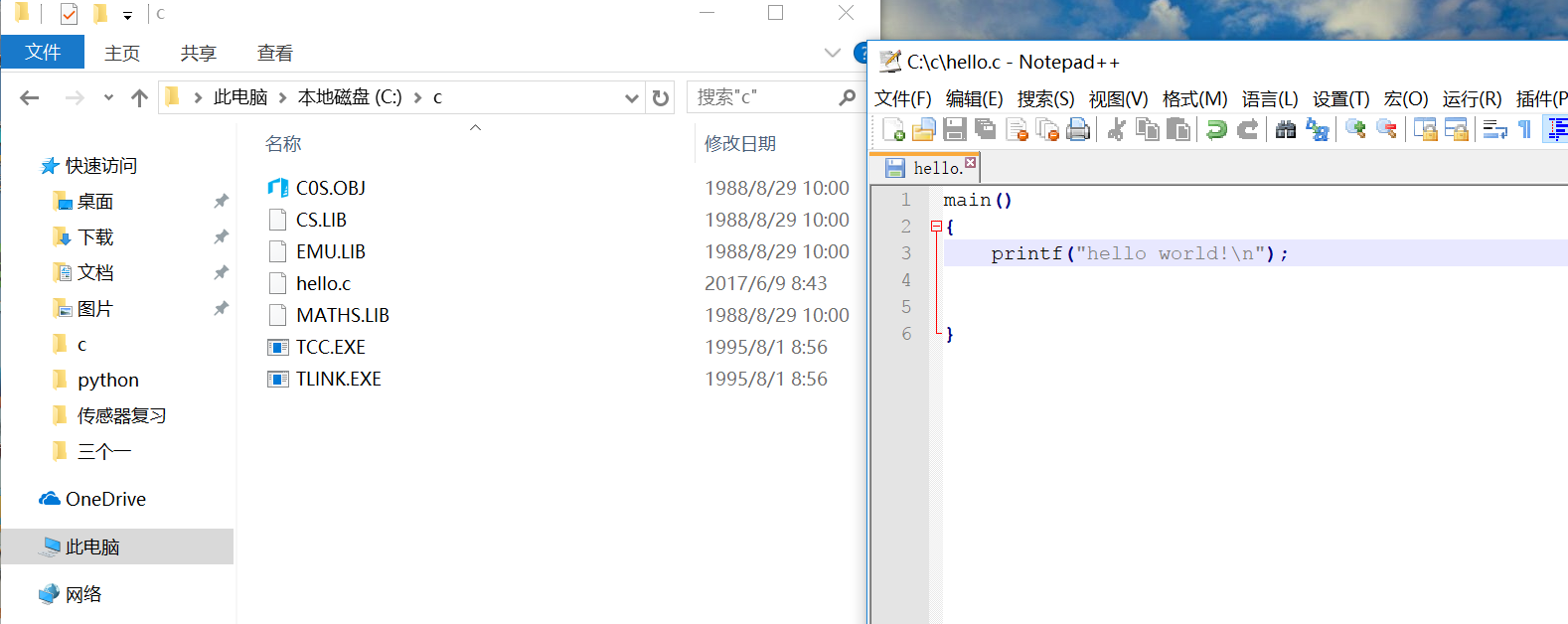
**c语言一二三宣讲会研究报告**

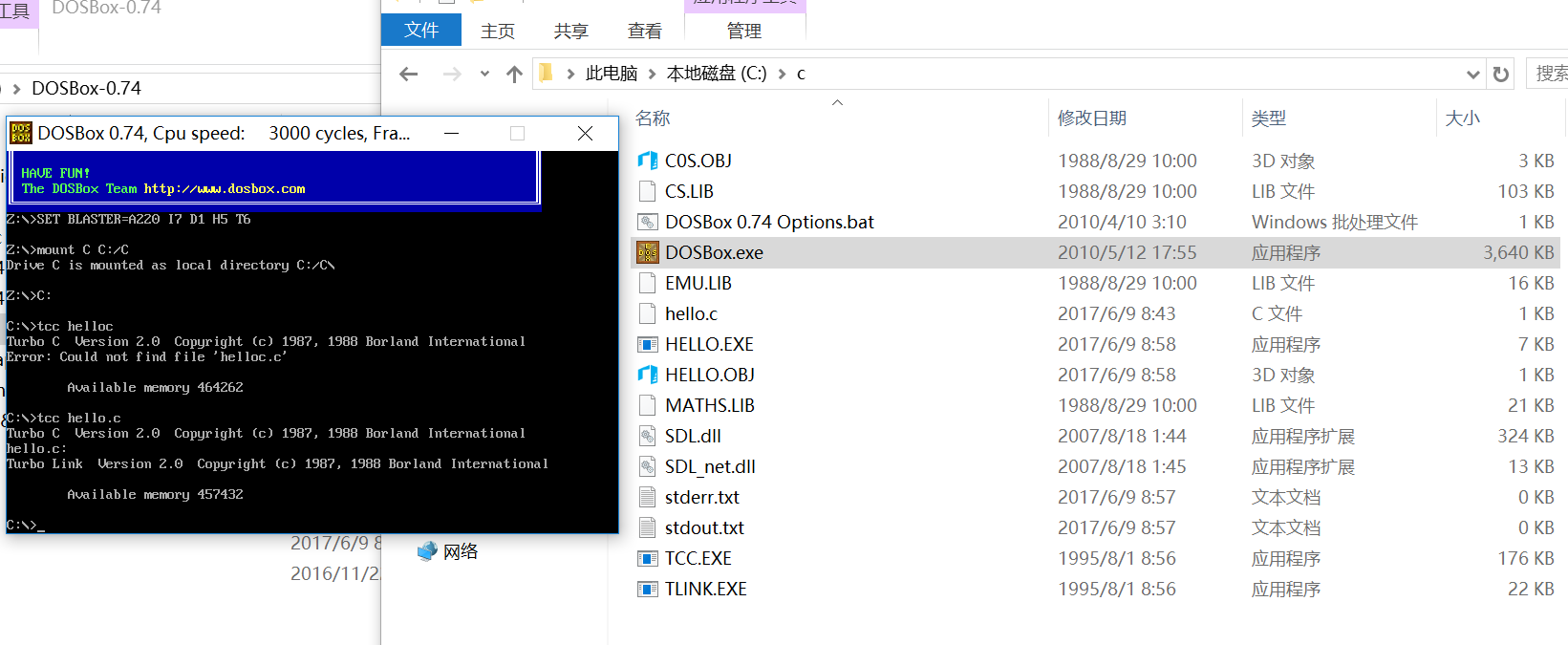
1. **研究过程展示**

**以前的：**

**一**



拷贝相关文件到c:\c目录下



使用tcc.exe成功编译链接hello.c文件并在c：\c目录下生成相应的hello.exe可执行程序。



成功显示“hello world！”

以上是第一次成功执行的情况。

下面开始自我验证。

书上说：

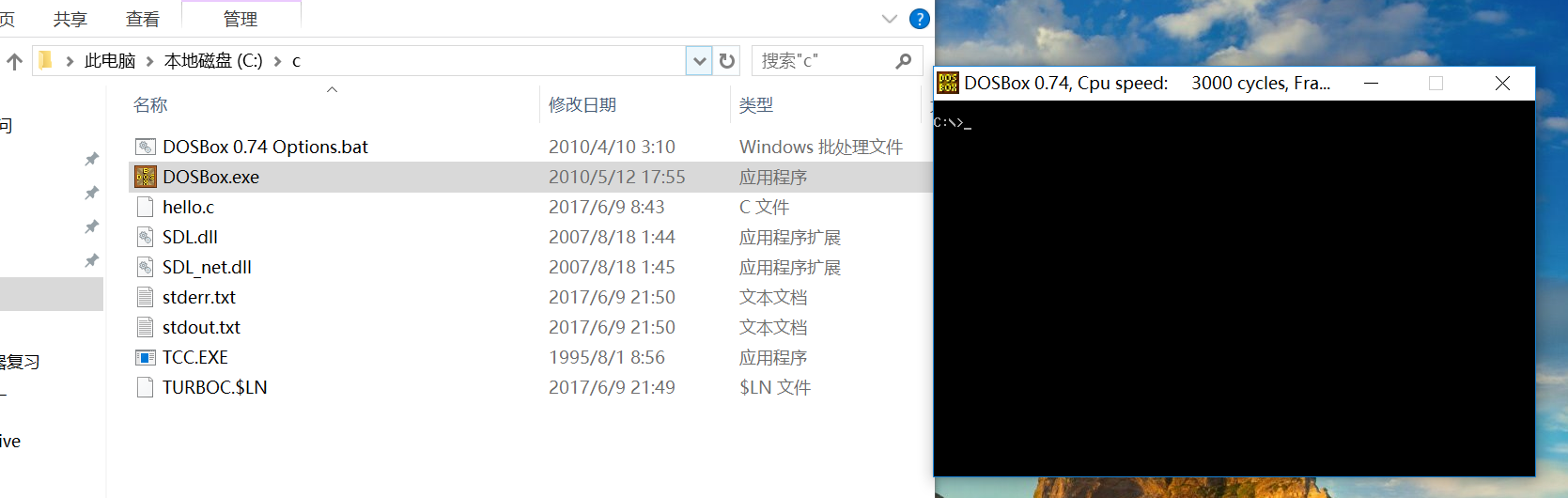
“1.tcc.exe将a.c编译为a.obj；

2.tlink 链接所需文件并最终生成相应的\*.exe文件；

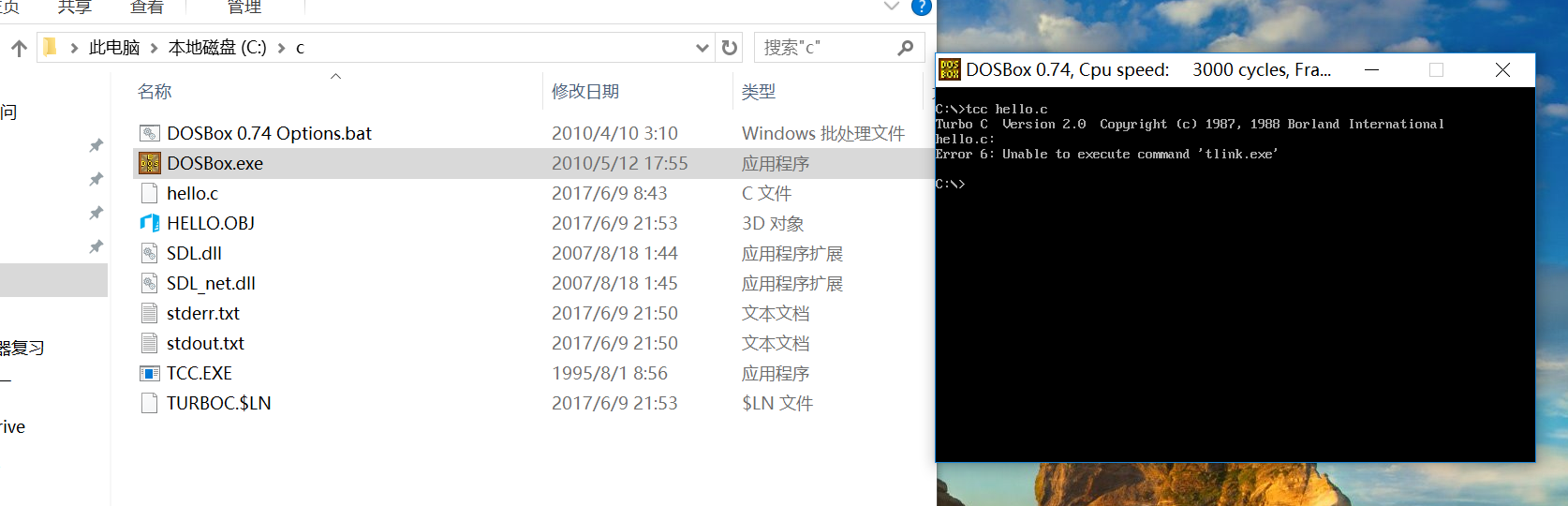
3.tcc.exe调用tlink.exe”

