《软件安全》实验报告

姓名：冯思程 学号：2112213 班级： 计算机科学与技术二班

**实验名称：**

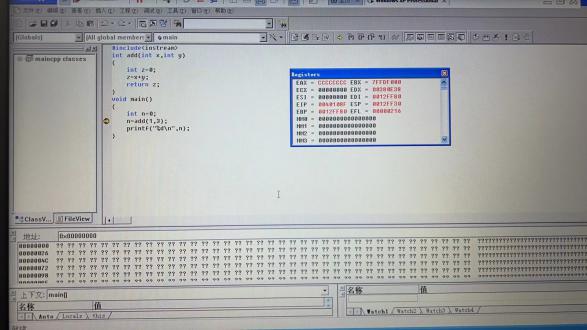
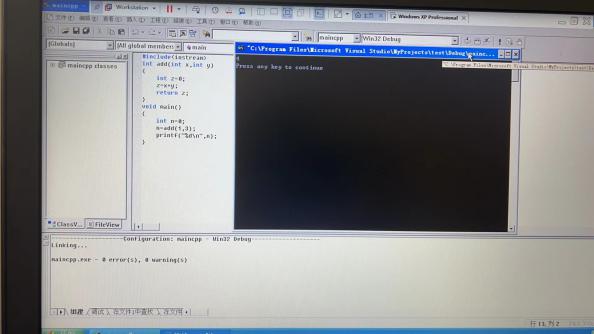
IDE反汇编实验

**实验要求：**

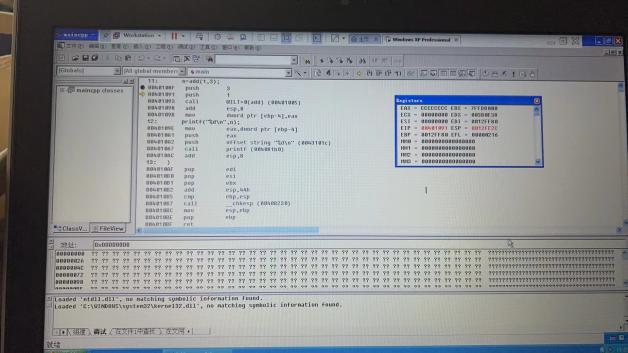
根据第二章示例2-1，在XP环境下进行VC6反汇编调试，熟悉函数调用、栈帧切换、CALL和RET指令等汇编语言实现，将call语句执行过程中的EIP变化、ESP、EBP变化等状态进行记录，解释变化的主要原因。

**实验过程：**

1. 进入VC反汇编



先在vmware中打开visual c++，然后将c语言下的代码输入，运行（如上左图）可以运行，然后按F9在add语句处添加断点，然后按F5进入调试界面（上右图），然后通过右键中的go to disassembly进入反汇编界面（如下右图）。开始进行实验内容。

