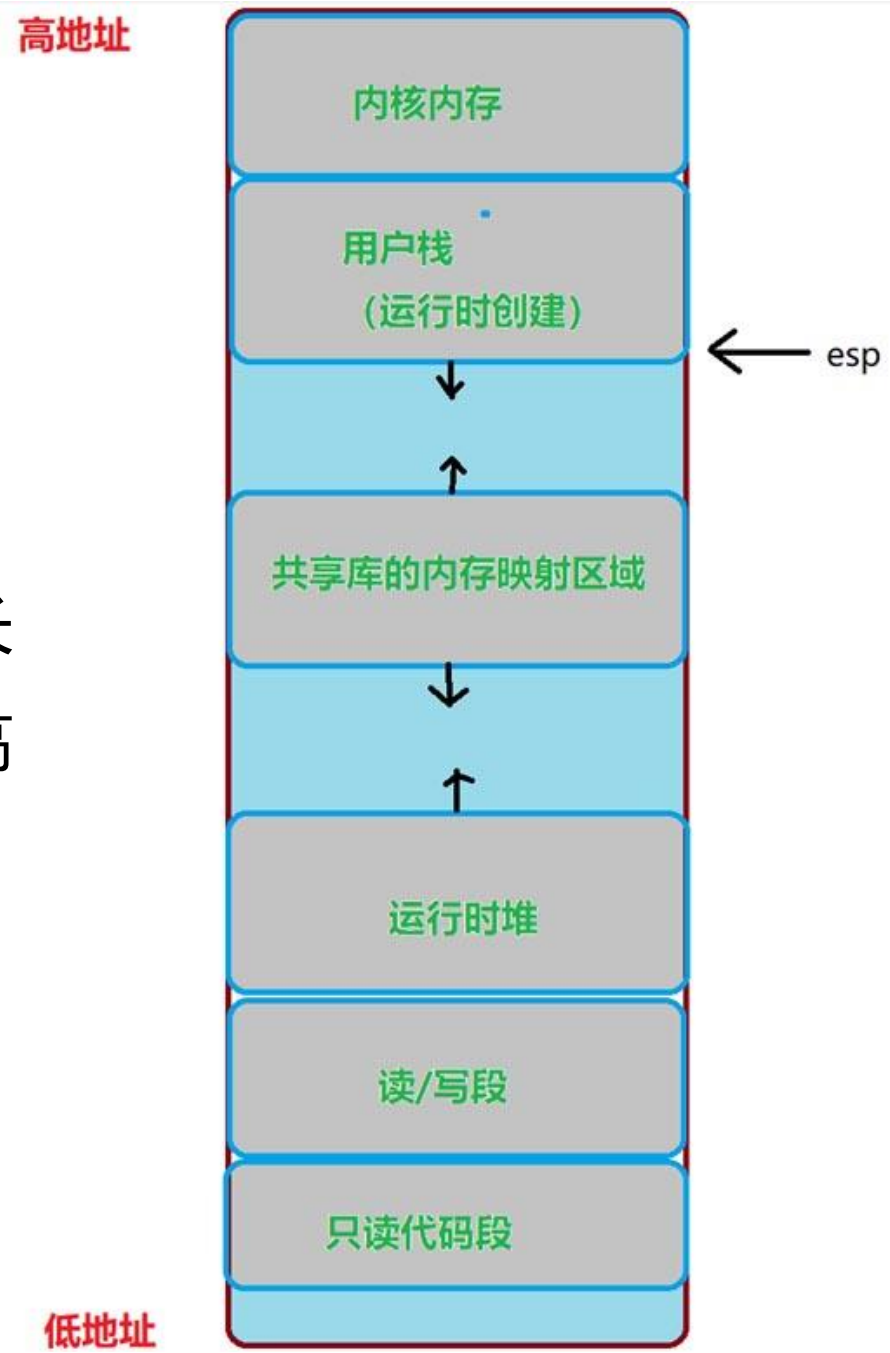
C语言的运行栈

https://blog.csdn.net/cainiaochufa2021/article/details/123693134

原帖题目：深入了解运行时栈（C语言）

计算机系统中的内存布局

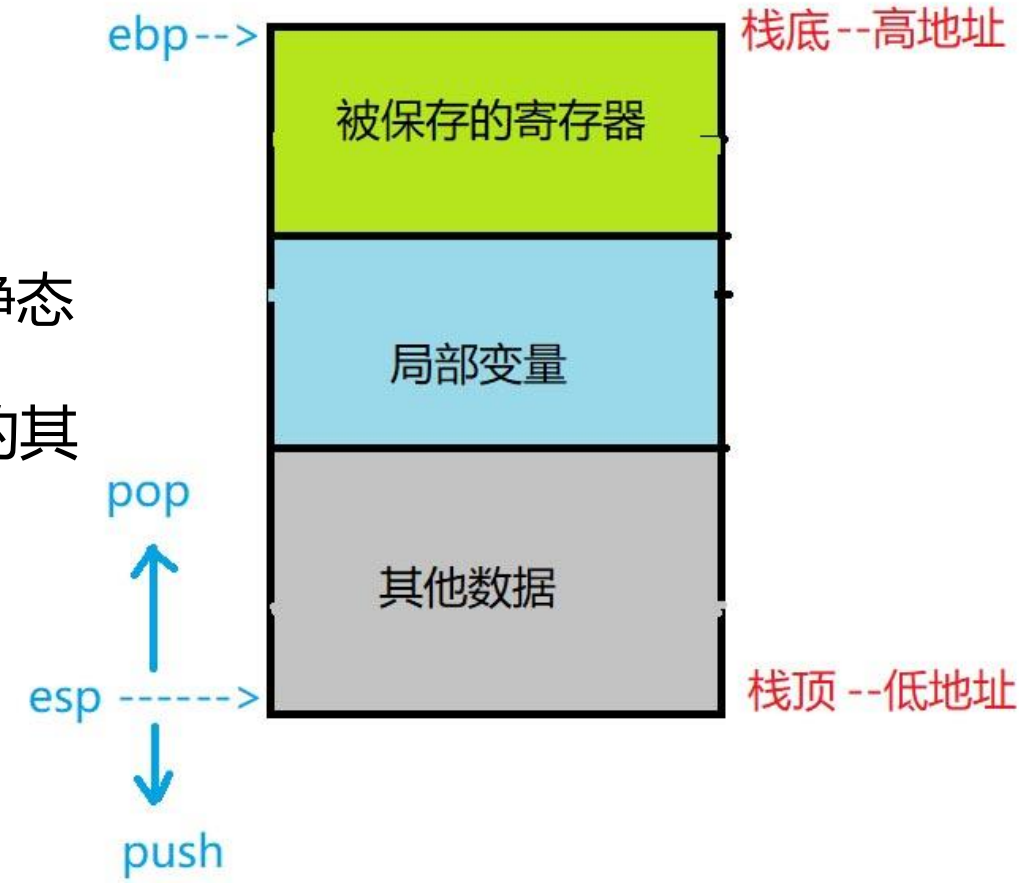
- 栈是由高地址向低地址增长
- 栈顶在低地址处，栈底在高地址处



函数的栈帧

程序在每一次调用函数的时候就会在栈区创建一块空间，这块空间就被称为该函数的栈帧。这块空间中一般包括了下面一些信息：

- 被保存的寄存器
- 临时变量：包括函数的非静态局部变量以及编译器生成的其他临时变量
- 函数的返回地址和参数

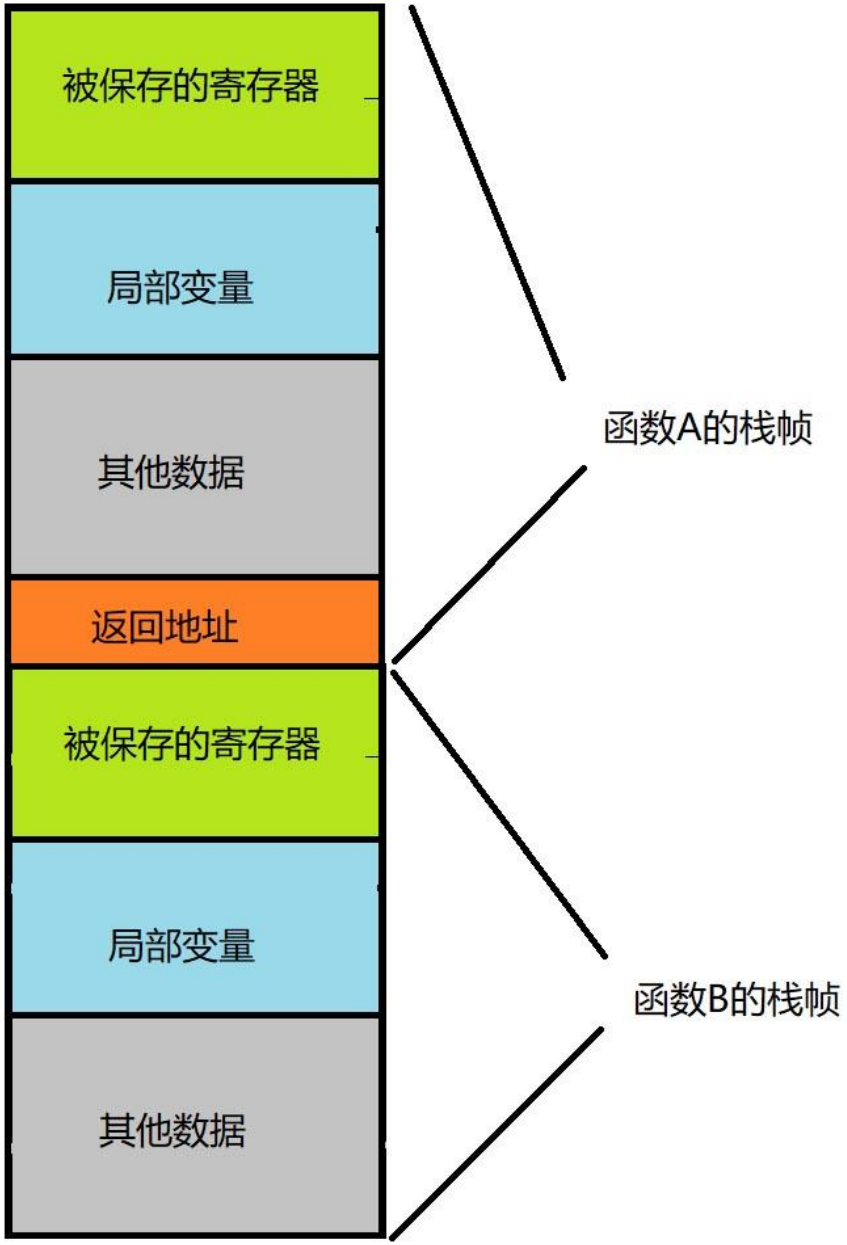


函数调用时的栈帧变化

当函数A调用函数B:

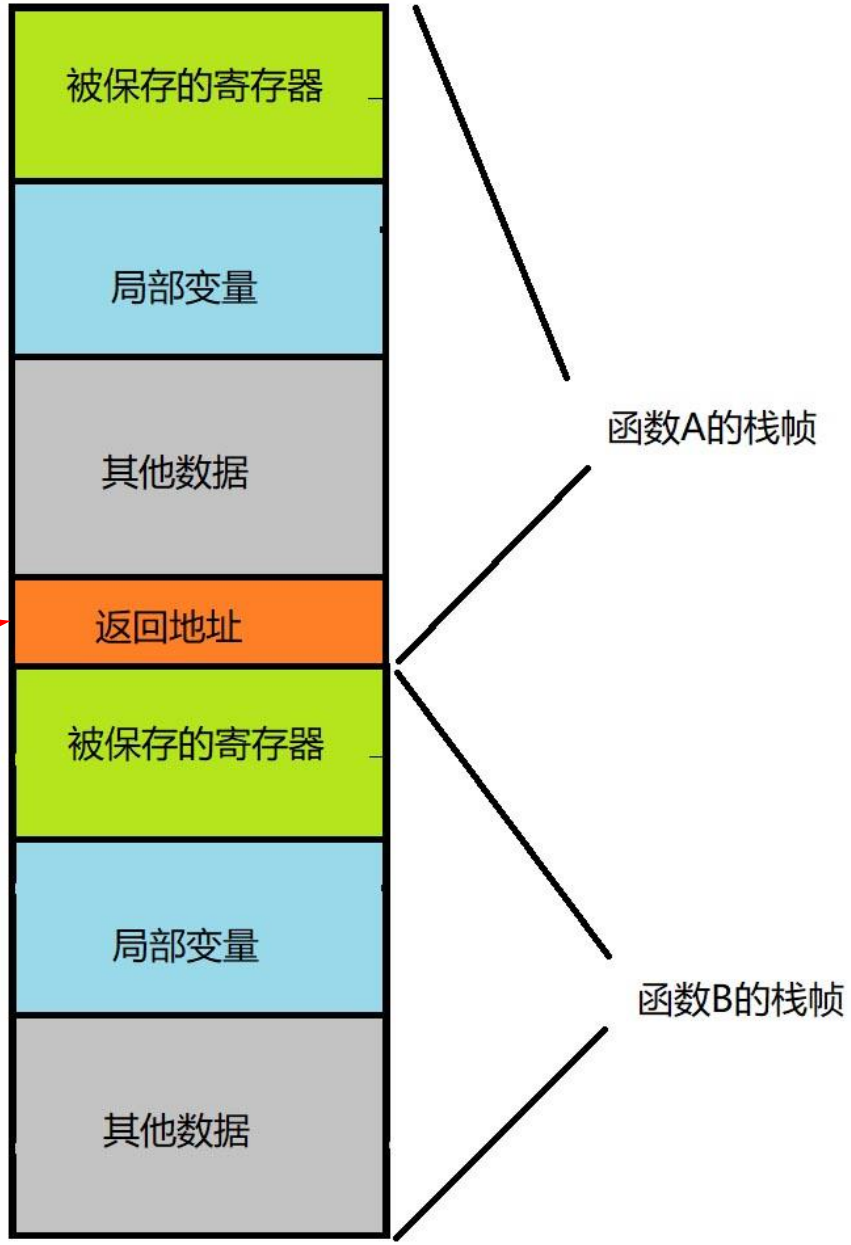
当运行中的程序调用另一个函数时，就要进入一个新的栈帧。原来函数的栈帧称为调用者的帧，新的栈帧称为当前帧。

被调用的函数运行结束后当前帧全部回收，回到调用者的帧。



函数调用时的栈帧变化

当函数A调用函数B:



当函数A调用函数B的时候，会把返回地址压入栈中，我们把返回地址当做A函数栈帧的一部分，因为它存放的是与A相关的状态.
这种设计可以让函数在稍后返回到程序中正确的位置。

寄存器（X64系统，可存64位）

64位	低32位	低16位	低8位	描述
rax	eax	ax	al	累加器
rbx	ebx	bx	bl	基地址
rcx	ecx	cx	cl	循环记数器
rdx	edx	dx	dl	数据寄存器，通常扩展A寄存器
rsi	esi	si	sil	字符串操作的源索引
rdi	edi	di	dil	字符串操作的目的索引
rbp	ebp	bp	bpl	基地址指针(栈帧基地址)
rsp	esp	sp	spl	栈指针

机器指令（与栈帧有关的汇编代码）

算术操作：

指令	效果	描述
ADD S，D	D = D + S	加法
SUB S，D	D = D - S	减法
lea S，D	D <-- &S	加载有效地址
MOV S，D	D <-- S	传送（复制）