按顺序验证吧。

验证一过程：

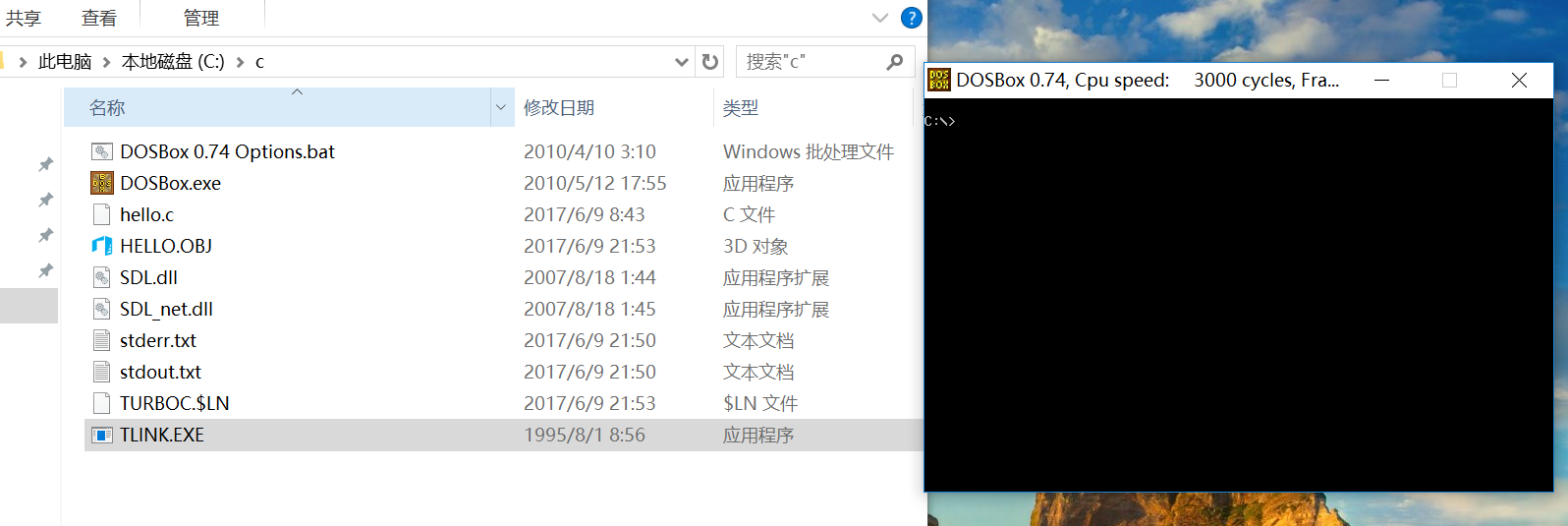


如图一开始，除了必要的配置文件之外，在c:\c目录下保留tcc.exe文件，移除原有的tlink.exe文件和一系列\*.lib和\*.obj文件。

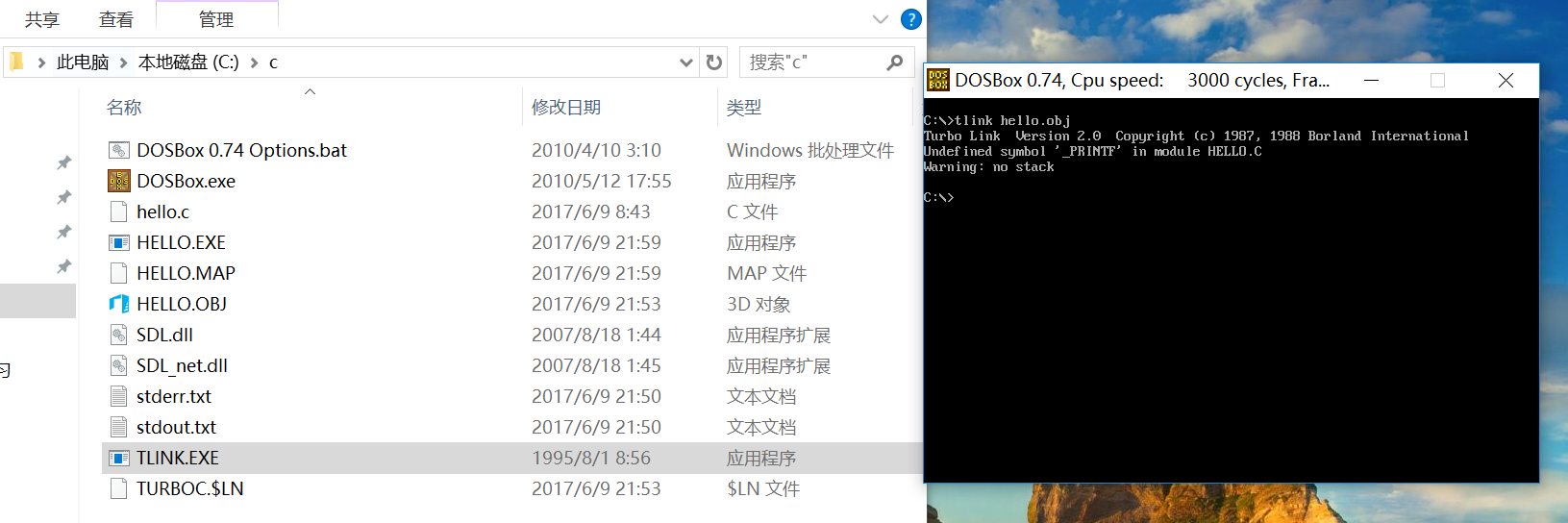


在我执行“tcc hello.c”命令之后，c:\c目录下成功生成了hello.obj文件，这一步，既验证了第一点，也再一次说明变异的时候并不需要\*.lib、\*.obj等文件。

