

# **CSCI 2410 Systems Programming**

## **Fall 2026**

**Instructor: Jie Zhou**

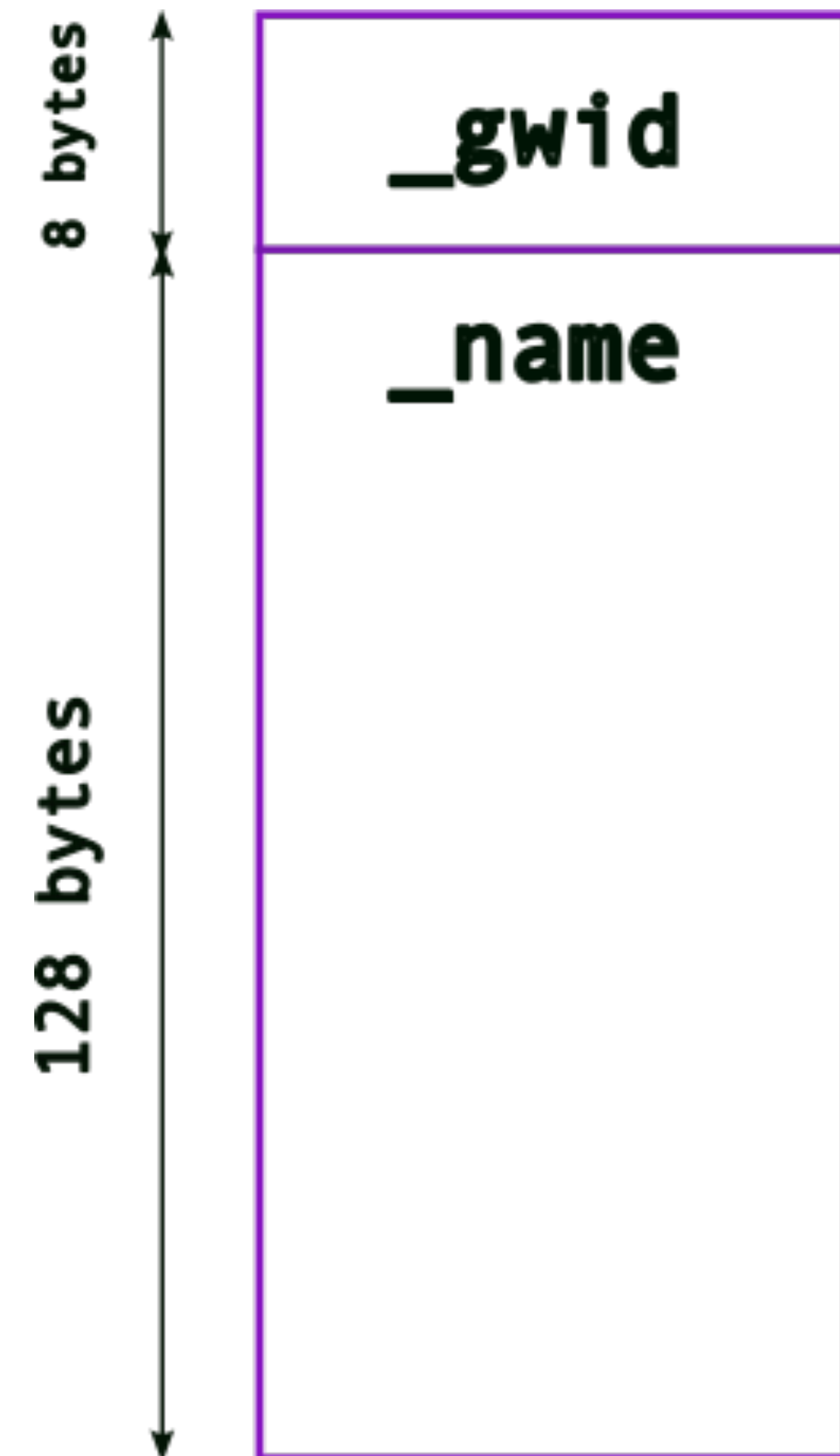
**Department of Computer Science**  
**George Washington University**



Slides materials are mostly credited to Gabriel Parmer and Sibin Mohan at GWU.

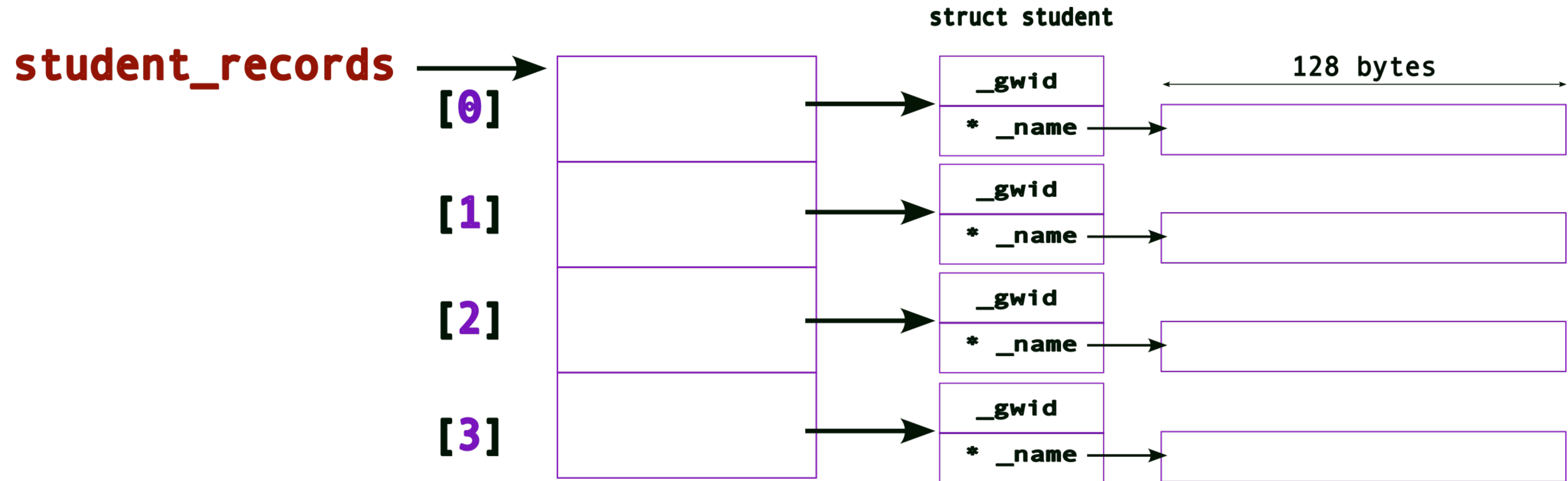
# Memory Layout of Objects

```
struct student{  
    unsigned long _gwid;  
    char _name [128];  
};
```



- `_name [128]` have a fixed size.
- This can be problematic.
  - Could be not long enough for certain names
  - May be too long for most names. Waste of memory!
    - If most names take up to 64 bytes, 100,000 records would waste  $100,000 * 64 = 6,400,000$  6.4 million bytes!

