Local Variables, Data Frames and Recursive Functions

Static Local Variables

Global variables

Accessible by all modules

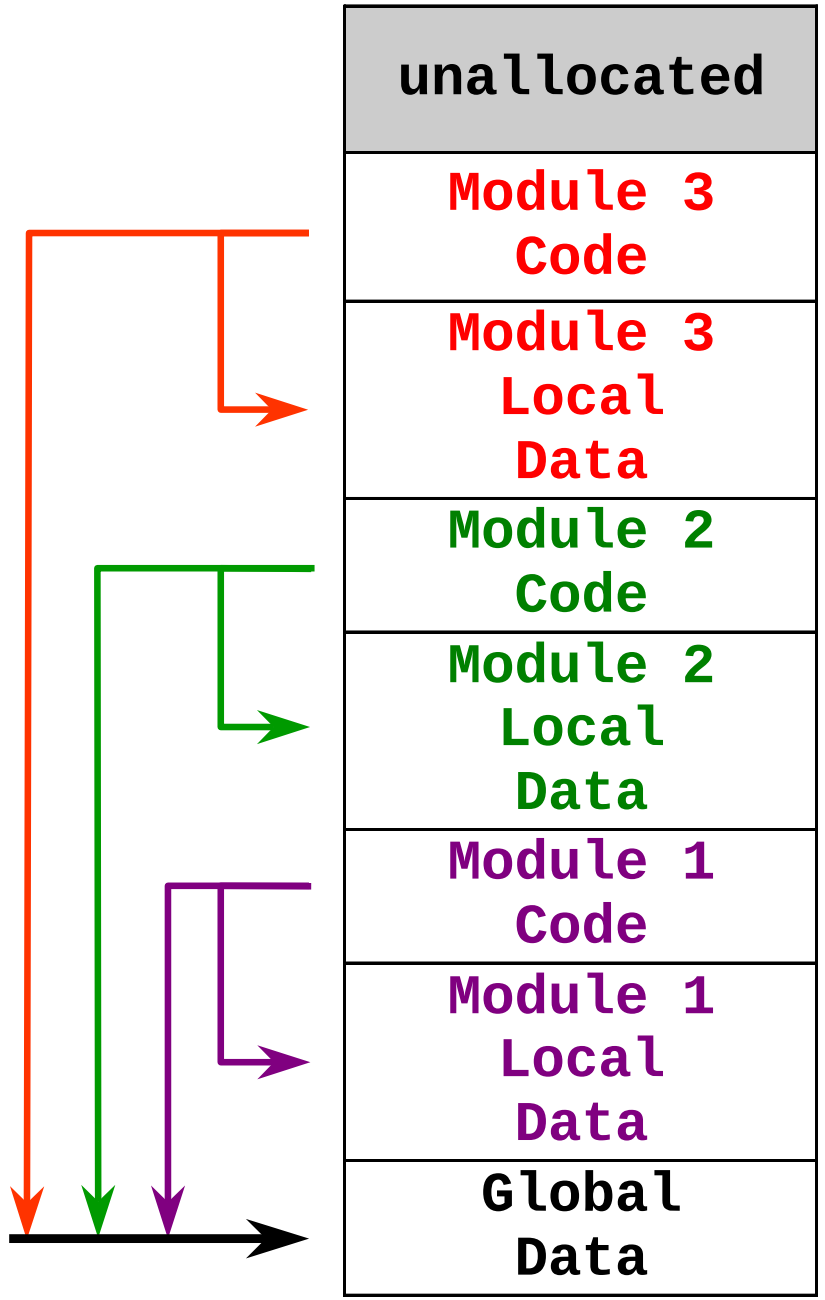
Any running code can read or write value

Local variables

Defined for one module

Only one module can read or write value

Stored in separate sections of memory



Recursive Code

Recursive function calls itself

Example: **`factorial`**

Coded as one module

Compiled into one section of machine code

```
factorial(n)
{
   if (n == 0)
      return 1;
   else
      return  n * factorial(n-1);
}
```

main creates first instance of function

Calls **`factorial(n)`**

Machine code runs with local variable **`n`**

factorial creates new instance of function

Calls **`factorial(n-1)`**

Same section of machine code runs again with local variable **`n-1`**

Each instance (function call) requires separate local data section

Local Variables — 2

Global variables

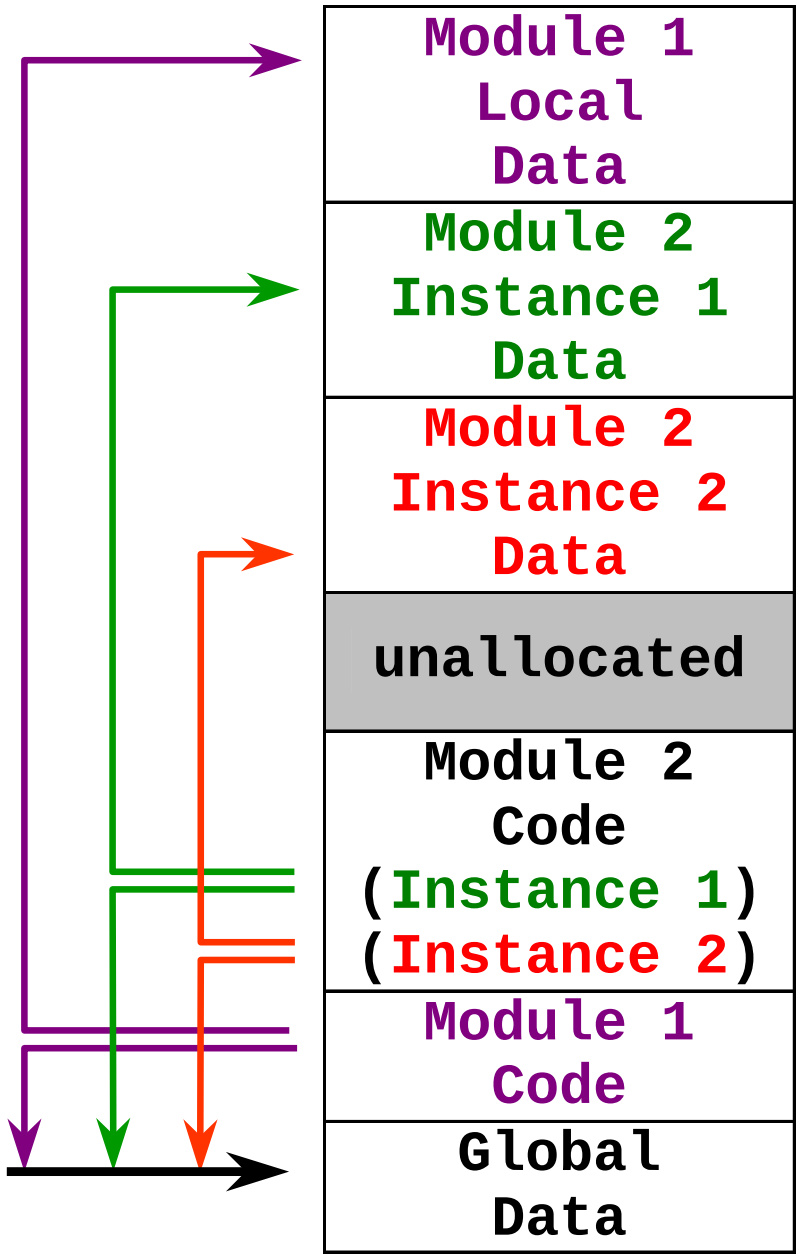
Accessible by all code modules

Local variables

Only one instance of code module can read or write value

Each instance stores local variables in separate section of memory

Local data sections allocated dynamically from top of memory



Global Variables

Store global variables in default data segment

Begin at address **`DS:start`**

- Linux
  - **`DS, CS`** point to same segment
  - System sets **`start`** after user code
- Windows
  - **`DS, CS`** point to separate segments
  - System sets **`start = 0`**

Allocate

- 4 bytes per integer (32-bit integers)
- 4 bytes per character (8-bit characters)

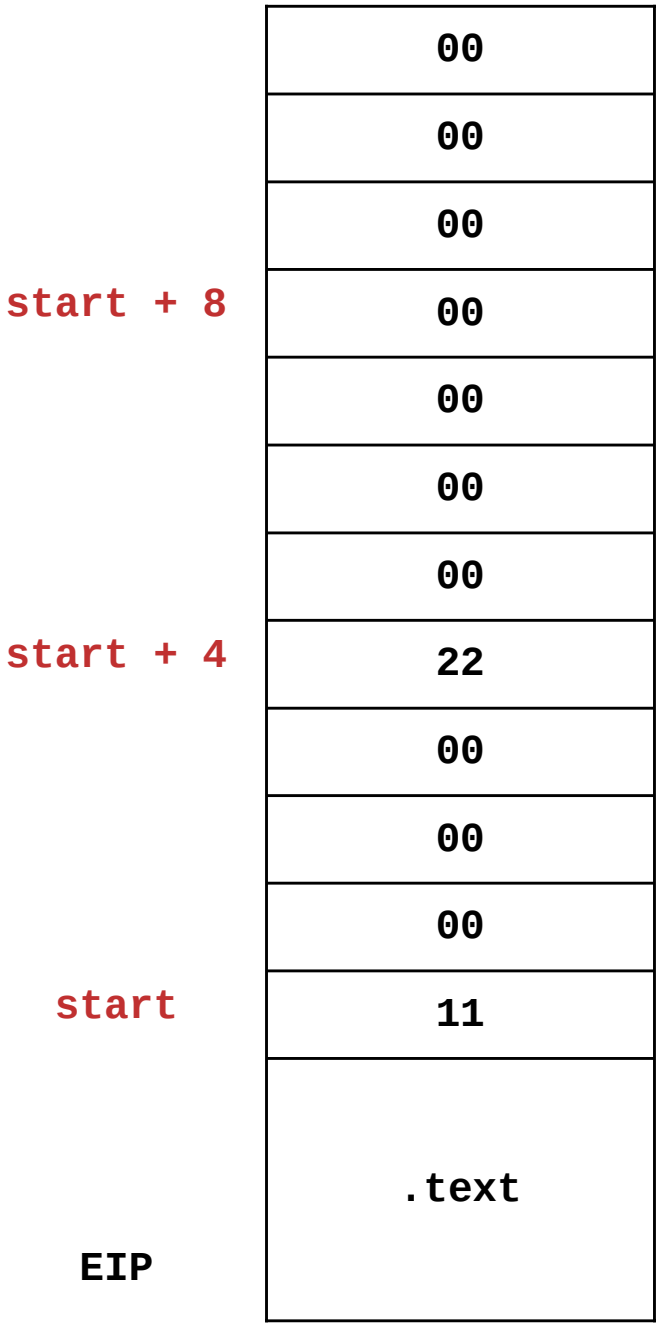
Continue in order of definition

Offset	Contents
`start + F`	`Z`
`start + E`	
`start + D`	
`start + C`	
`start + B`	`0`
`start + A`	`0`
`start + 9`	`0`
`start + 8`	`'A'`
`start + 7`	`Y`
`start + 6`	
`start + 5`	
`start + 4`	
`start + 3`	`X (byte 3)`
`start + 2`	`X (byte 2)`
`start + 1`	`X (byte 1)`
`start`	`X (byte 0)`

Global Variables with NASM

```
section .data
      start: dd 0x11
             dd 0x22
             dd 0
section .text
global _start
_start:
      lea edi,[start]      ; &(11)
      lea esi,[start + 4] ; &(22)
      call L1
mov eax,1
mov ebx,0
int 80h

L1:   mov ebx,[edi]         ; *(start)
      mov eax,[esi]         ; *(start + 4)
      mov [start + 8],eax
      ret
```



Local Data Table using BX

Table in data segment

Copy **`start`** into **`EBX`**

- First variable stored at **`[EBX]`**
- Second variable stored at **`[EBX + 04]`**
- Third variable stored at **`[EBX + 08]`**
- Continue in order of definition

