第 2 章　汇编讲解

2.1 指令格式

操作码字段　　地址码字段

操作码字段：表征指令的操作特性与功能　（指令的唯一标识）　不同的指令操作码不能相同
地址码字段：指定参与操作的操作数的地址码

指令中指定操作数存储位置的字段称为地址码，地址码中可以包含存储器地址。也可包含寄存器编号。
指令中可以有一个、两个或者三个操作数，也可没有操作数，根据一条指令有几个操作数地址，可将指令分为零地址指令。一地址指令、二地址指令、三地址指令。4 个地址码的指令很少被使用

操作码字段	地址码	
操作码	A1　A2　A3	三指令地址
操作码	A1　A2	二指令地址
操作码	A1	一指令地址
操作码		零指令地址

零地址指令：只有操作码，没有地址码（空操作　停止等）

一地址指令：指令编码中只有一个地址码，指出了参加操作的一个操作数的存储位置，如果还有另一个操作数则隐含在累加器中

eg: INC AL　自加指令

二地址指令：指令编码中有两个地址，分别指出了参加操作的两个操作数的存储位置，结果存储在其中一个地址中

（op a1,a2:a1 op a2 a1）

eg:　MOV　AL,BL

　　ADD　AL,30

三地址指令：指令编码中有 3 个地址码，指出了参加操作的两个操作数的存储位置和一个结果的地址

（op a1,a2,a3: a1 op a2　a3）

二地址指令格式中，从操作数的物理位置来说有可归为三种类型

寄存器-寄存器（RR）型指令：需要多个通用寄存器或个别专用寄存器，从寄存器中取操作数，把操作结果放入另一个寄存器，机器执行寄存器-寄存器型的指令非常快，不需要访存。

寄存器-存储器（RS）型指令：执行此类指令时，既要访问内存单元，又要访问寄存器。

存储器-存储器（SS）型指令：操作时都是涉及内存单元，参与操作的数都是放在内存里，从内存某单元中取操作数，操作结果存放至内存另一单元中，因此机器执行指令需要多次访问内存。