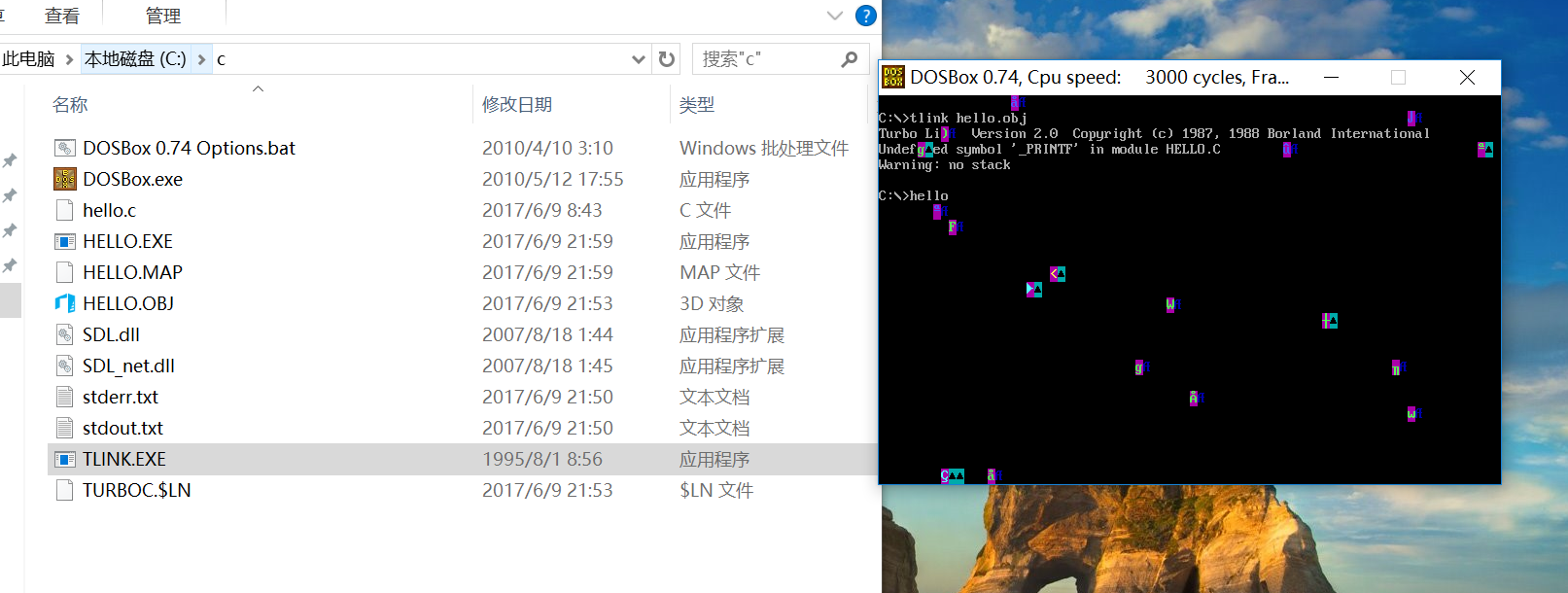
验证二过程：



一开始，只将验证一过程的tcc.exe文件由tlink.exe文件替换掉，直接tlink，观察结果：



可以看到，出人意料的，在没有一大串\*.lib、\*.obj文件的情况下，同样生成了相应的hello.exe文件，运行一下看看呢：

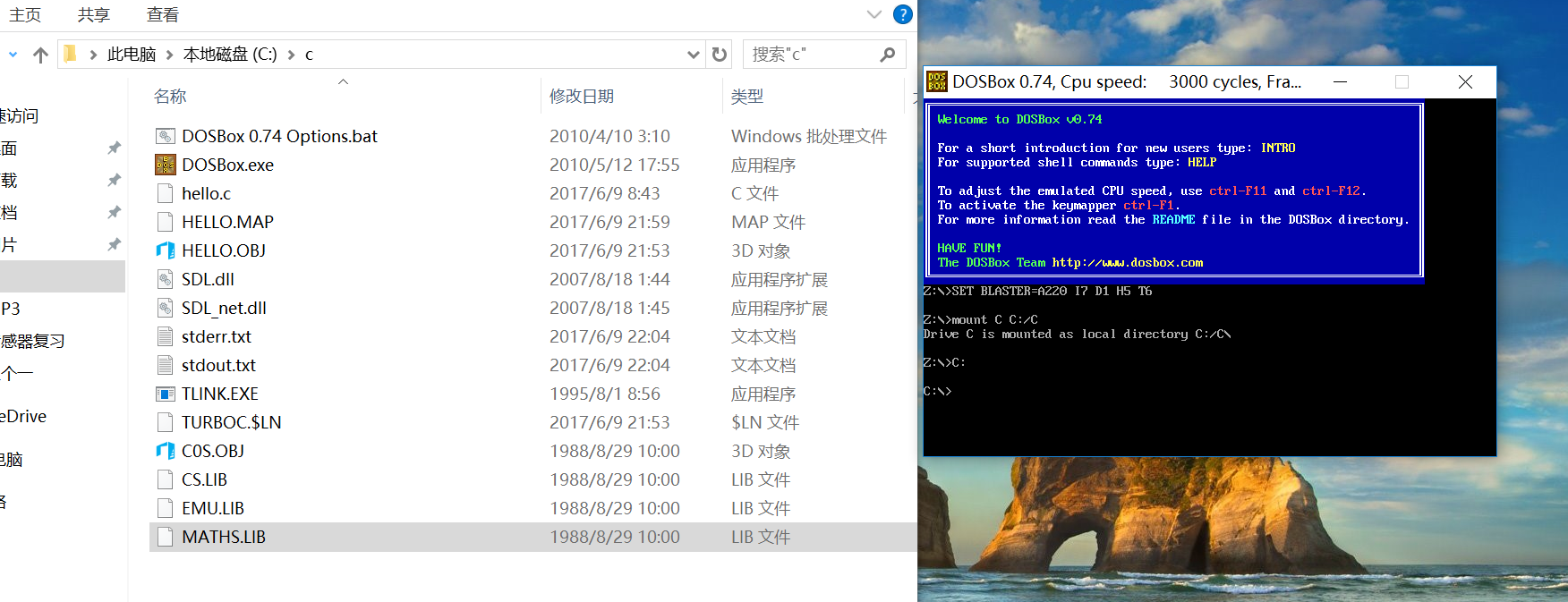


额，出现了“一闪一闪亮晶晶”的情况。

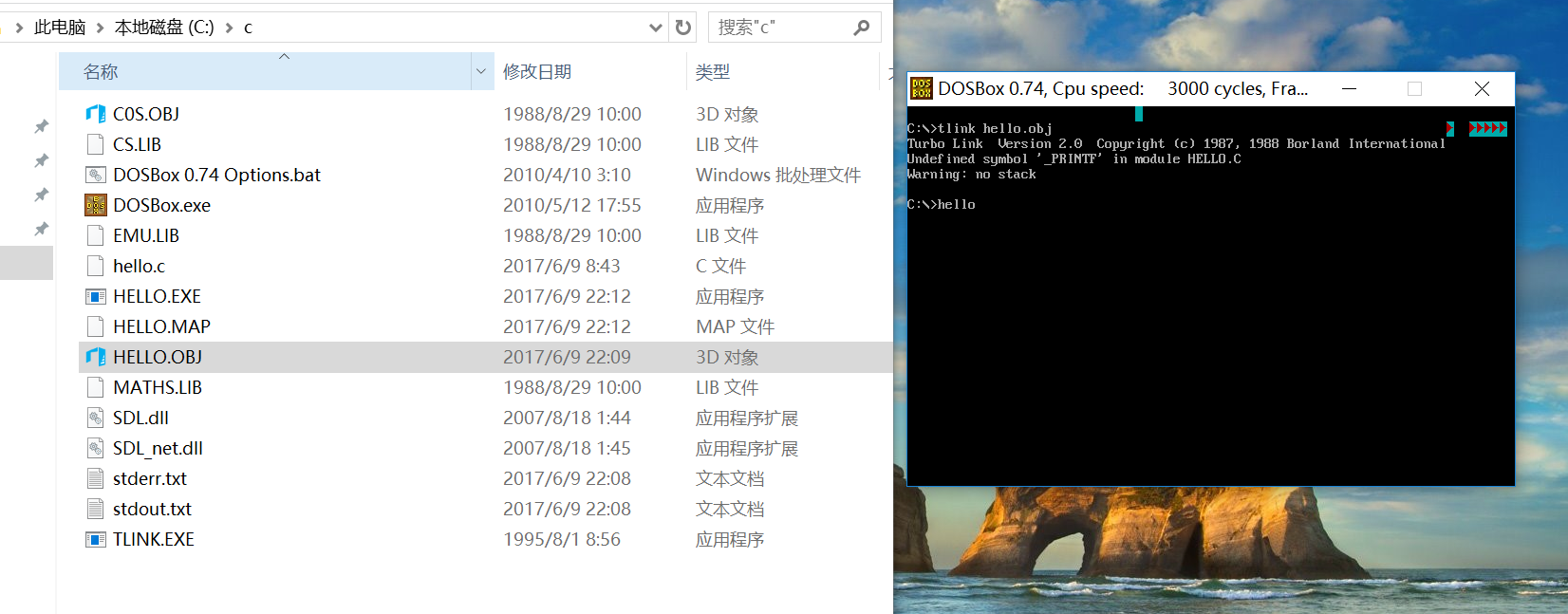
那么将一大串其他\*.lib文件、\*.obj文件放入之后再tlink呢（先将之前的hello.exe删掉）：

来个前后对比吧：

前：



后：



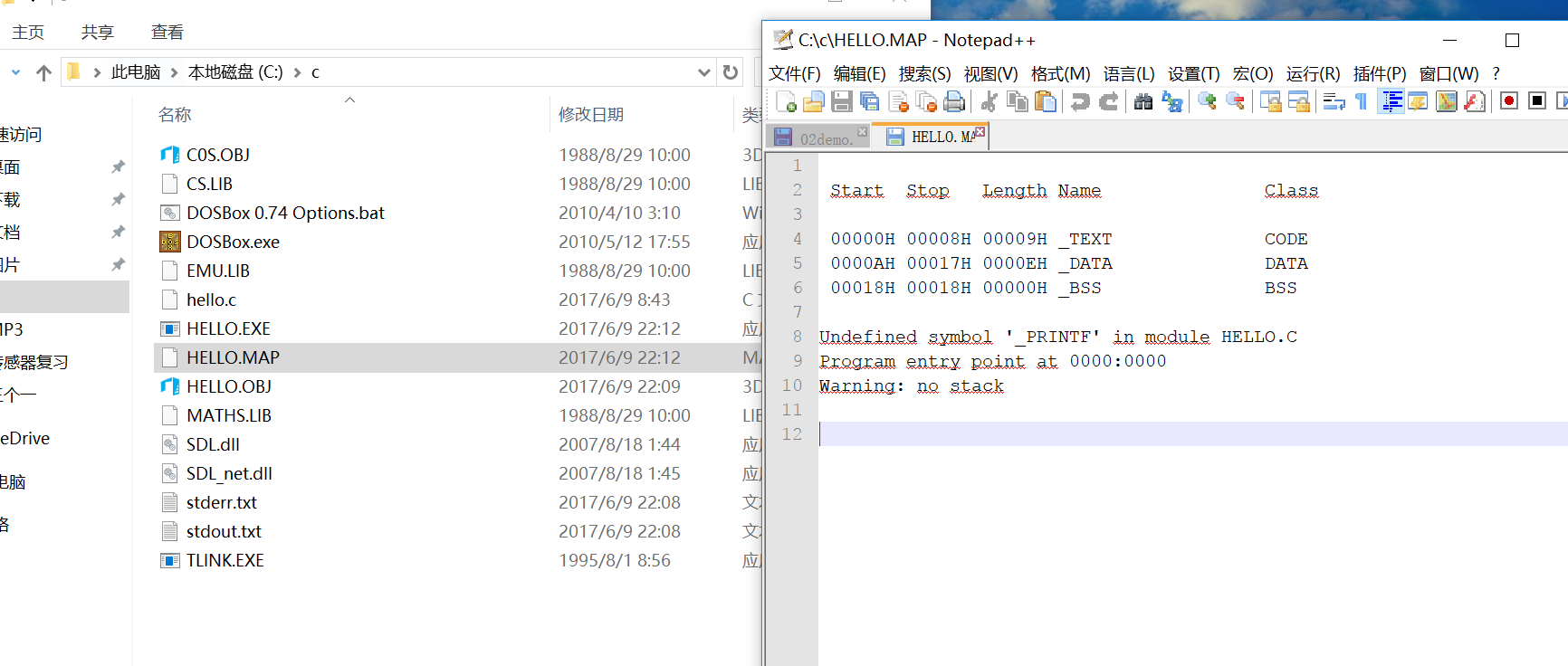
可以看到，“小星星”现象还是发生了，而且仔细对比之后，发现莫名奇妙多出一个hello.map文件，这又是嘛玩意儿，一个大写的？。

上网搜一下：

一般的大型工程都会在生成可执行文件的同时让链接器生成一个map文件，从而大致查看一下可执行文件中符号的内存布局以及从哪里引入可执行文件。这个通常对于小型工程是作用不大，因为代码就那么多，随便grep一下就知道符号定义位置了。但是对于一些大型工程或者涉及了比较多的第三方库、或者涉及了比较多的功能模块的时候，就需要知道这些符号是在哪里定义，或者说如果一个符号引用了但是没有知道函数定义，此时也需要找到这个符号是哪个模块引入的，为什么需要，所以需要一些通用的(形式化)的方法来搜索这些符号，而map文件就是一个比较好的切入点。

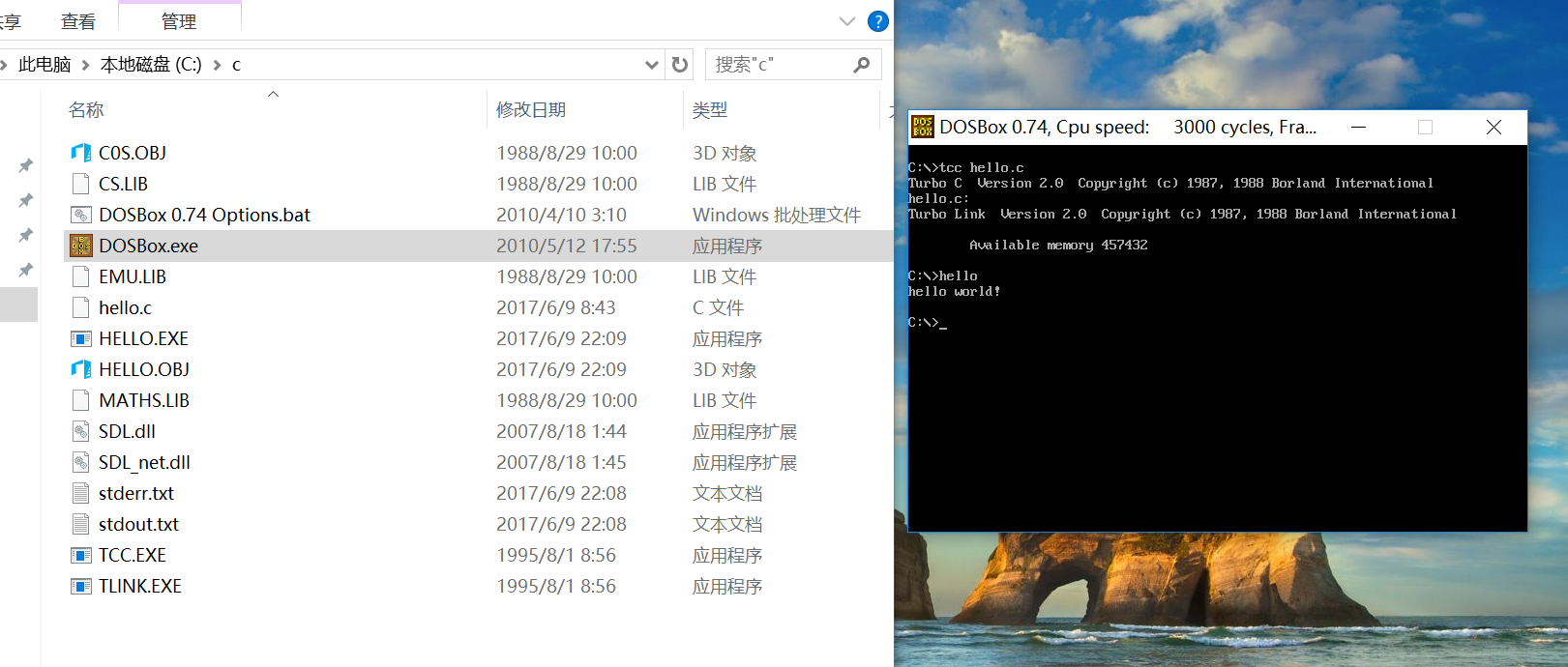
既然这么好，怎么还是出现了“undefined symbol ‘\_PRINTF’ in module HELLO.C”提示呢？而且对比验证三成功的现象，并没有出现hello.map文件，所以应该有很大可能问题出现在这里。