Apply code to multiple data tables

Copy new pointer into **`EBX`**

Code remains generic

Used in some assembly programs

Offset	Contents
EBX + F	Z
EBX + E	
EBX + D	
EBX + C	
EBX + B	0
EBX + A	0
EBX + 9	0
EBX + 8	'A'
EBX + 7	Y
EBX + 6	
EBX + 5	
EBX + 4	
EBX + 3	X (byte 3)
EBX + 2	X (byte 2)
EBX + 1	X (byte 1)
EBX	X (byte 0)

Local Data Table with NASM

```
section .data
      T1: dd 0x11
          dd 0x22
          dd 0
      T2: dd 0x33
          dd 0x44
          dd 0
section .text
global _start
_start:
      mov ebx, T1
      call func
      mov ebx, T2
      call func
mov eax,1
mov ebx,0
int 80h
func: mov eax, [ebx]
      add eax, [ebx + 4]
      mov [ebx + 8], eax
      ret
```

T2 + 8	00 00 00 00
T2 + 4	00 00 00 44
T2	00 00 00 33
T1 + 8	00 00 00 00
T1 + 4	00 00 00 22
T1	00 00 00 11
EIP	.text

Local Data Frame using EBP / ESP

Local data frame (activation frame)

EBP / ESP point to system stack

ESP points to last pushed value

Copy **ESP** into **EBP**

First variable stored at **[EBP – 4]**

Second variable stored at **[EBP – 8]**

Third variable stored at **[EBP – C]**

Continue in order of definition

Point **ESP** below variables

Data frame

Local variables between **ESP** and **EBP**

User stack starts below variables

Used by C compilers

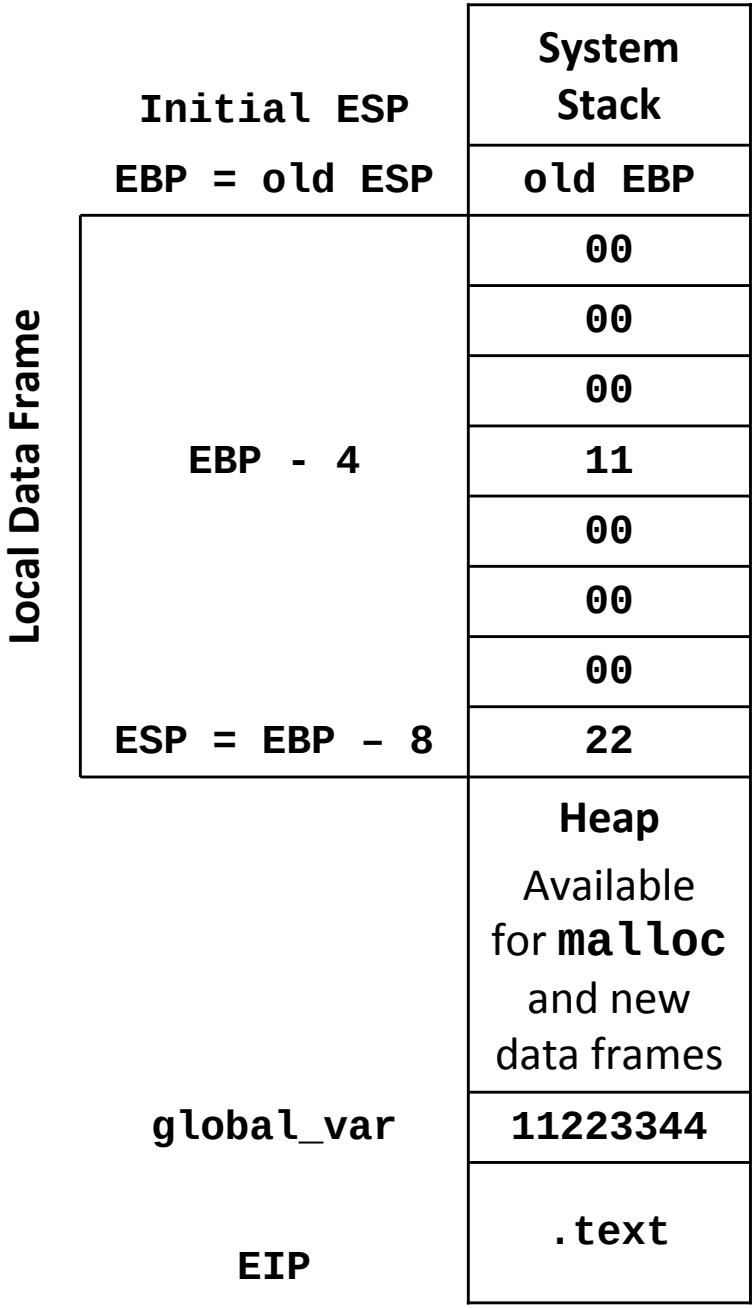
Offset	Contents
EBP - 1	X (byte 3)
EBP - 2	X (byte 2)
EBP - 3	X (byte 1)
EBP - 4	X (byte 0)
EBP - 5	Y
EBP - 6	
EBP - 7	
EBP - 8	
EBP - 9	0
EBP - A	0
EBP - B	0
EBP - C	'A'
EBP - D	Z
EBP - E	
EBP - F	
EBP - 10	

Local Data Frame with NASM

```
section .data
      global_var dd 0x11223344
section .text
global _start
_start:
   push ebp            ; save old EBP
   mov ebp, esp        ; EBP ← ESP
   sub esp, 8          ; allocate 2
                       ; integers X, Y

   mov [ebp-4], 0x11 ; initialize X
   mov [ebp-8], 0x22 ; initialize Y
   mov eax, [epb-4]
   add eax, [ebp-8]
   mov [ebp-4], eax  ; X = X + Y

   mov esp, ebp        ; de-allocate frame
                       ; ESP ← old ESP
   pop ebp             ; restore EBP
                       ; ESP ← initial ESP
mov eax,1
mov ebx,0
int 80h
```



C Language Compilations

Enter / Leave

Enter

Instruction

```
ENTER imm, nesting
```

Equivalent to code

```
PUSH EBP
MOV EBP, ESP
SUB ESP, imm        ; immediate = bytes allocated
```

If **`nesting > 0`** push pointers to frames 1 to n – 1 onto frame n

Not used by **`gcc`**

Leave

Instruction

```
LEAVE
```

Equivalent to code

```
MOV ESP, EBP
POP EBP
```

Simplified 32-bit C Program Structure

```
start_up_code:
        ; initialize registers
        ; prepare linked libraries
        ; prepare exception handlers
        ; push environment variables onto stack
        ; push argc and argv[] onto stack
        ; CALL main
        ; CALL finish code
        ; exit
main:   PUSH EBP
        MOV EBP, ESP
        SUB ESP, 4_bytes_per_integer
        Local operations
        MOV ESP, EBP
        POP EBP
        RET
finish_code:
```

De-allocate memory and return to DOS

Note: `gcc` rounds up bytes allocated to frame

```
SUB ESP, 0x10*[int(integers/4)+1]
```

Review: objdump

```
~/asm$ objdump -s ex1
ex27:     file format elf32-i386

Contents of section .interp:
 8048134 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048144 2e3200                               .2.
                            . . .
Contents of section .text:
 8048360 31ed5e89 e183e4f0 50545268 a0840408  1.^.....PTRh....
 8048370 68b08404 08515668 14840408 e8affffff  h....QVh........
                            . . .
```

```
~/asm$ objdump -d ex1
Disassembly of section .init:
080482e0 <_init>:
 80482e0:       55                      push   %ebp
 80482e1:       89 e5                   mov    %esp,%ebp
                            . . .
08048414 <main>:
 8048414:       55                      push   %ebp
 8048415:       89 e5                   mov    %esp,%ebp
 8048417:       83 e4 f0                and    $0xfffffff0,%esp
 804841a:       83 ec 20                sub    $0x20,%esp
 804841d:       c7 44 24 1c 03 00 00    movl   $0x3,0x1c(%esp)
```

Convenient Script for NASM Disassembler with gcc

Script file

Combine **objdump + grep + awk + less + ndisasm**

Edit file

ndmp

Write following line into file

```
ndisasm -b32 -e`objdump -d $1 | grep \<main\>
    | awk '{print substr($1,6,3)}'`h $1 | less
```

Save file and make executable

```
~/asm$ chmod u+x ndmp
```

Run

```
~/asm$ ndmp ex1
```

Displays file disassembly starting at <main>

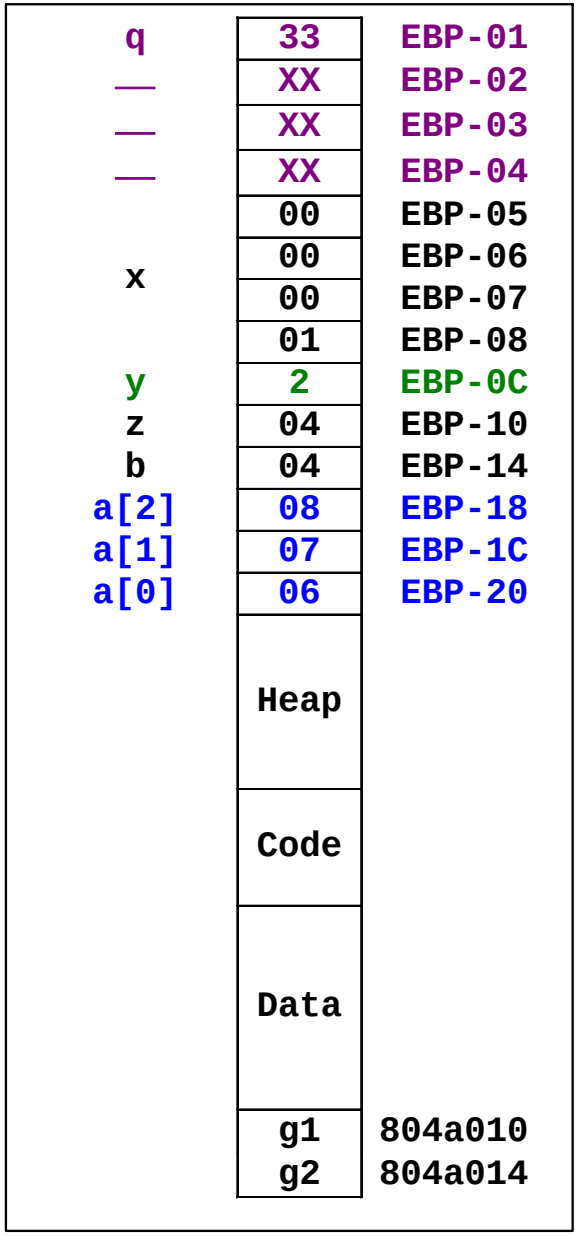
```
~/asm$ ndmp ex1
```

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83E4F0            and esp,byte -0x10
00000006  83EC20            sub esp,byte +0x20
00000009  C744241C03000000  mov dword [esp+0x1c],0x3
```

Variable Allocation

```
        int g1 = 9 , g2 = 10;
main()
{
     int x = 1;
     long y = 2;
     char q = '3';
     int z = 4;
     int a[3];
     int b = 5;
     a[0] = 6;
     a[1] = 7;
     a[2] = 8;
     g1 = g2;
}
080483b4 <main>:
080483b4   push ebp
080483b5   mov ebp,esp
080483b7   sub esp,byte +0x20
080483ba   mov dword [ebp-0x8],0x1
080483c1   mov dword [ebp-0xc],0x2
080483c8   mov byte [ebp-0x1],0x33
080483cc   mov dword [ebp-0x10],0x4
080483d3   mov dword [ebp-0x14],0x5
080483da   mov dword [ebp-0x20],0x6
080483e1   mov dword [ebp-0x1c],0x7
080483e8   mov dword [ebp-0x18],0x8
080483ef   mov eax,[0x804a014]
080483f4   mov [0x804a010],eax
080483f9   leave
080483fa   ret
```

`long = int` in 32-bit Linux



Variable Allocation — Counting Up From ESP

