学习汇编的一个重要的方法就是将汇编代码和 c 代码之间进行转换练习，这样的练习做的越多，对汇编就越熟悉，同时对 c 代码的理解也更加深刻。很多时候，拿到一个程序的反汇编代码，虽然可能每一行汇编代码都知道什么意思，但却发现对于整个程序干了什么却不知道，原因在于一个简单的程序，翻译成汇编的代码量相对于 c 程序而言多了好多，而且因为汇编代码和人的思维差别比较大，所以，对于整个逻辑的理解就很困难。然而如果经常做一些将汇编翻译成 c 代码的练习，那么你渐渐就会发现，汇编代码其实也很都有规律，看到这一段汇编代码，你就很自然的联想到它对应的 c 代码。

下面我们就开始做一下这样的练习，首先看一下下面的汇编代码：

第一段第代码

```
00401020  push      ebp                 ; 保存 epb
00401021  mov       ebp,esp             ; 将 ebp 指向栈顶
00401023  sub       esp,4Ch             ;为局部变量分配空间，sub esp, xxx 相当于多个 push
00401026  push      ebx                 ; 保存 ebx
00401027  push      esi                 ; 保存 esi
00401028  push      edi                 ; 保存 edi, 上面 3 个寄存器在使用之前必须保存
00401029  lea       edi,[ebp-4Ch]       ; 将刚刚分配的局部空间的地址送到 edi
0040102C  mov       ecx,13h             ; ecx 这里是循环次数=4ch/4h =13h
00401031  mov       eax,0CCCCCCCCh      ; 将 4 个 int 3 指令放入 eax
00401036  rep stos    dword ptr [edi]   ; 将分配的局部变量空间都用 int 3 指令填充
00401038  mov       dword ptr [ebp-4],3
0040103F  mov       dword ptr [ebp-8],4
00401046  mov       eax,dword ptr [ebp-8]
00401049  push      eax
0040104A  mov       ecx,dword ptr [ebp-4]
0040104D  push      ecx
0040104E  call      @ILT+0(add) (00401005)
00401053  mov       dword ptr [ebp-0Ch],eax
00401056  mov       edx,dword ptr [ebp-0Ch]
00401059  push      edx
0040105A  mov       eax,dword ptr [ebp-8]
0040105D  push      eax
0040105E  mov       ecx,dword ptr [ebp-4]
00401061  push      ecx
00401062  push      offset string "%d+%d=%d\n" (0042201c)
00401067  call      printf (004010d0)
0040106C  add       esp,10h
0040106F  xor       eax,eax
00401071  pop       edi                 ; 恢复 edi
```

```
00401072  pop     esi                 ; 恢复 esi
00401073  pop     ebx                 ; 恢复 ebx
00401074  add     esp,4Ch             ;平衡栈空间，这说明了上面调用的函数是 cdecl call
00401077  cmp     ebp,esp
00401079  call    __chkesp (00401150);这里是 vc 的检查栈是否平衡，应该是 vc 特有的
0040107E  mov     esp,ebp             ; 恢复函数调用之前的那个 esp
00401080  pop     ebp                 ; 恢复 ebp
00401081  ret                         ;返回，相当于 pop eip

00401005 jmp      add (0040d760)
```

#####第二段代码
下面的红色部分跟上面的红色部分基本一样，这里不再注释。

```
0040D760  push    ebp
0040D761  mov     ebp,esp
0040D763  sub     esp,40h
0040D766  push    ebx
0040D767  push    esi
0040D768  push    edi
0040D769  lea     edi,[ebp-40h]
0040D76C  mov     ecx,10h
0040D771  mov     eax,0CCCCCCCCh
0040D776  rep stos  dword ptr [edi]
0040D778  mov     eax,dword ptr [ebp+8]     ; 取得第一个参数
0040D77B  add     eax,dword ptr [ebp+0Ch]   ; 将第二个参数和第一个参数相加，结果保存在 eax 中，所有的函数调用，无论是系统函数，还是自定义函数，结果都会保存在 eax 中返回的
0040D77E  pop     edi
0040D77F  pop     esi
0040D780  pop     ebx
0040D781  mov     esp,ebp
0040D783  pop     ebp
0040D784  ret     8; 这里的 ret 8 相当于 pop eip; add esp, 8;从这里面也可以看出，该函数是内部平衡栈空间的，是 stdcall 的调用方式
```