用notpad++打开查看一下：



嗯，大人，此事必有蹊跷。

验证三过程：

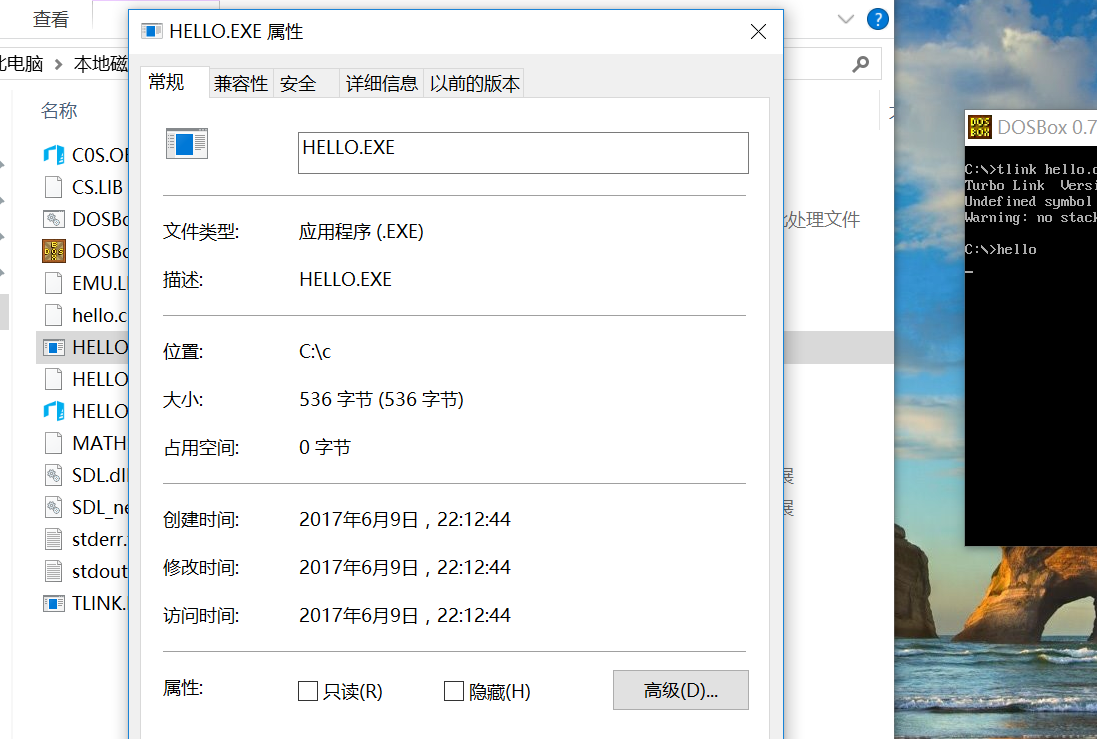


没办法，放在一起才可以。

就目前的观察结果而言，tlink并不能自主将相应hello.c文件所需的\*.lib等文件的相关代码连接成成功的hello.exe 文件，要想成功，tcc.exe文件可能除了编译功能之外，在链接阶段，对tlink的调用可能还涉及了对hello.map 文件的操作，使得相应的出现的“undefined symbol ‘\_PRINTF’ in module HELLO.C”得到了有效地解决。

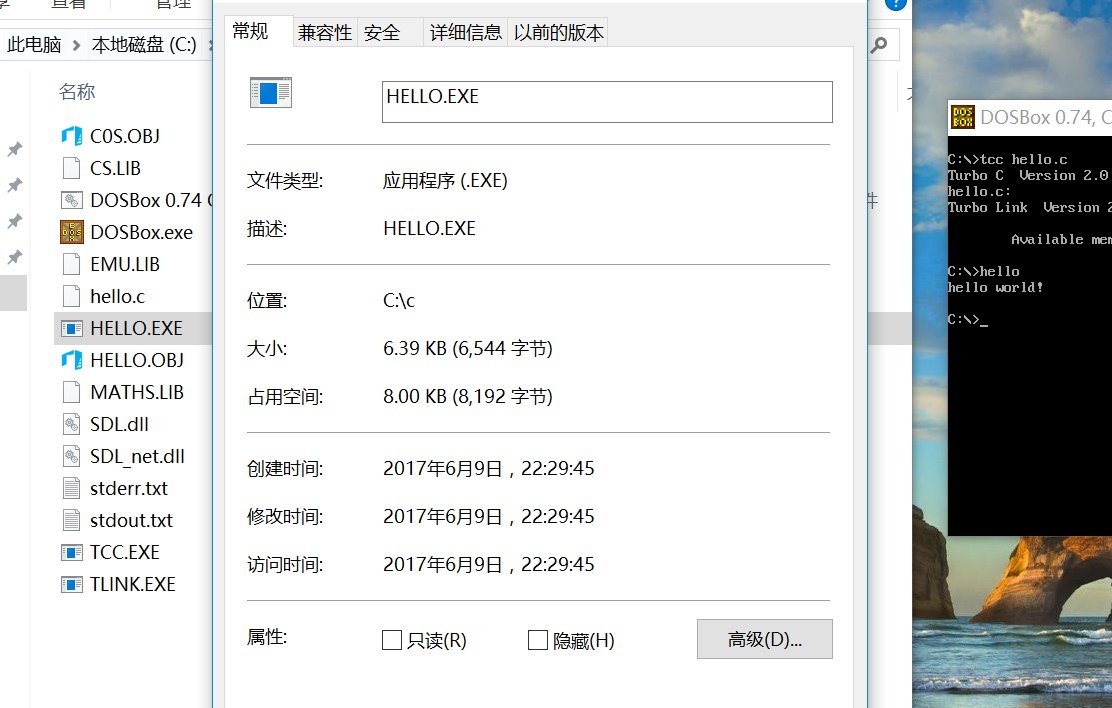
最后可以看一下两种hello.exe文件的大小属性：

失败的：



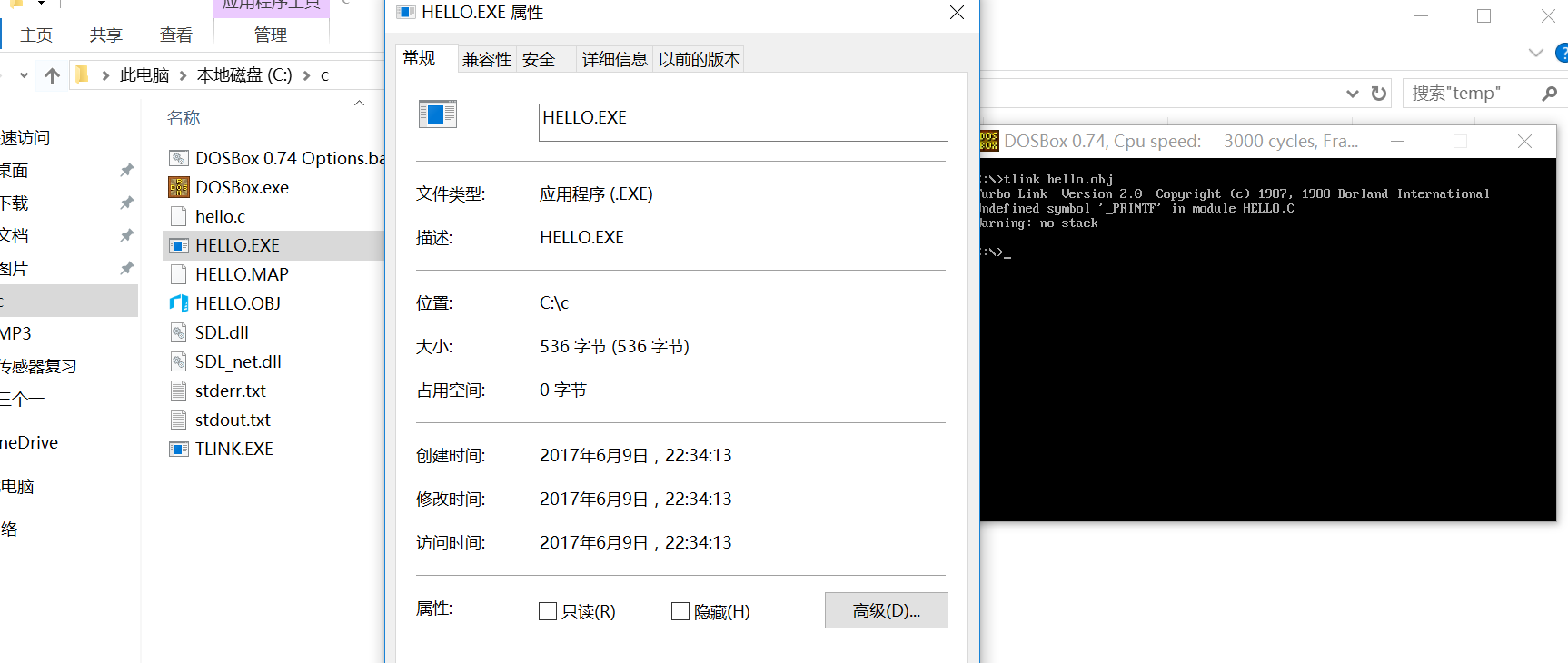
嗯，536字节。

成功的：



嗯，6.39kb

大小悬殊，不难推测，只用tlink.exe根本没有能够将其中的相应的\*.lib文件与\*.obj文件有效地链接起来，“tcc调用tlink”——这句话是关键。还是不太可靠，我可以再做最后的正面验证：



嗯，不出所料，还是536字节。

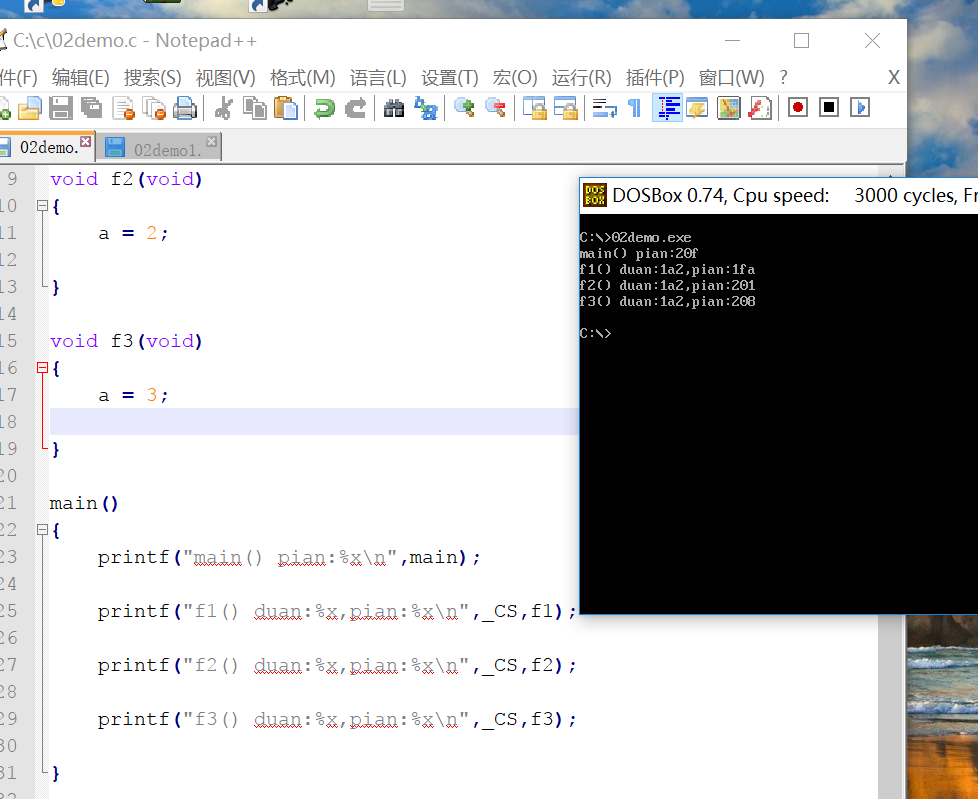
这种情况，如图，是我再次移除了c:\c目录下的一系列\*.lib文件与\*.obj文件还有tcc.exe文件之后得到的结果。

呼~长吁一口气，可算验证了。

看来一个小小的问题，想要细致完整地彻底解决还是挺花功夫的。

为什么会这样呢？是我的验证方式错误吗？还是事实本就是这样呢？

**二：**



代码：

int a;

void f1(void)

{

a = 1;

}

void f2(void)

{

a = 2;

}

void f3(void)

{

a = 3;

}

main()

{

printf("main() pian:%x\n",main);

