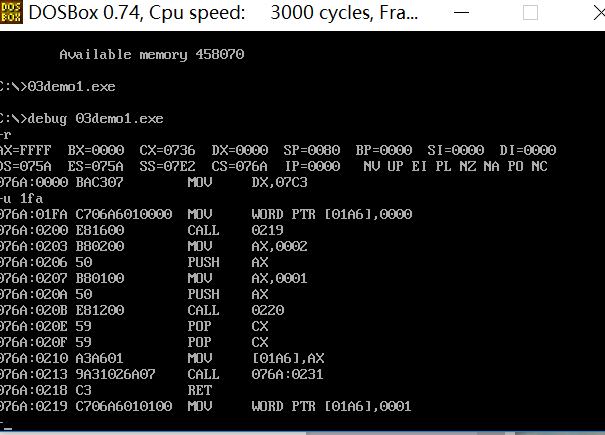
**c语言研究报告三**

1. **研究过程展示**

**1>.程序1部分：**

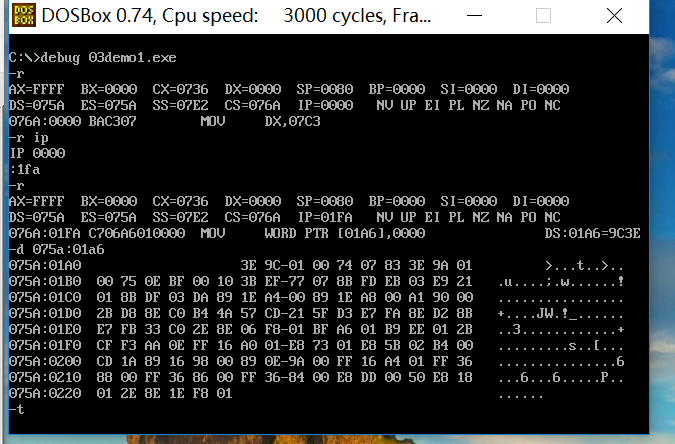


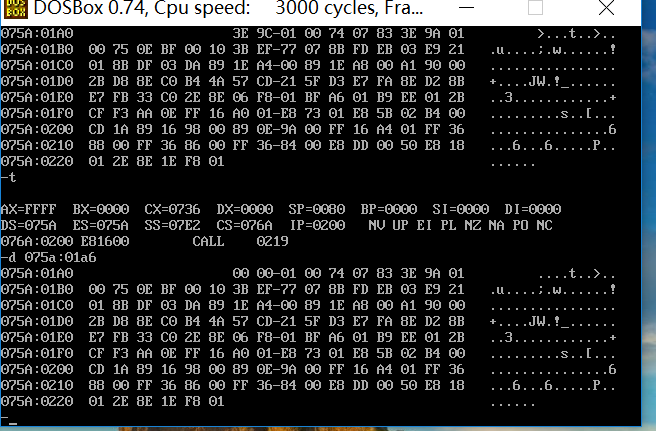
由上图可知，

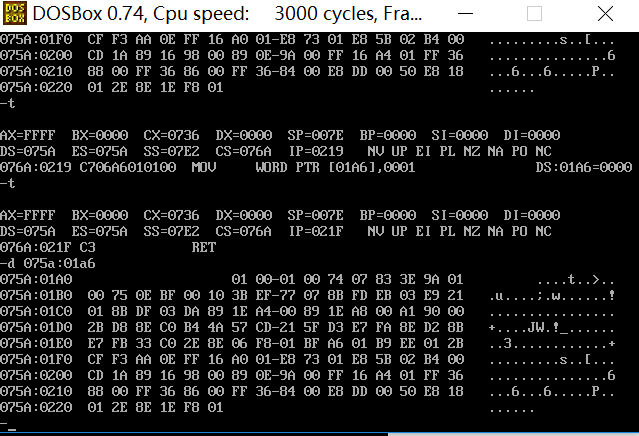
变量n在内存中，段地址为ds=075ah，偏移地址为01a6h，大小为两个字节，对应“unsigned int ”类型。

