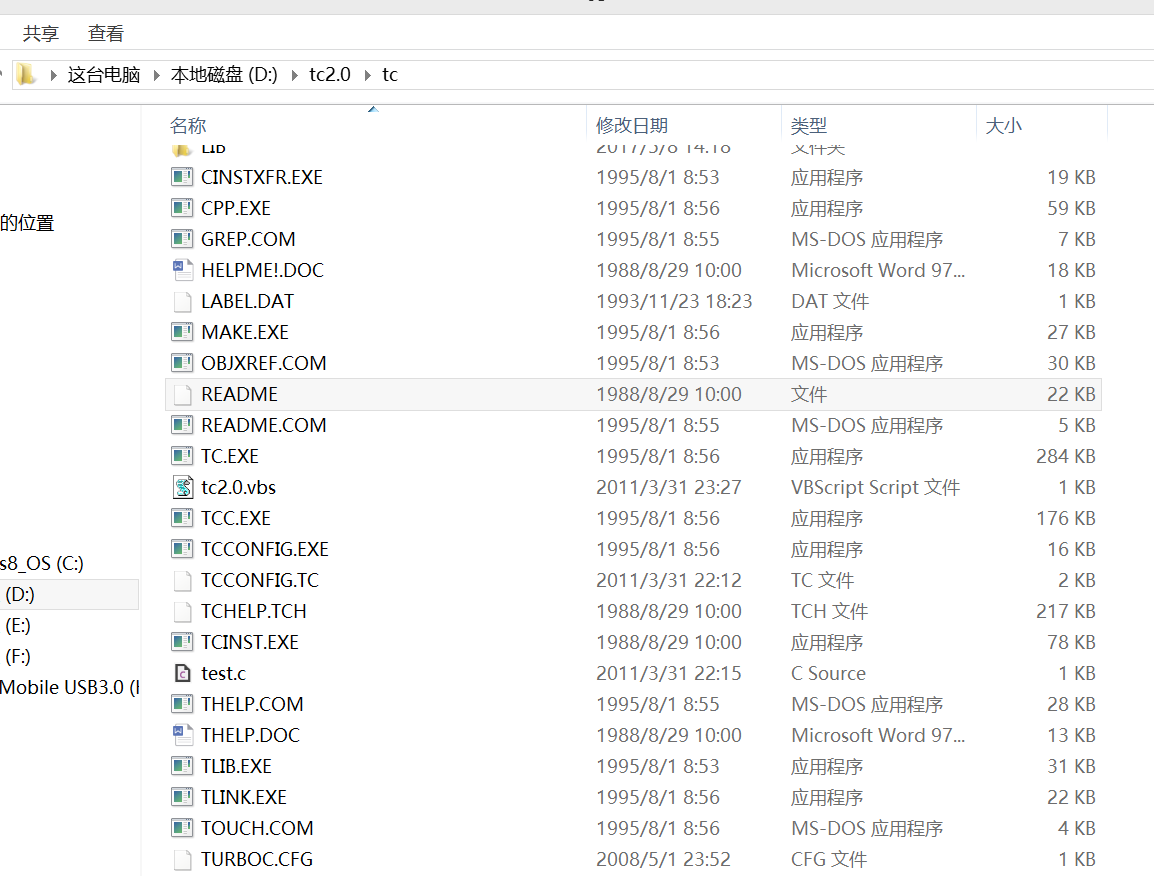


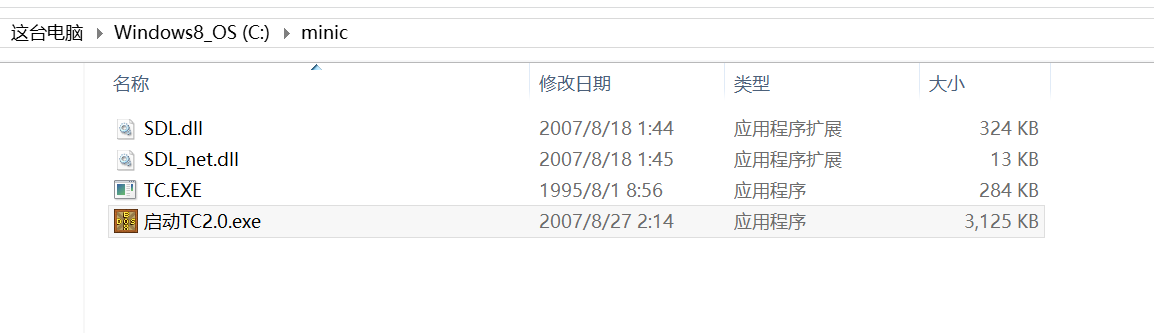
出现dosbox启动界面，不过有一些配置参数如图中横线。

并且在“未关闭状态”，此时的目录下文件情况如下图，可以看到，明显多出两个文本文档“stderr.txt、stdout.txt”；而当我关闭时，这两个文本文档又不在了。

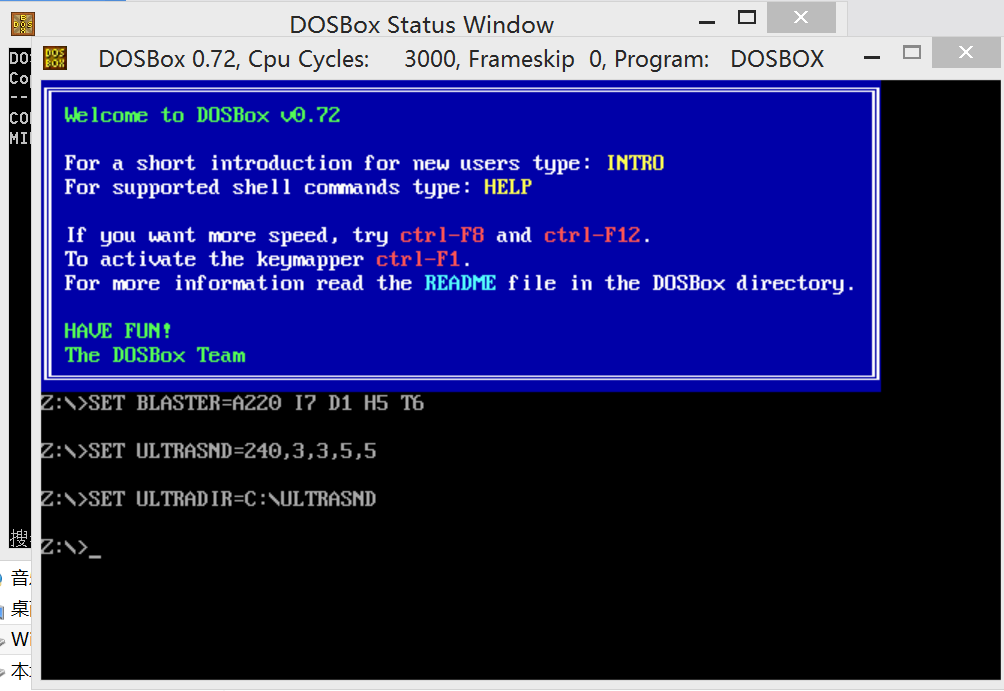


拷贝**D:\tc2.0\tc\TC.EXE**文件：





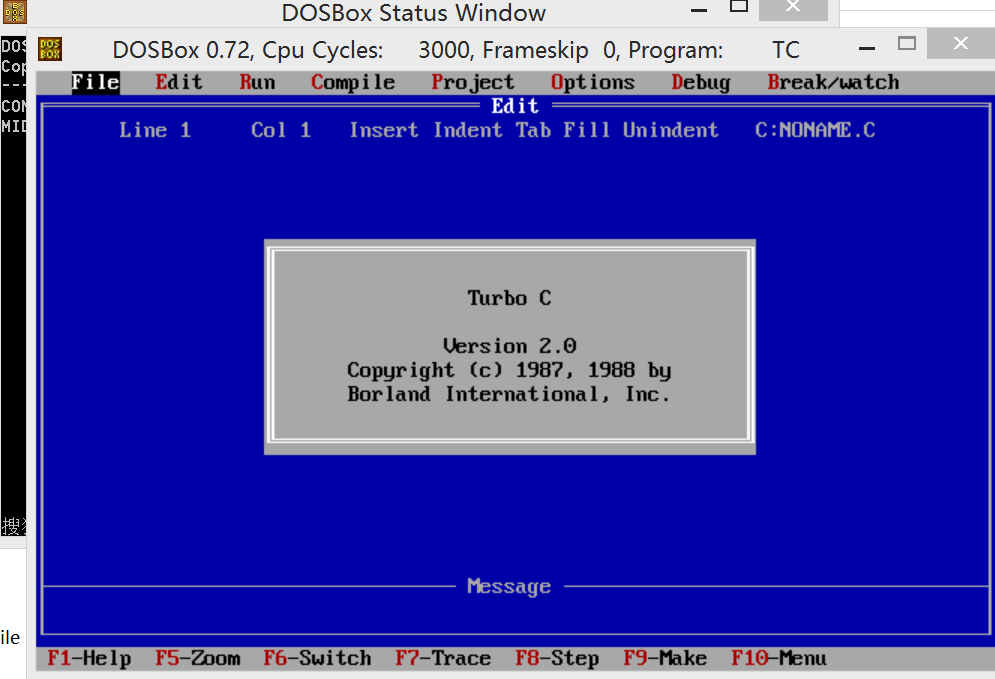
第四次双击之后：



拷贝**D:\tc2.0\dosbox.conf**文件：

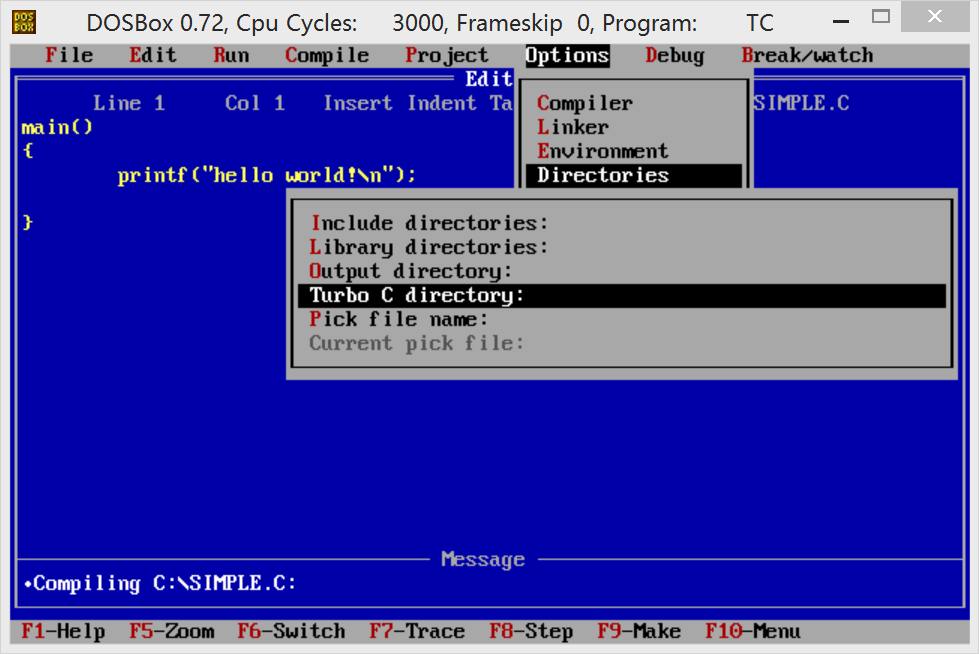
****

第五次双击之后：

****

效果至此，看起来好像可以用了。

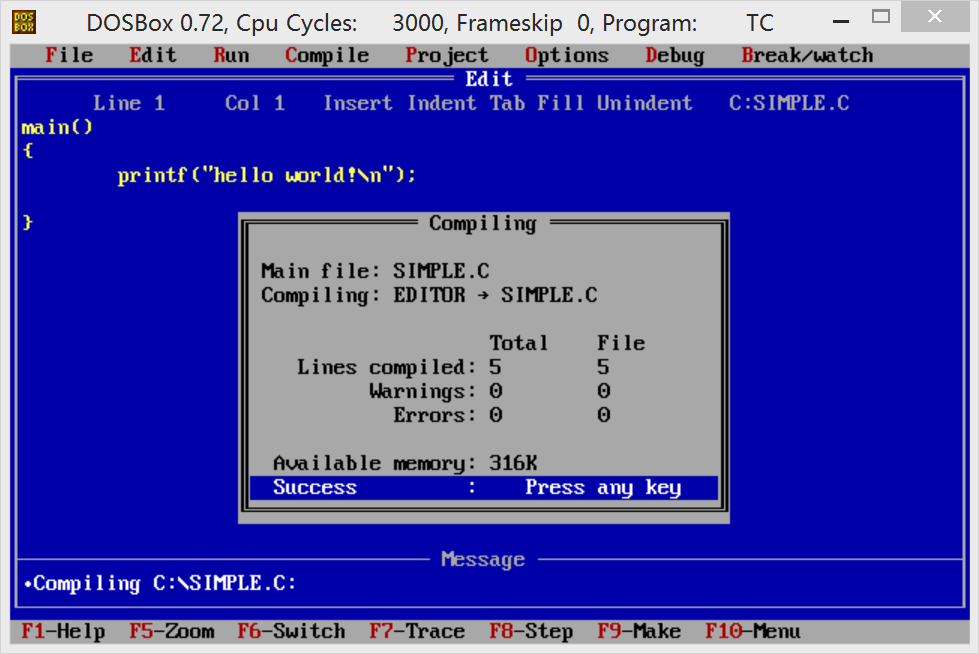
按照书上步骤做到第4步：

****

全部清除

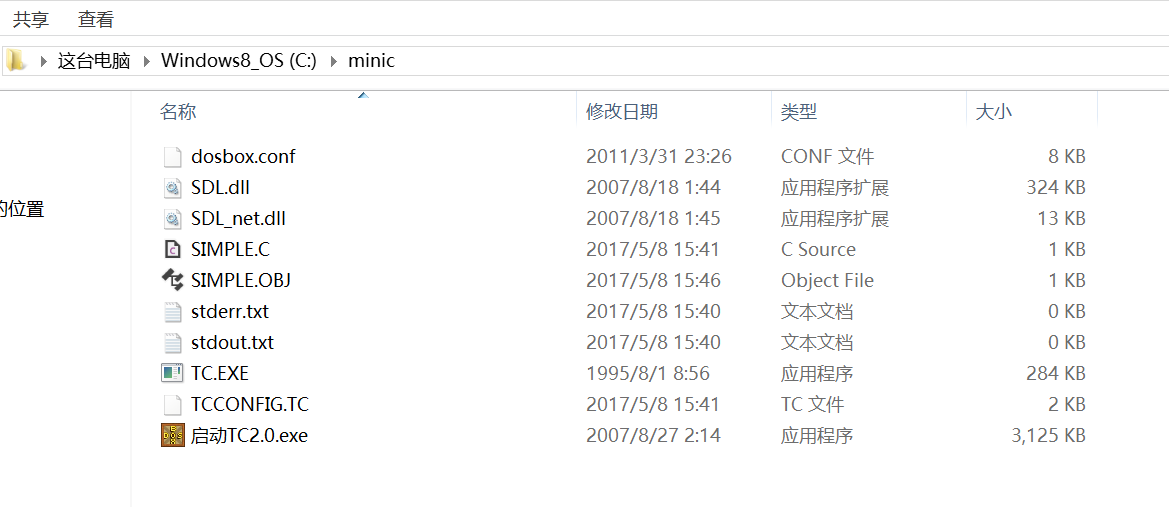
按照书上步骤做到第6步：

****

****

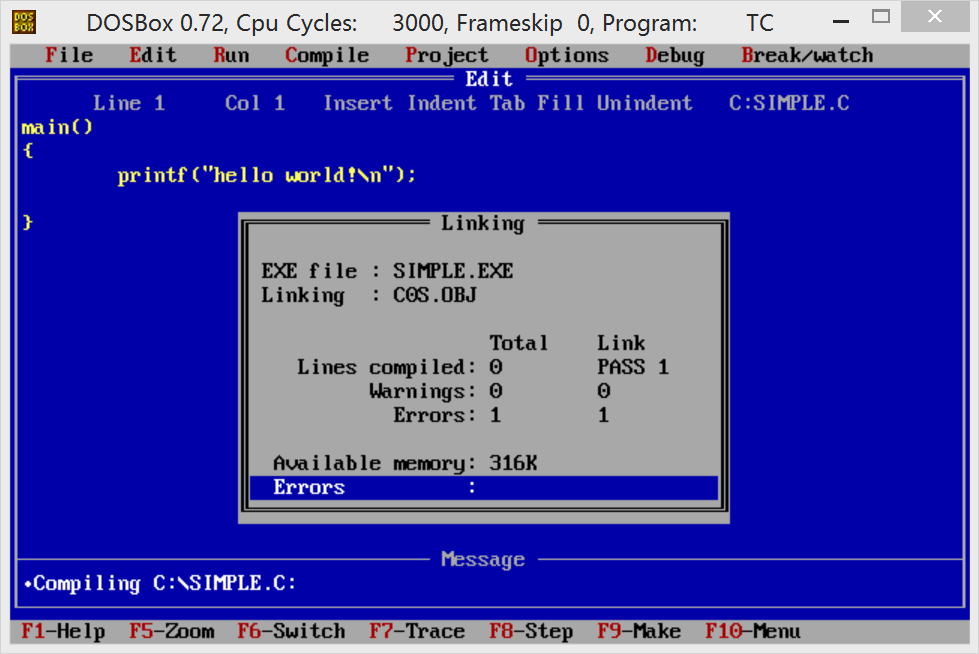
出现“Success”，应该是“编译成功OBJ文件”

再次检查**C:\minic** :

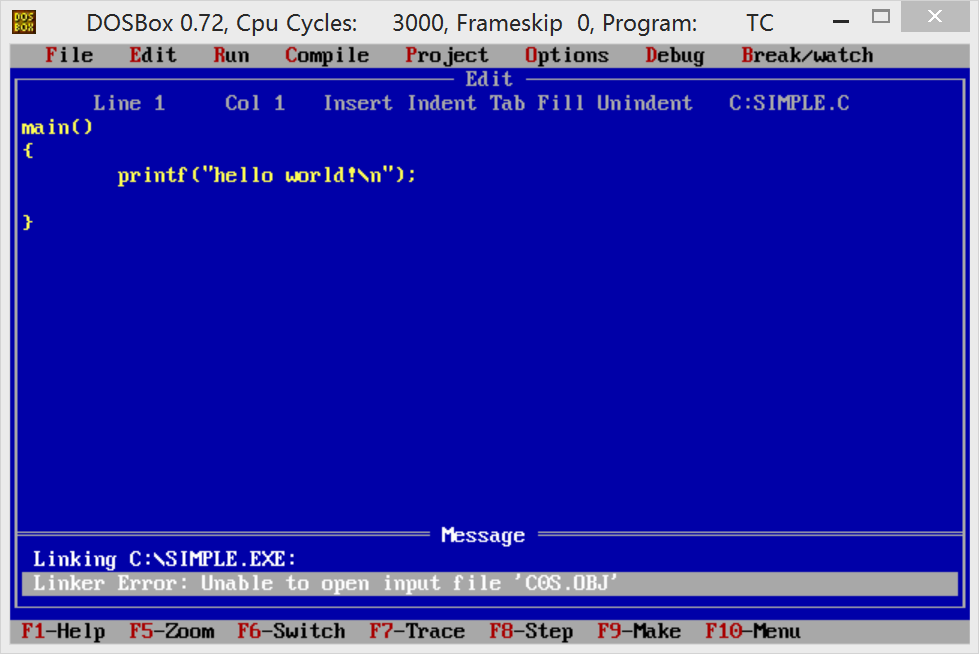
****

由此，确认“编译成功simple.OBJ文件”

按照书上步骤做到第7步：

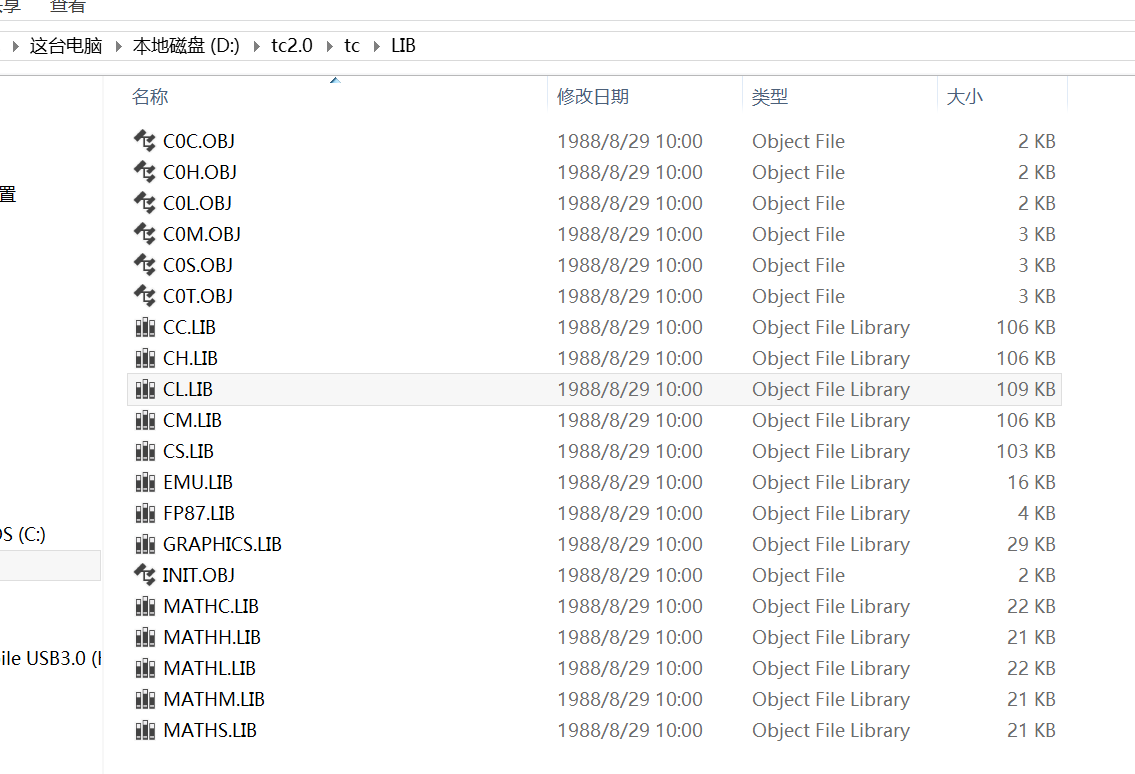
****

出现“Errors”，应该是“连接错误”