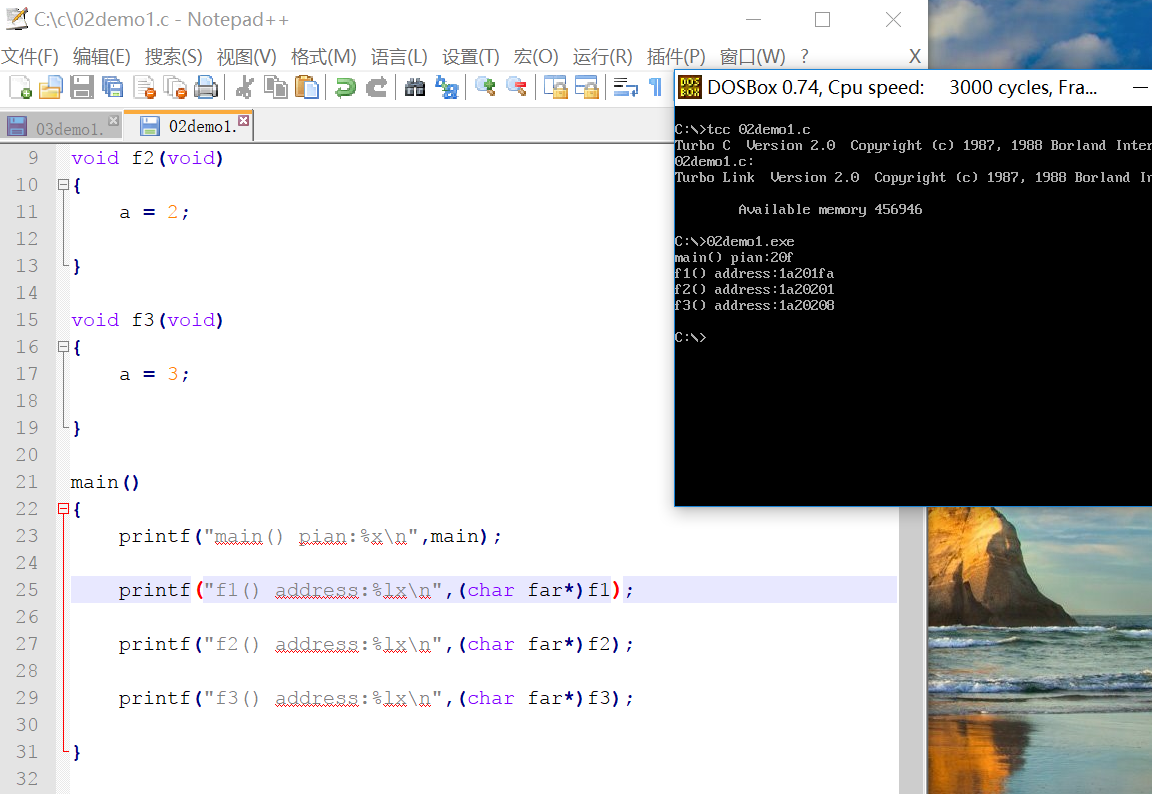
printf("f1() duan:%x,pian:%x\n",\_CS,f1);

printf("f2() duan:%x,pian:%x\n",\_CS,f2);

printf("f3() duan:%x,pian:%x\n",\_CS,f3);

}

加一个刚刚想要拿char far\*类型的结果及代码：



结果是相同的，第二种我觉得可能更可靠，不过之后还需要修改成段地址，偏移地址分别显示的样子。

int a;

void f1(void)

{

a = 1;

}

void f2(void)

{

a = 2;

}

void f3(void)

{

a = 3;

}

main()

{

printf("main() pian:%x\n",main);

printf("f1() address:%lx\n",(char far\*)f1);

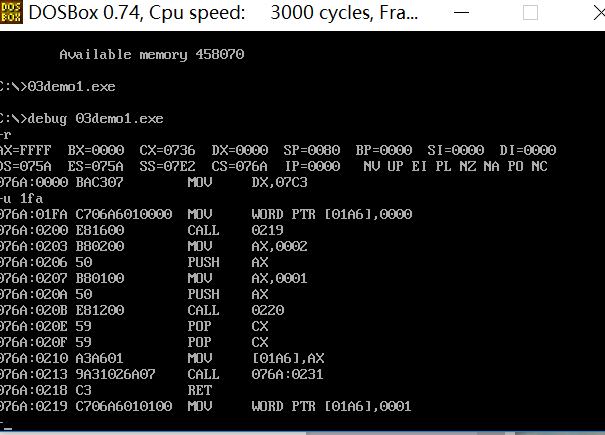
printf("f2() address:%lx\n",(char far\*)f2);

printf("f3() address:%lx\n",(char far\*)f3);

}

**三：**

**1>.程序1部分：**

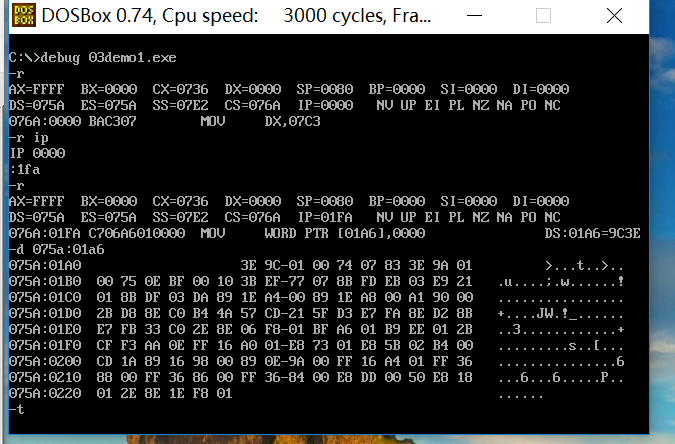


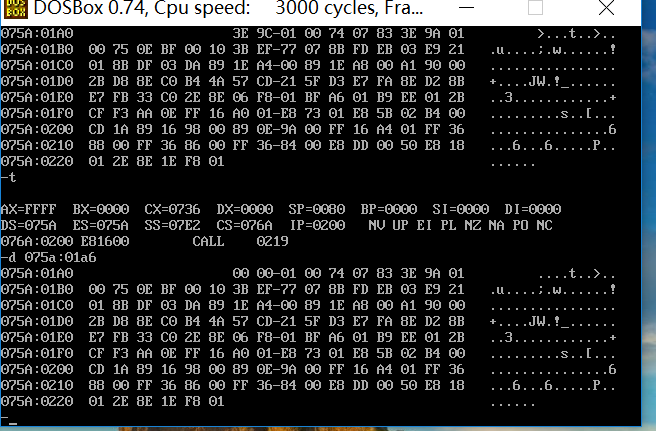
由上图可知，

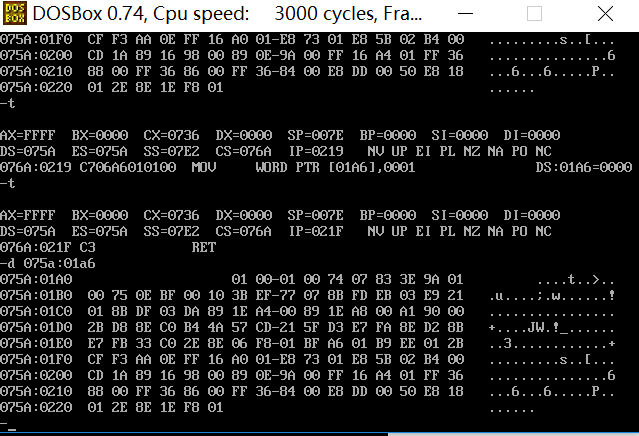
变量n在内存中，段地址为ds=075ah，偏移地址为01a6h，大小为两个字节，对应“unsigned int ”类型。

跳到076a：0219看看：“mov word ptr [01a6],0001h”，再一次验证变量n的实际情况。

当然了，最直接的就是跳到内存地址处看一下，1就是1,0就是0：



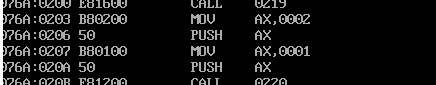




到此，变量n在03demo1.exe程序运行中的分配情况已显而易见。

之前是依次将实参2,1（按照从右往左的顺序）通过寄存器ax压入栈中。

f2(1,2)有传参的过程，而这个过程总体目的应该是令a=1,b=2;



再查看之前的四条汇编语句

“

Mov ax,0002h

Push ax

Mov ax,0001h

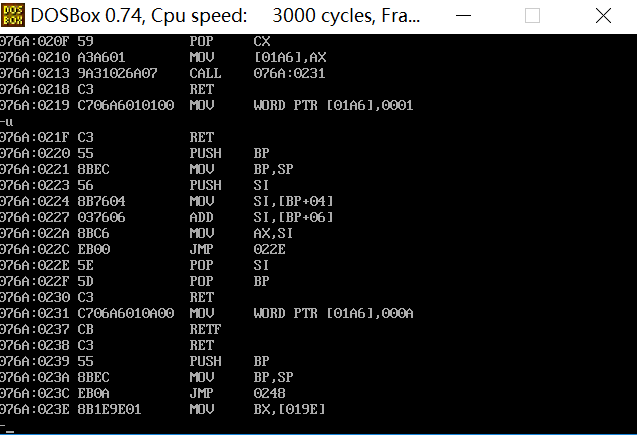
Push ax

”

根据之前所学，通过栈传递参数的过程顺序是从右往左的。也就是说这个过程是将实参2、1分别压入栈中，但是此时还无法确定形参a、b段地址就一定是ss段寄存器中的数值，或许栈只是多次传递数据到变量a、b的中转站之一呢？

但是接着看，假设我们“a=1，b=2”的任务完成了之后呢？我们首要做的是什么？对，首当其冲的，一定是马上执行f2（int,int）的具体内容了。

跳到076a：0220看看：



这里在最基本的保护寄存器操作之后，并没有将原来栈中的1,2进行二次转移，反而是直接进行了运算

“

Mov si,[bp+4]

Add si,[bp+6]

”

到这里，又说明什么了呢。我个人认为，这个现象绝对证明了在实参“2、1”分别压入栈中的过程中，同时也完成了对形参“a、b”赋值的任务，也就是“a=1,b=2”就是在这里一起完成的。

因此，a、b的段地址应该是存放在ss栈段寄存器中。

“

Mov si,[bp+4]

Add si,[bp+6]

”

还是这两条语句，参与运算的还有一个显眼的si寄存器，对于a、b而言，既然存储在栈中，在函数运行完毕，也就意味着“灰飞烟灭”。

这里si有可能就是变量c，也有可能只是传递数据的一个中转站。

可以先看下一句：

“

Mov ax,si

”

疑惑更大了，难不成ax才是真的变量c的真实所在，抑或ax也是中转？

但是再结合076a:0210h：

“

Mov [01a6],ax

”

前一个，我们已经得到验证了，变量n呗，现在将ax中的值赋给变量n；

回到03demo1.c源程序中，我们的原句是：

“

N = f2(1,2);

”

正常的话，应该是将函数的返回值赋值给变量n。而此处ax显然担任了f2()返回值的保存任务，此时在回头看

“

Mov ax,si

”

Ax寄存器是保存返回值，而si将值赋给ax，对应的语句中看来只有si负责变量c了。因为反编译出的语句中所有显示出的寄存器或内存地址都已经名花有主了，只能si和c大眼对小眼了。但是

“

Pop si

”

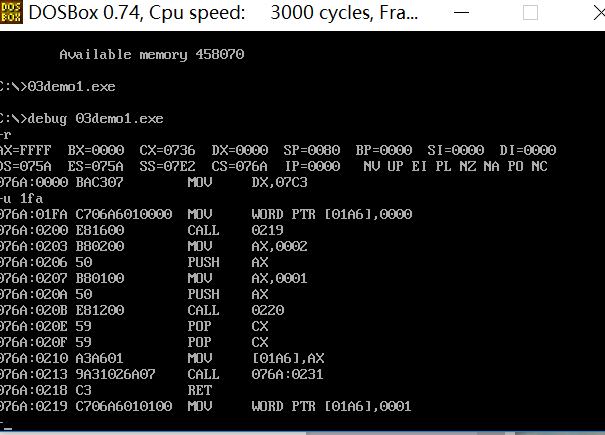
显然听了我的话不高兴了，这里变量c同样处在函数中属于局部变量，从生存期的角度，他们应该是一样的。A、b在栈中，但是函数调用完毕之后，a、b生存期也就到了。但是si呢？同样在“ret”之前有“pop si”，也不过是“五十步”与“百步”的差别吧。