复杂指令集：变长　x86
精简指令集：等长　arm

2.2 编译过程及 VS 生成汇编方法

掌握把 C 语言转换汇编的手法，那么就不会再害怕汇编了

编译过程

.c-->预处理---》编译--》.s 文件，也就是我们的汇编文件---》汇编—》main.obj--》链接--》可执行文件 exe

VS 生成汇编的步骤

1、如图 2.2.1 所示，右键项目，点击属性

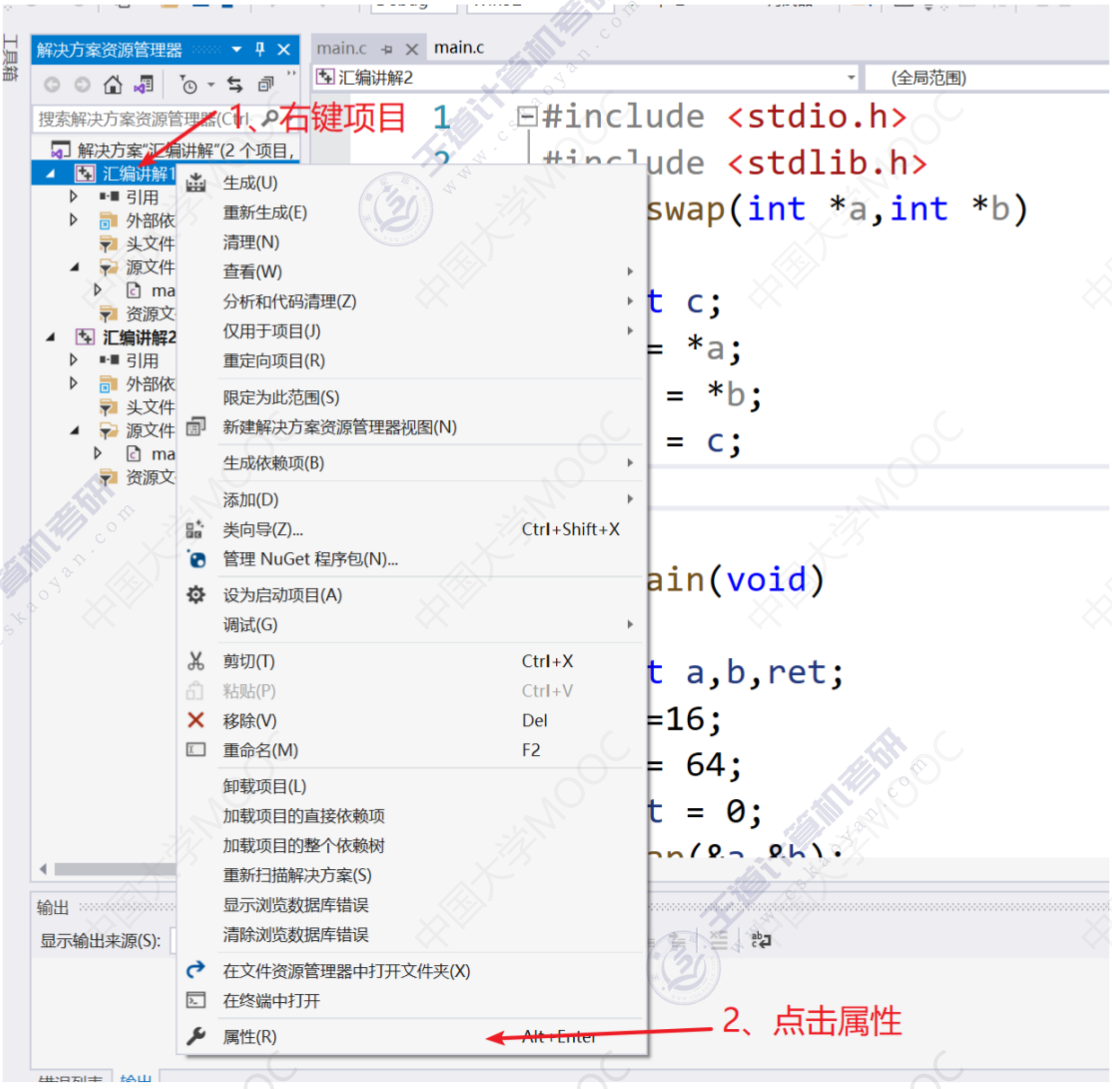

图 2.2.1 右键项目点击属性

2、如果 2.2.2 所示，选择输出文件，汇编程序输出选择程序集、机器码和源代码

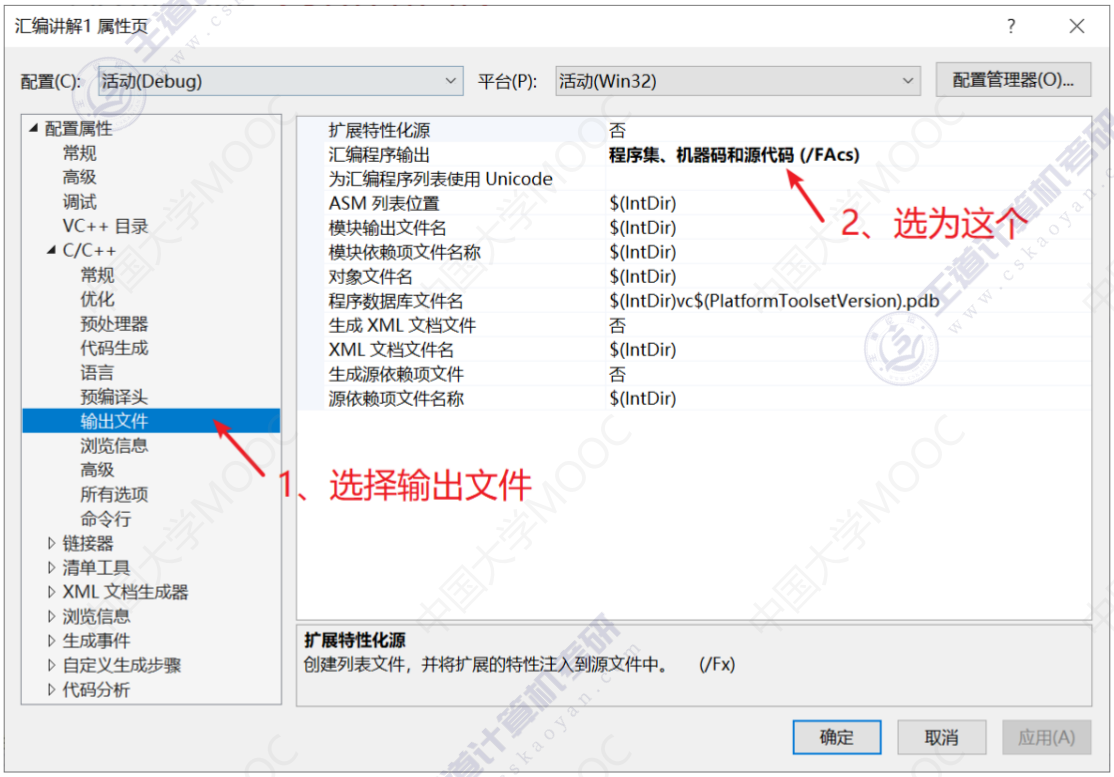

图 2.2.2 选择输出文件

2.3 理解数组与指针对应的汇编

2.3.1 相关寄存器

通用寄存器

31 16	15 8	7 0
	AH	AL
	BH	BL
	CH	CL
	DH	DL
ESI		
EDI		
EBP		
ESP		

16bit	32bit	说明
AX	EAX	累加器（Accumulator）
BX	EBX	基地址寄存器（Base Register）
CX	ECX	计数寄存器（Count Register）
DX	EDX	数据寄存器（Data Register）
	ESI	变址寄存器（Index Register）
	EDI	
	EBP	堆栈基指针（Base Pointer）
	ESP	堆栈顶指针（Stack Pointer）

除 EBP 和 ESP 外，其他几个寄存器的用途是比较任意的。

2.3.2 常用指令

汇编指令通常可以分为数据传送指令、逻辑计算指令和控制流指令，下面以 Intel 格式为例，介绍一些重要的指令。以下用于操作数的标记分别表示寄存器、内存和常数。

- <reg>：表示任意寄存器，若其后带有数字，则指定其位数，如<reg32>表示 32 位寄存器（eax、ebx、ecx、edx、esi、edi、esp 或 ebp）；<reg16>表示 16 位寄存器（ax、