弹出和压入栈数据：

指令	效果
push	压栈
pop	出栈

控制转移：

指令	效果
call	调用函数
ret	从函数中返回

push和pop指令

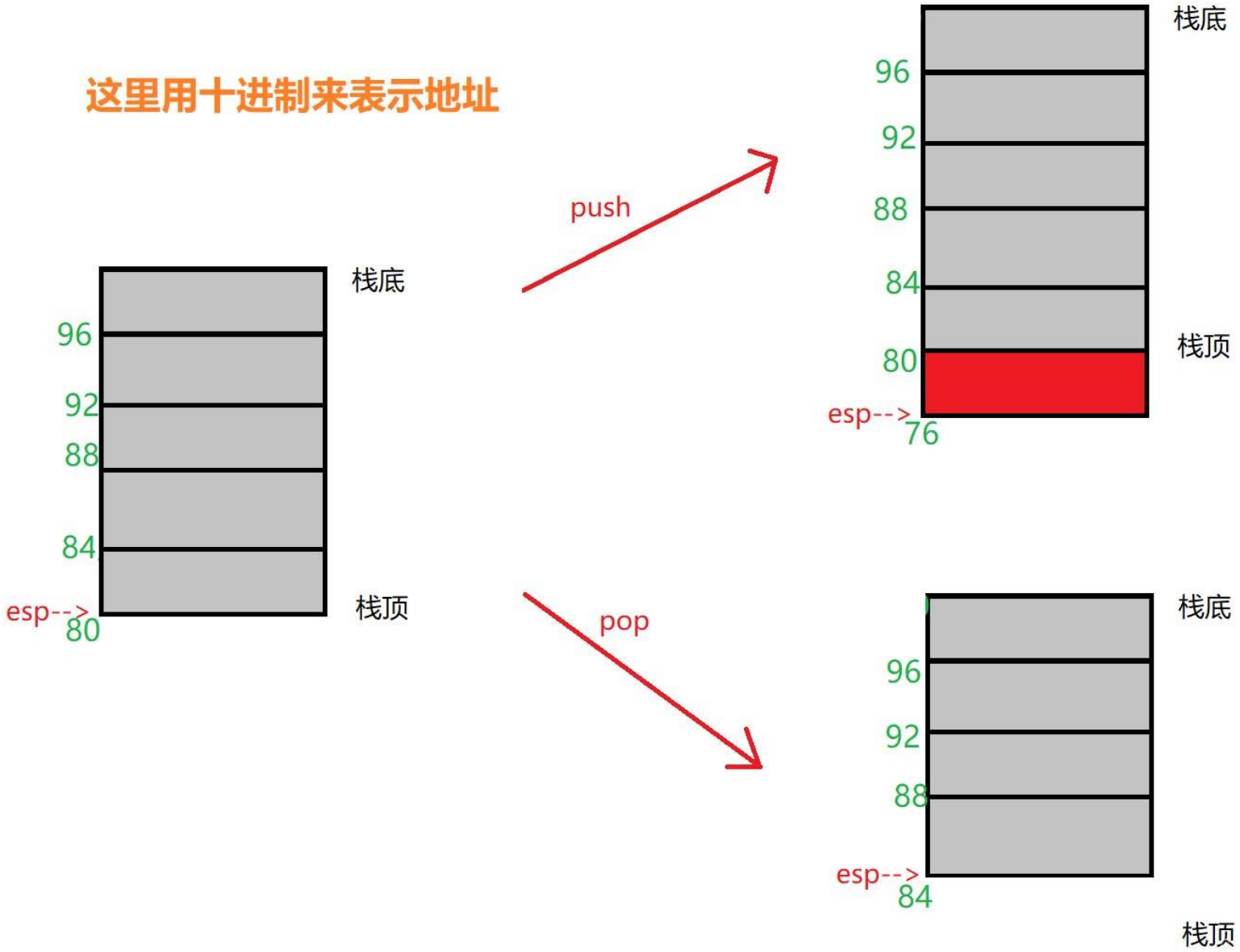
push指令的功能是把数据压入到栈上，同时改变栈指针，每一次push操作都会让栈指针减少。