# Final Memory Layout

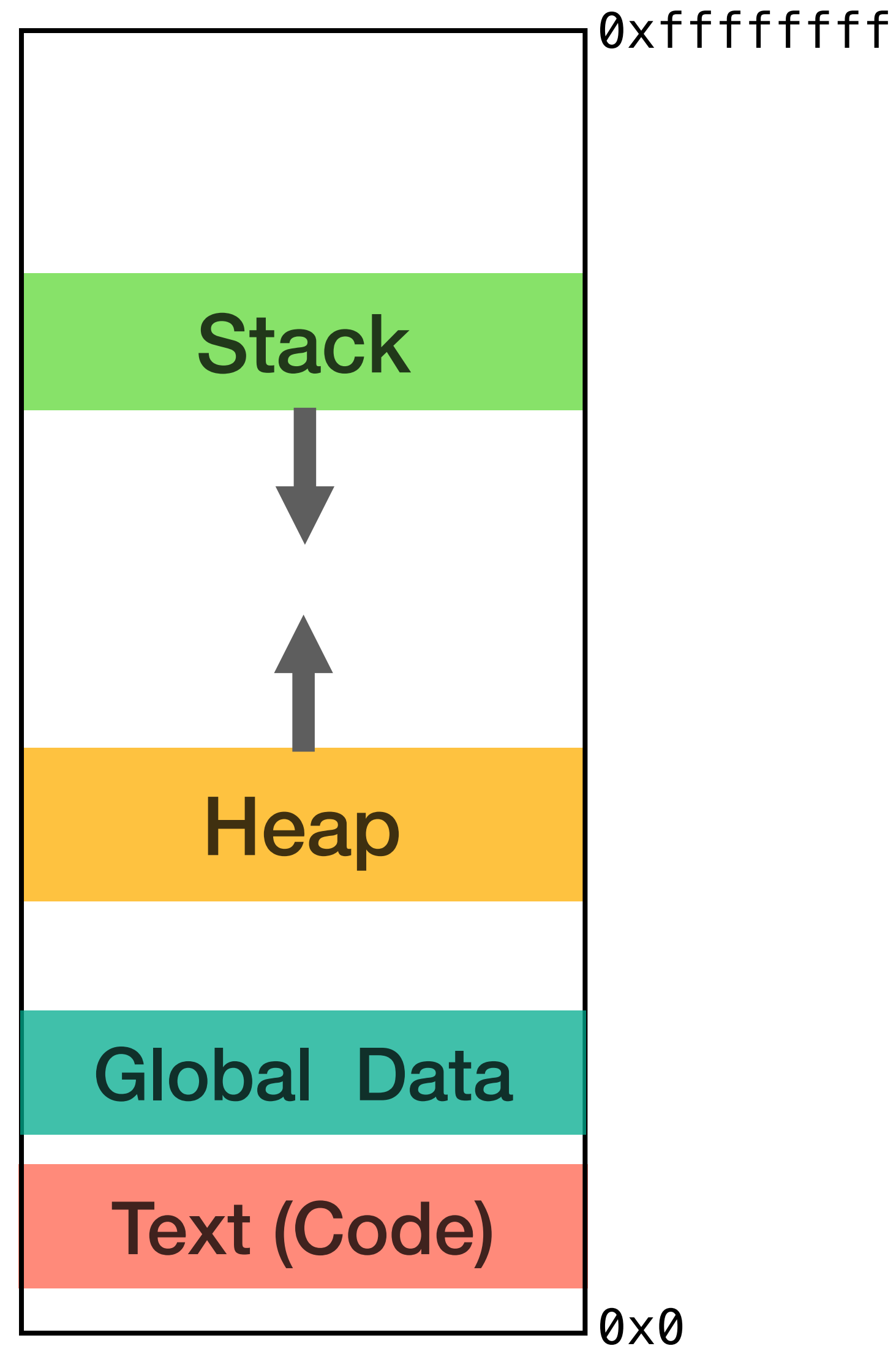


- Now we have pointers to pointers, i.e., double/multi-level pointers!

# Memory Layout

What do programs need in memory?

- Code
- Data
  - Globals
  - Stack for local variables
  - Heap for dynamic memory