bx、cx 或 dx)；<reg8>表示 8 位寄存器（ah、al、bh、bl、ch、cl、dh、dl)。

- <mem>：表示内存地址（如[eax]、[var+4]或 dword ptr [eax+ebx])。
- <con>：表示 8 位、16 位或 32 位常数。<con8>表示 8 位常数；<con16 >表示 16 位常数；<con32>表示 32 位常数。

(1) 数据传送指令

1）**mov 指令**。将第二个操作数（寄存器的内容、内存中的内容或常数值）复制到第一个操作数（寄存器或内存)。但不能用于直接从内存复制到内存。

其语法如下：

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<con>
mov <mem>,<con>
```

举例：

```
mov eax, ebx                    #将 ebx 值复制到 eax
mov byte ptr [var], 5           #将 5 保存到 var 值指示的内存地址的一字节中
```

2）**push 指令**。将操作数压入内存的栈，常用于函数调用。ESP 是栈顶，压栈前先将 ESP 值减 4（栈增长方向与内存地址增长方向相反)，然后将操作数压入 ESP 指示的地址。

其语法如下：

```
push <reg32>
push <mem>
push <con32>
```

举例（注意，栈中元素固定为 32 位)：

```
push eax                        #将 eax 值压栈
push [var]                      #将 var 值指示的内存地址的 4 字节值压栈
```

3）**pop 指令**。与 push 指令相反，pop 指令执行的是出栈工作，出栈前先将 ESP 指示的地址中的内容出栈，然后将 ESP 值加 4。

其语法如下：