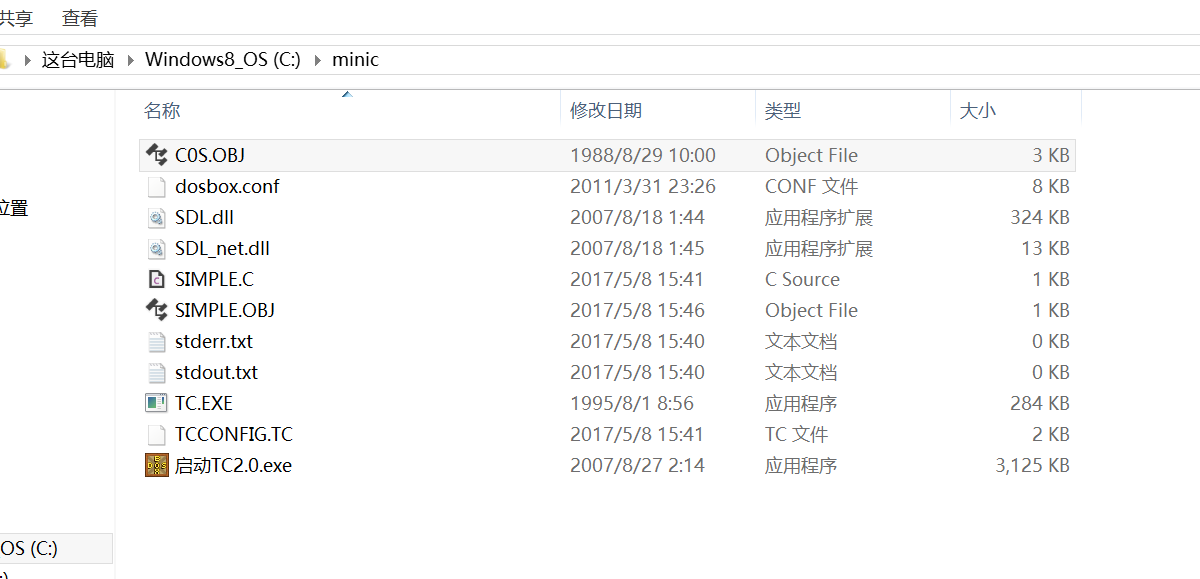
****

出现和书上预想的一样的错误提示信息。

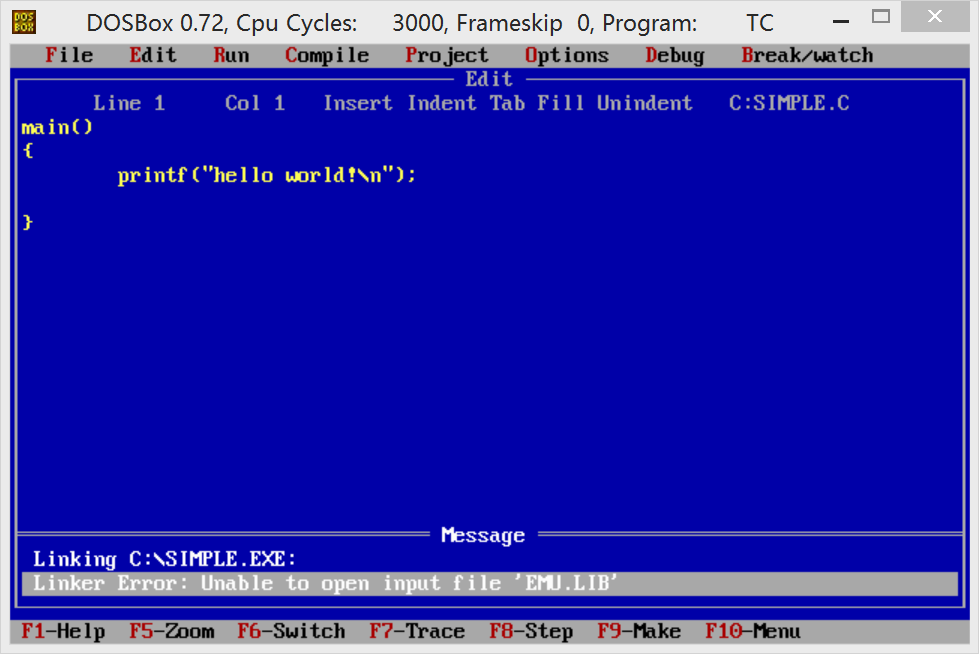
找到C0S.OBJ文件，路径如图：

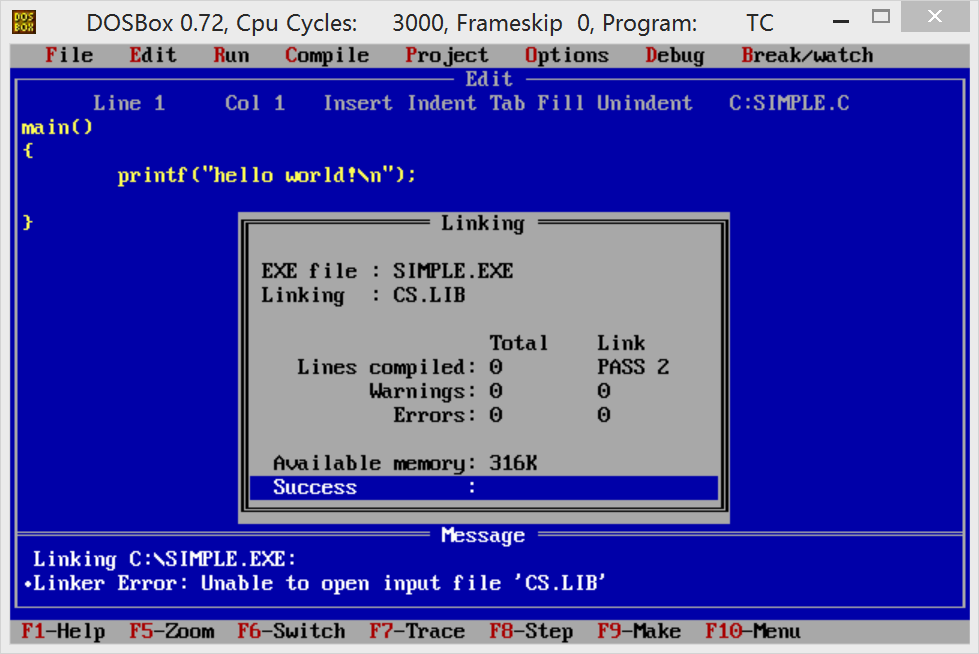
****

拷贝：

****

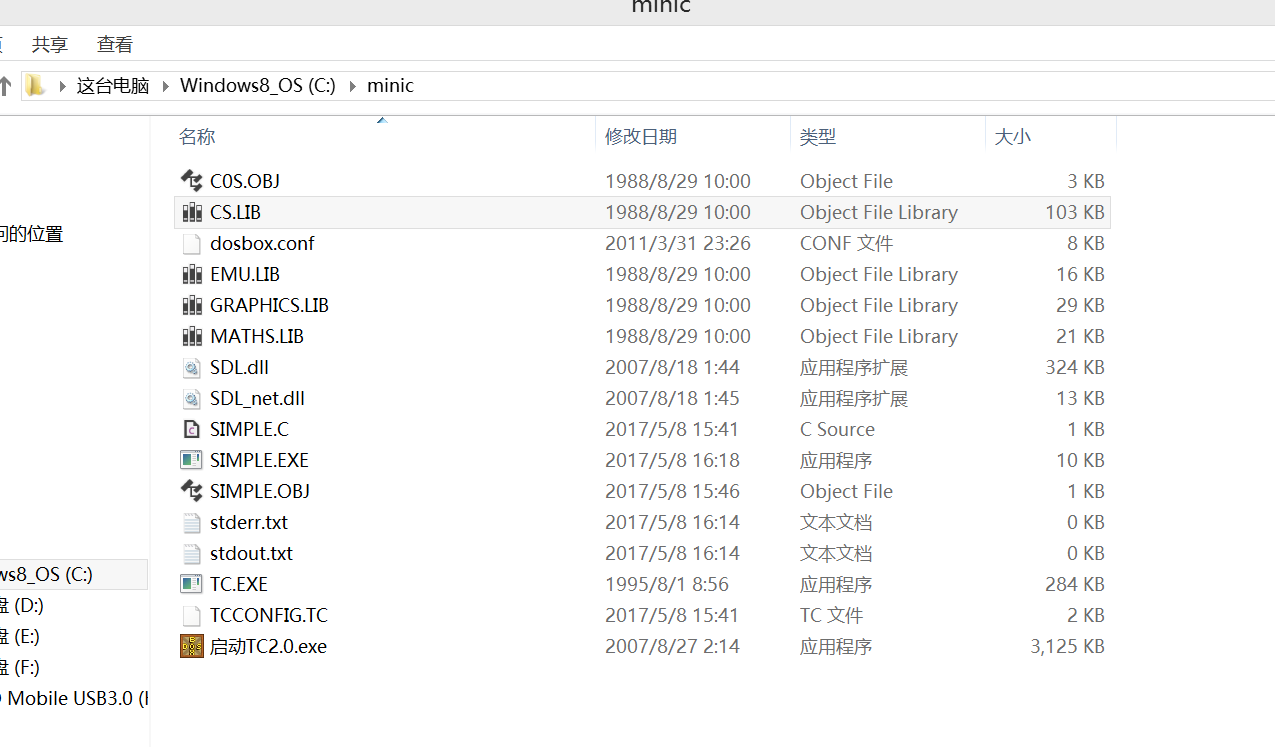
再次连接，结果出现“新挑战”：

****

效仿以上步骤，直到出现下图情况：

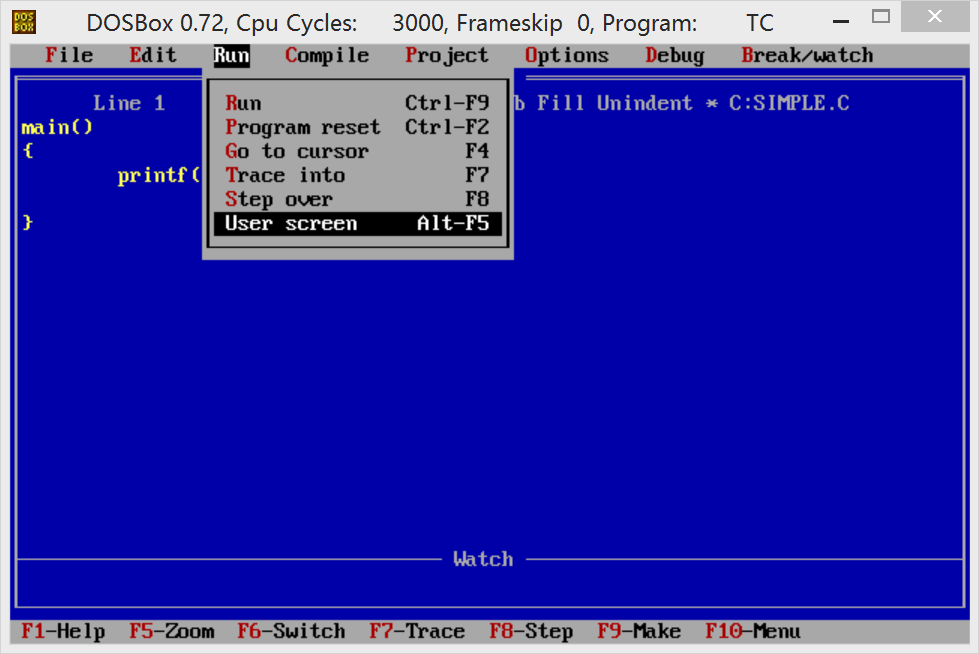
出现“Success”，意为连接成功。

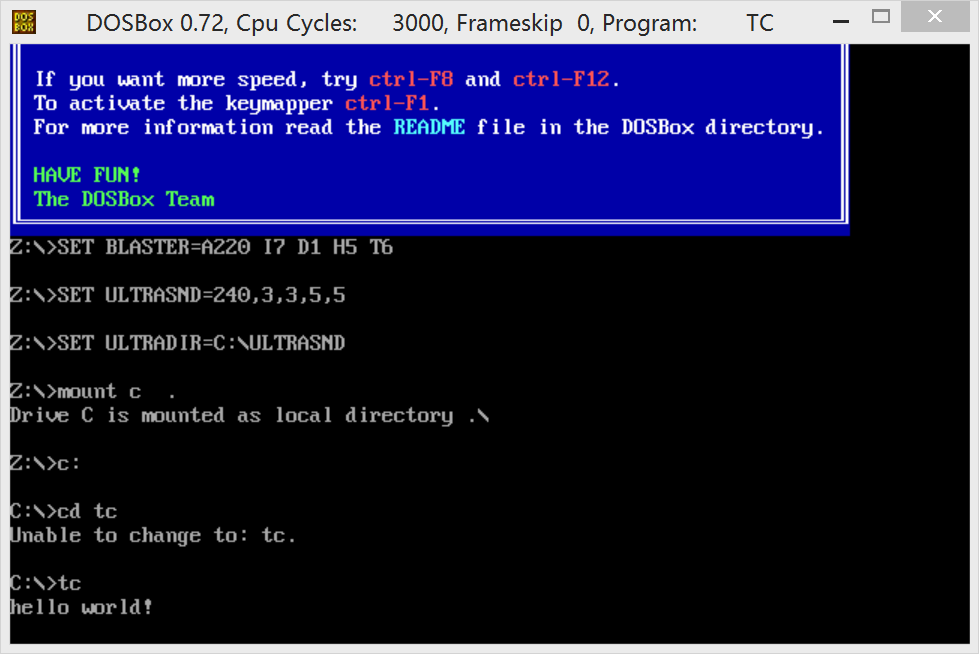
至此，再次检查**C:\minic** :



如图，已生成“TC.EXE”文件此过程中一共搬运了“C0S.OBJ、CS.LIB、EMU.LIB、GRAPHICS.LIB、MATHS.LIB”,而且提示消息有前有后，这个顺序是按照什么规则排的？或者这一点并没有必要了解？存疑。

最后运行一下，看看效果

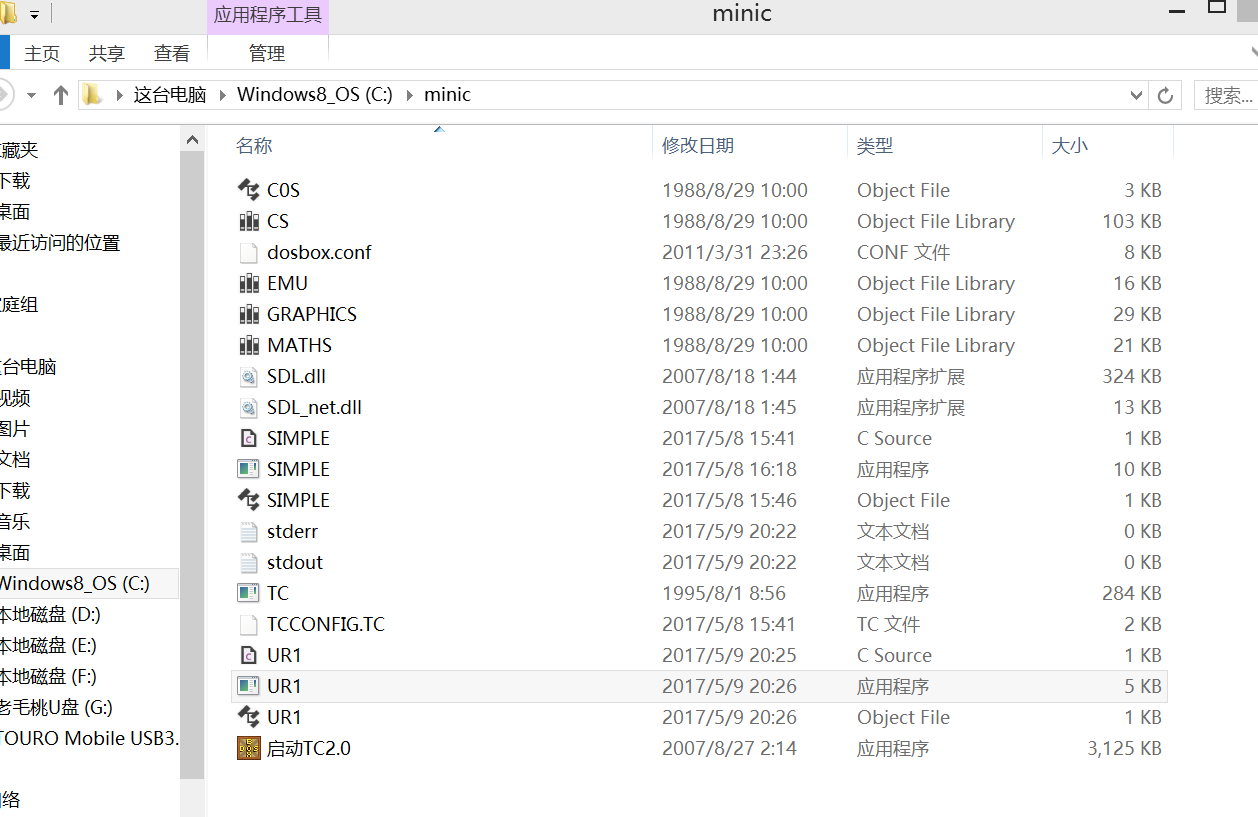


****

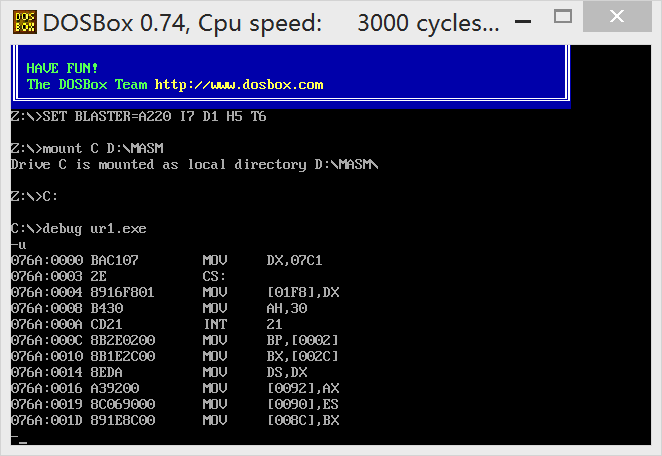
**Hello world！**

**综合研究二：**

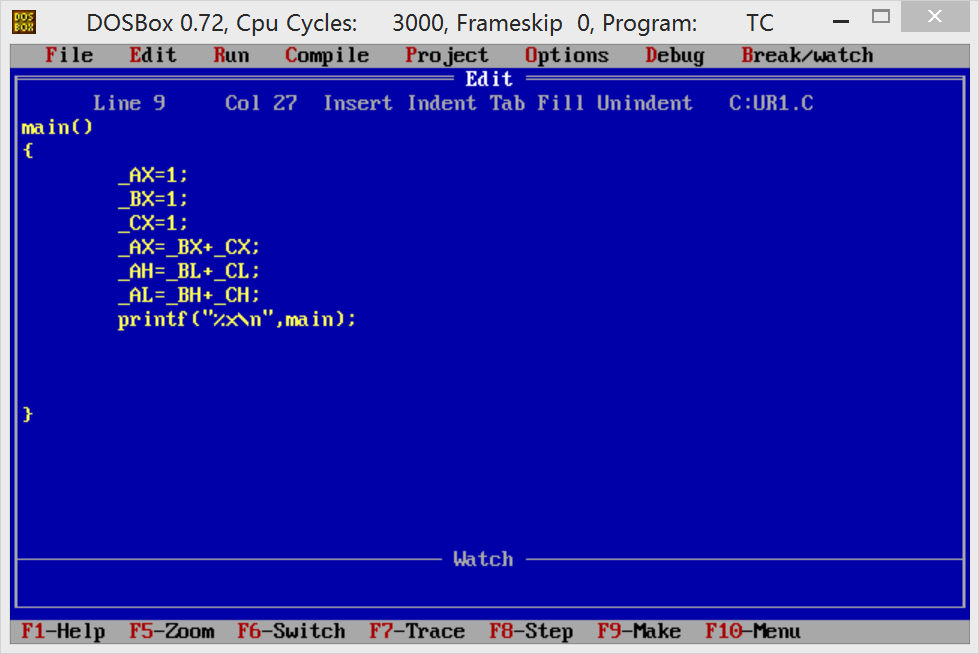
在综合研究一的基础上，很快地编译连接生成了UR1.EXE文件。如图所示：

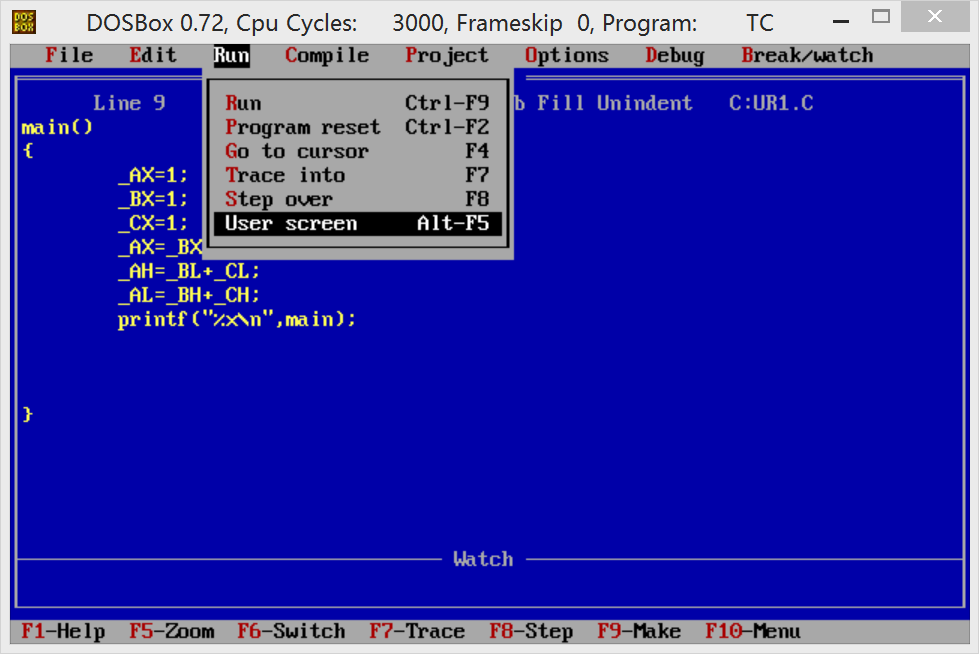
****

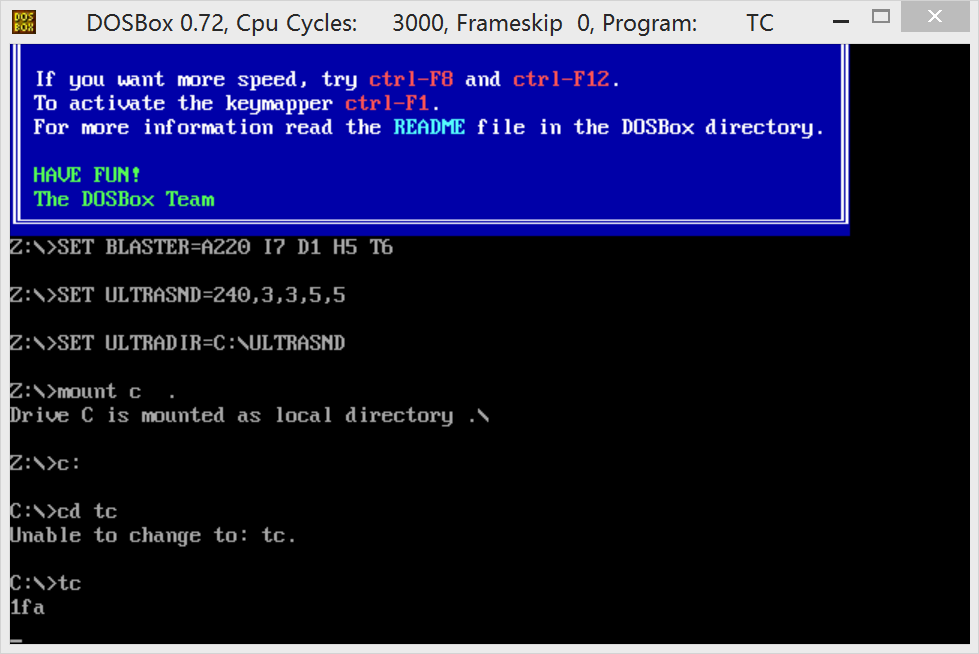
用Debug加载UR1.exe，用u命令查看UR1.c编译之后的机器码和汇编代码，但是用u命令之后，出现的反汇编代码却不是预想的那样，没有一处符合，有很大可能这不是main函数存放的位置，或者我应该向前或者向后移一点？

