# 计算机系统
# 程序的机器级表示：过程

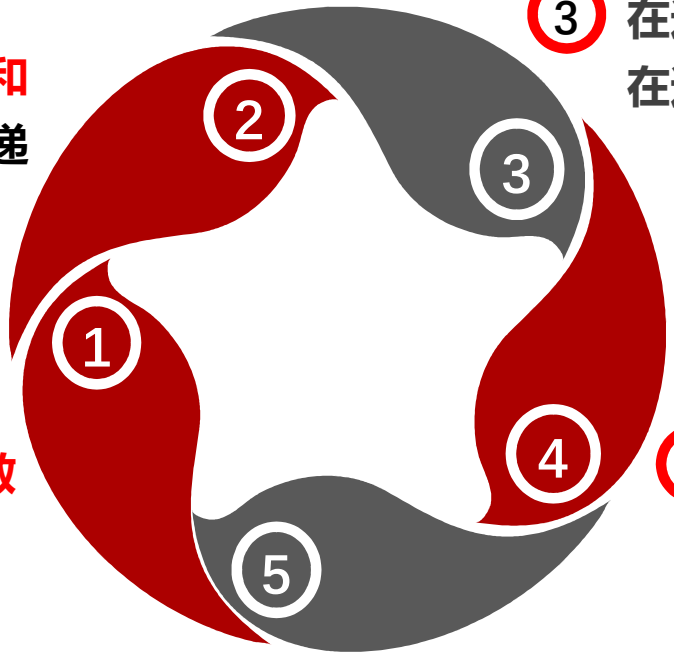
湖南大学

《计算机系统》课程教学组

# 什么是过程?

2 过程的调用包括**将数据和控制**从代码的一部分传递到另一部分

3 在进入时为过程的局部变量**分配空间**，在退出时**释放空间**




1 在高级语言中也称为**函数**

4 大多数机器，只提供**转移控制到过程**和**从过程中转移出控制**这种简单指令

5 数据传递和局部变量的分配释放都是通过操纵**程序栈**来实现。

# 过程的作用

能够使程序变得
更简短而清晰

可以提高程序
开发的效率

有利于程序维护

提高了代码的复用性

# 为什么用栈?

机器用栈来**传递过程参数、存储返回信息、保存寄存器**用于以后恢复，以及**本地存储**。而为单个过程分配的那部分栈称为栈帧（stack frame）。

## 参数传递

- 寄存器数量有限
- 避免在递归调用时可能发生的内容冲突

## 局部变量

- 避免存储区域冲突
- 提高访问效率

## 寄存器保存

- 防止有用的寄存器内容被覆盖
- 提供方便的寄存器内容保存与取回方式

# 为什么用栈?

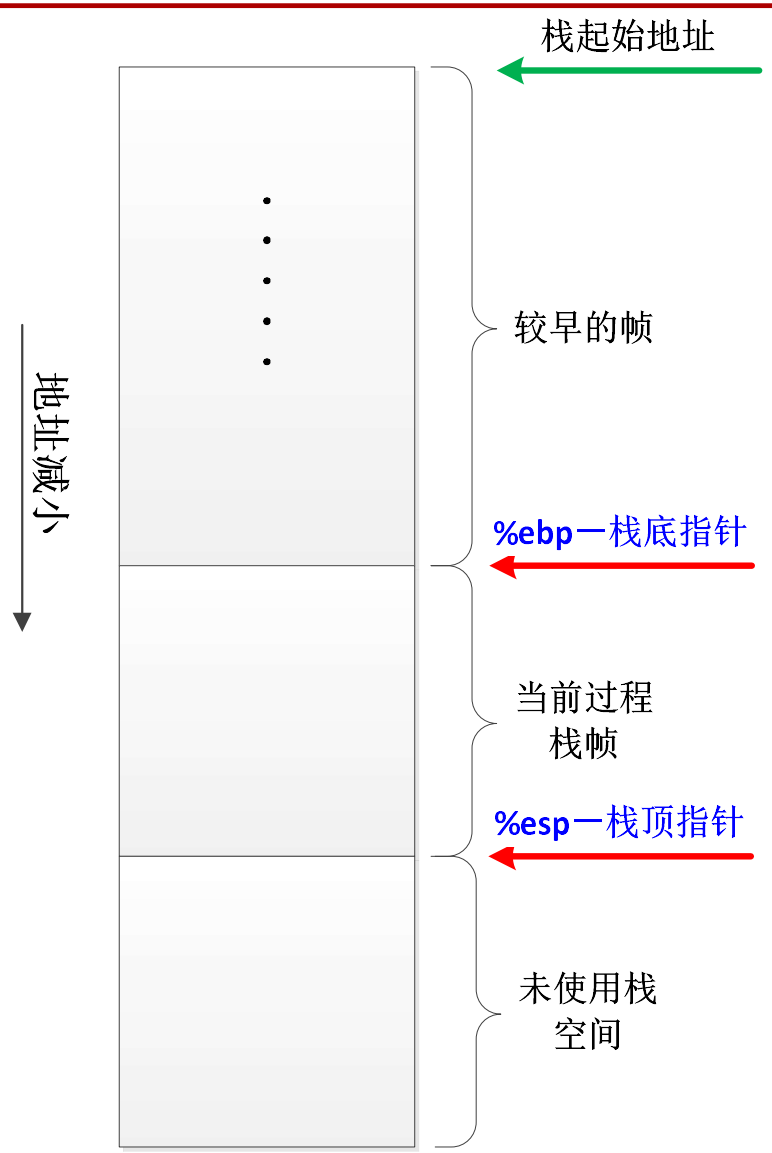
- 栈为**程序的调用**提供了一个数据交换和暂存的**场所**，就像超市为各地商品的交换提供了场所一样。
- 由于栈中数据操作的**规律性**（先进后出，如同货架），使得数据的传送更利于**机器操作**的实现。


**货物中转 先进后出**

# 栈帧

每一个函数或过程在执行时，都需要在内存中分配一个空间来保存运行时数据，这个空间由于是采用栈的方式进行操作，所以也称为**栈帧**。

- 当前函数或过程的栈顶地址保存在%esp中，栈底地址保存在%ebp中；
- 栈是向“下”增长的，或者说是向地址0x0处增加的，因此%esp中的值小于或等于%ebp中的值；
- 栈帧是内存中一段连续的内存空间；
- 被调用者的栈帧紧挨着调用者的栈帧；



# 课堂习题

当前寄存器状况如右图，数据均为int。则执行下列指令后：

```
push  %eax
pop   %edx
```

寄存器内容为：

A. %eax=0x123, %edx=0x108, %esp=0x108

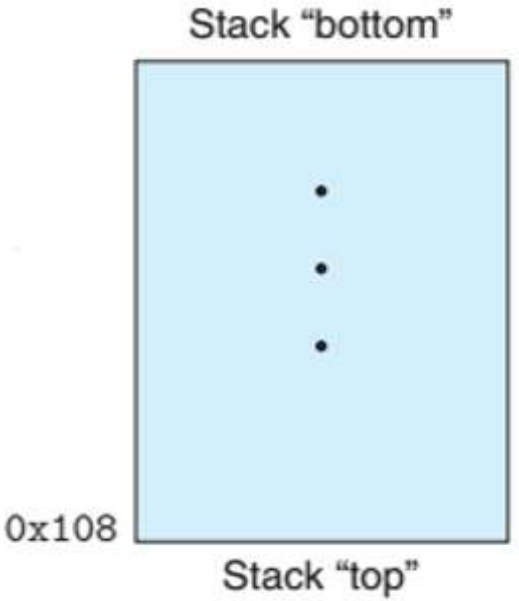
B. %eax=0x123, %edx=0x123, %esp=0x108

C. %eax=0x123, %edx=0x123, %esp=0x104

D. %eax=0x123, %edx=0x108, %esp=0x123

Initially

| | |
|---|---|
| %eax | 0x123 |
| %edx | 0 |
| %esp | 0x108 |

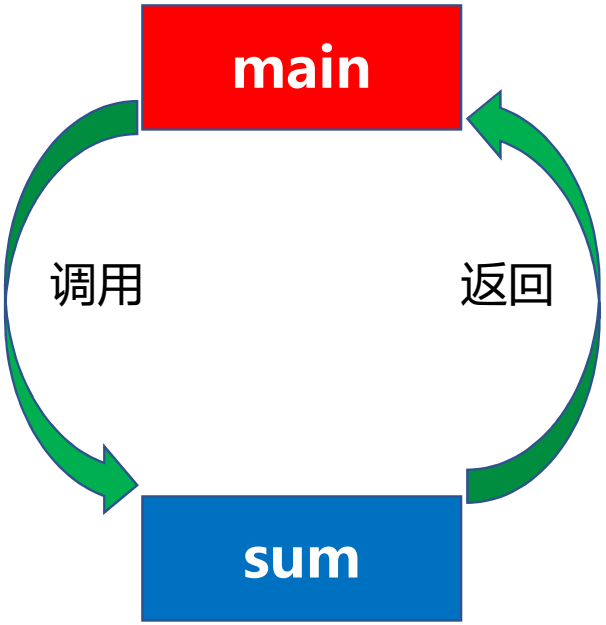


# 一个简单例子

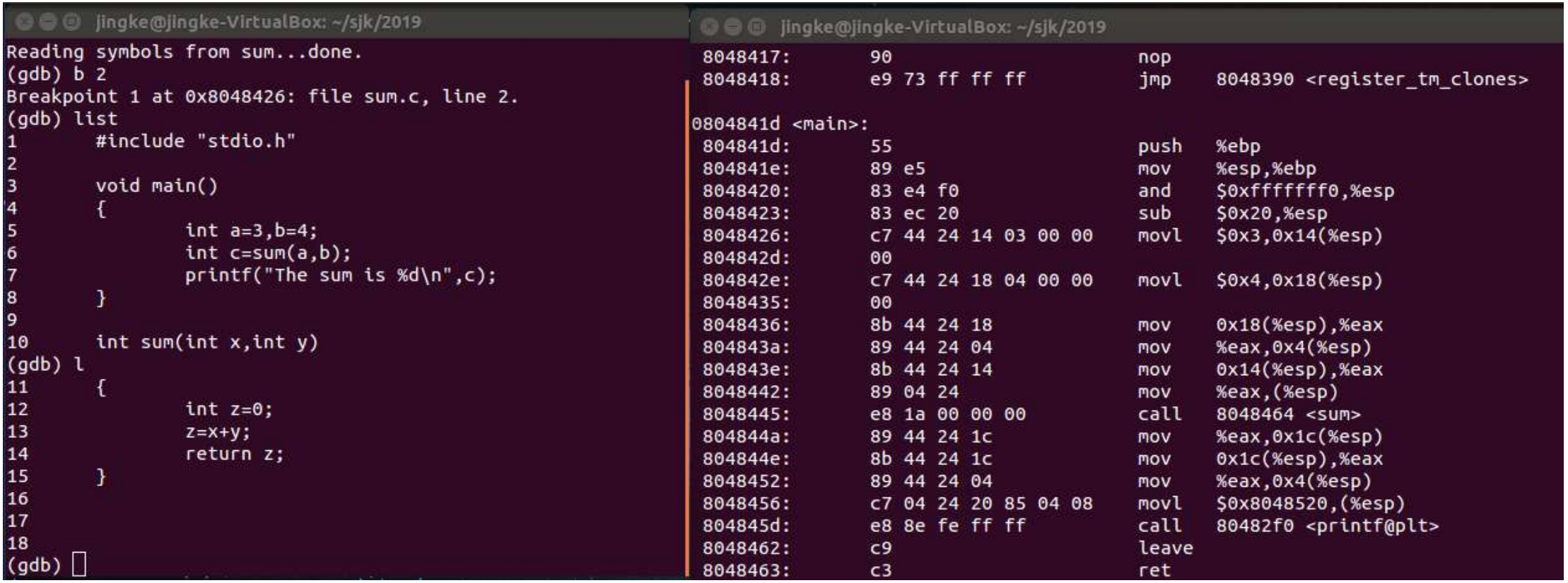
```
#include "stdio.h"

void main()
{
      int a=3,b=4;
      int c=sum(a,b);
      printf("The sum is %d\n",c);
}

int sum(int x,int y)
{
      int z=0;
      z=x+y;
      return z;
}
```