```
pop edi                         #弹出栈顶元素送到 edi
pop [ebx]                       #弹出栈顶元素送到 ebx 值指示的内存地址的 4 字节中
```

(2) 算术和逻辑运算指令

1）**add/sub 指令**。add 指令将两个操作数相加，相加的结果保存到第一个操作数中。sub 指令用于两个操作数相减，相减的结果保存到第一个操作数中。

它们的语法如下：

```
add <reg>,<reg> / sub <reg>,<reg>
add <reg>,<mem> / sub <reg>,<mem>
add <mem>,<reg> / sub <mem>,<reg>
add <reg>,<con> / sub <reg>,<con>
add <mem>,<con> / sub <mem>,<con>
```

举例：

```
sub eax, 10                     #eax ← eax-10
add byte ptr [var], 10          #10 与 var 值指示的内存地址的一字节值相加，并将结果
                                保存在 var 值指示的内存地址的字节中
```

2）**inc/dec 指令**。inc、dec 指令分别表示将操作数自加 1、自减 1。

它们的语法如下：

```
inc <reg> / dec <reg>
inc <mem> / dec <mem>
```

举例：

```
dec eax                         #eax 值自减 1
```

```
inc dword ptr [var]          #var 值指示的内存地址的 4 字节值自加 1
```

3）**imul 指令**。带符号整数乘法指令，有两种格式：①两个操作数，将两个操作数相乘，将结果保存在第一个操作数中，第一个操作数必须为寄存器；②三个操作数，将第二个和第三个操作数相乘，将结果保存在第一个操作数中，第一个操作数必须为寄存器。

其语法如下：

```
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

举例：

```
imul eax, [var]              #eax ← eax * [var]
imul esi, edi, 25            #esi ← edi * 25
```

乘法操作结果可能溢出，则编译器置溢出标志 OF = 1，以使 CPU 调出溢出异常处理程序。

4）**idiv 指令。**带符号整数除法指令，它只有一个操作数，即除数，而被除数则为 edx:eax 中的内容（64 位整数），操作结果有两部分：商和余数，商送到 eax，余数则送到 edx。

其语法如下：

```
idiv <reg32>
idiv <mem>
```

举例：

```
idiv ebx
idiv dword ptr [var]
```

5）**and/or/xor 指令**。and、or、xor 指令分别是逻辑与、逻辑或、逻辑异或操作指令，用于操作数的位操作，操作结果放在第一个操作数中。

它们的语法如下：

```
and <reg>,<reg> / or <reg>,<reg> / xor <reg>,<reg>
and <reg>,<mem> / or <reg>,<mem> / xor <reg>,<mem>
and <mem>,<reg> / or <mem>,<reg> / xor <mem>,<reg>
and <reg>,<con> / or <reg>,<con> / xor <reg>,<con>
and <mem>,<con> / or <mem>,<con> / xor <mem>,<con>
```

举例：

```
and eax, 0fH                 #将 eax 中的前 28 位全部置为 0，最后 4 位保持不变
xor edx, edx                 #置 edx 中的内容为 0
```

6）**not 指令。**位翻转指令，将操作数中的每一位翻转，即 0→1、1→0。

其语法如下：

```
not <reg>
not <mem>
```

举例：

```
not byte ptr [var]          #将 var 值指示的内存地址的一字节的所有位翻转
```

7）**neg 指令**。取负指令。

其语法如下：

```
neg <reg>
neg <mem>
```

举例：

```
neg eax                      #eax ← -eax
```

8）**shl/shr 指令**。逻辑移位指令，shl 为逻辑左移，shr 为逻辑右移，第一个操作数表示被操作数，第二个操作数指示移位的位数。

它们的语法如下：

```
shl <reg>,<con8> / shr <reg>,<con8>
shl <mem>,<con8> / shr <mem>,<con8>
```

```
shl <reg>,<cl> / shr <reg>,<cl>
shl <mem>,<cl> / shr <mem>,<cl>
```

