CALL and RET

BASIC ASSEMBLY

Assembly language programming
By xorpd

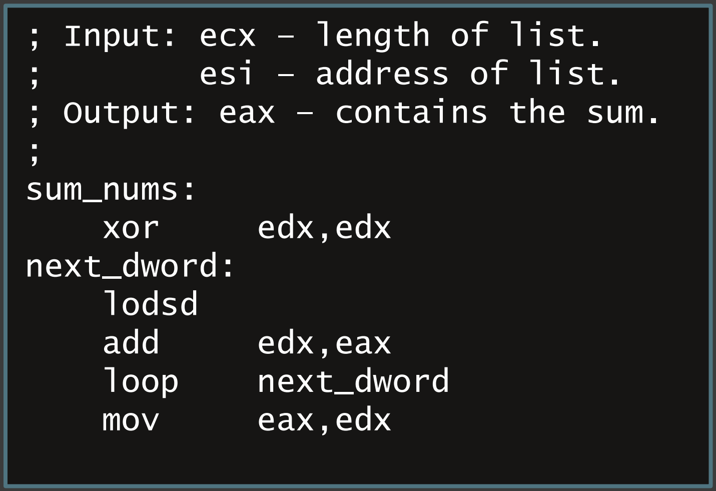
xorpd.net

Objectives

- We will study the CALL and RET instructions.
- We will see examples of using CALL and RET.
- We will understand the stack’s meaning with respect to function calls.

Example

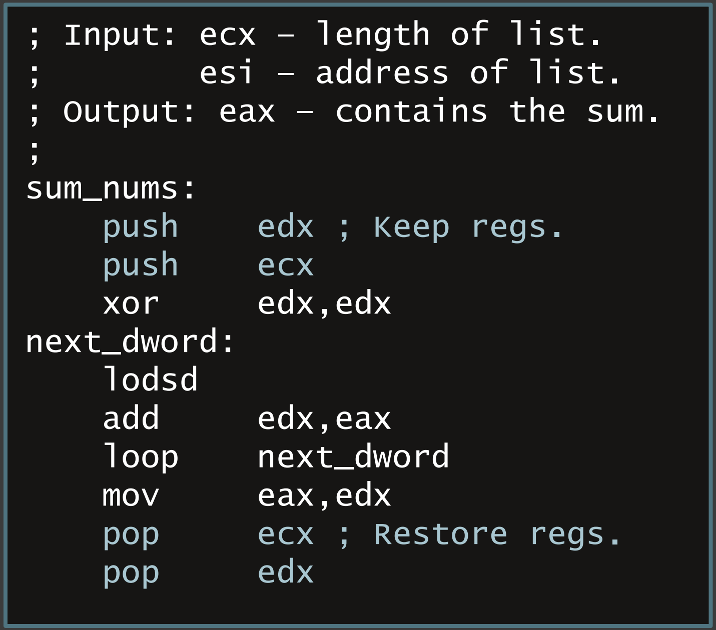
- A function that calculates the sum of a list of numbers (dwords):



```
; Input: ecx – length of list.
;        esi – address of list.
; Output: eax – contains the sum.
;
sum_nums:
    xor     edx,edx
next_dword:
    lodsd
    add     edx,eax
    loop    next_dword
    mov     eax,edx
```

Example (Cont.)

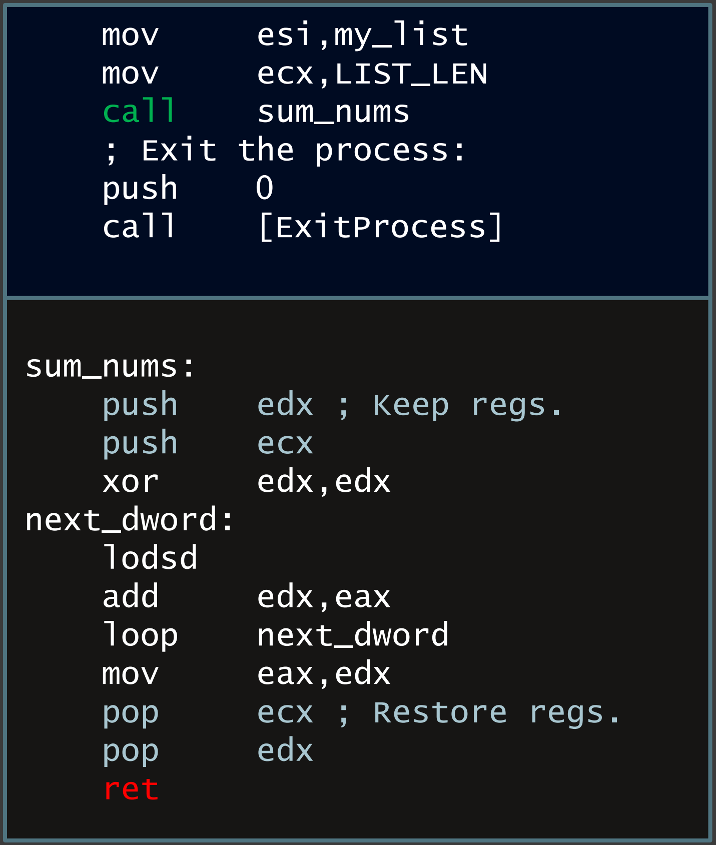
- A function that calculates the sum of a list of numbers (dwords):



```
; Input: ecx - length of list.
;        esi - address of list.
; Output: eax - contains the sum.
;
sum_nums:
    push    edx ; Keep regs.
    push    ecx
    xor     edx,edx
next_dword:
    lodsd
    add     edx,eax
    loop    next_dword
    mov     eax,edx
    pop     ecx ; Restore regs.
    pop     edx
```

Example (Cont.)

- Using sum_nums:

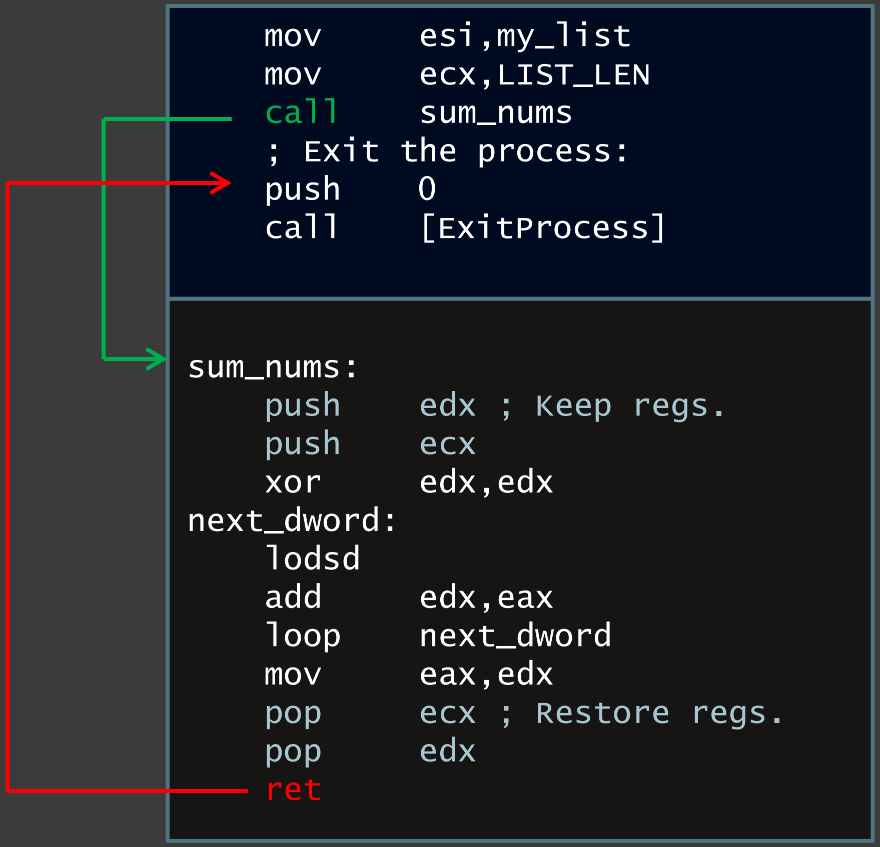


```
    mov     esi,my_list
    mov     ecx,LIST_LEN
    call    sum_nums
    ; Exit the process:
    push    0
    call    [ExitProcess]
```

```
sum_nums:
    push    edx ; Keep regs.
    push    ecx
    xor     edx,edx
next_dword:
    lodsd
    add     edx,eax
    loop    next_dword
    mov     eax,edx
    pop     ecx ; Restore regs.
    pop     edx
    ret
```

Example (Cont.)

- Using sum_nums:

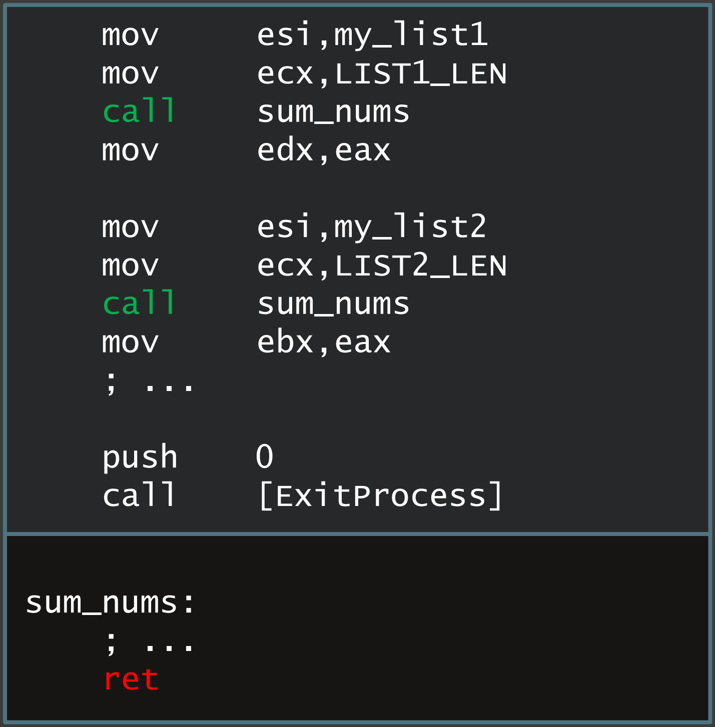


```
    mov     esi,my_list
    mov     ecx,LIST_LEN
    call    sum_nums
    ; Exit the process:
    push    0
    call    [ExitProcess]
```

```
sum_nums:
    push    edx ; Keep regs.
    push    ecx
    xor     edx,edx
next_dword:
    lodsd
    add     edx,eax
    loop    next_dword
    mov     eax,edx
    pop     ecx ; Restore regs.
    pop     edx
    ret
```

Example (Cont.)