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：



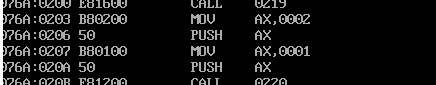




到此，变量n在03demo1.exe程序运行中的分配情况已显而易见。

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

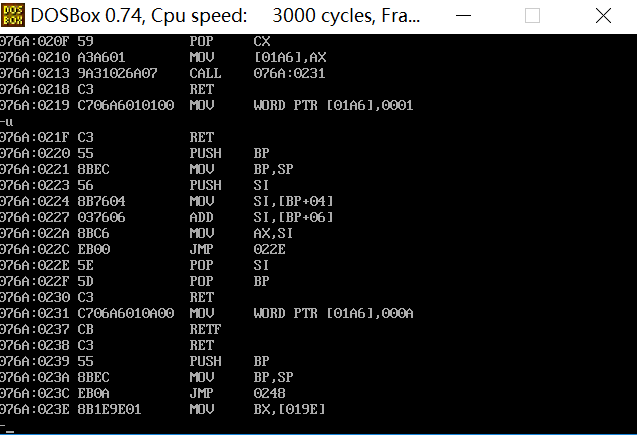
Push ax

”

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，有说明什么了呢。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

”

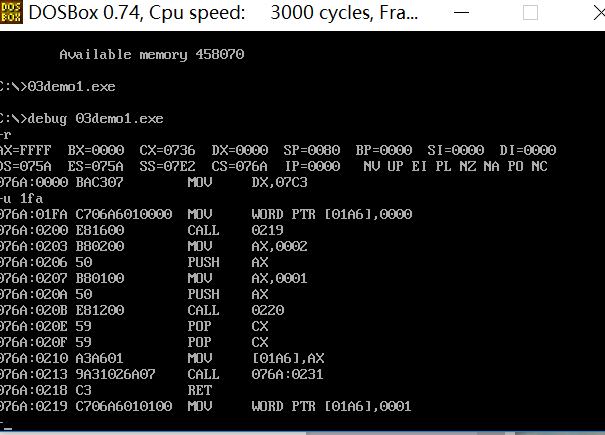
显然听了我的话不高兴了，这里变量c同样处在函数中属于局部变量，从生存期的角度，他们应该是一样的。A、b在栈中，但是函数调用完毕之后，a、b生存期也就到了。但是si呢？同样在“ret”之前有“pop si”，也不过是“五十步”与“百步”的差别吧。

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

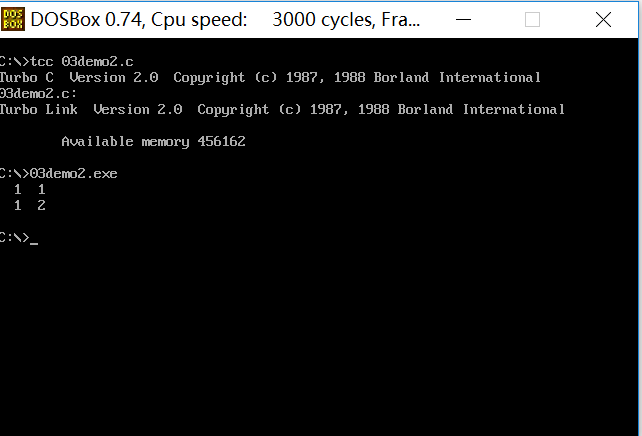
F3() retf

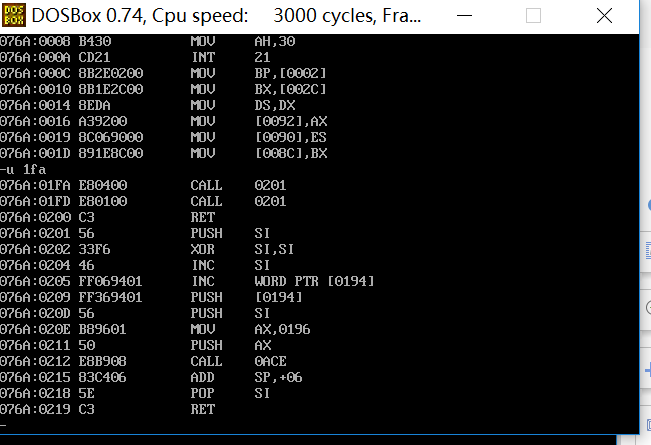
主函数里调用f3()使用的语句是“call 076a:0231h”,也就是直接call函数的段地址+偏移地址。而f1()、f2()则是直接call函数的偏移地址，段地址默认cs寄存器中的数值。