```
     int g1 = 9 , g2 = 10;
main()
{
      int x = 1;
      long long y = 2;
      char q = '3';
      int z = 4;
      int a[3];
      int b = 5;
      a[0] = 6;
      a[1] = 7;
      a[2] = 8;
      g1 = g2;
}
080483b4   push ebp
080483b5   mov ebp,esp
080483b7   and esp,byte -0x8
080483ba   sub esp,byte +0x30
080483bd   mov dword [esp+0x28],0x1
080483c5   mov dword [esp+0x18],0x2
080483cd   mov dword [esp+0x1c],0x0
080483d5   mov byte [esp+0x2f],0x33
080483da   mov dword [esp+0x24],0x4
080483e2   mov dword [esp+0x20],0x5
080483ea   mov dword [esp+0xc],0x6
080483f2   mov dword [esp+0x10],0x7
080483fa   mov dword [esp+0x14],0x8
08048402   mov eax,[0x804a014]
08048407   mov [0x804a010],eax
0804840c   leave
0804840d   ret
```

EBP	ESP + 30	
EBP - 04	ESP + 2C	q
EBP - 08	ESP + 28	x
EBP – 0C	ESP + 24	z
EBP – 10	ESP + 20	b
EBP – 14	ESP + 1C	y (H)
EBP – 18	ESP + 18	y (L)
EBP – 1C	ESP + 14	a[2]
EBP – 20	ESP + 10	a[1]
EBP – 24	ESP + 0C	a[0]
EBP – 28	ESP + 08	
EBP – 2C	ESP + 04	
EBP – 30	ESP	

EBP ← ESP

ESP ← ESP – 30h = EBP – 30h

Counting up from ESP
Counting down from EBP

equivalent

User program cannot use PUSH / POP

if Control Block

```
main()
{
     int x = 0, y = 2;
     if (x == y) {
          x = 2 * y;
          y = 2 * x;
     }
}
```

```
080483b4 <main>:
080483b4  push ebp
080483b5  mov ebp,esp
080483b7  sub esp,byte +0x10
080483ba  mov dword [ebp-0x4],0x0    ; x <– 0
080483c1  mov dword [ebp-0x8],0x2    ; y <– 2
080483c8  mov eax,[ebp-0x4]          ; eax <– x
080483cb  cmp eax,[ebp-0x8]          ; cmp x , y
080483ce  jnz 080483e0 <main+0x2c>   ; if x <> y LEAVE
080483d0  mov eax,[ebp-0x8]          ; EAX <– y
080483d3  add eax,eax                ; EAX <– 2 * EAX
080483d5  mov [ebp-0x4],eax          ; y <– EAX
080483d8  mov eax,[ebp-0x4]          ; EAX <– x
080483db  add eax,eax                ; EAX <– 2 * EAX
080483dd  mov [ebp-0x8],eax          ; y <– EAX
080483e0  leave                      ; exit program
080483e1  ret
```

Compound `if` Control Block

```
main
{
     int x = 0, y = 2, z = 3;
     if <--x == y || y > z && x > z) {
          x = 2 * y;
          y = 2 * x;
     }
}
```

```
080483b4 <main>
080483b4  push ebp
080483b5  mov ebp,esp
080483b7  sub esp,byte +0x10
080483ba  mov dword [ebp-0x4],0x0      ; x = [BP-02] <– 0
080483c1  mov dword [ebp-0x8],0x2      ; y = [BP-04] <– 2
080483c8  mov dword [ebp-0xc],0x3      ; z = [BP-06] <– 3
080483cf  mov eax,[ebp-0x4]            ; AX <– x
080483d2  cmp eax,[ebp-0x8]            ; CMP AX = x, y
080483d5  jz 80483e7 <main+0x33>       ; if x = y, LEAVE
080483d7  mov eax,[ebp-0x8]            ; AX <– y
080483da  cmp eax,[ebp-0xc]            ; CMP AX = y, z
080483dd  jng 80483f7 <main+0x43>      ; if y <= z, LEAVE
080483df  mov eax,[ebp-0x4]            ; AX <– x
080483e2  cmp eax,[ebp-0xc]            ; CMP AX = x, z
080483e5  jng 80483f7 <main+0x43>      ; if x <= z, LEAVE
080483e7  mov eax,[ebp-0x8]            ; AX <– y
080483ea  add eax,eax                  ; AX <– 2 * AX
080483ec  mov [ebp-0x4],eax            ; x <– AX
080483ef  mov eax,[ebp-0x4]            ; AX <– x
080483f2  add eax,eax                  ; AX <– 2 * AX
080483f4  mov [ebp-0x8],eax            ; y <– AX
080483f7  leave
080483f8  ret
```

while Control Block

```
main()
{
     int x = 0, y, z;
     while (x < 10){
          y = x++;
          z = ++x;
     }
}
080483b4 <main>
                    080483b4  push ebp
                    080483b5  mov ebp,esp
                    080483b7  sub esp,byte +0x10
                    080483ba  mov dword [ebp-0x4],0x0          ; x = [BP-02] <— 0
                    080483c1  jmp 80483d7 <main+0x23>
                    080483c3  mov eax,[ebp-0x4]                ; AX <— x
                    080483c6  mov [ebp-0x8],eax                ; y <— AX
                    080483c9  add dword [ebp-0x4],byte +0x1    ; x <— x + 1
                    080483cd  add dword [ebp-0x4],byte +0x1    ; x <— x + 1
                    080483d1  mov eax,[ebp-0x4]                ; AX <— x
                    080483d4  mov [ebp-0xc],eax                ; z <— AX
                    080483d7  cmp dword [ebp-0x4],byte +0x9    ; break x >= 10
                    080483db  jng 80483c3 <main+0xf>           ; loop
                    080483dd  leave
                    080483de  ret

Loop runs 5 times, starting with x = 0, 2, 4, 6, 8
```

Looping with `loop` Instruction

Static loop

Uses **`loop`** instruction with preset **ECX**

```
        PUSH EBP
        MOV EBP, ESP
        SUB ESP, 0x0C
        MOV     DWORD PTR [EBP-04],0
        MOV     ECX,0005                    ; loop runs 5 times
L1:     MOV     EAX,[EBP-04]
        MOV     [EBP-08],EAX
        INC     DWORD [EBP-04]
        INC     DWORD [EBP-04]
        MOV     EAX,[EBP-04]
        MOV     [EBP-0x0C],EAX
        LOOP    L1
        LEAVE
```

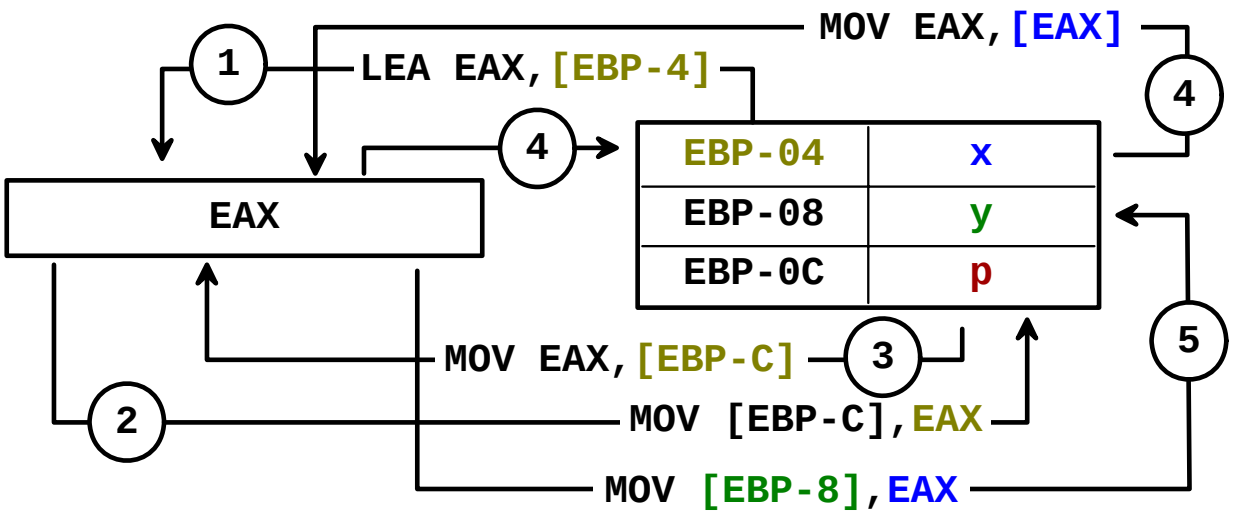
for Loop Example

```
main()
{
        int i,j;
        for (i = 0; i < 10; i++){
                j = 2 * i;
        }
}

080483b4 <main>
080483b4  push ebp
080483b5  mov ebp,esp
080483b7  sub esp,byte +0x10
080483ba  mov dword [ebp-0x4],0x0
080483c1  jmp 80483cf <main+0x1b>
080483c3  mov eax,[ebp-0x4]                    ; AX <– i
080483c6  add eax,eax                          ; AX <– 2 * AX
080483c8  mov [ebp-0x8],eax                    ; j <– AX
080483cb  add dword [ebp-0x4],byte +0x1        ; i++
080483cf  cmp dword [ebp-0x4],byte +0x9        ; break on i >= 10
080483d3  jng 80483c3 <main+0xf>               ; loop
080483d5  leave
080483d6  ret
```

Pointer Example

```
main()
{
        int x = 1025, y;
        int *p;
        p = &x;
        y = *p;
}
080483b4 <main>
080483b4  push ebp
080483b5  mov ebp,esp
080483b7  sub esp,byte +0x10
080483ba  mov dword [ebp-0x4],0x401   ; x  <- 1025
080483c1  lea eax,[ebp-0x4]           ; EAX <- &x
080483c4  mov [ebp-0xc],eax           ; p <- AX
080483c7  mov eax,[ebp-0xc]           ; EAX <- p
080483ca  mov eax,[eax]               ; EAX <- *p
080483cc  mov [ebp-0x8],eax           ; y <- EAX
080483cf  leave
080483d0  ret
```



String Allocation