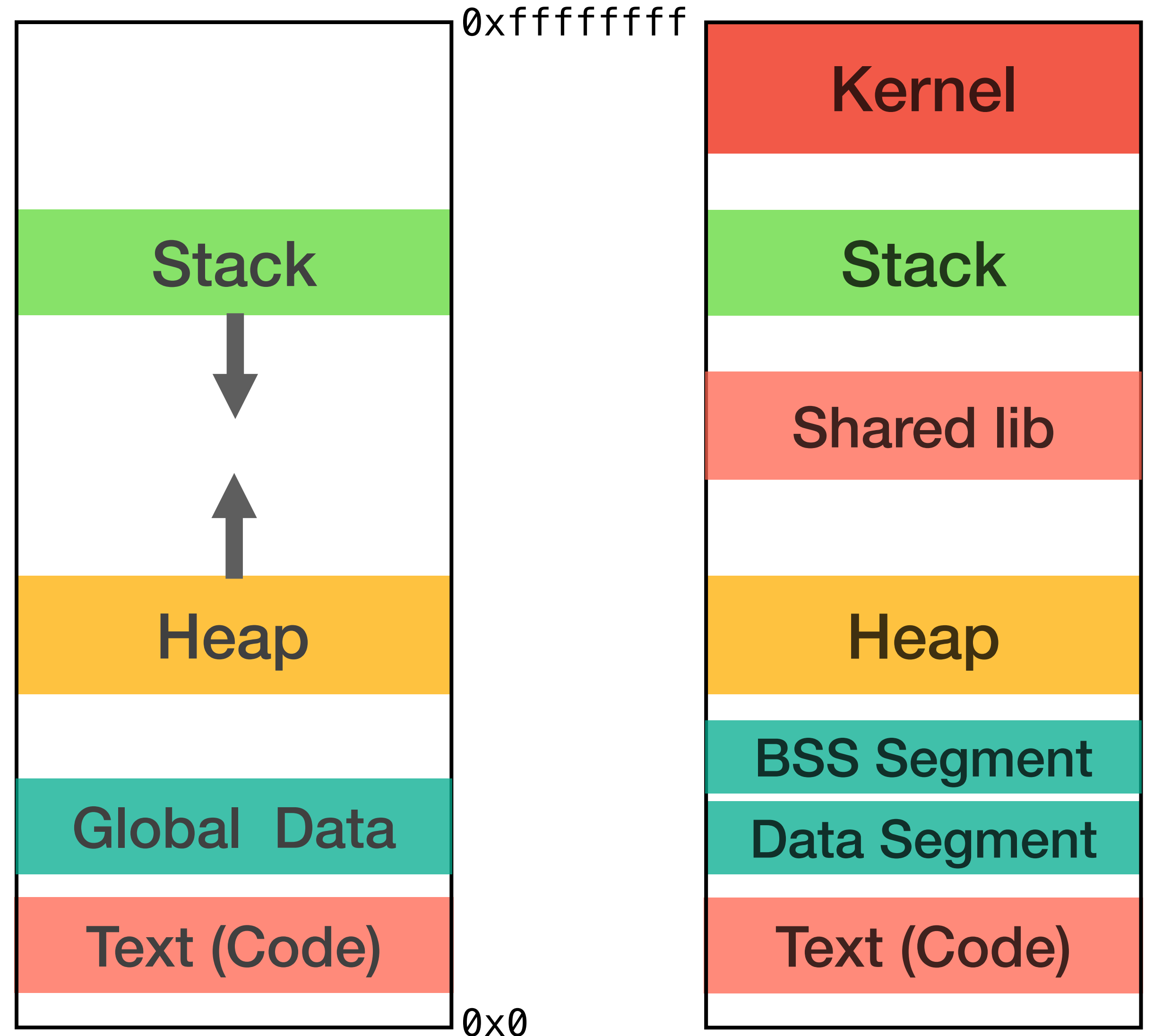
# Address Space of

What do programs need in memory?

- Code
- Data Segment
  - Initialized global variables
- BSS Segment
  - Uninitialized global data
- Heap
- Shared libraries
- Stack
- Kernel



Check “/proc/pid/maps” to see how memory mapping looks in a real system.



# Stack

- Consider this code:

```
1 int foo( int count, double d) {  
2     int total = 0 ;  
3  
4     for (unsigned int i = 0 ; i < count ; ++ i)  
5         ++total ;  
6  
7     return total ;  
8 }  
9  
10 int main() {  
11     foo() ;  
12  
13     return 0 ;  
14 }
```

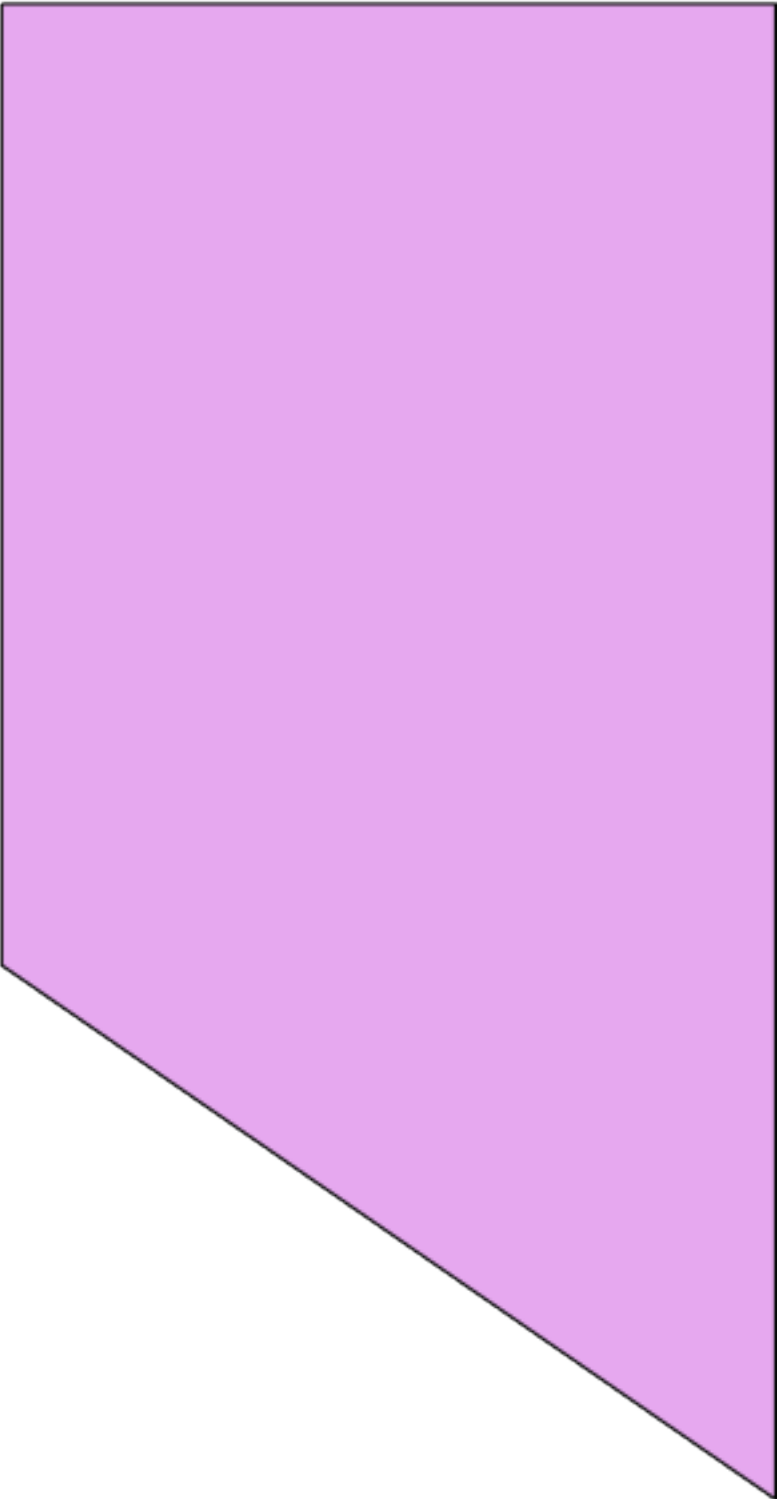


What happens in memory when `main()` calls `foo()`?