- 在32位系统上：每一次push会让栈指针减4
- 在64位系统上，每一次push会让栈指针减8

pop指令的功能是从栈中弹出数据，同时改变栈指针，每一次弹出都会让栈指针增加。

- 在32位系统上：每一次pop会让栈指针加4
- 在64位系统上：每一次pop会让栈指针加8

这里用十进制来表示地址

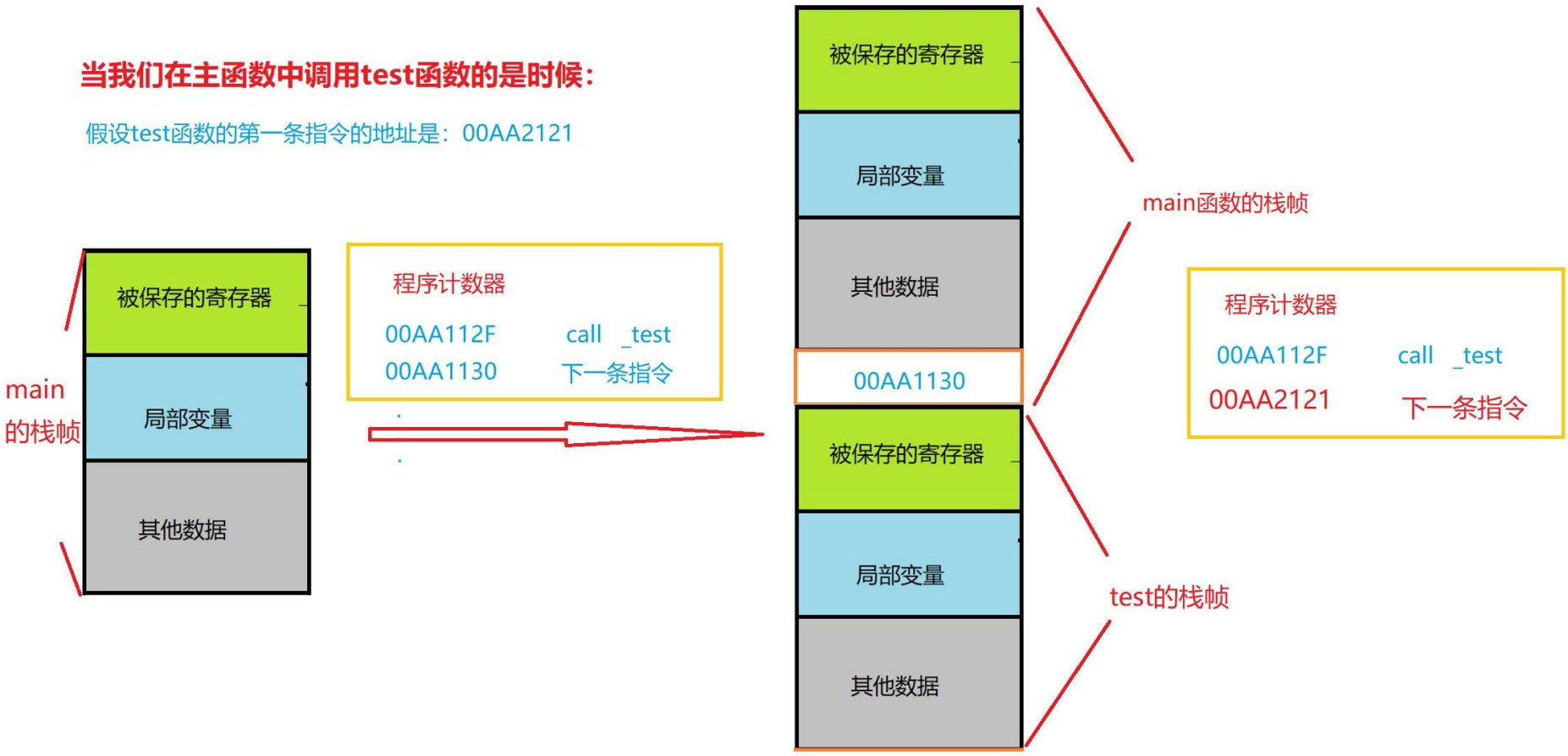


程序计数器PC

作用：给出将要执行的下一条指令在内存中的地址

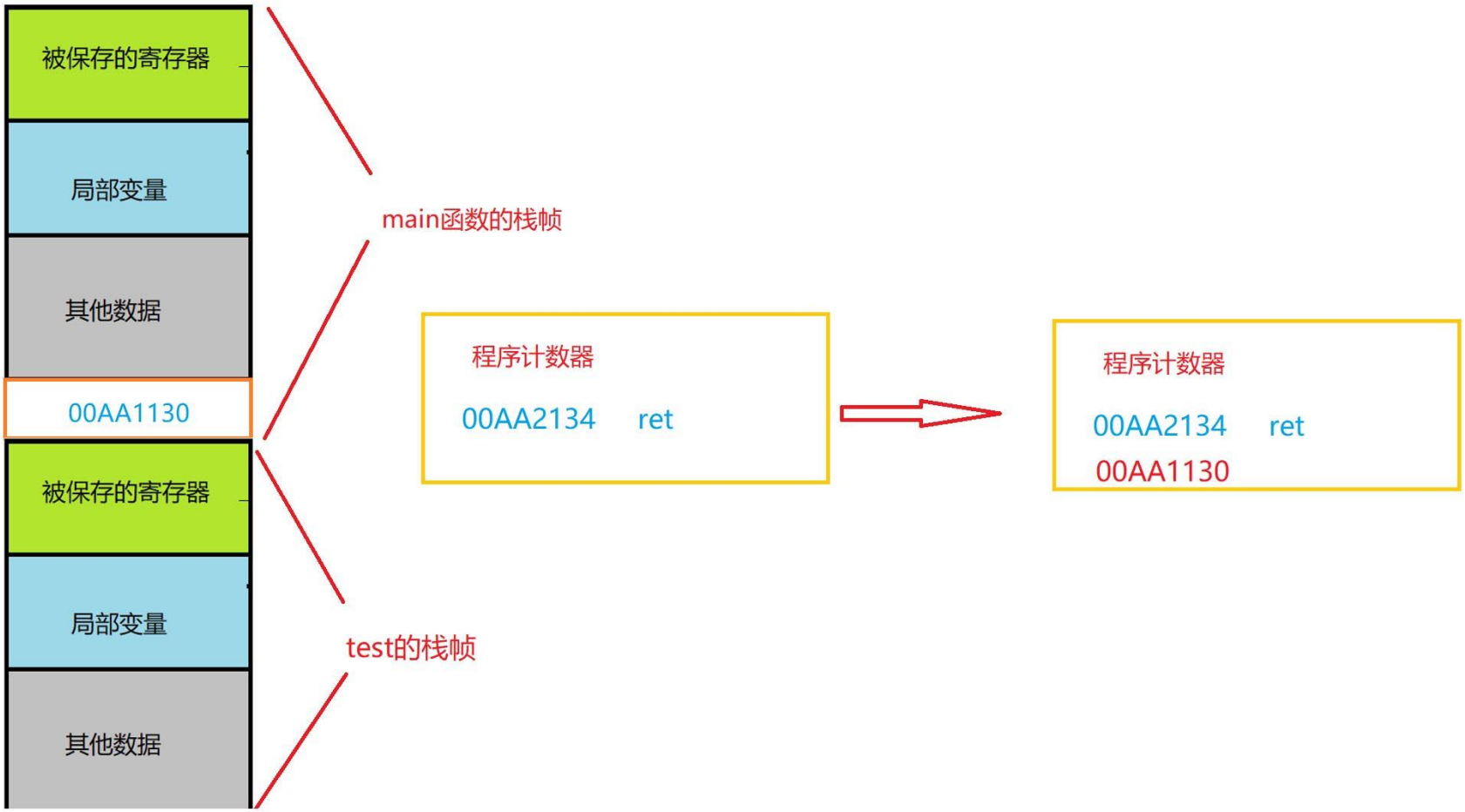
指令call—调用函数

- 第一步：将要执行的下一条指令的地址压入栈中。
- 第二步：将程序计数器设置为被调函数的起始位置



指令ret—从函数中返回

- 从栈中弹出地址，并且将PC的值设置为该地址



数据传送

- 参数的传递

当一个过程调用另一个过程的时候，第一个过程中的代码必须先将参数赋值到适当的寄存器中。

- 返回值的传递

函数的返回值通常是由寄存器**eax**来保存的！

当后一个过程返回到前一个过程的时候，会将要返回的内容保存在寄存器eax中，该函数调用结束后，前一个过程中的代码可以通过访问寄存器eax中的值来得到返回值。

例：函数栈帧创建和销毁的全过程

```c
#include<stdio.h>

int ADD(int x,int y)
{
    int z =  x + y;
    return z;
}

int main()
{
    int a = 10;
    int b = 20;
    int sum = 0;
    sum = ADD(a,b);
    return 0;
}
```

gcc –S –masm=intel test.c –o test.s

汇编

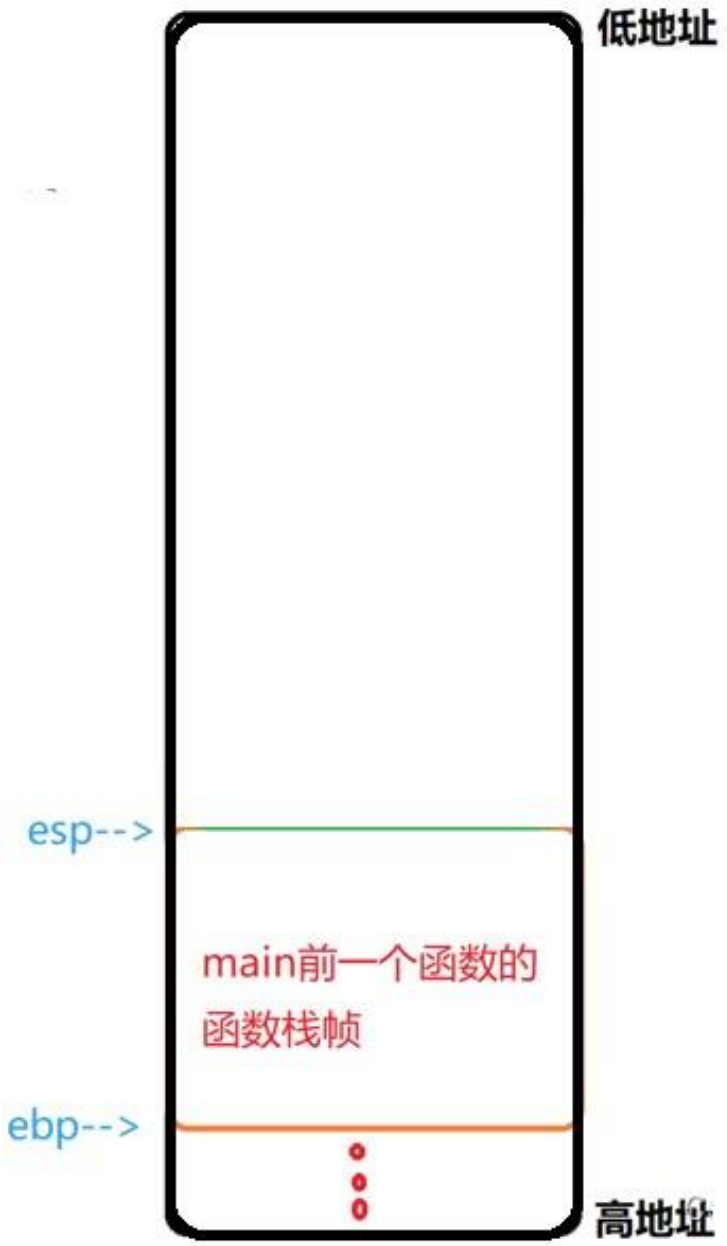
main函数的汇编代码

```
int main()
{
00091E10  push        ebp
00091E11  mov         ebp,esp
00091E13  sub         esp,0E4h
00091E19  push        ebx
00091E1A  push        esi
00091E1B  push        edi
00091E1C  lea         edi,[ebp-24h]
00091E1F  mov         ecx,9
00091E24  mov         eax,0CCCCCCCCh
00091E29  rep stos    dword ptr es:[edi]
00091E2B  mov         ecx,9C003h
00091E30  call        0009130C
    int a = 10;
00091E35  mov         dword ptr [ebp-8],0Ah
    int b = 20;
00091E3C  mov         dword ptr [ebp-14h],14h
    int sum = 0;
00091E43  mov         dword ptr [ebp-20h],0
    sum = ADD(a, b);