# SUM代码演示



```
jingke@jingke-VirtualBox: ~/sjk/2019
Reading symbols from sum...done.
(gdb) b 2
Breakpoint 1 at 0x8048426: file sum.c, line 2.
(gdb) list
1       #include "stdio.h"
2
3       void main()
4       {
5               int a=3,b=4;
6               int c=sum(a,b);
7               printf("The sum is %d\n",c);
8       }
9
10      int sum(int x,int y)
(gdb) l
11      {
12              int z=0;
13              z=x+y;
14              return z;
15      }
16
17
18
(gdb)
```

```
jingke@jingke-VirtualBox: ~/sjk/2019
 8048417:       90                      nop
 8048418:       e9 73 ff ff ff          jmp    8048390 <register_tm_clones>

0804841d <main>:
 804841d:       55                      push   %ebp
 804841e:       89 e5                   mov    %esp,%ebp
 8048420:       83 e4 f0                and    $0xfffffff0,%esp
 8048423:       83 ec 20                sub    $0x20,%esp
 8048426:       c7 44 24 14 03 00 00    movl   $0x3,0x14(%esp)
 804842d:       00
 804842e:       c7 44 24 18 04 00 00    movl   $0x4,0x18(%esp)
 8048435:       00
 8048436:       8b 44 24 18             mov    0x18(%esp),%eax
 804843a:       89 44 24 04             mov    %eax,0x4(%esp)
 804843e:       8b 44 24 14             mov    0x14(%esp),%eax
 8048442:       89 04 24                mov    %eax,(%esp)
 8048445:       e8 1a 00 00 00          call   8048464 <sum>
 804844a:       89 44 24 1c             mov    %eax,0x1c(%esp)
 804844e:       8b 44 24 1c             mov    0x1c(%esp),%eax
 8048452:       89 44 24 04             mov    %eax,0x4(%esp)
 8048456:       c7 04 24 20 85 04 08    movl   $0x8048520,(%esp)
 804845d:       e8 8e fe ff ff          call   80482f0 <printf@plt>
 8048462:       c9                      leave
 8048463:       c3                      ret
```

# SUM代码演示

```
(gdb) r
Starting program: /home/jingke/sjk/2019/sum

Breakpoint 1, main () at sum.c:5
5               int a=3,b=4;
(gdb)
```

```
esp            0xbffff0d0      0xbffff0d0
ebp            0xbffff0f8      0xbffff0f8
```

当前(main)栈帧

```
0804841d <main>:
 804841d:       55                      push   %ebp
 804841e:       89 e5                   mov    %esp,%ebp
 8048420:       83 e4 f0                and    $0xfffffff0,%esp
 8048423:       83 ec 20                sub    $0x20,%esp
```

main函数栈帧准备

```
 8048426:       c7 44 24 14 03 00 00    movl   $0x3,0x14(%esp)
 804842d:       00
 804842e:       c7 44 24 18 04 00 00    movl   $0x4,0x18(%esp)
 8048435:       00
```

局部变量a, b赋值

```
(gdb) x/4bx 0xbffff0e4
0xbffff0e4:     0x03    0x00    0x00    0x00
(gdb) x/4bx 0xbffff0e8
0xbffff0e8:     0x04    0x00    0x00    0x00
```

a, b赋值后栈中情况

# SUM代码演示

```
8048436:       8b 44 24 18                 mov    0x18(%esp),%eax
804843a:       89 44 24 04                 mov    %eax,0x4(%esp)
804843e:       8b 44 24 14                 mov    0x14(%esp),%eax
8048442:       89 04 24                    mov    %eax,(%esp)
```

```
(gdb) x/4bx 0xbffff0d0
0xbffff0d0:     0x03    0x00    0x00    0x00
(gdb) x/4bx 0xbffff0d4
0xbffff0d4:     0x04    0x00    0x00    0x00
```

准备传递参数：
esp=0xbffff0d0

1. a 拷贝至esp
2. b拷贝至esp+4

```
8048445:       e8 1a 00 00 00              call   8048464 <sum>
804844a:       89 44 24 1c                 mov    %eax,0x1c(%esp)
```

保存返回地址

```
(gdb) x/4bx 0xbffff0cc
0xbffff0cc:     0x4a    0x84    0x04    0x08
```

返回地址：0x804844a

# SUM代码演示

```
(gdb) s
6               int c=sum(a,b);
(gdb)
```

sum函数调用

```
08048464 <sum>:
 8048464:       55                              push   %ebp
 8048465:       89 e5                           mov    %esp,%ebp
 8048467:       83 ec 10                        sub    $0x10,%esp
```

保存旧ebp
&准备sum栈帧

```
(gdb) x/4bx 0xbffff0c8
0xbffff0c8:     0xf8    0xf0    0xff    0xbf
```

旧ebp：0xbffff0f8

```
esp             0xbffff0b8      0xbffff0b8
ebp             0xbffff0c8      0xbffff0c8
```

当前（sum）栈帧

# SUM代码演示

```
804846a:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
```

z=0赋值

```
(gdb) x/4bx 0xbffff0c4
0xbffff0c4:      0x00     0x00     0x00     0x00
```

赋值后z存放位置及数值

```
8048471:       8b 45 0c                mov    0xc(%ebp),%eax
8048474:       8b 55 08                mov    0x8(%ebp),%edx
8048477:       01 d0                   add    %edx,%eax
```

取传递参数并相加

```
eax                0x7        7
ecx                0x6849af32         1749659442
edx                0x3        3
```

%eax(4)+%edx(3)->%eax(7)

```
8048479:       89 45 fc                mov    %eax,-0x4(%ebp)
804847c:       8b 45 fc                mov    -0x4(%ebp),%eax
```

结果传递给z并准备返回结果

```
(gdb) x/4bx 0xbffff0c4
0xbffff0c4:      0x07     0x00     0x00     0x00
```

栈中z的值由0变7

# SUM代码演示

```
804847f:       c9                              leave
8048480:       c3                              ret
```

结束调用，返回

```
esp            0xbffff0d0      0xbffff0d0
ebp            0xbffff0f8      0xbffff0f8
```

恢复main的栈帧

# sum.c 的栈帧

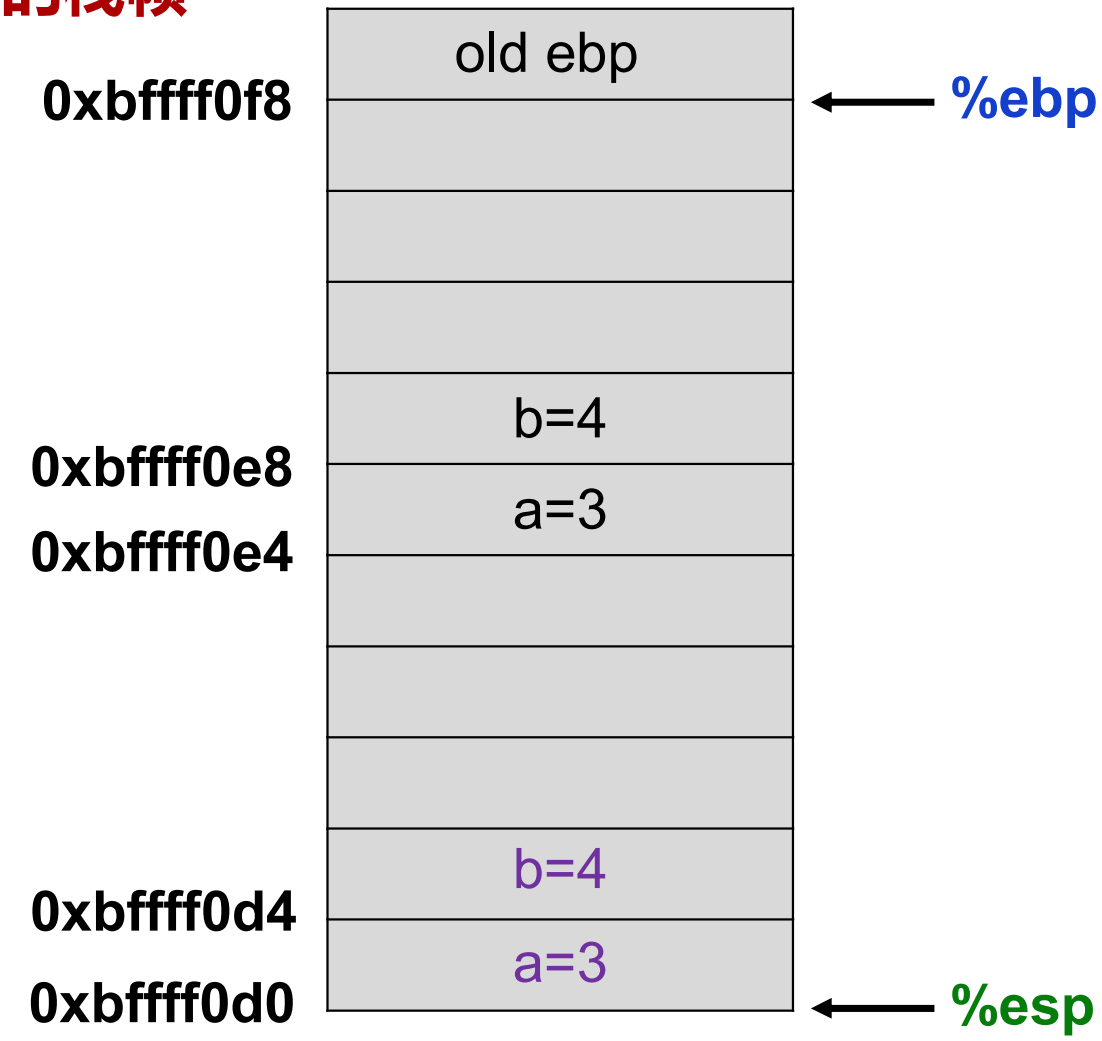
## main函数栈帧准备（参数及传递）

```
push  %ebp
mov   %esp, %ebp
and   $0xfffffff0, %esp
sub   $0x20, %esp
movl  $0x3, 0x14(%esp)
movl  $0x4, 0x18(%esp)
mov    0x18(%esp), %eax
mov   %eax, 0x4(%esp)
mov   0x14(%esp), %eax
mov   %eax, (%esp)
```

