Basic Assembly

Local state

Assembly language programming
By xorpd

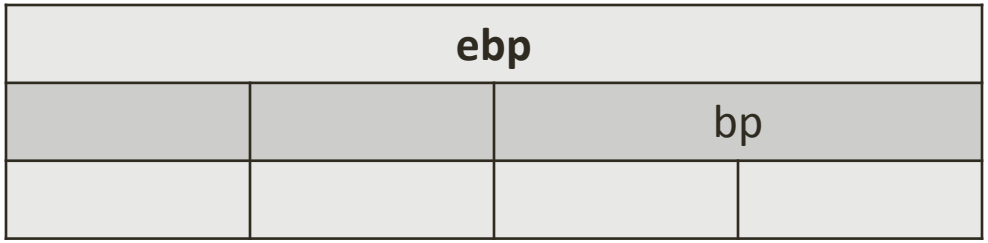
xorpd.net

Objectives

- We will study the EBP register, and its use as the stack base pointer.
- We will learn about function's local variables, and where to store them.
- We will understand the general structure of the stack during the invocation of functions in our program.

EBP

- EBP – Extended base pointer.
  - A register of size 32 bits.



- Historically, ESP could not be used to access memory directly.
  - “dword [esp + 4]” was not possible.
  - The EBP register was used instead.

- These days ESP can access memory directly.
  - EBP is usually used to “hold” the stack frame.

Example

- Extracting the arguments from the stack:

```
sum_nums:

    mov    esi,dword [esp + 4]
    mov    ecx,dword [esp + 8]
    xor    edx,edx
    jecxz  .no_nums
.next_dw:
    lodsd
    add    edx,eax
    loop   .next_dw
.no_nums:
    mov    eax,edx

    ret
```

Example

- Extracting the arguments from the stack:

```
sum_nums:

    mov     esi,dword [esp + 4]
    mov     ecx,dword [esp + 8]
    xor     edx,edx
    jecxz   .no_nums
.next_dw:
    lodsd
    add     edx,eax
    loop    .next_dw
.no_nums:
    mov     eax,edx

    ret
```

```
sum_nums:
    push    ecx
    mov     esi,dword [esp + 8]
    mov     ecx,dword [esp + 0ch]
    xor     edx,edx
    jecxz   .no_nums
.next_dw:
    lodsd
    add     edx,eax
    loop    .next_dw
.no_nums:
    mov     eax,edx
    pop     ecx
    ret
```

Example

- Extracting the arguments from the stack:

```
sum_nums:

    mov     esi,dword [esp + 4]
    mov     ecx,dword [esp + 8]
    xor     edx,edx
    jecxz   .no_nums
.next_dw:
    lodsd
    add     edx,eax
    loop    .next_dw
.no_nums:
    mov     eax,edx

    ret
```

```
sum_nums:
    push    ecx
    mov     esi,dword [esp + 8]
    mov     ecx,dword [esp + 0ch]
    xor     edx,edx
    jecxz   .no_nums
.next_dw:
    lodsd
    add     edx,eax
    loop    .next_dw
.no_nums:
    mov     eax,edx
    pop     ecx
    ret
```

- ESP might change many times during the function.
- We have to follow the current offset of ESP to be able to access arguments.
  - Not fun :(

Holding the stack frame

- We could use EBP to remember what ESP was initially.
  - Then we don't care about changes to esp.

```
sum_nums:
    mov     ebp,esp
    push    ecx
    mov     esi,dword [ebp + 4]
    mov     ecx,dword [ebp + 8]
    xor     edx,edx
    jecxz   .no_nums
.next_dw:
    lodsd
    add     edx,eax
    loop    .next_dw
.no_nums:
    mov     eax,edx
    pop     ecx
    ret
```

Holding the stack frame

- We could use EBP to remember what ESP was initially.
  - Then we don't care about changes to esp.

```
sum_nums:
    push   ebp ; Keep ebp
    mov    ebp,esp
    push   ecx
    mov    esi,dword [ebp + 4]
    mov    ecx,dword [ebp + 8]
    xor    edx,edx
    jecxz  .no_nums
.next_dw:
    lodsd
    add    edx,eax
    loop   .next_dw
.no_nums:
    mov    eax,edx
    pop    ecx
    pop    ebp ; Restore ebp
    ret
```

Holding the stack frame

- We could use EBP to remember what ESP was initially.
  - Then we don't care about changes to esp.

```
func:
    push   ebp ; Keep ebp
    mov    ebp,esp

    ...

    pop    ebp ; Restore ebp
    ret
```

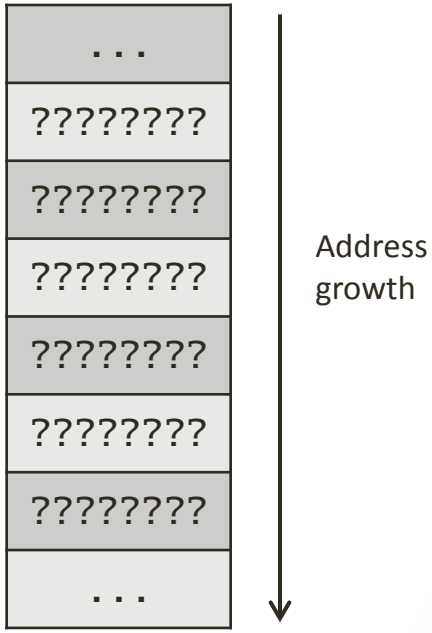
Holding the stack frame

- We could use EBP to remember what ESP was initially.
  - Then we don't care about changes to esp.

```
func:
    push   ebp ; Keep ebp
    mov    ebp,esp

    ...

    pop    ebp ; Restore ebp
    ret
```



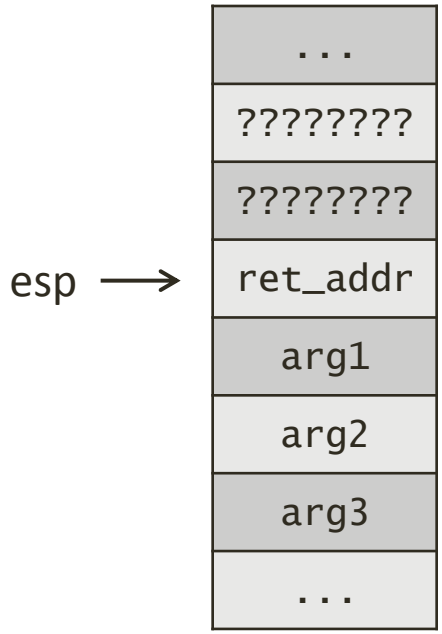
Holding the stack frame

- We could use EBP to remember what ESP was initially.
  - Then we don't care about changes to esp.

```
func:
→ push   ebp ; Keep ebp
  mov    ebp,esp

  ...

  pop    ebp ; Restore ebp
  ret
```



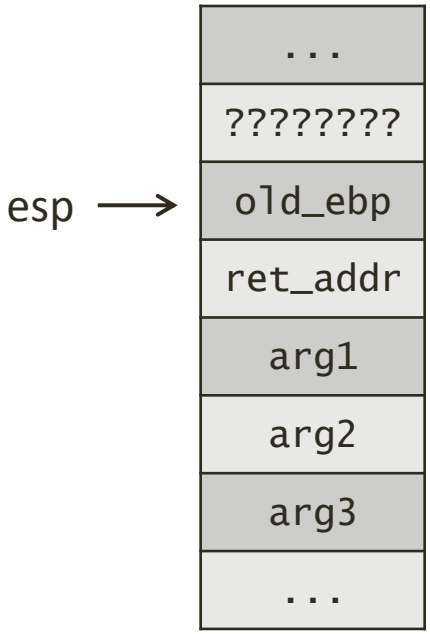
Holding the stack frame

- We could use EBP to remember what ESP was initially.
  - Then we don't care about changes to esp.

```
func:
    push   ebp ; Keep ebp
    mov    ebp,esp

    ...

    pop    ebp ; Restore ebp
    ret
```