00091E4A  mov         eax,dword ptr [ebp-14h]
00091E4D  push        eax
00091E4E  mov         ecx,dword ptr [ebp-8]
00091E51  push        ecx
00091E52  call        000913A2
00091E57  add         esp,8
00091E5A  mov         dword ptr [ebp-20h],eax
    return 0;
00091E5D  xor         eax,eax
}
00091E5F  pop         edi
00091E60  pop         esi
00091E61  pop         ebx
00091E62  add         esp,0E4h
00091E68  cmp         ebp,esp
00091E6A  call        00091230
00091E6F  mov         esp,ebp
00091E71  pop         ebp
00091E72  ret
```

ADD函数的汇编代码

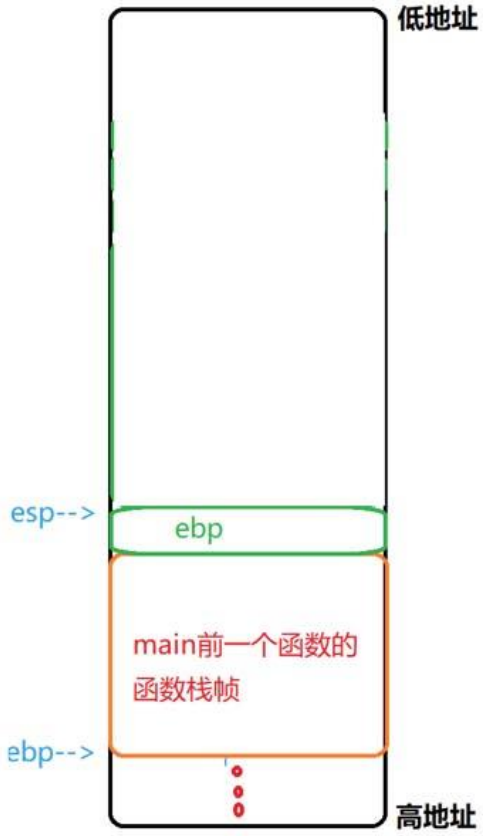
```
int ADD(int x, int y)
{
00091740  push        ebp     已用时间 <= 1ms
00091741  mov         ebp,esp
00091743  sub         esp,0CCh
00091749  push        ebx
0009174A  push        esi
0009174B  push        edi
0009174C  lea         edi,[ebp-0Ch]
0009174F  mov         ecx,3
00091754  mov         eax,0CCCCCCCCh
00091759  rep stos    dword ptr es:[edi]
0009175B  mov         ecx,9C003h
00091760  call        0009130C
    int z = x + y;
00091765  mov         eax,dword ptr [ebp+8]
00091768  add         eax,dword ptr [ebp+0Ch]
0009176B  mov         dword ptr [ebp-8],eax
    return z;
0009176E  mov         eax,dword ptr [ebp-8]
}
00091771  pop         edi
00091772  pop         esi
00091773  pop         ebx
00091774  add         esp,0CCh
0009177A  cmp         ebp,esp
0009177C  call        00091230
00091781  mov         esp,ebp
00091783  pop         ebp
00091784  ret
```