```
main()
{
      char a = '7';
      char *b = "12345678";
      int i, x, y;
      x = a;
      y = a - '0';
      for (i = 0 ; b[i] != '\0' ; ++i){
            y = b[i] - '0';
      }
}
080483b4   push ebp
080483b5   mov ebp,esp
080483b7   sub esp,byte +0x20
080483ba   mov byte [ebp-0x1],0x37        ; a <— '7'
080483be   mov dword [ebp-0x8],0x80484d0 ; &b <— 0x80484d0
080483c5   movsx eax,byte [ebp-0x1]       ; EAX <— '7'
080483c9   mov [ebp-0x10],eax             ; x <— EAX
080483cc   movsx eax,byte [ebp-0x1]       ; EAX <— '7'
080483d0   sub eax,byte +0x30             ; EAX <— EAX - '0'
080483d3   mov [ebp-0x14],eax             ; y <— EAX
080483d6   mov dword [ebp-0xc],0x0        ; i <— 0
080483dd   jmp 80483f5 <main+0x41>        ; loop
080483df   mov eax,[ebp-0xc]              ; EAX <— i
080483e2   add eax,[ebp-0x8]              ; EAX <— EAX + &b
080483e5   movzx eax,byte [eax]           ; EAX <— b[i]
080483e8   movsx eax,al
080483eb   sub eax,byte +0x30             ; EAX <— EAX - '0'
080483ee   mov [ebp-0x14],eax             ; y <— EAX
080483f1   add dword [ebp-0xc],byte +0x1 ; i<— i + 1
080483f5   mov eax,[ebp-0xc]              ; EAX <— i
080483f8   add eax,[ebp-0x8]              ; EAX <— i + &b
080483fb   movzx eax,byte [eax]           ; EAX <— [i + &b]
080483fe   test al,al                     ; CMP [i + &b], 0
08048400   jnz 80483df <main+0x2b>       ; loop if [i + &b] <> 0
08048402   leave
08048403   ret
```

a	37000000	EBP-04
&b	0x80484d0	EBP-08
i	00	EBP-0C
x	00	EBP-10
y	00	EBP-14
	code	
\0	00	0x80484d8
b[7]	38	0x80484d7
b[6]	37	0x80484d6
b[5]	36	0x80484d5
b[4]	35	0x80484d4
b[3]	34	0x80484d3
b[2]	33	0x80484d2
b[1]	32	0x80484d1
b[0]	31	0x80484d0

C Program with Called Functions

Variable name

Reference to a data value in memory

Symbolic label of a pointer to memory

Each instance of function

Receives parameters from calling function

Calling function passes data on stack

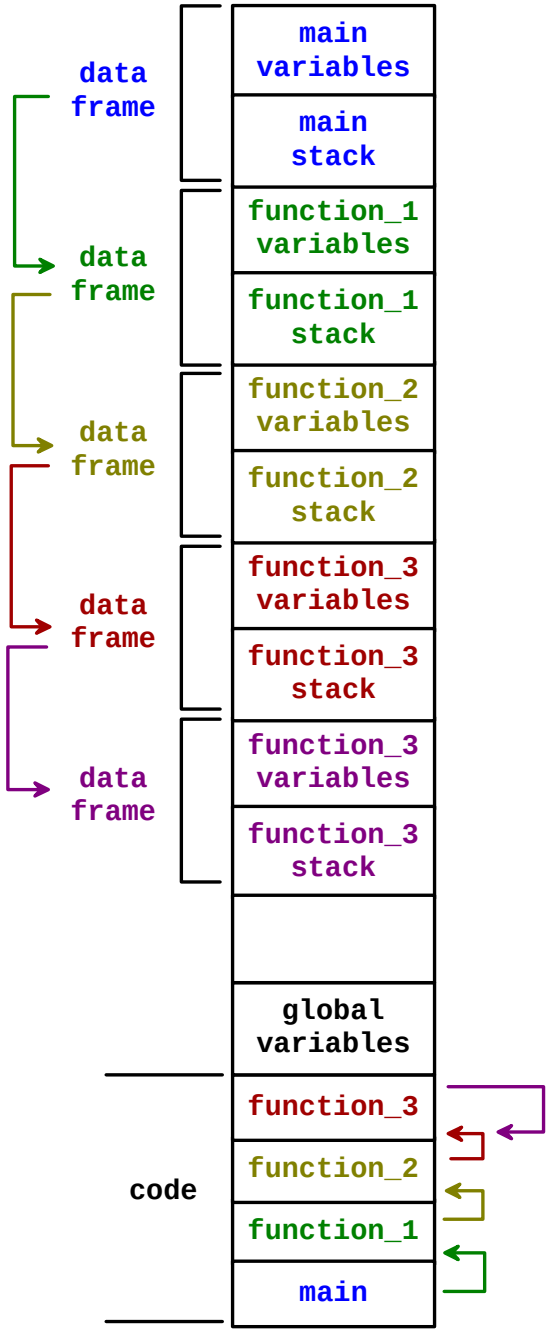
Stores variables in local data frame

Passes return value in register

Pushes and pops on local stack

Problem

How to keep track of local data frames?



x86 Function Model — 1

Global data

`int_g0` → `g0`

`int_g1` → `g0 + 4`