变量c才不会为自己与a、b身世不同，没有处在栈中而自怨自艾，相反，变量c存储在寄存器si中，简直是“上错花轿嫁对郎”！

全局变量的存储空间在程序开始就分配了，在整个程序执行完才释放。

局部变量是在子函数开始执行时分配的，那么是在函数入口处将局部变量全部分配，还是在函数中局部变量定义处分配呢？局部变量的存储空间是在函数结束时释放的。

从程序中可以看到，函数参数的存储空间是在主函数里对函数进行调用时就分配的，也就是将参数的值入栈，而在函数返回后，用pop cx将参数从栈段中释放。



由上图可得：

调用时：

F1() call 0219h

F2() call 0220h

F3() call 076a:0231h

返回时：

F1() ret

F2() ret

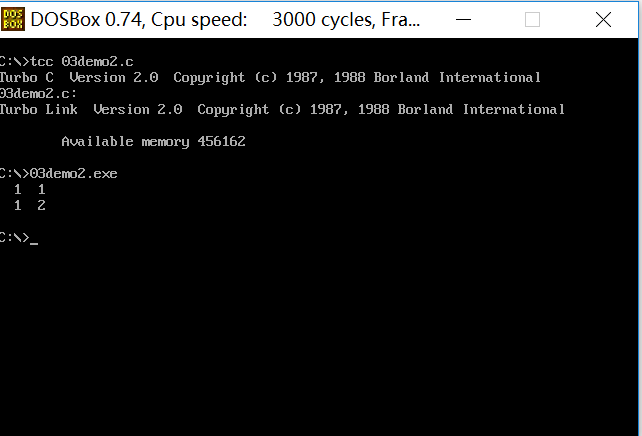
F3() retf

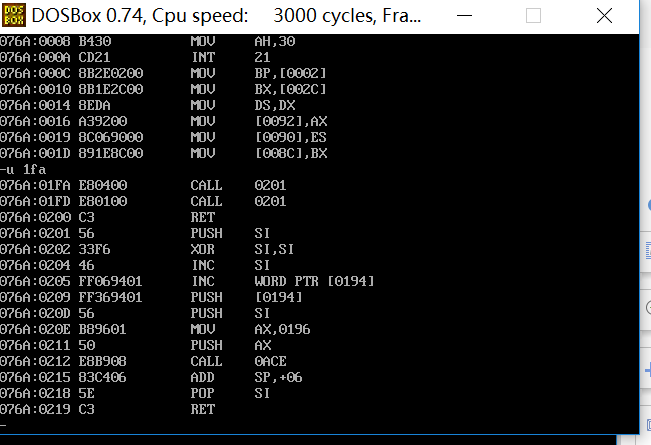
主函数里调用f3()使用的语句是“call 076a:0231h”,也就是直接call函数的段地址+偏移地址。而f1()、f2()则是直接call函数的偏移地址，段地址默认cs寄存器中的数值。

f3()返回时是用retf返回的，也就是将ip和cs都出栈。所以对于far型的函数，调用时要用call 段地址+偏移地址，返回时要用retf将段地址和偏移地址都出栈。而f1()、f2()则是ret返回的，也就是将ip出栈，与call指令相对应。

**2>.程序2部分：**

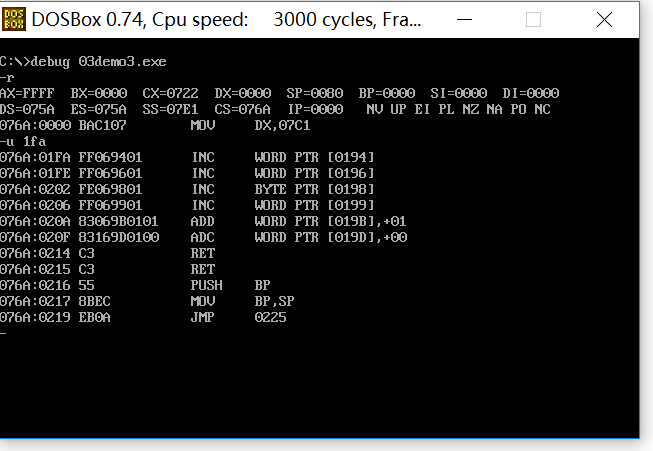
先看一下结果：





如图可知，n的存储空间为si寄存器，a的存储空间为以ds：0194为地址的两个字节。局部变量n的存储空间是在函数开始时分配的，而a的存储空间是固定的内存空间，不是栈段，在函数结尾处n的空间被释放了而a的空间并没有被释放。局部变量n与函数内部定义的静态变量a在生存期方面的差别体现出来。

**3>.程序3部分：**



单纯从汇编代码中查看：

word ptr [0194],也就是占据ds:0194h~ds:0195h两个字节单元；

word ptr [0196],也就是占据ds:0196h~ds:0197h两个字节单元；

Byte ptr [0198],也就是占据ds:0198h一个字节单元；

word ptr [0199],也就是占据ds:0199h~ds:019ah两个字节单元；

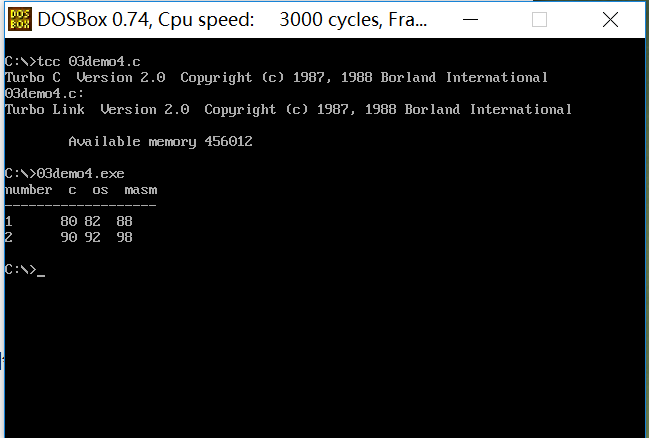
word ptr [019b],word ptr [019d],也就是占据ds:019bh~ds:019eh四个字节单元；

整型的存储空间为2个字节，字符型为1个字节，长整型为4个字节。

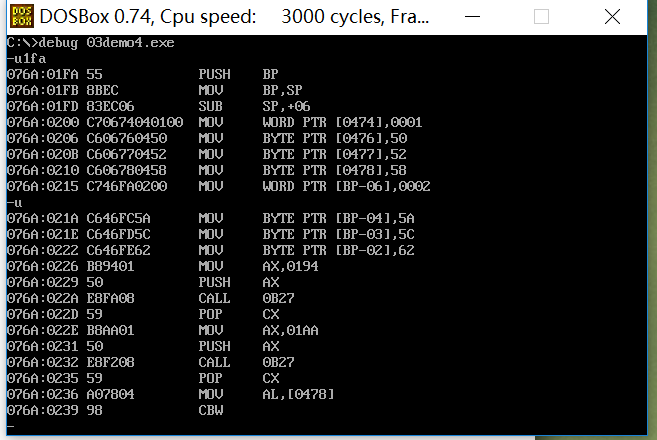
自加1运算时，整型是inc word ptr，对1个字的数据进行操作；字符型是inc byte ptr，对1个字节的数据进行操作；长整型是先对低四位数据进行运算，再用位运算符adc对高四位进行运算得到结果。

**4>.程序4部分：**

结果:



Debug:



通过查看后面要赋值的具体数值，就可判断变量a、b的各个数据项表示：

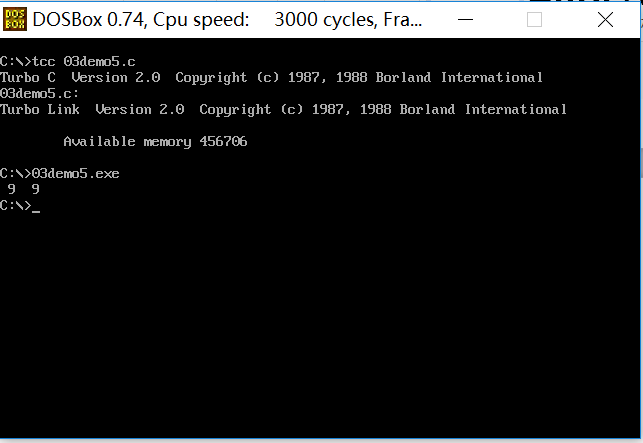
前四条mov语句是依次赋值给变量a的4个数据项；

后四条mov语句是依次赋值给变量b的4个数据项；

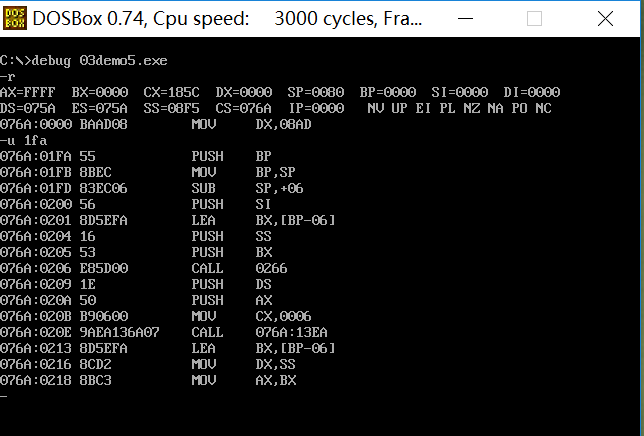
变量a、b各个数据项的赋值部分：a的每个数据项都有固定的内存地址，而b的数据项都是存储在栈段里面，因为a是全局变量而b是局部变量。而且a、b里面的数据项的各自的存储空间是相邻的。

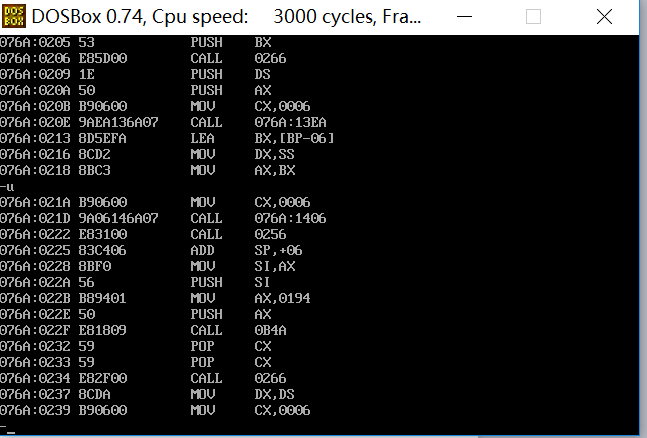
**5>.程序5部分：**

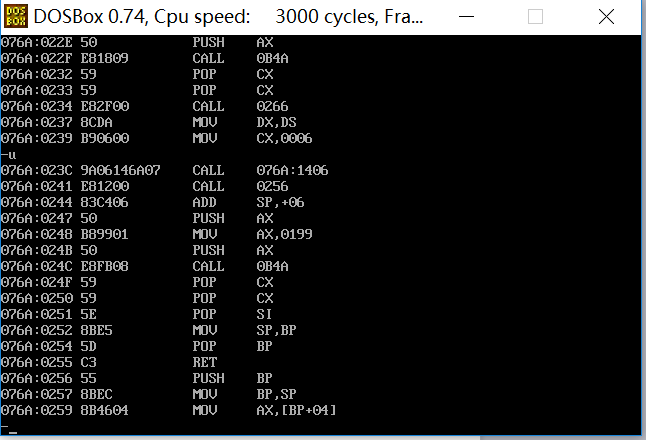
结果：



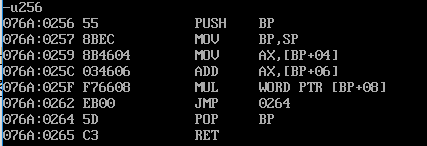
Debug：







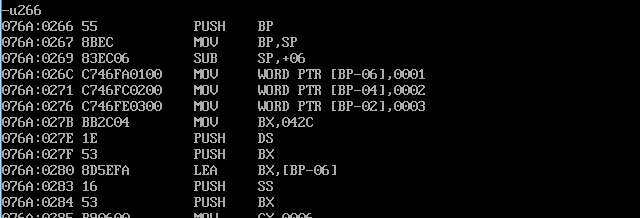
f中调用的结构体数据在什么地方：

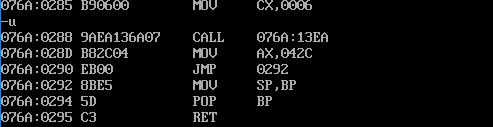


调用的数据在栈段里面，a.a是bp+4，a.b是bp+6，a.c是bp+8.