栈帧准备

局部变量

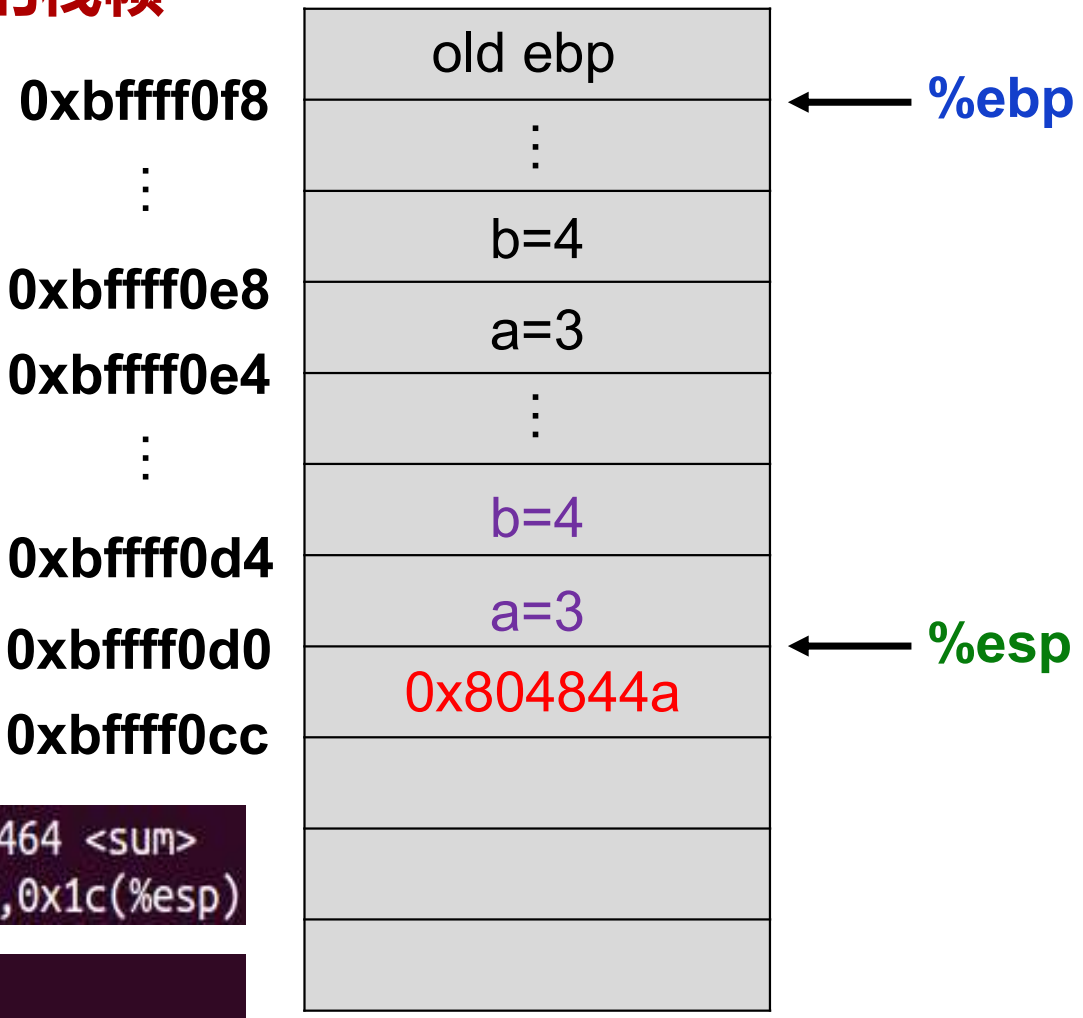
传递参数



# sum.c 的栈帧

**main函数栈帧准备（保存返回地址）**

```
call    8048464 <sum>
```



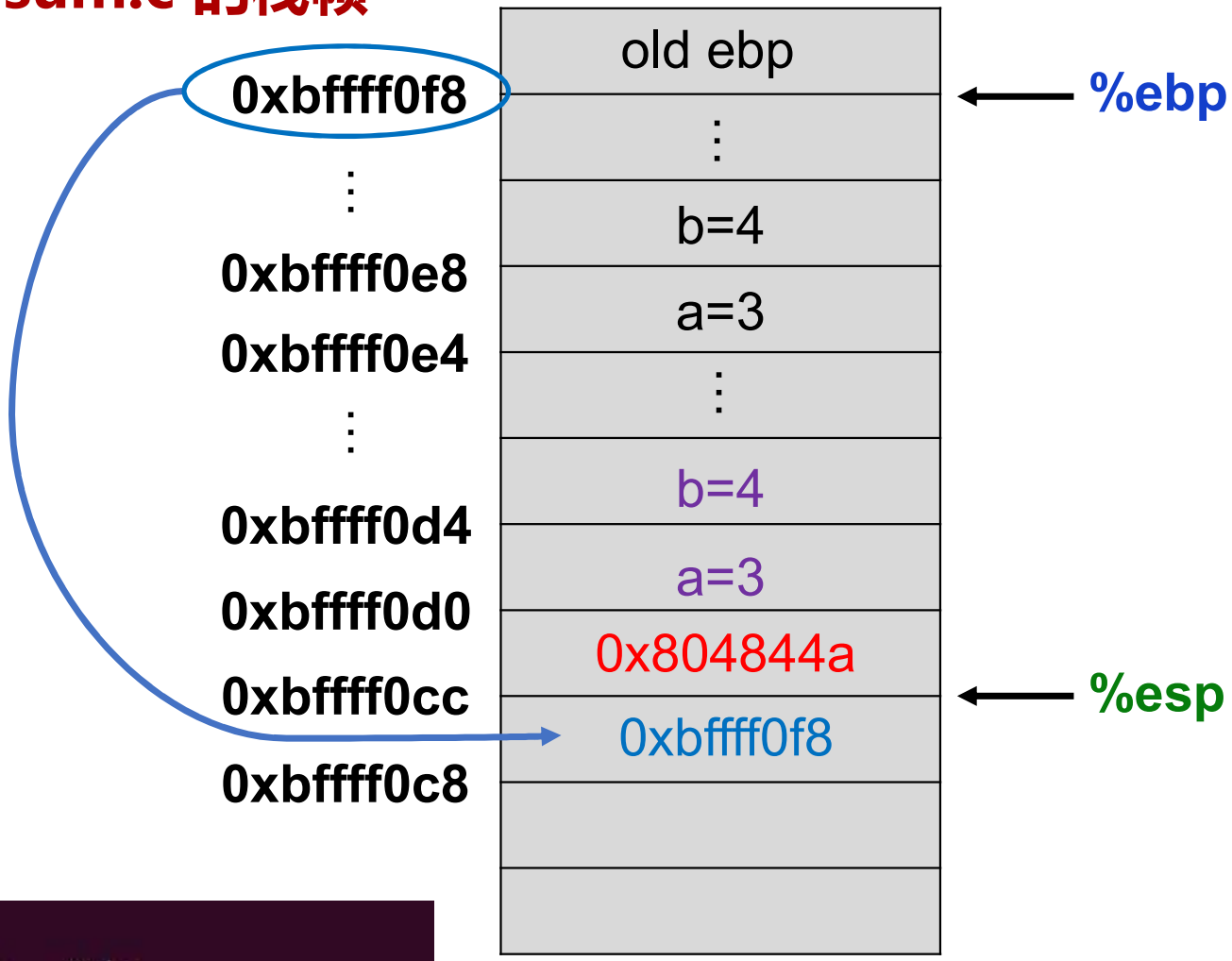

```
8048445:       e8 1a 00 00 00          call   8048464 <sum>
804844a:       89 44 24 1c             mov    %eax,0x1c(%esp)
```

```
08048464 <sum>:
 8048464:      55                      push   %ebp
```

# sum.c 的栈帧

**sum的栈帧（初始化）**

push %ebp

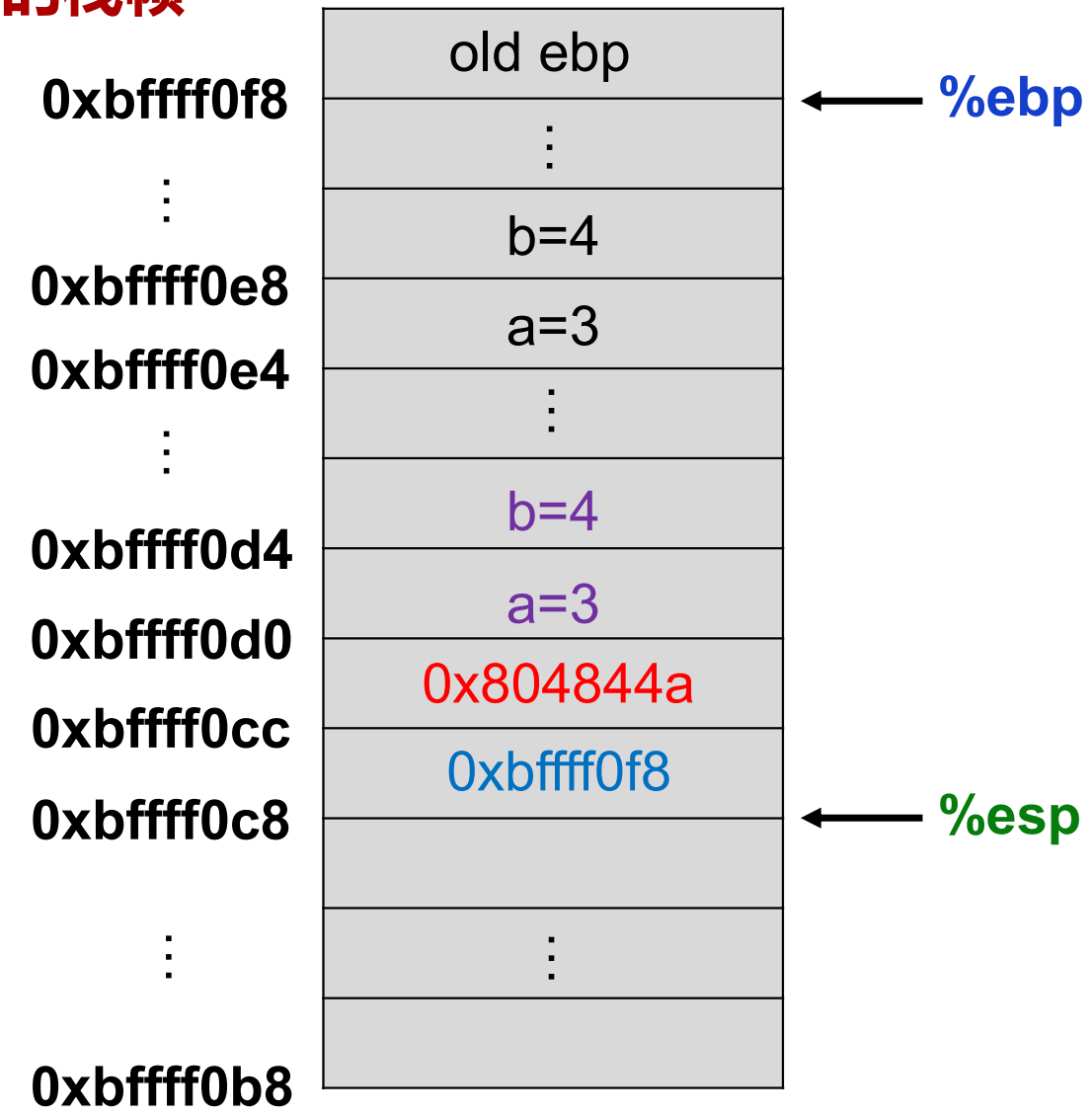



```
08048464 <sum>:
 8048464:       55                      push   %ebp
```