****

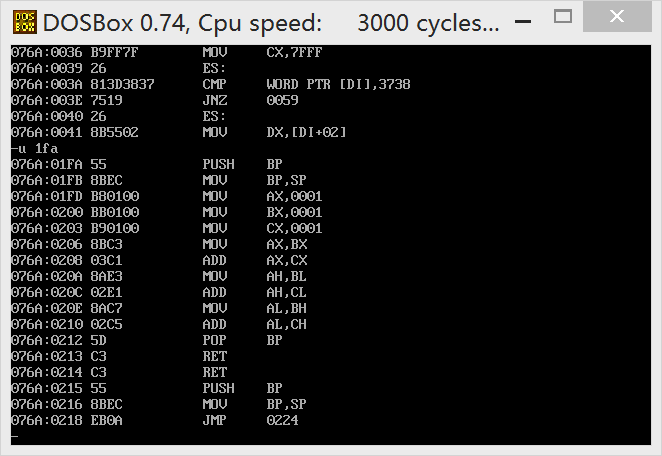
不过好在书上在提出这个问题的时候，也提出一个建议，试着做一下：

****

****

****

再u一下，这样明显很符合预期：

****

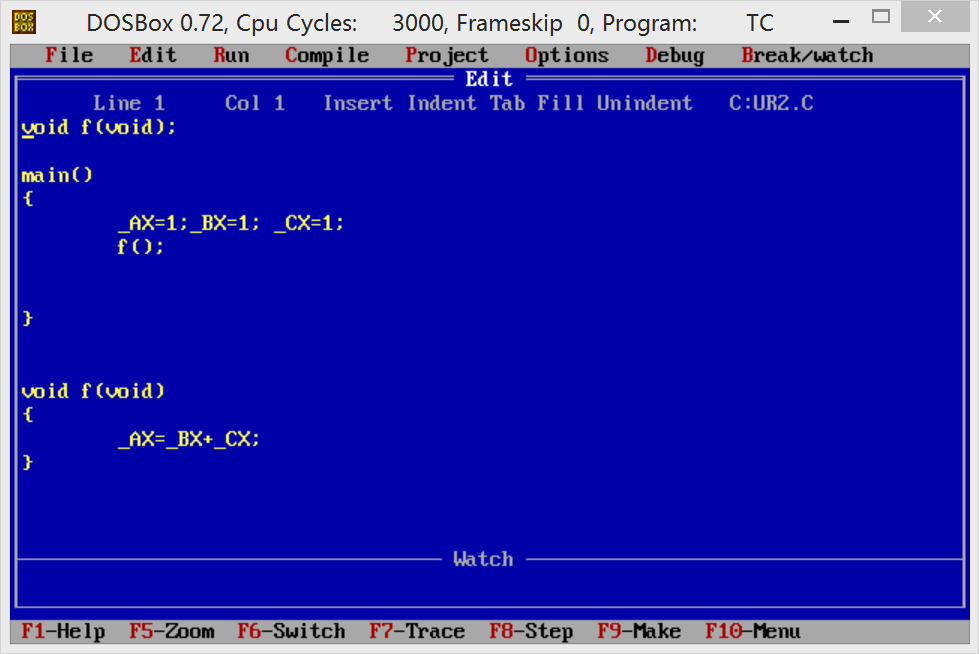
比较一下前后的地址，前：076a：0、后：076a:1fa。段地址都是cs段地址，偏移地址不同，为什么会相差 1faH呢？存疑。

0-1faH之间的代码有用吗？何用？存疑。

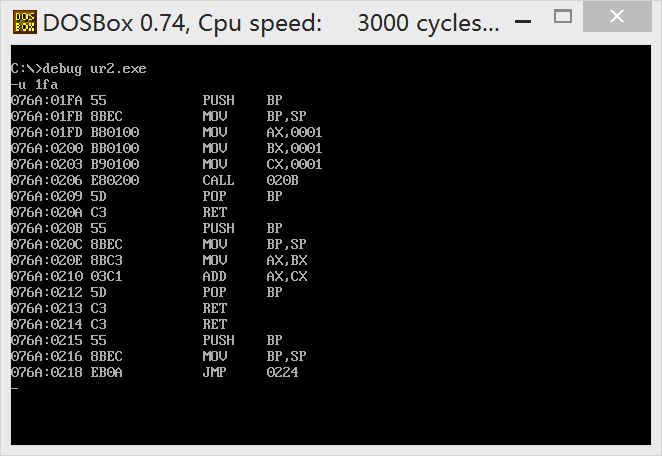
而在这里，再回头看一下在这一过程中指点迷津、至关重要的那条语句“printf(“%x\n”,main);”格式如此，是否可以理解为这里的函数名简直是在当作变量用，但是区别呢？联系呢？函数名、变量，这些区别于汇编的新生名词，在C语言反汇编为汇编语言的时候又是具体以怎样的熟悉的形式展开的呢？

相对于函数，很容易就想到了子程序，包括子程序的标号、调用、返回等都与函数感觉有某种近亲关系一样。

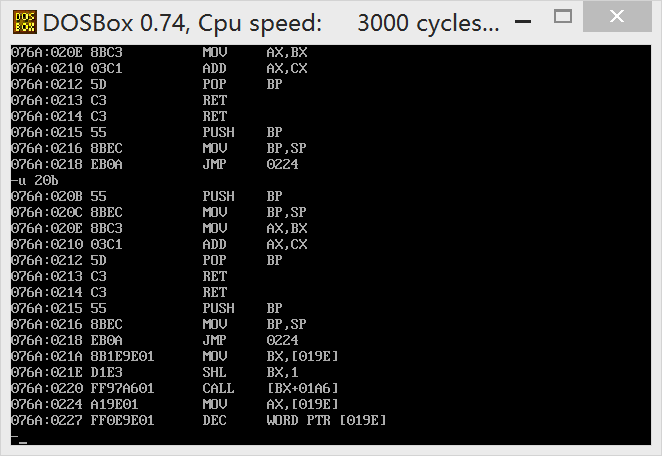
接下来的第5小点正好就是验证此项疑问的。



再用u 1fa命令：



再用u 20b命令：

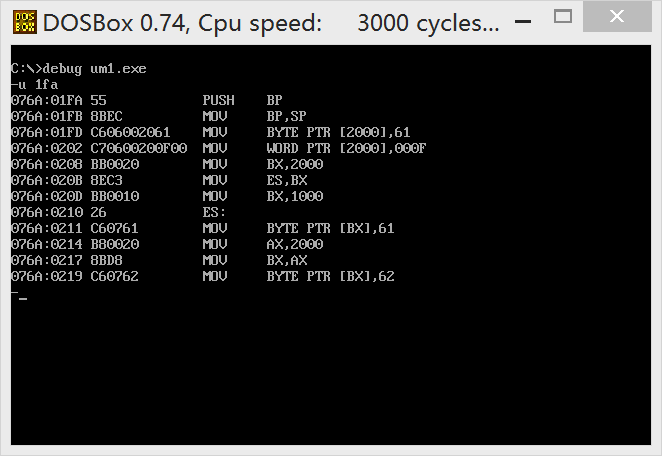


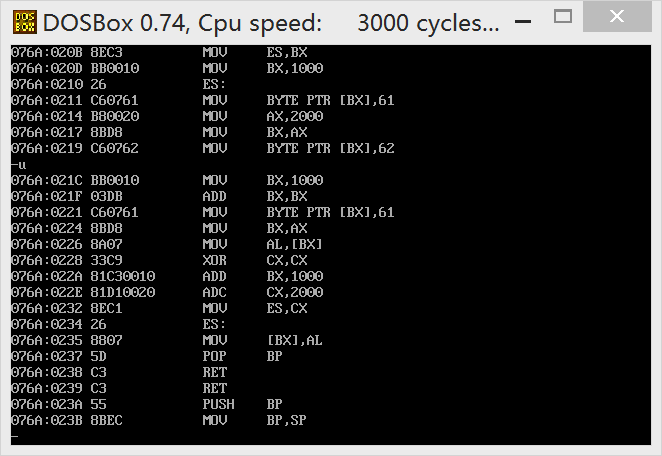
至此，综合研究二课程内容完结。

**综合研究三：**

1>.um1部分：在综合研究一的基础上。如图所示：

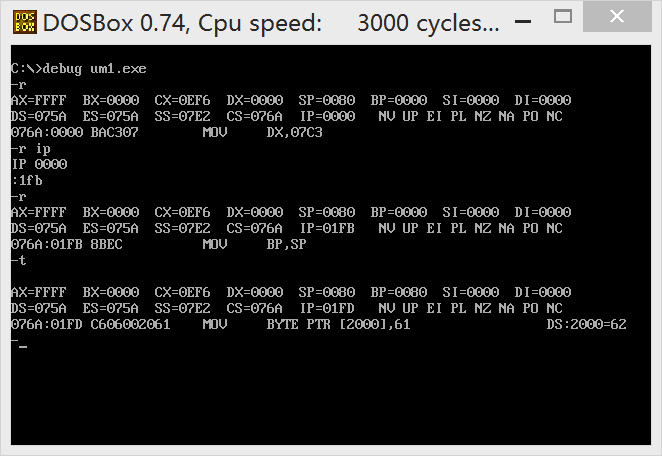
![}]($]E8_MTAJ~L8J@CVX3$K](data:image/png;base64,)

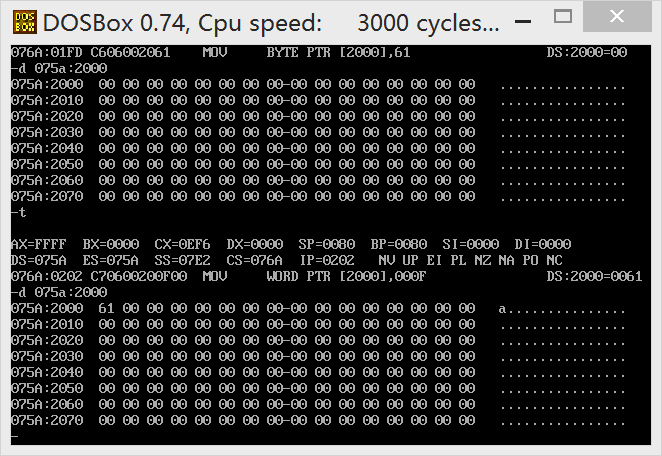




开始进行单步跟踪，并查看相关内存单元的内容：

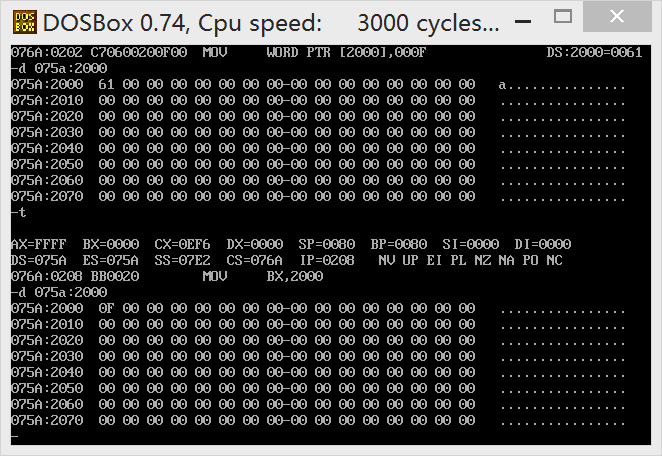
\*(char \*)0x2000=’a’; 执行前后比较：





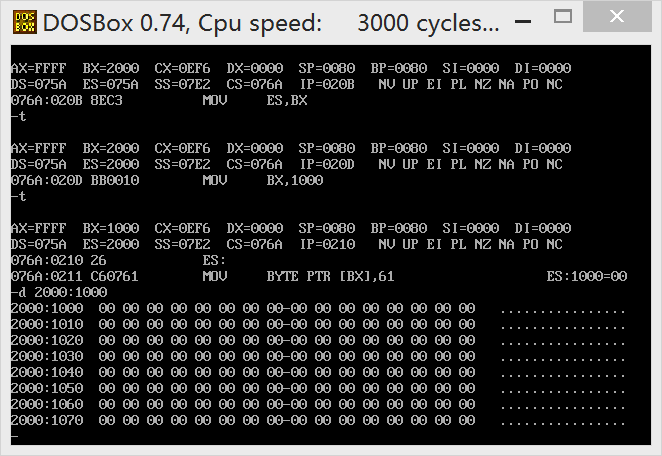
；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；

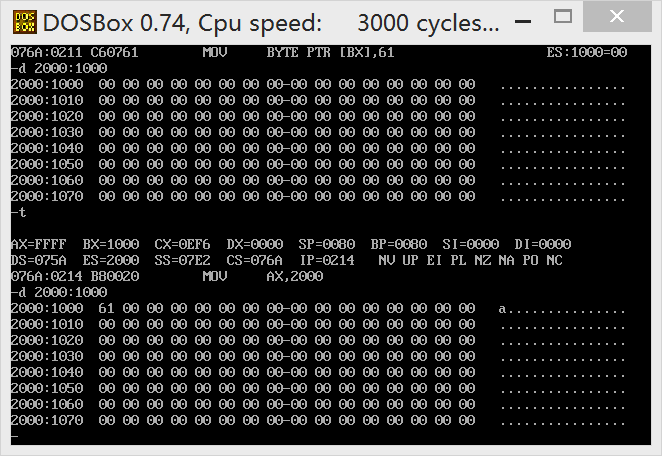
\*(int \*)0x2000=0xf; 执行前后比较：



；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；；

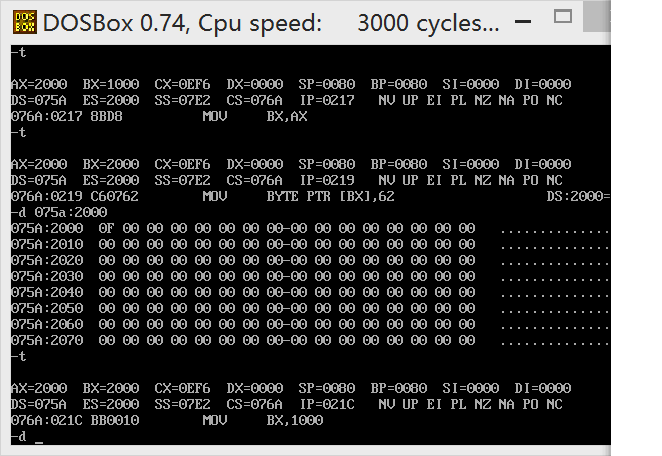
\*(char far \*)0x20001000=’a’; 执行前后比较：

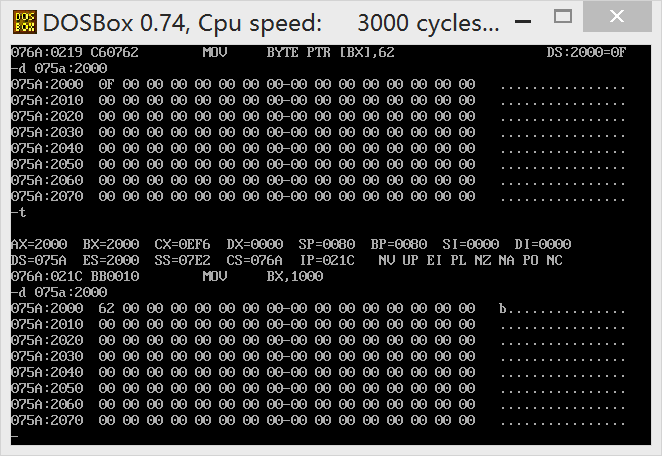




\_AX=0x2000;

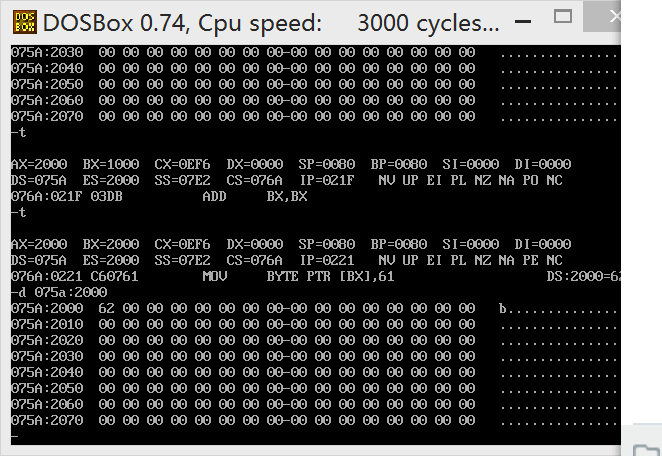
\*(char　\*) \_AX=’b’; 执行前后比较：

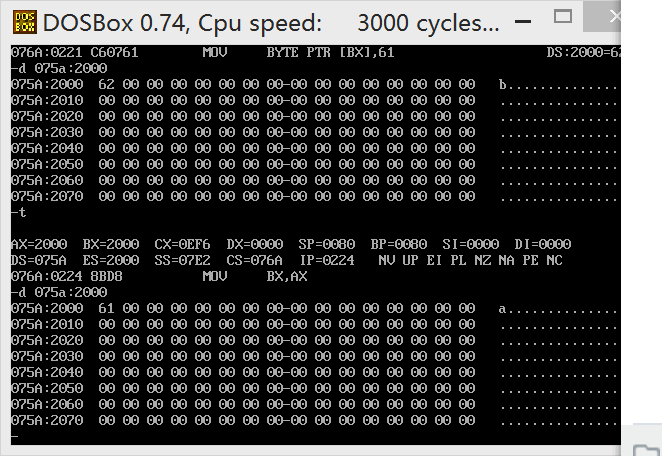




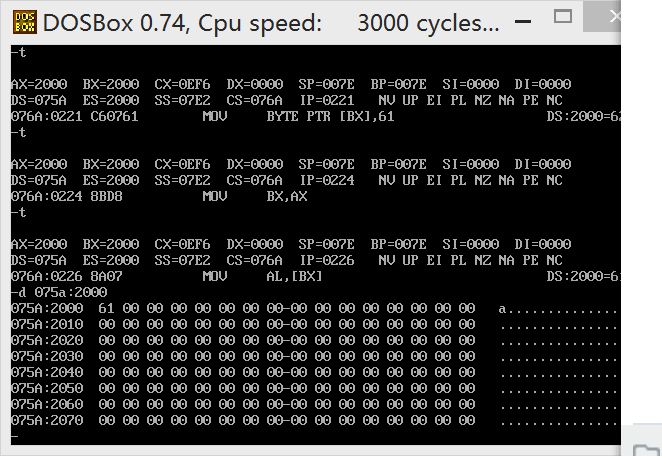
\_BX=0x1000;