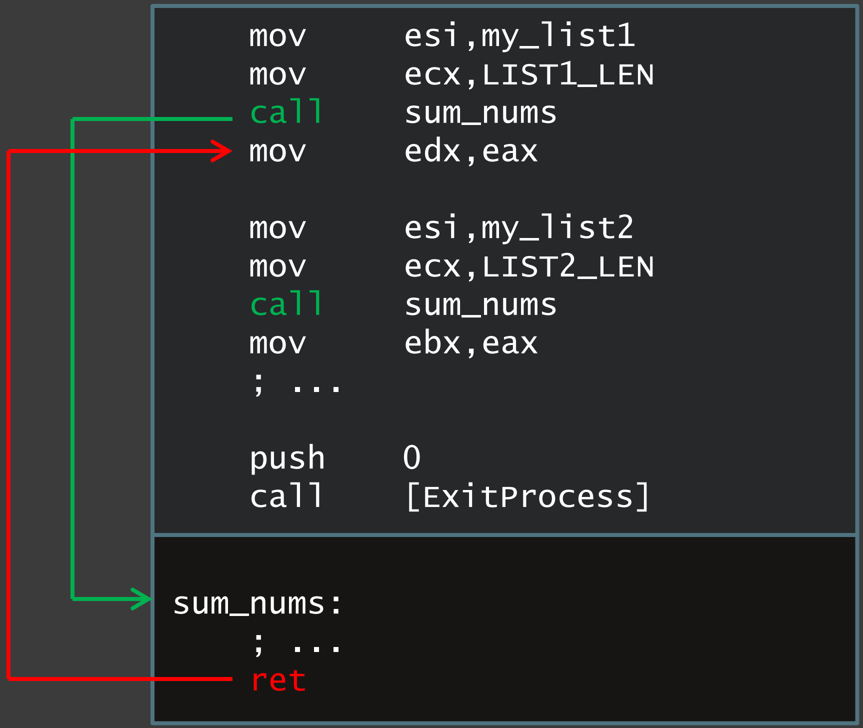
- Using sum_nums for two different lists:



```
    mov     esi,my_list1
    mov     ecx,LIST1_LEN
    call    sum_nums
    mov     edx,eax

    mov     esi,my_list2
    mov     ecx,LIST2_LEN
    call    sum_nums
    mov     ebx,eax
    ; ...

    push    0
    call    [ExitProcess]

sum_nums:
    ; ...
    ret
```

Example (Cont.)

- Using sum_nums for two different lists:

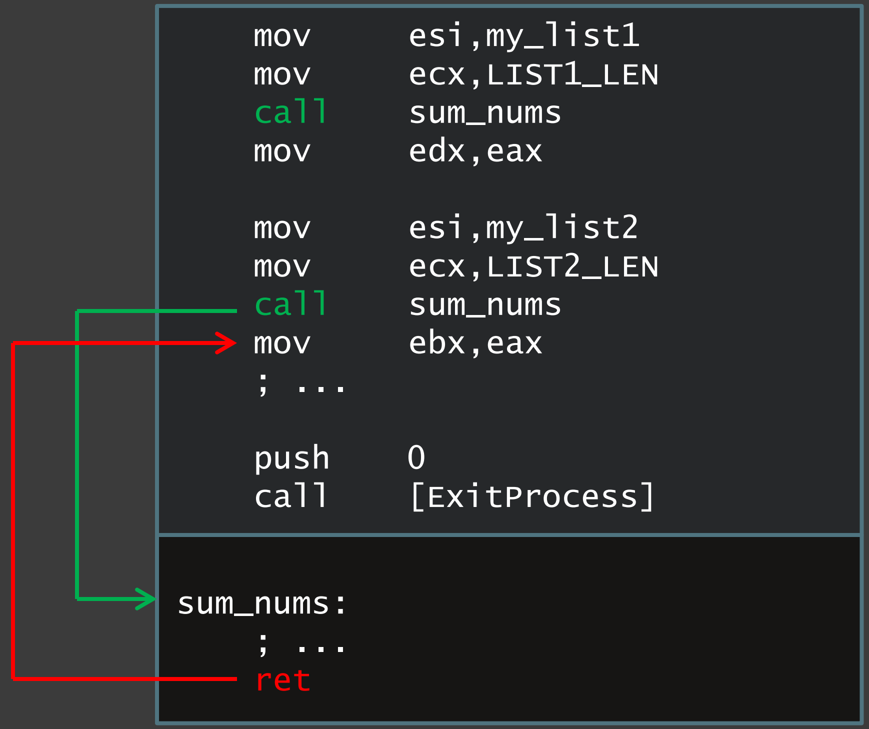


```
    mov     esi,my_list1
    mov     ecx,LIST1_LEN
    call    sum_nums
    mov     edx,eax

    mov     esi,my_list2
    mov     ecx,LIST2_LEN
    call    sum_nums
    mov     ebx,eax
    ; ...

    push    0
    call    [ExitProcess]

sum_nums:
    ; ...
    ret
```

First call to sum_nums

Example (Cont.)

- Using sum_nums for two different lists:

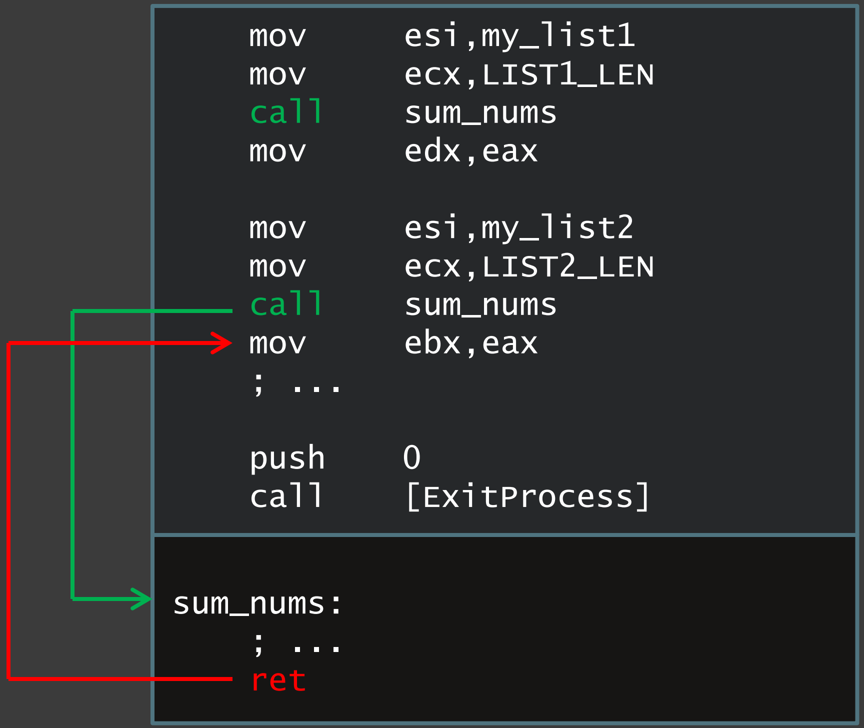


```
    mov     esi,my_list1
    mov     ecx,LIST1_LEN
    call    sum_nums
    mov     edx,eax

    mov     esi,my_list2
    mov     ecx,LIST2_LEN
    call    sum_nums
    mov     ebx,eax
    ; ...

    push    0
    call    [ExitProcess]

sum_nums:
    ; ...
    ret
```

Second call to sum_nums

Example (Cont.)

- Using sum_nums for two different lists:



```
    mov     esi,my_list1
    mov     ecx,LIST1_LEN
    call    sum_nums
    mov     edx,eax

    mov     esi,my_list2
    mov     ecx,LIST2_LEN
    call    sum_nums
    mov     ebx,eax
    ; ...

    push    0
    call    [ExitProcess]

sum_nums:
    ; ...
    ret
```

Second call to sum_nums

- How can **ret** know where to return?

CALL and RET

- CALL arg
  - Call procedure
    - Push the address of the next instruction to the stack.
    - $eip \leftarrow arg$ (Jump to arg).

- RET
  - Return from procedure
    - Pop a dword x from the stack.
    - $eip \leftarrow x$ (Jump to x).

- The return address is kept on the stack!

Example – Simple calling

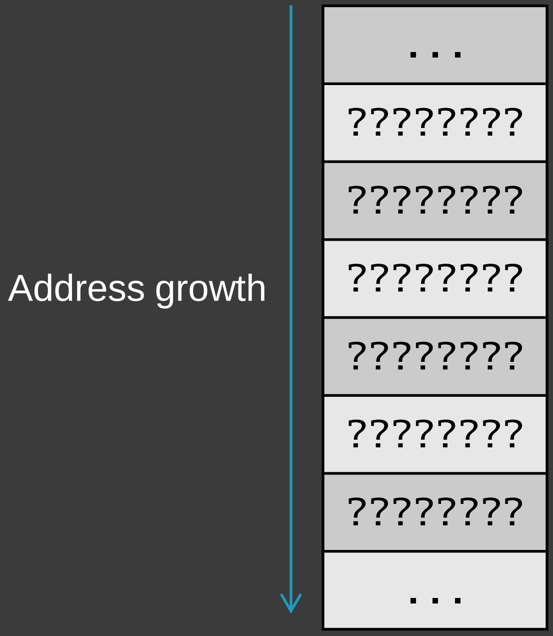
- Simple calling and returning:

```
    call    my_func
    ...

my_func:
    ...
    ret
```

Example – Simple calling

- Simple calling and returning:

```
    call    my_func
    ...
my_func:
    ...
    ret
```

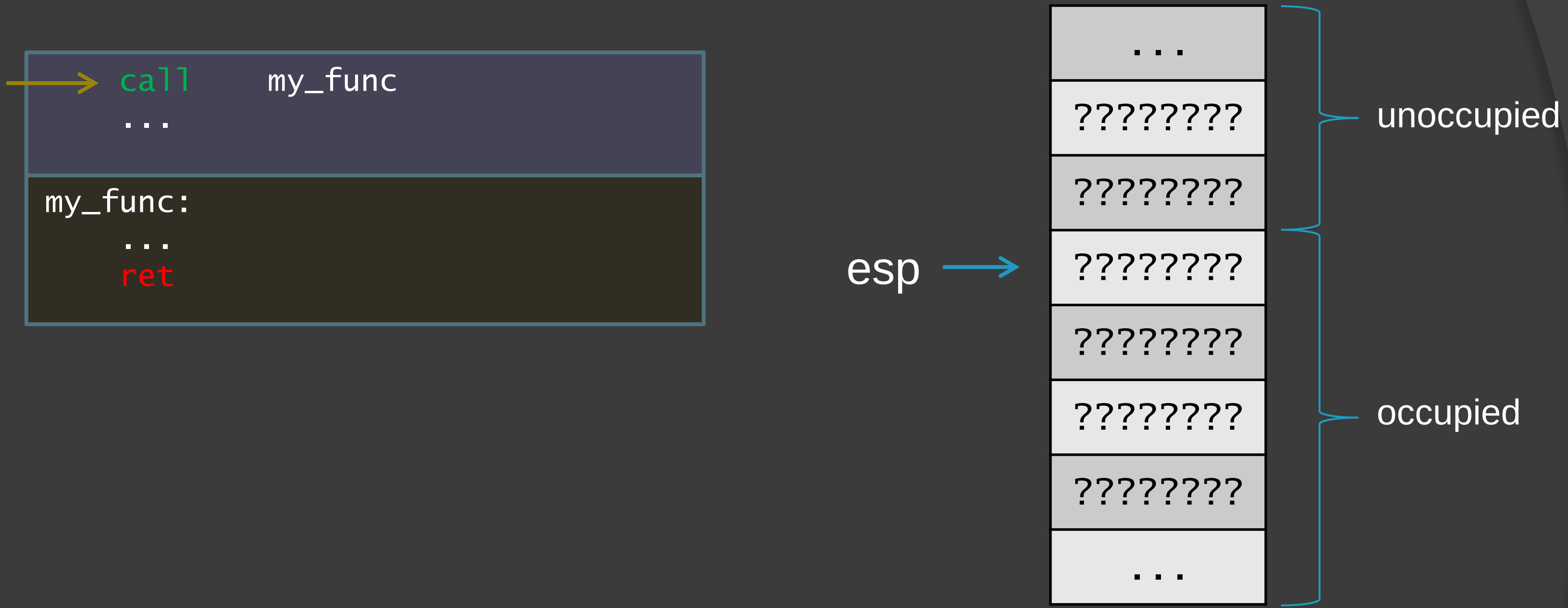


Address growth

...
????????
????????
????????
????????
????????
????????
...

