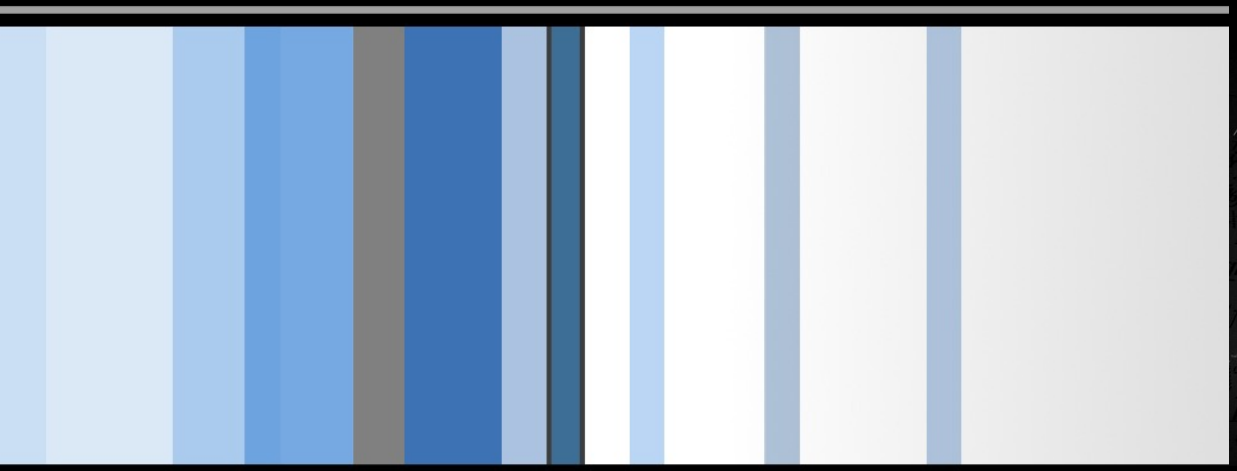
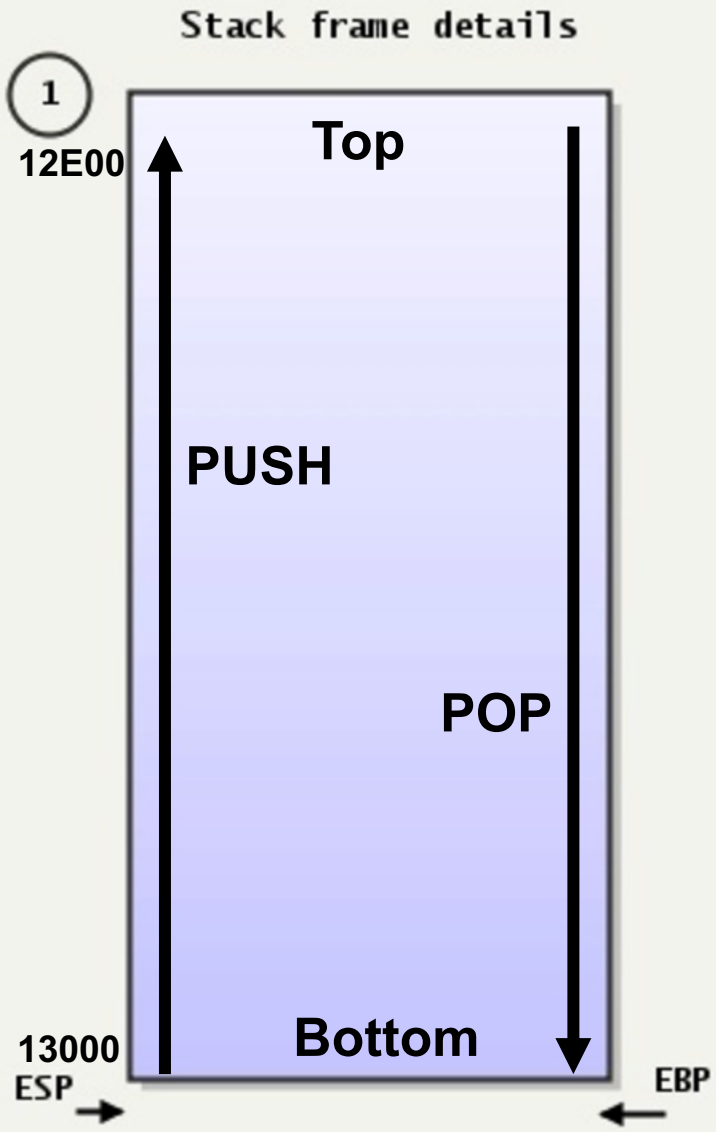
Class 6

CSC 471 Modern Malware Analysis
Stack and Stack Frame

Si Chen (schen@wcupa.edu)



The Stack



Stack:

- A special region of your computer's memory that **stores temporary variables** created by each functions
- The stack is a "**LIFO**" (last in, first out) data structure
- Once a stack variable is freed, that region of memory becomes available for other stack variables.

Properties:

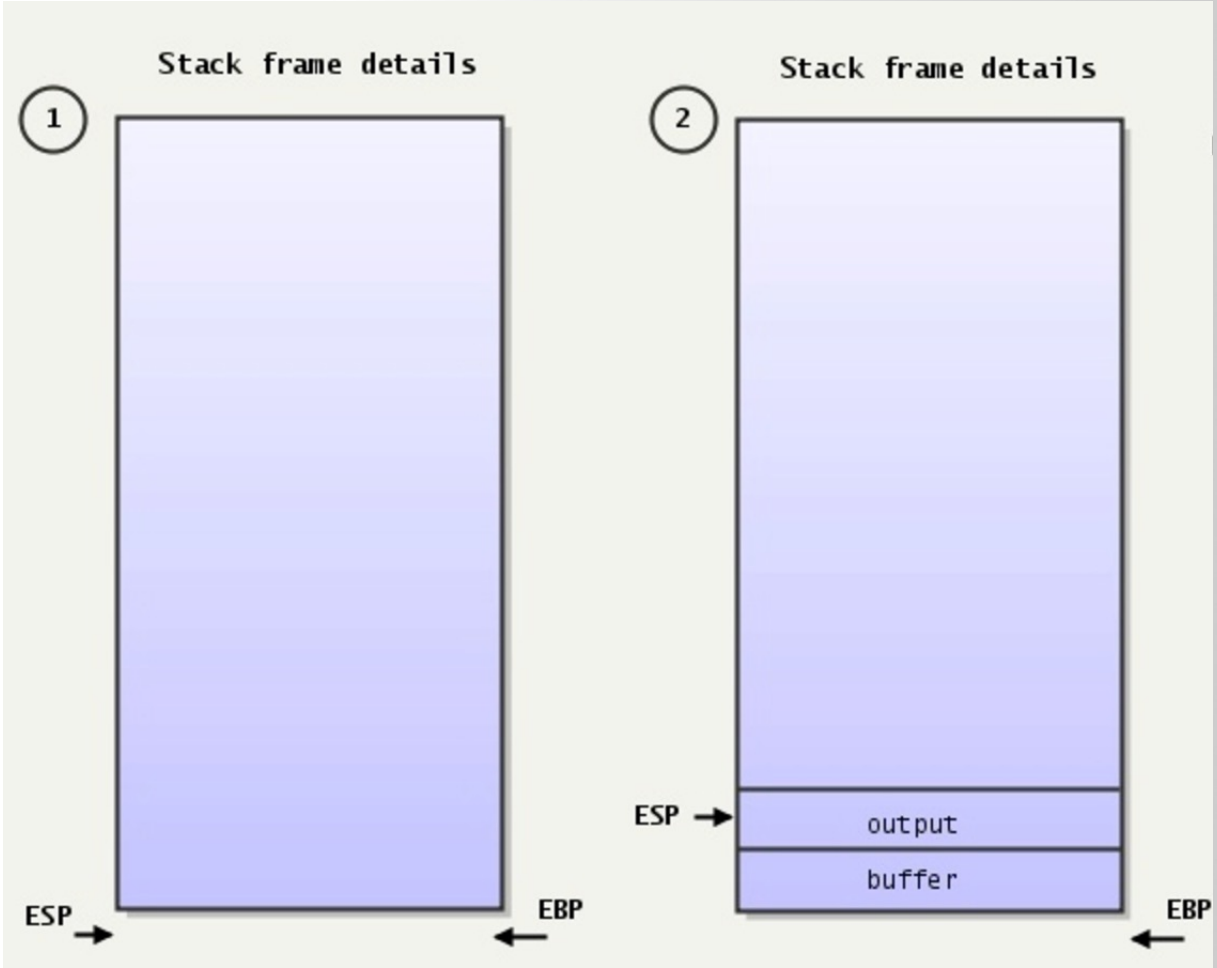
- the stack grows and shrinks as functions **push and pop** local **variables**
- there is no need to manage the memory yourself, variables are allocated and freed **automatically**
- the **stack has size limits**
- stack variables only exist while the function that created them, is running

EBP—Pointer to data on the stack
ESP—Stack pointer

The Stack

Stack:

- A special region of your computer's memory that **stores temporary variables** created by each functions
- The stack is a "**LIFO**" (last in, first out) data structure
- Once a stack variable is freed, that region of memory becomes available for other stack variables.