举例：

```
shl eax, 1              #将 eax 值左移 1 位，相当于乘以 2
shr ebx, cl             #将 ebx 值右移 n 位（n 为 cl 中的值），相当于除以 2^n
```

（3）控制流指令

x86 处理器维持着一个指示当前执行指令的指令指针（IP），当一条指令执行后，此指针自动指向下一条指令。IP 寄存器不能直接操作，但可以用控制流指令更新。通常用标签（label）指示程序中的指令地址，在 x86 汇编代码中，可在任何指令前加入标签。例如，

```
        mov esi, [ebp+8]
begin:  xor ecx, ecx
        mov eax, [esi]
```

这样就用 begin 指示了第二条指令，控制流指令通过标签就可以实现程序指令的跳转。

1）**jmp 指令。**jmp 指令控制 IP 转移到 label 所指示的地址（从 label 中取出指令执行）。其语法如下：

```
jmp <label>
```

举例：

```
jmp begin           #转跳到 begin 标记的指令执行
```

2）***jcondition* 指令。**条件转移指令，依据 CPU 状态字中的一系列条件状态转移。CPU 状态字中包括指示最后一个算术运算结果是否为 0，运算结果是否为负数等。

其语法如下：

```
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)
```

举例：

```
cmp eax, ebx
jle done  #如果 eax 的值小于等于 ebx 值，跳转到 done 指示的指令执行，否则执行下一
          条指令。
```

3）**cmp/test 指令。**cmp 指令用于比较两个操作数的值，test 指令对两个操作数进行逐位与运算，这两类指令都不保存操作结果，仅根据运算结果设置 CPU 状态字中的条件码。

其语法如下：

```
cmp <reg>,<reg> / test <reg>,<reg>
cmp <reg>,<mem> / test <reg>,<mem>
cmp <mem>,<reg> / test <mem>,<reg>
cmp <reg>,<con> / test <reg>,<con>
```

cmp 和 test 指令通常和 *jcondition* 指令搭配使用，举例：

```
cmp dword ptr [var], 10   #将 var 指示的主存地址的 4 字节内容，与 10 比较
jne loop                  #如果相等则继续顺序执行；否则跳转到 loop 处执行
test eax, eax             #测试 eax 是否为零
jz xxxx                   #为零则置标志 ZF 为 1，转跳到 xxxx 处执行
```

4）**call/ret 指令。**分别用于实现子程序（过程、函数等）的调用及返回。

其语法如下：

```
call <label>
ret
```

call 指令首先将当前执行指令地址入栈，然后无条件转移到由标签指示的指令。与其他简单的跳转指令不同，call 指令保存调用之前的地址信息（当 call 指令结束后，返回调用之前的地址）。ret 指令实现子程序的返回机制，ret 指令弹出栈中保存的指令地址，然后无条件转

移到保存的指令地址执行。call 和 ret 是程序（函数）调用中最关键的两条指令。

```
lea   eax, DWORD PTR _arr$[ebp]
```

lea 指令的作用，是 DWORD PTR _arr$[ebp]对应空间的内存地址值放到 eax 中

2.3.3 条件码

编译器通过条件码（标志位）设置指令和各类转移指令来实现程序中的选择结构语句。

（1）条件码（标志位）

除了整数寄存器，CPU 还维护着一组条件码（标志位）寄存器，它们描述了最近的算术或逻辑运算操作的属性。可以检测这些寄存器来执行条件分支指令，最常用的条件码有：

- **CF**：进（借）位标志。最近无符号整数加（减）运算后的进（借）位情况。有进（借）位，CF=1；否则 CF=0。
- **ZF**：零标志。最近的操作的运算结算是否为 0。若结果为 0，ZF=1；否则 ZF=0。
- **SF**：符号标志。最近的带符号数运算结果的符号。负数时，SF=1；否则 SF=0。
- **OF**：溢出标志。最近带符号数运算的结果是否溢出，若溢出，OF=1；否则 OF=0。