那么main函数传值应该是将数据项压栈的过程。

函数func的内容：





网上资料:

C 语言中函数返回结构体时如果结构体较大, 则在调用函数中产生该结构的临时变量，并将该变量首地址传递给被调用函数，被调用函数返回时根据该地址修改此临时变量的内容，之后在调用函数中再将该变量复制给用户定义的变量，这也正是C 语言中所谓值传递的工作方式。  
     如果结构体较小, 则函数返回时所用的临时变量可保存在寄存器中，返回后将寄存器的值复制给用户定义的变量即可。

可以看到，f()与func()的区别:

F()是接收值是结构体；func()是返回值是结构体。如果用图表示的话是这样的：

结构体-->f();

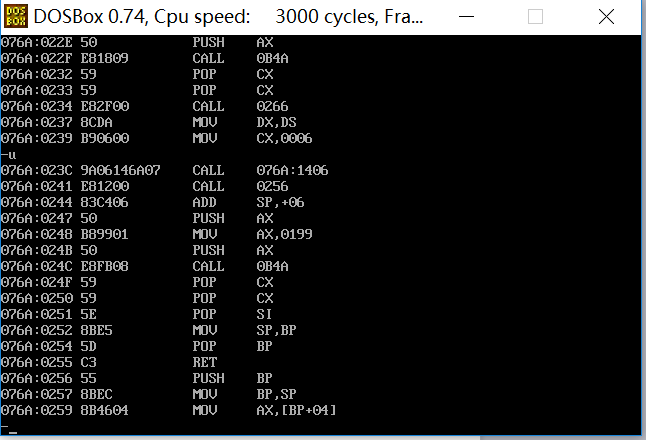
Func()-->结构体。

但是两个过程也是有相同点的。

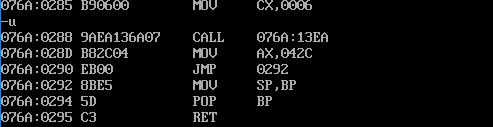
两者明面上都是值传递。

而且

当f()接收结构体之前，有一条call语句“call 076a:1406h”

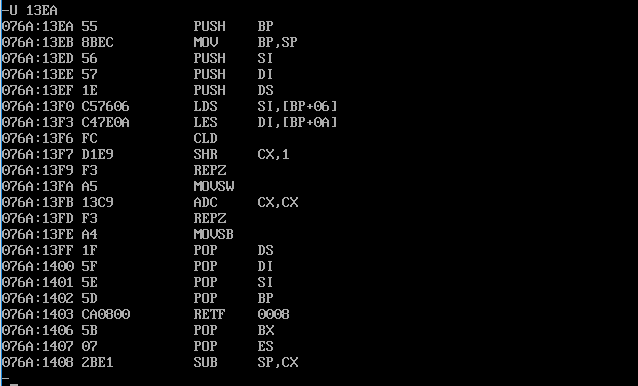


当func()返回结构体之前，同样有一条call语句“call 076a:13eah”



可以分别查看一下：

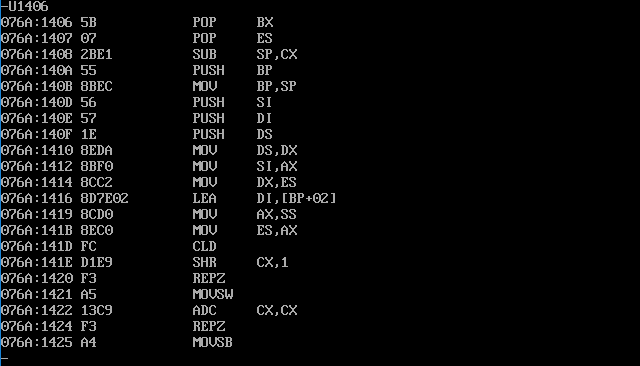
看一下076a:13eah:

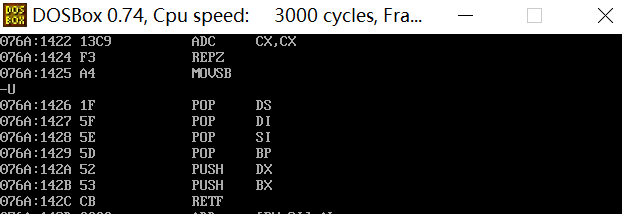


虽然有些没见过的指令，但是“万花丛中一点绿”——“movsb”

有点搬运数据的意思了。

看一下076a:1406h:





再看一下这个呢？“万花丛中二点绿”——“movsw”

而且再留心一下，两者的地址是相连的，结合起来看，二者的关系绝对是不一般的。功能上存在相似性。就看到的，基本可以推测这两个函数都是搬运函数。

在func()返回结构体之前的call语句“call 076a:13eah”是实现：

Func()-->结构体。

在f()接收结构体之前的call语句“call 076a:1406h”是实现：

结构体-->f();

但是我这里的“结构体”实际上是原数据的拷贝。但是之前我提出的返回指针，结合网上收集的资料整合来看：当结构体数据量很大时，返回的是指向这个复制品的指针，这样是否没有更改原来的值而且提升了效率呢。

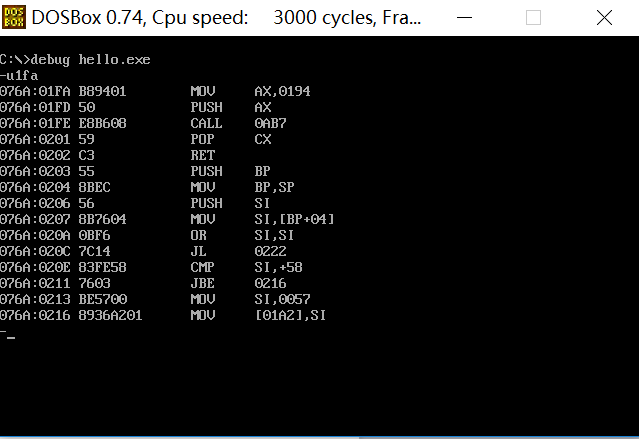
其实我觉得，这里的处理机制，其实很像输入输出操作、读写操作等。中间取出buf空间存储复制品，然后进行操作就很灵活了。

**补充的：**

1. map文件。（之前已解决）
2. 搭建精简环境：分别只放tcc，tlink，移除其他文件。执行tcc发现编译不需要其他库文件和c0s.OBJ。单独用tlink需要指出所需的全部文件以及路径。这种研究方法值得借鉴。（之前已解决）
3. tcc编译的代码与tc编译的代码有区别，tcc函数缺少push bp mov bp,sp-等代码。相对来说tcc生成的代码更精简。

比较一下：

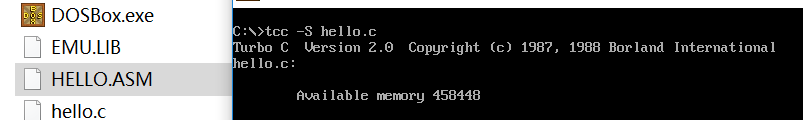
内容相同的hello.c与hello1.c，分别用tcc与tc编译：



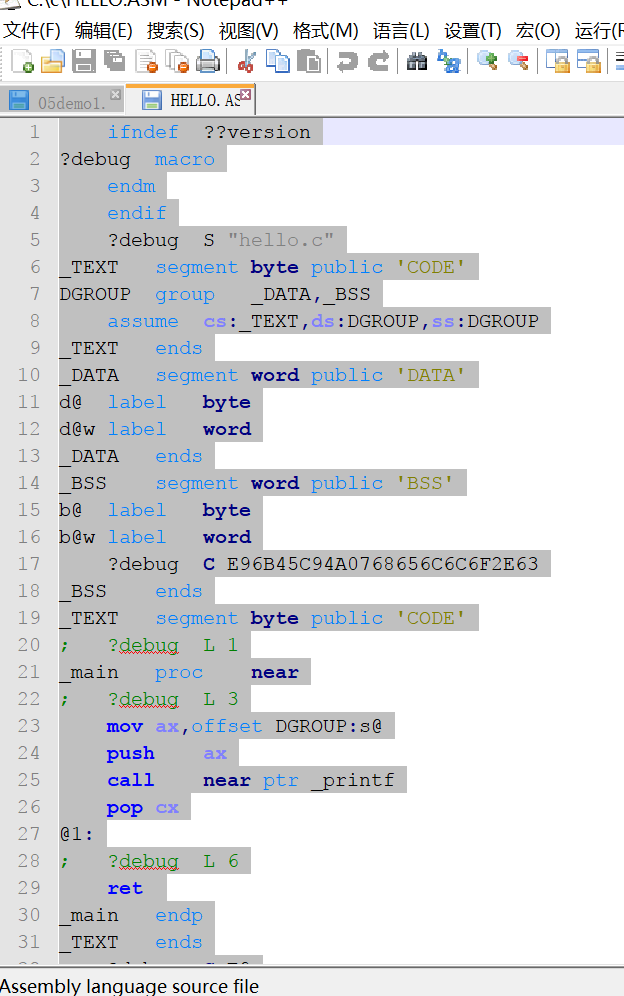
结论：成立

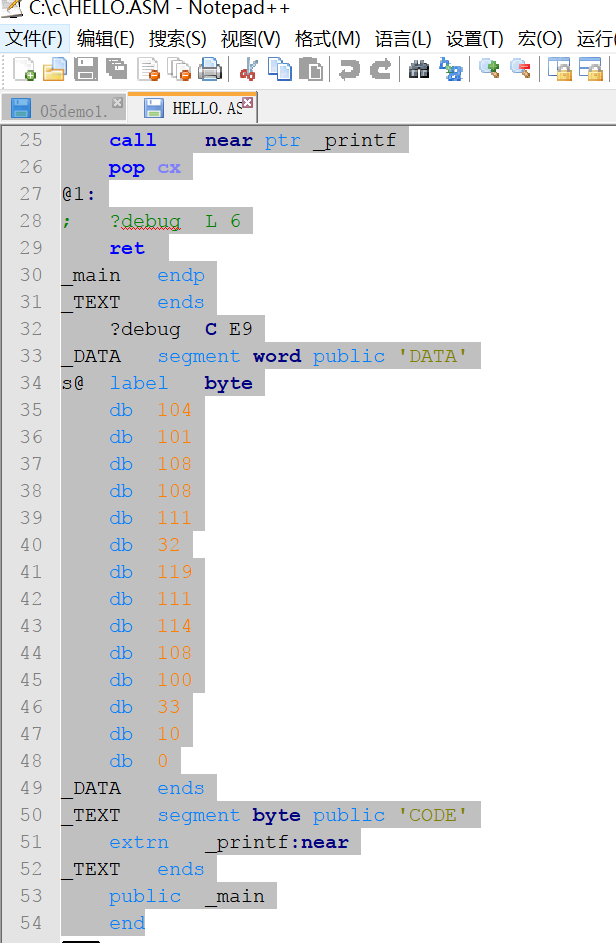
1. 如何调用tlink：用的system系统函数。（暂时不管）
2. 不同函数的联动：Tcc，tlink均可，甚至也可以自己写库。（用到再说）
3. 内联汇编：tcc -s功能，可生成.asm文件。（此处正确应为tcc -S）

以hello.c为例：



确实生成了hello.asm文件，用notpad++打开查看一下：





结论：成立。

1. 打印段地址：一、打印cs。二、直接强制类型转换（char far \*）f。三、有人提出函数指针打印long far \*p(void); P=f; printf(“%lx”,p); 注意应该加far强转。

（已解决）

1. 取错了的段地址内容：call语句入栈的ip。

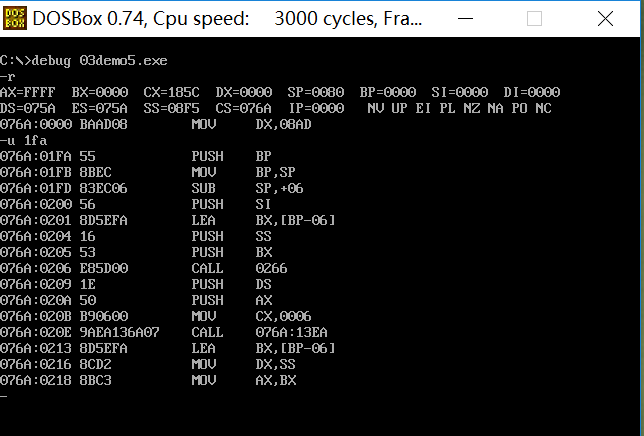
此处如果将原本只含有偏移地址的数据变量强制成“lx”类型打印出来，会将偏移地址显示为高位，并且会从当前栈中取出一个数据，也就是call语句入栈的ip值显示为低位。