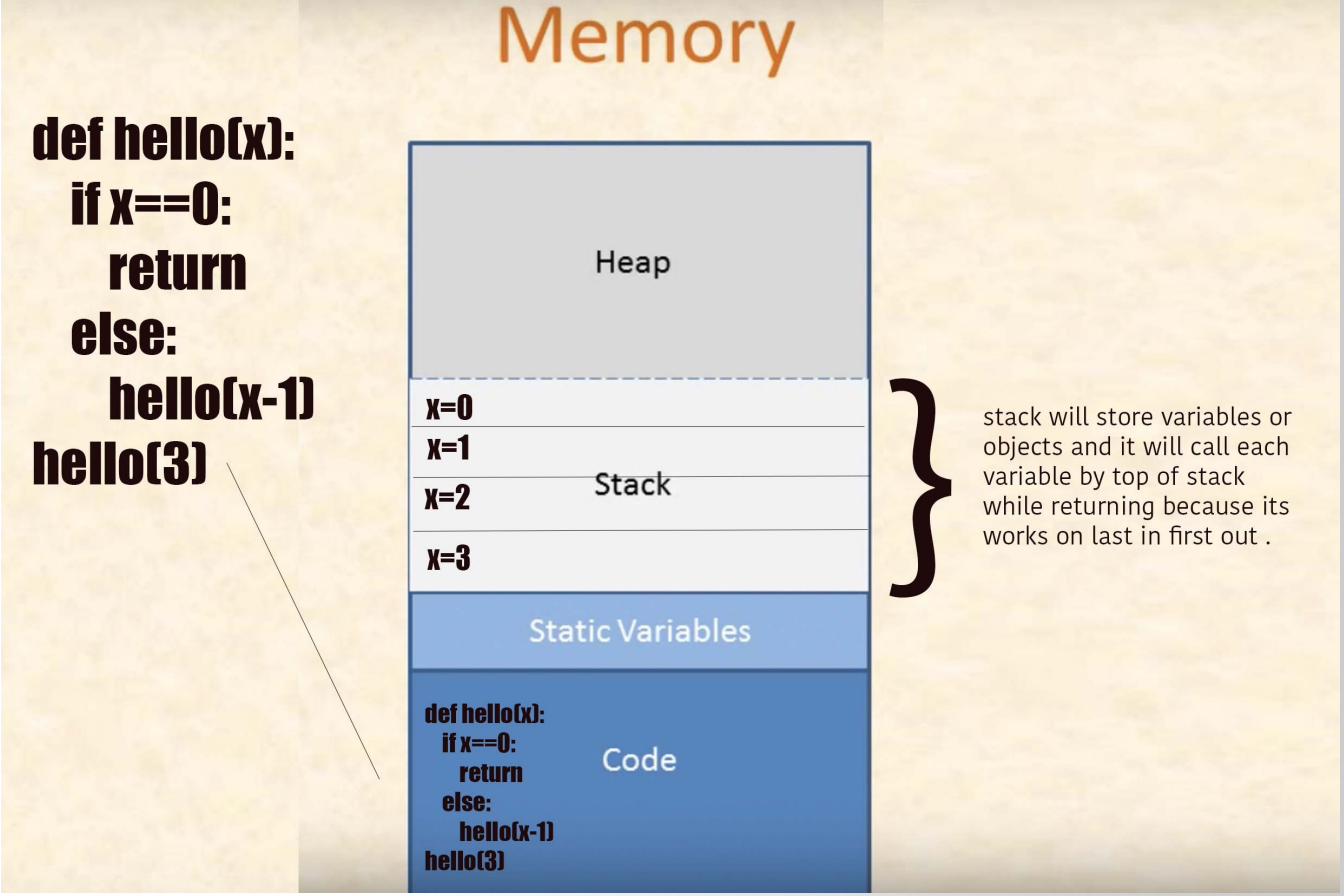
Stack

- Pass arguments
- Save the return address
- Save **local variable**

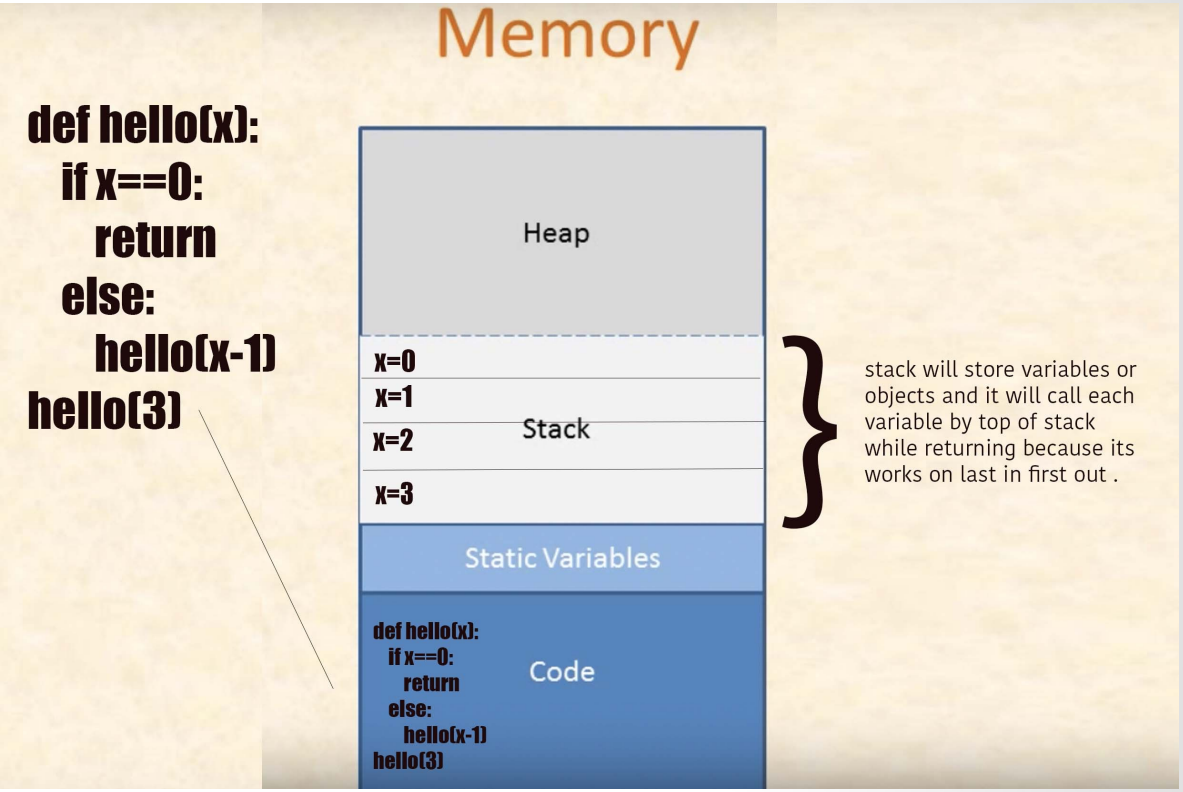
Stack Frame

Stack Frame

- A stack frame is **a frame of data that gets pushed onto the stack**.
- In the case of a **call stack**, a stack frame would represent **a function call and its argument data**.



Stack Frame

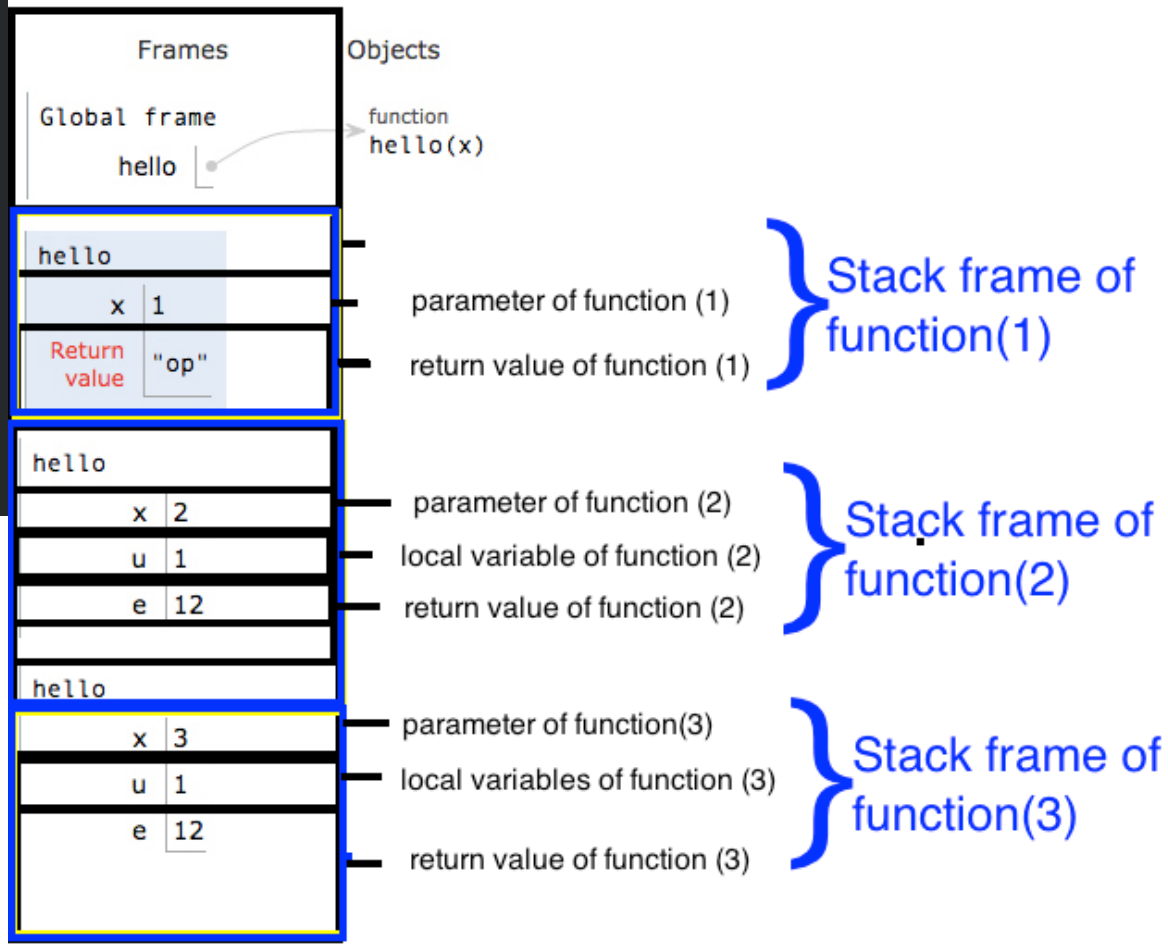



```
1  def hello(x):
2      if x == 0:
3          return
4      else:
5          hello(x-1)
6
7  hello(9999999)
```

```
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
File "stack.py", line 5, in hello
  hello(x-1)
RuntimeError: maximum recursion depth exceeded
 quake0day@quakes-iMac ~
```