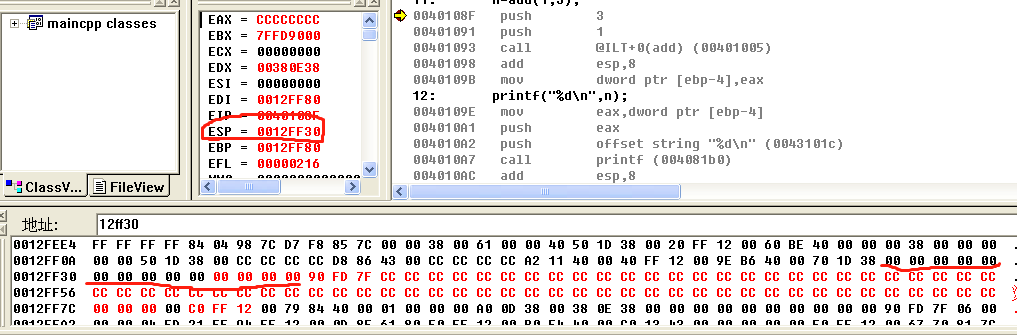


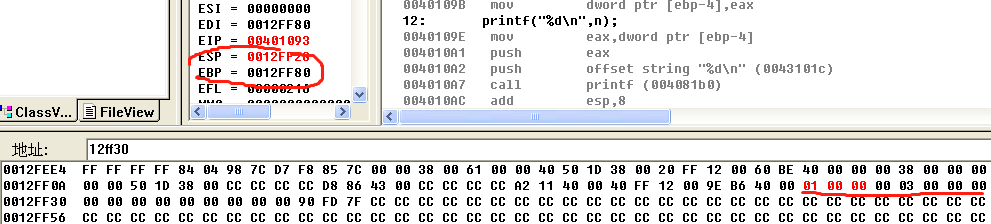
2. 观察add函数调用前后语句

调用add函数语句是n=add(1,3),这句语句将要调用add函数进行指令执行。

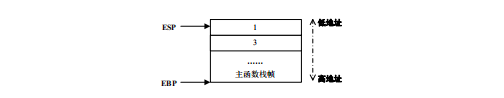
1. 在调用函数之前要先进行传参，对应的反汇编代码是push 3，push 1.传参时的顺序是从右到左，所以需要先push3然后push2.然后调试进行验证。1和4都是占4bytes。

Eip的内容是要执行的下一条指令地址，在要执行push 3时，eip内容是0040108F，要执行push 1时eip内容是00401091，都是要执行的下一条指令的地址。





观察上面的前后对比图（红色下划线的部分），发现先后将3和1进栈，esp从12FF30变到12FF28。3和1分别占用4bytes，一共占了8bytes。3是先进栈的，地址要比后进栈的1高，可见地址是由高到低存参数。传参之后的栈区图下图所示。



1. 在调用add函数之后要将esp先恢复传参之前的原位，然后将函数调用后在寄存器中存储的结果复制给main函数的n。对应的反汇编代码的是：

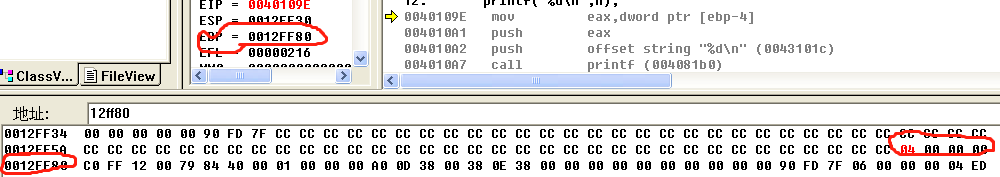
Add esp，8

Mov dword ptr [ebp-4],eax

第一句是将函数调用后的esp恢复原位，在函数调用之前传了3和1两个参数，一共

8bytes，由于原来传参地址是从高到低，所以此时要将esp的地址加8。

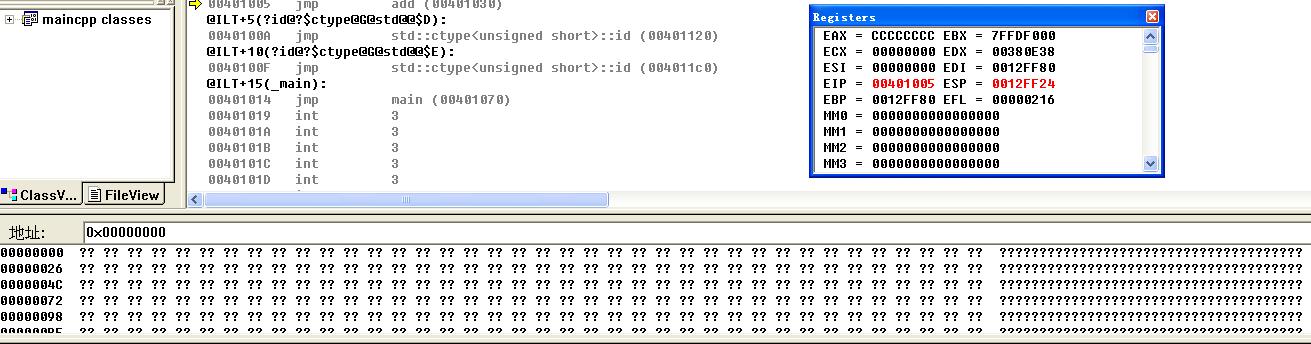
第二句是将函数调用后的返回值赋值给n，函数调用后返回值存在eax寄存器中，而原来的n作为main函数的局部变量，在上文中已经存在栈中，由于地址是从高到低，所以局部变量n是在ebp-4的位置上，dword ptr是重载定义的变量类型，类似于c++的强制数据类型转换，为了保证是dword型存入栈。Eip的内容同理是要执行下一条指令的地址。在call语句执行完成后，eip执行printf语句，内容是下一条的mov指令地址，是0040109E。下面进行调试验证。