`int_g2` → `g0 + 8`

Local data frame

Local variables at top of frame

`int_0` → **`EBP-04`**

`int_1` → **`EBP-08`**

`int_2` → **`EBP-0C`**

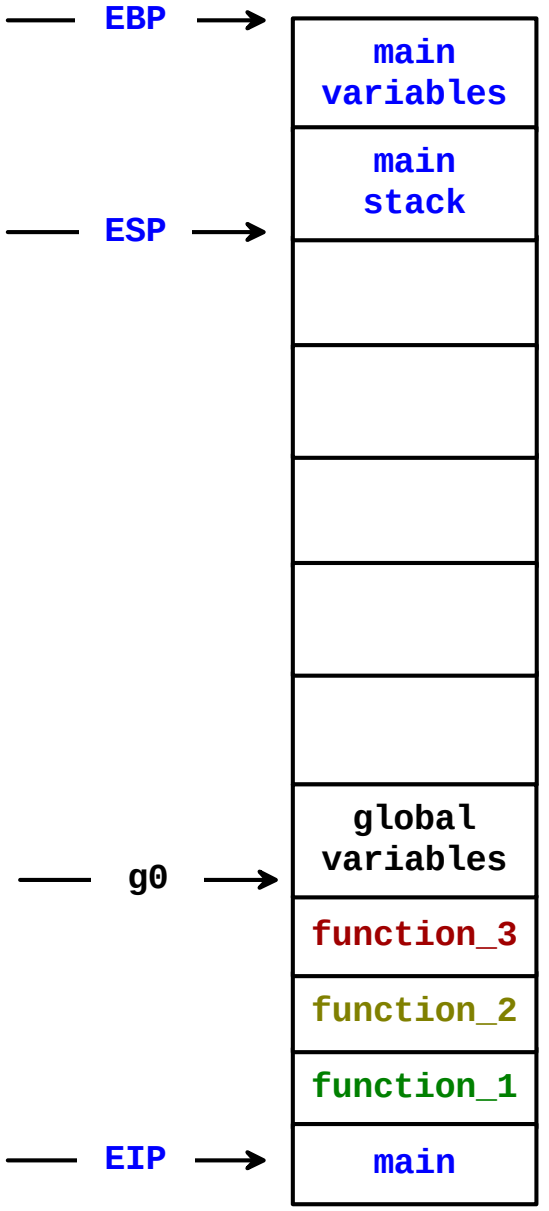
Local stack below local variables

Stack expands down (**`ESP`** ← **`ESP - 4`**)

Frame pointers

`EBP` points to top of frame

`ESP` points to bottom of frame



x86 Function Model — 2

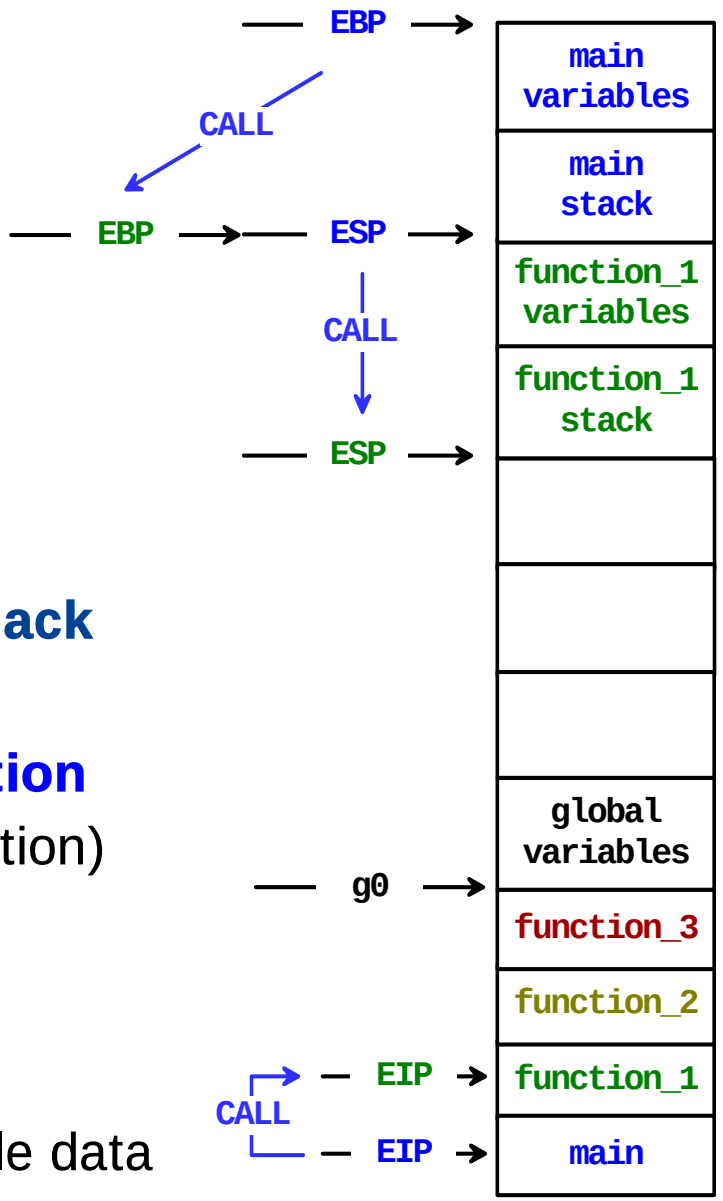
Data frame

EBP points to top of frame

ESP points to bottom of frame

Empty part of stack:

ESP = EBP – 4 bytes per variable defined in function



On function call push new data frame onto stack

New data frame for **called function**

Starts below data frame for **calling function**

Save $EBP_{calling}$ (for return to calling function)

Top of **new frame** is bottom of **old frame**

$EBP_{called} \leftarrow ESP_{calling}$

New stack starts below **new variables**

$ESP_{called} \leftarrow ESP_{calling}$ - size of variable data

ESP_{called} expands down ($ESP \leftarrow ESP - 4$)

x86 Function Model — 3

Called function performs:

PUSH EBP

Saves **$EBP_{calling}$** on stack of calling function

MOV EBP,ESP

EBP_{called} ← **$ESP_{calling}$**

SUB ESP, 4 bytes per defined variable

ESP_{called} ← **$ESP_{calling}$** - size of variable data

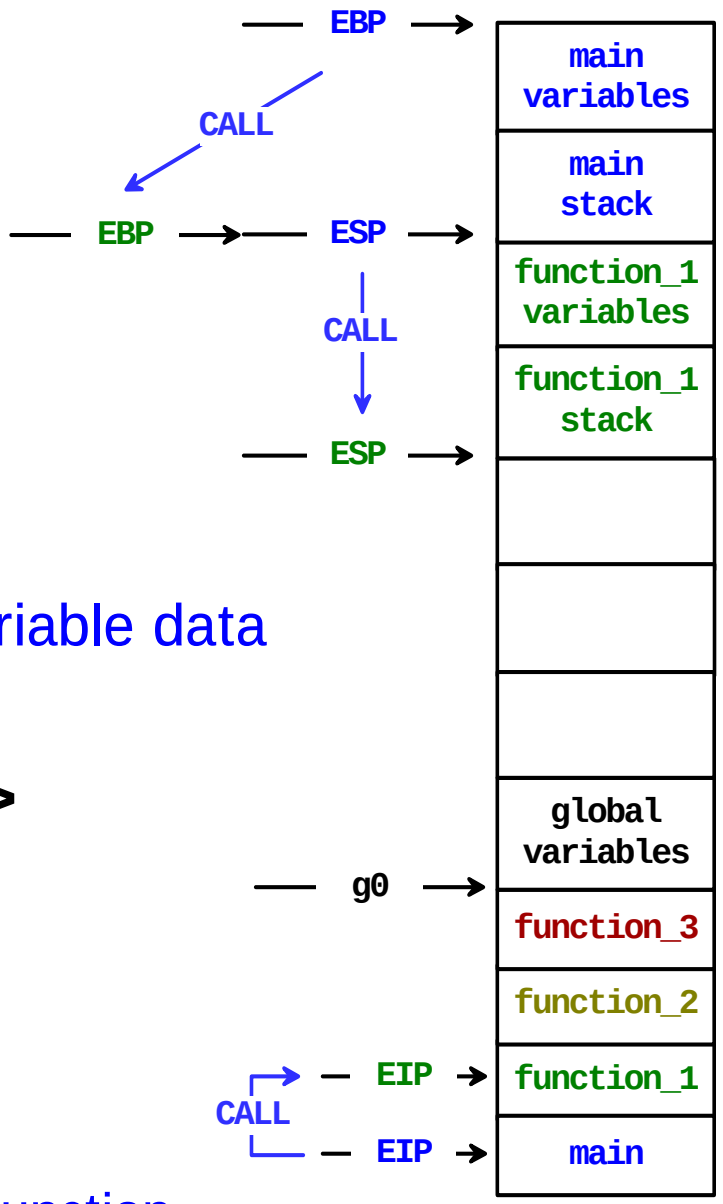
<Local operations of called function>

MOV ESP,EBP

Restores **ESP** ← **$ESP_{calling}$**

POP EBP

Restores **$EBP_{calling}$** from stack of calling function



x86 Function Model — 4

Calling module

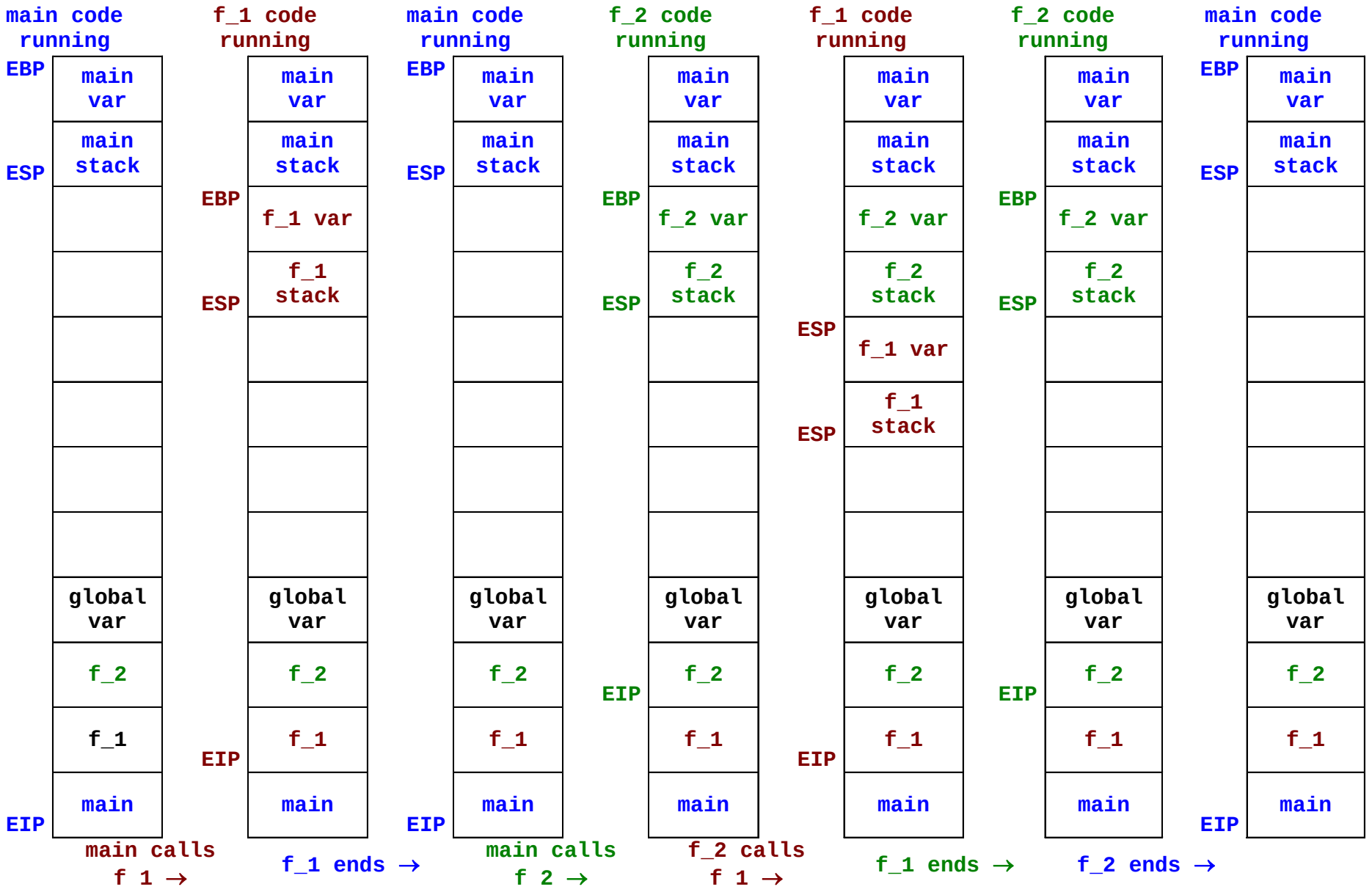
- **PUSH PARAMETERS** onto stack
- **Function Call**
  - **CALL EIP** — pushes **EIP** onto stack

Called function

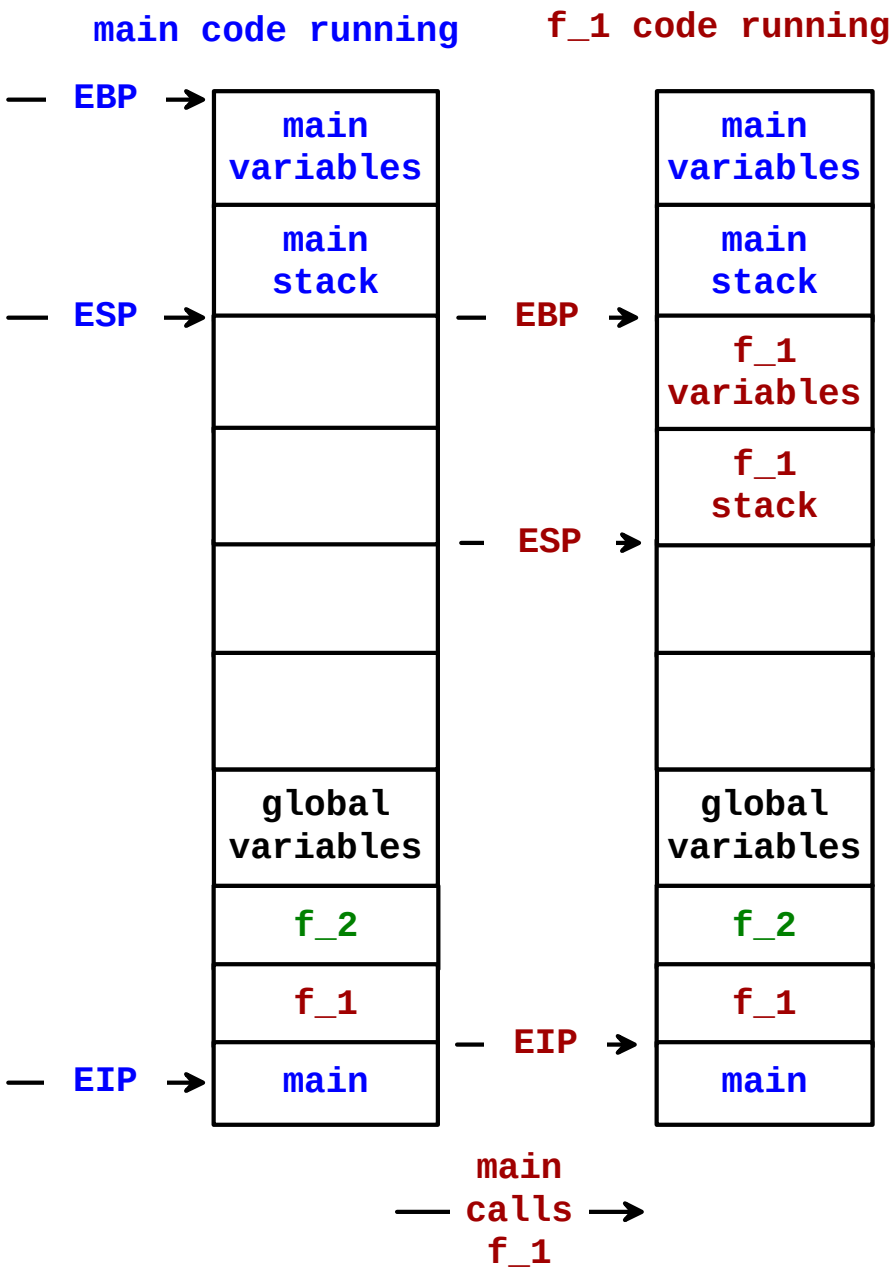
- **PUSH EBP**
- **MOV EBP, ESP**
- Copy **PARAMETERS** from stack
  - **MOV EAX, [EBP+08]**
- Perform **function**
- Return
  - **Single value** in **AX**
  - **List pointer** in **AX**

CALL
[EBP+08] = param
[EBP+04] = EIP
[EBP] = EBP

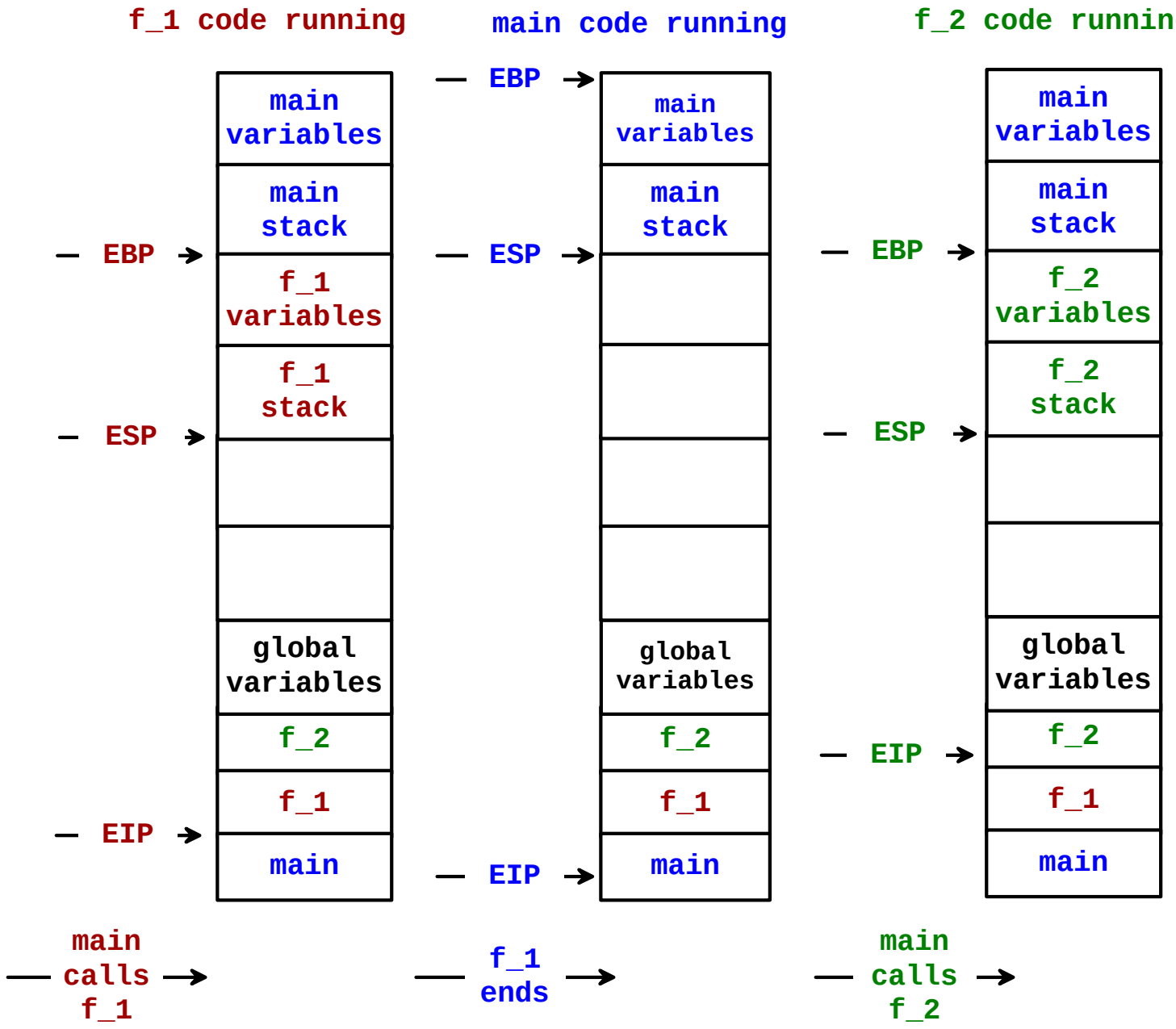
Memory in Time-Sequence — 1



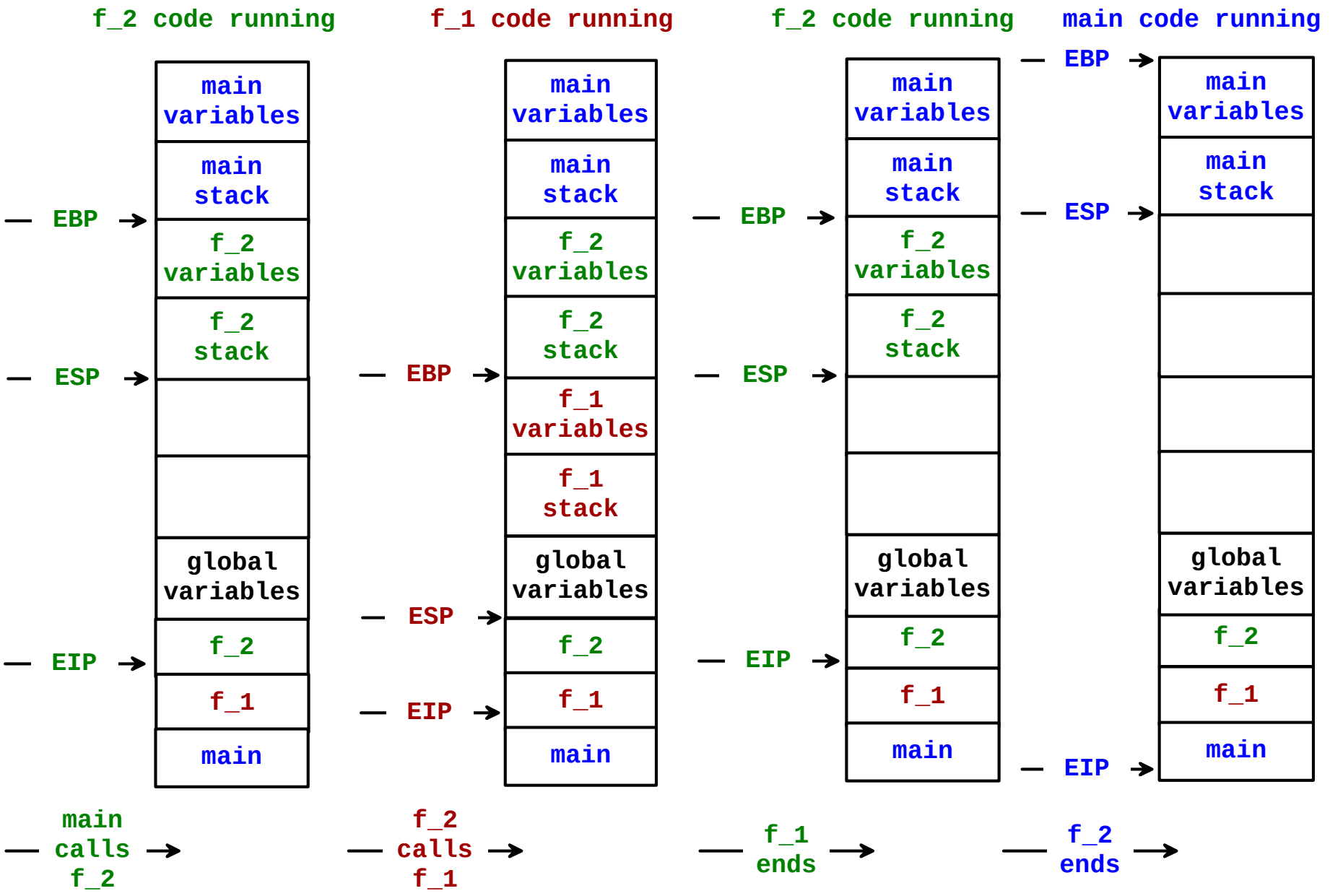
Memory in Time-Sequence — 2



Memory in Time-Sequence — 3



Memory in Time-Sequence — 4



Function Example — 1