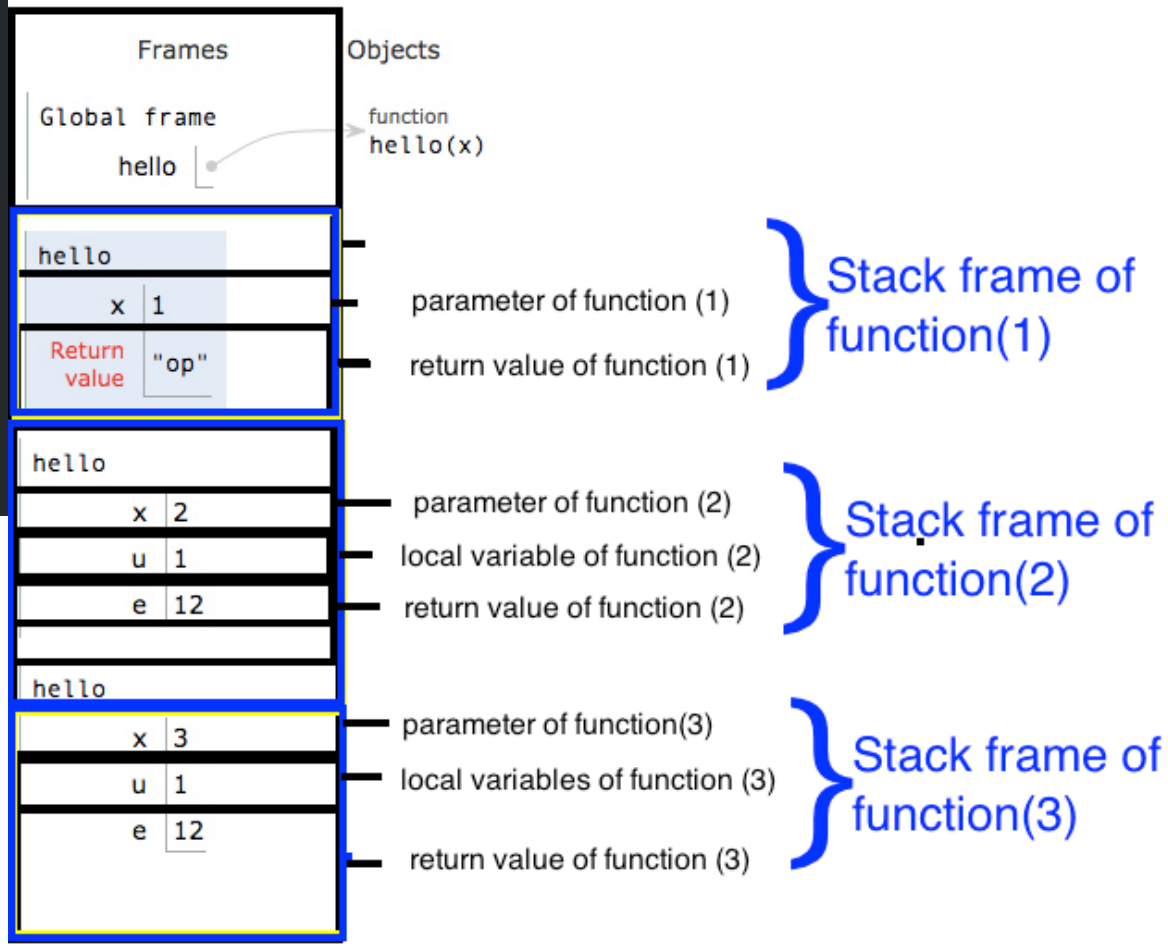
Stack Frame

```
1  def hello(x):
2      if x == 1:
3          return "op"
4      else:
5          u = 1
6          e = 12
7          s = hello(x - 1)
8          e += 1
9          print(s)
10         print(x)
11         u += 1
12     return e
13
14
15 hello(3)
```