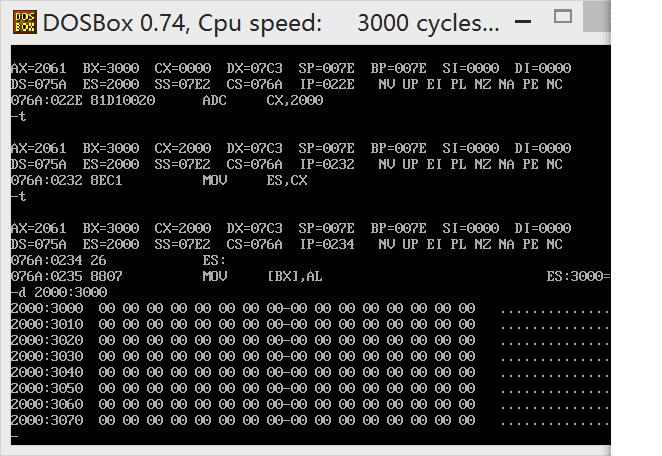
\*(char \*)(\_BX+\_BX)=’a’; 执行前后比较：

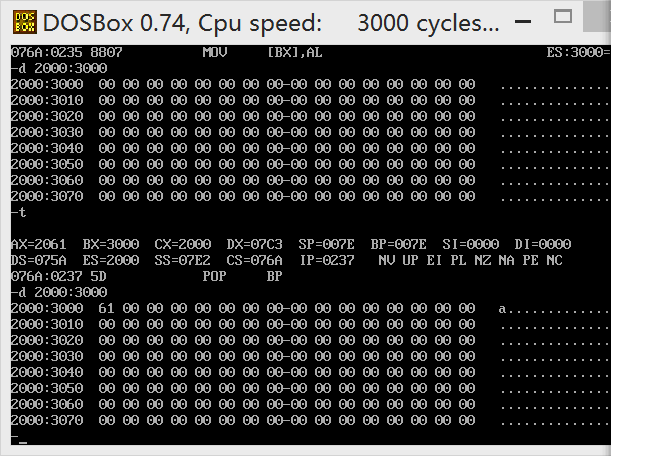




\*(char far \*)(0x20001000+\_BX)=\*(char \*)\_AX; 执行前后比较：



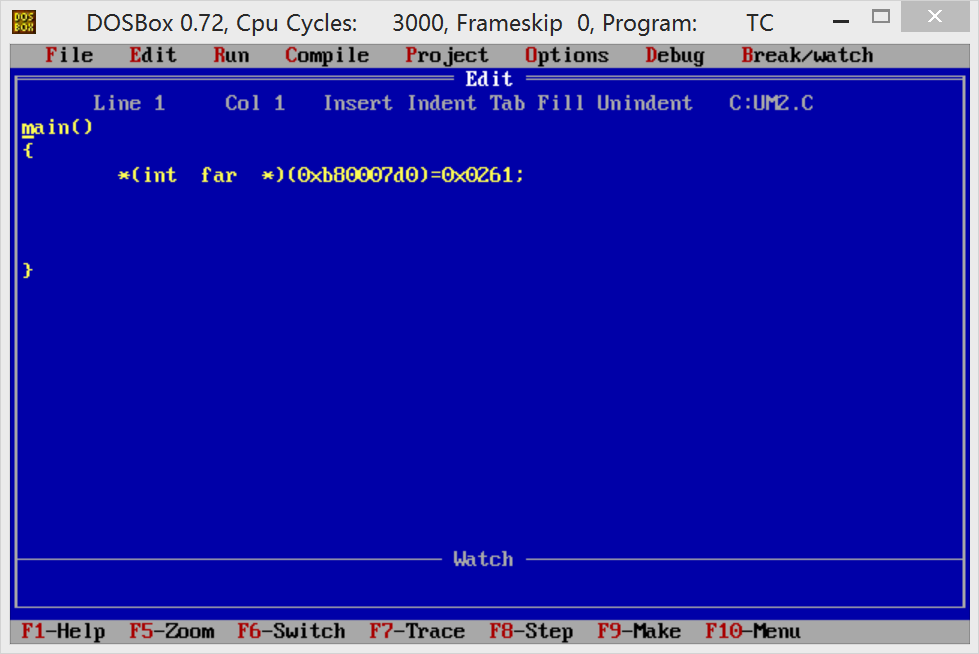




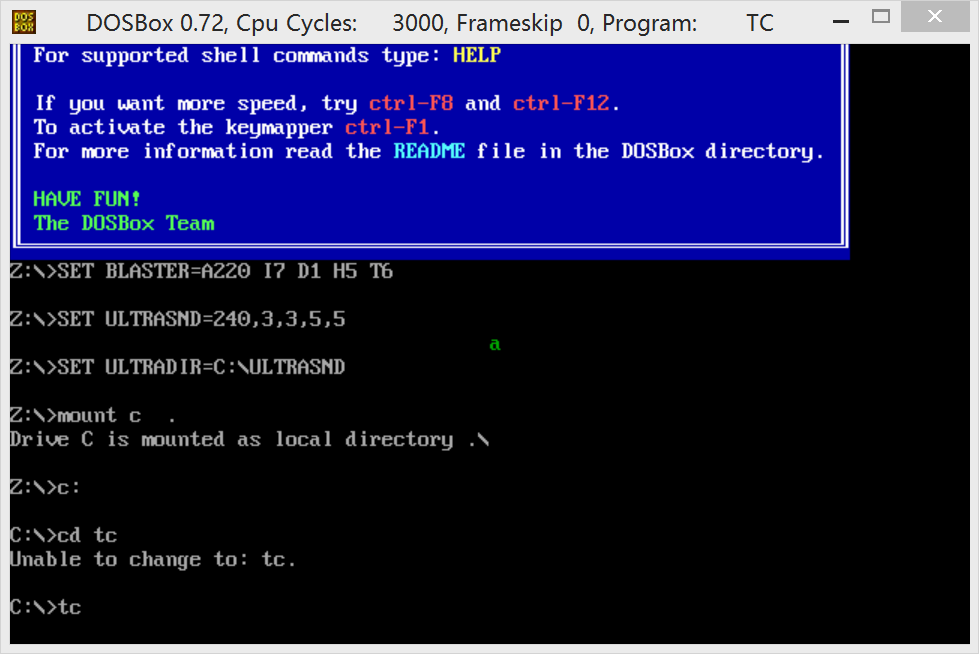
2>.um2部分：

由于char型是一个字节，而只用一条语句得用至少两个字节，所以这里可以用int型。如图：

程序：

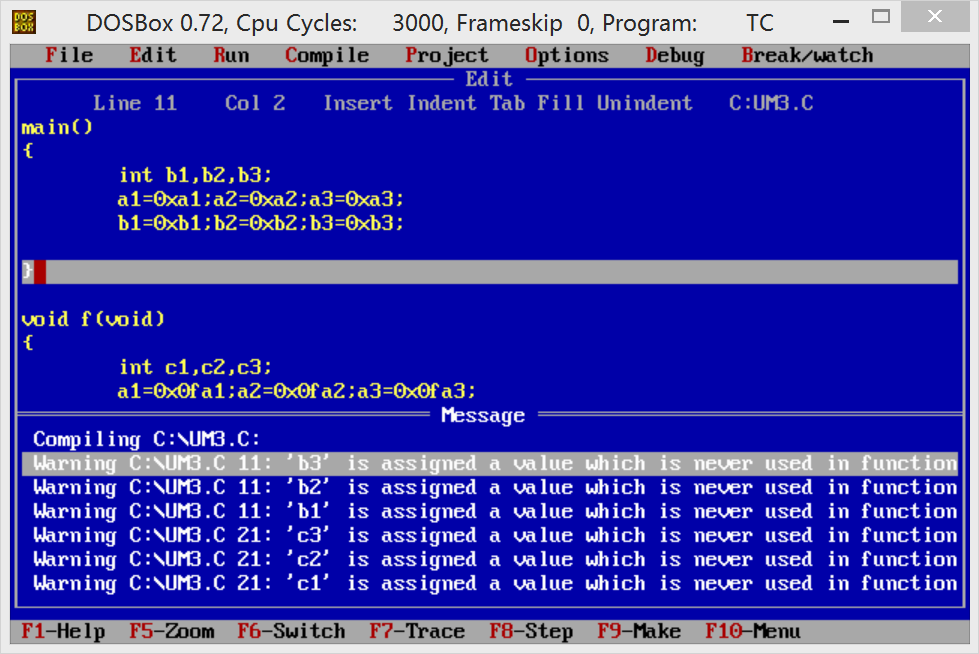


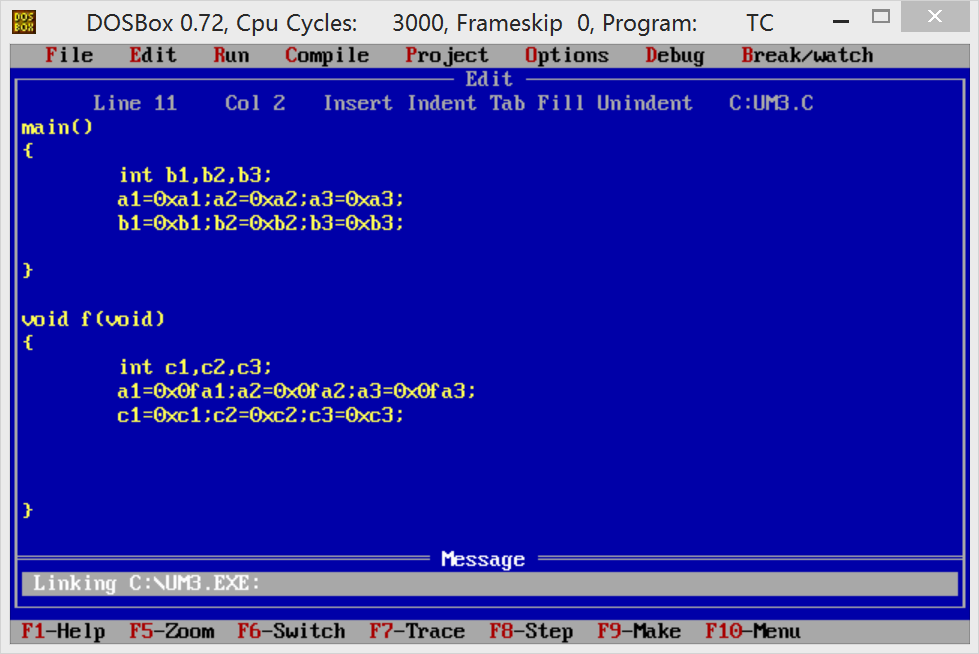
效果：



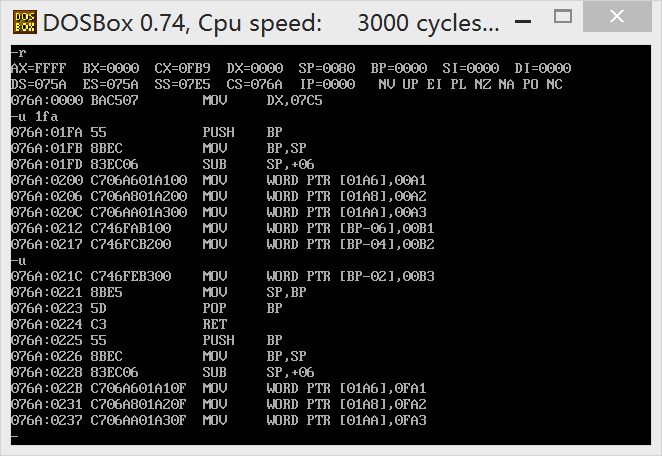
3>.um3部分：

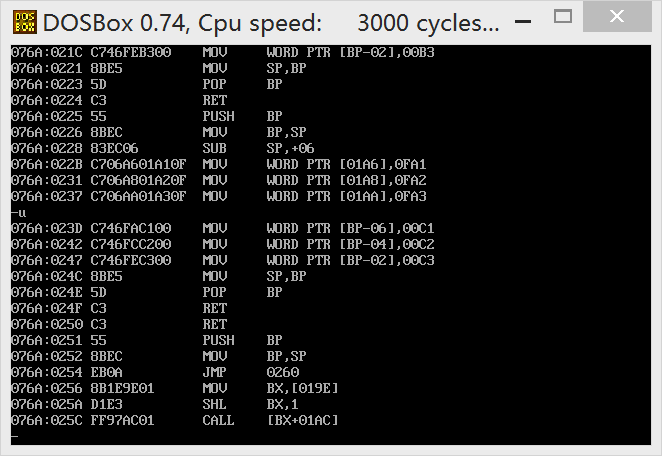
编译时有许多警告，最终可以连接成um3.exe。





Debug：



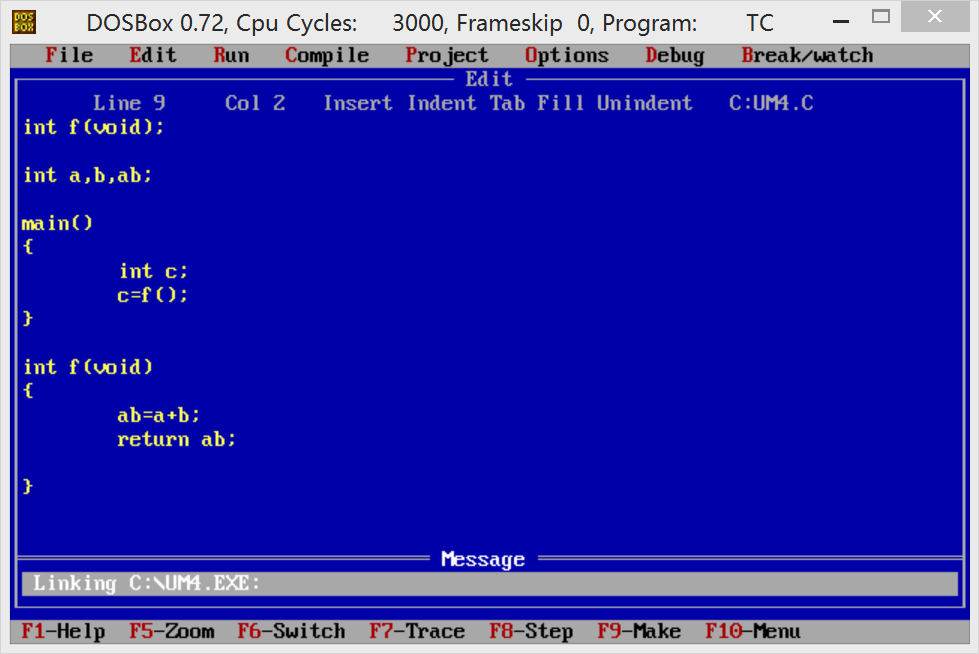


反汇编可看出全局变量a1的地址为01a6，a2的为01a8，a3的为01aa，局部变量b1、b2、b3的地址是由bp-6、bp-4、bp-2表示，而在反汇编程序开始处有将sp的赋值给bp的语句**“mov bp，sp ”**，sp自减了6，因为sp存放的是栈的栈顶偏移地址，可知实际上程序是将b1、b2、b3依次放入了栈中。但是在程序结尾处有**“mov bp，sp”**，把栈顶指针又还原了，这时b1、b2、b3的内存空间已不在栈中，随时可以被其他程序的数据覆盖。

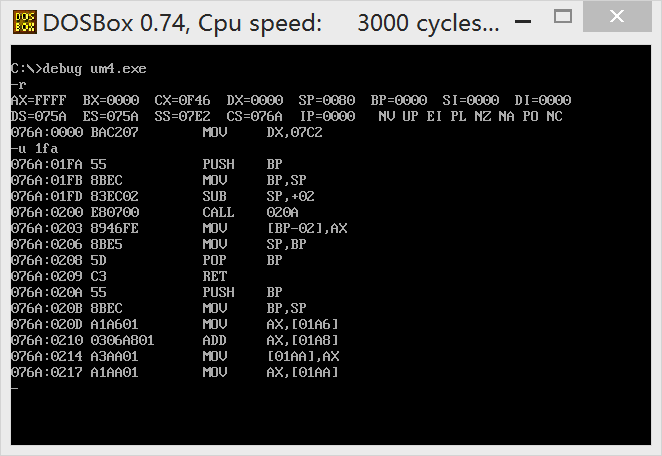
可以看出，全局变量是存放在内存空间的指定内存地址处，而局部变量是存放在栈中。

因为要用bp记录sp原来的位置，以便函数结束时将sp还原，所以要对bp进行保护，所以会有push bp;mov bp,sp。

4>.um4部分：



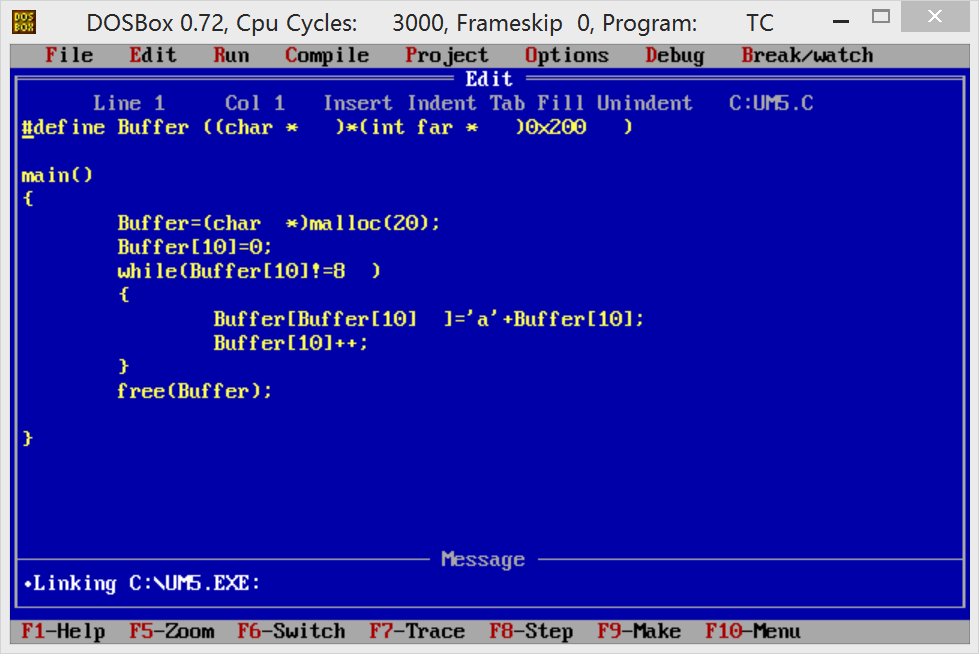
用debug 的u命令反汇编一下，要查看返回值存放在哪里，主要看下图中的一条关键语句就可以了。



即“mov [BP-2],AX”,可知这个程序中将返回值存放在寄存器AX中。

但是这又是否是特例呢？还是这是通用的做法？既然涉及到值传递，这是否又与参数的传递有关呢，或许对了解综合研究五的printf()有关。

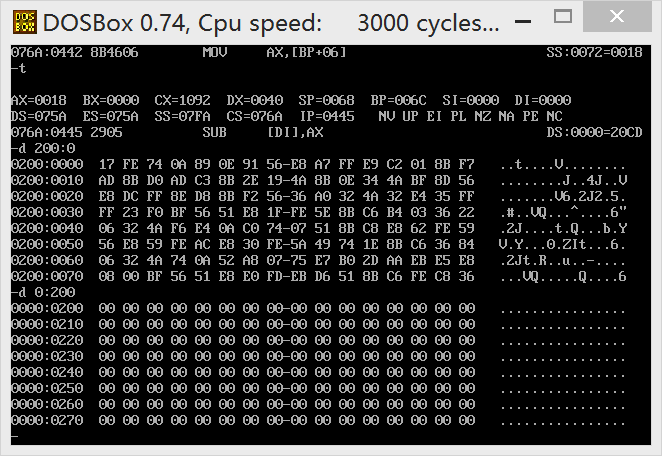
5>.um5部分：



这里两处内存地址都没有改变，需要再验证。

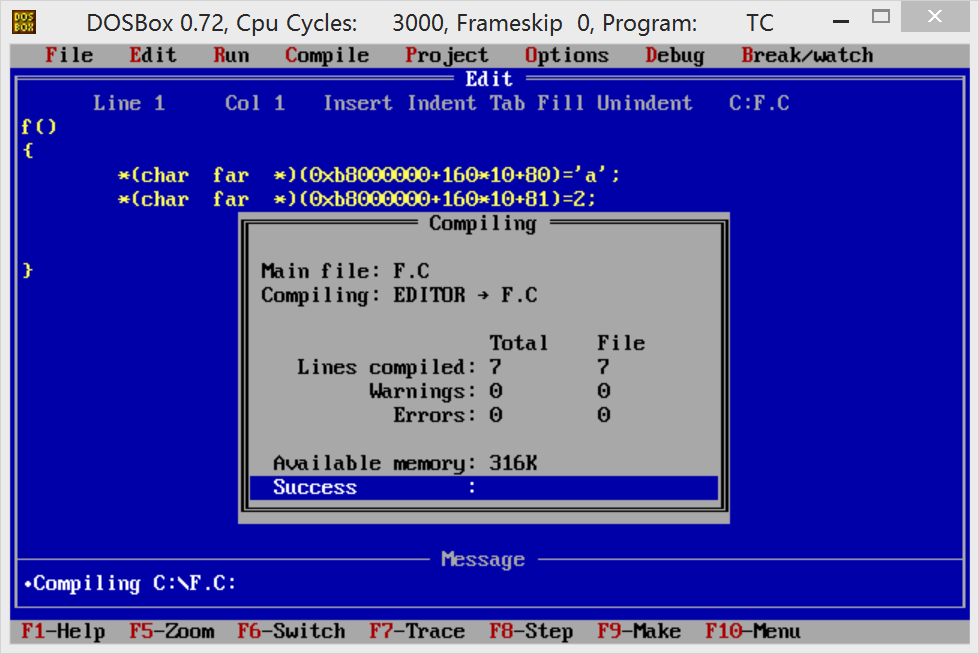
#define是宏定义，在程序中用Buffer代替((char \*)\*(int far \*)0x02000000)