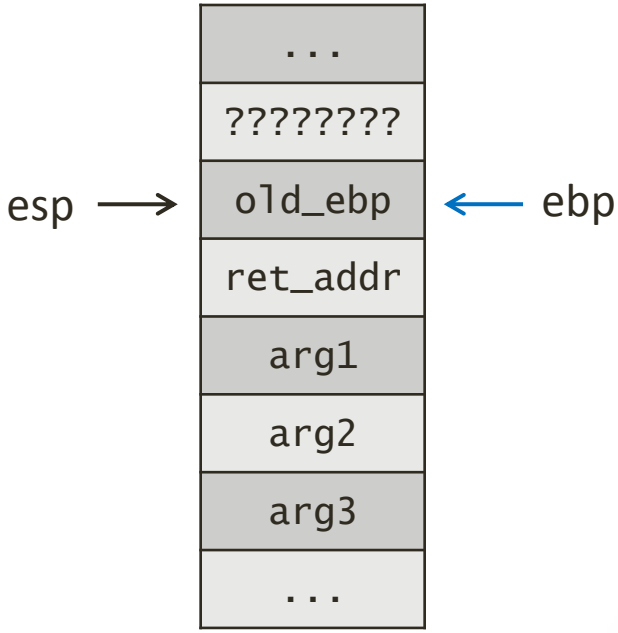


Holding the stack frame

- We could use EBP to remember what ESP was initially.
  - Then we don't care about changes to esp.

```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    
    ...

    pop     ebp ; Restore ebp
    ret
```

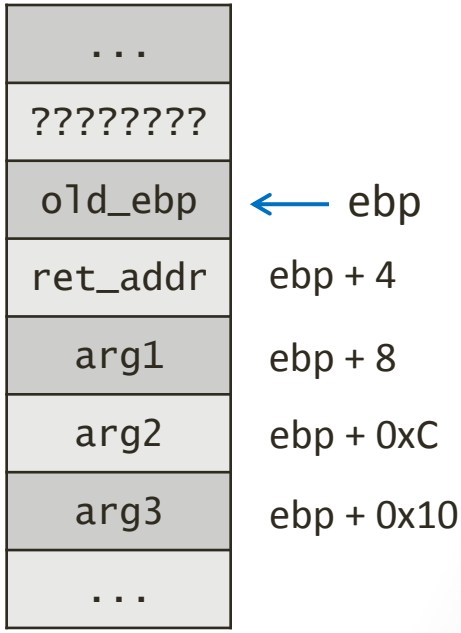


Holding the stack frame

- We could use EBP to remember what ESP was initially.
  - Then we don't care about changes to esp.

```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
→
    ...

    pop     ebp ; Restore ebp
    ret
```



Inside the function:

- old_ebp: $[ebp]$
- ret_addr: $[ebp + 4]$
- arg 1: $[ebp + 8]$
- arg k: $[ebp + 4 + 4 \cdot k]$

Local Variables

- We might need some memory during function execution.
- Using the global memory is usually not a good option.
  - Not very modular. (We want the function to be self contained).
  - Other cons. (Not reentrant)
- We could use the stack!

Local Variables (Cont.)

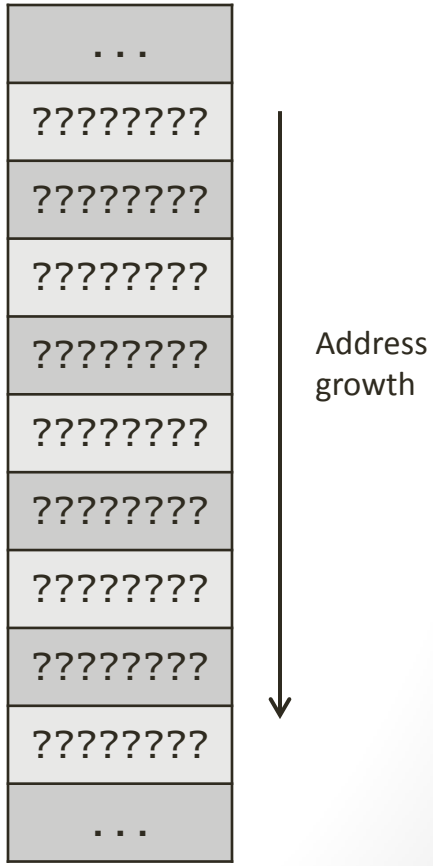
- Decrease esp to allocate space on stack:

```
func:
    push   ebp ; Keep ebp
    mov    ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub    esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add    esp,4*3
    pop    ebp ; Restore ebp
    ret
```

Local Variables (Cont.)

- Decrease esp to allocate space on stack:

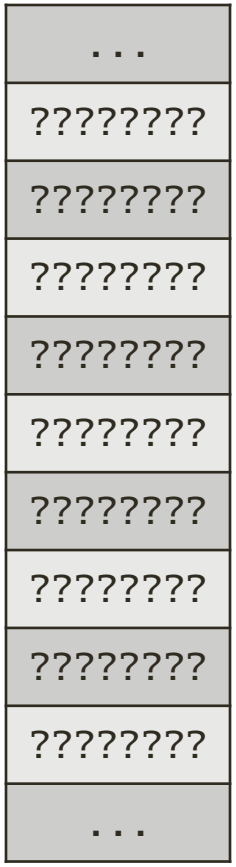
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Decrease esp to allocate space on stack:

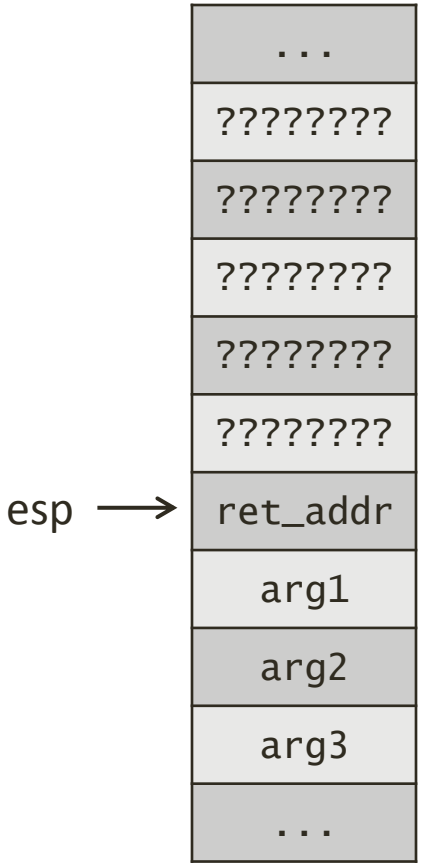
```
func:
    push   ebp ; Keep ebp
    mov    ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub    esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add    esp,4*3
    pop    ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Decrease esp to allocate space on stack:

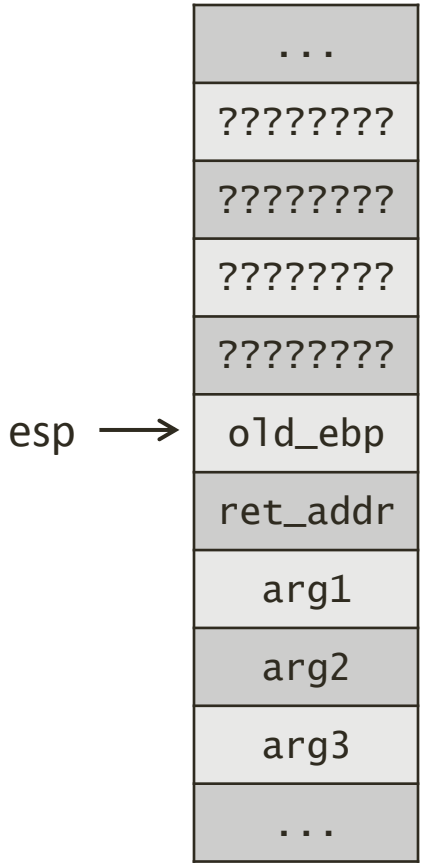
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Decrease esp to allocate space on stack:

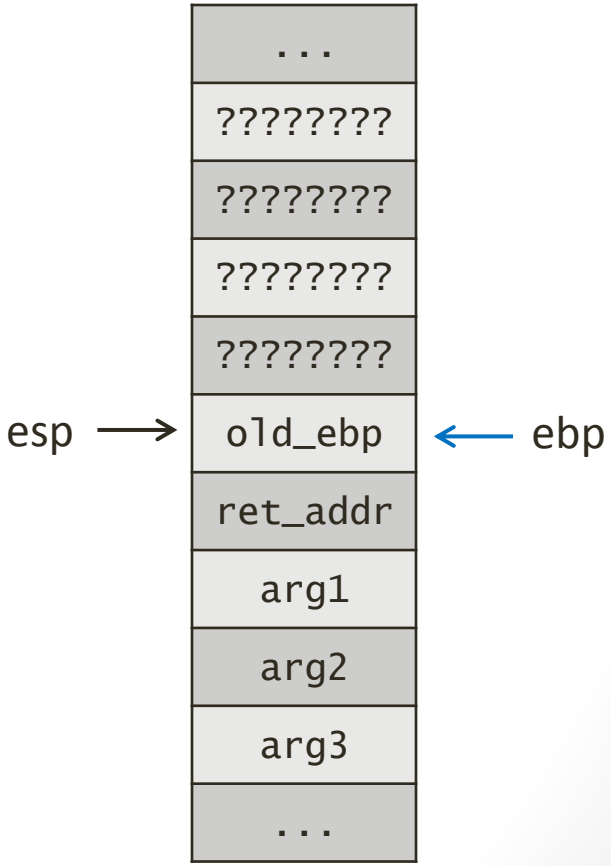
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Decrease esp to allocate space on stack:

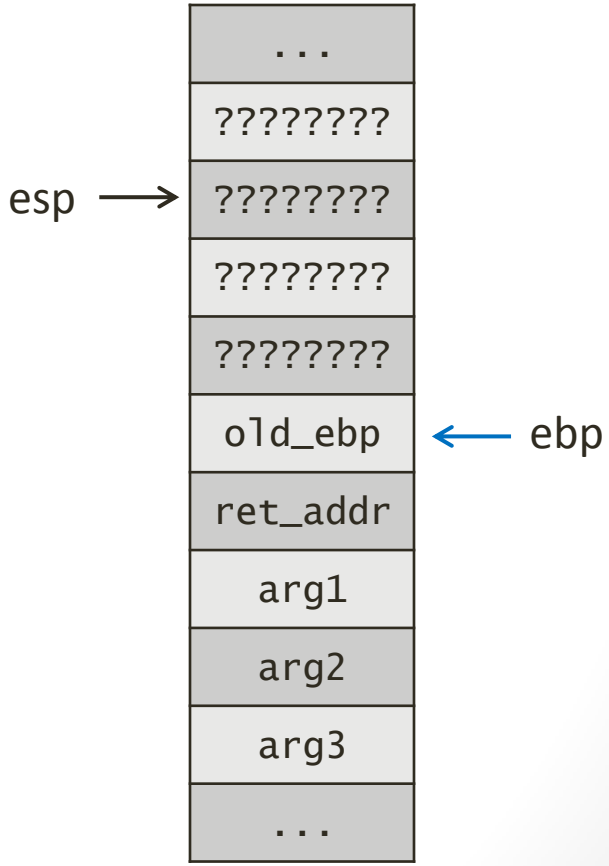
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Decrease esp to allocate space on stack:

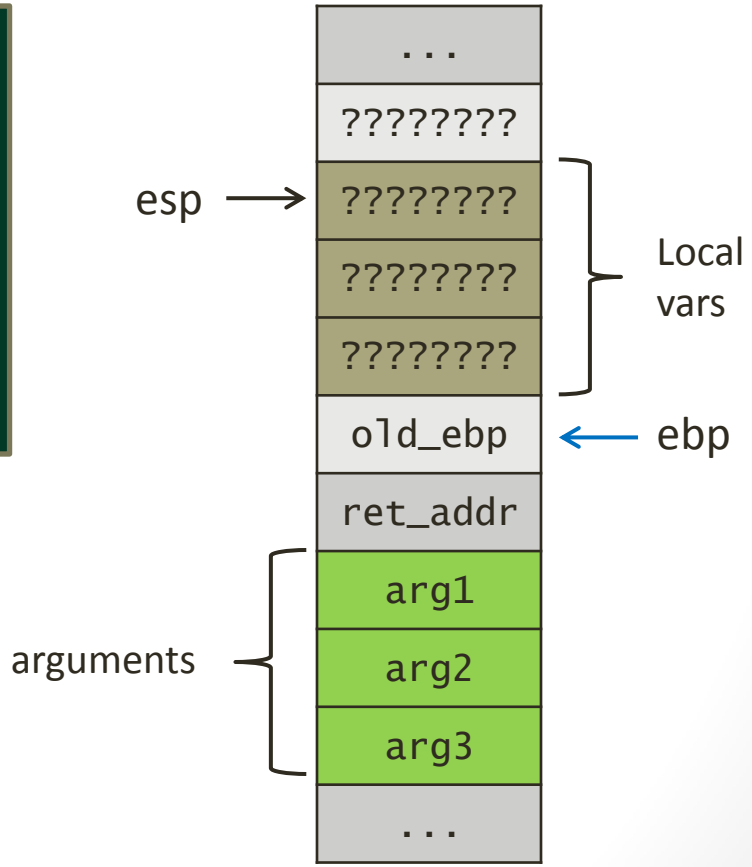
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Decrease esp to allocate space on stack:

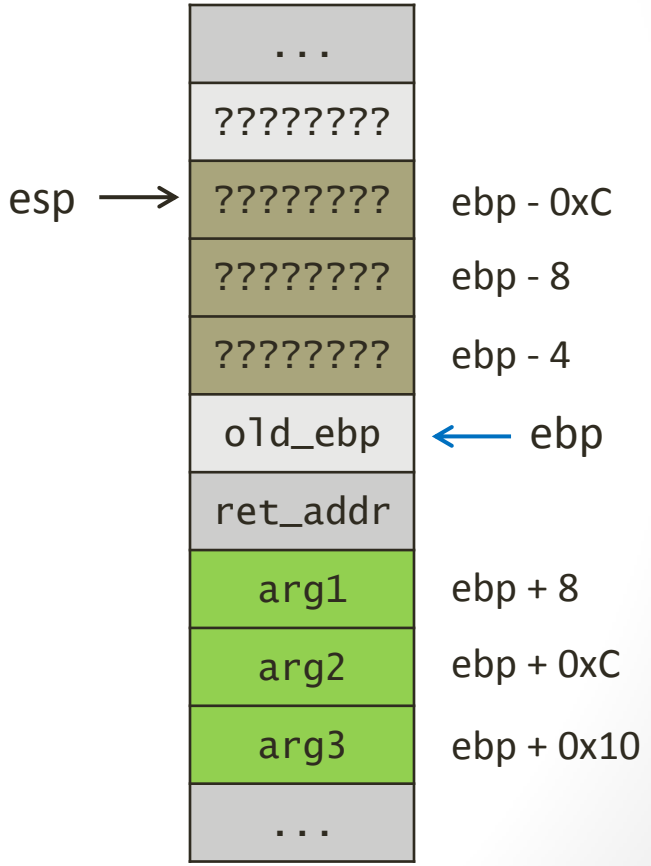
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Decrease esp to allocate space on stack:

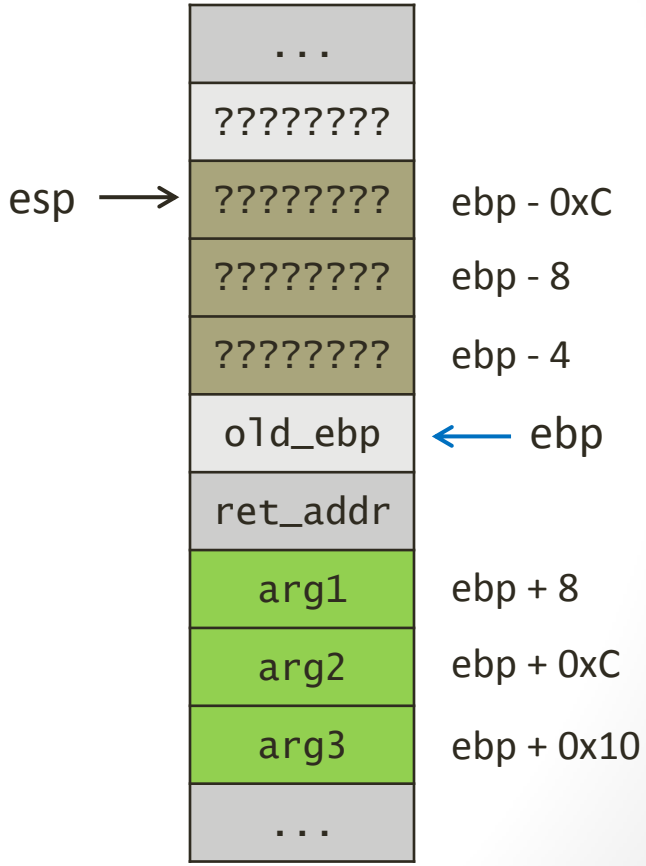
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Decrease esp to allocate space on stack:

```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```

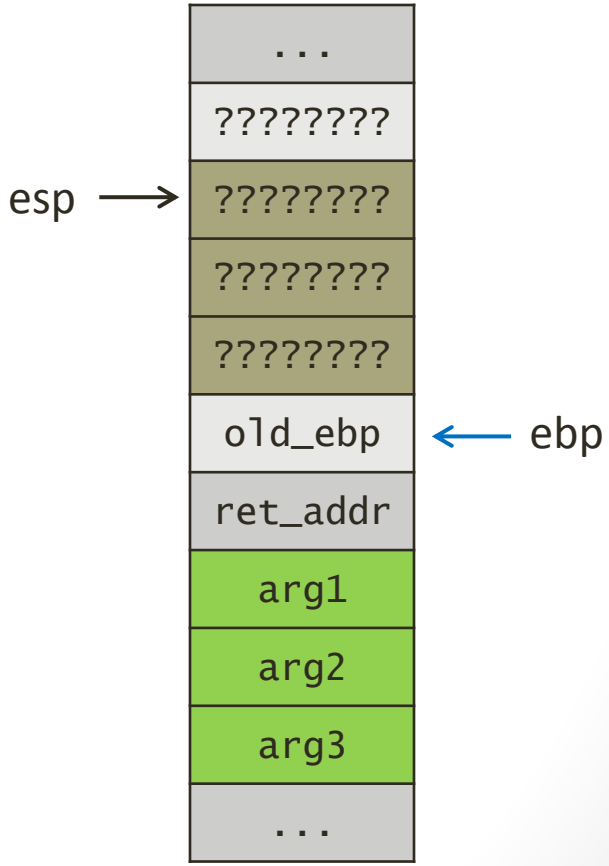


- Variables: $[ebp - 4 \cdot k]$
- Arguments: $[ebp + 4 + 4 \cdot r]$

Local Variables (Cont.)

- Increase esp to free the allocated stack space:

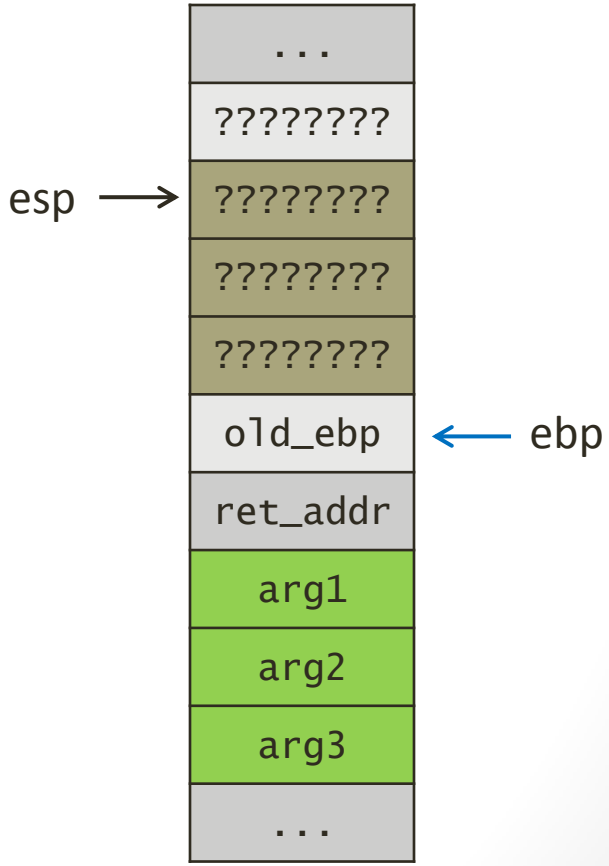
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Increase esp to free the allocated stack space:

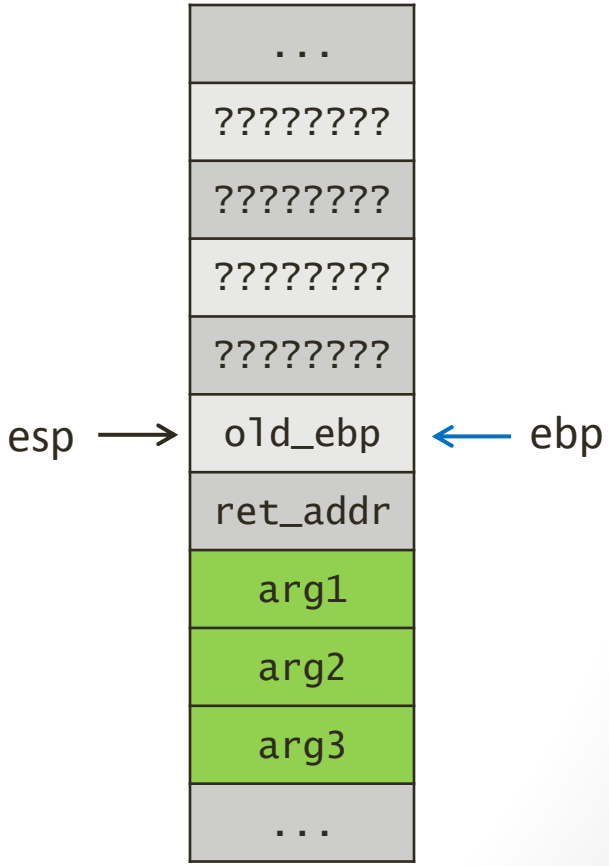
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Increase esp to free the allocated stack space:

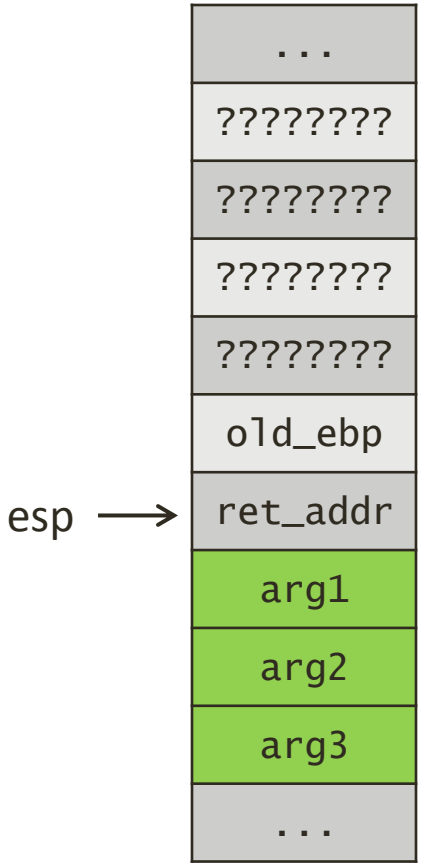
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Increase esp to free the allocated stack space:

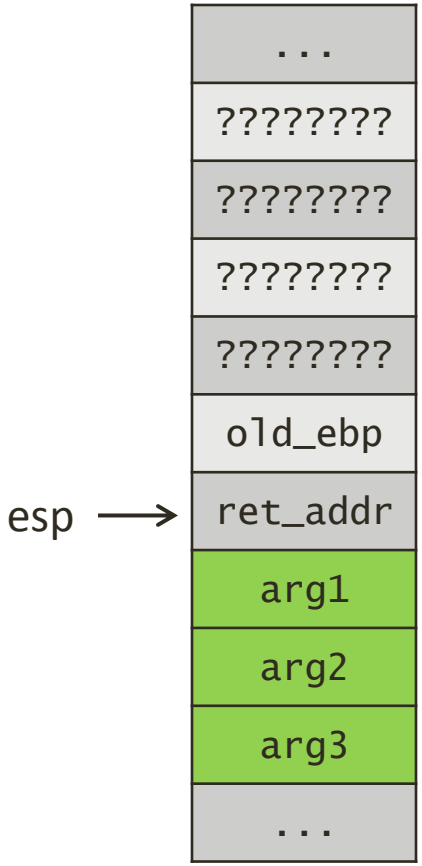
```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



Local Variables (Cont.)

- Increase esp to free the allocated stack space:

```
func:
    push    ebp ; Keep ebp
    mov     ebp,esp
    ; 'Allocate' 3 dwords on stack:
    sub     esp,4*3
    ...
    ; 'Free' the 3 allocated dwords:
    add     esp,4*3
    pop     ebp ; Restore ebp
    ret
```



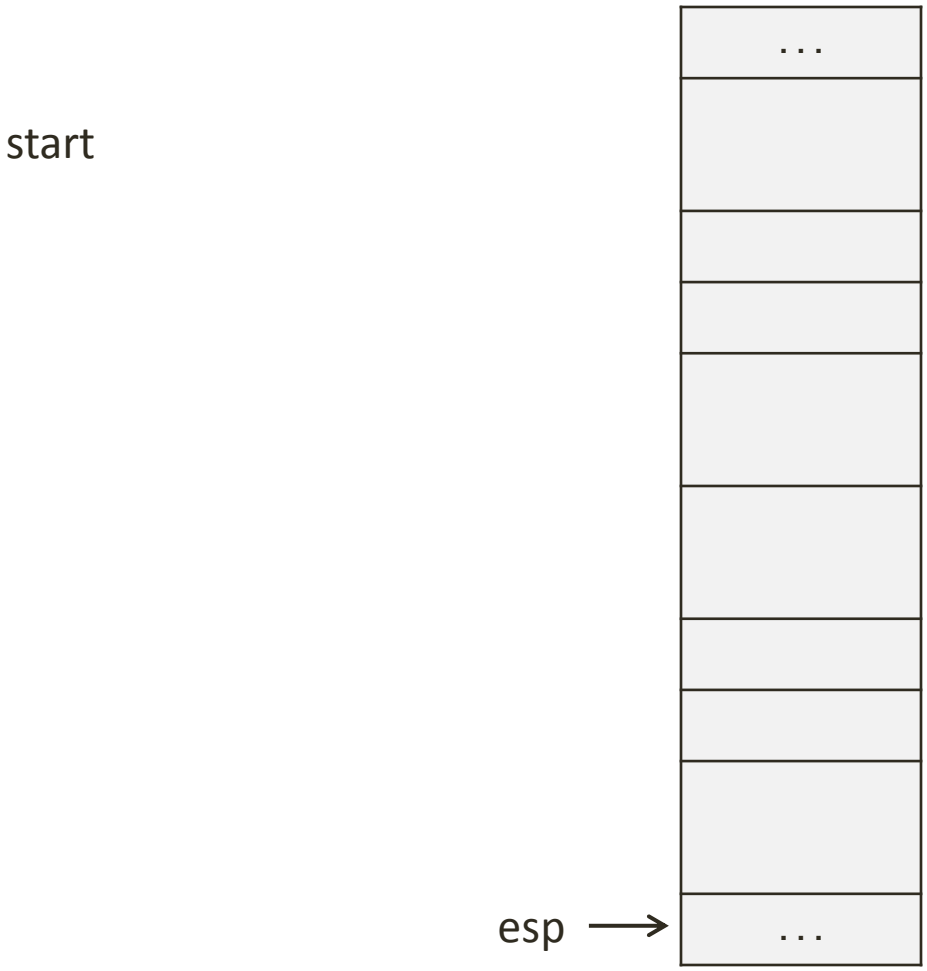
- The local state is freed when the function returns.
  - Local data lives a short life.

The call stack

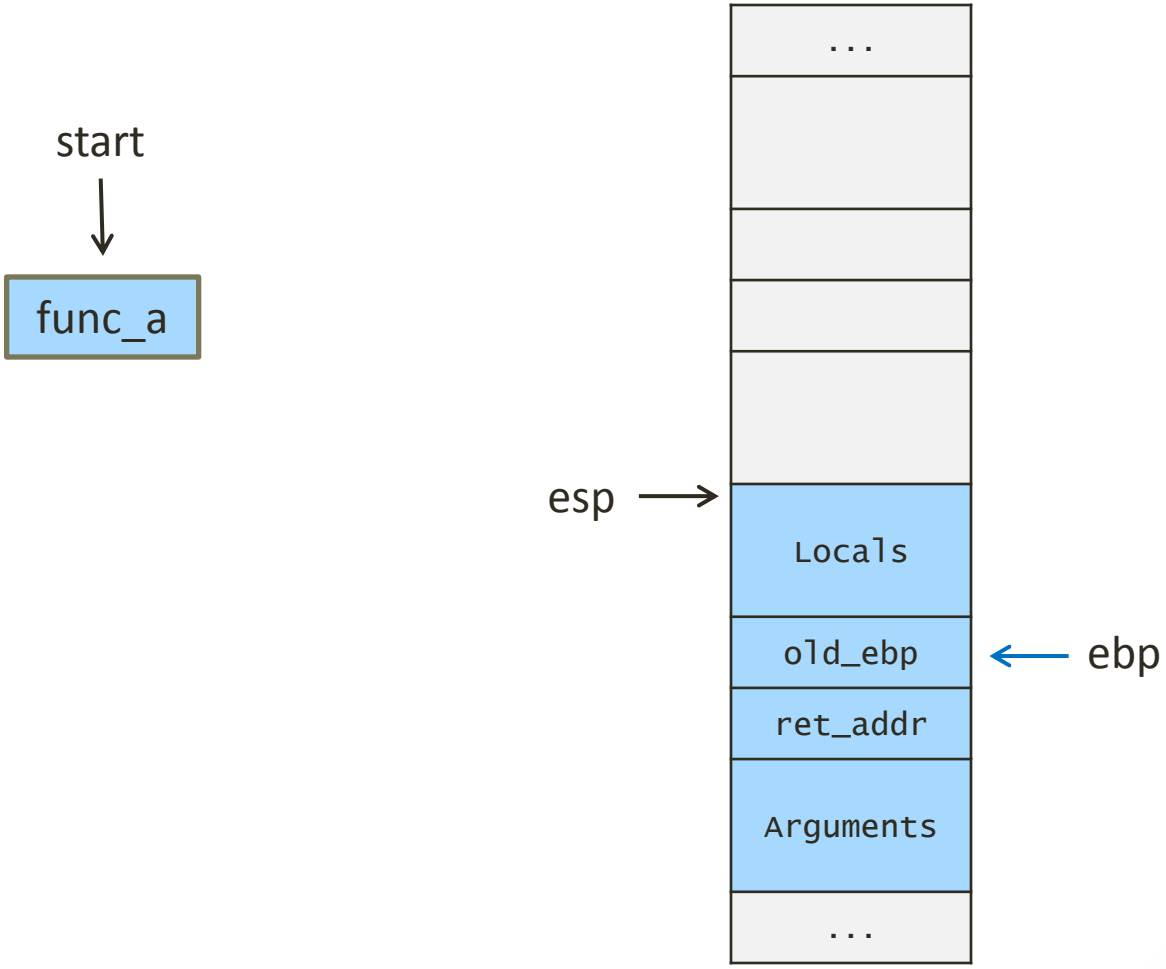
- Every function being called has a “frame” inside the stack.
  - Arguments
  - Return address
  - Old EBP (Previous frame)
  - Local variables

- The stack tells the story of all functions currently “open”, all the way to the current function.

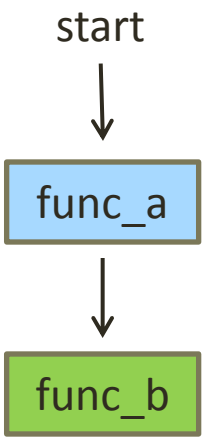
The call stack (Cont.)

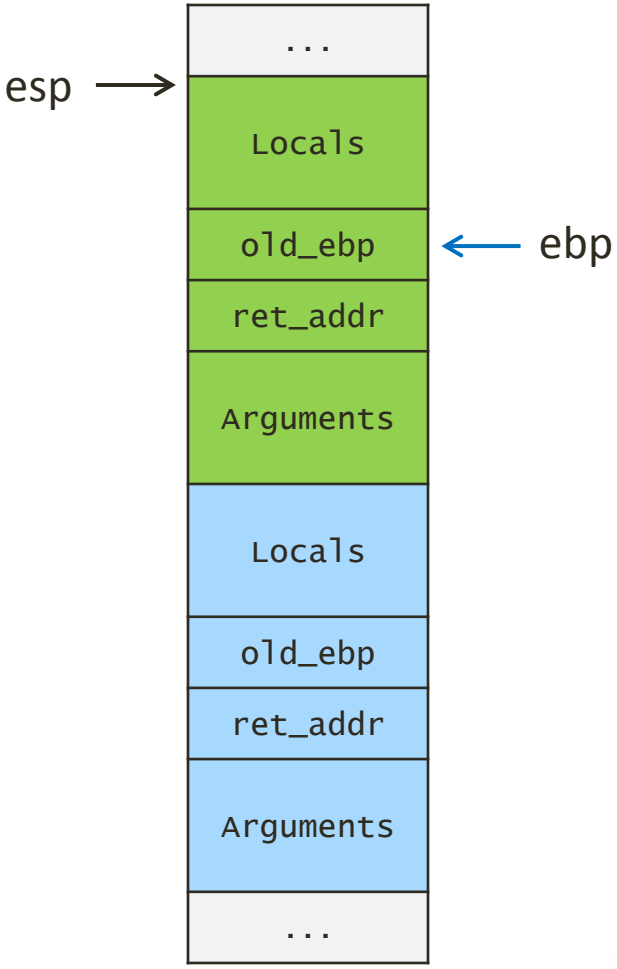


The call stack (Cont.)

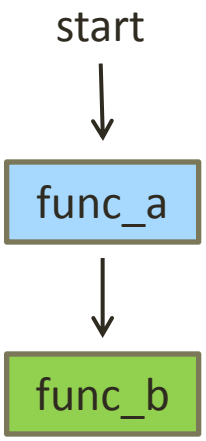


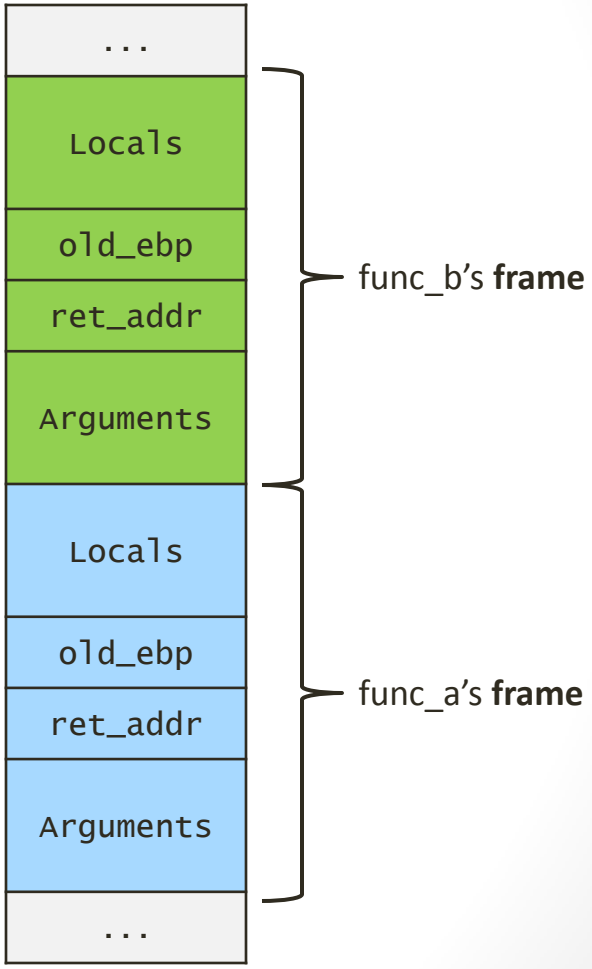
The call stack (Cont.)



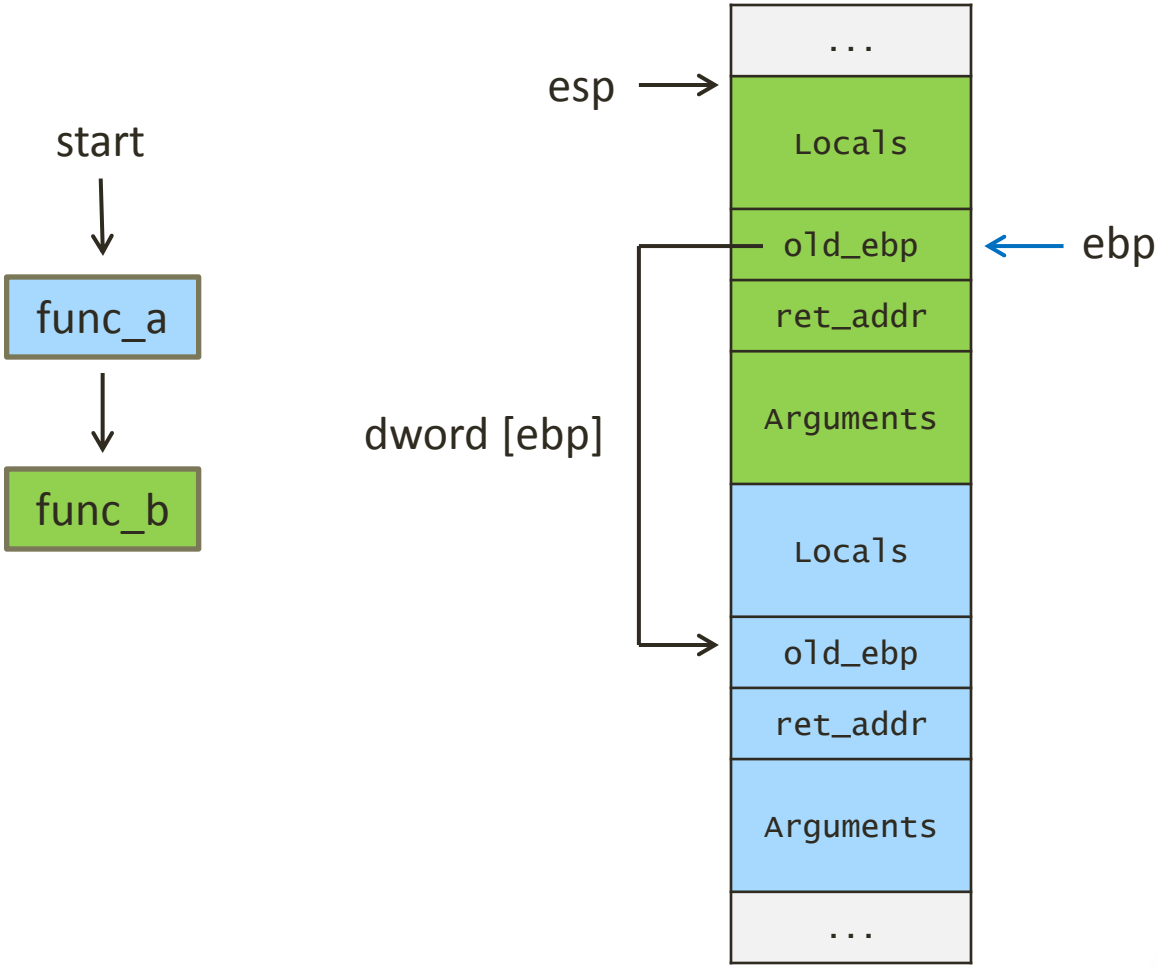


The call stack (Cont.)





The call stack (Cont.)