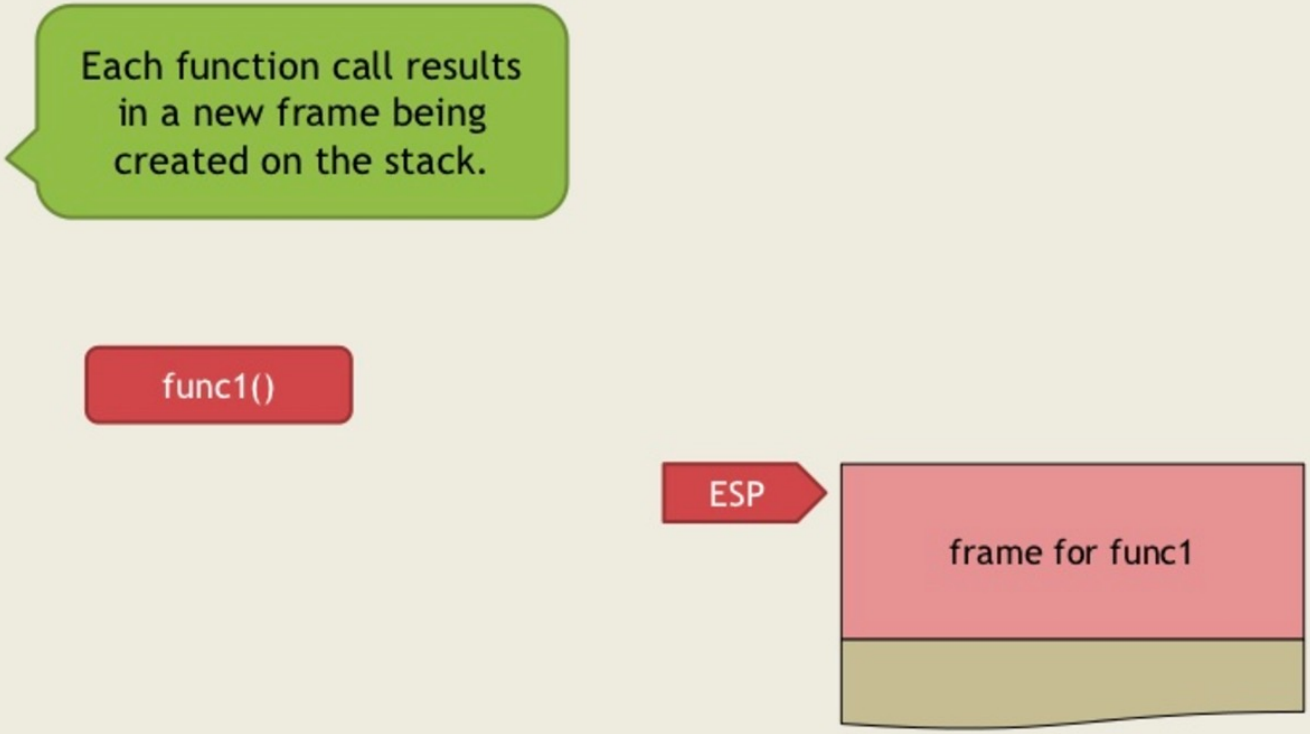


Stack Frame

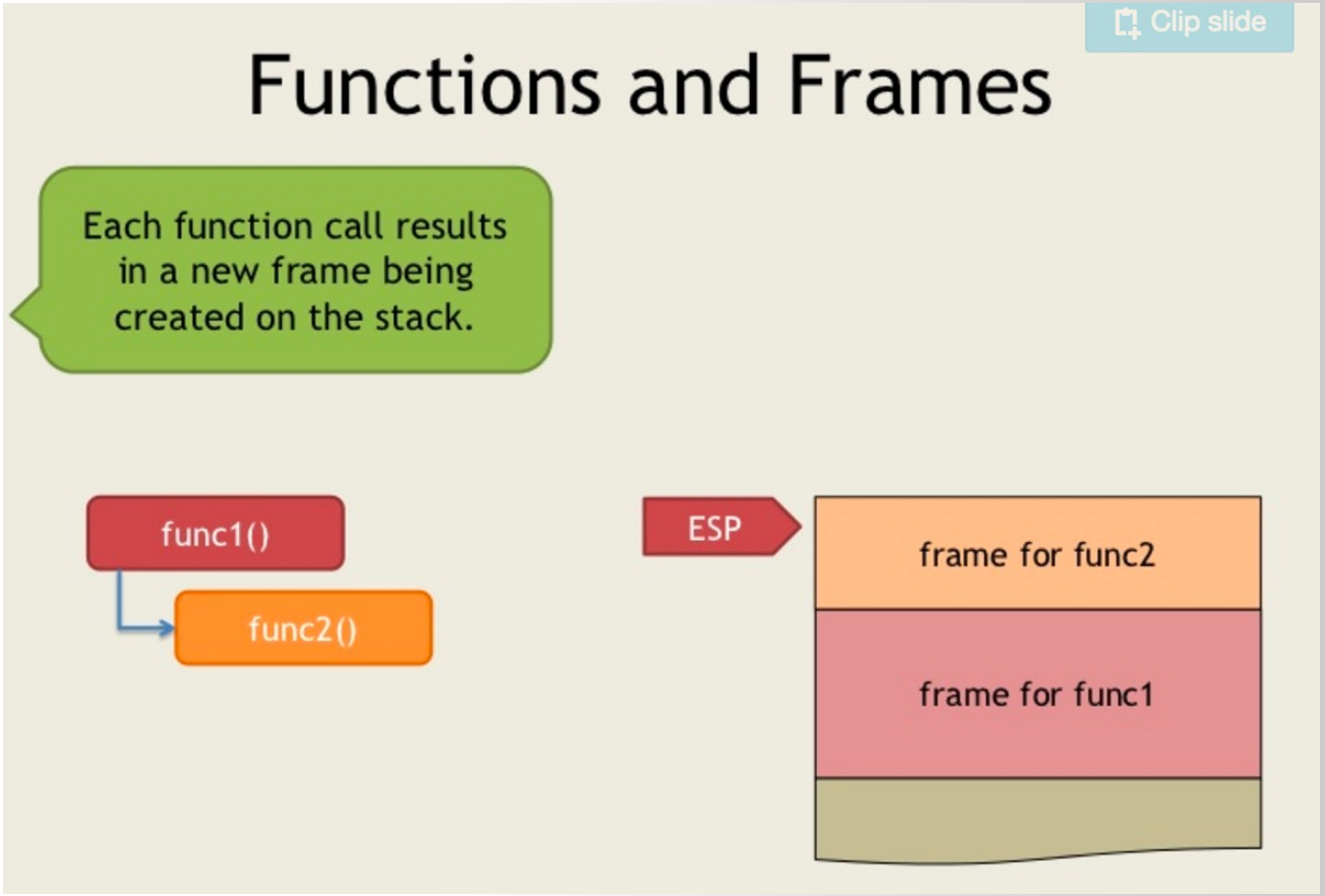
```
1  def hello(x):
2      if x == 1:
3          return "op"
4      else:
5          u = 1
6          e = 12
7          s = hello(x - 1)
8          e += 1
9          print(s)
10         print(x)
11         u += 1
12     return e
13
14
15 hello(3)
```



Functions and Frames

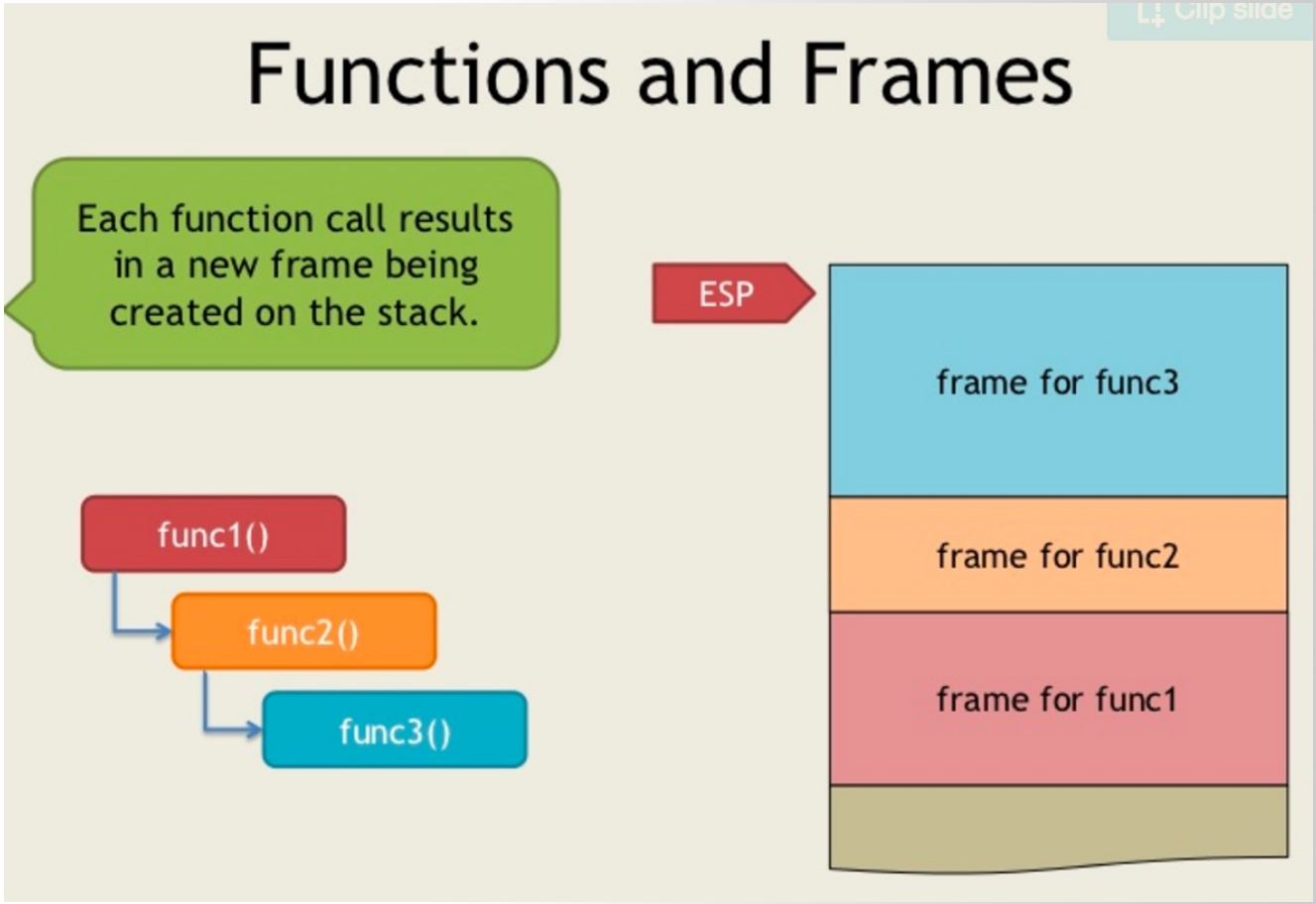


https://www.slideshare.net/saumilshah/how-functions-work-7776073



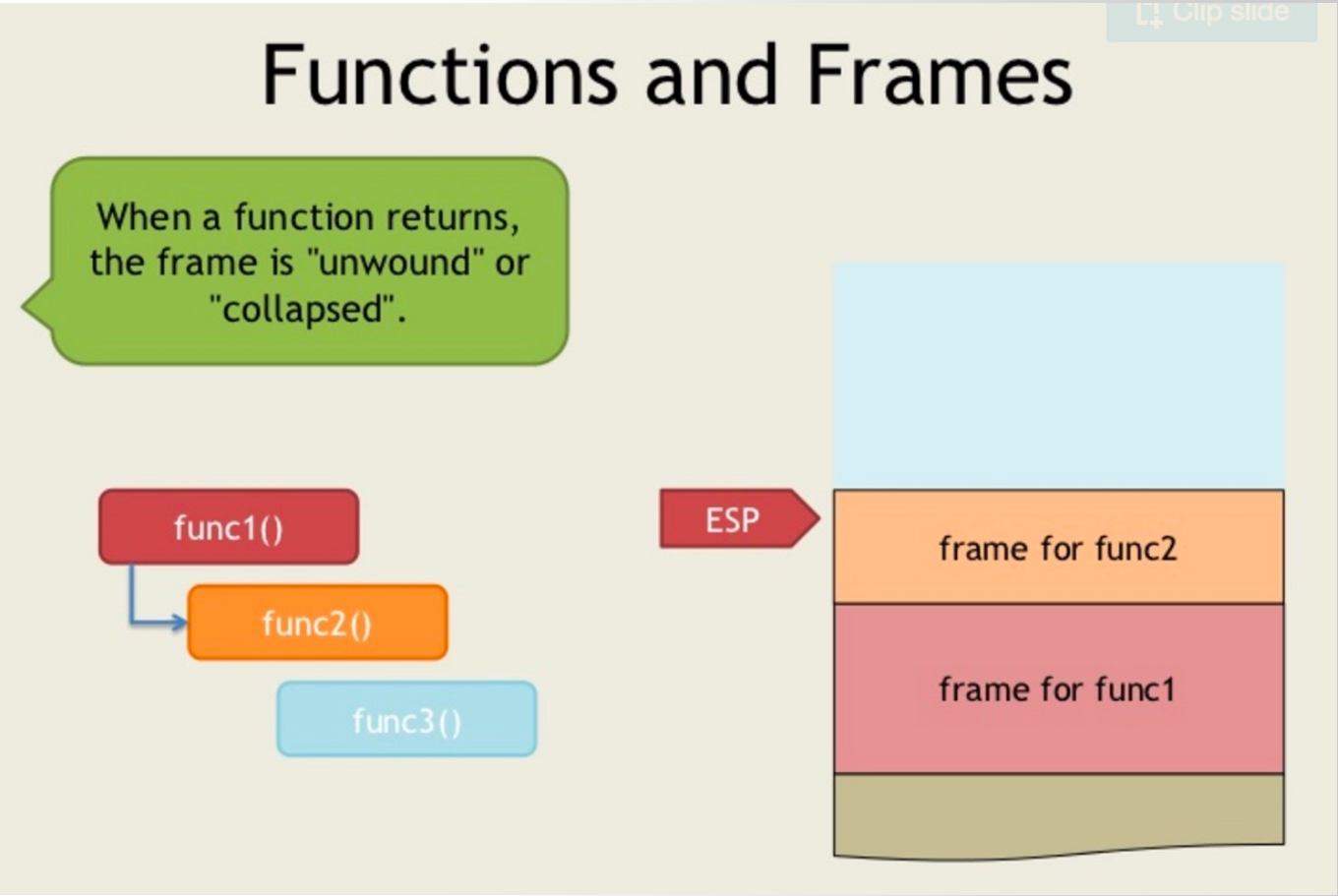
https://www.slideshare.net/saumilshah/how-functions-work-7776073