可见，**OF 和 SF 对无符号数运算来说没有意义**，而 CF 对带符号数运算来说没有意义。常见的算术逻辑运算指令（add、sub、imul、or、and、shl、inc、dec、not、sal 等）会设置条件码。但有两类指令只设置条件码而不改变任何其他寄存器：**cmp 指令和 sub 指令的行为一样，test 指令与 and 指令的行为一样，但它们只设置条件码，而不更新目的寄存器。**之前介绍过的 ***Jcondition*** 条件转跳指令，就是根据条件码 ZF 和 SF 来实现转跳。

例如 汇编讲解 1 例子中我们的 if 比较，就是通过 SF 为 1，来判断跳转

2.4 函数调用原理详解—汇编手法

每个函数使用的栈空间称为函数帧，也称栈帧。掌握函数在相互调用过程中栈帧的变化（前面在关于指针的章节中我们初步解析了函数调用），对于理解 C 语言中函数的调用原理是非常重要的。

2.4.1 关于栈

首先必须明确的一点是，栈是向下生长的。所谓向下生长，是指从内存高地址向低地址的路径延伸。于是，栈就有栈底和栈顶，栈顶的地址要比栈底的低。

对 x86 体系的 CPU 而言，寄存器 ebp 可称为帧指针或基址指针（base pointer），寄存器 esp 可称为栈指针（stack pointer）。

这里需要说明的几点如下。

（1）ebp 在未改变之前始终指向栈帧的开始（也就是栈底），所以 ebp 的用途是在堆栈中寻址（寻址的作用会在下面详细介绍）。

（2）esp 会随着数据的入栈和出栈而移动，即 esp 始终指向栈顶。

如图 2.2.1 所示，假设函数 A 调用函数 B，称函数 A 为调用者，称函数 B 为被调用者，则函数调用过程可以描述如下：

（1）首先将调用者（A）的堆栈的基址（ebp）入栈，以保存之前任务的信息。

(2) 然后将调用者 (A) 的栈顶指针 (esp) 的值赋给 ebp，作为新的基址（即被调用者 B 的栈底）。

(3) 再后在这个基址（被调用者 B 的栈底）上开辟（一般用 sub 指令）相应的空间用作被调用者 B 的栈空间。

(4) 函数 B 返回后，当前栈帧的 ebp 恢复为调用者 A 的栈顶 (esp)，使栈顶恢复函数 B 被调用前的位置；然后调用者 A 从恢复后的栈顶弹出之前的 ebp 值（因为这个值在函数调用前一步被压入堆栈）。

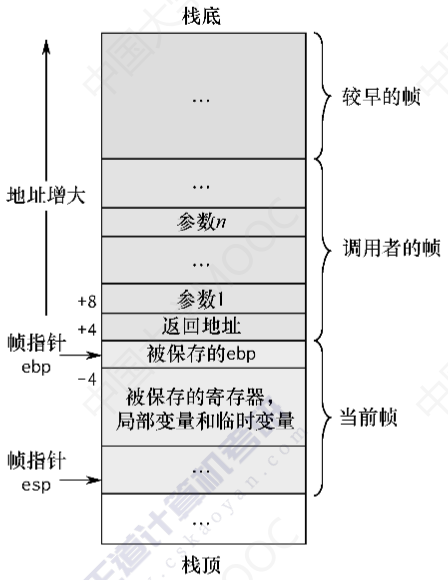

图 2.2.1　函数调用过程

这样，ebp 和 esp 就都恢复了调用函数 B 前的位置，即栈恢复函数 B 调用前的状态。相当于

```
mov   %ebp , %esp      //把 ebp 内的内容复制到 esp 寄存器中
pop   %ebp             //弹出栈顶元素，放到 ebp 寄存器中
```