Example – Simple calling

- Simple calling and returning:



```
    call     my_func
    ...
my_func:
    ...
    ret
```

esp

...

????????

????????

????????

????????

????????

????????

...

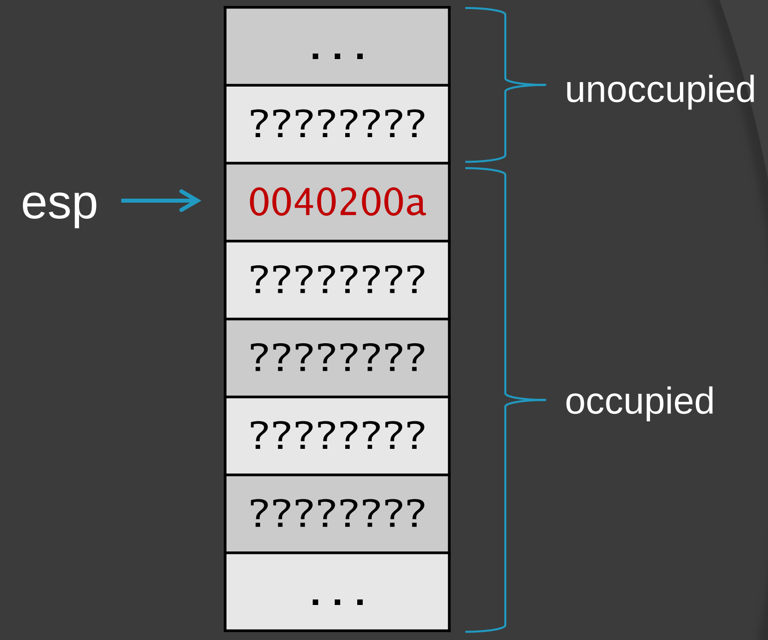
unoccupied

occupied

Example – Simple calling

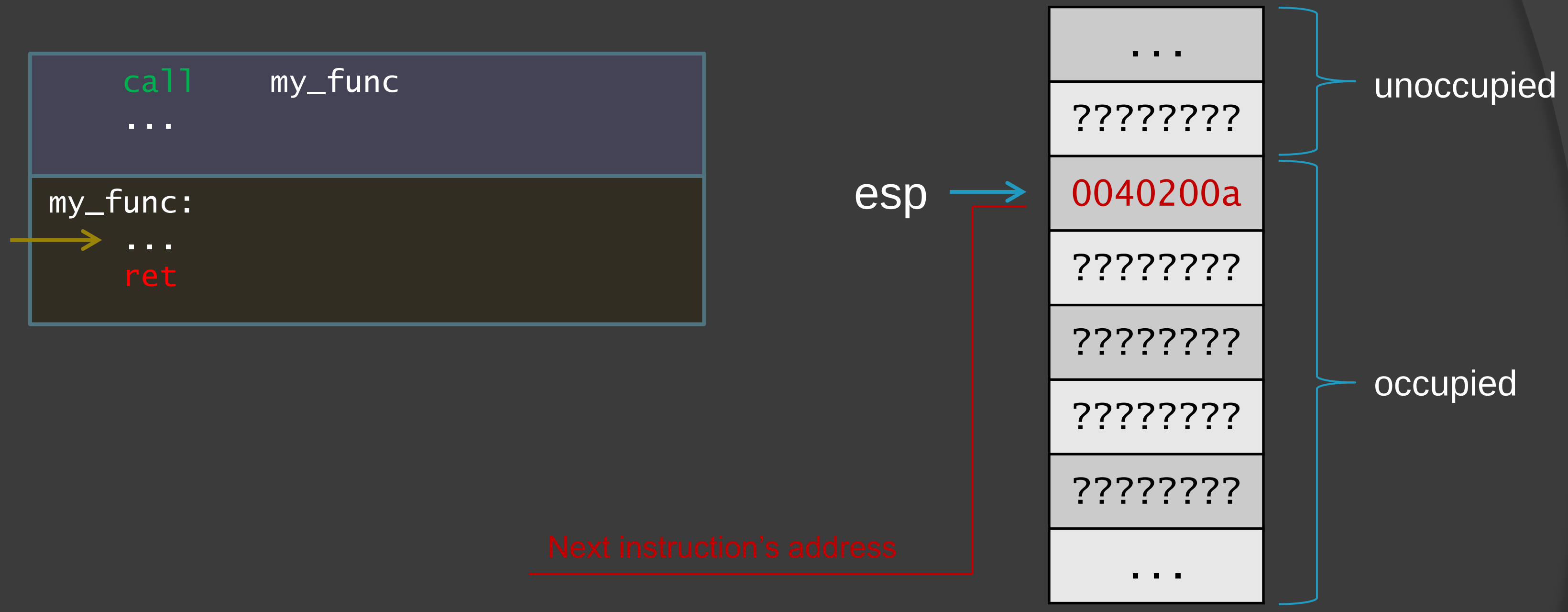
- Simple calling and returning:

```
    call     my_func
    ...

my_func:
    ...
    ret
```



Example – Simple calling

- Simple calling and returning:



```
    call     my_func
    ...
my_func:
    ...
    ret
```

esp

...
????????
0040200a
????????
????????
????????
????????
...

unoccupied

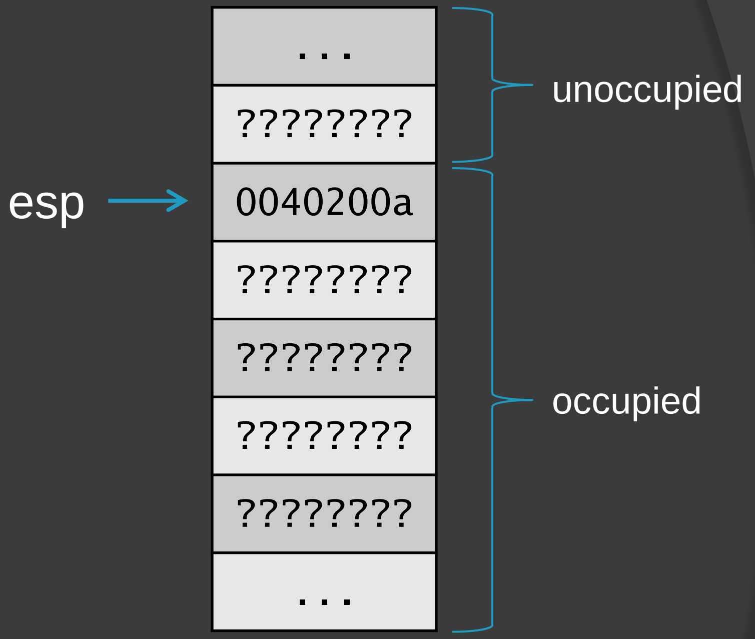
occupied

Next instruction's address

Example – Simple calling

- Simple calling and returning:

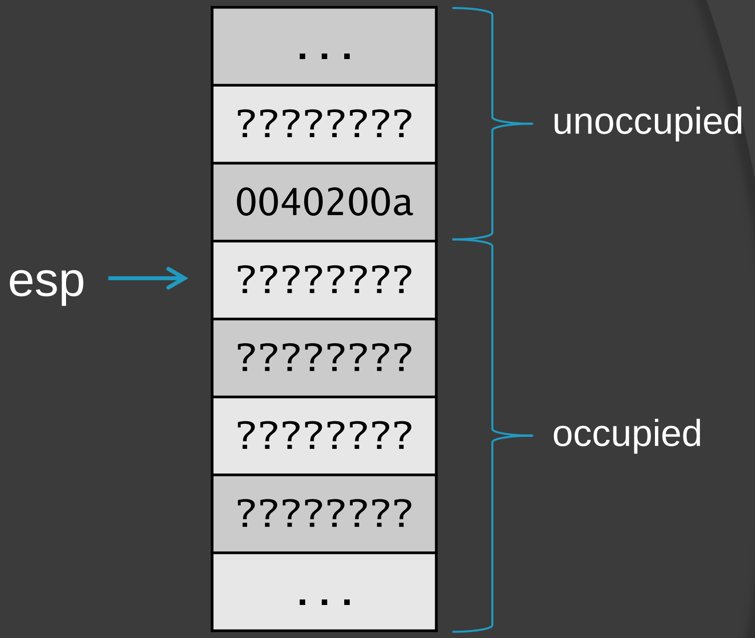
```
    call    my_func
    ...
my_func:
    ...
    ret
```



Example – Simple calling

- Simple calling and returning:

```
    call    my_func
    ...
my_func:
    ...
    ret
```



...

????????

0040200a

esp → ????????

????????

????????

????????

...

unoccupied

occupied

Example – Nested calling

- Nested calling:

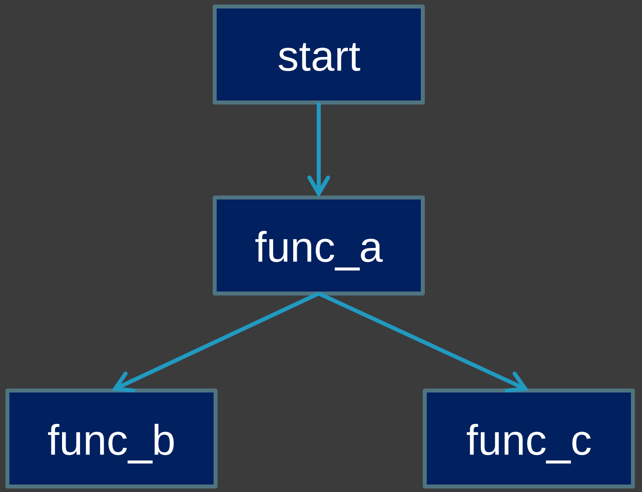
```
     call    func_a
     ...

func_a:
     call    func_b
     call    func_c
     ret

func_b:
     ...
     ret

func_c:
     ...
     ret
```

Example – Nested calling

- Nested calling:

```
    call    func_a
    ...

func_a:
    call    func_b
    call    func_c
    ret

func_b:
    ...
    ret

func_c:
    ...
    ret
```