Functions and Frames



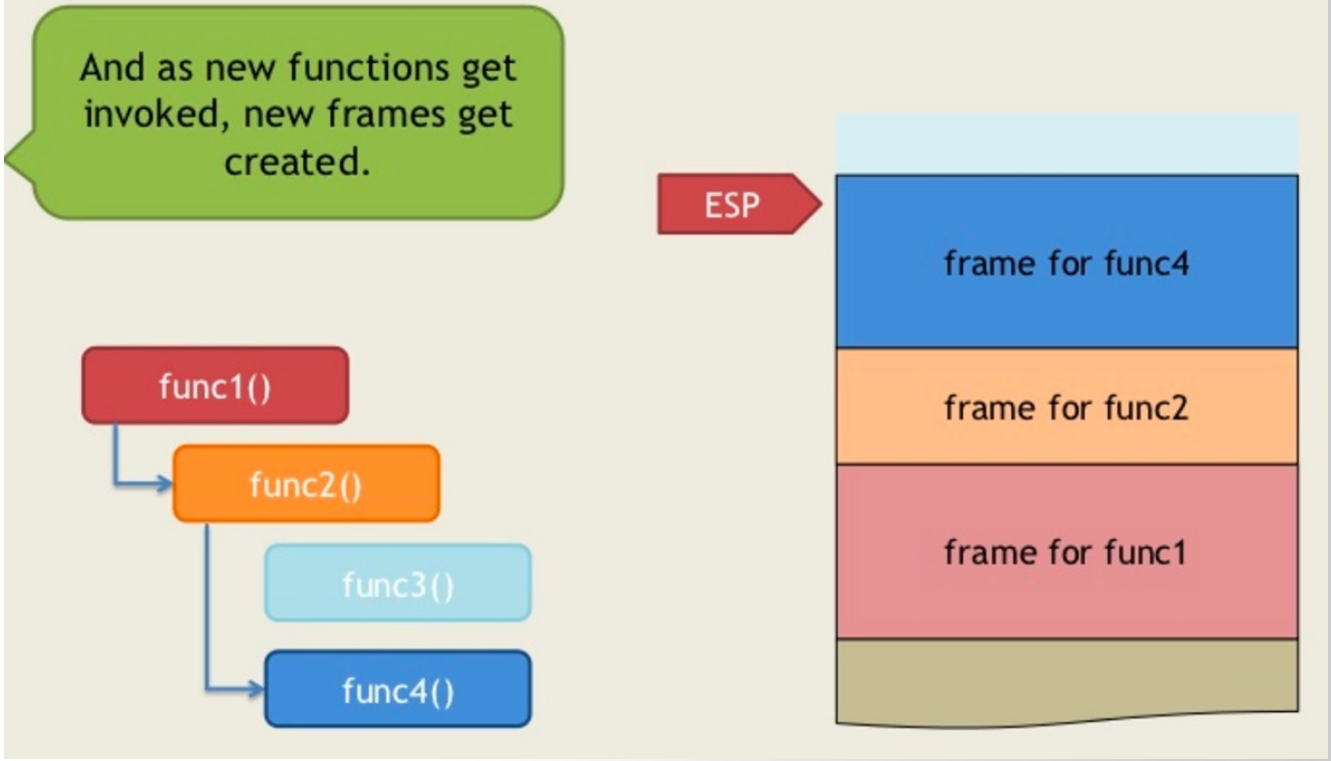
https://www.slideshare.net/saumilshah/how-functions-work-7776073

Functions and Frames



https://www.slideshare.net/saumilshah/how-functions-work-7776073

Functions and Frames



https://www.slideshare.net/saumilshah/how-functions-work-7776073

Stack Frame

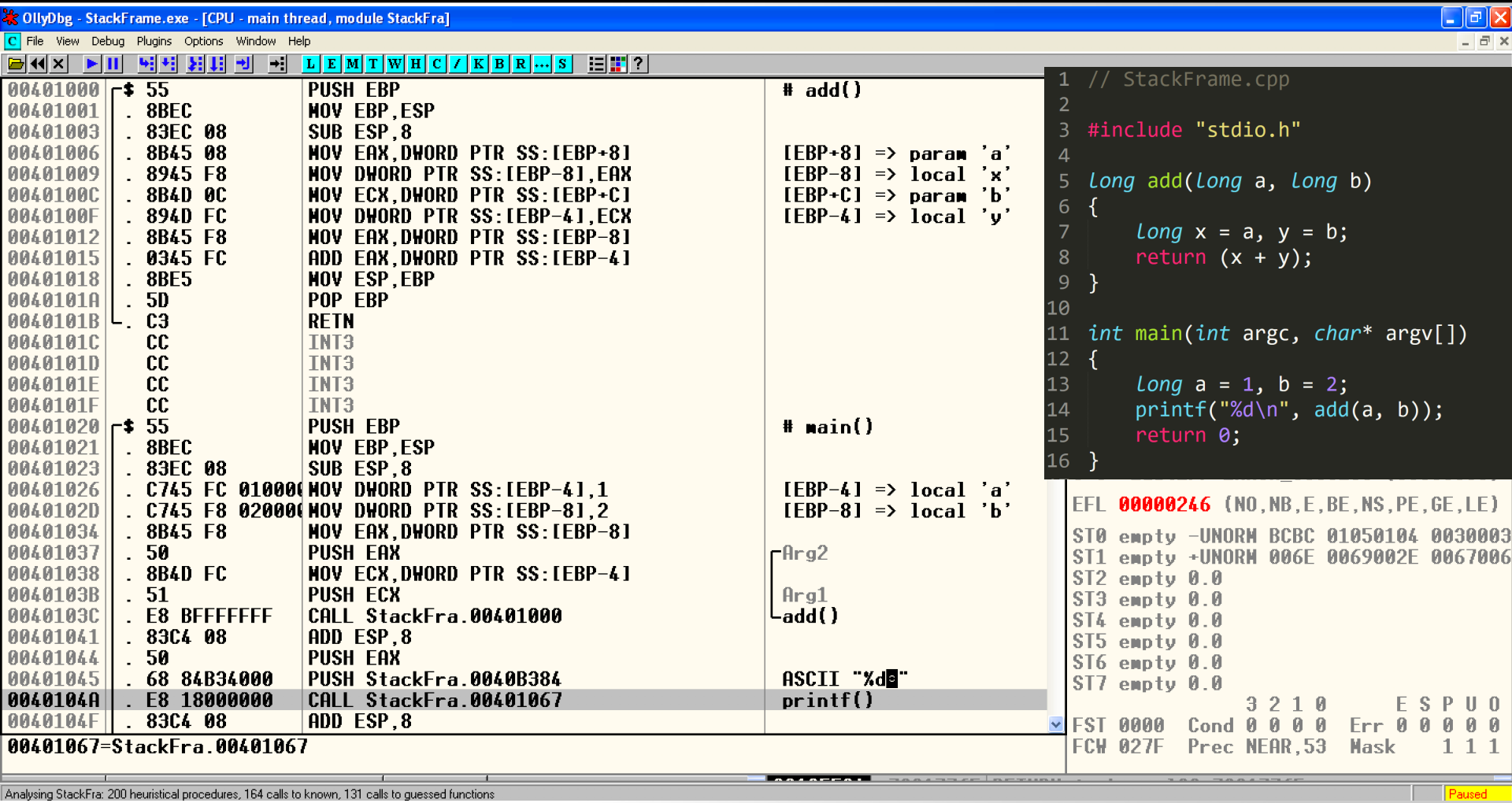
File Edit View Terminal Tabs Help

```
PUSH EBP          ; start of the func (save current EBP to stack)
MOV EBP, ESP      ; save current ESP to EBP

....              ; function body
                  ; no matter how ESP changes, the EBP remains unchanged


MOV ESP, EBP      ; move the saved function start addr back to ESP
POP EBP           ; before return the func, pop the stored EBP
RETN              ; end of the func

~
~
~
~
~
~
~
~
~
~
~
-- INSERT --                                    12,1          All
```