# Stack

---

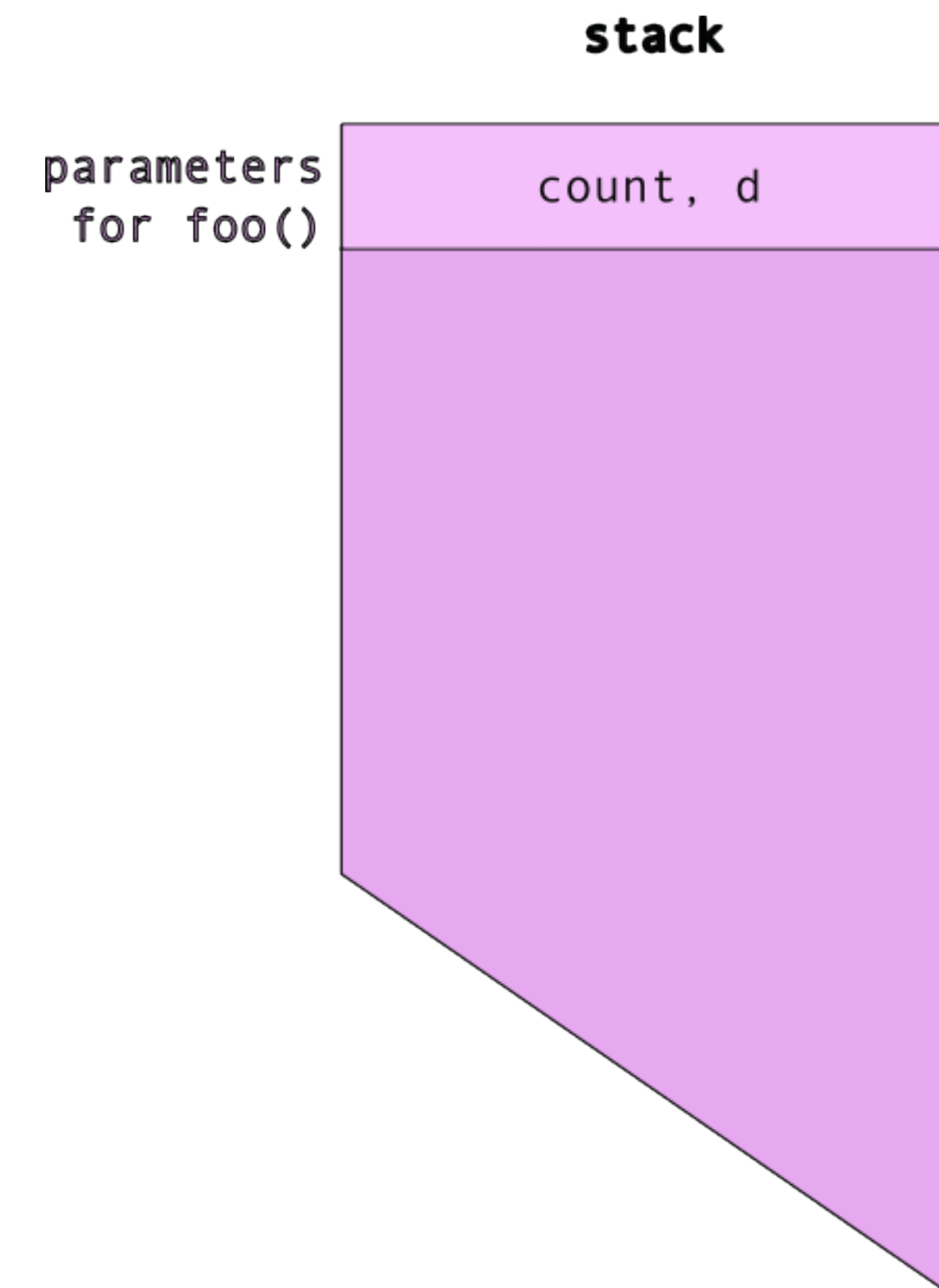
**stack**



# Stack

---

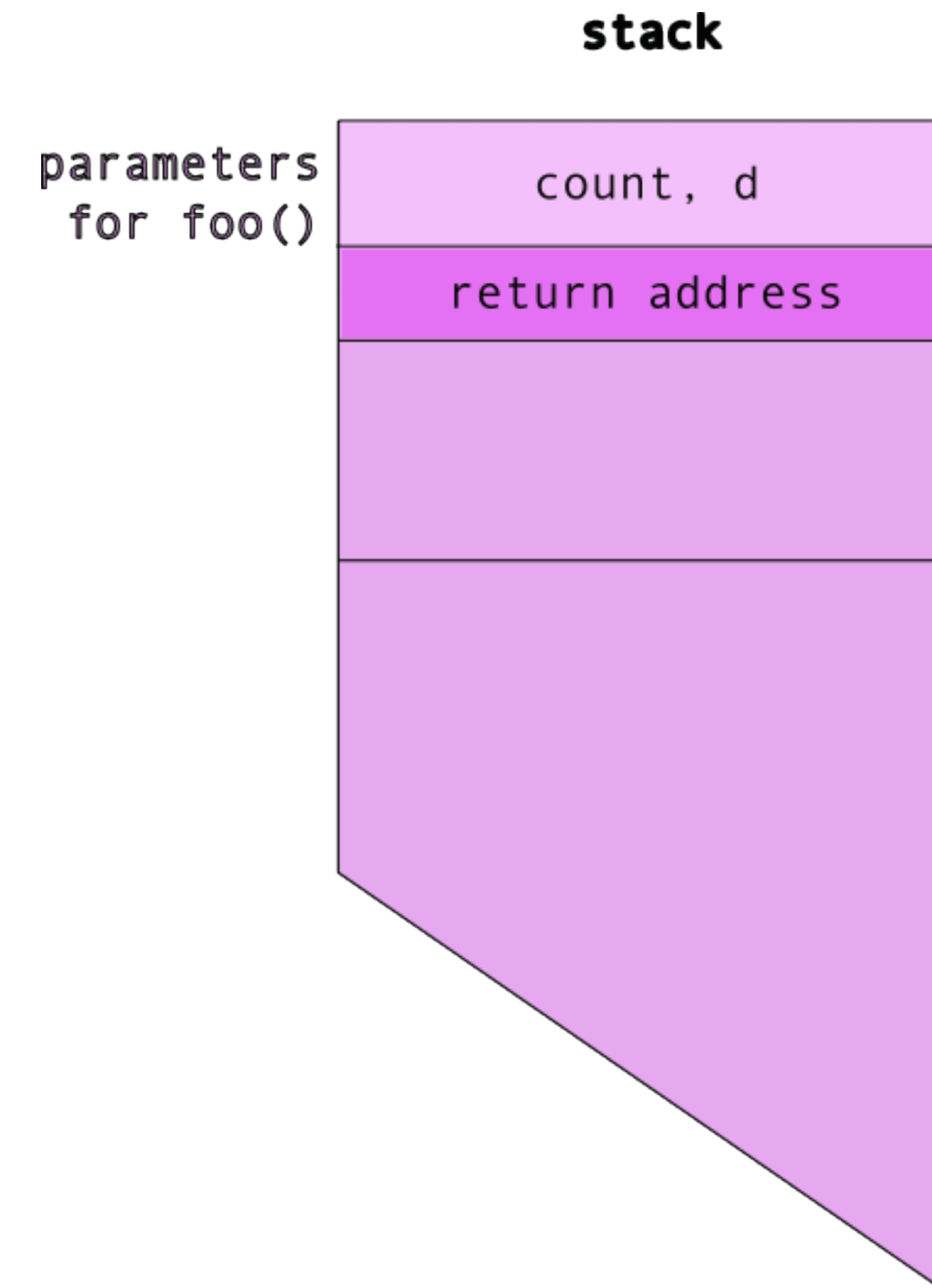
- Function parameters live in registers and stack.
  - Depends on the OS + architecture.