# sum.c 的栈帧

**sum的栈帧（初始化）**

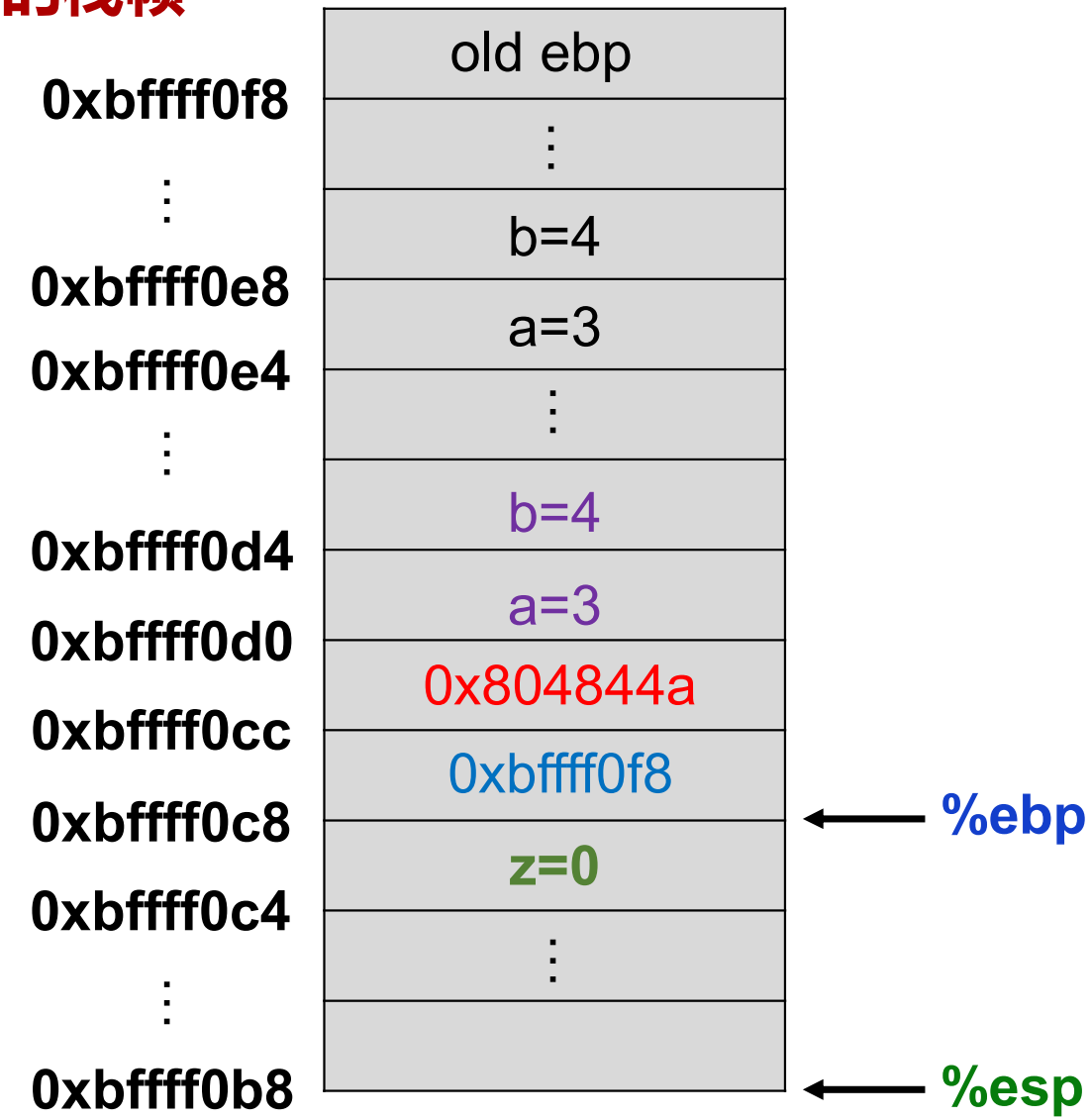
```
push  %ebp
mov   %esp, %ebp
sub   $0x10, %esp
```



# sum.c 的栈帧

**sum的栈帧（z的初值）**

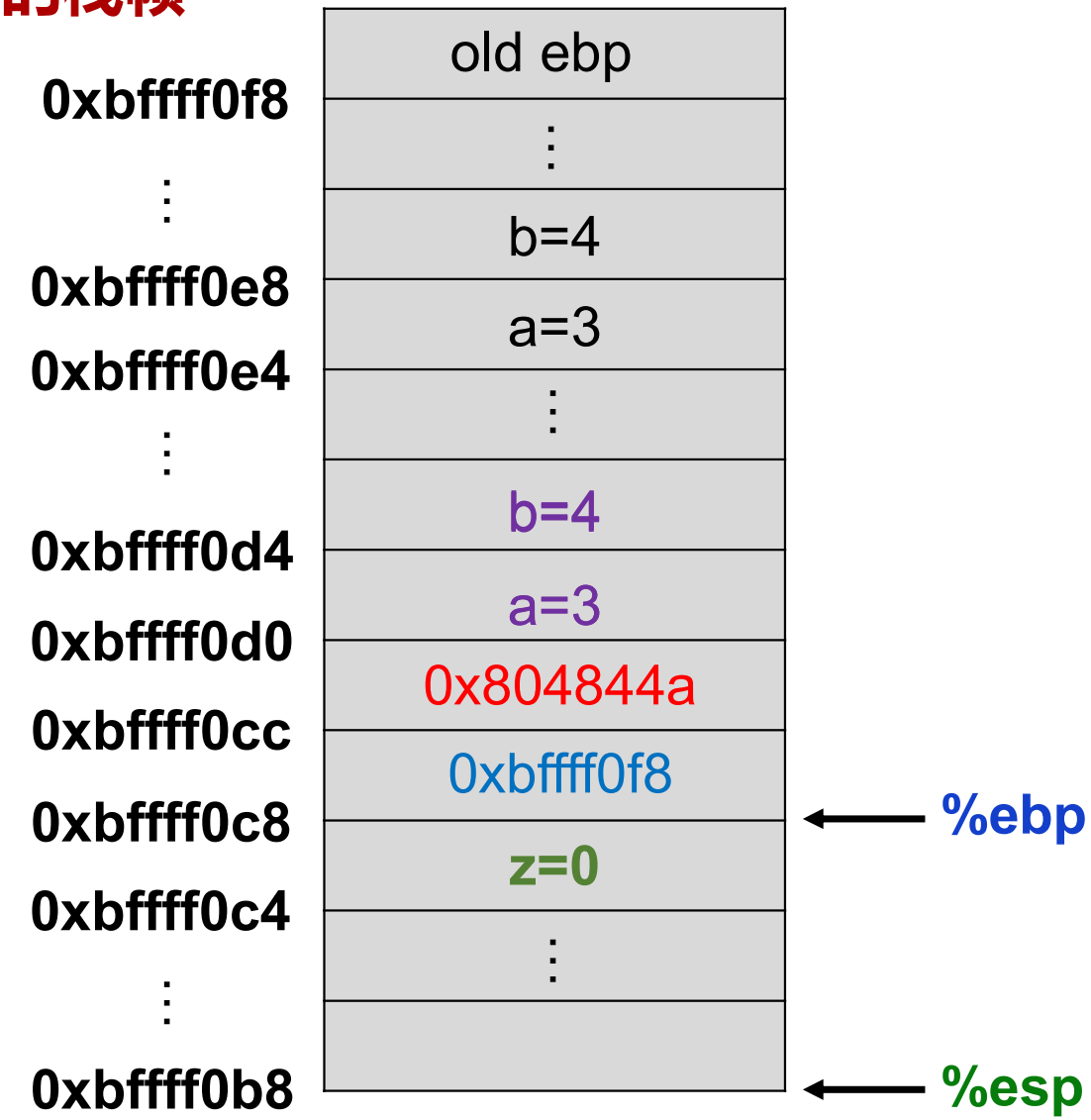
```
push  %ebp
mov   %esp, %ebp
sub   $0x10, %esp
mov   $0x0, -0x4(%ebp)
```



# sum.c 的栈帧

**sum的栈帧（执行）**

```
mov   0xc(%ebp), %eax
mov   0x8(%ebp), %edx
```

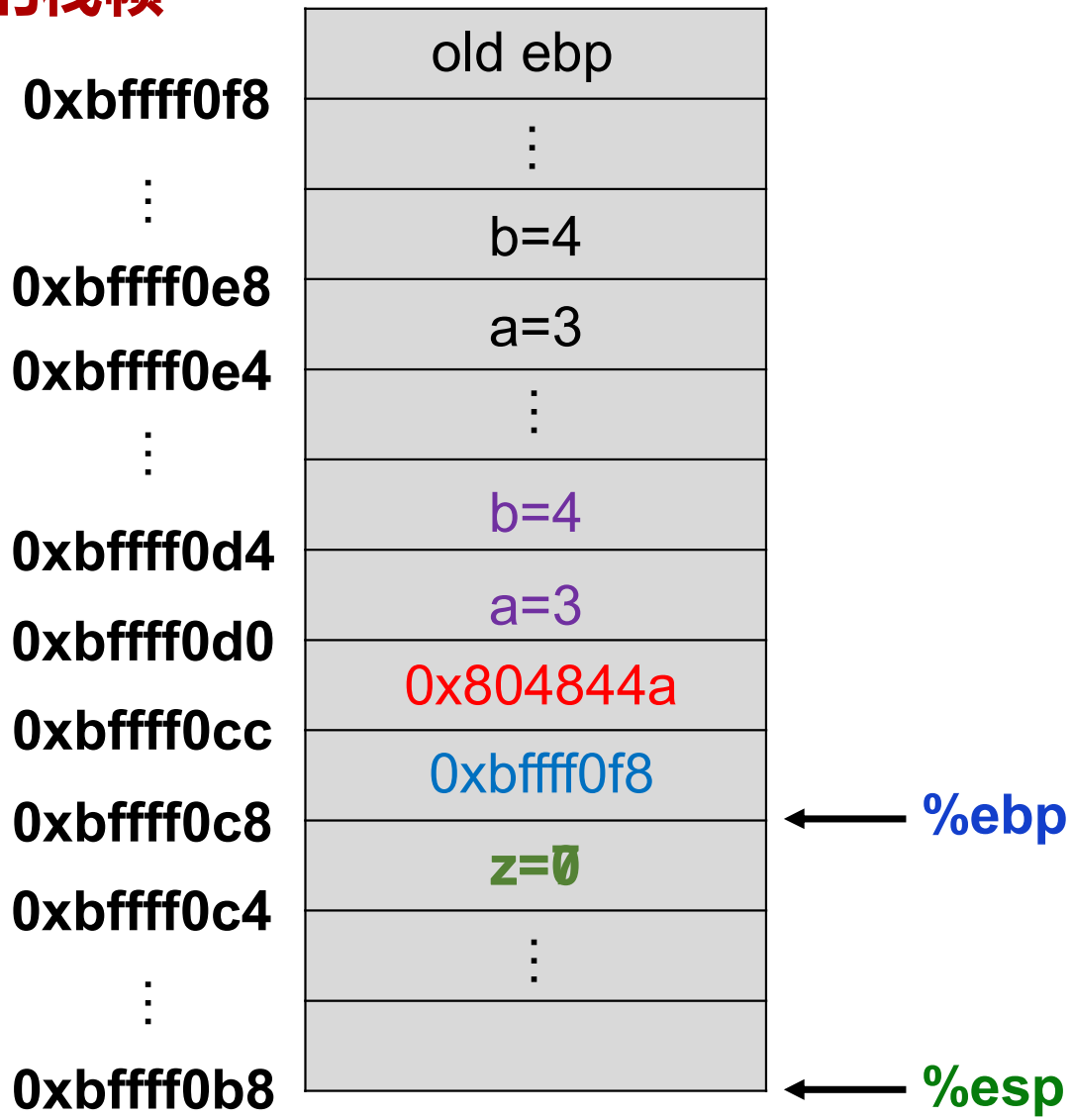
# sum.c 的栈帧

## sum的栈帧（执行）

```
mov   0xc(%ebp), %eax
mov   0x8(%ebp), %edx
add   %edx, %eax
mov   %eax, -0x4(%ebp)
mov   -0x4(%ebp), %eax
```