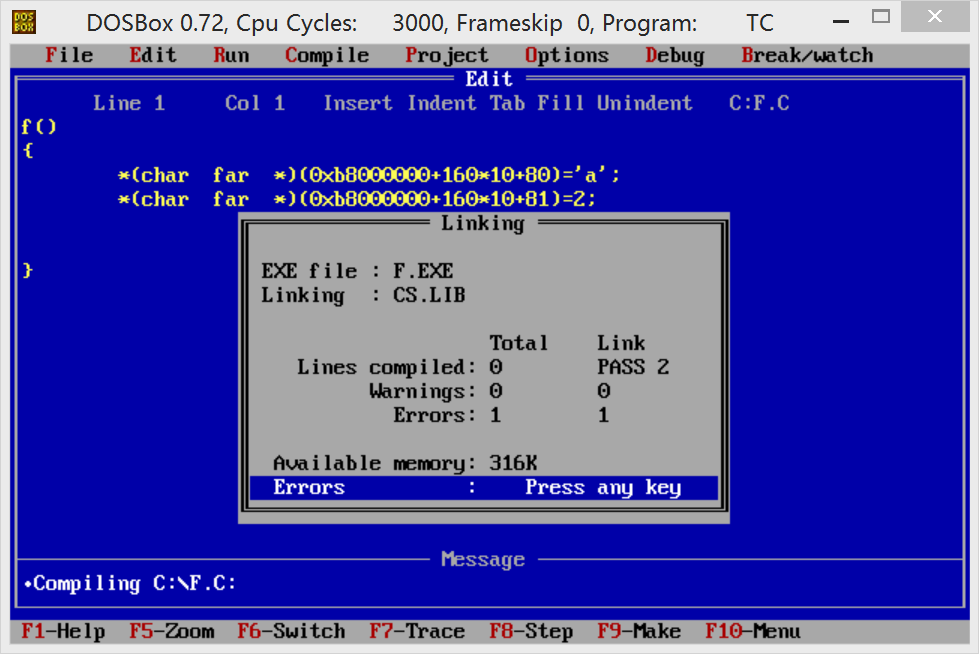
malloc（20）是开辟20个字节的内存空间。

****

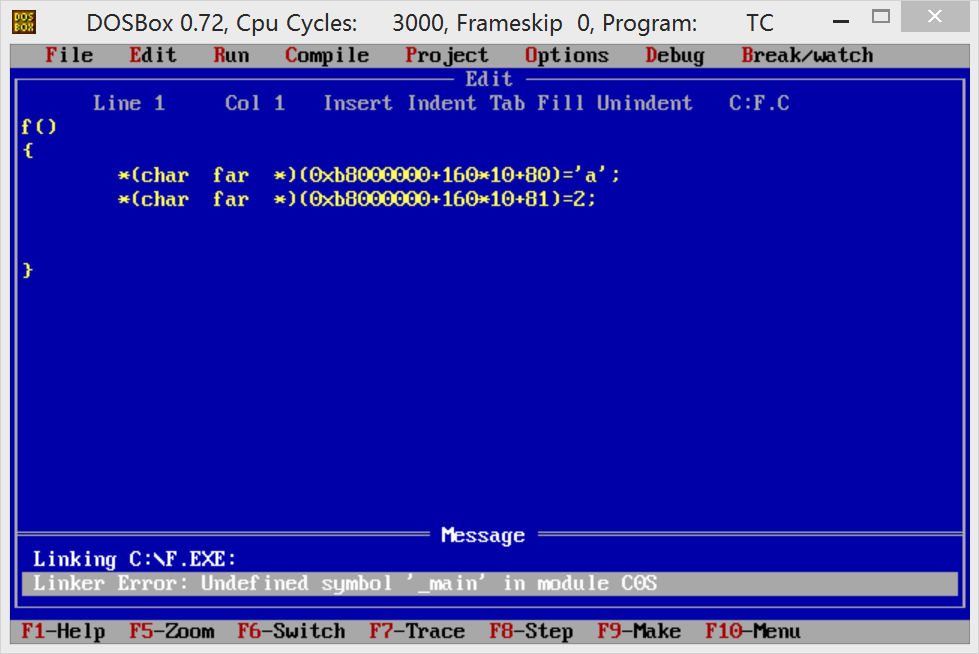
**综合研究四：**

1>.编写的f.c程序编译可以通过，但是连接的时候出现问题。如图：

****

****

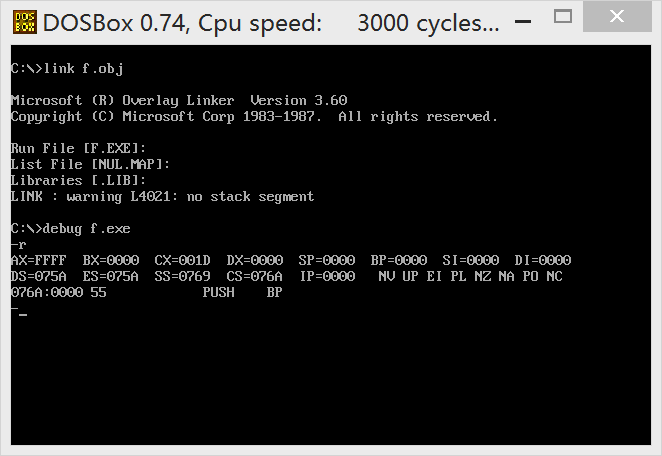
显示出的错误信息如图：

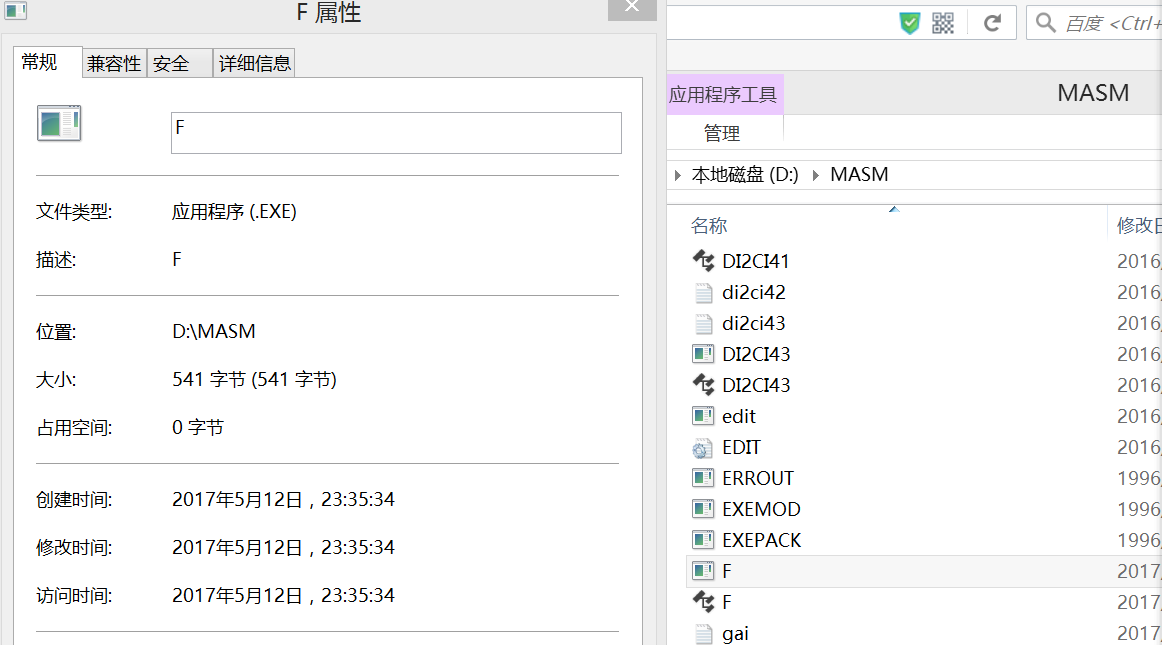


翻译成中文是：在c0s模块没有定义符号’\_main’。

这个错误信息应该与存放在**C:/minic**目录下的 **c0S.OBJ**文件有关。

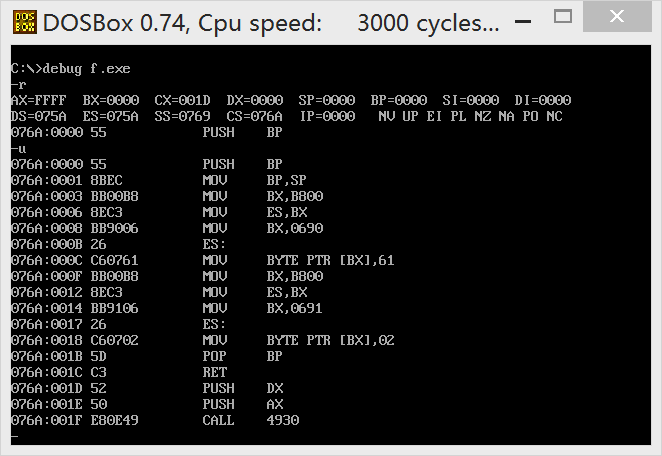
2>.此处cx=0x1d，即29字节。代码长度29字节。



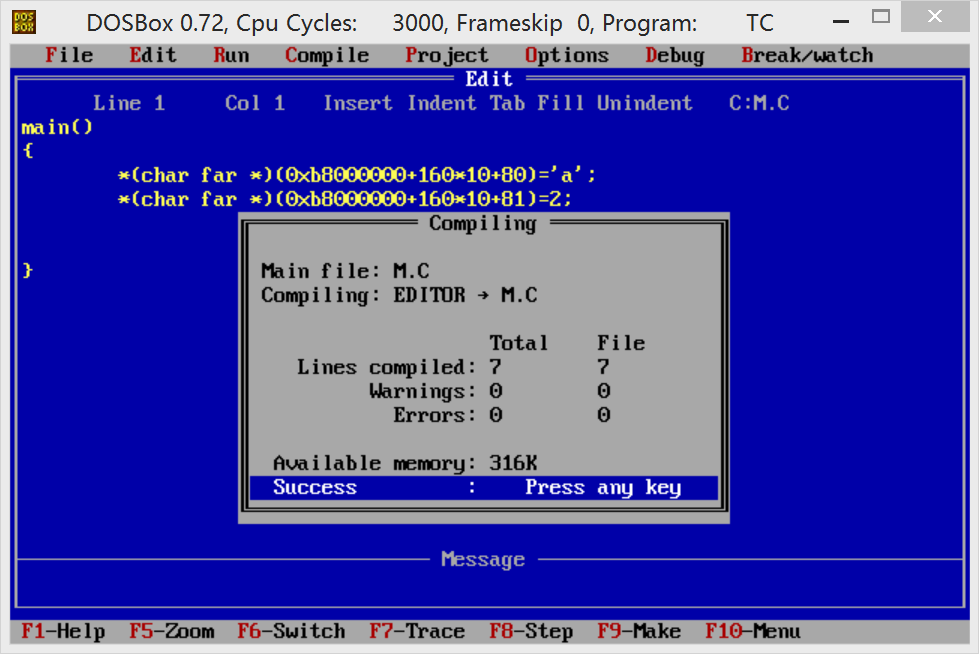


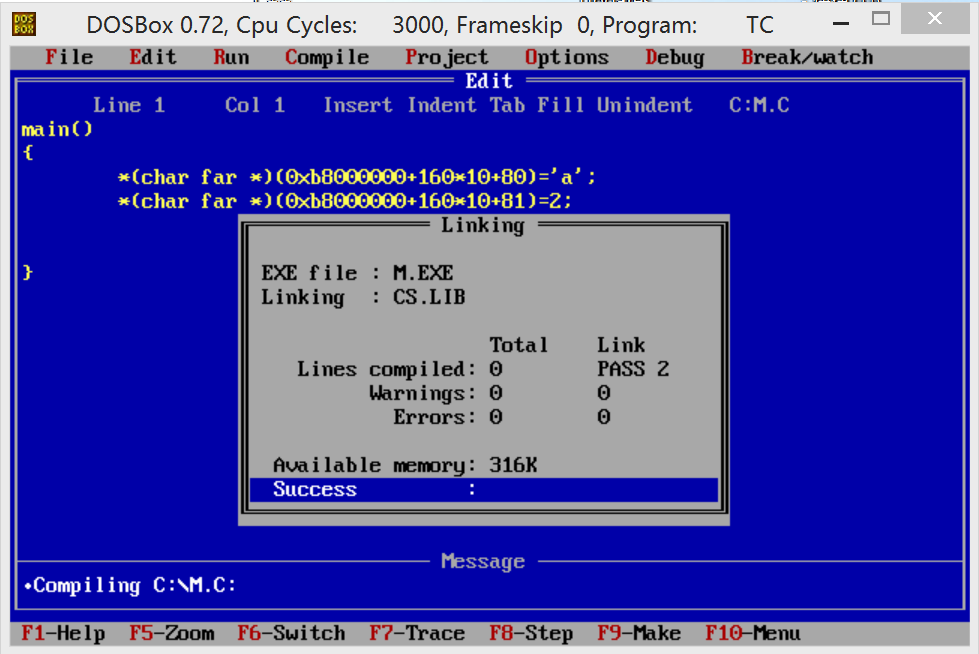
但是查看f.exe的文件属性时，又显示一个比29字节要大得多的541字节。

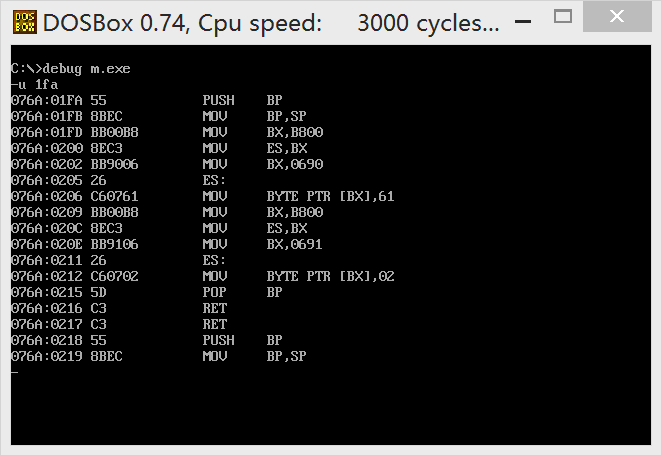
当直接用“u”命令反汇编的时候，正好出现了f.c程序的反汇编代码。可得，f函数的偏移地址是0。

****

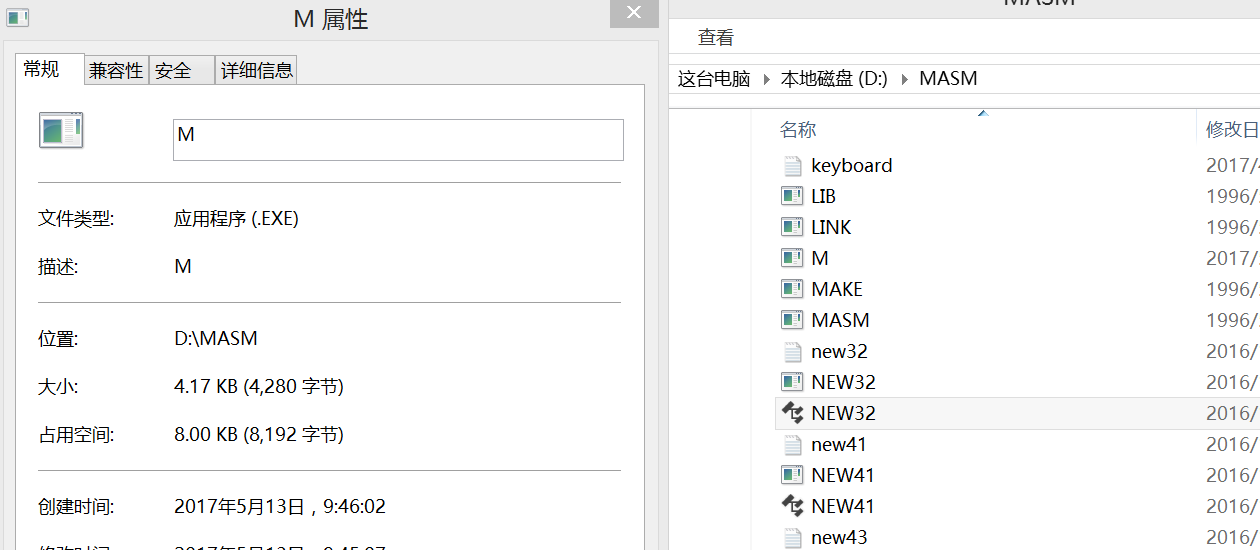
3>.用tc对m.c进行编译、连接，并用的debug查看m.exe整个程序的汇编代码。

****

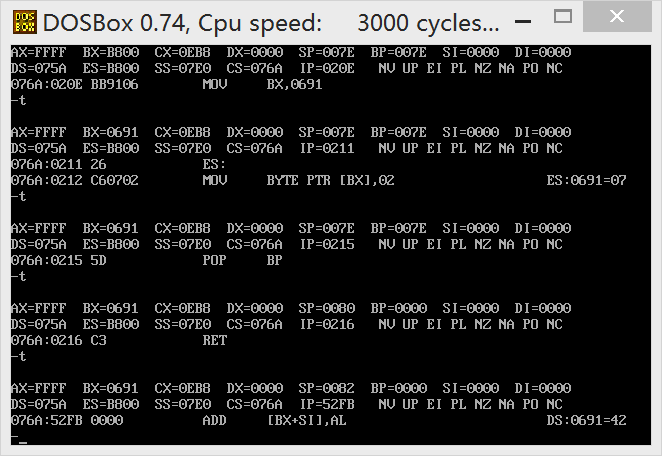
****

****

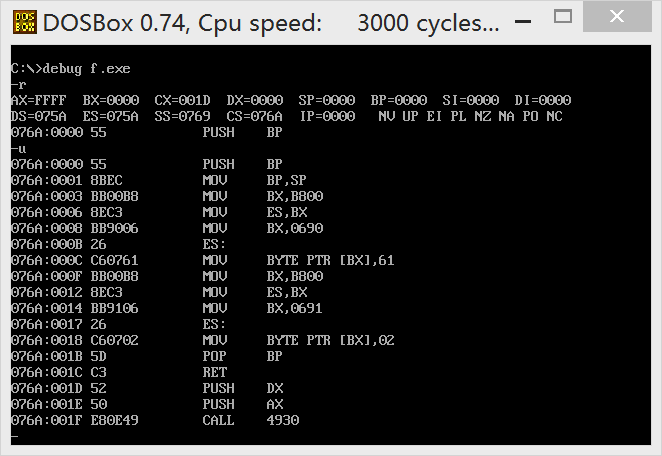
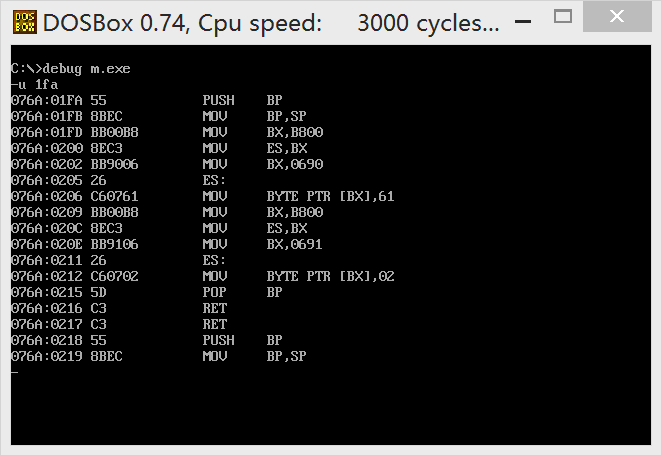
程序是从076a：01fa到076a：0216，一共29个字节。

****

属性查看的话，又是4280字节的大容量。

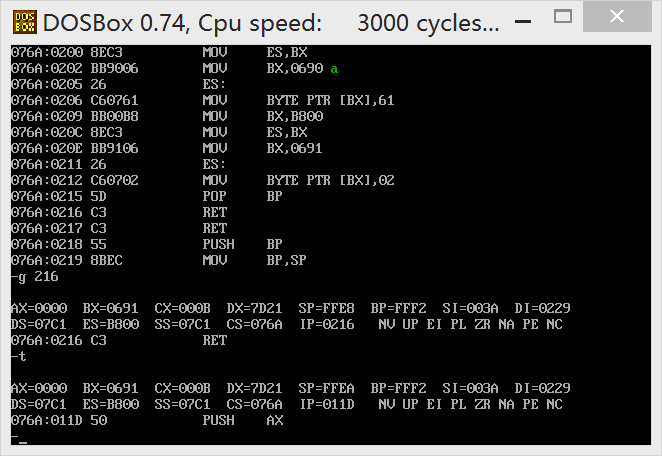
****

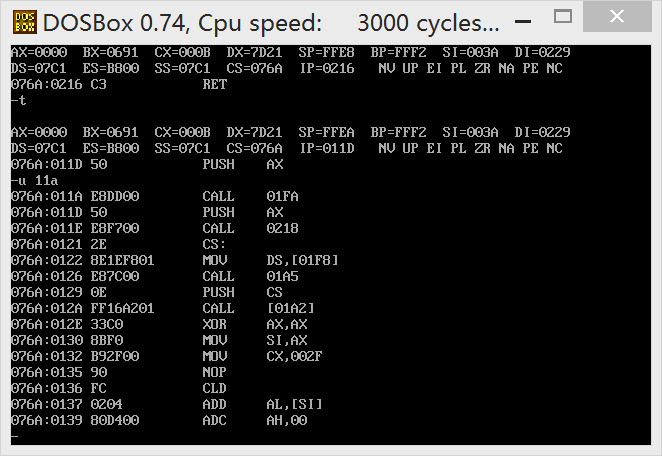
**m.exe的汇编代码与f.exe的汇编代码对比：**

****

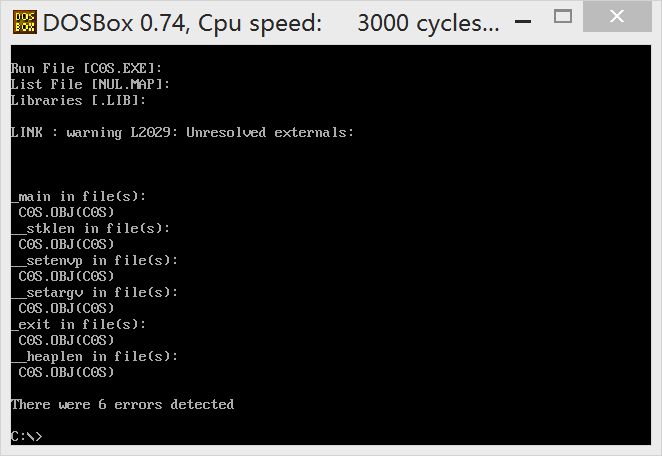
可见两者所在位置不同，还有就是m.exe可以正确返回，但是f.exe不行。

4>.

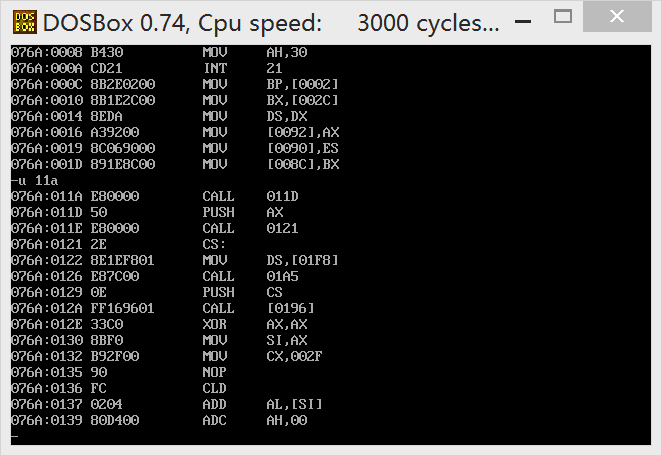
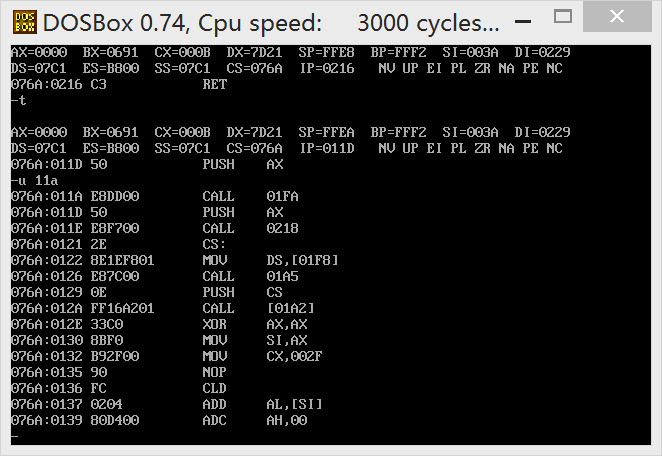




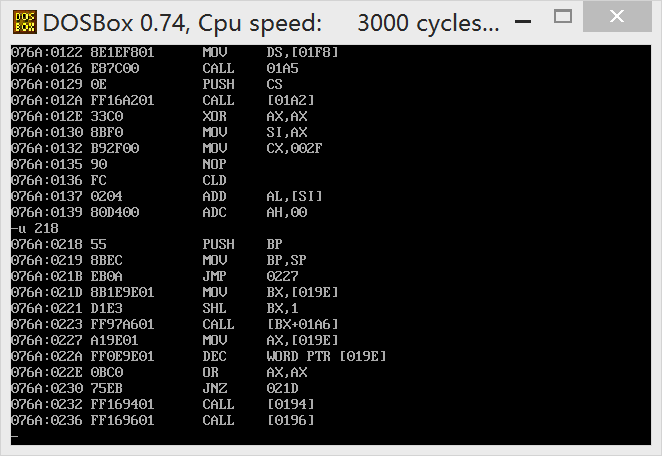
6>.要对c0s.obj进行研究，可以用link.exe对它进行连接，再用debug查看汇编代码。发现虽然link提示多个错误，但是还是生成了c0s.exe文件：



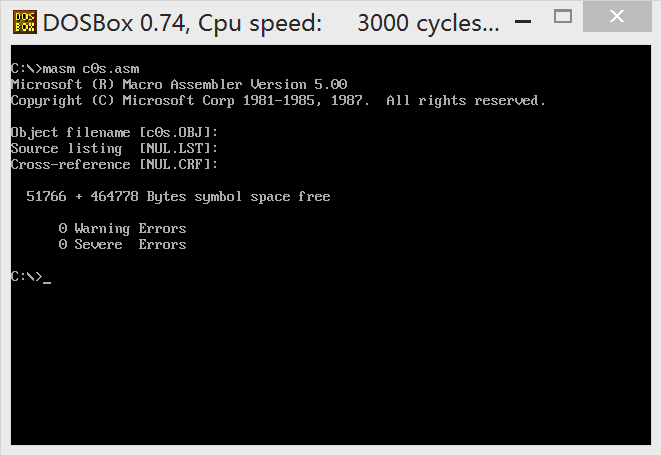
7>.