# Stack

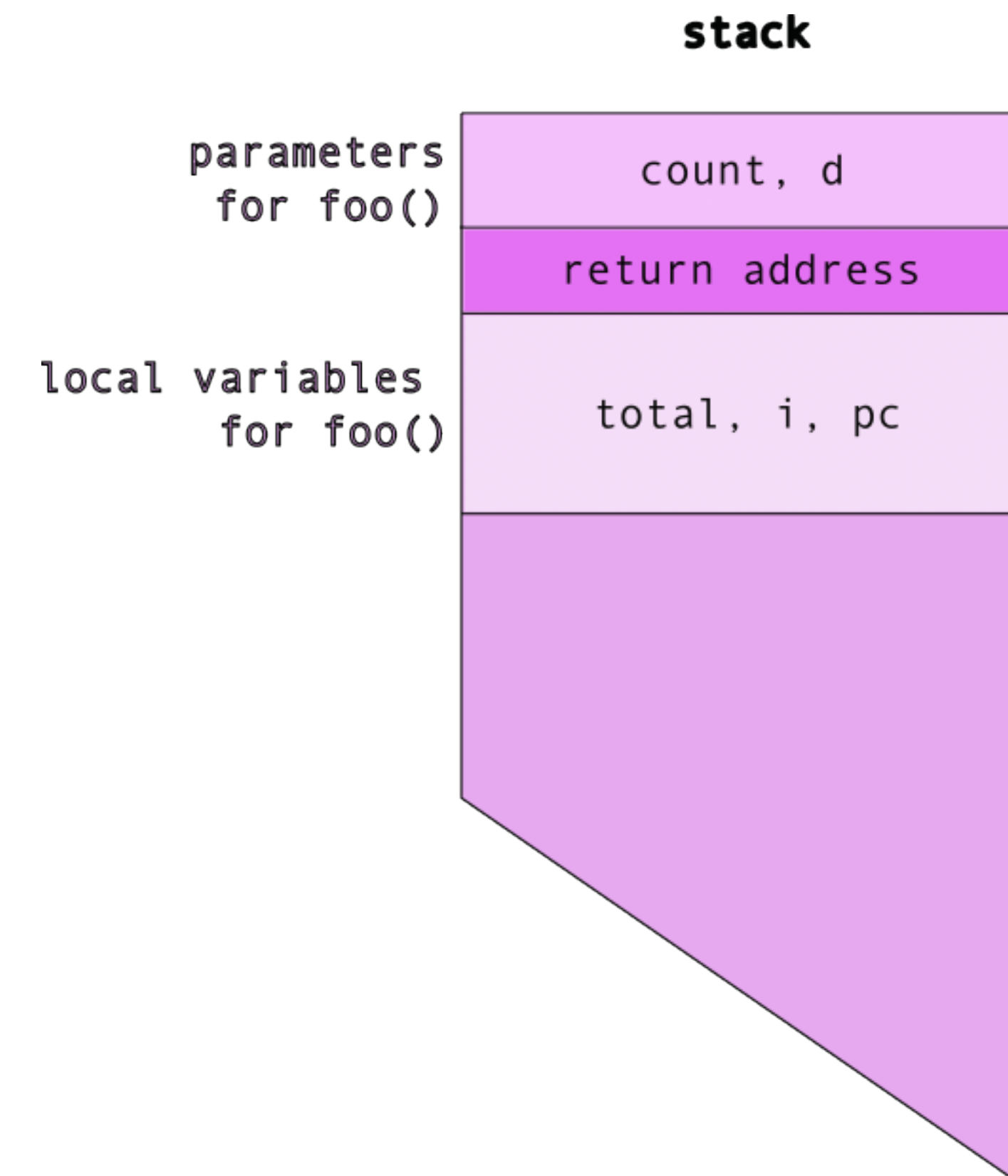
---

- Function parameters live in registers and stack.
  - Depends on the OS + architecture.
- Return address lives on the stack.