```
main()
{
 int a = 0, x;
     x = function(a);
     a = x + 1;
}


int function(a)
{
 int r;
 r = a + 5;
 return(r);
}
```

Function Example — 2

080483b4 <main>: Value of ESP on entry to main: ESP_0

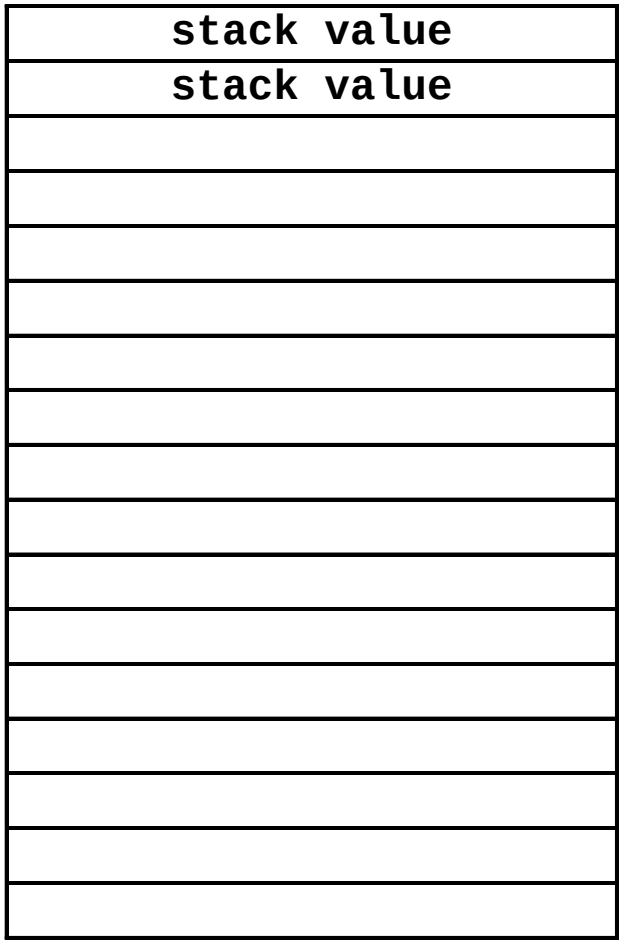
080483b4	push ebp		ESP_0 - 04
080483b5	mov ebp,esp	; EBP <– ESP_0 - 04	
080483b7	and esp,byte -0x10		ESP_0 - 04
080483ba	sub esp,byte +0x20		ESP_0 - 24
080483bd	mov dword [esp+0x1c],0x0	; a <– 0	
080483c5	mov eax,[esp+0x1c]	; EAX <– a	
080483c9	mov [esp],eax	; copy EAX to stack	ESP_0 - 24
080483cc	call 80483e2 <function>	; CALL function	ESP_0 - 28
080483d1	mov [esp+0x18],eax	; x <– EAX (return)	ESP_0 - 24
080483d5	mov eax,[esp+0x18]	; EAX <– x	
080483d9	add eax,byte +0x1	; EAX <– x + 1	
080483dc	mov [esp+0x1c],eax	; a <– EAX = x + 1	
080483e0	leave		ESP_0
080483e1	ret		

Example — 3

080483e2 <function>: Value of ESP on entry to function: **ESP_0 - 28**

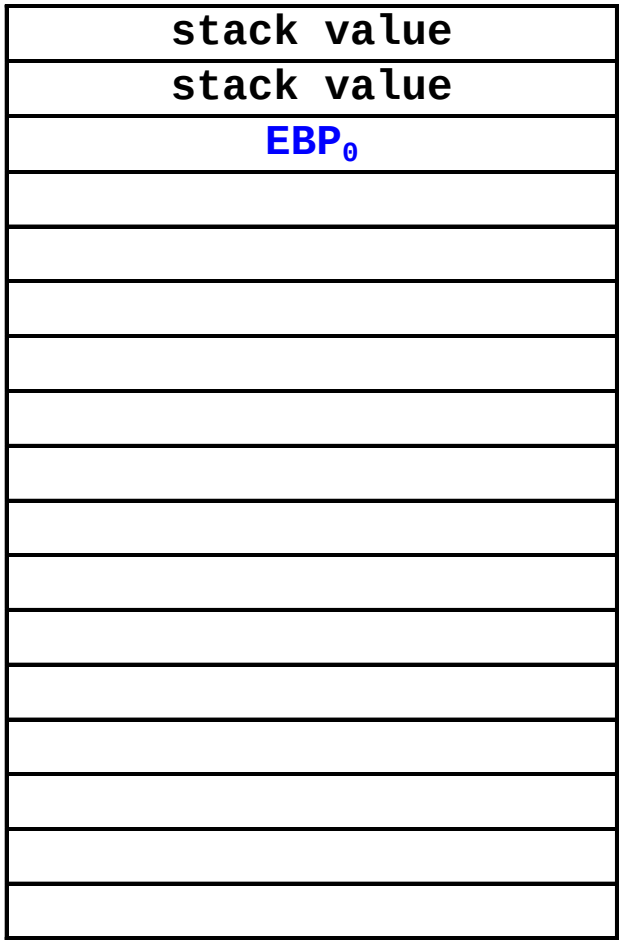
		; after CALL pushes EIP	ESP_0 – 28
080483e2	push ebp		ESP_0 – 2C
080483e3	mov ebp,esp	; EBP <– ESP_0 – 2C	
080483e5	sub esp,byte +0x10		ESP_0 – 3C
080483e8	mov eax,[ebp+0x8]	; EAX <– parameter a	
		; EBP + 0x8 = ESP_0 – 2C + 8 ; = ESP_0 – 24	
080483eb	add eax,byte +0x5	; EAX <– EAX + 5	
080483ee	mov [ebp-0x4],eax	; r <– EAX = a + 5	
080483f1	mov eax,[ebp-0x4]	; EAX <– a (return value)	
080483f4	leave		ESP_0 – 2C
080483f5	ret		ESP_0 – 28

Example — 4



← ESP_0 (points to the old stack value)

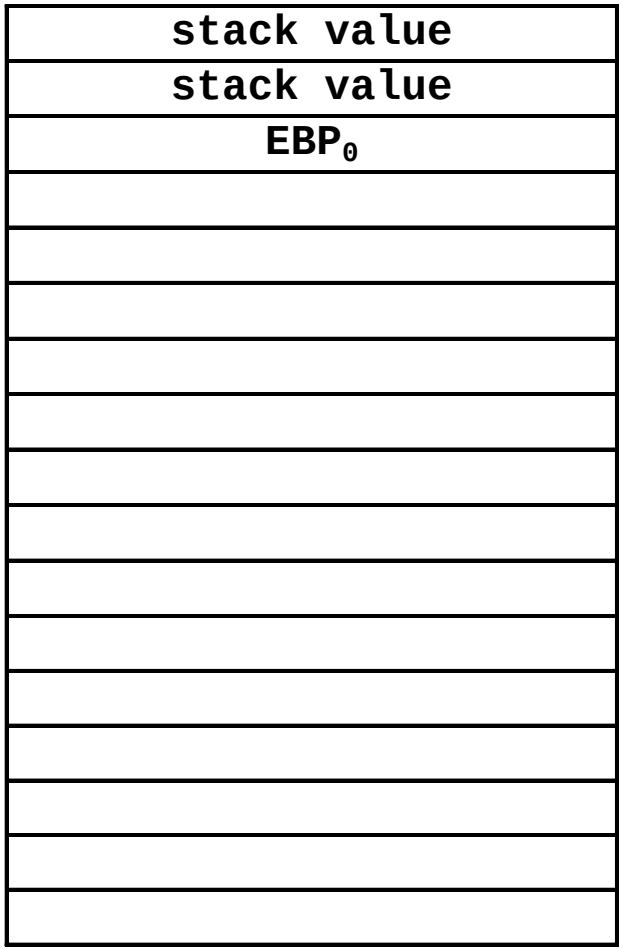
Example — 5



← ESP_0 (points to the old stack value)

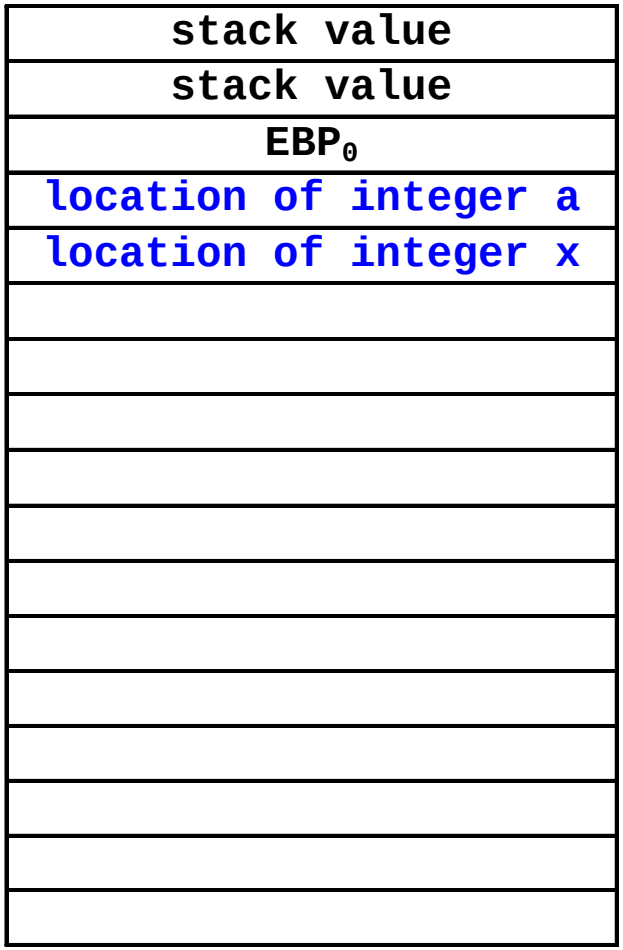
← ESP_1 after pushing old EBP_0 (is also EBP_1)

Example — 6



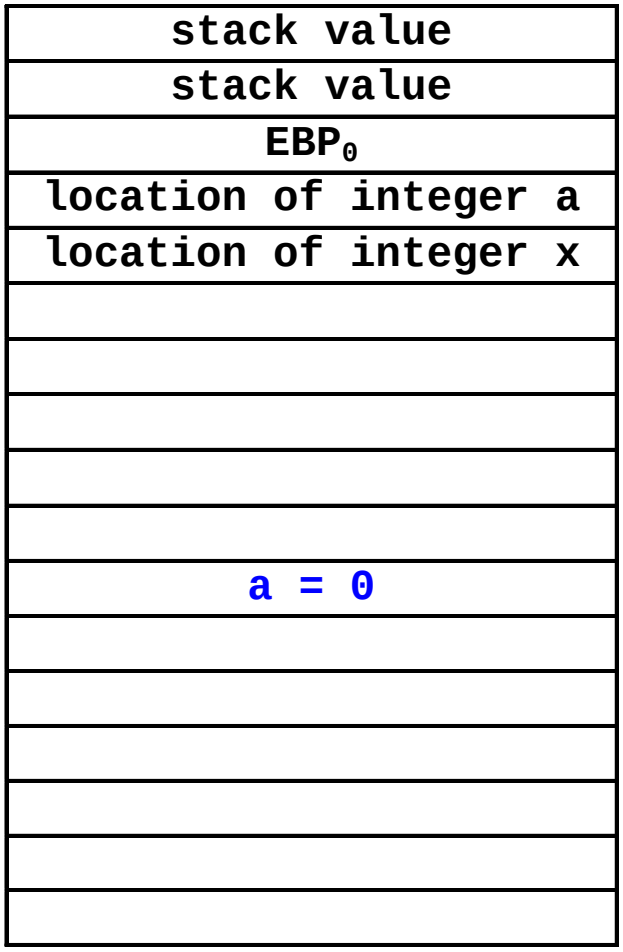
← ESP_0 (points to the old stack value)
← ESP_1 after pushing old EBP_0 (is also EBP_1)
← $ESP_2 + 1C = EBP_1 - 4$
← $ESP_2 + 18 = EBP_1 - 8$
← $ESP_2 + 14$
← $ESP_2 + 10$
← $ESP_2 + 0C$
← $ESP_2 + 08$
← $ESP_2 + 04$
← ESP_2 after adjusting ESP

Example — 7



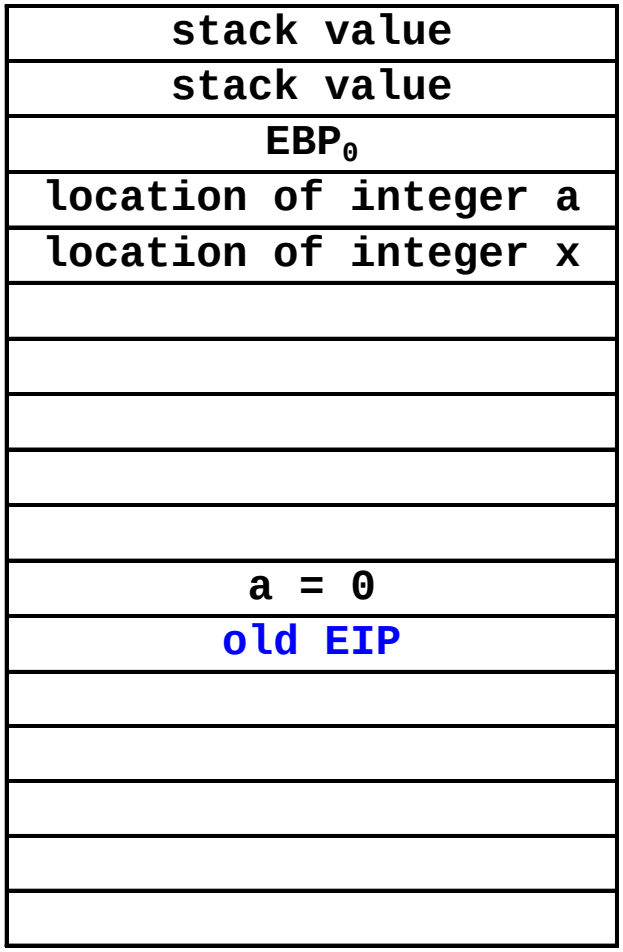
← ESP_0 (points to the old stack value)
← ESP_1 after pushing old EBP_0 (is also EBP_1)
← $ESP_2 + 1C = EBP_1 - 4$
← $ESP_2 + 18 = EBP_1 - 8$
← $ESP_2 + 14$
← $ESP_2 + 10$
← $ESP_2 + 0C$
← $ESP_2 + 08$
← $ESP_2 + 04$
← ESP_2 after adjusting ESP

Example — 8