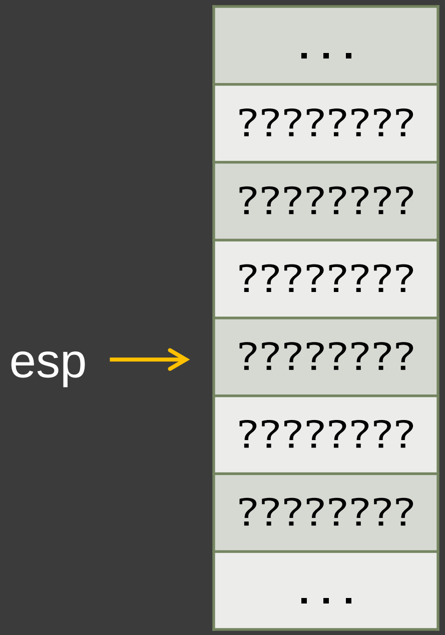
Call graph



Example – Nested calling

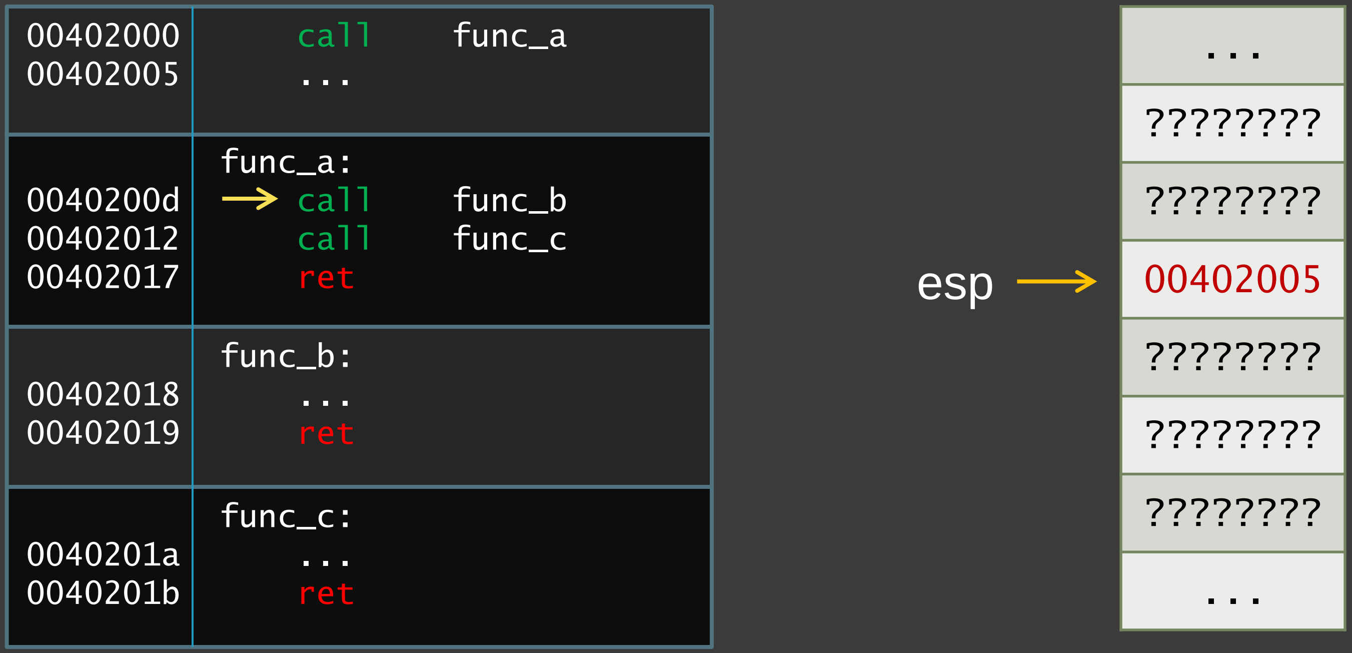
- Nested calling:

```
00402000  → call    func_a
00402005      ...

            func_a:
0040200d      call    func_b
00402012      call    func_c
00402017      ret

            func_b:
00402018      ...
00402019      ret

            func_c:
0040201a      ...
0040201b      ret
```



Example – Nested calling

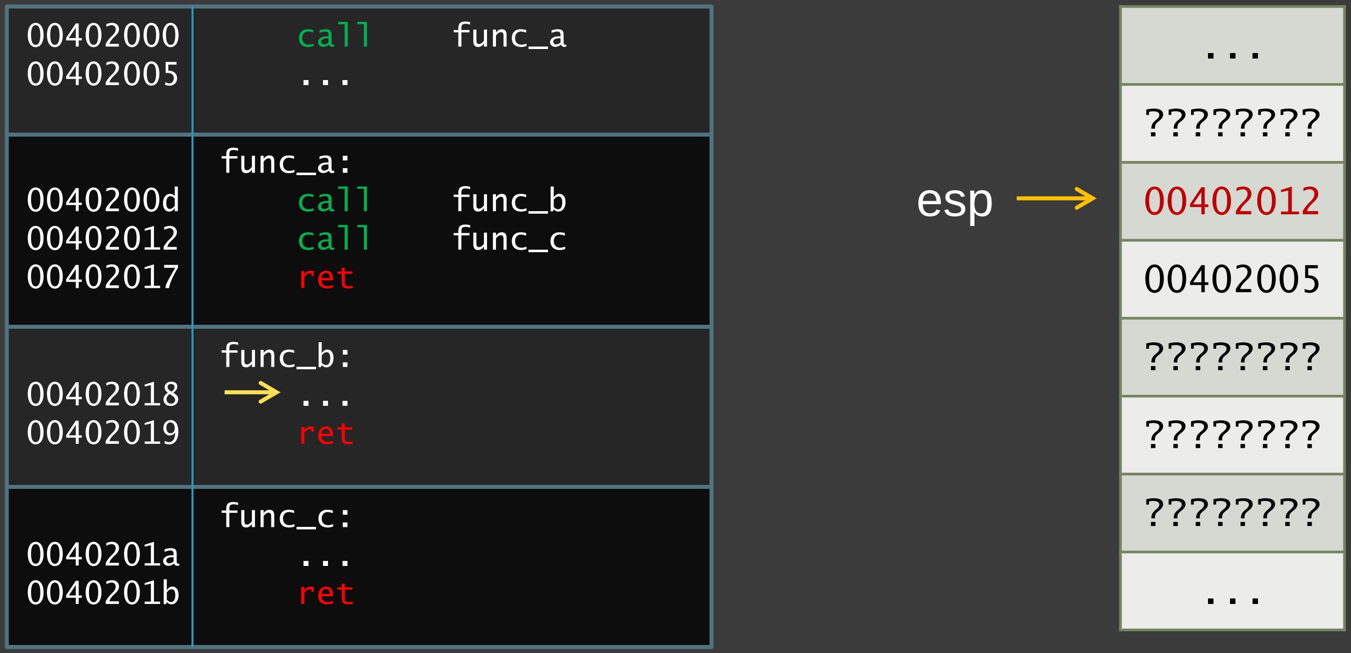
- Nested calling:



```
00402000        call     func_a
00402005        ...

              func_a:
0040200d    →   call     func_b
00402012        call     func_c
00402017        ret

              func_b:
00402018        ...
00402019        ret

              func_c:
0040201a        ...
0040201b        ret
```

Example – Nested calling

- Nested calling:

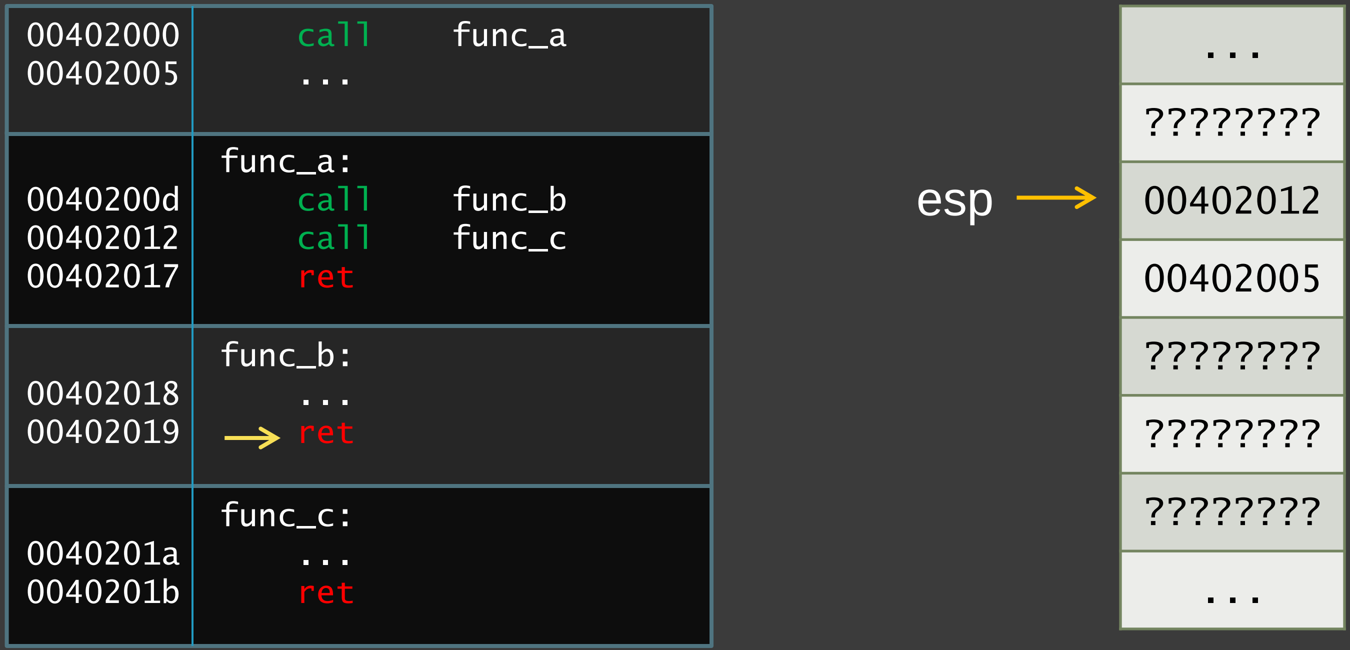


```
00402000        call     func_a
00402005        ...

            func_a:
0040200d        call     func_b
00402012        call     func_c
00402017        ret

            func_b:
00402018    →   ...
00402019        ret

            func_c:
0040201a        ...
0040201b        ret
```

```
...
????????
00402012
00402005
????????
????????
????????
...
```

Example – Nested calling

- Nested calling:



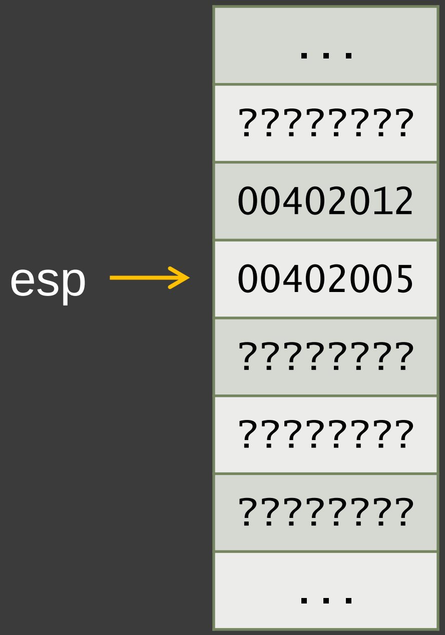
```
00402000        call     func_a
00402005        ...

            func_a:
0040200d        call     func_b
00402012        call     func_c
00402017        ret

            func_b:
00402018        ...
00402019   →    ret

            func_c:
0040201a        ...
0040201b        ret
```

```
           ...
         ????????
esp →    00402012
         00402005
         ????????
         ????????
         ????????
           ...
```

Example – Nested calling

- Nested calling:

```
00402000        call      func_a
00402005        ...

             func_a:
0040200d        call      func_b
00402012   →    call      func_c
00402017        ret

             func_b:
00402018        ...
00402019        ret

             func_c:
0040201a        ...
0040201b        ret
```



...

????????

00402012

esp → 00402005

????????

????????

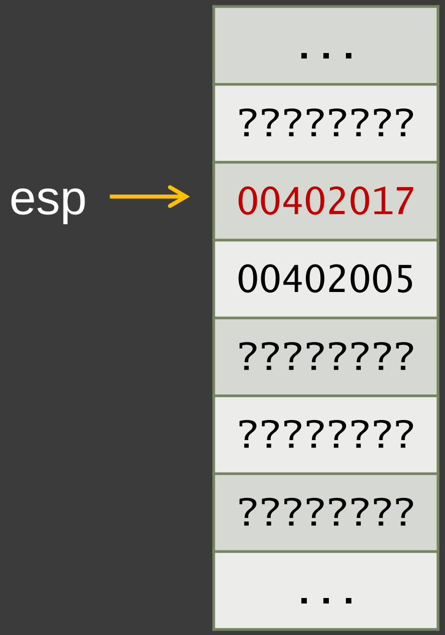
????????

...