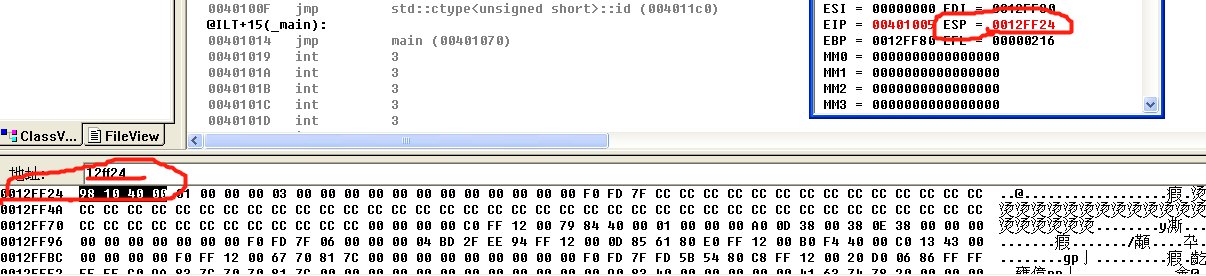
观察上图红线部分，发现esp又变成了12FF30，恢复到原位，同时观察ebp-4的位置，发现数据是4，是调用add函数进行1和3的加法的结果。所以上述语句解释是正确的。

3. add函数内部栈帧切换等关键汇编代码

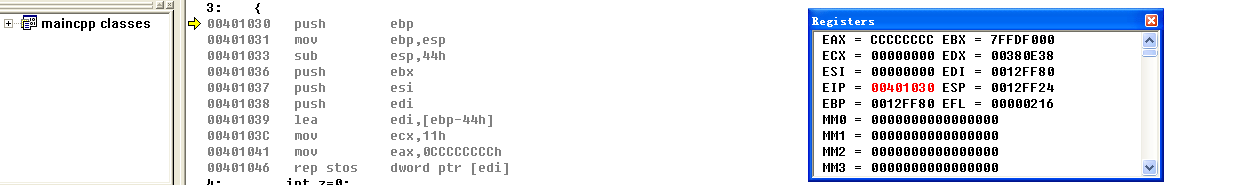
重新开启一遍调试过程，指到call语句的时候按F11，进入下图界面



发现esp值变为12FF24，于是读取该地址的内容为00401098，恰好为add函数调用完要执行的下一条指令地址，就是call指令的第一条操作，返回地址入栈。如下图



再按F11进入跳转，跳转到地址为00401030的反汇编指令（add函数的入口地址），此部分为add函数部分的指令，这一部是call的第二条操作，跳转到调用函数指令段。

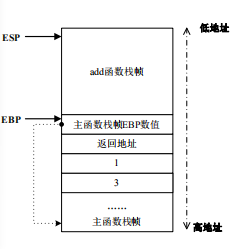


Push ebp

Mov ebp，esp

Sub esp，44h

这段代码是将将EBP入栈，存储了原栈帧的ebp留着为了以后结束add函数调用时可以返回原栈帧。然后并将ESP的值赋给EBP，切换到新栈帧。此时esp和ebp的内容都是0012FF20.随后，esp的值减去44h，更新栈帧顶部，为add函数分配地址空间，得到栈大小为0CCH。Esp的内容变为0012FEDC。此时栈帧如下图所示。