对于那些汇编大牛来说，估计随便扫一下就知道这段代码是干什么的。红色部分一看就知道是用 vc 编译出来的程序，非常的典型。如果你不信的话， 可以把每个用 vc(我这里用的是 vc6，其它版本是否有变化，没有试过，估计差不多)写的程序反汇编看一下，基本上都会有红色部分的开头和结尾。这是两段汇编代码，比较一下这两段汇编代码， 开头和结尾的部分是不是很像：

开头部分：

```
00401020  push    ebp
00401021  mov     ebp,esp
00401023  sub     esp,4Ch
00401026  push    ebx
00401027  push    esi
00401028  push    edi
00401029  lea     edi,[ebp-4Ch]
0040102C  mov     ecx,13h
00401031  mov     eax,0CCCCCCCCh
00401036  rep stos  dword ptr [edi]
```

```
0040D760  push    ebp
0040D761  mov     ebp,esp
0040D763  sub     esp,40h
0040D766  push    ebx
0040D767  push    esi
0040D768  push    edi
0040D769  lea     edi,[ebp-40h]
0040D76C  mov     ecx,10h
0040D771  mov     eax,0CCCCCCCCh
0040D776  rep stos  dword ptr [edi]
```

开头部分除了和 4ch 和 40h 这种常量相关的代码不一样外，其它的一模一样。这里面的 4ch 和 40h 都是局部变量所占空间的大小，第一个要大一些， 第二个小一些。这两段开头可以说是 vc 写的程序的经典开头，很多其它工具写的程序也有类似的开头。它们的功能如下：

1. 保存 ebp 寄存器
2. 将 esp 最初的值保存在 ebp 中，后面可以用 mov esp, ebp 进行恢复，无论中间 esp 的值怎么改变，都可以保证最后的 esp 值是正确的。这里有一个非常重要的前提是 ebp 的值在此过程中不能被更改
3. 分配局部变量的空间，通过 sub esp, xxxx 预留空间
4. 保存三个非常重要的寄存器 ebx, esi, edi
5. 将 int 3 指令填充到刚刚分配的整个局部变量空间, int 3 指令是一个中断指令，如果被执行，程序将会被中断，提示出错，因为局部变量变量空间都是用来保存数据的，如果被当作指令执行，显然是错误的

结尾部分，差别稍微大一点，这个跟函数的调用方式有关，但从总体上来看，都要 pop 出三个寄存器 edi, esi, ebx， 并且最后都要恢复 esp 和 ebp:

```
00401071  pop     edi
00401072  pop     esi
00401073  pop     ebx
00401074  add     esp,4Ch
00401077  cmp     ebp,esp
00401079  call    __chkesp (00401150)
0040107E  mov     esp,ebp
00401080  pop     ebp
00401081  ret
```

```
0040D77E  pop     edi
0040D77F  pop     esi
0040D780  pop     ebx
0040D781  mov     esp,ebp
0040D783  pop     ebp
0040D784  ret     8
```

结尾部分尽管随着调用方式的不同，如 stdcall 和 cdecl 方式，但总会做以下的事情：

1. 恢复三个重要寄存器 edi, esi, ebx
2. 恢复最开始的 esp 值
3. 恢复最开始的 ebp 值
4. 返回，如果是 cdecl 调用方式，通常是 ret 直接返回，如果是 stdcall 方式则通常会在 ret 后面加一个常数。因为 cdecl 调用方式是调用者平衡栈空间，所以在函数返回的时候，不需要自己来平衡栈空间；而 stdcall 则规定有被调用者自己来平衡栈空间，这样，如果有一个参数，则 ret 后面是 4， 有 2 个则是 8， 有 n 个则是 n*4 个字节，要是该函数没有参数的话，则看起来跟 cdecl 一样，也是一个 ret 直接返回。