b=4 → %eax　+　a=3 → %edx　➡　%eax

**%eax=7**



# sum.c 的栈帧

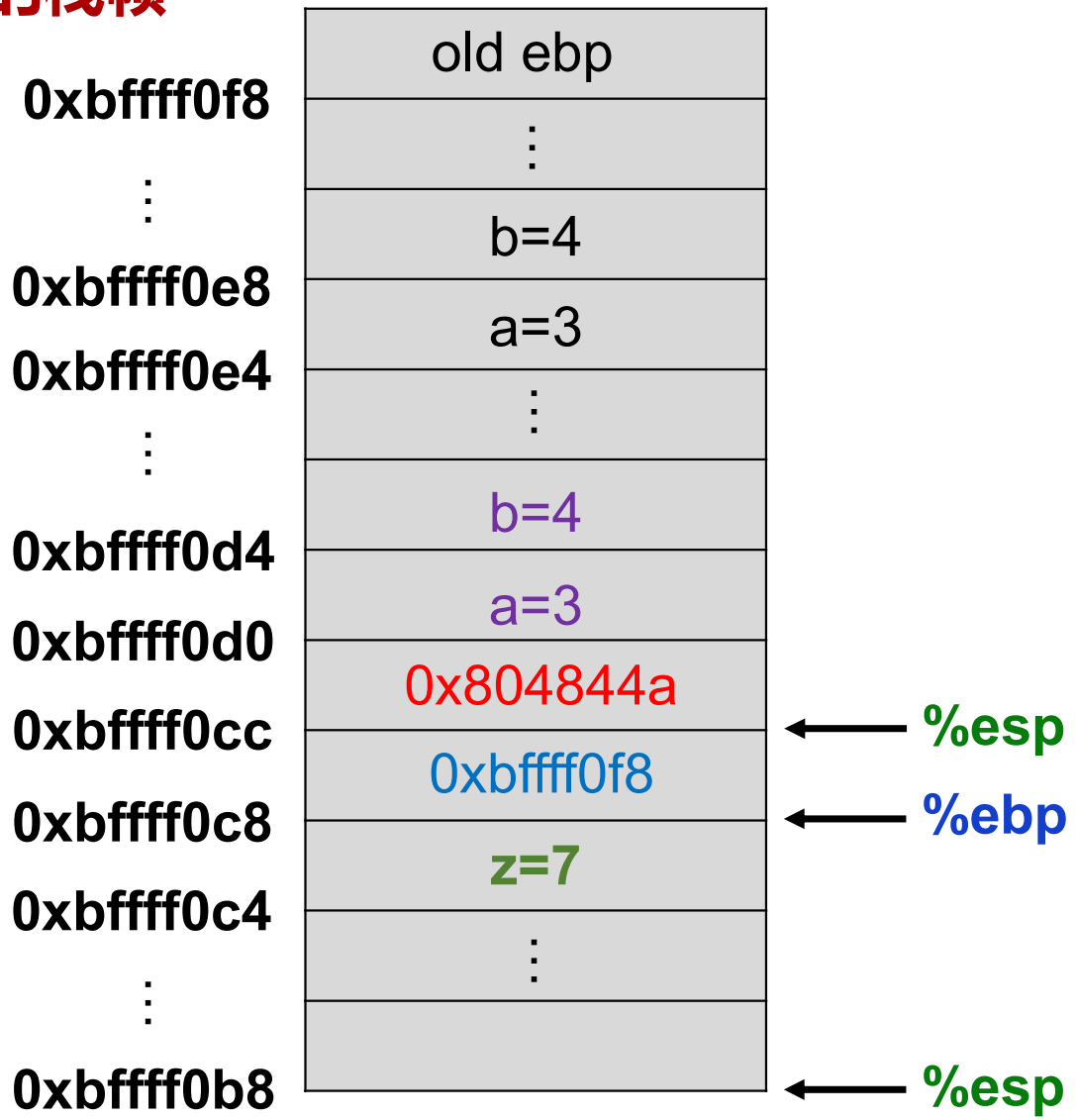
**sum的栈帧（返回）**

**leave =**

```
mov   %ebp, %esp
pop   %ebp
```

%ebp =

```
804847f:       c9                      leave
8048480:       c3                      ret
```



# sum.c 的栈帧

**sum的栈帧（返回）**

**leave =**

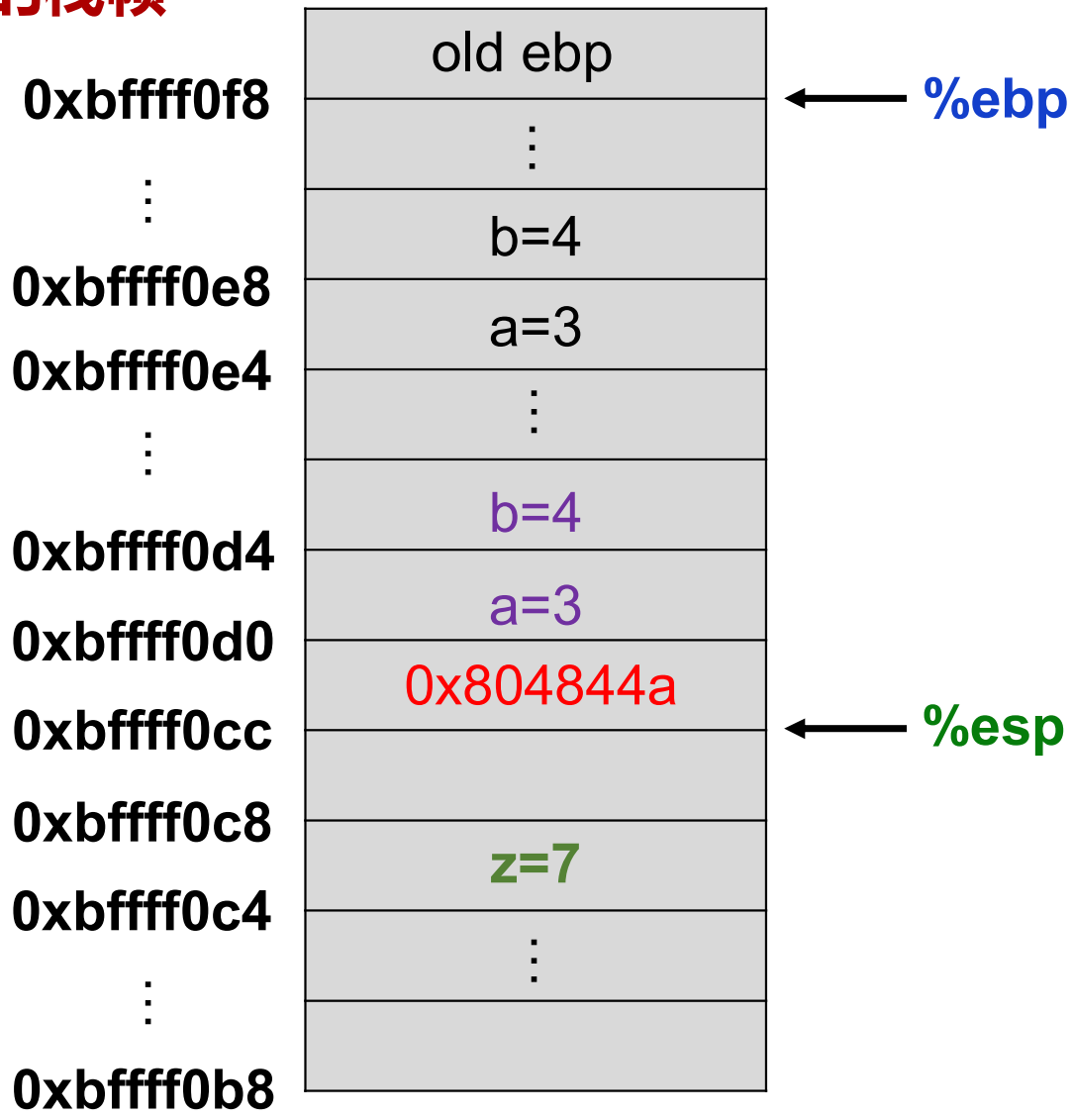
```
mov   %ebp, %esp
pop   %ebp
```

**ret =**

```
pop   %eip
```

%eip =

```
804847f:       c9                      leave
8048480:       c3                      ret
```



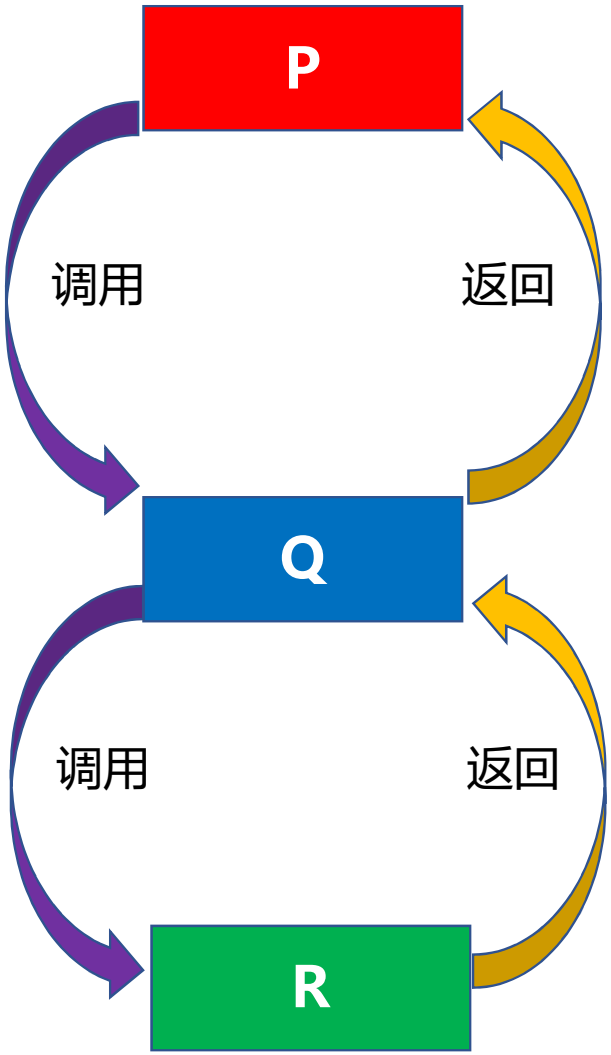
# 嵌套调用

**P调用Q，**

**Q在运行中再调用R**

```
void P()
{
        …… ……
        Q()
        …… ……
}

int Q()
{
        …… ……
        R()
        return;
}
```