StackFrame.exe

```
1  // StackFrame.cpp
2
3  #include "stdio.h"
4
5  long add(long a, long b)
6  {
7      long x = a, y = b;
8      return (x + y);
9  }
10
11 int main(int argc, char* argv[])
12 {
13     long a = 1, b = 2;
14     printf("%d\n", add(a, b));
15     return 0;
16 }
```

StackFrame.exe

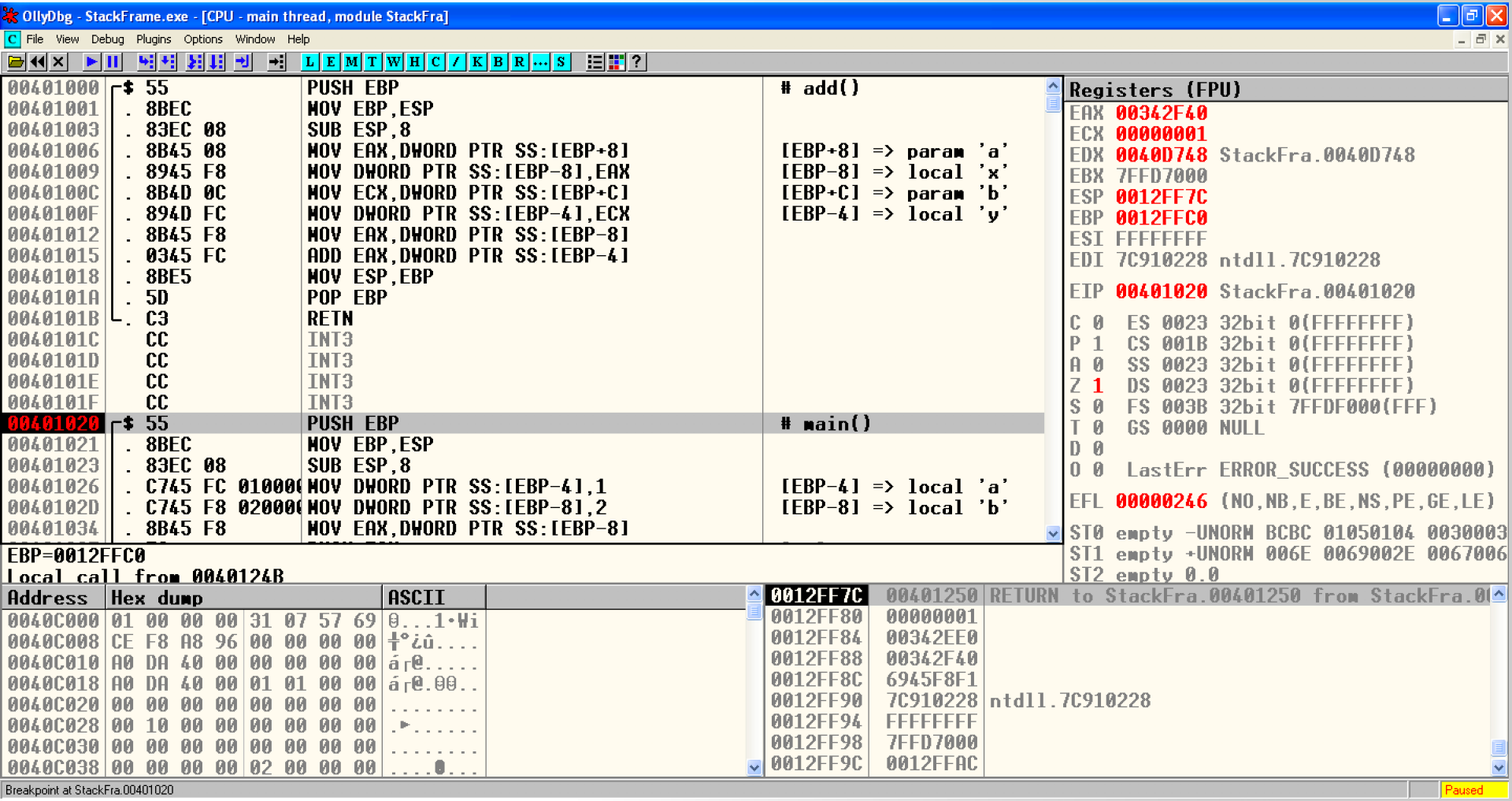


EBP - n: Local vars
EBP + n: Parameters

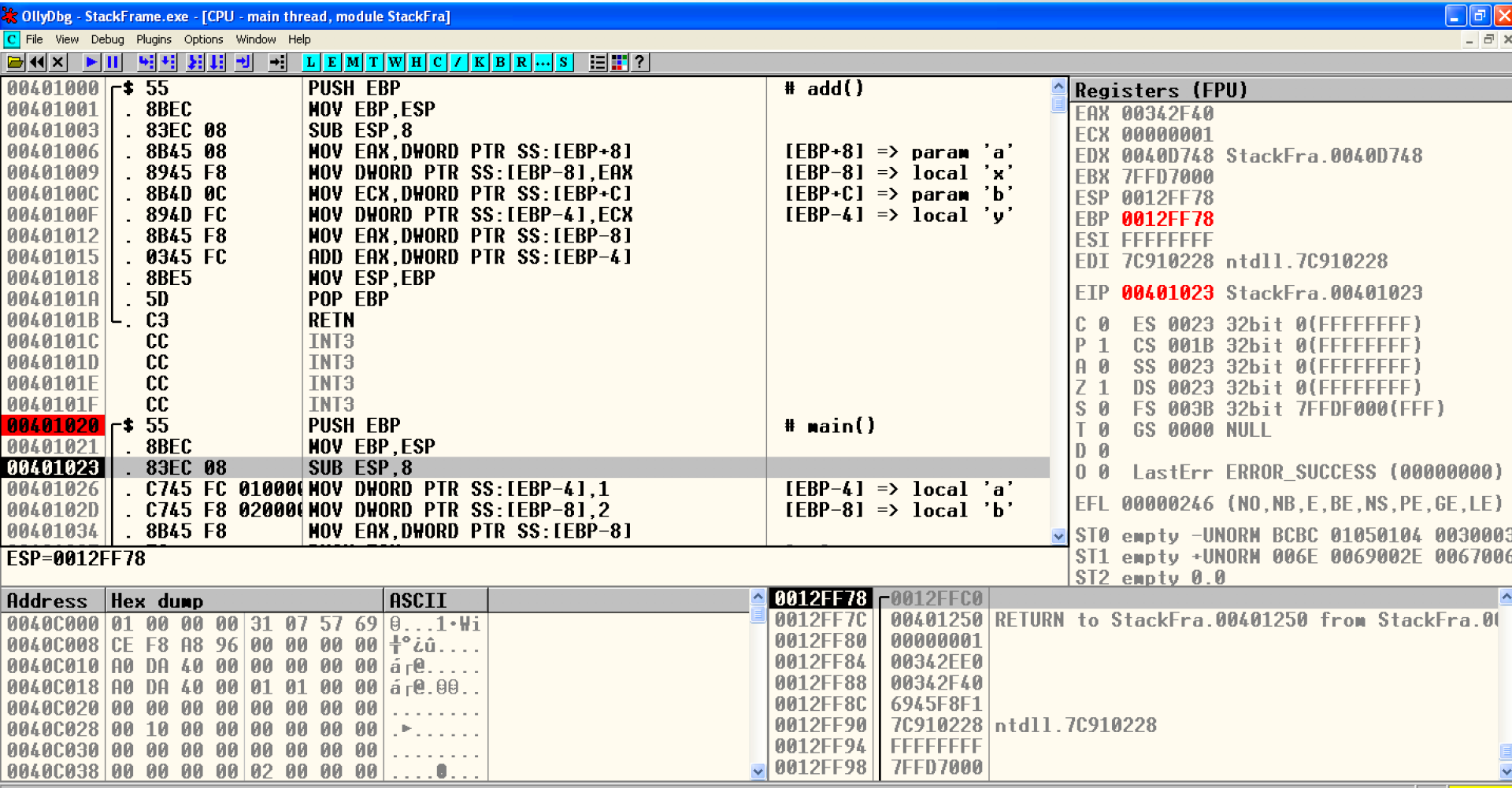
StackFrame.exe

```
 1  // StackFrame.cpp
 2
 3  #include "stdio.h"
 4
 5  long add(long a, long b)
 6  {
 7      long x = a, y = b;
 8      return (x + y);
 9  }
10
11  int main(int argc, char* argv[])
12  {
13      long a = 1, b = 2;
14      printf("%d\n", add(a, b));
15      return 0;
16  }
```