还没为main函数创建栈帧的时候

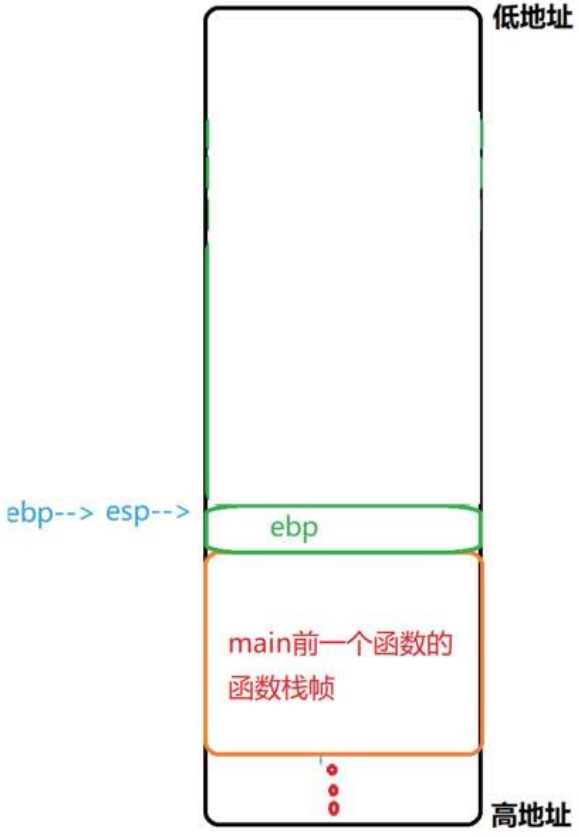


```
00E050F0  push          ebp
00E050F1  mov           ebp, esp
00E050F3  sub           esp, 0E4h
```

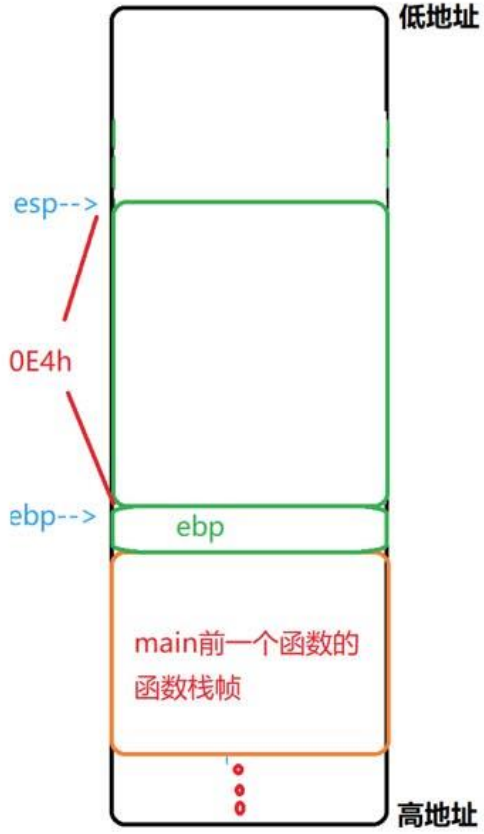
第一步：将ebp的值压入栈中：push ebp



第二步：将esp的值传给ebp：

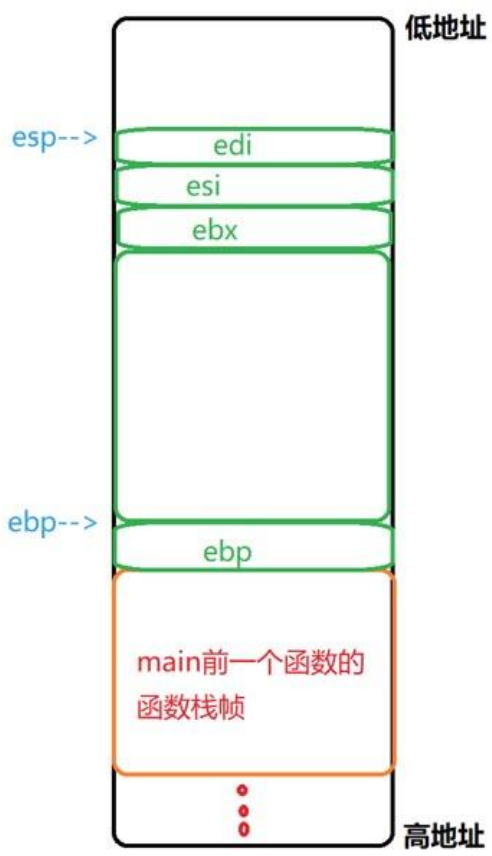


第三步：让esp减0E4h(esp指向esp减0E4h的位置)：



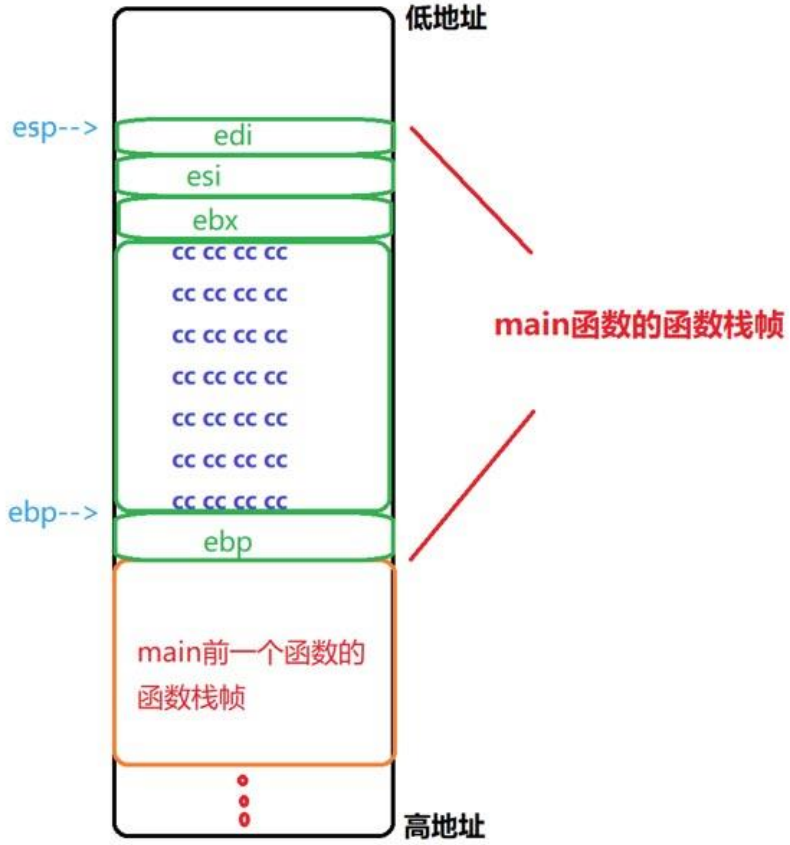
```
00E050F9  push        ebx
00E050FA  push        esi
00E050FB  push        edi
```

将三个寄存器压入栈中：




```
00E050FC  lea         edi,[ebp-24h]
00E050FF  mov         ecx,9
00E05104  mov         eax,0CCCCCCCCh
00E05109  rep stos    dword ptr es:[edi]
00E0510B  mov         ecx,0E0C003h
```