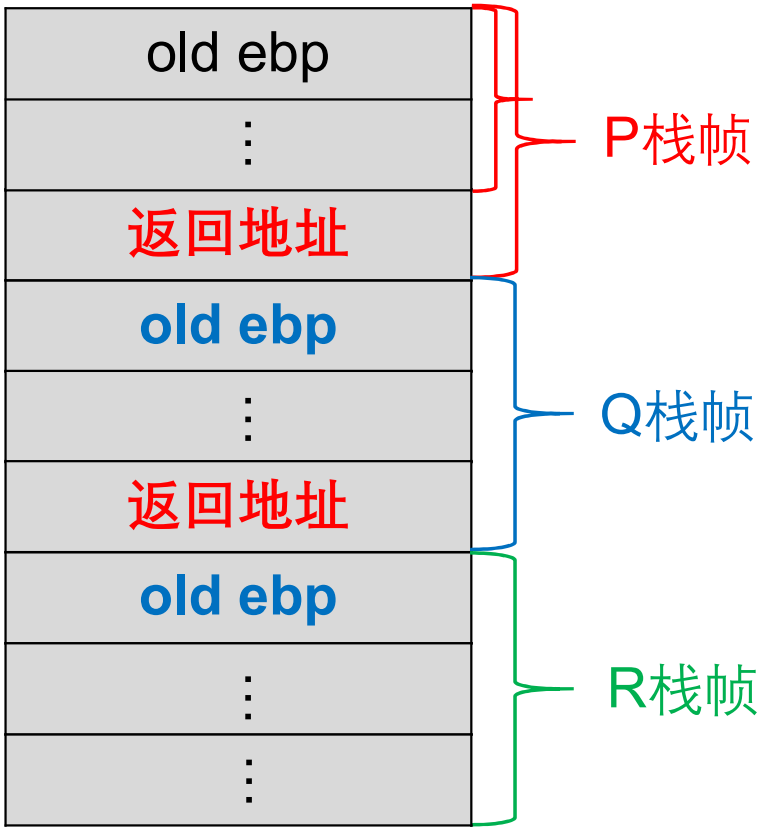


# 嵌套调用

## 嵌套调用举例

```
void P()
{
        …… ……
        Q()
        …… ……
}

int Q()
{
        …… ……
        R()
        return;
}
```



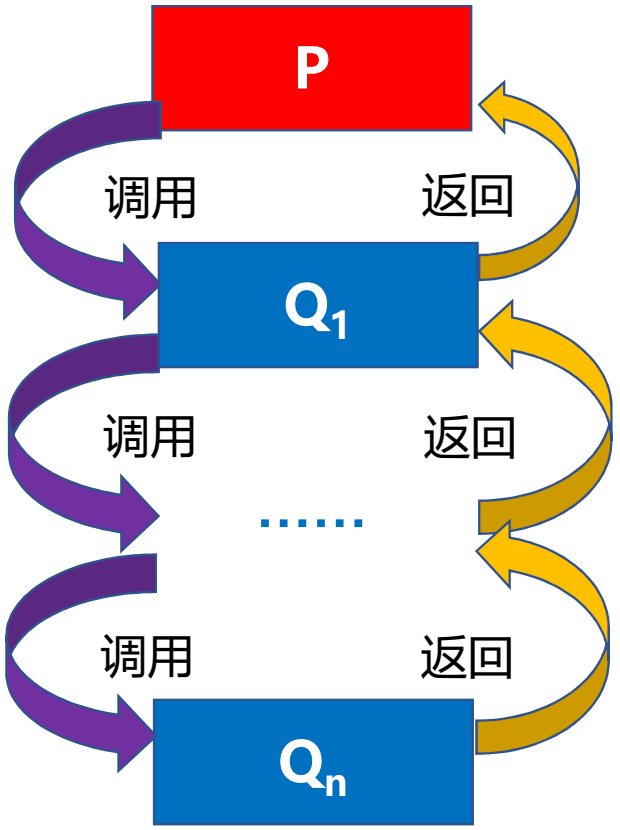
# 递归调用

**P调用Q，**

**Q在运行中再调用自身**

```
void P()
{
        …… ……
        Q()
        …… ……
}

int Q()
{
        …… ……
        Q()
        return;
}
```

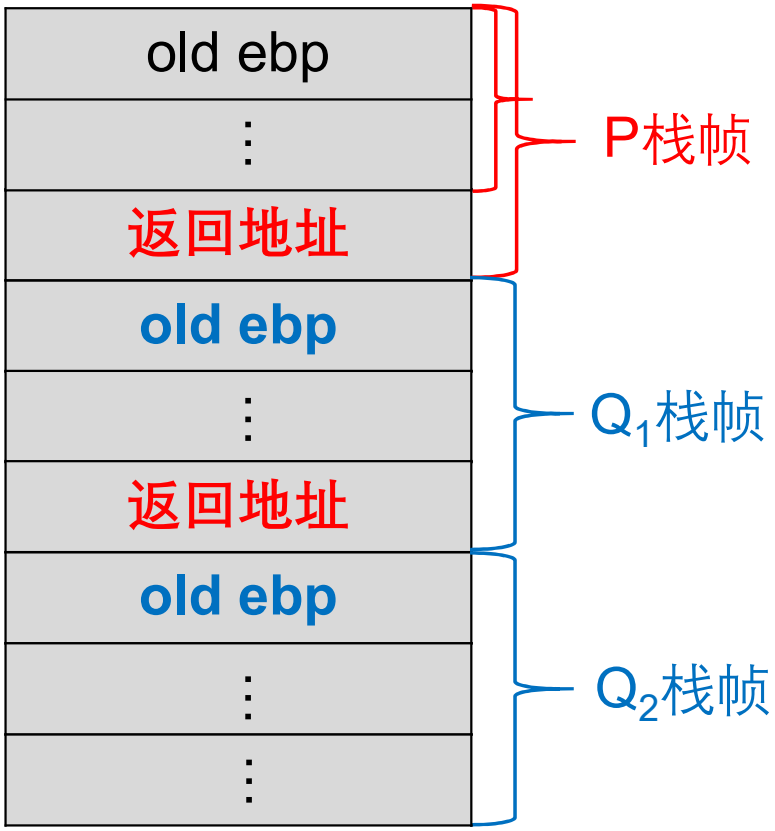


# 递归调用

## 递归调用举例

```
void P()
{
        …… ……
        Q()
        …… ……
}

int Q()
{
        …… ……
        Q()
        return;
}
```



# SWAP函数

```
void swap(int *xp,int *yp)
{
int t0 = *xp;
int t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

```
swap:
pushl   %ebp
movl    %esp,%ebp
pushl   %ebx

movl    8(%ebp),%edx
movl    12(%ebp),%ecx
movl    (%edx), %ebx
movl    (%ecx), %eax
movl    %eax, (%edx)
movl    %ebx, (%ecx)

popl    %ebx
popl    %ebp
ret
```

Set Up

Body

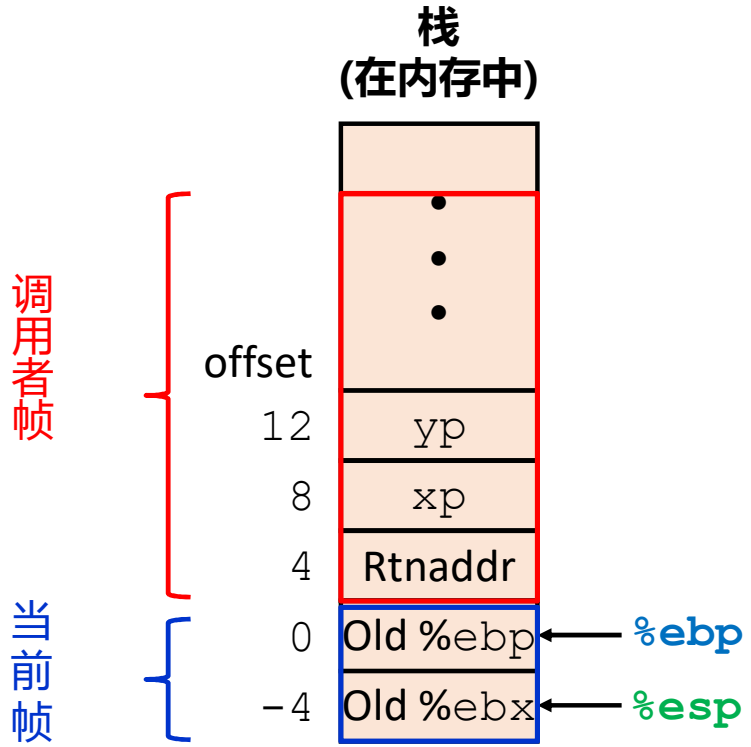
Finish

# SWAP函数

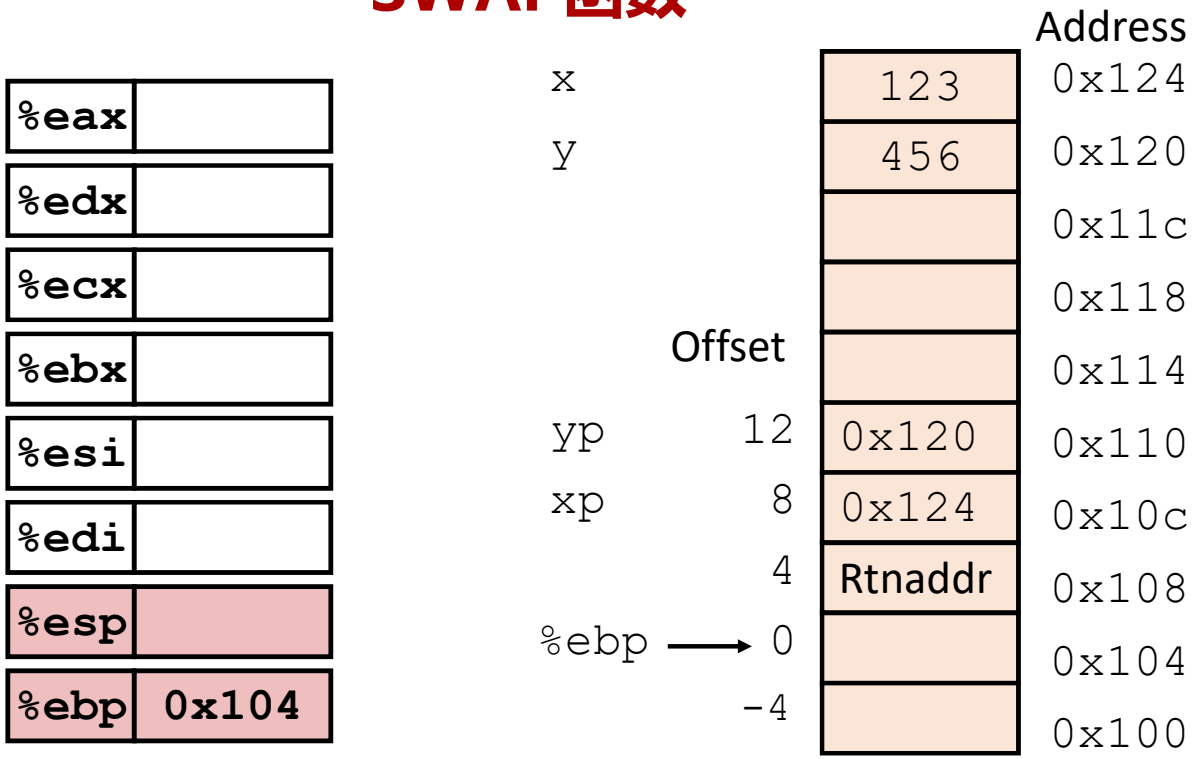
```
void swap(int *xp,int *yp)
{
int t0 = *xp;
int t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp),%ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp(t0)
movl (%ecx), %eax  # eax = *yp(t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```

| Register | Value |
|---|---|
| %edx | xp |
| %ecx | yp |
| %ebx | t0 |
| %eax | t1 |