2.4.2　代码实例分析

函数调用过程的具体代码如例 2.2.1 所示。

【例 2.2.1】函数调用过程的代码。

```
void swap(int *a,int *b)
{
    int c;
    c = *a;
    *a = *b;
    *b = c;
}

int main(void)
{
    int a,b,ret;
```

```
    a =16;
    b = 64;
    ret = 0;
    swap(&a,&b);
    ret = a – b;
    return ret;
}
```

以上代码并不复杂，其实就是 main 函数调用 swap 函数，实现整型变量 a 和 b 的交换。图 2.2.2 是栈指针的实际变化过程。

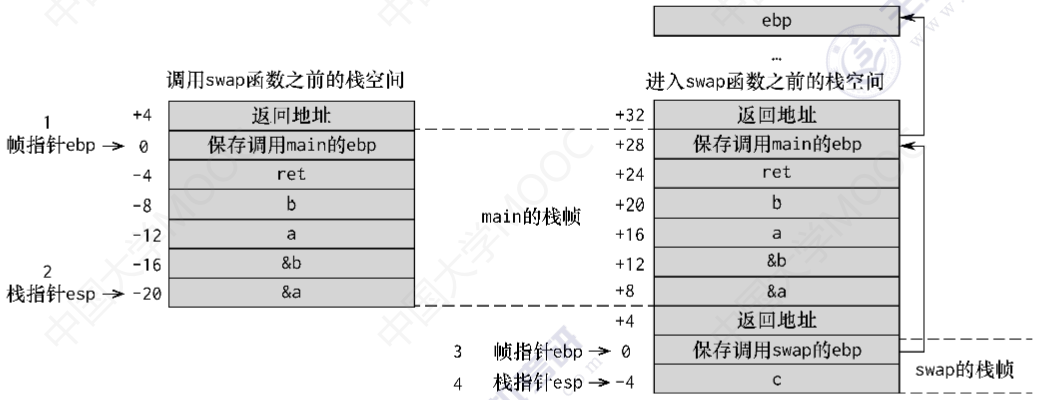

图 2.2.2　栈指针的实际变化过程

首先需要知道可执行程序并不是自动进入内存的，而是其他程序调用的，因此 main 函数虽然是入口函数，但是依然由其他函数调用，所以图 2.2.2 的左边部分是未调用 swap 函数时栈空间的情况，这时帧指针 ebp 保存的是调用 main 的 ebp。有的读者可能不理解这是怎么回事，不过不用担心，其原理与 main 调用 swap 的原理是相同的。接着当 main 函数调用 swap 函数时，我们首先 push %ebp，即把 main 函数的帧指针 ebp（图 2.2.2 中标为 1 的位置）压栈，这时将 main 函数的栈指针 esp（图 2.2.2 中标为 2 的位置）作为新函数 swap 的帧指针 ebp（图 2.2.2 中标为 3 的位置），也就是 main 函数的栈顶指针作为新函数 swap 的基指针（也称帧指针），新函数 swap 的栈顶指针 esp 会随着定义变量的数量依次增加（图 2.2.2 中标为 4 的位置）。

```
c = *a;
```

```
0001e 8b 45 08    mov    eax, DWORD PTR _a$[ebp]    先把 a 内存的地址值给 eax
00021 8b 08       mov    ecx, DWORD PTR [eax]       对 eax 对应地址的数据给 ecx
00023 89 4d f8    mov    DWORD PTR _c$[ebp], ecx    再把 ecx 中的值放到 c 中
```

思考题：学习函数调用原理后，是否明白函数调用时的开销在哪里？直接在 main 函数内实

现变量交换与使用 swap 进行交换有什么区别?

2.4.3 Linux(或者 Mac)下看汇编代码和机器码的方法

如果只看汇编，那么 gcc -S main.c 即可得到 main.s
如果汇编和机器码都需要看
gcc -g -o main main.c
objdump -d main > main.dump

2.5 真题实战

首先我们实战 2017 考研真题的 35 题
接着我们实战 2019 考研真题的 45 题
针对 19 题的 call 调用地址计算，windows 的汇编看不出来，需要 Linux 下，溢出自陷指令是 into，这个了解一下（王道书没介绍这个指令，可以看下教材）