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：**

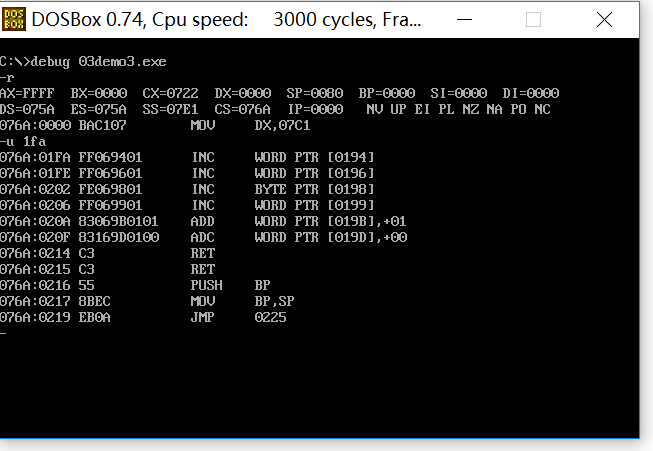
先看一下结果：





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；

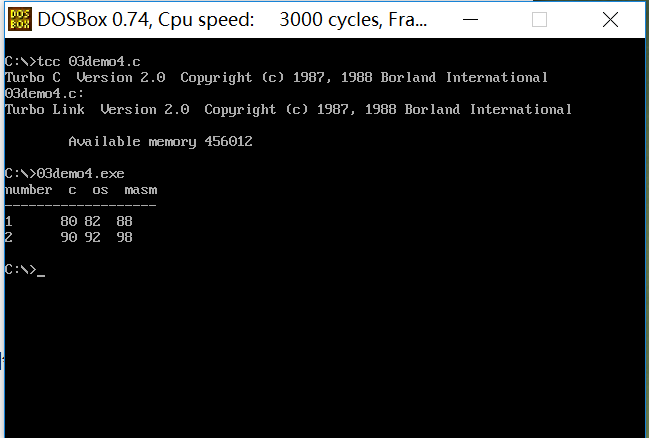
word ptr [019b],word ptr [019d],也就是占据ds:019bh~ds:019eh四个字节单元；

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。

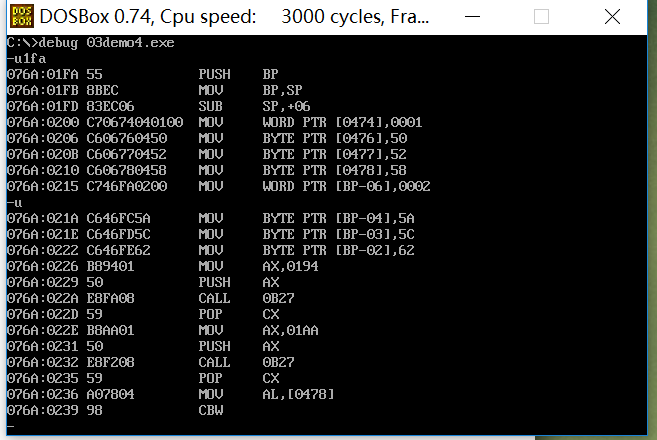
自加1运算时，整型是inc word ptr，对1个字的数据进行操作；字符型是inc byte ptr，对1个字节的数据进行操作；长整型是先对低四位数据进行运算，再用位运算符adc对高四位进行运算得到结果。