StackFrame.exe



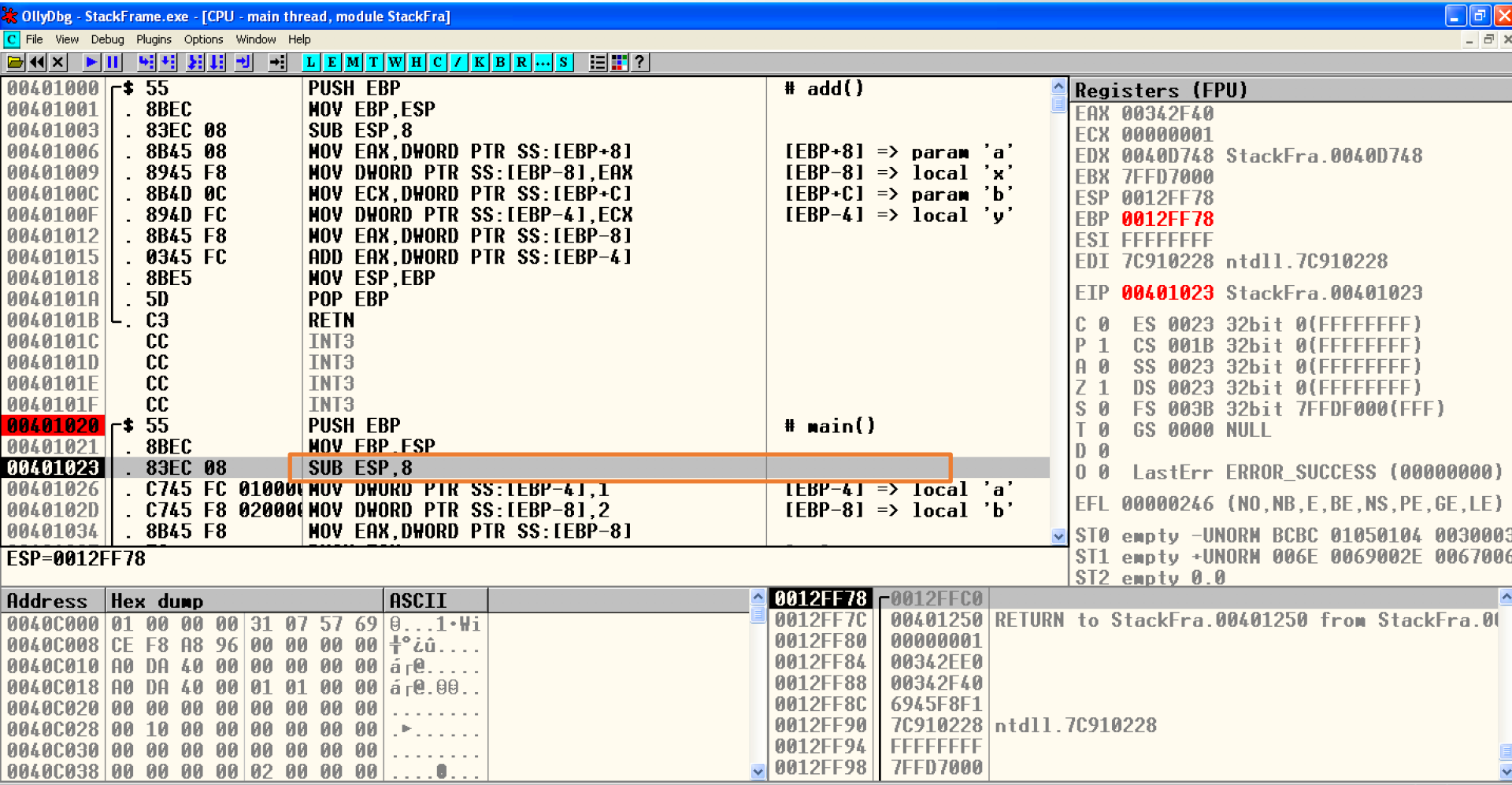
StackFrame.exe



StackFrame.exe

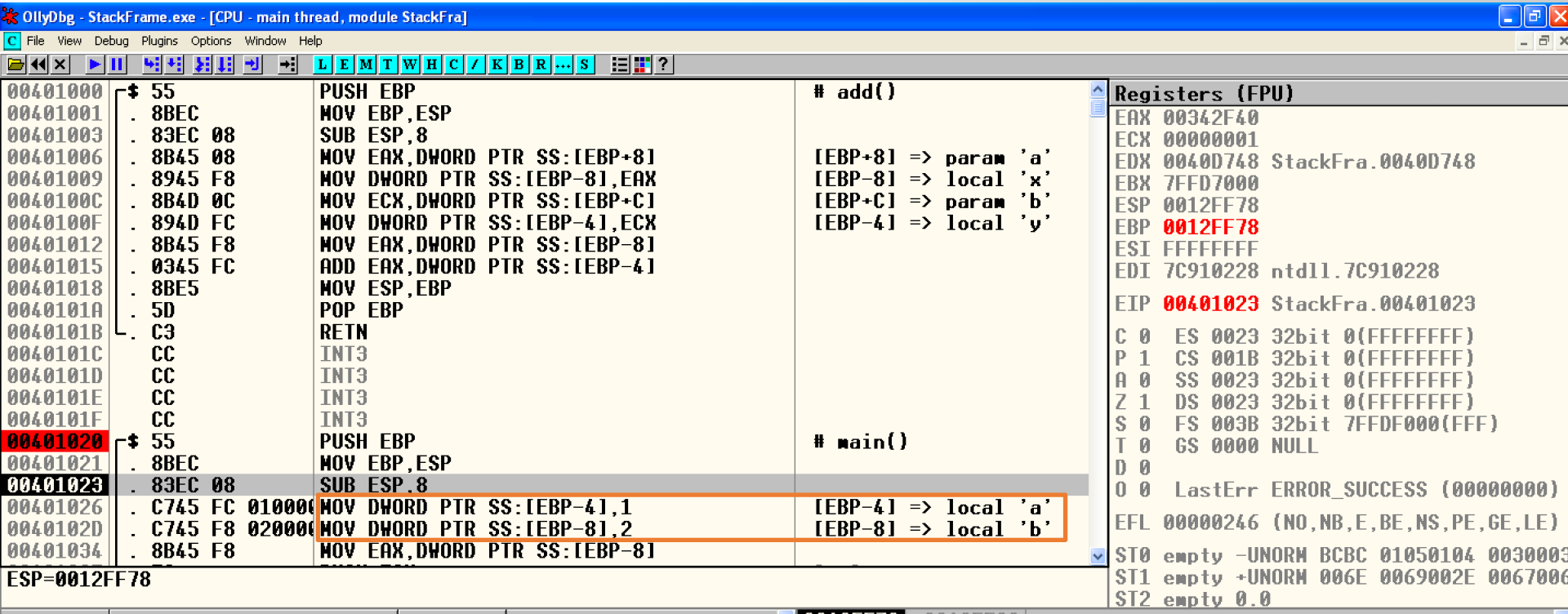
```
 1  // StackFrame.cpp
 2
 3  #include "stdio.h"
 4
 5  long add(long a, long b)
 6  {
 7      long x = a, y = b;
 8      return (x + y);
 9  }
10
11  int main(int argc, char* argv[])
12  {
13      long a = 1, b = 2;
14      printf("%d\n", add(a, b));
15      return 0;
16  }
```

StackFrame.exe



Create space for ‘a’ and ‘b’ → long → 4 byte

StackFrame.exe



Create space for ‘a’ and ‘b’ → long → 4 byte

StackFrame.exe

```
00401026 . C745 FC 010000( MOV DWORD PTR SS:[EBP-4],1    [EBP-4] => local 'a'
0040102D . C745 F8 020000( MOV DWORD PTR SS:[EBP-8],2    [EBP-8] => local 'b'
```

Assembly	C	Type Conversion
DWORD PTR SS:[EBP-4]	*(DWORD*)(EBP-4)	DWORD (4 byte)
WORD PTR SS:[EBP-4]	*(WORD*)(EBP-4)	WORD (2 byte)
BYTE PTR SS:[EBP-4]	*(BYTE*)(EBP-4)	1 byte

4 Byte memory space at address [EBP-4]

```
0012FF70  00000002
0012FF74  00000001
0012FF78  0012FFC0
0012FF7C  00401250  RETURN to StackFra.00401250 from StackFra.0(
0012FF80  00000001
0012FF84  00342EE0
0012FF88  00342F40
0012FF8C  6945F8F1
0012FF90  7C910228  ntdll.7C910228
```

StackFrame.exe

```
1  // StackFrame.cpp
2
3  #include "stdio.h"
4
5  Long add(Long a, Long b)
6  {
7      Long x = a, y = b;
8      return (x + y);
9  }
10
11 int main(int argc, char* argv[])
12 {
13     Long a = 1, b = 2;
14     printf("%d\n", add(a, b));
15     return 0;
16 }
```

```
00401034 | . 8B45 F8      | MOV EAX,DWORD PTR SS:[EBP-8]  |
00401037 | . 50           | PUSH EAX                      | ┌Arg2
00401038 | . 8B4D FC      | MOV ECX,DWORD PTR SS:[EBP-4]  |
0040103B | . 51           | PUSH ECX                      | Arg1
0040103C | . E8 BFFFFFFF  | CALL StackFra.00401000        | └add()
```

StackFrame.exe

```
1  // StackFrame.cpp
2
3  #include "stdio.h"
4
5  long add(long a, long b)
6  {
7      long x = a, y = b;
8      return (x + y);
9  }
10
11 int main(int argc, char* argv[])
12 {
13     long a = 1, b = 2;
14     printf("%d\n", add(a, b));
15     return 0;
16 }
```

```
00401000 r$ 55       PUSH EBP          # add()
00401001 |. 8BEC     MOV EBP,ESP
```