Push ebx

Push esi

Push edi

分别依次把ebx，esi，edi入栈，ebx 作为内存偏移指针使用 ，esi 是源地址指针寄存器 ，edi 是目的地址指针寄存器。根据x86 cdecl调用约定，将这三个寄存器规定为非易失寄存器。所以采用将这三个寄存器入栈。

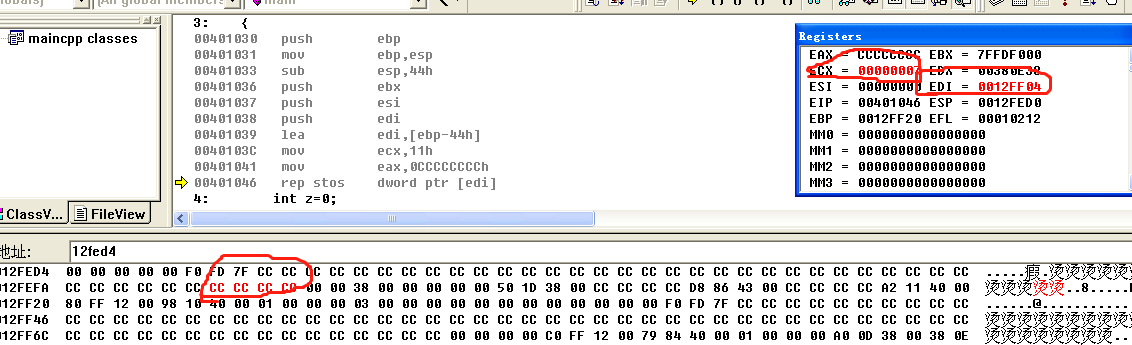
Lea edi，[ebp-44h] //将 ebp-0CCh 地址装入 EDI

Mov ecx,11h //设置计数器数值，即将 ECX 寄存器赋值为 11h

Mov eax,0CCCCCCCCh //向寄存器 EAX 赋值

Rep stos dword ptr [edi] //循环将栈区数据都初始化为 CCh

这段代码是一共初始化的过程，利用循环将循环次数设置成11h次，然后每次对4bytes进行赋值，直到把44h大小栈帧空间全部赋值成CCh。然后进行调试验证，连续按F10直到下一条执行指令的箭头指向rep指令语句，然后按F11，发现每按一次ecx减一，edi加4，下面的地址空间有4bytes被赋值成CCh。直到ecx为0时停止循环，整个0CCh空间都被初始化为0CCh.下图为循环过程截图。



然后执行函数体，反汇编代码如下：

mov dword ptr [ebp-4],0 ；将局部变量 z 初始化为 0

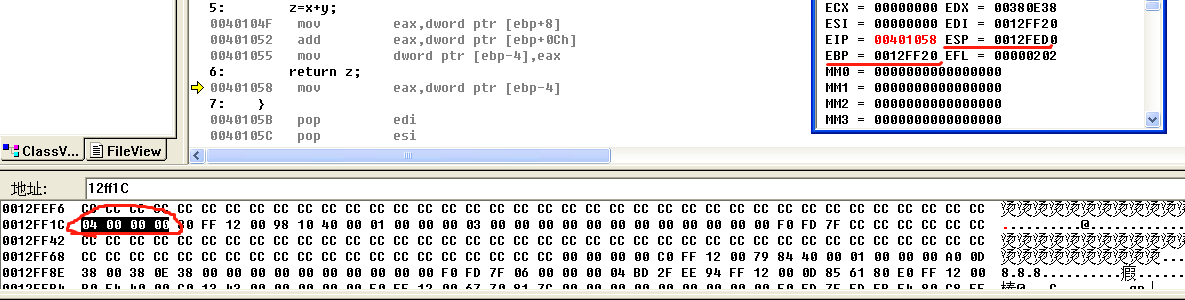
mov eax,dword ptr [ebp+8]；将寄存器 EAX 的值设置为形参 x 的值

add eax,dword ptr [ebp+0Ch] ；将寄存器 EAX 累加形参 y 的值

mov dword ptr [ebp],eax；将 EAX 的值复制给 z

mov eax,dword ptr [ebp-4]；将 z 的值存储到 EAX 寄存器中

这个过程是函数体的执行过程，已经处理好的ebp和esp内容不发生变化，esp为0012FED0，ebp为0012FF20.局部变量z的地址是0012FF1C，如下图所示。最后要将结果存入eax寄存器，存放函数的返回值。