**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：

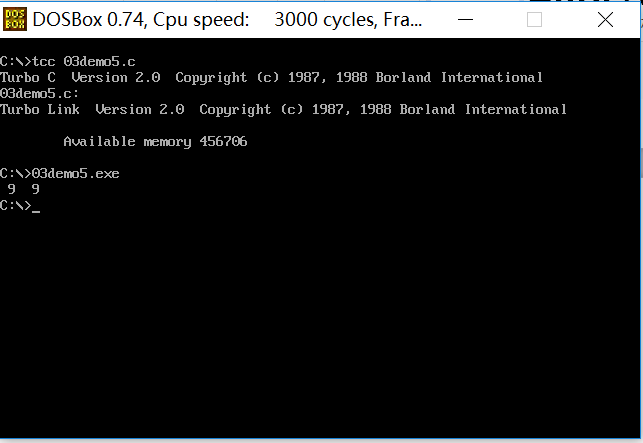
前四条mov语句是依次赋值给变量a的4个数据项；

后四条mov语句是依次赋值给变量b的4个数据项；

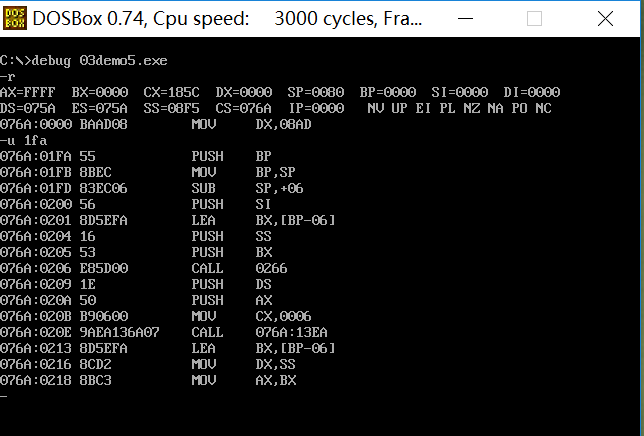
变量a、b各个数据项的赋值部分：a的每个数据项都有固定的内存地址，而b的数据项都是存储在栈段里面，因为a是全局变量而b是局部变量。而且a、b里面的数据项的各自的存储空间是相邻的。

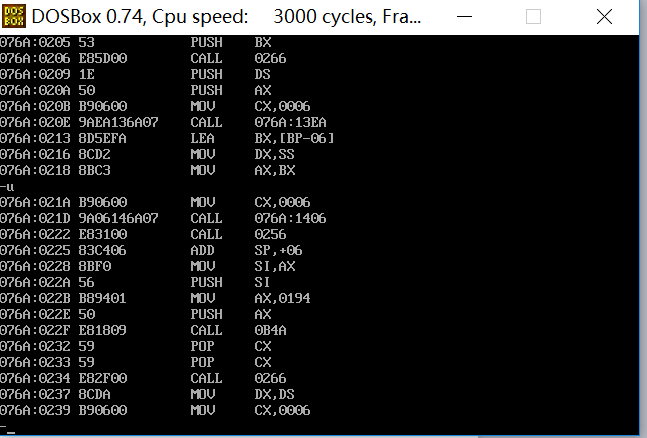
**5>.程序5部分：**

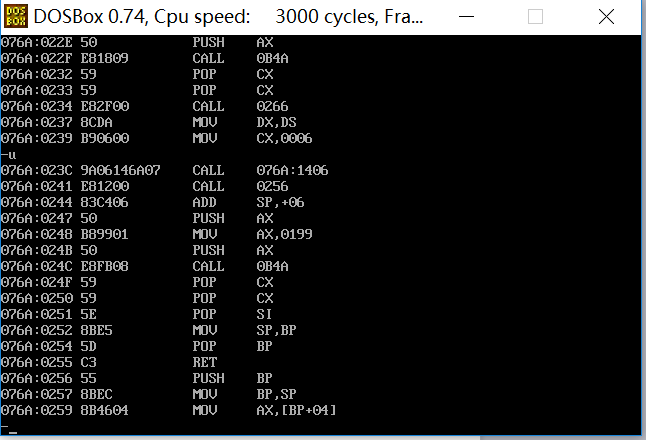
结果：



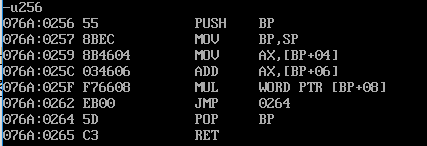
Debug：







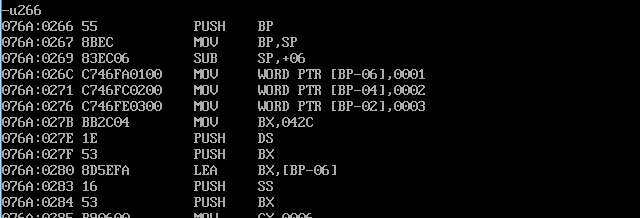
f中调用的结构体数据在什么地方：

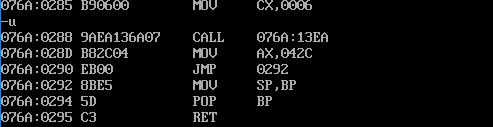


调用的数据在栈段里面，a.a是bp+4，a.b是bp+6，a.c是bp+8.

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。

1. **已思考研究并已解决问题汇总**

**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。

1. **已思考研究并未解决问题汇总**

**1>.对程序5的研究，有的地方还没实际的观察到具体效果，需要完善。**

**2>.自己还需要分别对现有的提供的程序进行自己想到的补充，并进行下一步的进阶验证，但是添加的可能又有很多种，不可能全部列出，需要完善。**

1. **研究感想（心得体会）**

一个变量难倒英雄好汉，因为这是一个强迫自己去学习别人的脑子（计算机）是如何思考的过程。习惯了用到啥学啥，了解变量到这么繁琐的程度，会不会显得冗余呢？以后有多少问题会因为这些引起的呢？值得思考。