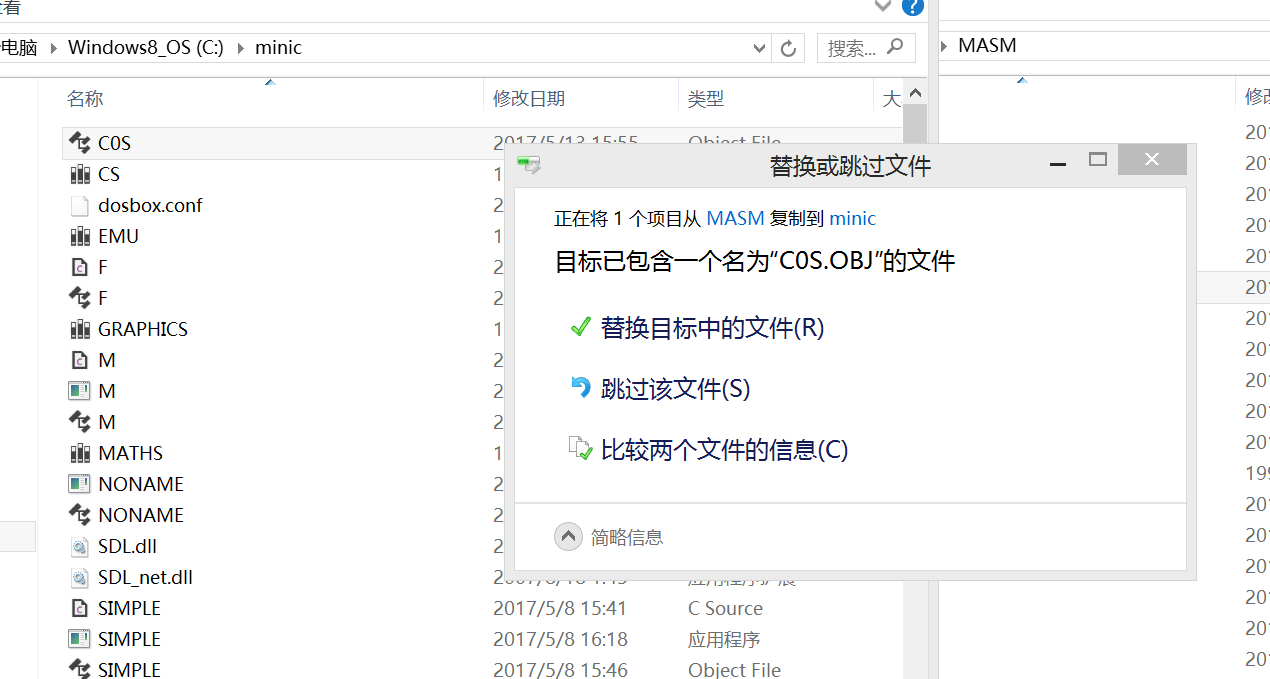


由图可以看出两者的主要区别体现在，在m.exe中第一条call指令，调用自己编写好的m.c中的程序，而c0s.exe中的第一条call指令，则是调用下一条语句，不过这样可以用吗？（没有配套的ret，存疑）；同样的，在m.exe中第二条call指令，调用地址为“076a:0218”中的程序，而c0s.exe中的第二条call指令，则是调用本身所在行的下一条语句，

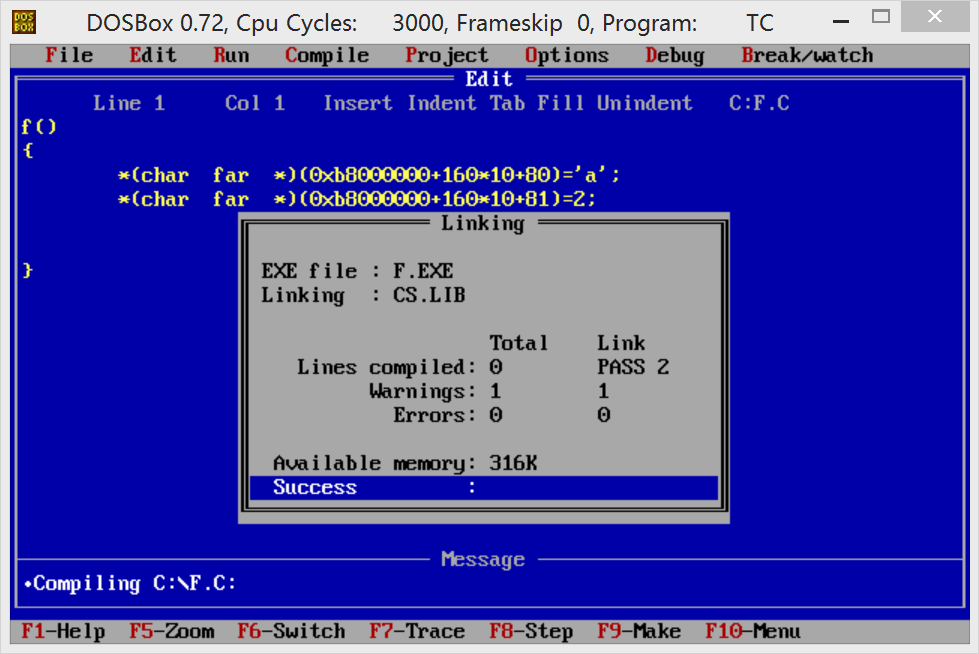


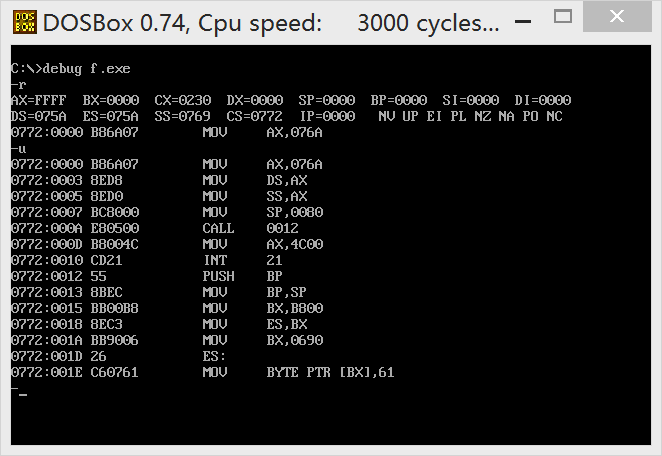
8>.



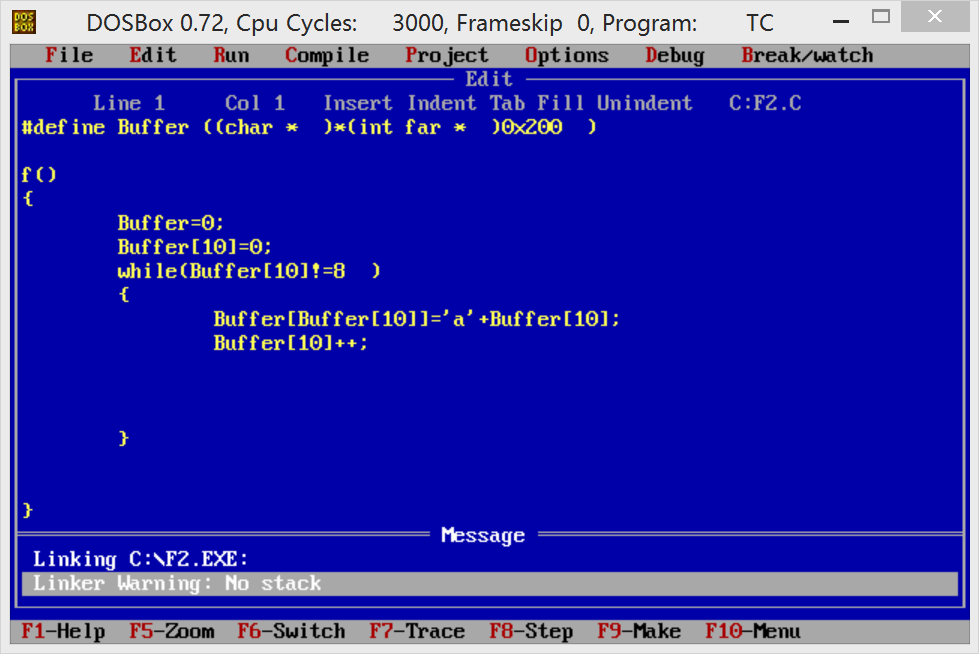


9>.用tc将f.c重新进行编译连接，生成f.exe。这次可以通过连接，可以正确运行。



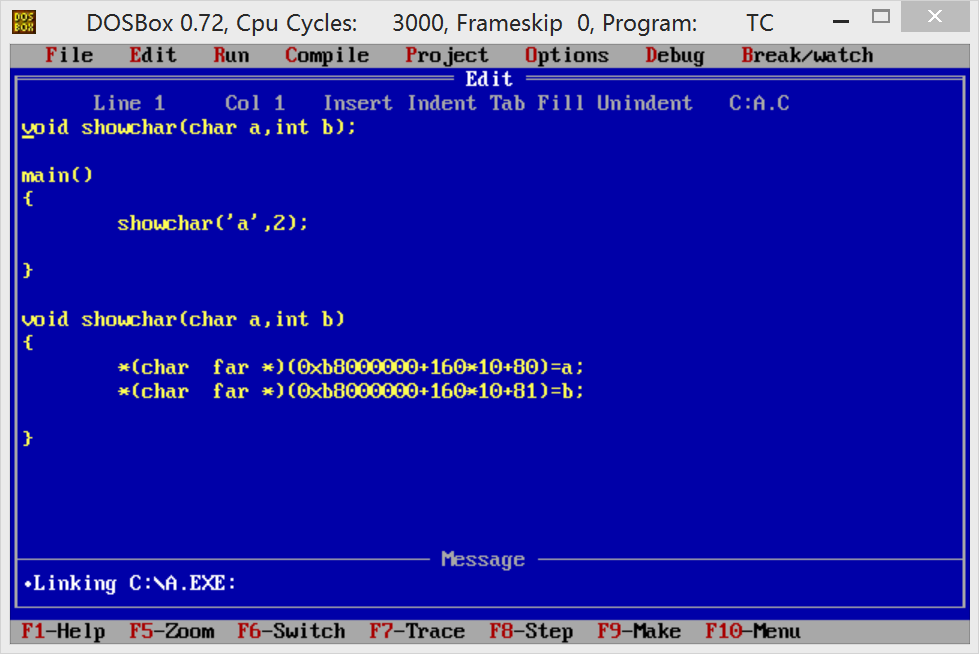


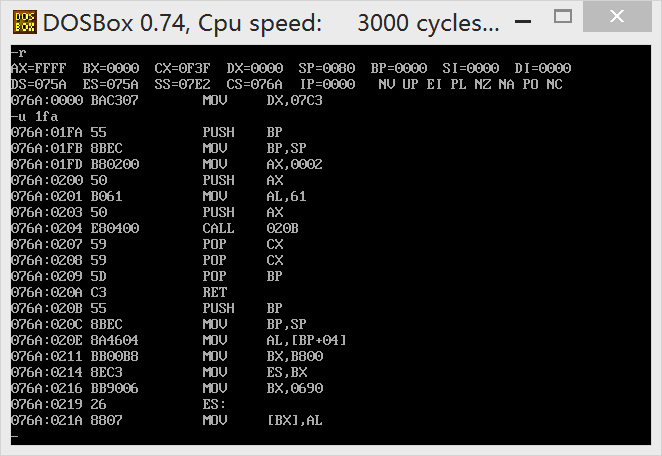
10>.



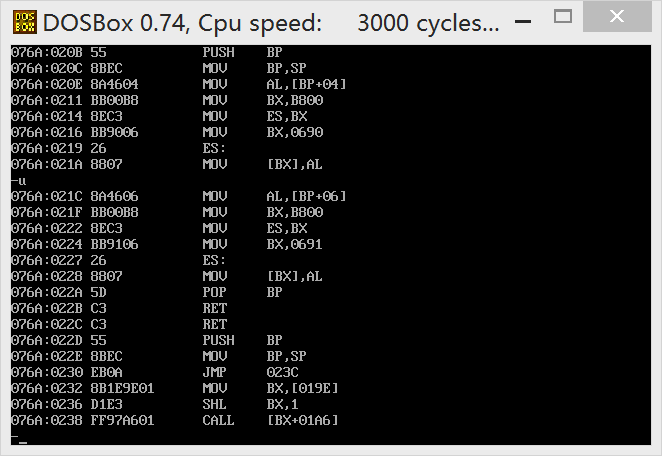
**综合研究五：**

1>.

****

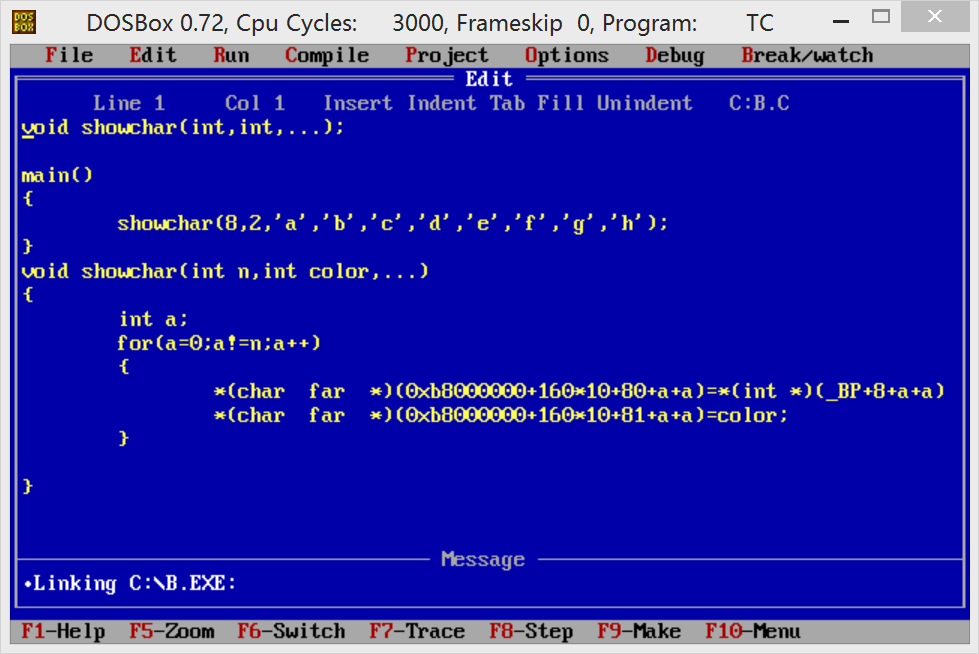
****

如图，main函数要传递两个参数‘a’和2，在汇编代码中是先将2赋给ax，再将ax入栈，然后将a赋给ax,将ax入栈。可见main()给showchar()传递参数是把要传递的值赋给ax，再将ax入栈，且如果有多个要传递的值，是由后往前将参数入栈。

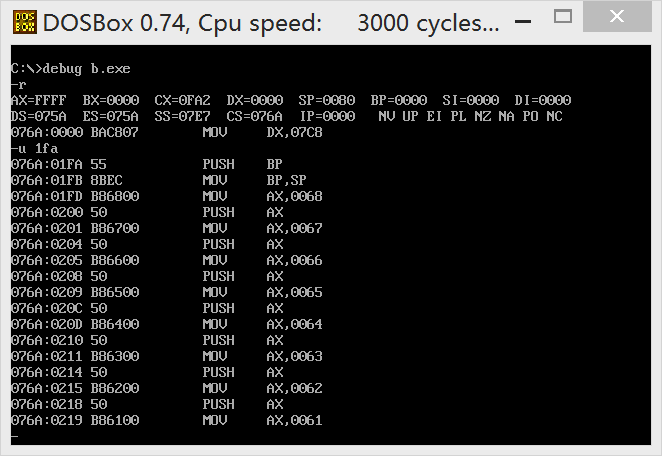
****

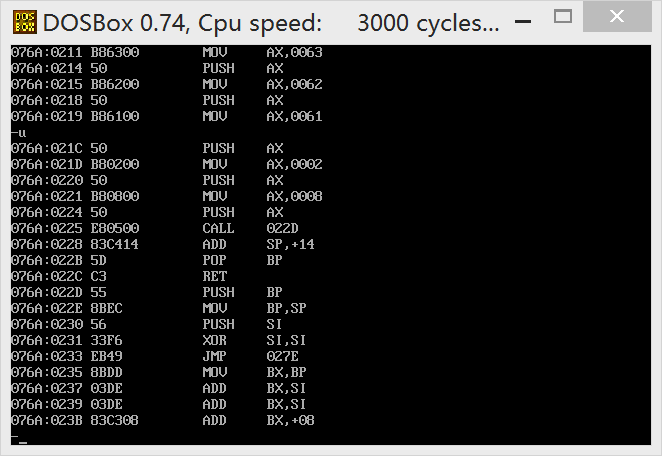
Showchar函数接收参数是将sp赋给bp，然后由bp+4找到栈中存储的参数a，由bp+6找到栈中存储的的参数b，为什么是bp+4和bp+6呢？因为程序在将两个参数入栈后，call指令将showchar的地址入栈占2个字节，在showchar中将bp入栈又占2个字节，所以要由bp+4找到第一个参数的地址。

2>.

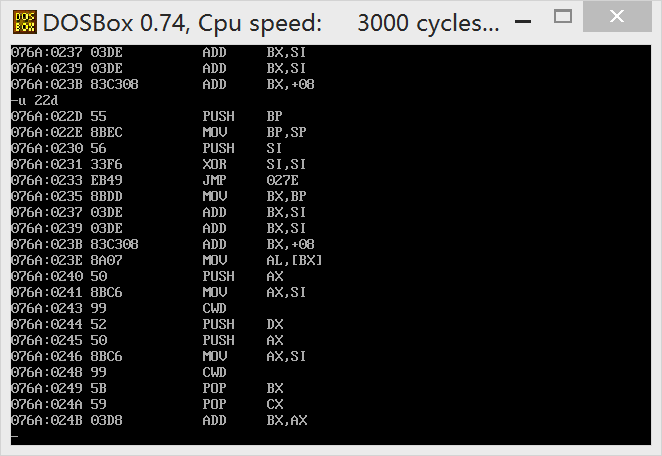
****

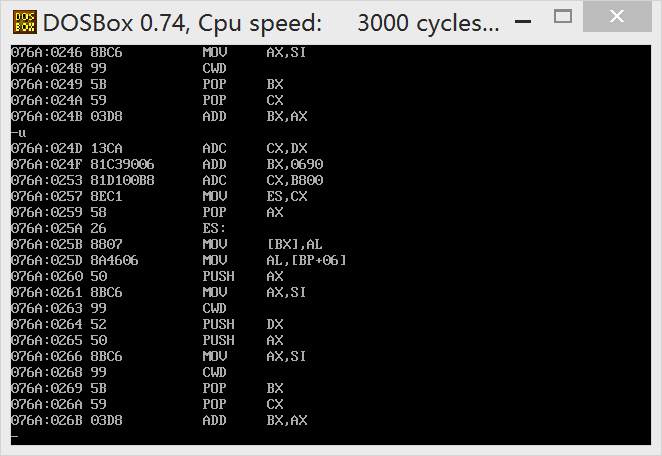
Main()部分：

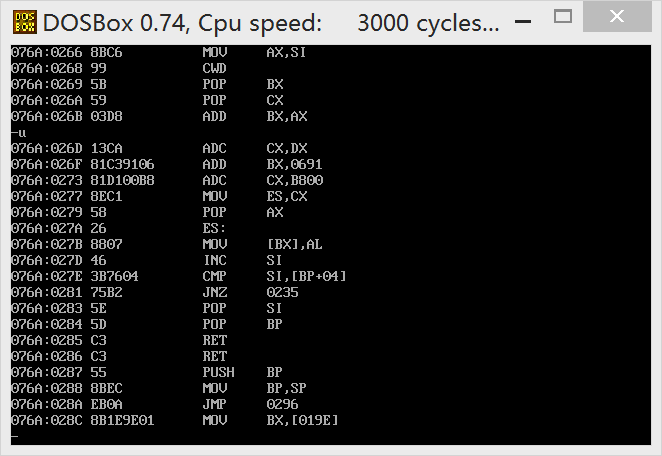
****

****

Showchar函数为：

****

****

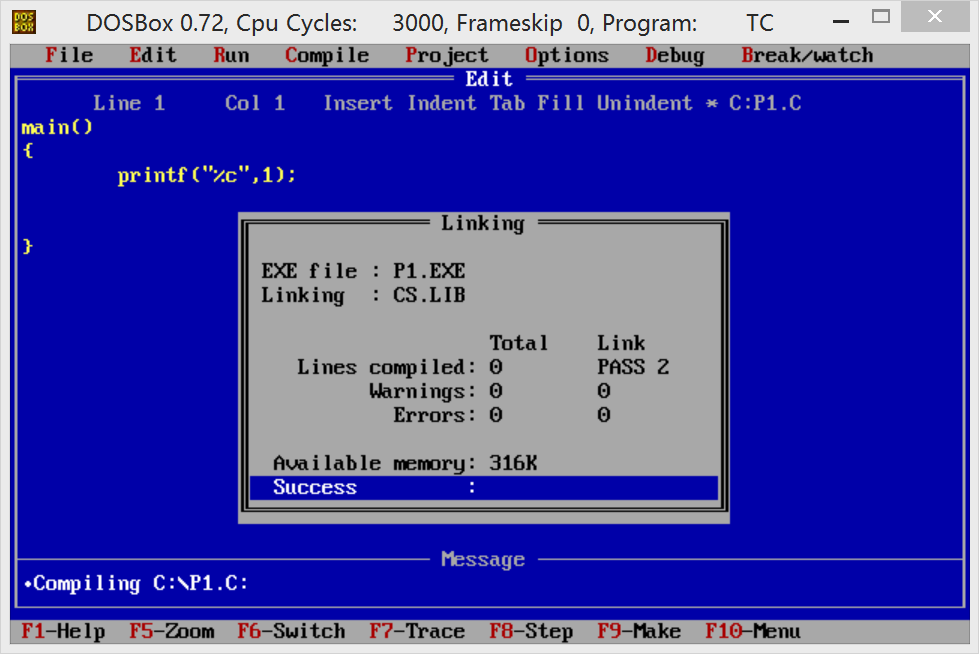
****

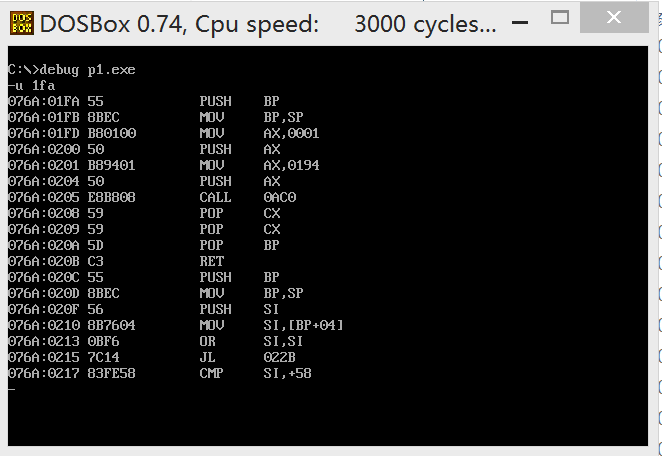
Showchar函数通过参数n来知道要显示多少个字符。然后通过循环来调用寄存器从栈中提取参数。