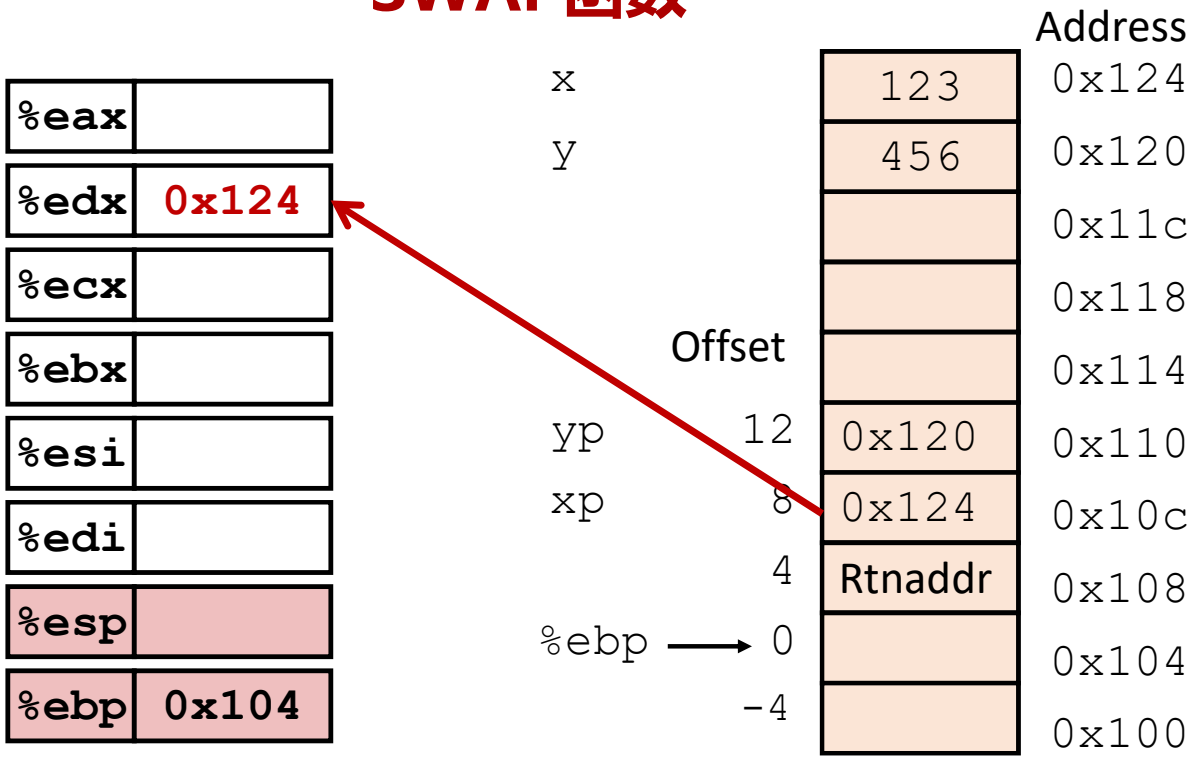


# SWAP函数



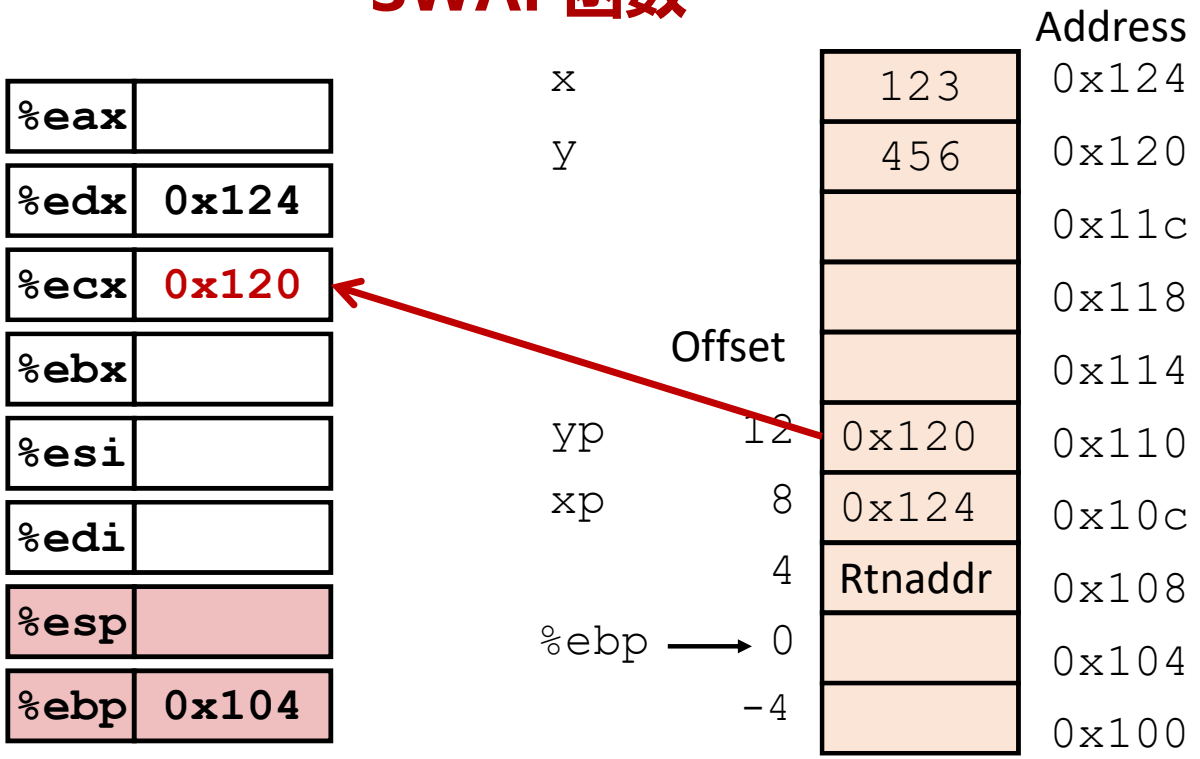

```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp),%ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp(t0)
movl (%ecx), %eax  # eax = *yp(t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```

# SWAP函数



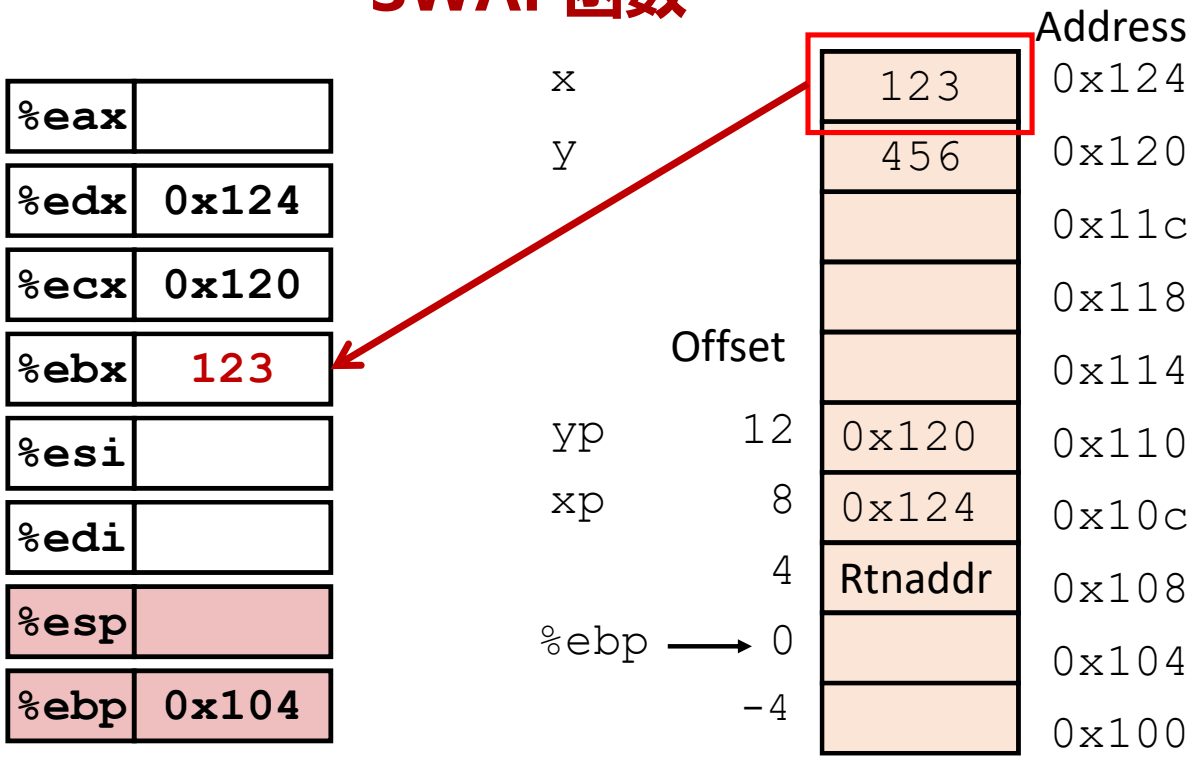

```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp),%ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp(t0)
movl (%ecx), %eax  # eax = *yp(t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```