因为开头部分和结尾部分基本上都是相同的，而且几乎每个函数都有这样的开头和结尾，所以，这里我们可以忽略这部分，只看中间的处理部分：

先看第一段代码：

FF...FFF 高地址	
.....	
ebp	原来的 ebp 地址
ebp-4 3	现在 ebp 指向的地址 局部变量空间开始
ebp-8 4	
ebp-c	
Int3 int 3 int 3 int 3	局部变量空间被 int3 填充
.....	
ebp-4c	局部变量空间结束
ebx(ebp-4c-4)	
esi(ebp-4c-8)	
edi(ebp-4c-c)	
.....	
.....	
00...000 低地址	

mov dword ptr [ebp-4],3和
mov dword ptr [ebp-8],4执行完毕之后，我们将看到的栈空间如左图所示

由上面的代码分析以及栈空间示意图，我们可以想到，这里到 mov dword ptr [ebp-8],4 指令为止的地方，是给两个临时变量赋值，我们假设临时变量为 x 和 y，则上述对应的 c 代码应该类似下面的代码：

```
int x, y;
x = 3;
y = 4;
```

这里是 4 个字节的数据，为什么要写成 int，而不写其它的数据类型呢？后面会有解释。

接着往下看：

```
00401046   mov       eax,dword ptr [ebp-8]
00401049   push      eax
0040104A   mov       ecx,dword ptr [ebp-4]
0040104D   push      ecx
0040104E   call      @ILT+0(add) (00401005)
00401053   mov       dword ptr [ebp-0Ch],eax
```

这里将两个局部变量的值分别放到 eax 和 ecx 中，然后再 push 到栈中，接着是一个 call 指令，显然这里的入栈是为了为 call 调用准备参数的，这 2 个参数就是 3 和 4.看看 call 指令的地方 call @ILT+0(add) (00401005)，这里可以看出是调用了一个叫做 add 的函数，可以猜测出是将 2 个数进行相加的函数，函数的地址是 00401005, 而我们看看这个地址的是什么：00401005 jmp add (0040d760) 这是一个无条件跳转的代码，跳转到 0040d760 这个地址，而这个地址则是第二段代码的起始地址，可以看出，第二段代码是 add 函数的实现。不知道大家有没有注意到这个问题， call 指令为什么不直接跳到 add 函数的实际地址，而是跳转到另外一个地址，而在另外一个地址则是一个 jmp 指令跳转到 add 的实际地址，为什么要多这么一道跳转呢？我猜想应该是 call 指令跳转的是导入地址表的地方，而导入地址表的地方才是真正的函数地址，不知道对不对?

接着往下看：

```
00401053   mov       dword ptr [ebp-0Ch],eax
```

显然这里是将 call 函数的返回值放入到局部变量空间 ebp-0ch 中。到目前为止，我们可以得出对应的 c 代码应该是：

```
int x, y, result;
x = 3;
y = 4;
result = add(x, y);
```

接着往下看：

```
00401056   mov       edx,dword ptr [ebp-0Ch]
00401059   push      edx
0040105A   mov       eax,dword ptr [ebp-8]
```

```
0040105D   push      eax
0040105E   mov       ecx,dword ptr [ebp-4]
00401061   push      ecx
00401062   push      offset string "%d+%d=%d\n" (0042201c)
00401067   call      printf (004010d0)
0040106C   add       esp,10h
0040106F   xor       eax,eax
```