Printf()部分：

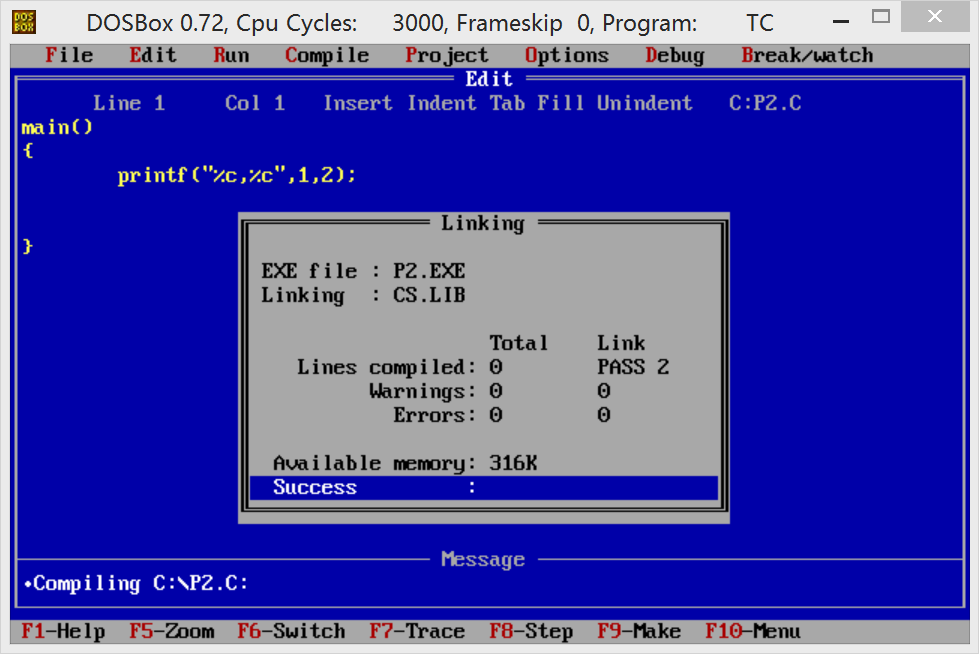
分别打印一个、两个、三个字符，分别反汇编看效果，观察规律。

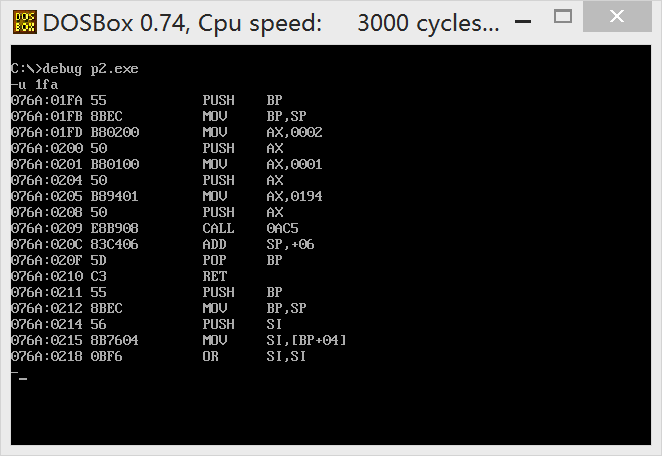
一个字符：



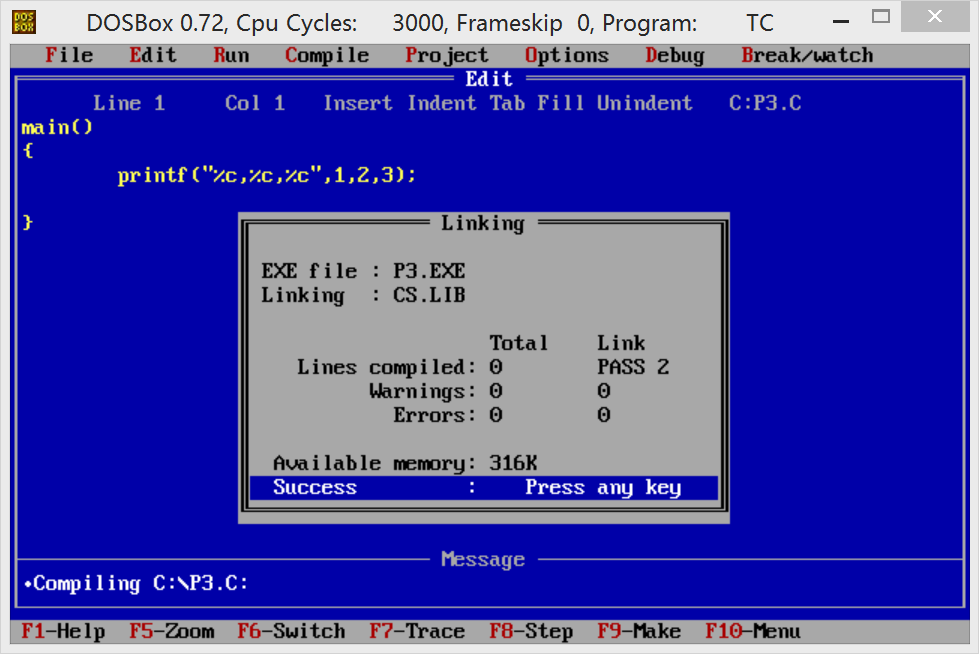
****

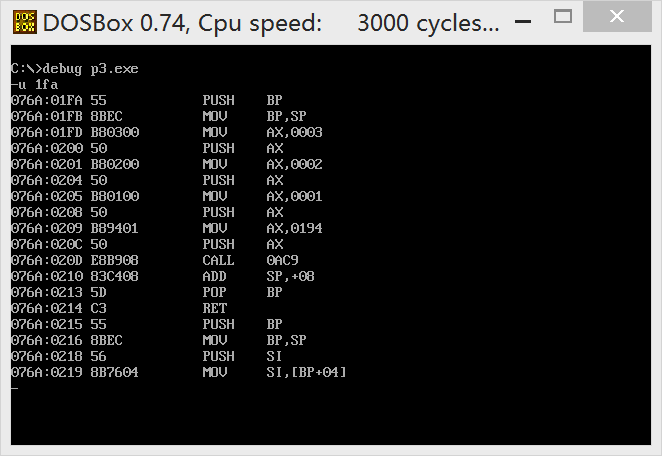
两个字符：





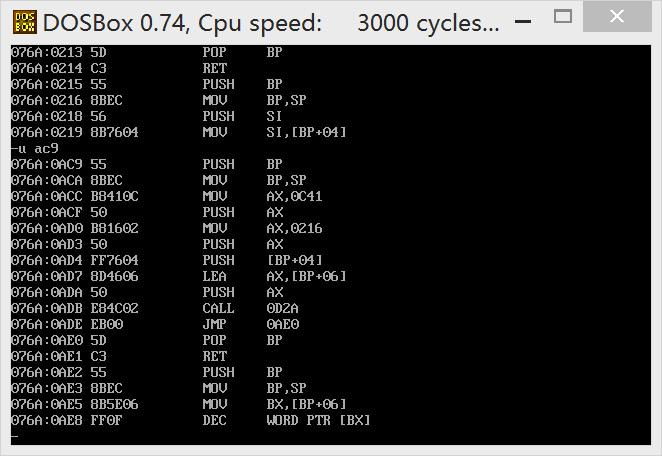
三个字符：

****

****

到此，规律已经比较明显了。每个程序中除了动态的参数入栈规律之外，还有不变的“0194H”和动态的“call 0ac0(0ac5/0ac9)”。

选择其中一个0ac9处代码进行查看，如下：



结合上网搜寻得到的答案，程序将%c和%d等符号放在偏移地址0194处，结尾加0，通过统计该地址处的%个数来确定要输出的字符数量。

由此，可以尝试在showchar()的基础上结合找到的规律进行编写新的printf()。

1. **已思考研究并已解决问题汇总**

**综合研究前言：**

以下地址，本意是指内存地址。不然实在要说的话，那什么东西肯定都有地址喽，只要它依赖了硬件，我们看得见摸得着检测得到，没有地址才有鬼喽，现在我是只比较一下最常用法的不同。

**1>.变量有名字吗？变量有地址吗？**

变量有变量名；变量有地址。

**2>.寄存器有名字吗？寄存器有地址吗？**

寄存器有寄存器名；寄存器没有地址（此处切记与arm单片机等区分开，现在谈论一下单纯的C语言）。

**3>.内存有名字吗？内存有地址吗？**

内存没有名字；内存有内存地址。

**4>.端口有名字吗？端口有地址吗？**

就目前看到的，好像常用的就是地址；名字？有吗？存疑。

**综合研究一：**

**1>.生成“TC.EXE”文件此过程中搬运了“C0S.OBJ、CS.LIB、EMU.LIB、GRAPHICS.LIB、MATHS.LIB”,各自是啥作用？**

“每一个库文件都相当于一个小模块，支持一种扩展。其实，一个lib文件是obj文件的集合。当然，其中还夹杂着其他一些辅助信息，目的是为了让编译器能够准确找到对应的obj文件。我们可以通过tlib.exe（在tc2.0下的根目录）来对lib文件进行操作，你可以把自己生成的obj文件通过tlib命令加入到一个lib文件中，也可以把lib文件内的obj文件进行删除操作，还可以把内部的obj文件给提取出来。明白了lib文件的大致结构以及对它的具体操作，在学习C语言的过程中，就会又多了一个切入点对C语言具体实现进行研究。”

经过上网查找，初步找到的各个文件的作用如下，待亲自佐证：

**C0s.obj**文件里包含了main函数的定义，找不到这个文件，就不能连接；

**TC.EXE** 集成开发环境

**C0?.OBJ** 不同模式启动代码

**C?.LIB** 不同模式运行库

**GRAPHICS.LIB** 图形库

**EMU.LIB** 8087仿真库

**MATHS.LIB**数学函数库

其中上面的?分别为:

**T** Tiny(微型模式)

**S** Small(小模式)

**C** Compact(紧凑模式)

**M** Medium(中型模式)

**L** Large(大模式)

**H** Huge(巨大模式)

几点解释:

（1）一般我们常使用的是TC.exe，实际上，Turbo C包有两种编译器，集成开发环境下的叫做TC.exe和命令行方式下的图形界面运行环境叫做TCC.exe（需要额外的graphics.lib文件提供支持）

（2）此程序中调用了printf函数，大家都知道调用C语言的库函数需要添加头文件，这里却不添加也可以，因为在 cs.lib中已经添加了printf函数的具体实现。

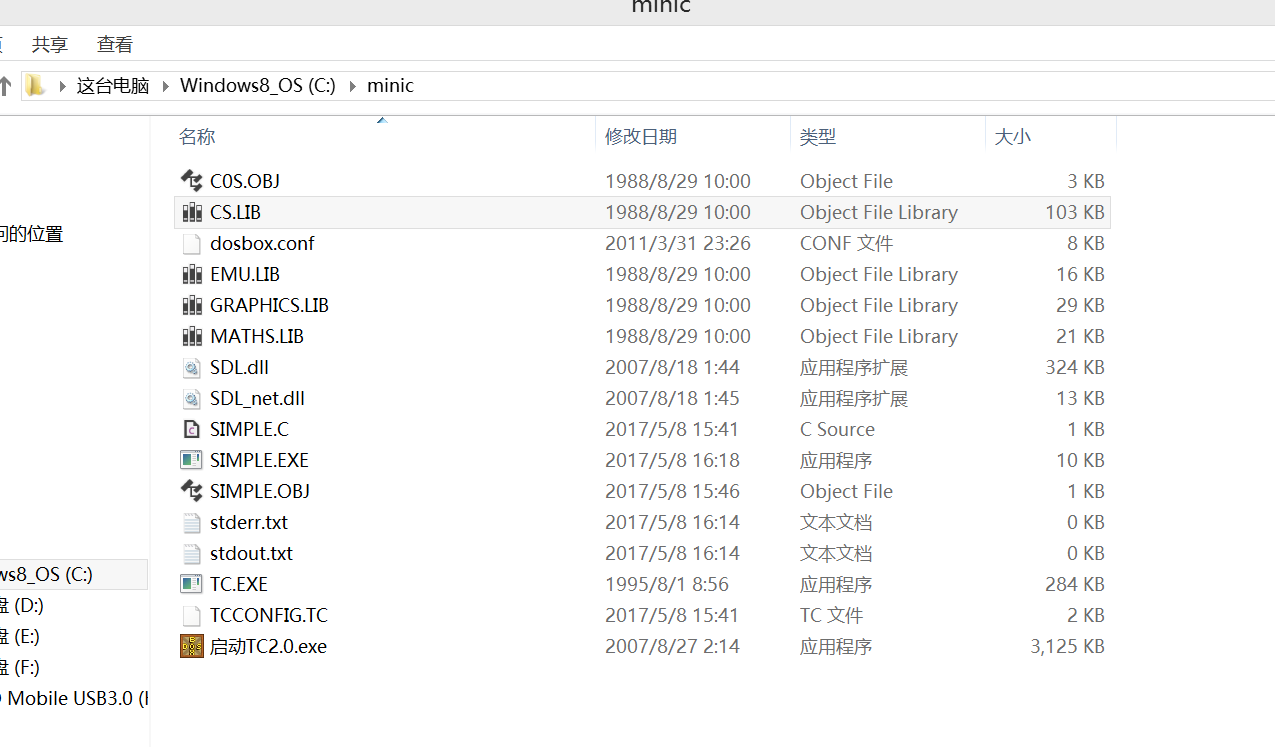
C 语言中提供了6种编译模式，这6种模式是：

微模式（Tiny），小模式（Small），中模式（Medium），紧凑模式（Compact），大模式（Large）和巨模式（Huge）。它们之间的关系如下图所示。用户可以按照自己的程序大小及需要进行选择。  
　　　　　　│ 小程序　　 │ 大程序  
　　━━━━┿━━━━━━┿━━━━━━━━  
　　 小数据 │ 微，小　　 │ 中  
　　 大数据 │ 紧凑　　　 │ 大，巨

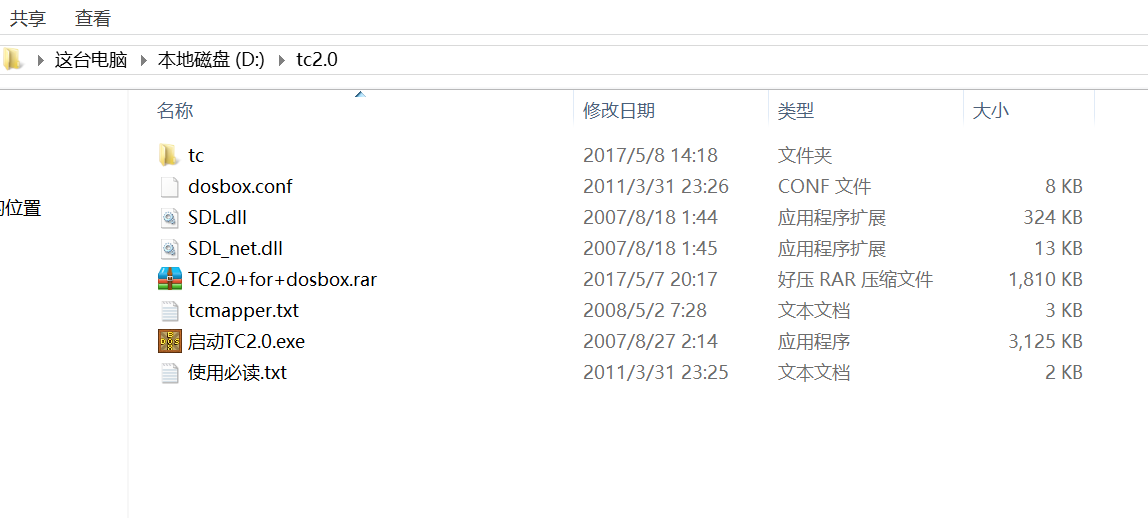
所谓小程序就是指程序只有一个程序段，大小不超过64KB，缺省的码（函数）指针是near（近程指针）。所谓大程序就是指程序只有多个程序段，每个程序段不超过64KB，但总程序量可超过64KB，缺省的码指针是far（远程指针）。小数据就是指数据只有一个数据段，缺省的数据指针是near。大数据就是指数据有多个数据段，缺省的数据指针是far。

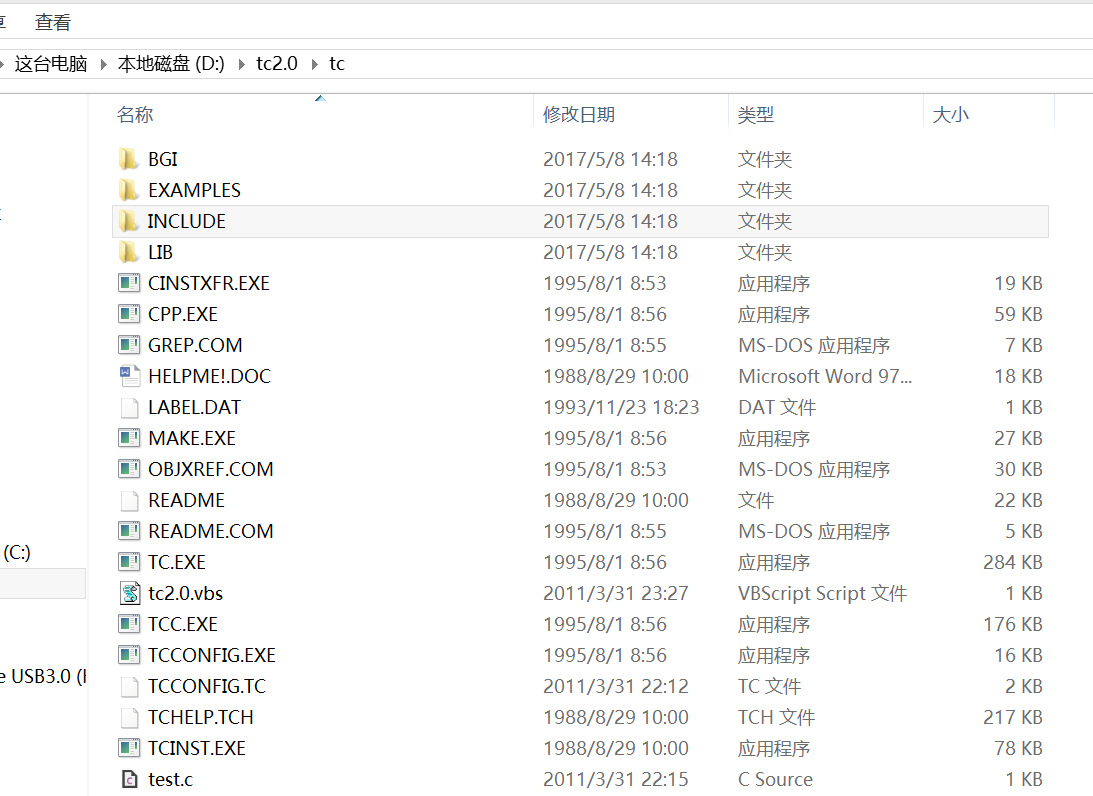
**2>. 程序在编译运行时，源码所需要的库（静态库和动态库）及头文件编译器是去哪找的？（库及头文件的查找）**

通过本实验过程，比较**C:\minic** 与**D:\tc2.0**：

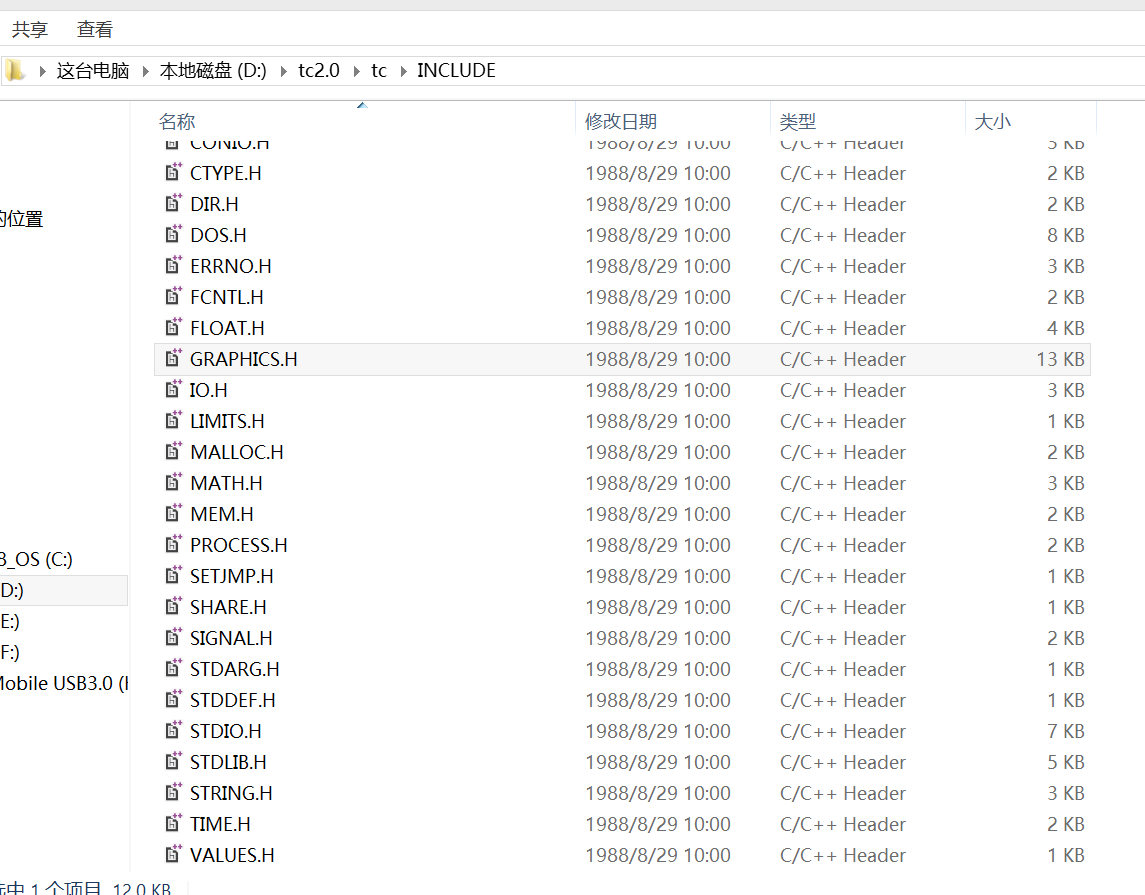


在**C:\minic**中，各个拷贝过来的\*.obj、\*.lib文件都是相对于simple.c、simple.obj文件而言，处于同一目录下，这是一种情况，此情况经过验证可以使simple.exe最终生成，并正确执行。