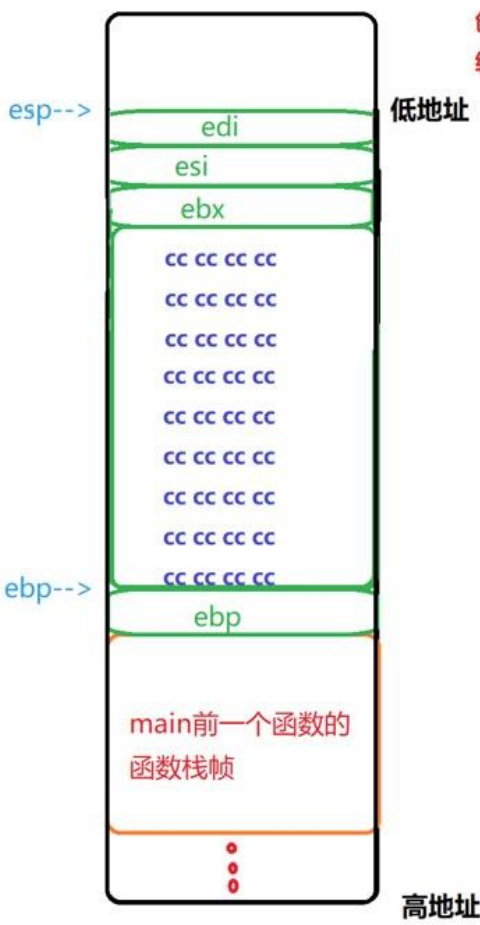
将新栈帧中的值赋为0cccccccch

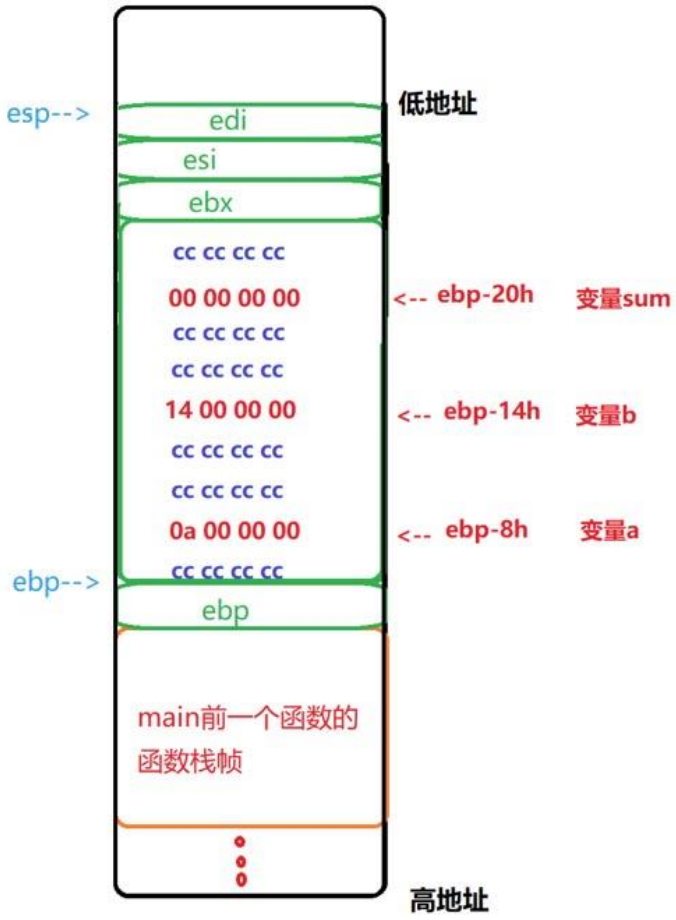


```
1  int a = 10;
2  int b = 20;
3  int sum = 0;
4  sum = ADD(a,b);
```

```
    int a = 10;
00E05115  mov         dword ptr [ebp-8],0Ah
    int b = 20;
00E0511C  mov         dword ptr [ebp-14h],14h
    int sum = 0;
00E05123  mov         dword ptr [ebp-20h],0
```