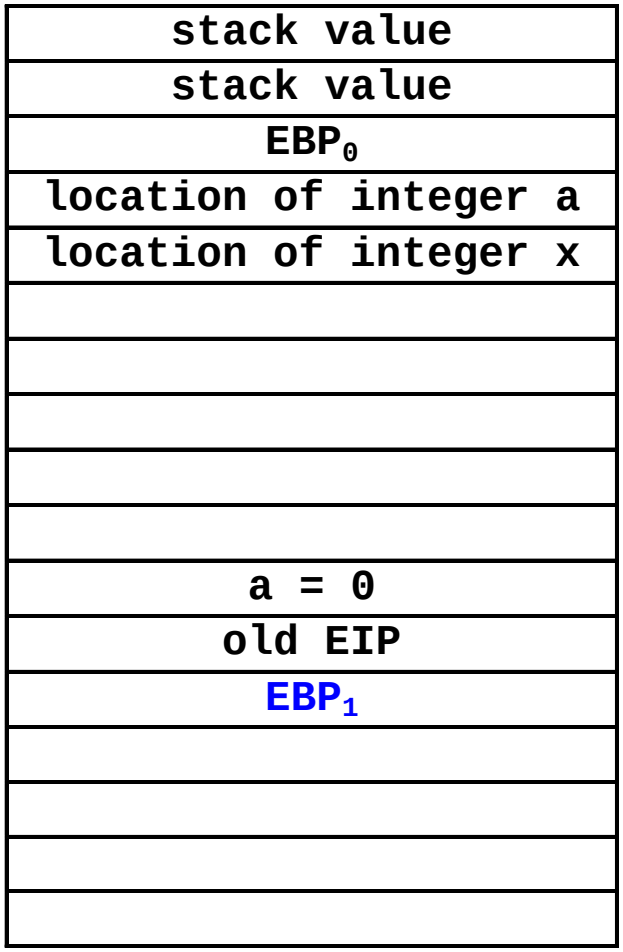
← ESP_0 (points to the old stack value)
← ESP_1 after pushing old EBP_0 (is also EBP_1)
← ESP_2 + 1C = EBP_1 – 4
← ESP_2 + 18 = EBP_1 – 8
← ESP_2 + 14
← ESP_2 + 10
← ESP_2 + 0C
← ESP_2 + 08
← ESP_2 + 04
← ESP_2 after adjusting ESP = parameter address

Example — 9



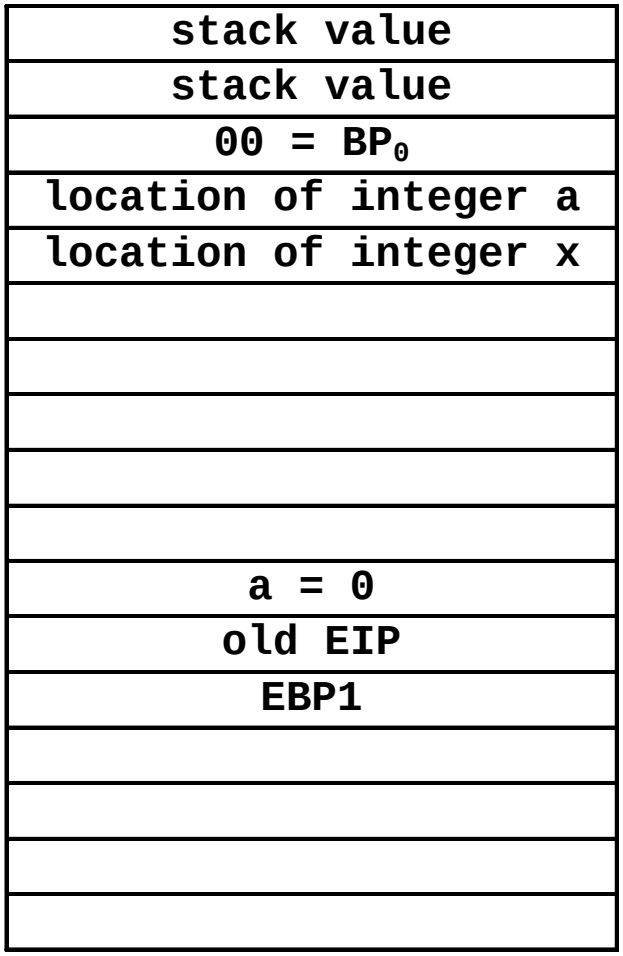
← ESP_0 (points to the old stack value)
← ESP_1 after pushing old EBP_0 (is also EBP_1)
← ESP_2 + 1C = EBP_1 – 4
← ESP_2 + 18 = EBP_1 – 8
← ESP_2 + 14
← ESP_2 + 10
← ESP_2 + 0C
← ESP_2 + 08
← ESP_2 + 04
← ESP_2 after adjusting ESP = parameter address = EBP_2 + 8
← ESP_4 after call instruction pushes old EIP

Example — 10



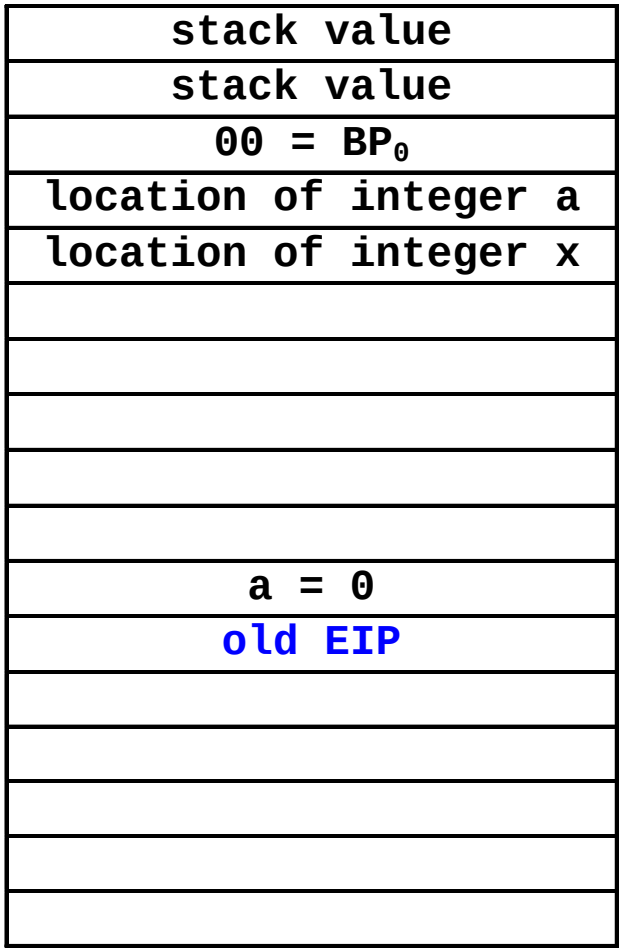
← ESP_0 (points to the old stack value)
← ESP_1 after pushing old EBP_0 (is also EBP_1)
← ESP_2 + 1C = EBP_1 – 4
← ESP_2 + 18 = EBP_1 – 8
← ESP_2 + 14
← ESP_2 + 10
← ESP_2 + 0C
← ESP_2 + 08
← ESP_2 + 04
← ESP_2 after adjusting ESP = parameter address = EBP_2 + 8
← ESP_4 after call instruction pushes old EIP
← ESP_5 after pushing EBP_1 (is also EBP_2)

Example — 11



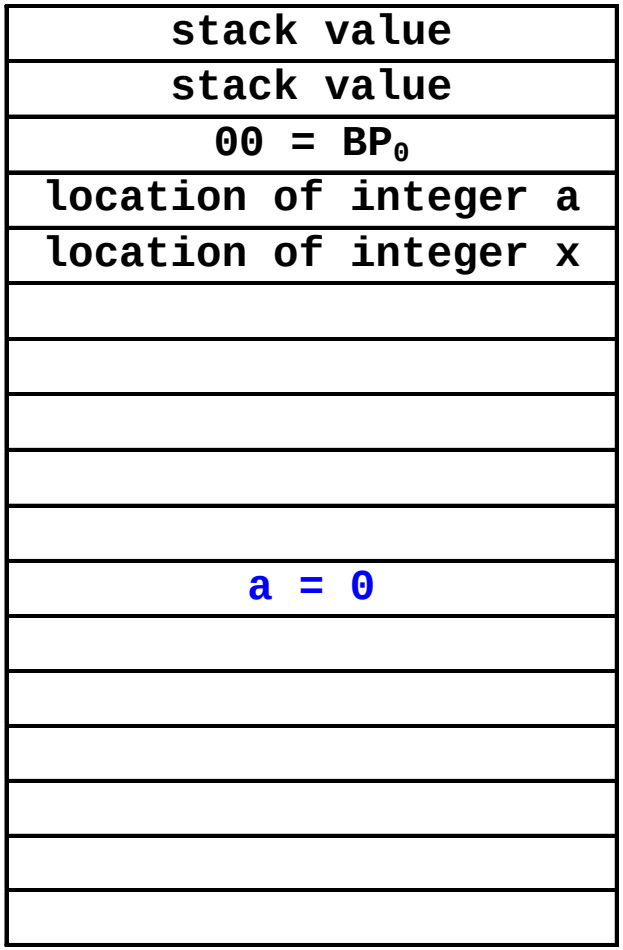
← ESP_0 (points to the old stack value)
← ESP_1 after pushing old EBP_0 (is also EBP_1)
← ESP_2 + 1C = EBP_1 – 4
← ESP_2 + 18 = EBP_1 – 8
← ESP_2 + 14
← ESP_2 + 10
← ESP_2 + 0C
← ESP_2 + 08
← ESP_2 + 04
← ESP_2 after adjusting ESP = parameter address = EBP_2 + 8
← ESP_4 after call instruction pushes old EIP
← ESP_5 after pushing EBP_1 (is also EBP_2)
← ESP_6 + 0C = EBP_2 – 4
← ESP_6 + 8
← ESP_6 + 4
← ESP_6 after adjusting ESP

Example — 12



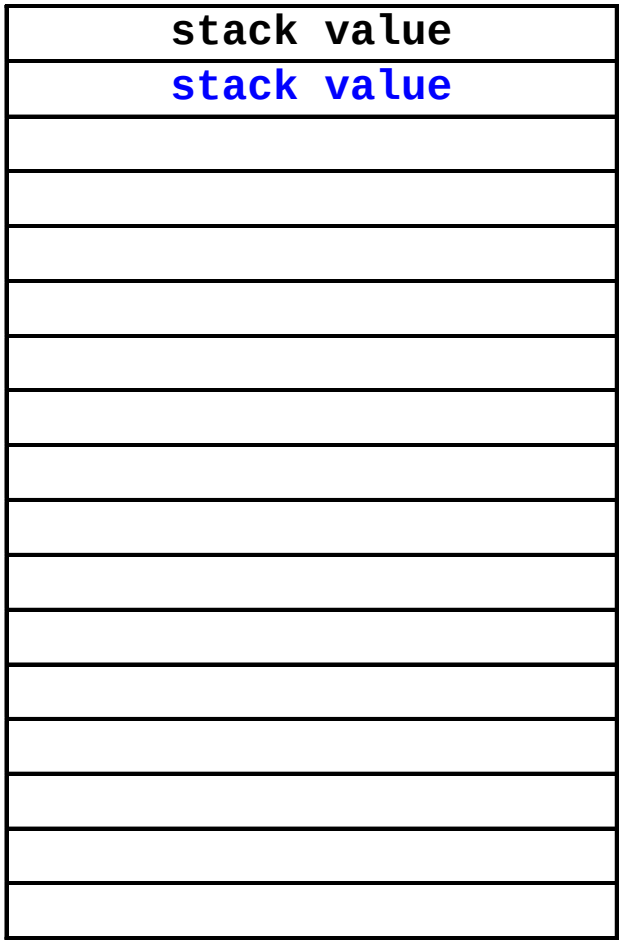
← ESP_0 (points to the old stack value)
← ESP_1 after pushing old EBP_0 (is also EBP_1)
← ESP_2 + 1C = EBP_1 – 4
← ESP_2 + 18 = EBP_1 – 8
← ESP_2 + 14
← ESP_2 + 10
← ESP_2 + 0C
← ESP_2 + 08
← ESP_2 + 04
← ESP_2 after adjusting ESP = parameter address = EBP_2 + 8
← ESP_7 after leave instruction

Example — 13



← ESP_0 (points to the old stack value)
← ESP_1 after pushing old EBP_0 (is also EBP_1)
← ESP_2 + 1C = EBP_1 – 4
← ESP_2 + 18 = EBP_1 – 8
← ESP_2 + 14
← ESP_2 + 10
← ESP_2 + 0C
← ESP_2 + 08
← ESP_2 + 04
← ESP_8 after return instruction

Example — 14



← ESP_9 after leave instruction

Factorial (3!) by Recursion