StackFrame.exe

```
1  // StackFrame.cpp
2
3  #include "stdio.h"
4
5  Long add(Long a, Long b)
6  {
7      Long x = a, y = b;
8      return (x + y);
9  }
10
11 int main(int argc, char* argv[])
12 {
13     Long a = 1, b = 2;
14     printf("%d\n", add(a, b));
15     return 0;
16 }
```

```
00401000 r$ 55       PUSH EBP              # add()
00401001 .  8BEC     MOV EBP,ESP
00401003 .  83EC 08  SUB ESP,8
00401006 .  8B45 08  MOV EAX,DWORD PTR SS:[EBP+8]   [EBP+8] => param 'a'
00401009 .  8945 F8  MOV DWORD PTR SS:[EBP-8],EAX   [EBP-8] => local 'x'
0040100C .  8B4D 0C  MOV ECX,DWORD PTR SS:[EBP+C]   [EBP+C] => param 'b'
0040100F .  894D FC  MOV DWORD PTR SS:[EBP-4],ECX   [EBP-4] => local 'y'
```

StackFrame.exe

```
1  // StackFrame.cpp
2
3  #include "stdio.h"
4
5  Long add(Long a, Long b)
6  {
7      Long x = a, y = b;
8      return (x + y);
9  }
10
11 int main(int argc, char* argv[])
12 {
13     Long a = 1, b = 2;
14     printf("%d\n", add(a, b));
15     return 0;
16 }
```

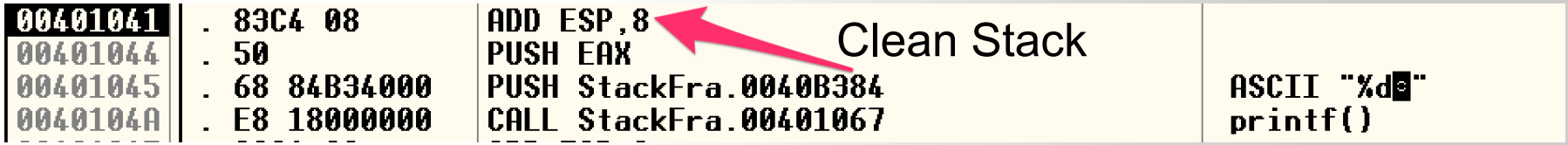
```
00401012 | . 8B45 F8      | MOV EAX,DWORD PTR SS:[EBP-8]
00401015 | . 0345 FC      | ADD EAX,DWORD PTR SS:[EBP-4]
```

StackFrame.exe

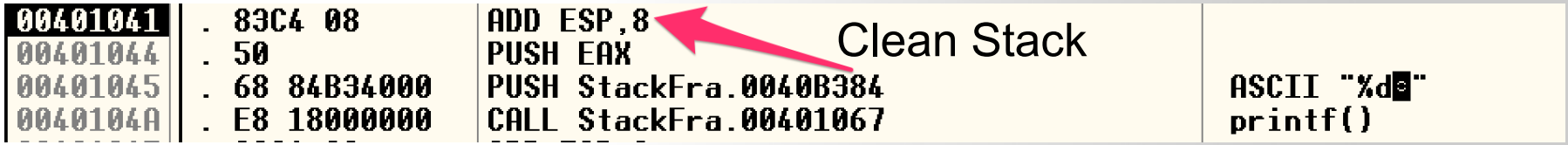
```
1  // StackFrame.cpp
2
3  #include "stdio.h"
4
5  Long add(Long a, Long b)
6  {
7      Long x = a, y = b;
8      return (x + y);
9  }
10
11 int main(int argc, char* argv[])
12 {
13     Long a = 1, b = 2;
14     printf("%d\n", add(a, b));
15     return 0;
16 }
```

```
00401018  |. 8BE5          MOV ESP,EBP
0040101A  |. 5D            POP EBP
0040101B  L. C3            RETN
```

StackFrame.exe



StackFrame.exe



StackFrame.exe

```
1  // StackFrame.cpp
2
3  #include "stdio.h"
4
5  Long add(Long a, Long b)
6  {
7      Long x = a, y = b;
8      return (x + y);
9  }
10
11 int main(int argc, char* argv[])
12 {
13     Long a = 1, b = 2;
14     printf("%d\n", add(a, b));
15     return 0;
16 }
```

```
00401044 | . 50           | PUSH EAX
00401045 | . 68 84B34000  | PUSH StackFra.0040B384   | ASCII "%d"
0040104A | . E8 18000000  | CALL StackFra.00401067   | printf()
0040104F | . 83C4 08      | ADD ESP,8
```

StackFrame.exe

```
1  // StackFrame.cpp
2
3  #include "stdio.h"
4
5  Long add(Long a, Long b)
6  {
7      Long x = a, y = b;
8      return (x + y);
9  }
10
11 int main(int argc, char* argv[])
12 {
13     Long a = 1, b = 2;
14     printf("%d\n", add(a, b));
15     return 0;
16 }
```

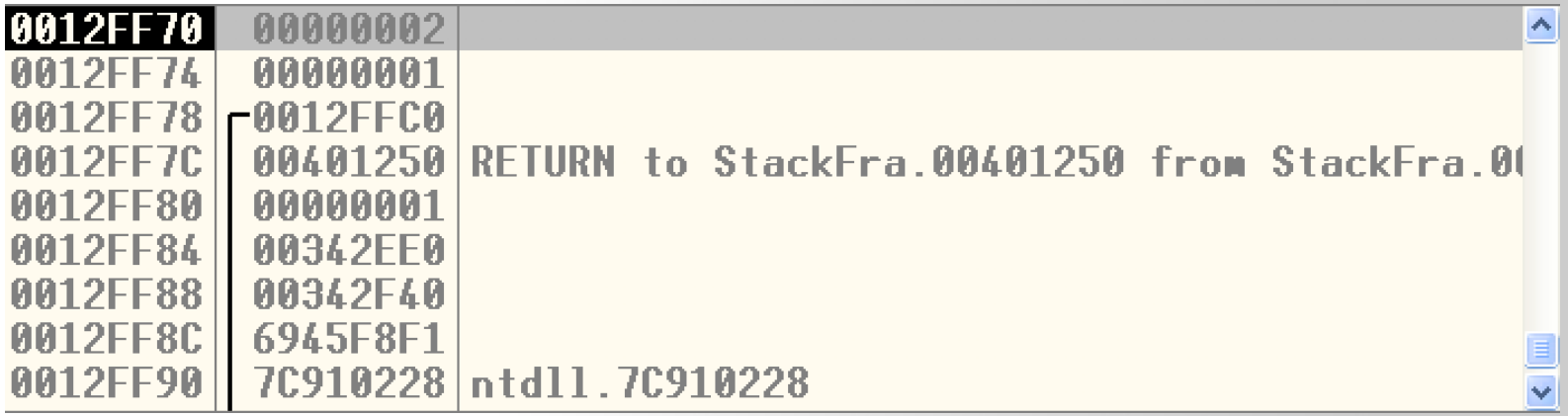
```
00401052 . 33C0      XOR EAX,EAX
00401054 . 8BE5      MOV ESP,EBP
00401056 . 5D        POP EBP
```

Set EAX –> 0
Faster than
MOV EAX,0

Calling Convention

Two Questions

- Q: When a function finished, how to handle the parameter left in the stack.



A: We don't care…

- Q: When a function finished, how change the ESP value?

  A: ESP should be restored to the previous value

Standard C Calling Conventions

- **Calling conventions** are a standardized method for functions to be implemented and called by the machine.
- A calling convention specifies the method that a compiler sets up to access a subroutine.
- There are three major calling conventions that are used with the C language on 32-bit x86 processors:
  - CDECL
  - STDCALL,
  - FASTCALL.

CDECL

- The C language, by default, uses the CDECL calling convention
- In the CDECL calling convention the following holds:
  - Arguments are passed on the stack in Right-to-Left order, and return values are passed in eax.
  - The ***calling* function cleans the stack**. This allows CDECL functions to have *variable-length argument lists*.

STDCALL

- The C language, by default, uses the CDECL calling convention
- In the CDECL calling convention the following holds:
  - Arguments are passed on the stack in Right-to-Left order, and return values are passed in eax.
  - The ***calling* function cleans the stack**. This allows CDECL functions to have *variable-length argument lists*.

```
_cdecl int MyFunction1(int a, int b)
{
  return a + b;
}
```

and the following function call:

```
x = MyFunction1(2, 3);
```

These would produce the following assembly listings, respectively:

```
_MyFunction1:
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret
```

and

```
push 3
push 2
call _MyFunction1
add esp, 8
```

STDCALL

- STDCALL, also known as "WINAPI" (and a few other names, depending on where you are reading it) is used almost exclusively by Microsoft as the standard calling convention for the Win32 API.
  - STDCALL passes arguments right-to-left, and returns the value in eax.
  - The **called function cleans the stack**, unlike CDECL. This means that STDCALL doesn't allow variable-length argument lists.

STDCALL

- STDCALL, also known as "WINAPI" (and a few other names, depending on where you are reading it) is used almost exclusively by Microsoft as the standard calling convention for the Win32 API.
  - STDCALL passes arguments right-to-left, and returns the value in eax.
  - The **called function cleans the stack**, unlike CDECL. This means that STDCALL doesn't allow variable-length argument lists.

RET 8 ➔ RET + POP 8 Byte

Consider the following C function:

```
_stdcall int MyFunction2(int a, int b)
{
   return a + b;
}
```

and the calling instruction:

```
x = MyFunction2(2, 3);
```

These will produce the following respective assembly code fragm

```
:_MyFunction2@8
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret 8
```

and

```
push 3
push 2
call _MyFunction2@8
```

FASTCALL

- The FASTCALL calling convention **is not completely standard** across all compilers, so it should be used with caution.
- The calling function most frequently is responsible for cleaning the stack, if needed.

Q & A