创建变量并且赋值：每一个
编译器的实现细节不同

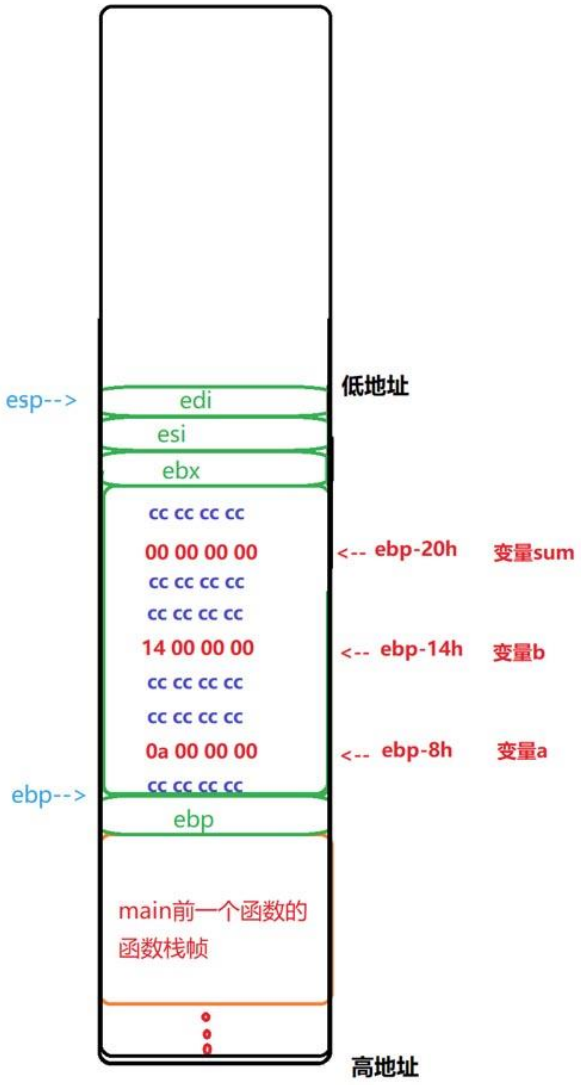




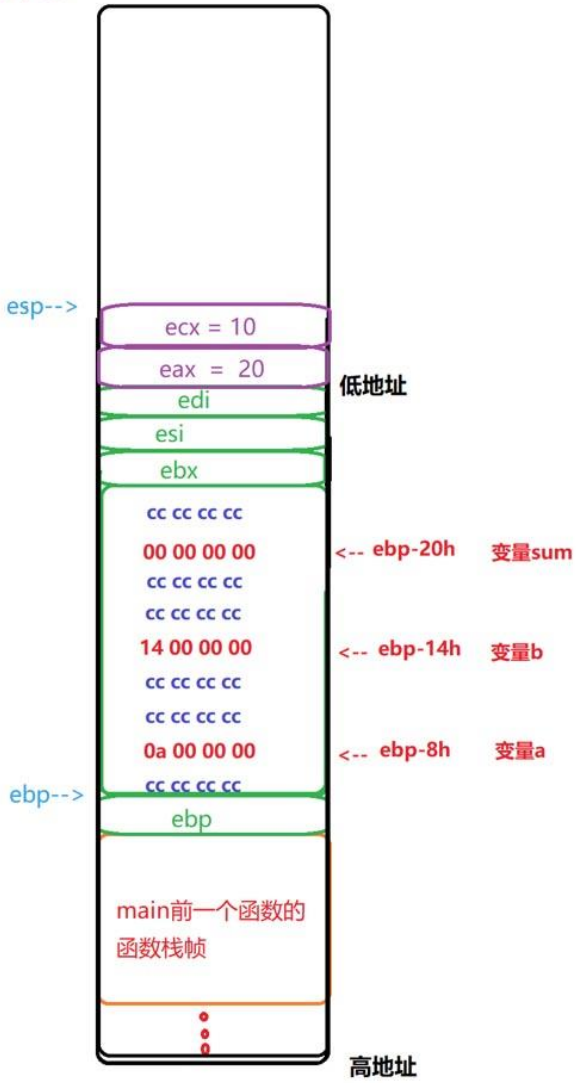
传递参数

执行到sum = ADD(a,b)的时候，要先传递参数和将返回地址压入栈中。

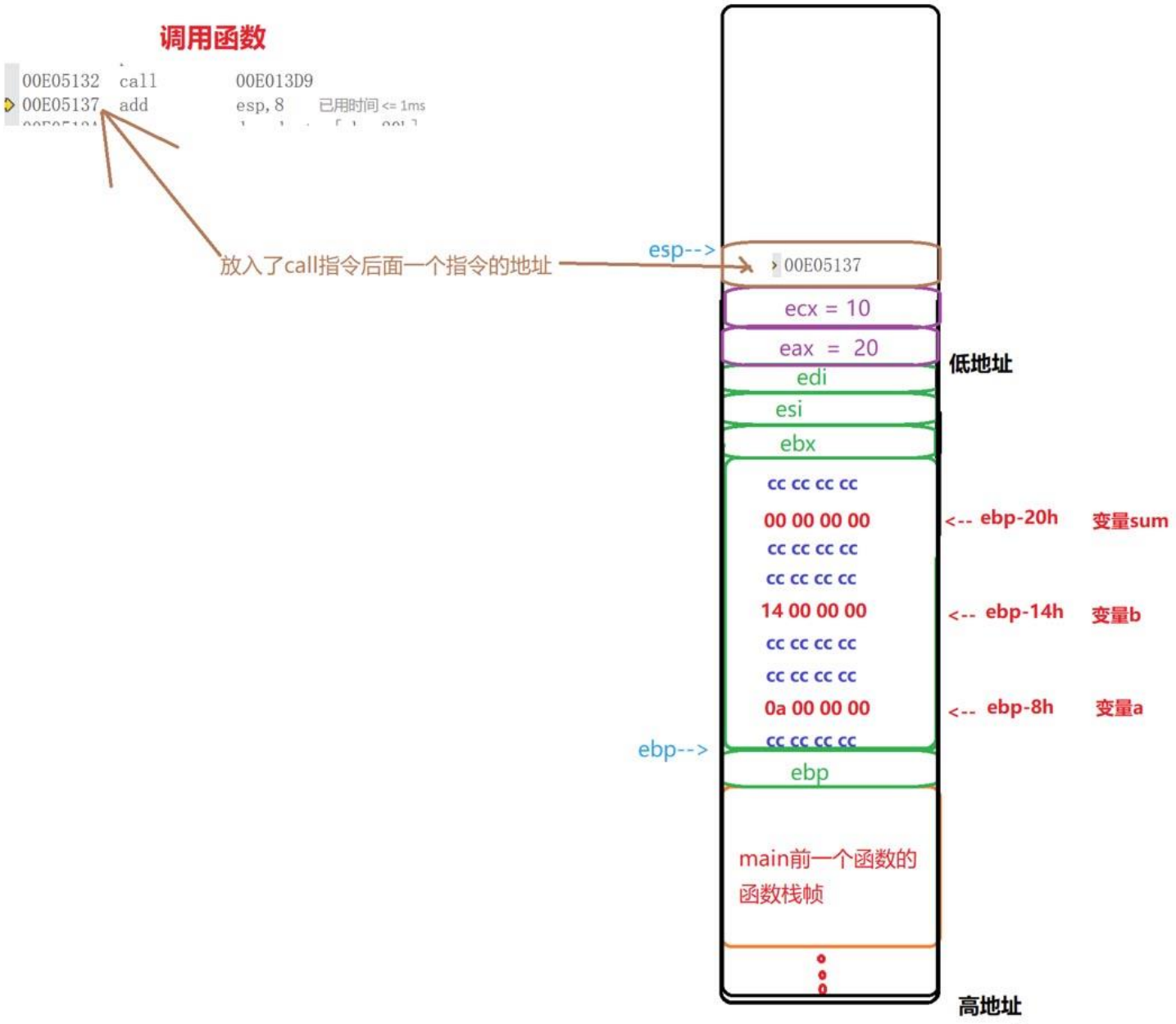
```
    sum = ADD(a, b);
00E0512A  mov         eax,dword ptr [ebp-14h]
00E0512D  push        eax
00E0512E  mov         ecx,dword ptr [ebp-8]
00E05131  push        ecx
```



1. 将ebp-14h地址的数据传递到寄存器eax中
2. 将eax压入栈中
3. 将ebp-8h地址的数据传递到寄存器ecx中
4. 将ecx压入栈中



压入返回值

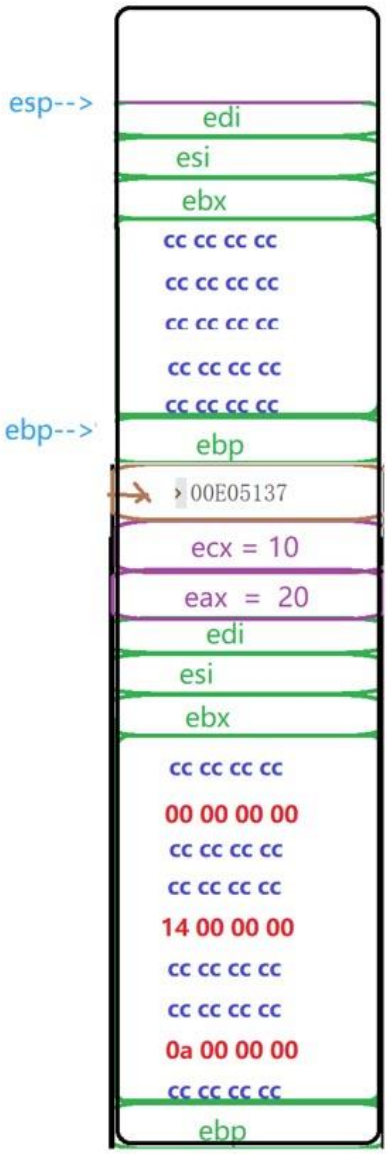

调用函数
00E05132 call 00E013D9
00E05137 add esp, 8 已用时间 <= 1ms
放入了call指令后面一个指令的地址
esp-->
00E05137
ecx = 10
eax = 20
edi
esi
ebx
cc cc cc cc
00 00 00 00
<-- ebp-20h
变量sum
cc cc cc cc
cc cc cc cc
14 00 00 00
<-- ebp-14h
变量b
cc cc cc cc
cc cc cc cc
0a 00 00 00
<-- ebp-8h
变量a
cc cc cc cc
ebp-->
ebp
main前一个函数的
函数栈帧
低地址
高地址

进入ADD函数

```
int ADD(int x, int y)
{
00E01760  push        ebp     已用时间 <= 1ms
00E01761  mov         ebp,esp
00E01763  sub         esp,0CCh
00E01769  push        ebx
00E0176A  push        esi
00E0176B  push        edi
00E0176C  lea         edi,[ebp-0Ch]
00E0176F  mov         ecx,3
00E01774  mov         eax,0CCCCCCCCh
00E01779  rep stos    dword ptr es:[edi]
00E0177B  mov         ecx,0E0C003h
00E01780  call        00E0130C
    int z = x + y;
00E01785  mov         eax,dword ptr [ebp+8]
00E01788  add         eax,dword ptr [ebp+0Ch]
00E0178B  mov         dword ptr [ebp-8],eax
    return z;
00E0178E  mov         eax,dword ptr [ebp-8]
}
```

和刚进入main函数一样，这里是在为该函数创建函数栈帧

```
    int z = x + y;
00E01785  mov         eax,dword ptr [ebp+8]
00E01788  add         eax,dword ptr [ebp+0Ch]
00E0178B  mov         dword ptr [ebp-8],eax
```



执行ADD函数中的操作

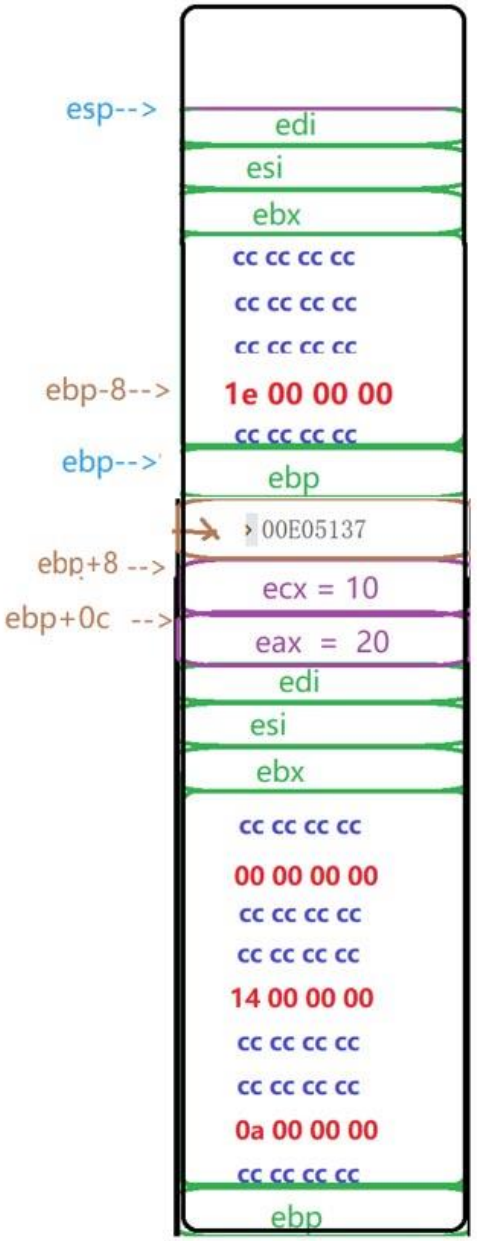
- 用寄存器eax存储计算后的值

```
    int z = x + y;
00E01785  mov         eax,dword ptr [ebp+8]
00E01788  add         eax,dword ptr [ebp+0Ch]
00E0178B  mov         dword ptr [ebp-8],eax
```

```
    return z;
00E0178E  mov         eax,dword ptr [ebp-8]
}
```