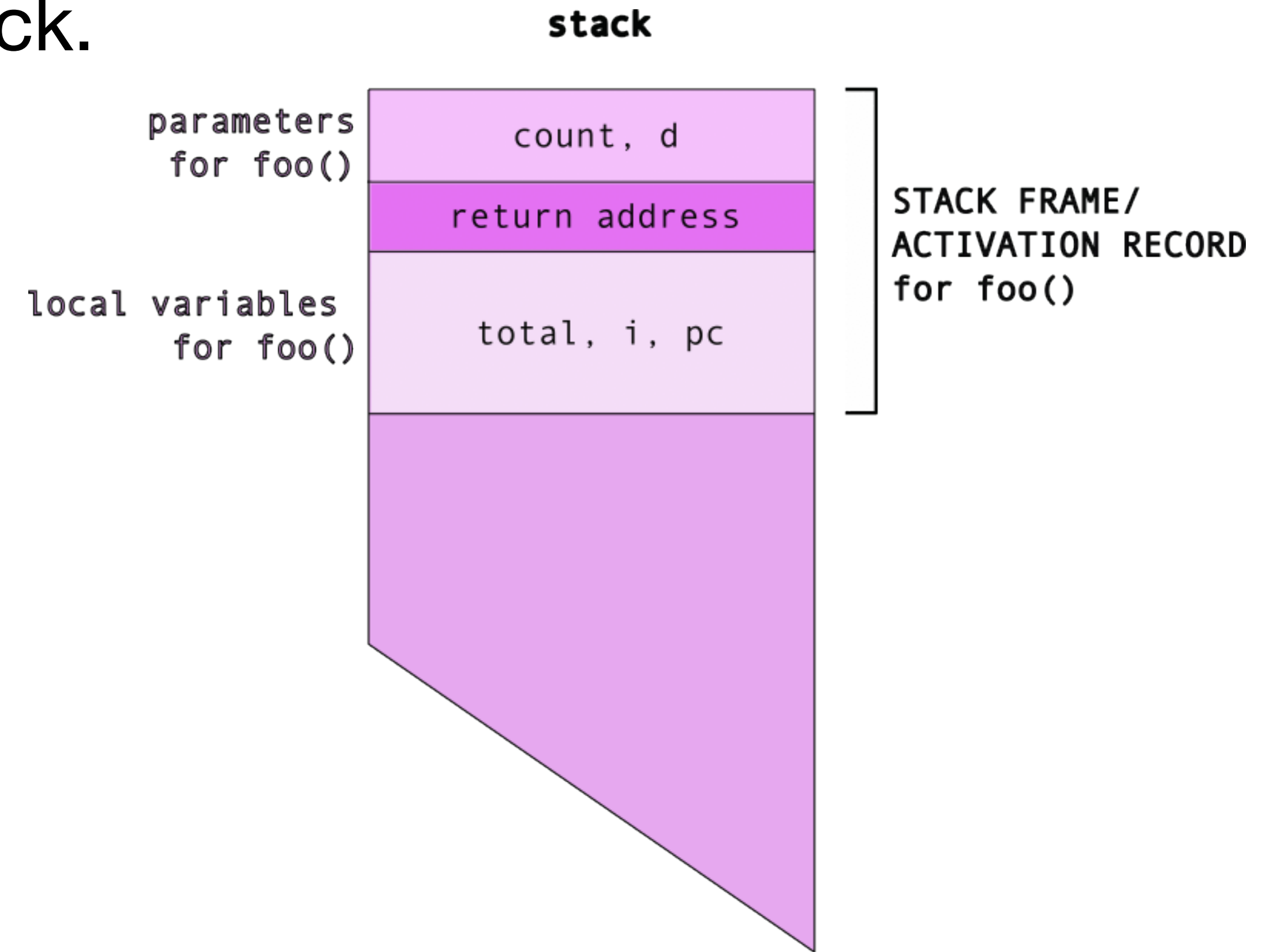
# Stack

- Function parameters live in registers and stack.
  - Depends on the OS + architecture.
- Return address lives on the stack.
- A function's local variables live on the *stack*.



# Stack

- Function parameters live in registers and stack.
  - Depends on the OS + architecture.
- Return address lives on the stack.
- A function's local variables live on the *stack*.
- The space used by a function is called stack frame or activation record.



# Stack

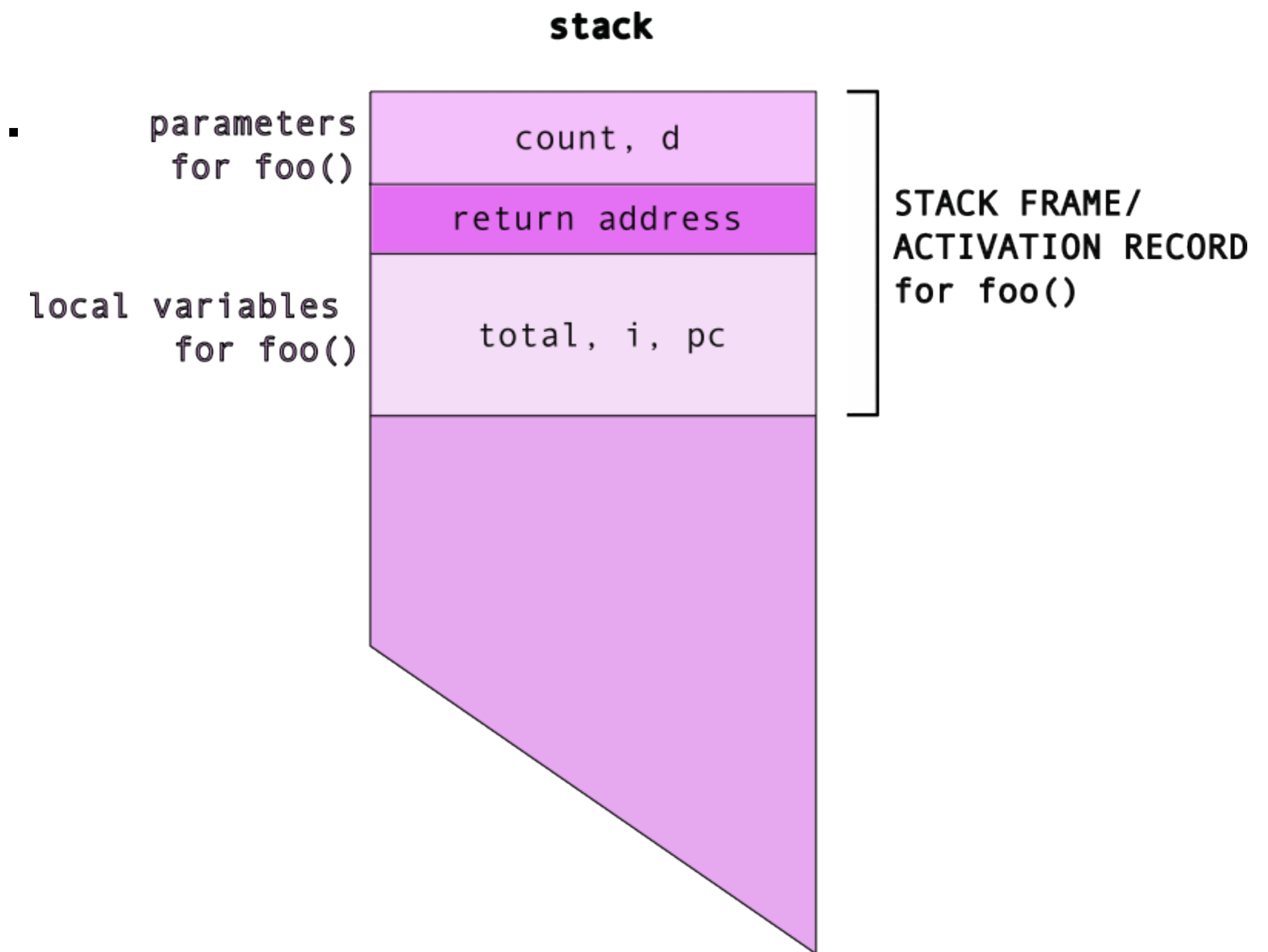
```
1 void bar( char *c ) {
2     int x ;
3     char* pc = c ;
4 }
5
6 int foo( int count, double d) {
7     int total = 0 ;
8     for (unsigned int i = 0 ; i < count ; ++ i)
9         ++total ;
10
11     char* pc = "hello!" ;
12     bar(pc) ;
13
14     return total ;
15 }
16
17 int main() {
18     foo() ;
19
20     return 0 ;
21 }
```



What happens in memory when foo( ) calls bar( ) ?