Example – Nested calling

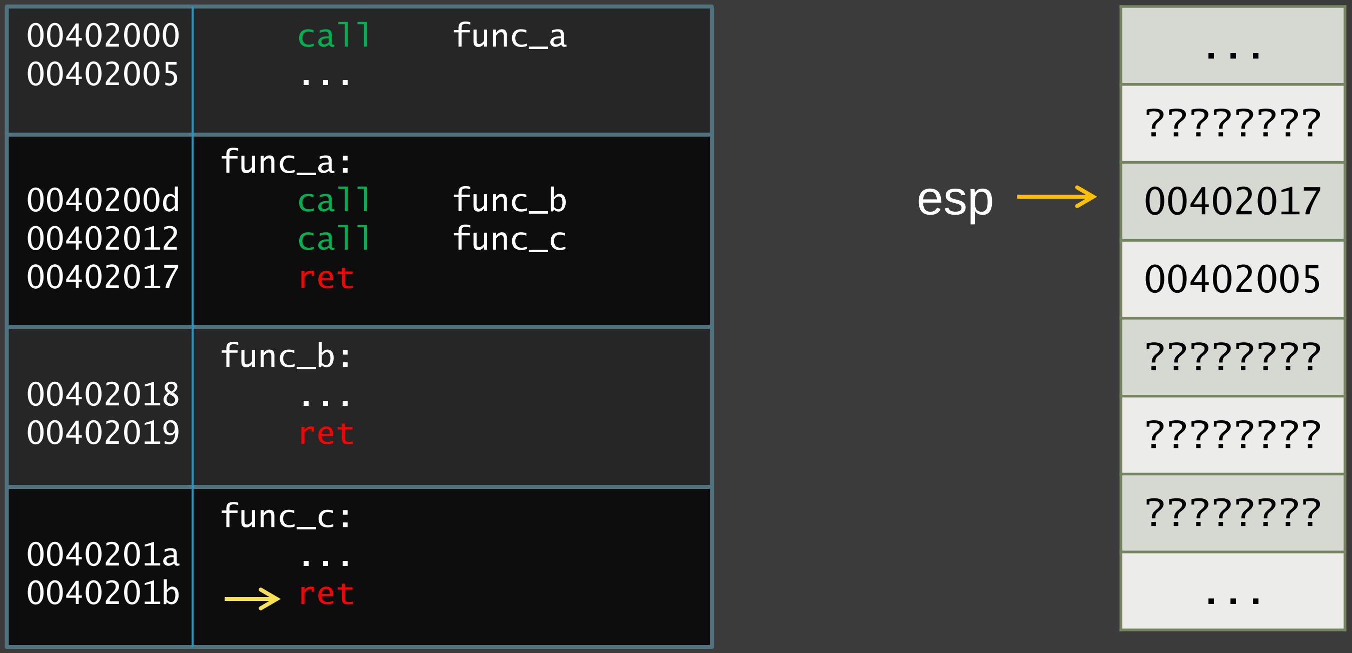
- Nested calling:

```
00402000        call      func_a
00402005        ...

              func_a:
0040200d        call      func_b
00402012        call      func_c
00402017        ret

              func_b:
00402018        ...
00402019        ret

              func_c:
0040201a   →    ...
0040201b        ret
```



Example – Nested calling

- Nested calling:

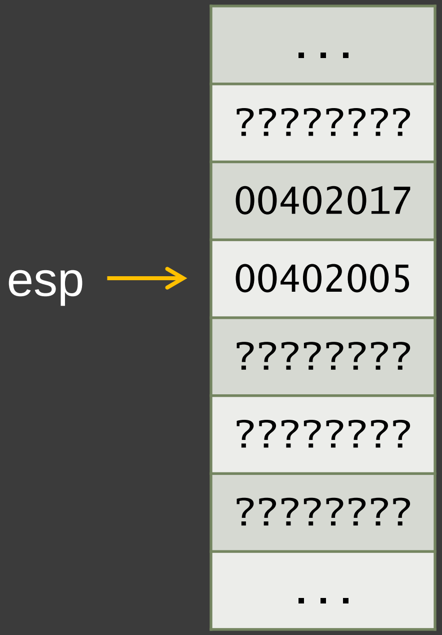


```
00402000      call     func_a
00402005      ...

            func_a:
0040200d      call     func_b
00402012      call     func_c
00402017      ret

            func_b:
00402018      ...
00402019      ret

            func_c:
0040201a      ...
0040201b  →   ret
```

Example – Nested calling

- Nested calling:

```
00402000        call     func_a
00402005        ...

             func_a:
0040200d        call     func_b
00402012        call     func_c
00402017     →  ret

             func_b:
00402018        ...
00402019        ret

             func_c:
0040201a        ...
0040201b        ret
```

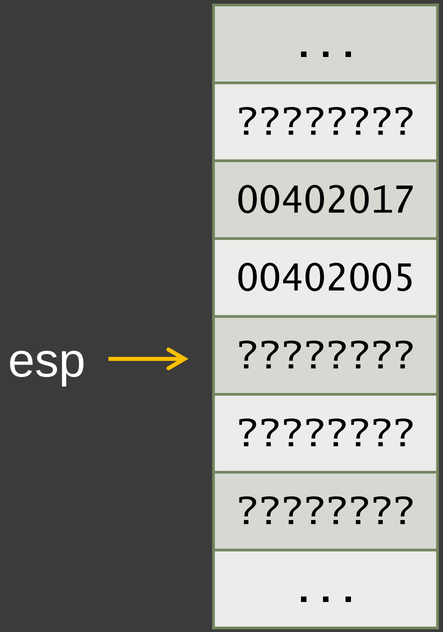


...
????????
00402017
esp → 00402005
????????
????????
????????
...

Example – Nested calling

- Nested calling:

```
00402000          call     func_a
00402005    →     ...

              func_a:
0040200d          call     func_b
00402012          call     func_c
00402017          ret

              func_b:
00402018          ...
00402019          ret

              func_c:
0040201a          ...
0040201b          ret
```



...
????????
00402017
00402005
????????
????????
????????
...

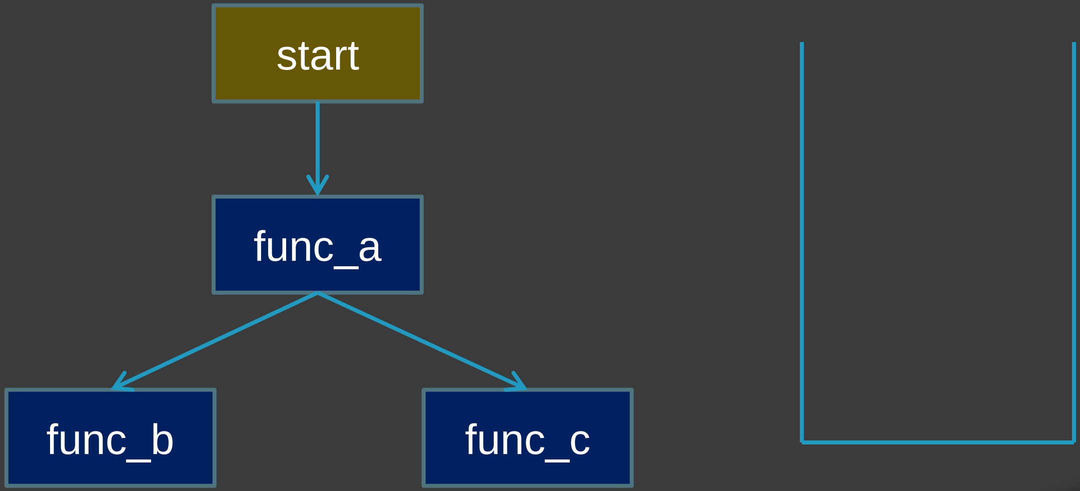
Observations

- At any point in the program, we are inside some function.
- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.

Observations

- At any point in the program, we are inside some function.

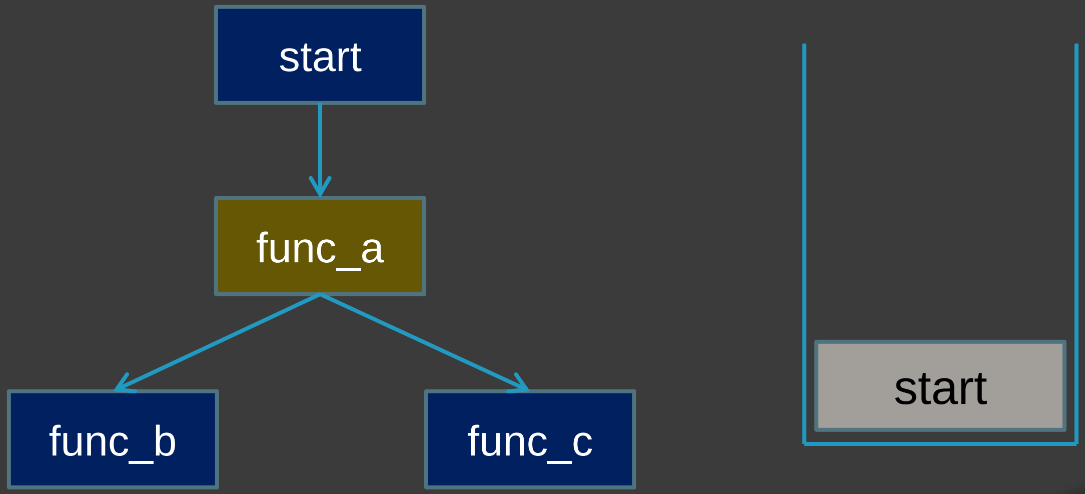
- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.



Observations

- At any point in the program, we are inside some function.

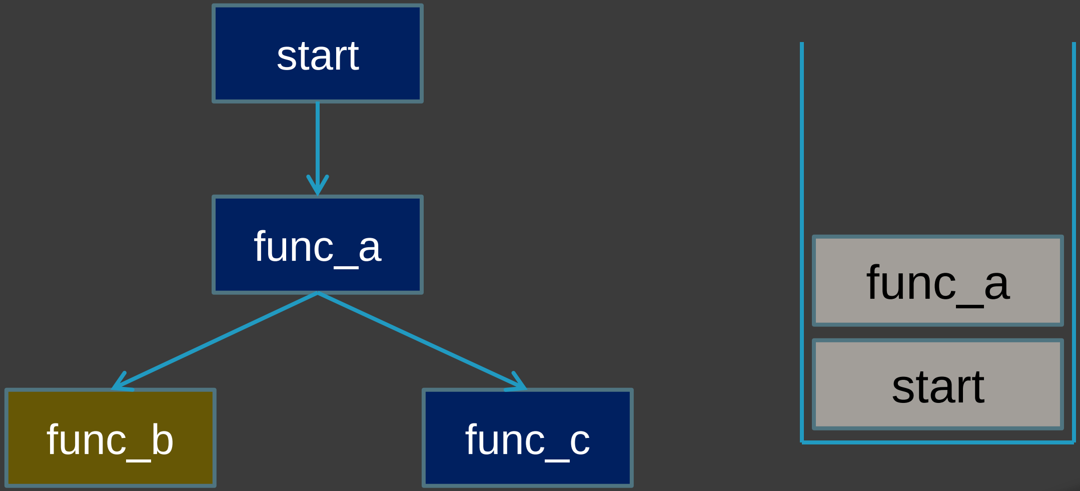
- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.



Observations

- At any point in the program, we are inside some function.