# SWAP函数



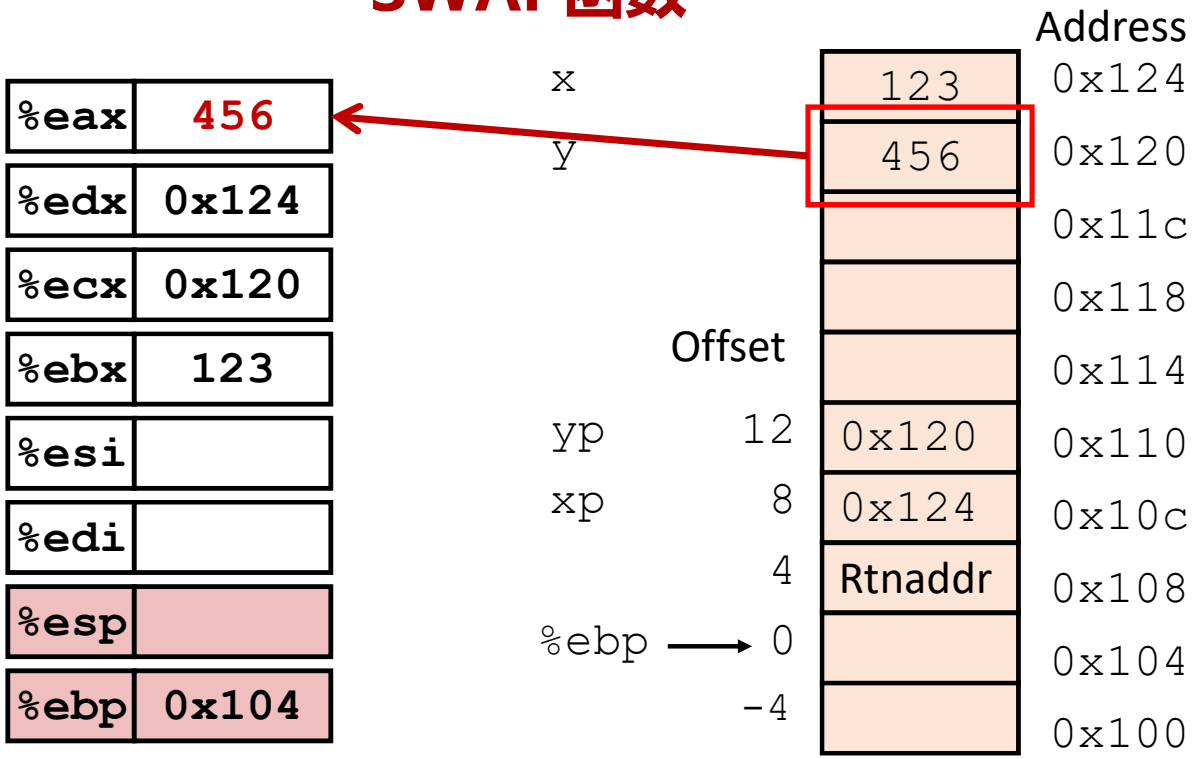

```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp),%ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp(t0)
movl (%ecx), %eax  # eax = *yp(t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```

# SWAP函数



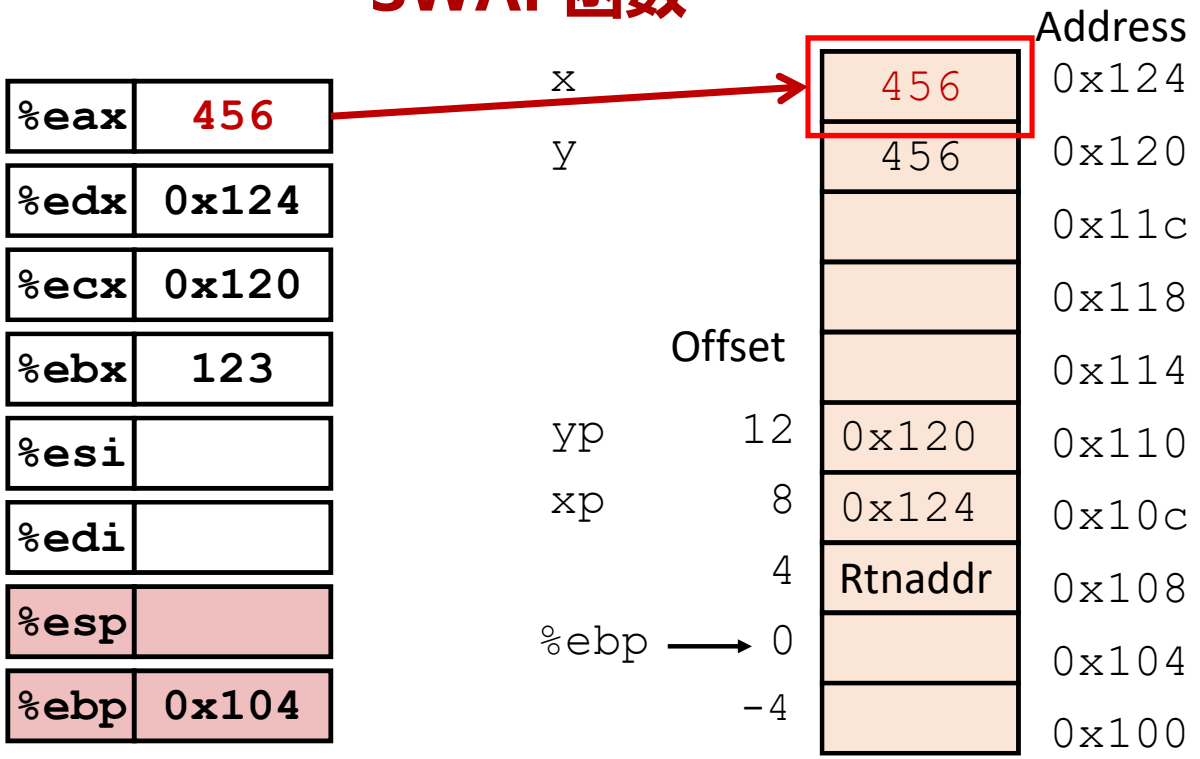

```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp),%ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp(t0)
movl (%ecx), %eax  # eax = *yp(t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```

# SWAP函数



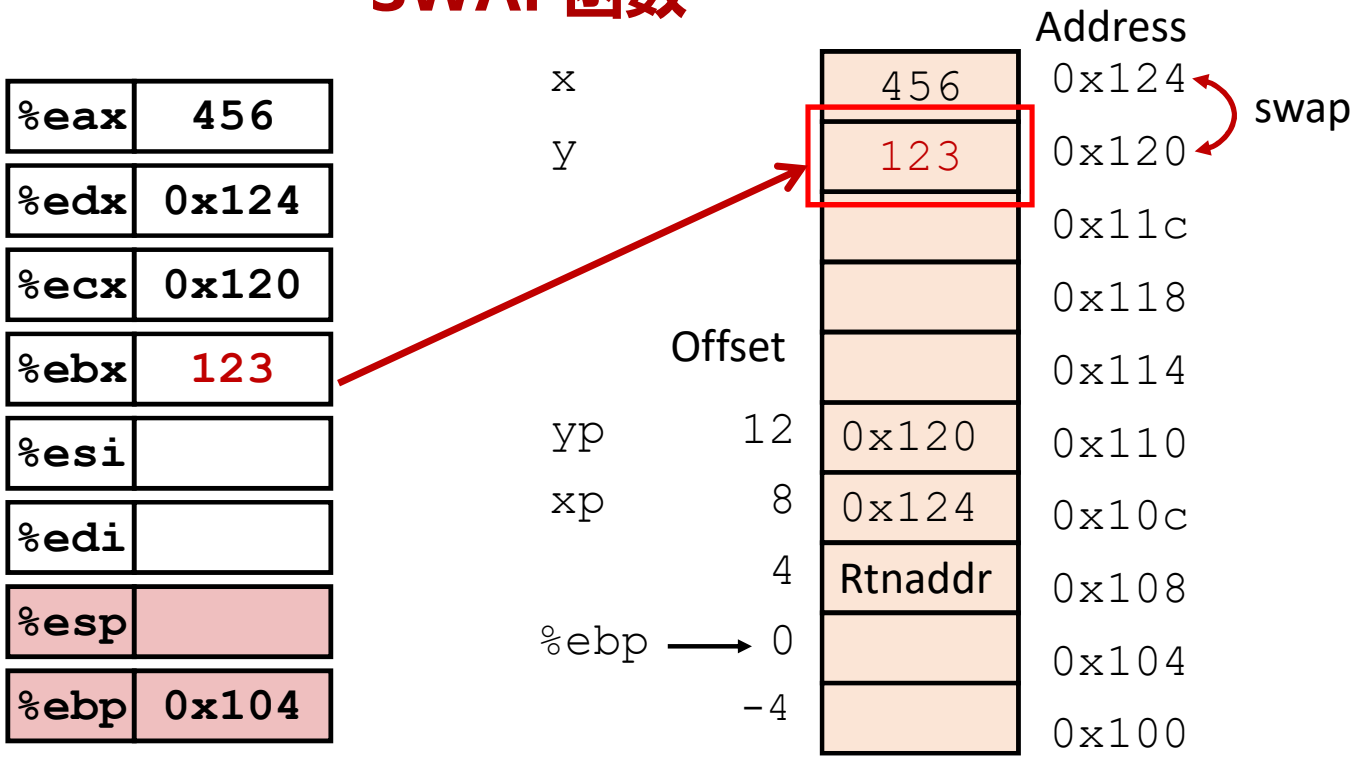

```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp),%ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp(t0)
movl (%ecx), %eax  # eax = *yp(t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```

# SWAP函数




```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp),%ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp(t0)
movl (%ecx), %eax  # eax = *yp(t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```

# SWAP函数




```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp),%ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp(t0)
movl (%ecx), %eax  # eax = *yp(t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```

# 课堂习题

栈中1、2、3处的内容分别是：

A. 1-返回地址；2-参数1；3-old ebp

B. 1-参数1；2-返回地址；3-old ebp

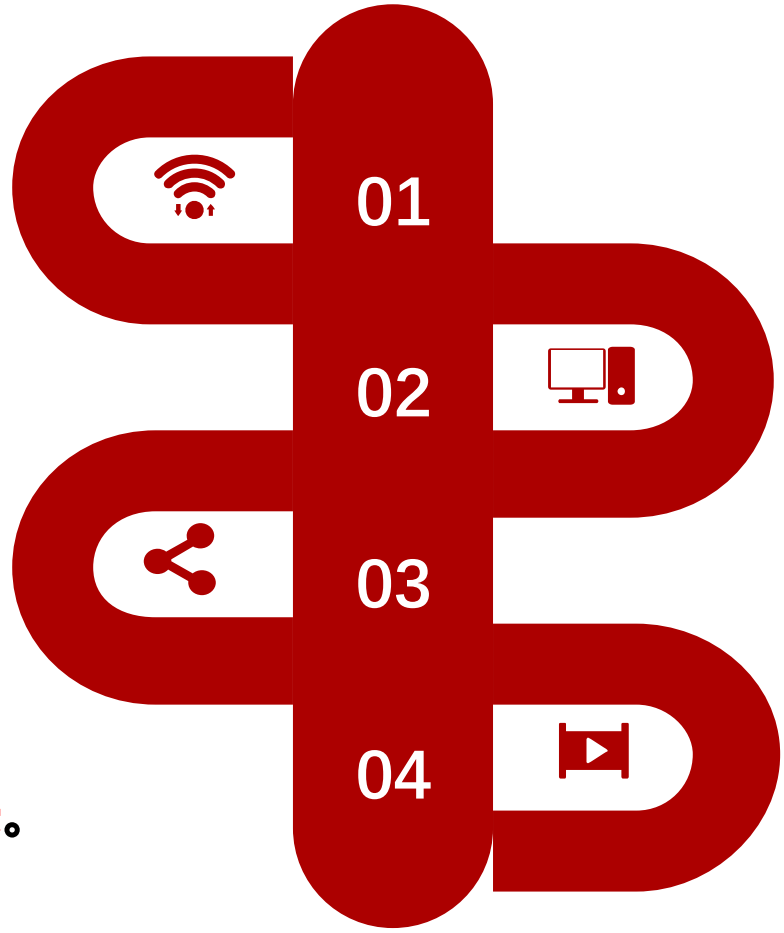
C. 1-返回地址；2-old ebp；3-参数1

D. 1-参数1；2-old ebp；3-返回地址

# 小结

① 过程及函数**调用**，
都是通过**栈**来实现

② 每次调用时的**参数**按照
**固定顺序**存放在栈中；
过程/函数**内部的变量**
**存放顺序与编译器有关**。




③ 过程/函数执行完毕后，
会**释放**其所占用的栈空间。

④ 对**返回地址**及**保存的**
**ebp**的修改会导致程序
错误和崩溃，常被用来
作为**黑客攻击**手段。

# 下一节：数据

湖南大学
《计算机系统》课程教学组