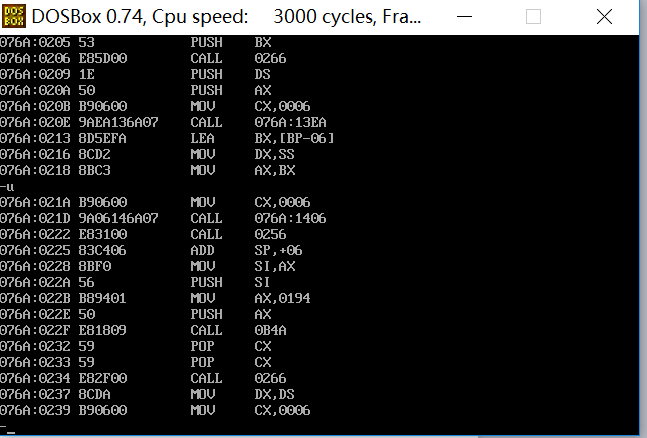
1. cmd和debug阶段执行程序打印的段地址不同的问题：因为要在内存里执行程序，cmd和debug 会把程序放在不同的内存中运行，取出的段地址故此不同。

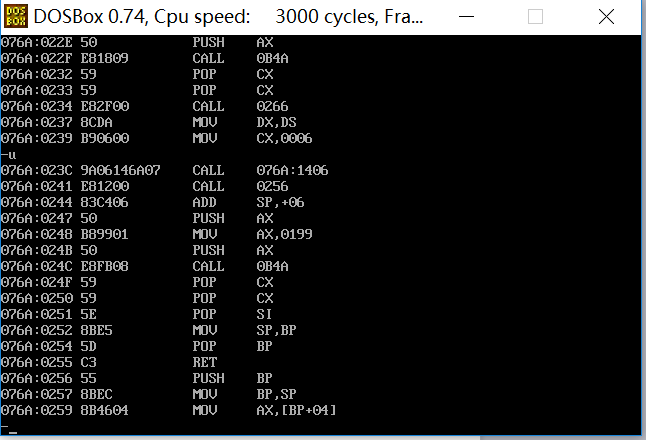
（已解决）

1. 变量的段地址所在寄存器，全局变量，局部变量存储空间，均通过运行时的调用语句判断的。（已解决）
2. tc下不能返回数组。
3. 静态局部变量和普通局部变量生存周期不同，局部静态变量与全局静态变量作用域不同。静态的生存周期在整个程序运行期间。静态局部变量存在数据段里。外部静态和内部静态作用域不同。各种数据类型存储空间。不赘述。
4. 不同数据类型对数据运算方式的影响。（double先不考量，自己私下了解）
5. 结构体内存分配：连续分配内存。
6. 传递结构体型数据：调用了一个库函数把数据入栈来传递。调用库函数之前需要给出起始地址、字节长度和目的地址。（已解决）

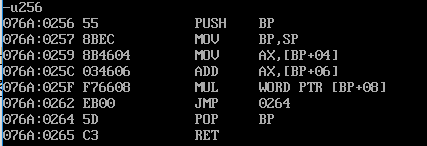
Debug：







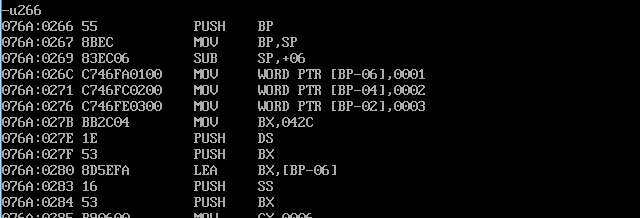
f中调用的结构体数据在什么地方：

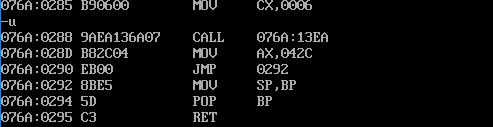


调用的数据在栈段里面，a.a是bp+4，a.b是bp+6，a.c是bp+8.

那么main函数传值应该是将数据项压栈的过程。

函数func的内容：





网上资料:

C 语言中函数返回结构体时如果结构体较大, 则在调用函数中产生该结构的临时变量，并将该变量首地址传递给被调用函数，被调用函数返回时根据该地址修改此临时变量的内容，之后在调用函数中再将该变量复制给用户定义的变量，这也正是C 语言中所谓值传递的工作方式。  
     如果结构体较小, 则函数返回时所用的临时变量可保存在寄存器中，返回后将寄存器的值复制给用户定义的变量即可。

可以看到，f()与func()的区别:

F()是接收值是结构体；func()是返回值是结构体。如果用图表示的话是这样的：

结构体-->f();

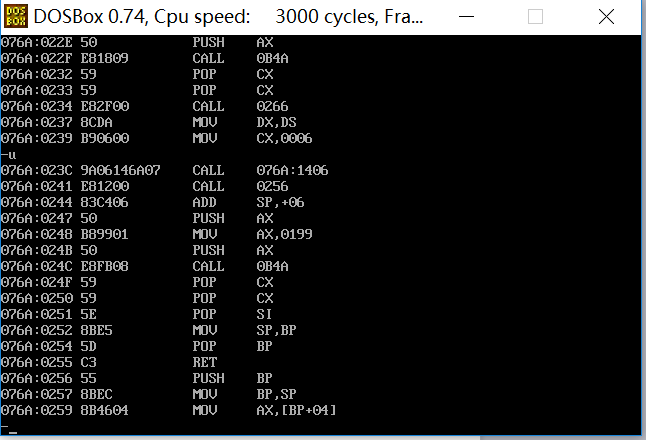
Func()-->结构体。

但是两个过程也是有相同点的。

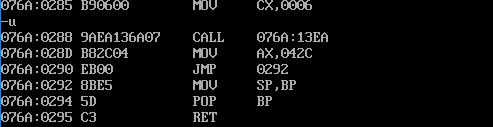
两者明面上都是值传递。

而且

当f()接收结构体之前，有一条call语句“call 076a:1406h”

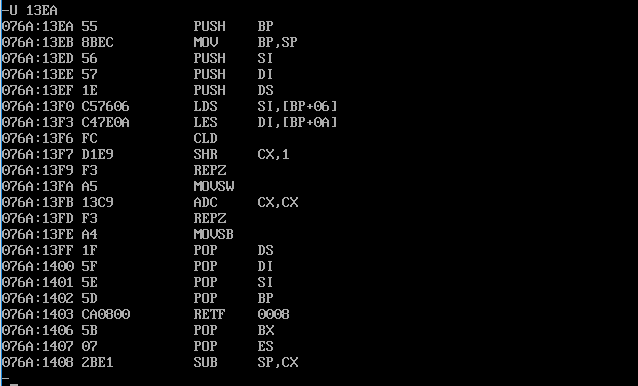


当func()返回结构体之前，同样有一条call语句“call 076a:13eah”



可以分别查看一下：

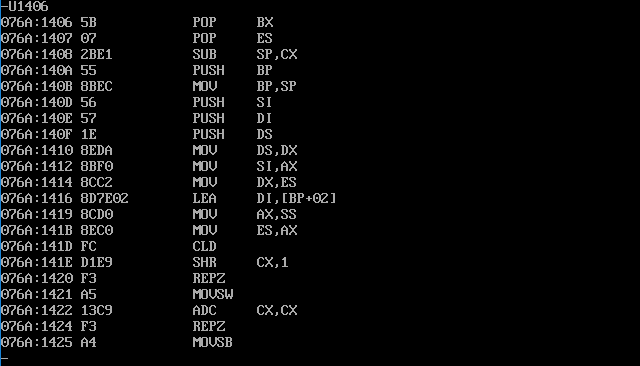
看一下076a:13eah:

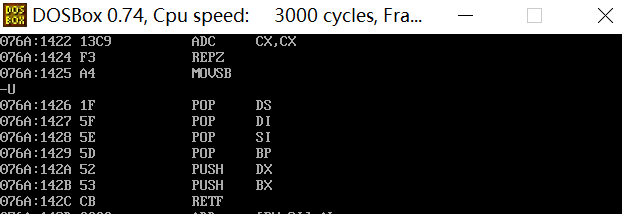


虽然有些没见过的指令，但是“万花丛中一点绿”——“movsb”

有点搬运数据的意思了。

看一下076a:1406h:





再看一下这个呢？“万花丛中二点绿”——“movsw”

而且再留心一下，两者的地址是相连的，结合起来看，二者的关系绝对是不一般的。功能上存在相似性。就看到的，基本可以推测这两个函数都是搬运函数。

在func()返回结构体之前的call语句“call 076a:13eah”是实现：

Func()-->结构体。

在f()接收结构体之前的call语句“call 076a:1406h”是实现：

结构体-->f();

但是我这里的“结构体”实际上是原数据的拷贝。但是之前我提出的返回指针，结合网上收集的资料整合来看：当结构体数据量很大时，返回的是指向这个复制品的指针，这样是否没有更改原来的值而且提升了效率呢。

其实我觉得，这里的处理机制，其实很像输入输出操作、读写操作等。中间取出buf空间存储复制品，然后进行操作就很灵活了。

16、子程序的栈：整个程序只有一个栈，但每个子程序用各自的区域。（了解）

1. **已思考研究并已解决问题汇总**

**以前的：**

**1>.既然大家都是变量都是用来存储的，为啥放的位置还不一样呢？或者全局变量为啥不用栈？局部变量为啥不选数据段？**

这个肯定的，首先全局变量，就是要让它从定义起就可以在程序中任意位置使用，如果选择栈了，栈用完就pop了，其他地方还咋用呢？这就是全局变量不选栈；

局部变量呢？如果所有的数据都直接放在数据段中，局部变量也变成了直到整个程序运行完毕才会得到释放的话，那么如果我们定义的变量越来越多。。。。。。一方面增大了内存的开销，另一方面，这个“局部变量”哪里都可以修改，函数的独立性不好保证。

**2>.程序1中，全局变量n，是由“unsigned int n”这条语句定义，还是由main函数中的“n=0”这条语句定义？**

应该是由前者定义的，函数外定义的变量，不管有没有加static，没有初始化的话，系统默认初始化为0。如果在n=0语句之前打印n，是能够打印出它的值的。

**补充的：**

1、map文件。

2、搭建精简环境：分别只放tcc，tlink，移除其他文件。执行tcc发现编译不需要其他库文件和c0s.OBJ。单独用tlink需要指出所需的全部文件以及路径。这种研究方法值得借鉴。

3、tcc编译的代码与tc编译的代码有区别，tcc函数缺少push bp mov bp,sp-等代码。相对来说tcc生成的代码更精简。

4、如何调用tlink：用的system系统函数。

5、不同函数的联动：Tcc，tlink均可，甚至也可以自己写库。

6、内联汇编：tcc -s功能，可生成.asm文件。

7、打印段地址：一、打印cs。二、直接强制类型转换（char far \*）f。三、有人提出函数指针打印long far \*p(void); P=f; printf(“%lx”,p); 注意应该加far强转。

8、取错了的段地址内容：call语句入栈的ip。

9、cmd和debug阶段执行程序打印的段地址不同的问题：因为要在内存里执行程序，cmd和debug 会把程序放在不同的内存中运行，取出的段地址故此不同。

10、变量的段地址所在寄存器，全局变量，局部变量存储空间，均通过运行时的调用语句判断的。

11、tc下不能返回数组。

12、静态局部变量和普通局部变量生存周期不同，局部静态变量与全局静态变量作用域不同。静态的生存周期在整个程序运行期间。静态局部变量存在数据段里。外部静态和内部静态作用域不同。各种数据类型存储空间。不赘述。

13、不同数据类型对数据运算方式的影响。

14、结构体内存分配：连续分配内存。

15、传递结构体型数据：调用了一个库函数把数据入栈来传递。调用库函数之前需要给出起始地址、字节长度和目的地址。

16、子程序的栈：整个程序只有一个栈，但每个子程序用各自的区域。

1. **已思考研究并未解决问题汇总**
2. 内存对齐：节省时间，以空间换时间。看资料自己推演。vc下可以写代码规定内存对齐。（希望大家的研究要有深度）

二、经典的错误：null pointer assignment

1. **研究感想（心得体会）**

“你懂c吗？不，你不懂c！”，怀着这样的态度继续思考吧。