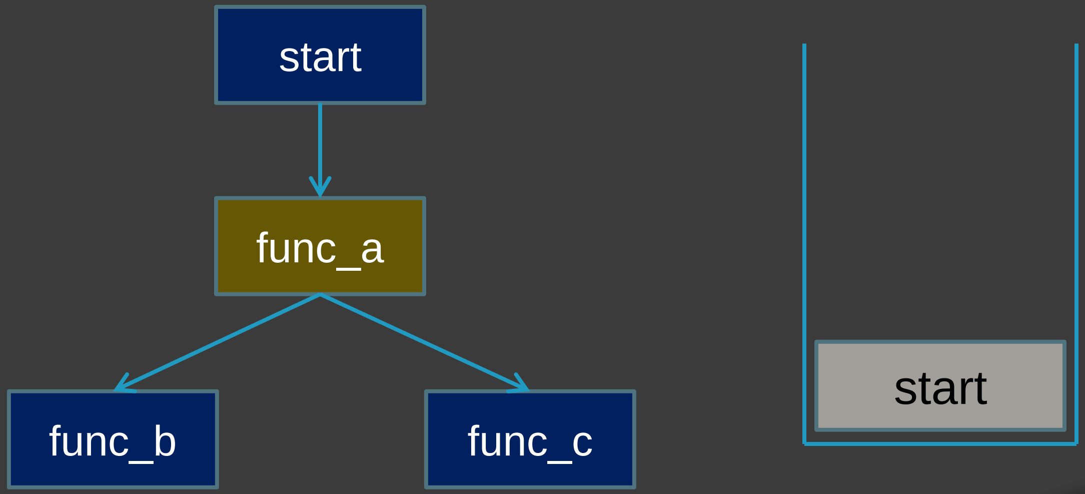
- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.



Observations

- At any point in the program, we are inside some function.

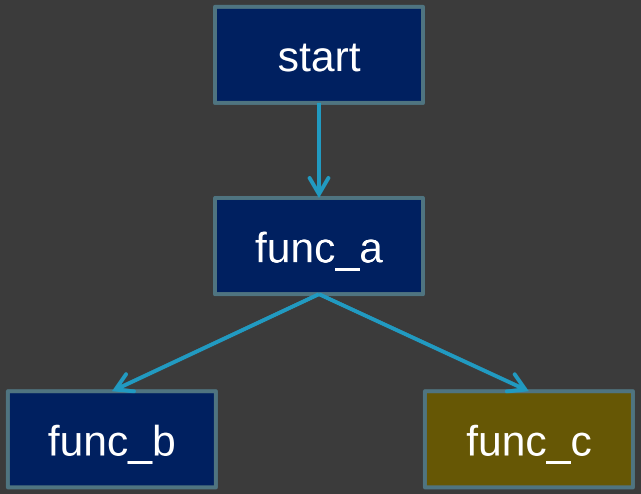
- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.

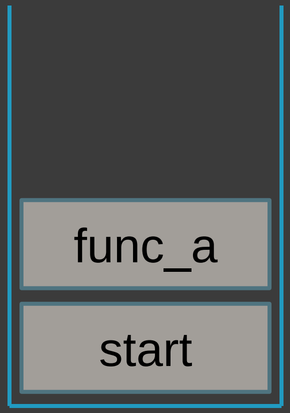


Observations

- At any point in the program, we are inside some function.

- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.

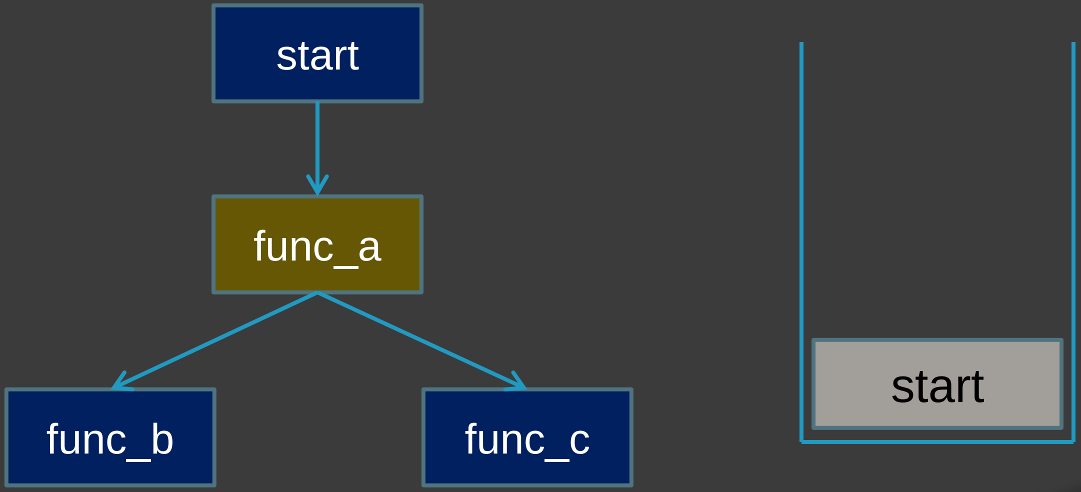




Observations

- At any point in the program, we are inside some function.

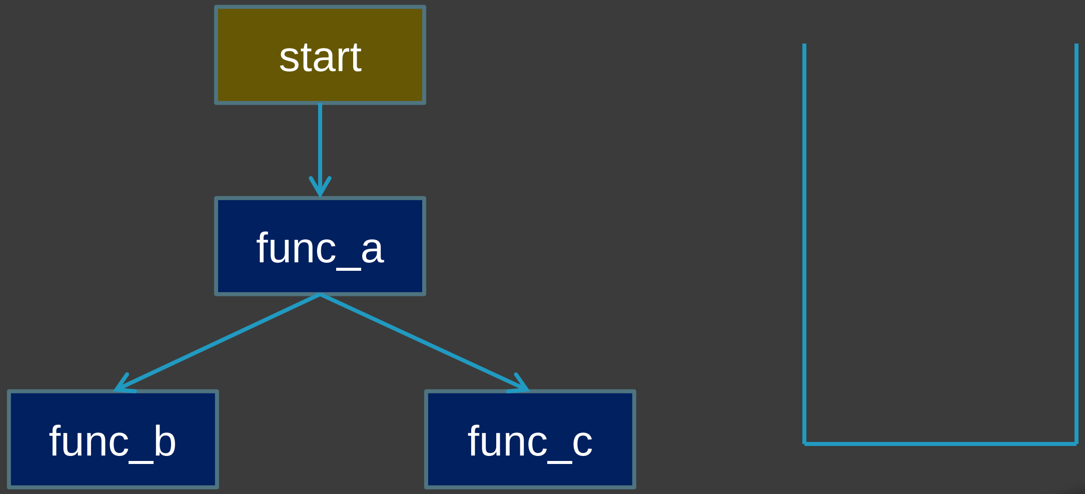
- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.



Observations

- At any point in the program, we are inside some function.

- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.



Observations

- At any point in the program, we are inside some function.

- The stack keeps the path to the current function.
  - We can use that information to return.
  - The stack is our tool to find our way in the calls graph.

- ESP before the CALL equals to the ESP after the RET.
  - Even if there are many calls in the middle.
  - Matching CALLS and RETS leave the stack **balanced**.

Stairs illustration

```
    call    func_a
    ...
```

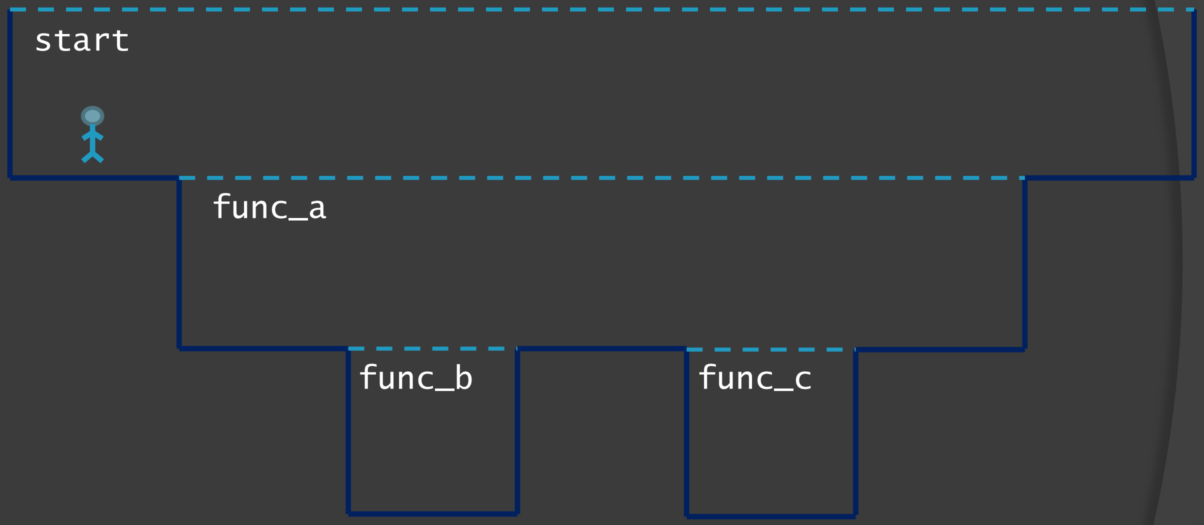
```
func_a:
    call    func_b
    call    func_c
    ret
```

```
func_b:
    ...
    ret
```

```
func_c:
    ...
    ret
```

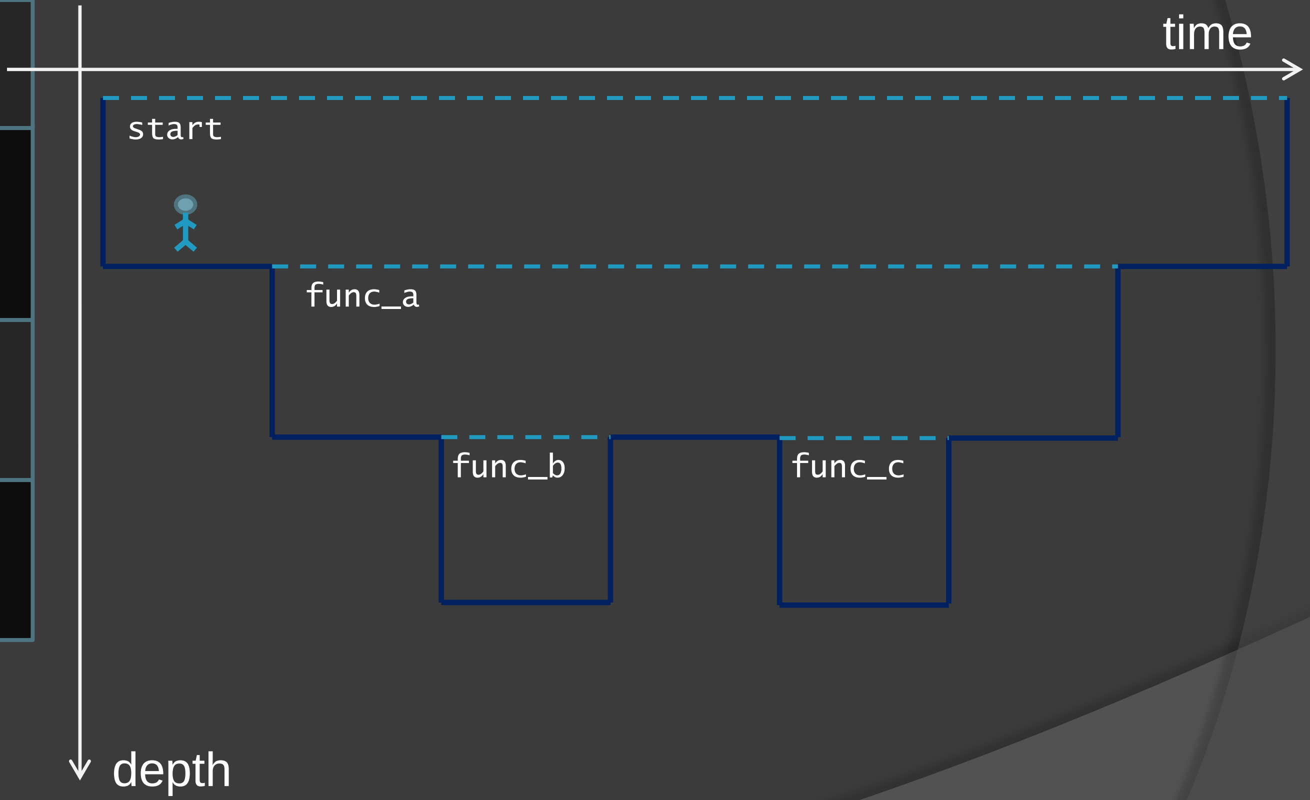
Stairs illustration

```
→  call     func_a
   ...

func_a:
   call     func_b
   call     func_c
   ret

func_b:
   ...
   ret

func_c:
   ...
   ret
```