```
main()
{
        int i,j = 3;
        i = factorial(j);
}

int factorial(n)
{
        if (n == 0)
                return 1;
        else
                return  n * factorial(n-1);
}
```

Nested (Recursive) Function Calls

```
main
    copies j = 3 onto stack as parameter
    calls factorial
        factorial (instance 1)
            copies 2 onto stack as parameter
            calls factorial
                factorial (instance 2)
                    copies 1 onto stack as parameter
                    calls factorial
                        factorial (instance 3)
                            copies 0 onto stack as parameter
                            calls factorial
                                factorial (instance 4) returns 1 to factorial (instance 3)
                            factorial returns (1)*1 to factorial (instance 2)
                    factorial returns (1*1)*2 to factorial (instance 1)
            factorial returns (1*1*2)*3 = 6 to main
    main sets i = 6
```

Main Module

080483b4 <main>:

080483b4	push ebp	
080483b5	mov ebp,esp	
080483b7	and esp,byte -0x10	
080483ba	sub esp,byte +0x20	
080483bd	mov dword [esp+0x18],0x3	; j <— 3
080483c5	mov eax,[esp+0x18]	; EAX <— j
080483c9	mov [esp],eax	; copy EAX onto stack
080483cc	call 80483d7 <factorial>	; CALL factorial
080483d1	mov [esp+0x1c],eax	; i <— return value
080483d5	leave	
080483d6	ret	

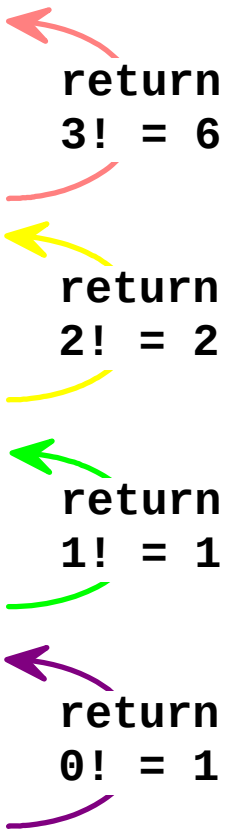
Function Module

080483d7 <factorial>:

080483d7	push ebp	
080483d8	mov ebp,esp	
080483da	sub esp,byte +0x18	
080483dd	cmp dword [ebp+0x8],byte +0x0	parameter == 0
080483e1	jnz 80483ea <factorial+0x13>	if parameter <> 0 continue
080483e3	mov eax,0x1	return 1
080483e8	jmp 80483fc <factorial+0x25>	LEAVE
080483ea	mov eax,[ebp+0x8]	EAX <— parameter
080483ed	sub eax,byte +0x1	EAX <— parameter – 1
080483f0	mov [esp],eax	Copy EAX to stack
080483f3	call 80483d7 <factorial>	new factorial instance
080483f8	imul eax,[ebp+0x8]	EAX <— return * parameter
080483fc	leave	
080483fd	ret	

Run-Time Memory Allocation

	EBP0	ESP0	Old frame
Old EBP	EBP1	ESP1	main frame
Define i,j		ESP2	
Pass 3 on stack		ESP3	
EIP 1 (return from call)		ESP4	
Main EBP	EBP2	ESP5	factorial frame 1
Pass 2 on stack		ESP6	
EIP 2 (return from call)		ESP7	
factorial frame 1 EBP	EBP3	ESP8	factorial frame 2
Pass 1 on stack		ESP9	
EIP 3 (return from call)		ESP10	
factorial frame 2 EBP	EBP4	ESP11	factorial frame 3
Pass 0 on stack		ESP12	
EIP 2 (return from call)		ESP13	
factorial frame 3 EBP	EBP5	ESP14	factorial frame 4



malloc — 1

```
main()
{
int a = 3;
int b = -1;
int *ptr;
ptr = malloc(2);
*ptr = 8;
int c = 0;
while (a < 10){
     b = a * 2 + *ptr;
     c = b + 7;
     a++;
     }
    free(ptr);
    ptr = NULL;
}
```

malloc — 2

Initialize variables

`push ebp`	
`mov ebp,esp`	
`and esp,byte -0x10`	
`sub esp,byte +0x20`	
`mov dword [esp+0x1c],0x3`	`a <– 3`
`mov dword [esp+0x18],0xffffffff`	`b <– -1`
`mov dword [esp],0x4`	`copy 4 onto stack as parameter`
`call 8048350 <malloc@plt>`	`CALL malloc(4) from library`
`mov [esp+0x14],eax`	`ptr <– returned pointer`
`mov eax,[esp+0x14]`	`EAX <– ptr`
`mov dword [eax],0x8`	`*ptr <– 8`
`mov dword [esp+0x10],0x0`	`c <– 0`

malloc — 3

0804844f	jmp 8048475 <main+0x61>	begin loop
08048451	mov eax,[esp+0x1c]	EAX <— a
08048455	lea edx,[eax+eax]	EAX <— 2 * EAX = 2 * a
08048458	mov eax,[esp+0x14]	EAX <— ptr
0804845c	mov eax,[eax]	EAX <— *ptr
0804845e	lea eax,[edx+eax]	EAX <— EAX + *ptr = 2 * a + *ptr
08048461	mov [esp+0x18],eax	b <— EAX = 2 * a + *ptr
08048465	mov eax,[esp+0x18]	EAX <— b
08048469	add eax,byte +0x7	EAX <— EAX + 7 = b + 7
0804846c	mov [esp+0x10],eax	c <— EAX = b + 7
08048470	add dword [esp+0x1c],byte +0x1	a++
08048475	cmp dword [esp+0x1c],byte +0x9	CMP a, 9
0804847a	jng 8048451 <main+0x3d>	if a <= 10 loop
0804847c	mov eax,[esp+0x14]	EAX <— ptr
08048480	mov [esp],eax	copy ptr onto stack as parameter
08048483	call 8048340 <free@plt>	CALL free(ptr) from library
08048488	mov dword [esp+0x14],0x0	ptr <— NULL
08048490	leave	
08048491	ret	