再把ebp-8的值保存再寄存器eax中

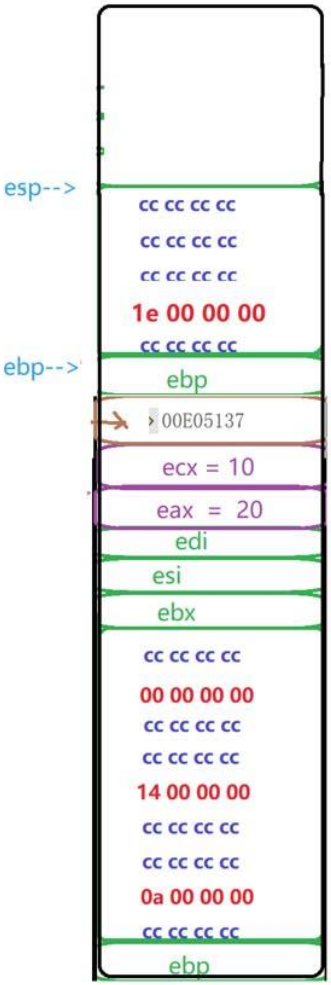
ADD函数调用结束，回到main函数中

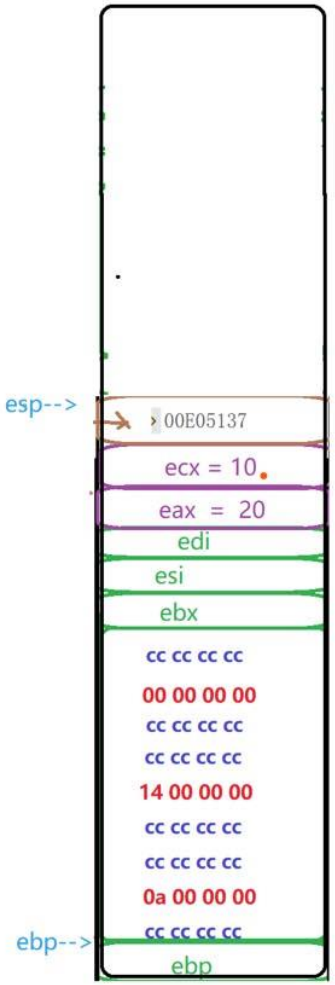


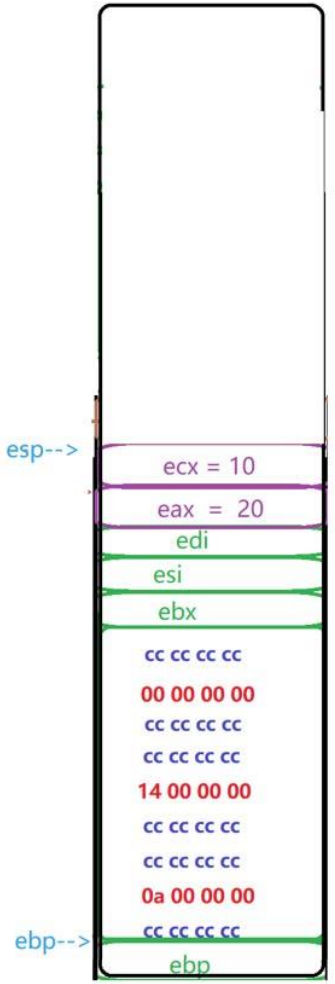
ADD结束后销毁该函数的栈帧

```
00E01791  pop         edi
00E01792  pop         esi
00E01793  pop         ebx
00E01794  add         esp,0CCh
00E0179A  cmp         ebp,esp
00E0179C  call        00E01235
00E017A1  mov         esp,ebp
00E017A3  pop         ebp
00E017A4  ret
```

会依次弹出edi、esi、ebx
并且会把ebp的值赋给esp
然后弹出ebp





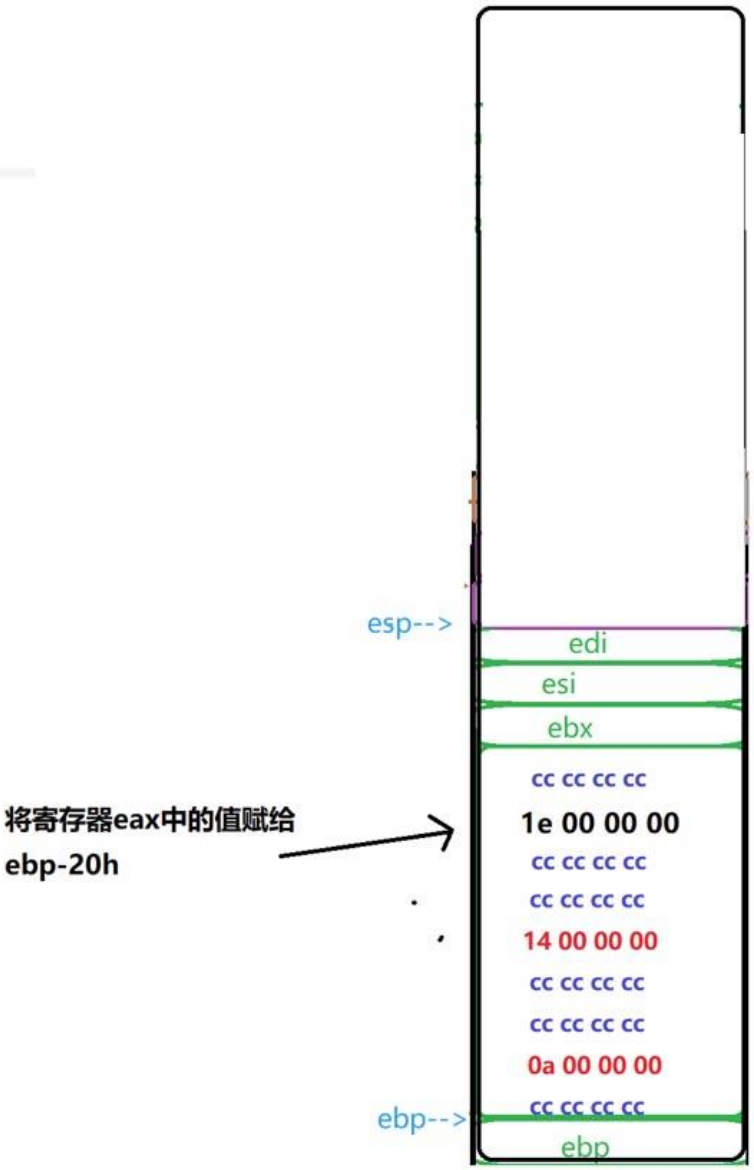


之后……

```
00E05137  add         esp,8     已用时间<=1ms
00E0513A  mov         dword ptr [ebp-20h],eax
    return 0;
```

形参销毁

main函数得到返回值，
执行计算和存储，然后
main函数调用完毕，
销毁main函数的栈帧。



再用同样的顺序将main函数
的栈帧销毁

总结

1. 局部变量是怎么创建的?

局部变量的创建首先是为所在函数分配好栈帧空间，栈帧空间里面会初始化一部分空间，然后给局部变量在栈帧中分配一点空间。

2. 为什么局部变量不初始化的时候值是随机的?

在创建所在函数的栈帧的时候，那一部分初始化的空间里面存放的是随机值。如果在创建局部变量的时候没有初始化，那么该空间的随机值不会改变；如果创建的时候初始化了局部变量，那么初始化的值会将随机值覆盖。

总结

3. 函数是怎么传参的？传参的顺序是怎么样的？

当还没有调用函数的时候，就已经将实参从右向左开始压栈。在真正进入形参函数的时候，用函数里面的指针偏移量反回来找到形参。

4. 形参和实参的关系？

形参是实参的一份临时拷贝，改变形参不会影响实参。

总结

5. 函数调用的结果是怎么返回的?

在函数调用前，就已将call指令的下一条指令的地址记住，caller函数的ebp已经存进去了。当函数使用完要返回的时候，弹出ebp就可以找到上一个函数的ebp，然后指针向下就可以找到esp的地址，这样就已经回到上一个函数的栈帧空间。又因为调用之前就记住了call下一条指令的地址，从函数调用返回的时候，返回值通过保存在寄存器里面被带回来。