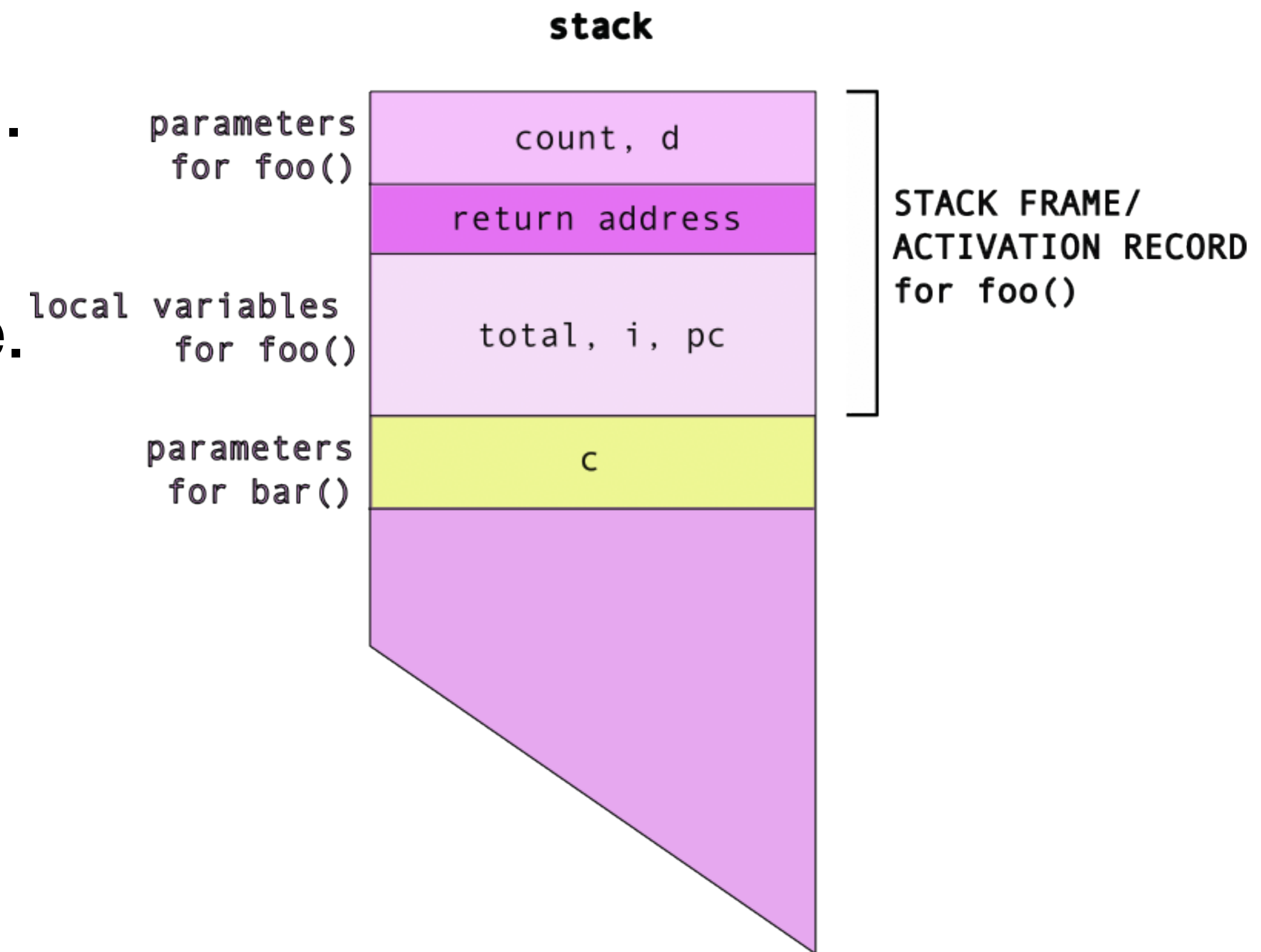
# Stack

- `foo()`'s stack frame is already on the stack.
- Now `bar()`'s stack frame should be created.