这里先把 add 函数计算的结果(ebp-0ch)放入 edx 中，把 y 的值(ebp-8)放入 eax 中，把 x 的值(ebp-4)放入 ecx 中，3 个 push 指令分别将入栈，接着将字符串"%d+%d=%d\n"的地址入栈，然后调用 printf 函数，这里我们已经看的很明白了：通过字符串里面的格式化，我们可以看出三个参数的类型都是 int 的，printf 我们都知道，是不定参数的函数，参数传递方式为__cdecl 的方式，参数从右到左，由调用者平衡栈空间，后面的 add esp, 10h 也说明了这一点, 总共 4 个参数，3 个整形参数，12 个字节，一个字符串地址，即指针，4 个字节，这样一来，总共 16 个字节，也就是 10h 个字节。xor eax, eax 将 eax 清零。接着是恢复栈平衡，前面已经分析过了。到目前为止，可以看到的 c 语言代码应该是：

```
int x, y, result;
x = 3;
y = 4;
result = add(x, y);
printf("%d+%d=%d\n", x, y, result);
return 0;
```

对于 add 函数的实现，真正计算的代码只有 2 行：

```
0040D778   mov       eax,dword ptr [ebp+8]
0040D77B   add       eax,dword ptr [ebp+0Ch]
```

将 ebp+8 第一个参数的值放到 eax，再将第一个参数的值和 eax（第一个参数）相加，结果保存在 eax 中。而 eax 通常是函数的返回值存放的地方，我们据此可以大致看出对应的 c 代码应该类似如此的形式：return x + y;

我们看看栈的空间：

FF…FFF 高地址	
……	
ebp+0c 4	参数 y 的地址
ebp+8 3	参数 x 的地址
ebp+4	Add 函数的返回地址
ebp	保存的 ebp
Ebp-4	新的 ebp 指向的地方 局部变量空间开始
…….int 3, int 3, int3,int3	
ebp-40h	局部变量空间结束
ebx(ebp-40h-4)	
esi(ebp-40h-8)	
edi (ebp-40h-c)	
……..	
…….	
00…000 低地址	

这里我们需要注意的是，参数的位置是以 add 函数中的 ebp 作为参考的，不再是调用 add 函数之前的那个 ebp 了。 从这里我们可以总结出一个规律：

1. 在一个函数内部，对传递过来的参数的引用是 ebp+xxxx 的形式，一般情况下，ebp+4 是返回地址，ebp+8 是第一个参数，ebp+c 是第二个参数，以此类推，第 n 个参数是 ebp+4*n+4
2. 在一个函数的内部，对局部变量的引用是 ebp-xxx 的形式，按照声明的顺序，第一个是 ebp-4，第二个是 ebp-8， 以此类推，第 n 个局部变量是 ebp-n*4

这里 add 的函数有 2 个参数，而在 ret 的后面有常量 8，正好是函数内部平衡，所以 可以确定这里的参数调用方式为 stdcall，据此，我们可以写出 add 函数的实现是：

```
int __stdcall add(int x, int y)
{
    return x + y;
}
```

到此为止，我们所有的细节基本分析完毕，完整的c代码如下：

```
#include <stdio.h>

int __stdcall add(int x, int y)
{
	return x + y;
}

int main()
{
	int x, y, result;

	x = 3;
	y = 4;

	result = add(x, y);
	printf("%d+%d=%d\n", x, y, result);

	return 0;
}
```

如此一个简单的程序， 对应的汇编代码确是一大堆，所以，我们可以想象出那些汇编牛人看汇编代码绝对不会是一行一行从头到尾的慢慢看，而是一片一片的看，找关键部分的看。像文中的红色部分，具有明显的 vc 编译特征，看的时候完全可以略过，这样剩下的部分再慢慢分析，显然要比从头到尾一行一行的看有效果。但是要达到这种水平，并非一朝一夕，这就需要我们平时写 c 代码的时候，要经常看看反汇编代码，多总结一下 c 语言代码对应的反汇编代码有啥特点，比如调用函数的特点、if, else, while, do .. while , for, switch 等控制流程的语句对应的汇编代码的特点以及其它的比较常用的函数段的特点。 作为初学，我们可以自己写一些小段程序，然后看反汇编代码，进行总结，这样做的多了，就慢慢的熟悉了。

好了，用了 3 个多小时，终于写完了。文中的那个疑惑如果有大侠帮忙解释一下，感激不尽了。