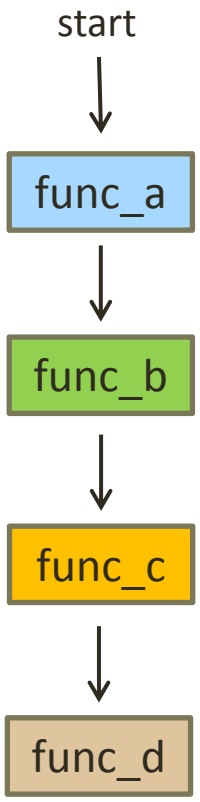


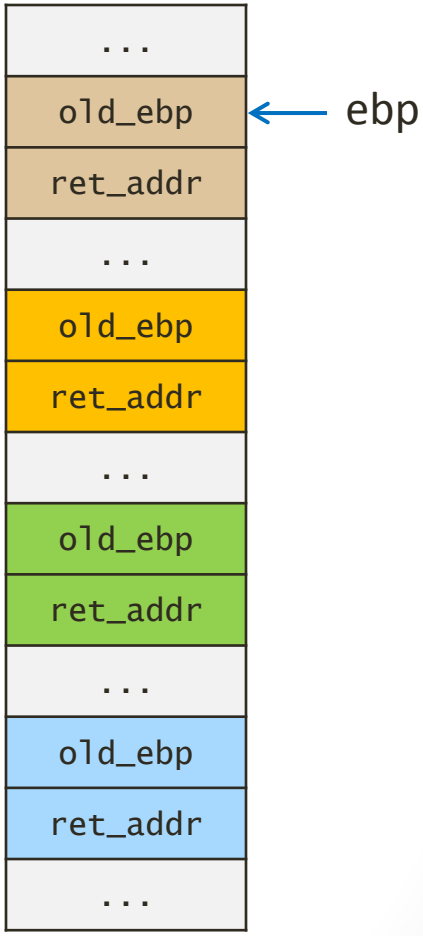
The call stack (Cont.)

- Traversing the call stack:
  - EBP entries form a “linked list”!

The call stack (Cont.)

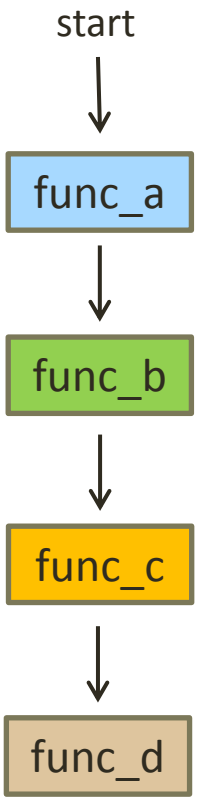
- Traversing the call stack:
  - EBP entries form a “linked list”!

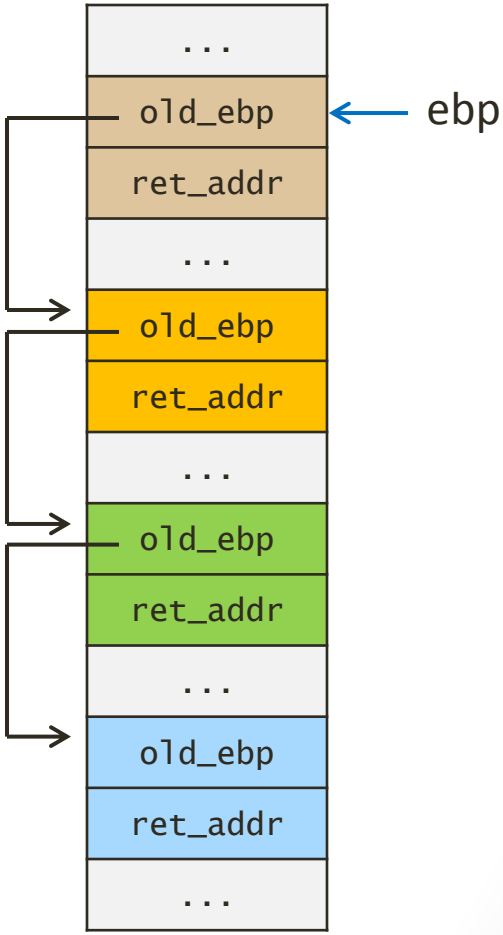




The call stack (Cont.)

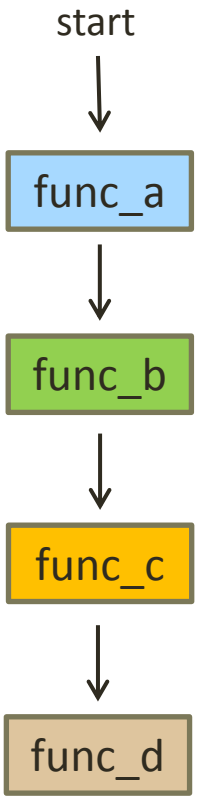
- Traversing the call stack:
  - EBP entries form a “linked list”!

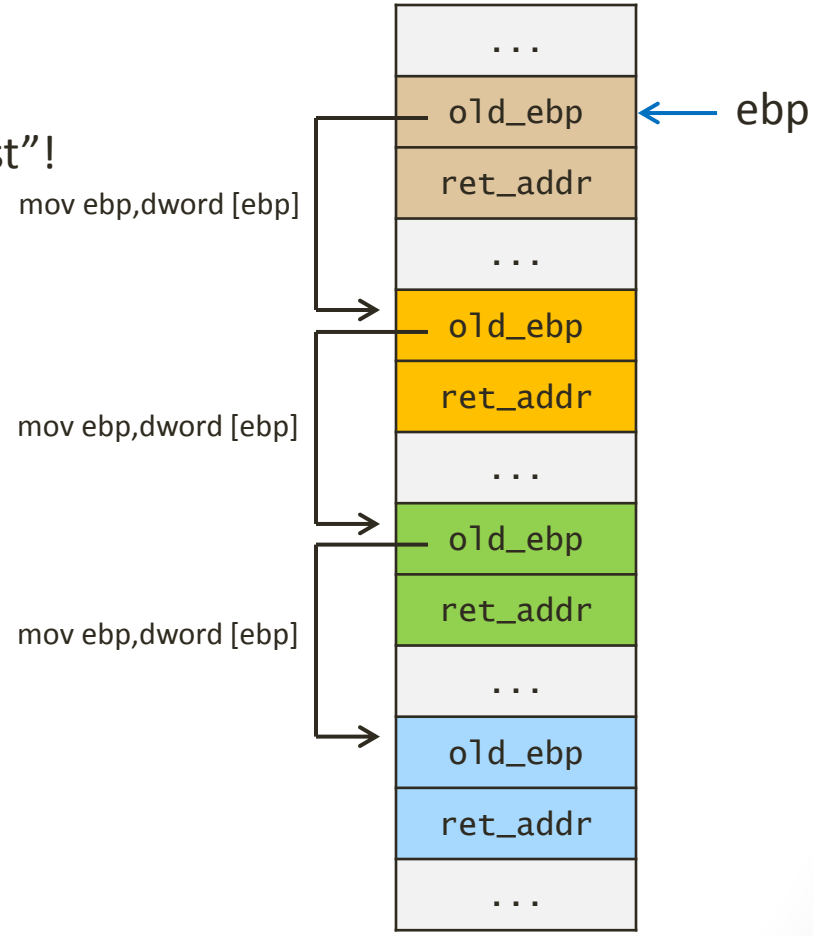




The call stack (Cont.)

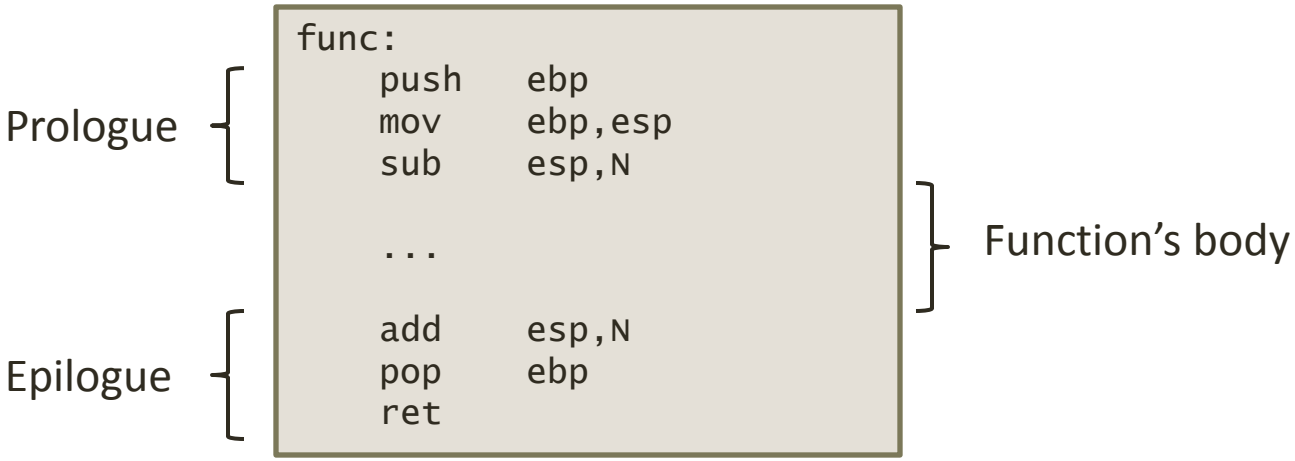
- Traversing the call stack:
  - EBP entries form a “linked list”!





ENTER and LEAVE

- Almost all functions begin and end in the same standard way.
- Those standard beginning and ending are also called **Prologue** and **Epilogue**.



- This construct is so common, that new instructions were introduced to do this job.

ENTER and LEAVE (Cont.)

- Building a stack frame using ENTER and LEAVE:

```
func:
    push   ebp
    mov    ebp,esp
    sub    esp,N

    ...

    add    esp,N
    pop    ebp
    ret
```

ENTER and LEAVE (Cont.)

- Building a stack frame using ENTER and LEAVE:

```
func:
    push    ebp
    mov     ebp,esp
    sub     esp,N

    ...

    add     esp,N
    pop     ebp
    ret
```

```
func:
    enter   N,0

    ...

    add     esp,N
    pop     ebp
    ret
```

ENTER and LEAVE (Cont.)

- Building a stack frame using ENTER and LEAVE:

```
func:
    push    ebp
    mov     ebp,esp
    sub     esp,N

    ...

    mov     esp,ebp
    pop     ebp
    ret
```

```
func:
    enter   N,0

    ...

    mov     esp,ebp
    pop     ebp
    ret
```

ENTER and LEAVE (Cont.)

- Building a stack frame using ENTER and LEAVE:

```
func:
    push    ebp
    mov     ebp,esp
    sub     esp,N

    ...

    mov     esp,ebp
    pop     ebp
    ret
```

```
func:
    enter   N,0

    ...

    leave
    ret
```

ENTER and LEAVE (Cont.)

- Building a stack frame using ENTER and LEAVE:

```
func:
    push    ebp
    mov     ebp,esp
    sub     esp,N

    ...

    mov     esp,ebp
    pop     ebp
    ret
```

```
func:
    enter   N,0

    ...

    leave
    ret
```

- Much cleaner :)

ENTER and LEAVE (Cont.)

- ENTER Size,NestingLevel
  - Make stack frame for procedure parameters.
  - Operation (For NestingLevel = 0):
    - ```
      push    ebp
      mov     ebp,esp
      sub     esp,N
      ```

      ENTER N,0

- LEAVE
  - High level procedure exit.
  - Operation:
    - ```
      mov     esp,ebp
      pop     ebp
      ```

      LEAVE

FAQ

- Do I have to use EBP for stack based argument passing?
  - Answer: No, but it is a good practice to do so.
- Which way should I choose to write my function prologue and epilogue – ENTER and LEAVE or `push ebp` … ?
  - Answer: It's your code, you decide.
  - ENTER is shorter. Some say it is slower though.

Summary

- We use EBP to “hold” the stack frame.
  - EBP has the initial value of ESP.
- We make space for local variables on the stack by decreasing ESP.
  - We free that space by increasing ESP.
- The stack is divided into frames of different functions.
- EBPs on the stack form a linked list that can be traversed.
- The ENTER and LEAVE instruction build and destroy the stack frame.

Exercises

- Read code.
- Write code.
- Enjoy :)