****

****

在**D:\tc2.0\tc**中，有INCLUDE、LIB文件夹等，本次实验所拷贝文件均出自LIB文件夹，而此次没有用到的INCLUDE文件夹内，打开之后，可以看到多个头文件，如图：

****

很明显，这又是另一种方式；

通过上网查找，发现还有一种，如下：

在turboc.CFG中可以指定tcc可以用来搜索的库文件的位置。但是用TC2.0修改路径不会保存在turboc.CFG中，而是生成另一个配置文件。（待佐证，可排到未解决问题中）

**综合研究二：**

**1>.用C语言编程可以不用变量吗？**

可以，综合研究二得出其中一种额外选择：可以使用寄存器替换变量的使用。

**2>.函数名、变量，这些区别于汇编的新生名词，在C语言反汇编为汇编语言的时候又是具体以怎样的熟悉的形式展开的呢？**

相对于函数，很容易就想到了子程序，包括子程序的标号、调用、返回等都与函数感觉有某种近亲关系一样。变量的话，存在的形式，还不确定就是唯一的形式。

**3>.** **用debug怎么查找函数的偏移地址？**

参考语句“printf(“%x\n”,main);”，最好能举一反三。

**综合研究三：**

**1>.** **C语言将全局变量存放在哪里？将局部变量存放在哪里？每个函数开头的“push bp；mov bp，sp。有何含义？”**

全局变量是存放在内存空间的指定内存地址处，而局部变量是存放在栈中。

因为要用bp记录sp原来的位置，以便函数结束时将sp还原，所以要对bp进行保护，所以会有push bp;mov bp,sp。

**2>.** **C语言将函数的返回值放在哪里？**

由“mov [BP-2],AX”,可知这个程序中将返回值存放在寄存器AX中。

**综合研究四：**

**1>.c0s.obj文件的作用**

c0s.obj文件的作用：在程序开始运行，进行相关初始化，再调用main函数，返回后进行相关的资源释放，环境恢复等工作，再将程序返回。

**2>.可以不用main函数编程吗？**

可以。只不过需要修改相应的原来在程序开始运行时，负责进行相关初始化操作、调用main函数、返回后进行相关的资源释放、环境恢复等工作的文件。在本次研究中用到的是c0s.obj文件，同样的道理，以后的话要学会举一反三。

**3>.文件的大小为何会显示两种结果，大小相差会如此巨大？**

在这里也查看了一下c0s.obj文件的属性：



在连接的过程中是将程序用到的目标文件都连接形成一个可执行文件的过程，大小自然就会变大了。

**综合研究五：**

**1>.Main函数是如何给showchar传递参数的？showchar是如何接受参数的？**

main函数要传递两个参数‘a’和2，在汇编代码中是先将2赋给ax，再将ax入栈，然后将a赋给ax,将ax入栈。可见main()给showchar()传递参数是把要传递的值赋给ax，再将ax入栈，且如果有多个要传递的值，是由后往前将参数入栈。

Showchar函数接收参数是将sp赋给bp，然后由bp+4找到栈中存储的参数a，由bp+6找到栈中存储的的参数b，为什么是bp+4和bp+6呢？因为程序在将两个参数入栈后，call指令将showchar的地址入栈占2个字节，在showchar中将bp入栈又占2个字节，所以要由bp+4找到第一个参数的地址。

**2>.printf()的参数个数的规律。**

由上述过程可以得出。

1. **已思考研究并未解决问题汇总**
2. **研究感想（心得体会）**

**综合研究前言：**

看过了书之后，感觉老师首先想要再次提醒我们的就是万变不离其宗：8086汇编学到最后了，我们对数据的存放，也应该再回到最初的起点，对数据的操作，无非还是找到一个合适的地址，并匹配合适的大小，再把它“踹进去”！包括之后几个综合研究一定也会如综合研究一一样，有许许多多的小细节等着我们去发掘。跟着学长学习“三个一”，我只希望我以后可以尽可能地犯错，不在学校里犯大量的错，以后就可能出去吃大量的亏。不过同样重要的是，要加强自我纠错的能力。希望接下来的综合研究自己可以进步。

**综合研究一：**

以前不懂什么是“一花一世界”，现在应该也不算懂，不过好在了解了许多。王爽老师的三个一课程设计的题目，让人做了之后总是很有启发，有一种醍醐灌顶的感觉，比对一下学校内的老师授课方式以及我个人不太理解的排课方式，高下立判。“师者，所以传道授业解惑也”，如果大部分老师都能摒弃教给学生的那些似是而非的知识，一步脚印一个坑，大胆的把“计算机是什么？编程怎么玩”讲得“一丝不挂”。那么，该是一件多么好的事！

**综合研究二：**

不要主观臆断，不要主观臆断！本来自以为的直接”u”一下就可以的事情，结果却不是，如果没有书里一步步给的推测线索，自己可以找到这样简洁有效的方法来检测吗？如果不行，自己到底还是哪里短缺呢？是对错误的敏感不足，还是缺少一步步推敲、总结、反思的能力与意识呢？再想想，printf()用的少吗？可是它的功能自己是不是都玩转了，printf()用起来简单，难道printf()本身真的很简单吗？那要看综合研究五了。

**综合研究三：**

有时候自己以为对一件东西了解了，可是真正再次接触的时候，又会让人觉得“这绝对是个新玩意儿，我不认识它”。当我们学了汇编，再学C语言，反汇编过来的汇编语言，结构还是原来的结构，但是套路就是“C”的套路了，之前编写汇编代码，自己有自己编写方式，但是现在不同了，要接触C，现在的学习方法是猜”C”这个人编写汇编时候的风格方式是如何的，然后再亲自看“C”这个人编写的汇编代码实际到底是咋样的，而且，这个“C”绝对是我的老师，比我要对的多得多，我还得抱着敬佩加好好学习的态度。真的是不服不行啊。

**综合研究四：**

这次的研究实验做的感受就是“又从空中飘回地面”。这回王爽老师让我们一步步朝着“陷阱”走下去。走到底，再想一下。对啊，我们修改的可是c0s.obj文件，这个文件可是tc自带的，算一个编译器商品中比较重要的一小部分了。可是现在呢，我们修改了它，的确是用的我们学习的汇编知识，当然很多是书上的引导，也就是所谓的“陷阱”了。这就有起码好几个启发点：1.这是个编译器，现在你也算对它稍有了解了，继续下去，自己DIY一个编译器，哇，成就感。2.这次我们是修改了c0s.obj文件，因为他让我们不爽了，它竟然必须让我们用main函数编程，我们就不用，偏要做叛逆的小孩。人总要经历叛逆期的，这样看来，程序员当然也要，不然还怎么学会“成长”呢？

**综合研究五：**

找规律，一件很重要的事情。而善于使用变量，并将变量灵活运用于建造规律公式的过程是十分重要的。以上两者，都是加强编程的必由之路。

1. **宣讲会附加问题**

**1>.头文件的作用和生成的四个阶段？**

1. 预处理（include这个库寻找定义头文件的内容插入源程序中生成.i文件）
2. 编译(将.i文件翻译成生成.s后缀的汇编语言程序)
3. 汇编 （将.s文件翻译成机器语言，生成可重定位的目标程序，打包生成.o文件）
4. 连接（将各种代码和数据整合成一个单一可执行文件）

**2>.为什么是实验中所写的源程序不需要stdio.h这个头文件呢？**

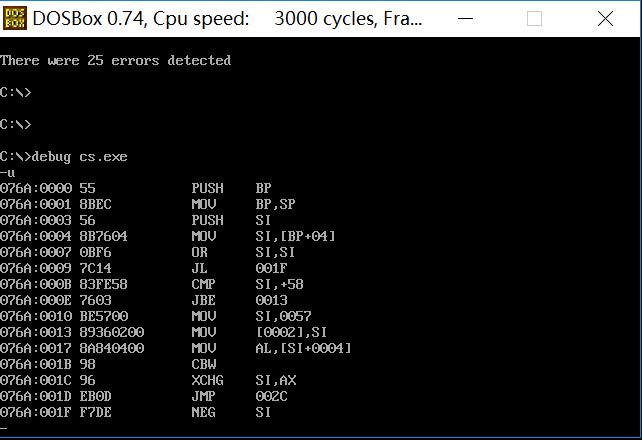
此程序中调用了printf函数，大家都知道调用C语言的库函数需要添加头文件，这里却不添加也可以，因为在 cs.lib中已经添加了printf函数的具体实现。

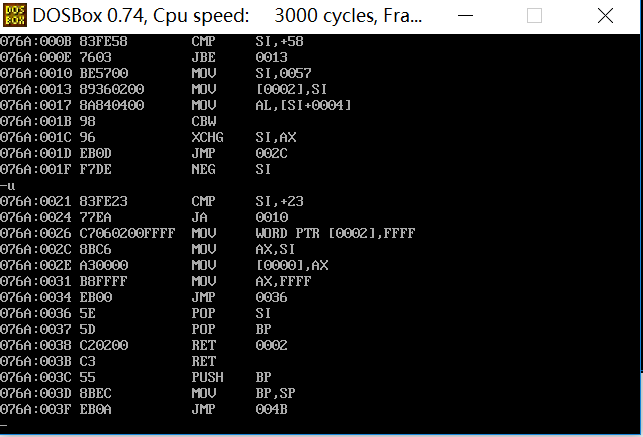
需要验证，但是\*.lib文件是不对外公开的，除非有专门的LIB文件查看工具，否则不能查看LIB文件中对函数的具体实现过程。

而查看lib文件有几种方式：

1. 通过msdn查看使用的\*.lib文件包含的函数名，来查找其对应的头文件，头文件里面有整个lib的函数声明；
2. 查看vc或者其他工具安装目录下的src目录，查看函数的代码；
3. 查看\*.lib文件和\*.dll文件都可以通过OLLYDBG中LOADDLL插件来反汇编查看各个函数的过程。

此处我先直接用link.exe 试图连接cs.lib 文件，报了25个错，但是生成了cs.exe ，debug反汇编查看一下，如图：





但是没看出来，看来还是需要msdn的辅助。

**3>.有关于程序的的编译过程？**

虽然最终都会编译成机器码，但存在有的程序不必编译成汇编语言，c语言必须转变成汇编语言，因为C语言是汇编编译。之后会讲到“汇编器”相关知识。

**4>.Main与Main（）的区别？**

在用printf()打印main时，打印出的是main()的偏移地址；而在打印main()时，得到的数是main()的返回值。但是这个返回值的选取又涉及到之前函数将返回值保存在哪里的问题，由于原本编写的main()返回值保存在ax寄存器中当我们再编写一个函数，并且该函数并没有在作用过程中修改ax寄存器的值。我们得到的该函数的返回值仍然是上一次保存在ax寄存器中的值。

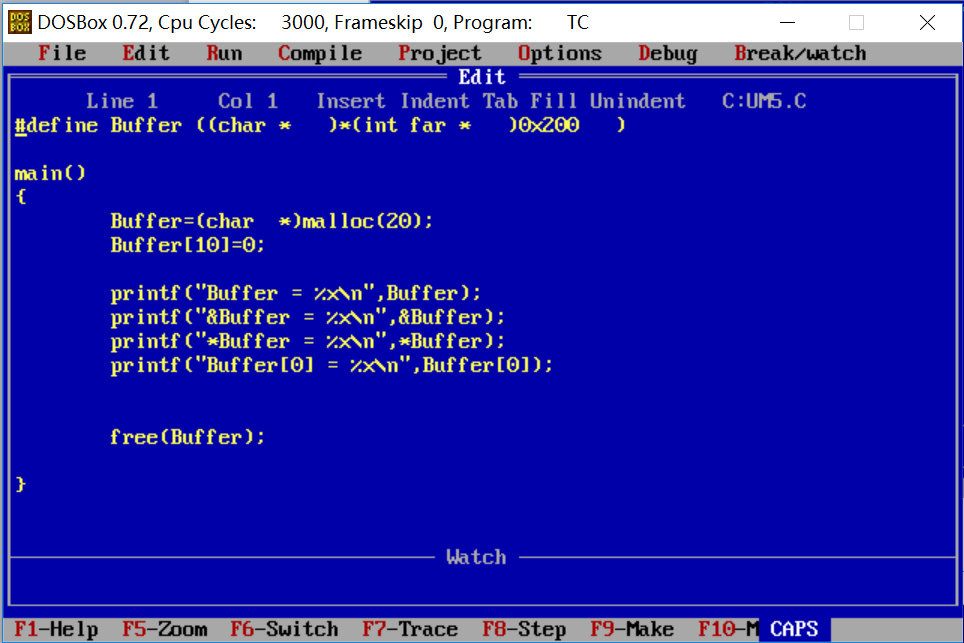
**5>.Sp、bp问题？**

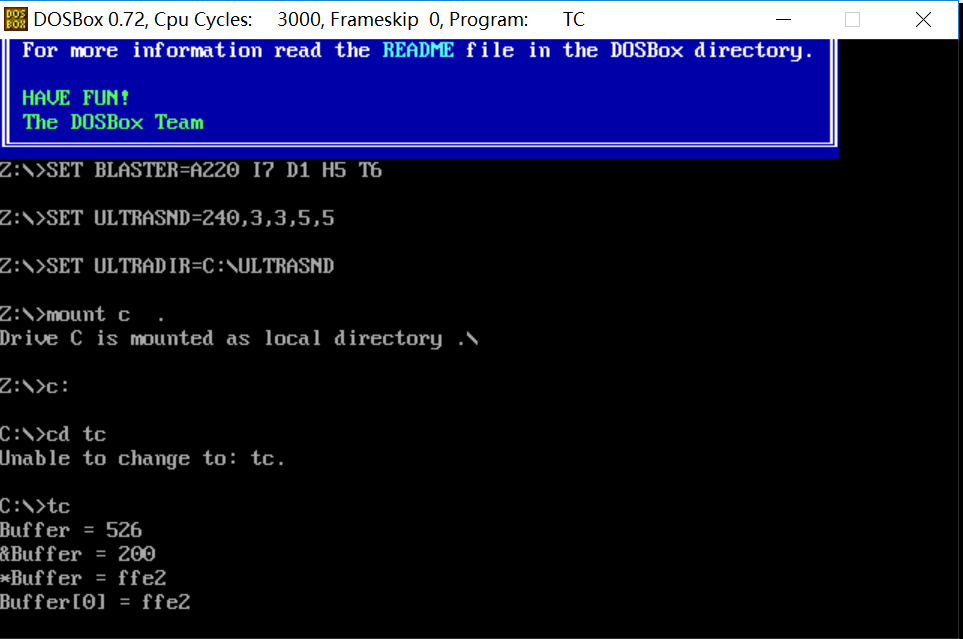
作用一：与SP联合使用，SP不能轻易更改，所以赋值给BP，用BP来实现我们的要求；

作用二：栈帧，相当于一个链表，bp,sp代表一个栈帧的序列，为了方便调用其他函数，返回之后恢复调用前状态。

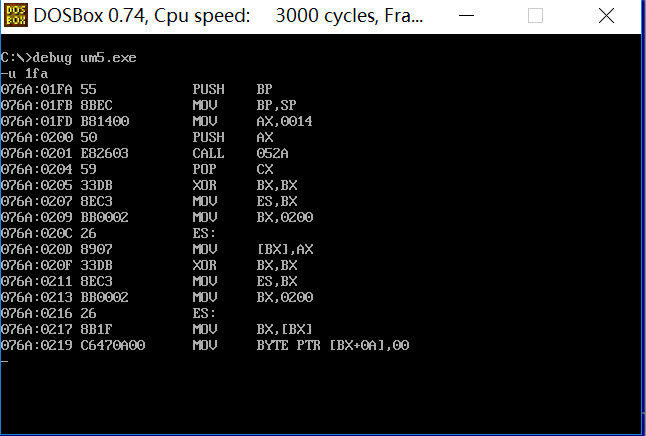
**6>.Buffer指针问题？**

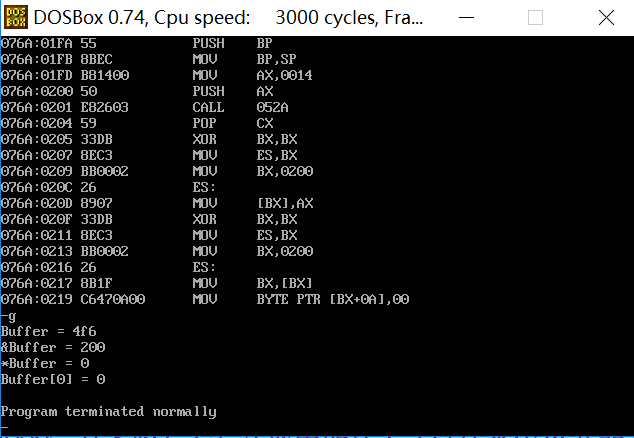
新编一个关于Buffer的函数，打印出各个值，如图：





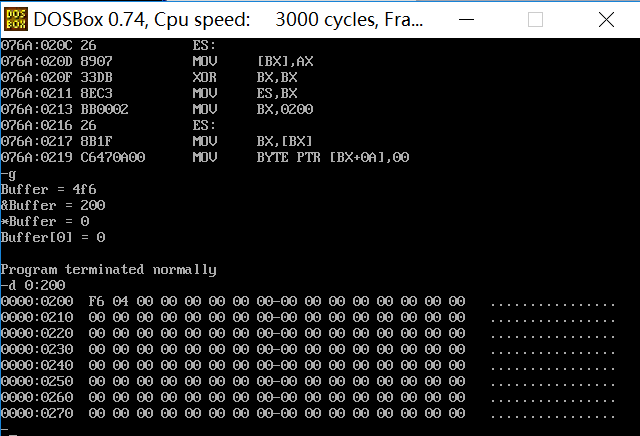
DEBUG看一下：





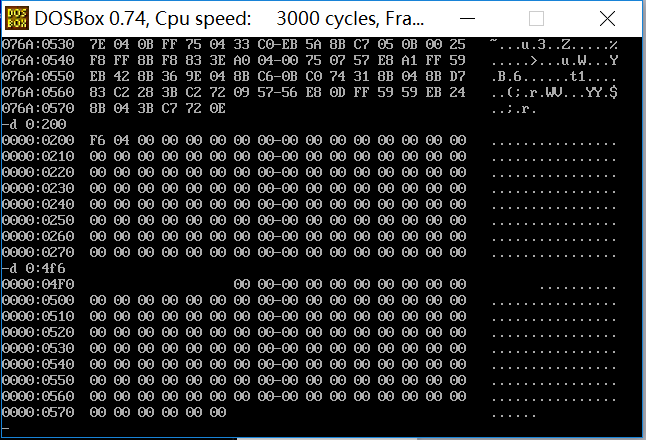
比较二者的结果，只有&Buffer的值是相同的，都是200.

再具体看一下：

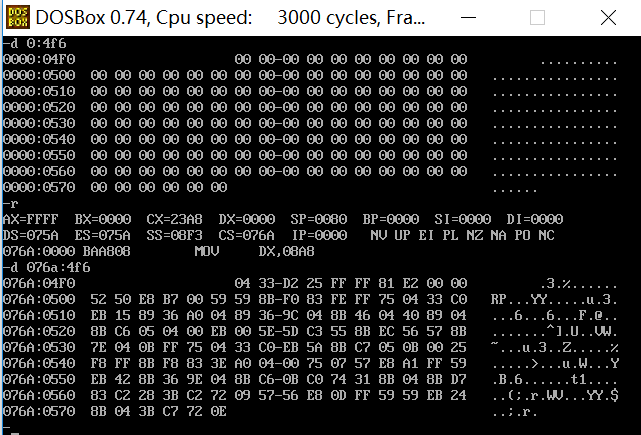


很好，出现了04f6h了，也就是对应的Buffer的值。

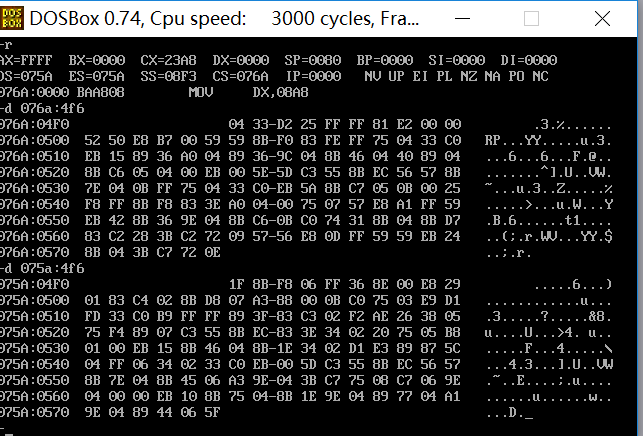
查看0：04f6h ：



查看cs(076a)：04f6 ：



查看 ds(075a)：04f6 ：



1. **拓展问题**

**1>.链接的时候，是否将\*.lib文件中的所有（包括必需以及非必需）的\*.obj文件链接生成最终的\*.exe文件？**

**2>.标号S问题。即如何写一个c0s.obj使编译器和连接器在编译、连接的时候可以一上来就调用main函数，而不用管main函数所在的地址是什么，都能打印出正确main函数地址（main函数的调用只与main本身有关，与main函数的地址无关）？**

**3>.“-32768”问题？**