然后要进行恢复但main函数的栈状态，对应的反汇编代码是：

pop edi //恢复寄存器值

pop esi //恢复寄存器值

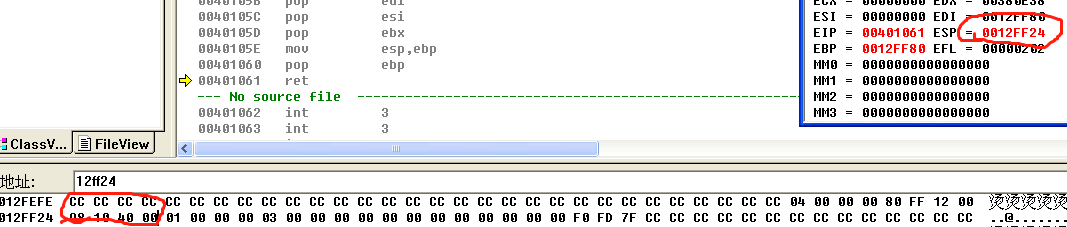
pop ebx //恢复寄存器值

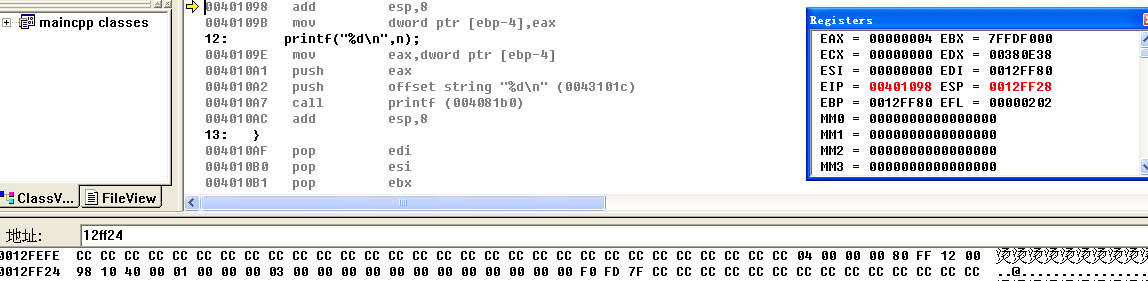
mov esp,ebp //恢复寄存器值

pop ebp //恢复寄存器值

ret //根据返回地址恢复 EIP 值，相当于 pop EIP

与上文相对应，将edi，esi，ebx分别出栈，然后将ebp值赋给esp，这一步是将add函数的栈帧空间取消掉，然后将ebp出栈，返回到原main函数栈帧的ebp位置。然后是关键的ret指令，连续按F10直到箭头指向ret指令，此时的esp为0012FF20,查看该地址发现这是add函数调用结束后的下一条指令地址。然后按F11执行ret指令，eip变为00401098，esp变为0012FF28，对应到原来栈帧。到此add函数调用结束，继续执行下面的指令。调试过程如下。





补：eip在执行过程中都指向执行的下一条指令。

**心得体会：**

1.我通过实验，掌握了RET指令的用法，RET指令实际就是执行了Pop EIP。

2.我同时更加深刻的了解在函数嵌套调用时，栈帧整个的变化过程，新函数调用时如何保存原函数栈帧的位置，如何在调用函数时回到原函数的栈帧的状态。

3.我通过本实验，掌握了多个汇编语言的用法，例如mov，sub，lea，push，pop，ret等等重要指令的用法。

4我对反汇编程序的分析更加熟练，可以明白调试的原理和快捷键，也能快速的通过反汇编分析分析出代码的执行逻辑。