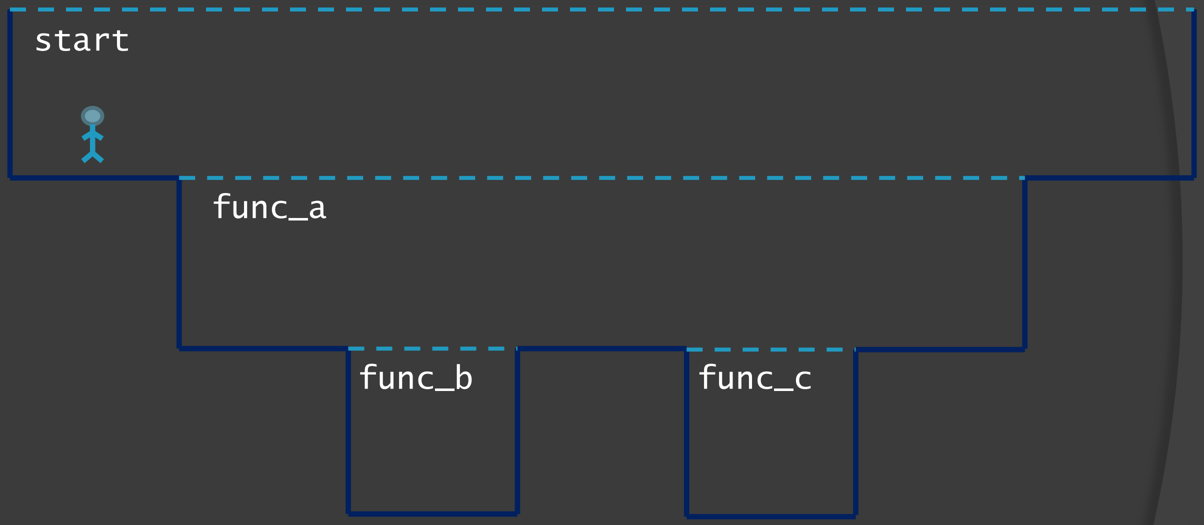
Stairs illustration

```
→   call     func_a
    ...

func_a:
    call     func_b
    call     func_c
    ret

func_b:
    ...
    ret

func_c:
    ...
    ret
```



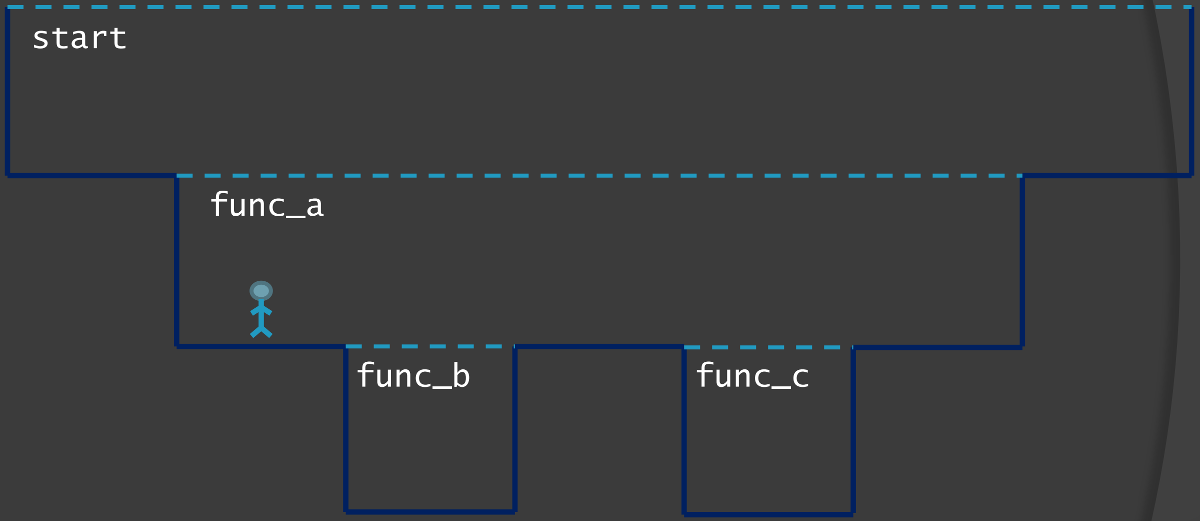
Stairs illustration

```
→  call      func_a
   ...

func_a:
   call      func_b
   call      func_c
   ret

func_b:
   ...
   ret

func_c:
   ...
   ret
```



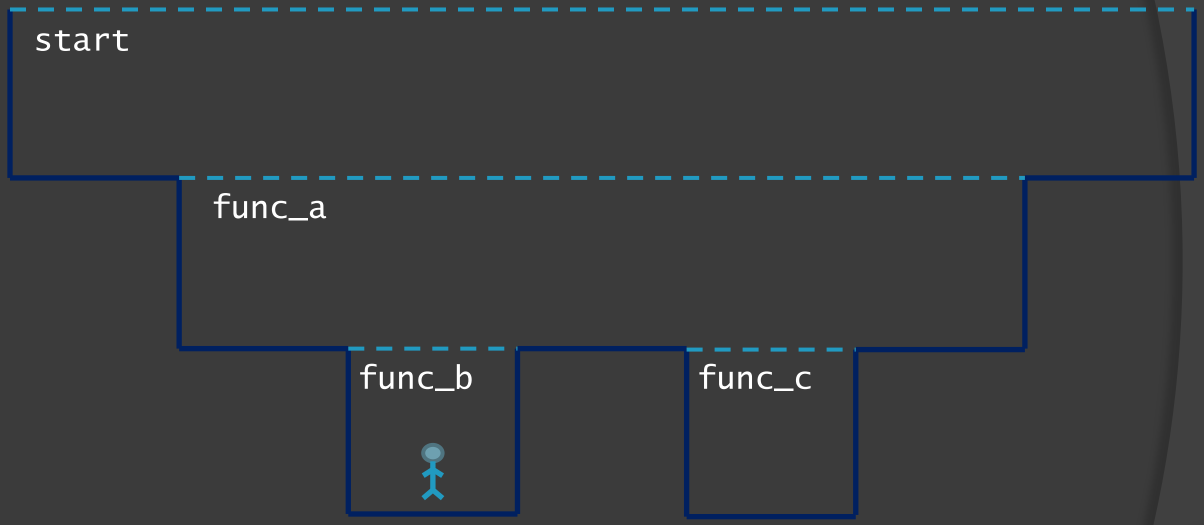
Stairs illustration

```
        call    func_a
        ...
    func_a:
  →     call    func_b
        call    func_c
        ret
    func_b:
        ...
        ret
    func_c:
        ...
        ret
```



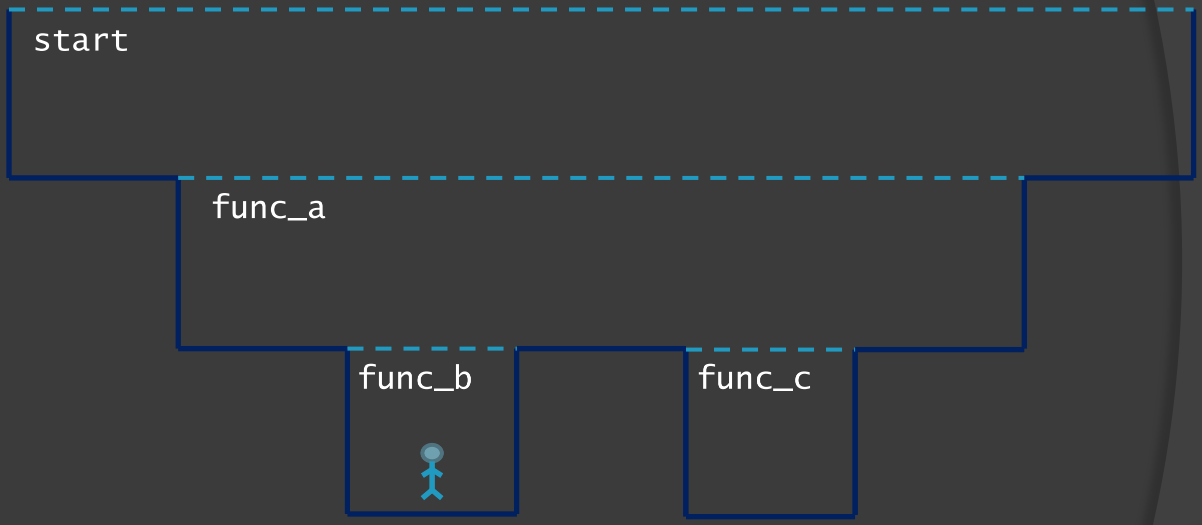
Stairs illustration

```
    call    func_a
    ...
func_a:
    call    func_b
    call    func_c
    ret
func_b:
→   ...
    ret
func_c:
    ...
    ret
```



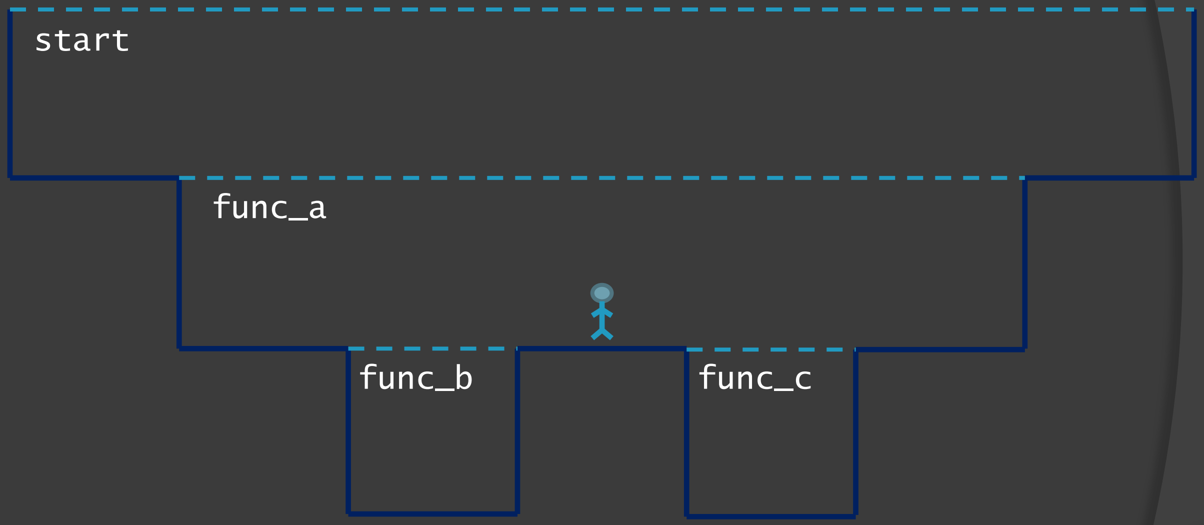
Stairs illustration

```
        call    func_a
        ...
    func_a:
        call    func_b
        call    func_c
        ret
    func_b:
        ...
  →     ret
    func_c:
        ...
        ret
```



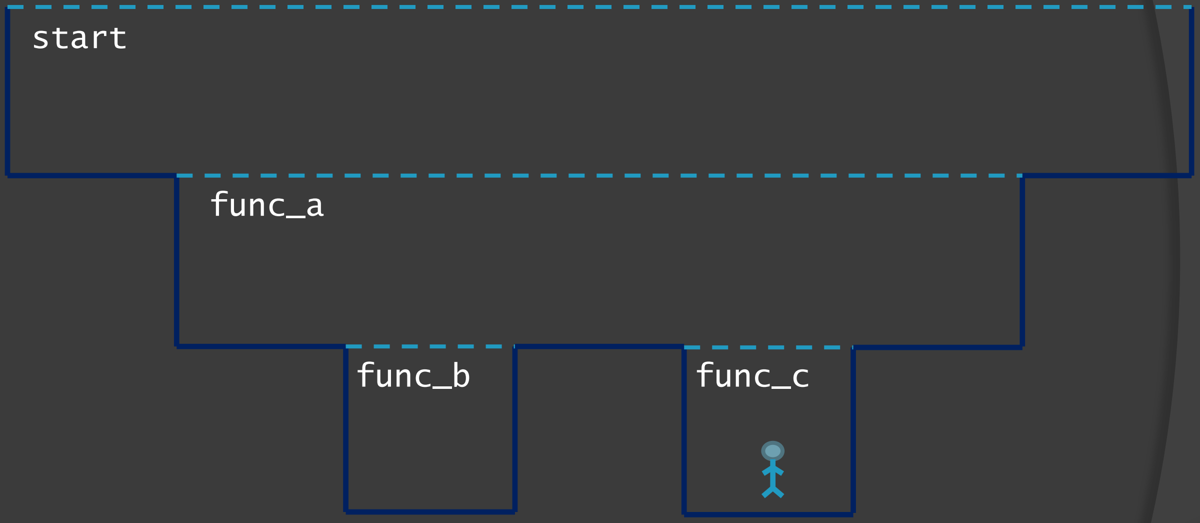
Stairs illustration

```
    call    func_a
    ...
func_a:
    call    func_b
 →  call    func_c
    ret
func_b:
    ...
    ret
func_c:
    ...
    ret
```



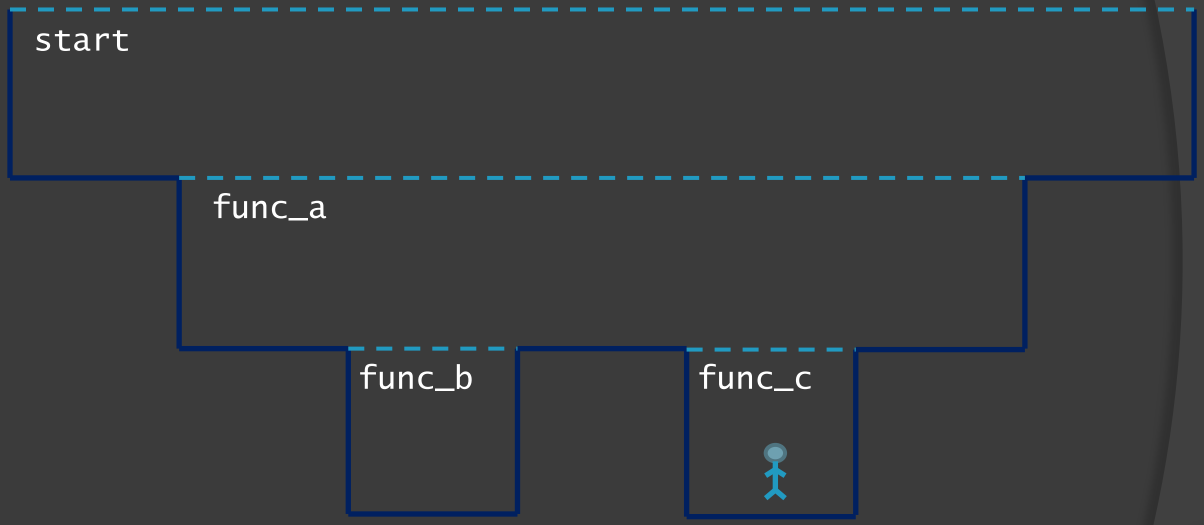
Stairs illustration

```
          call      func_a
          ...

func_a:
          call      func_b
          call      func_c
          ret

func_b:
          ...
          ret

func_c:
   →      ...
          ret
```



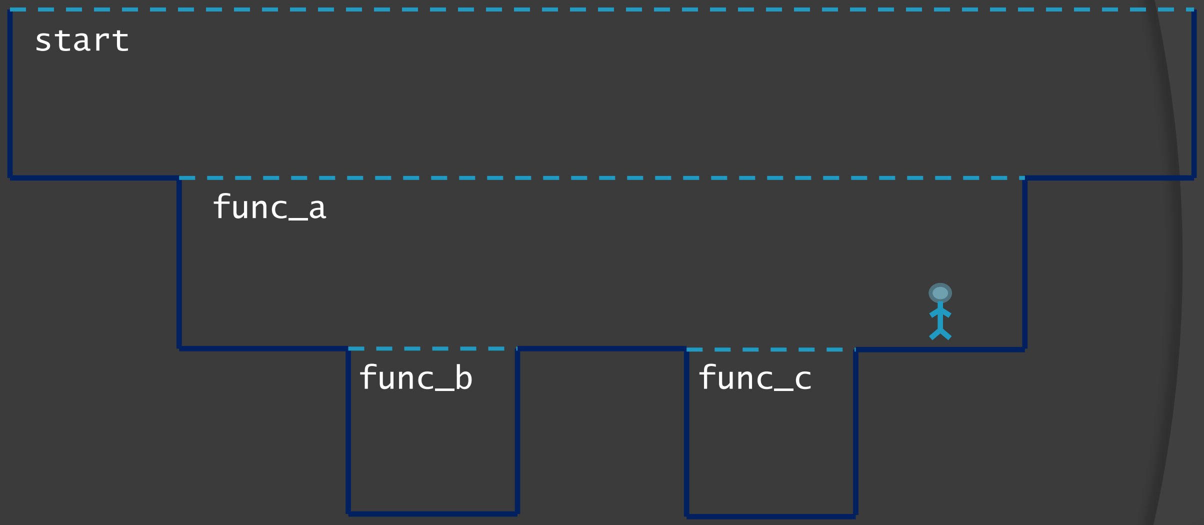
Stairs illustration

```
        call    func_a
        ...
func_a:
        call    func_b
        call    func_c
        ret
func_b:
        ...
        ret
func_c:
        ...
   →    ret
```



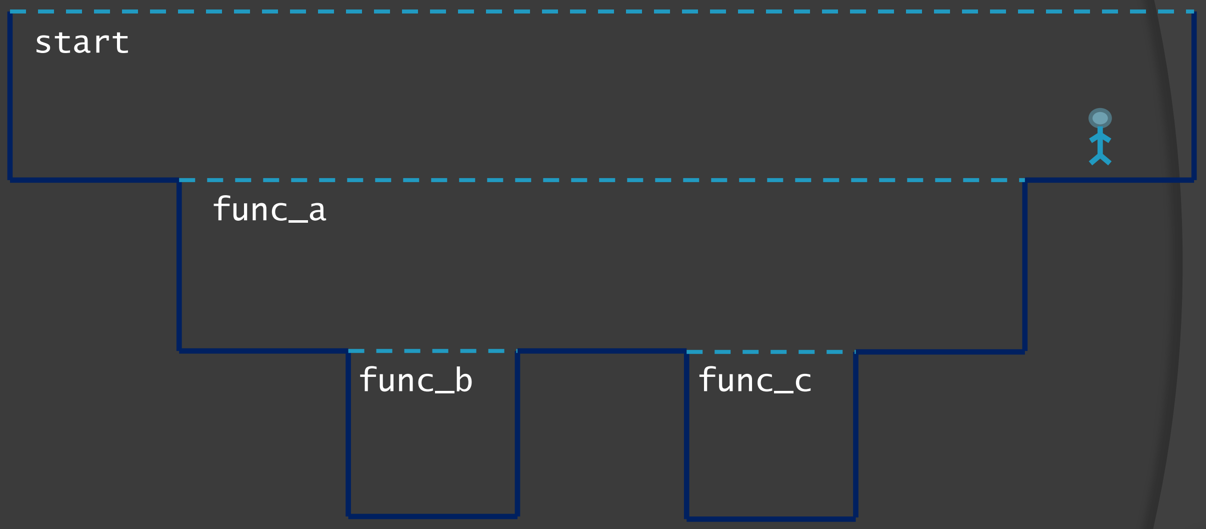
Stairs illustration

```
    call    func_a
    ...

func_a:
    call    func_b
    call    func_c
→   ret

func_b:
    ...
    ret

func_c:
    ...
    ret
```



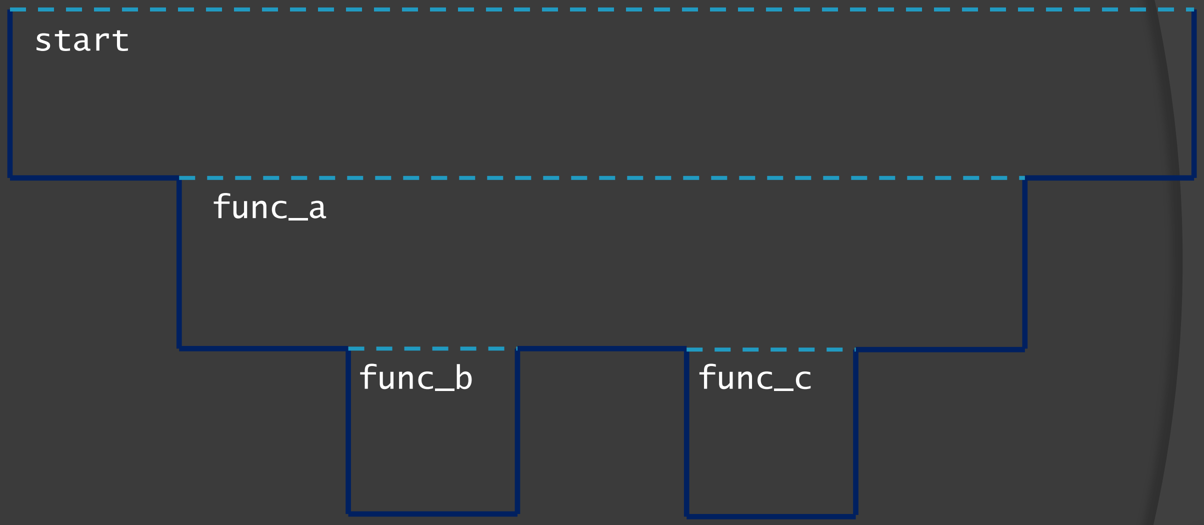
Stairs illustration

```
        call    func_a
→       ...

func_a:
        call    func_b
        call    func_c
        ret

func_b:
        ...
        ret

func_c:
        ...
        ret
```



Stairs illustration

```
      call      func_a
      ...

func_a:
      call      func_b
      call      func_c
      ret

func_b:
      ...
      ret

func_c:
      ...
      ret
```



- The depth corresponds to the amount of elements currently occupied in the stack.

Summary

- CALL and RET are special purpose jumps.

- CALL and RET allow us to call a function and return from a function call.

  - CALL pushes the return address to the stack.
  - RET pops the return address from the stack.

- The stack helps us navigate the calls graph.
  - It contains the full path to the current function.

Exercises

- Intro
  - Local, Anonymous labels
  - Stack balancing
- Read Code
- Write code