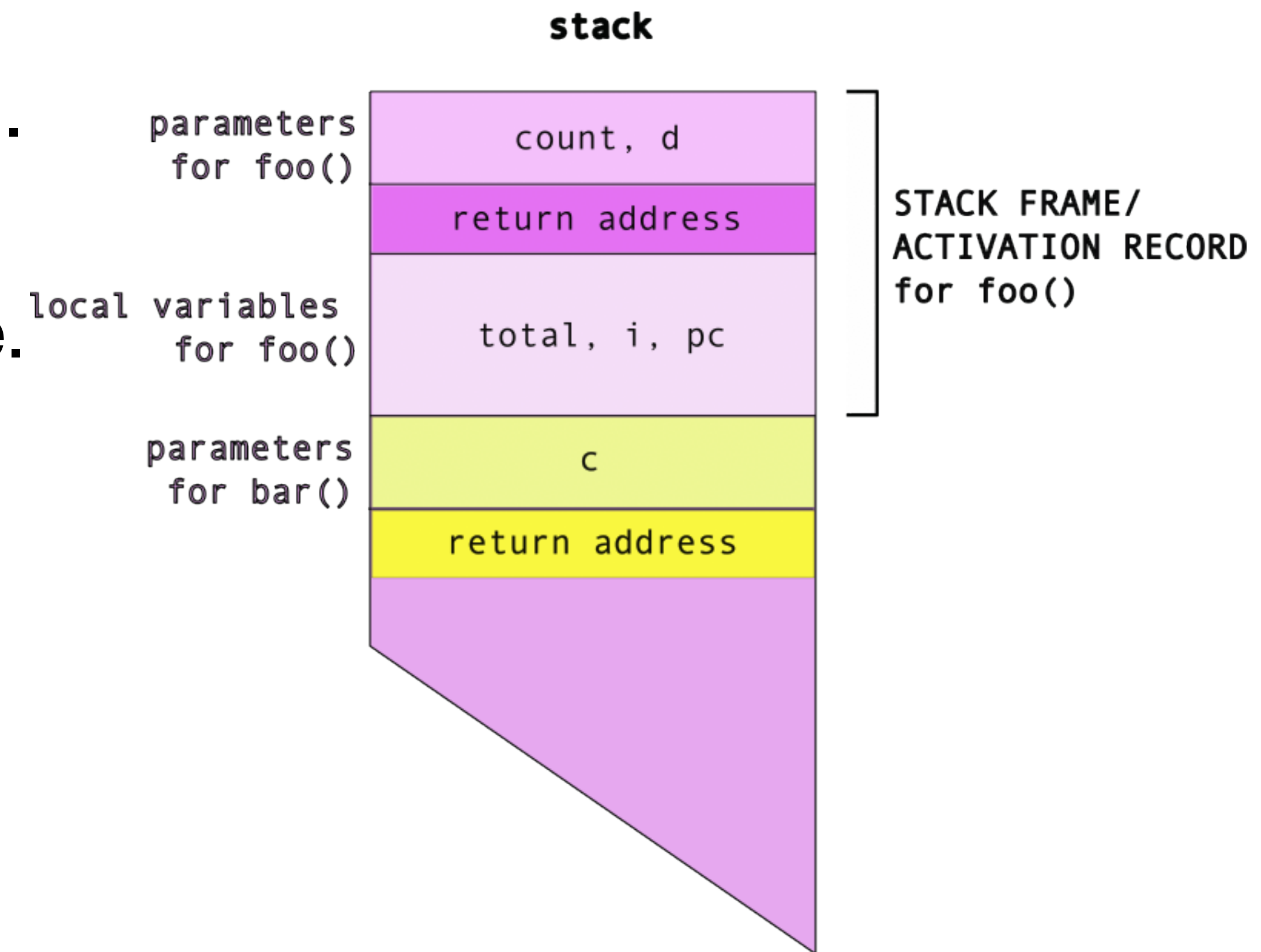
# Stack

- `foo()`'s stack frame is already on the stack.
- Now `bar()`'s stack frame should be created.
- First, function parameters.
  - Again, this depends on the OS + architecture.



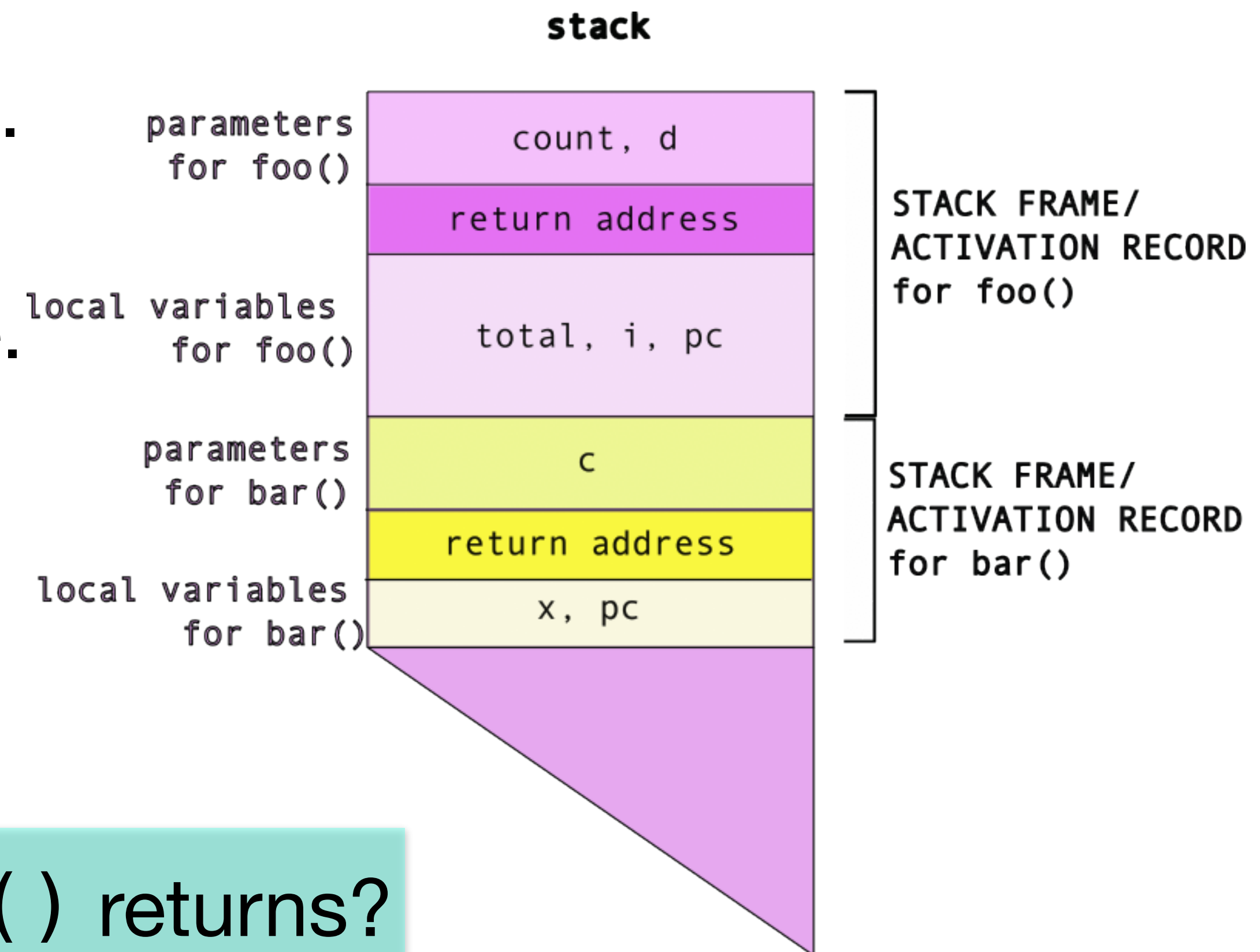
# Stack

- `foo()`'s stack frame is already on the stack.
- Now `bar()`'s stack frame should be created.
- First, function parameters.
  - Again, this depends on the OS + architecture.
- Then, return address back to `foo()`.



# Stack

- `foo()`'s stack frame is already on the stack.
- Now `bar()`'s stack frame should be created.
- First, function parameters.
  - Again, this depends on the OS + architecture.
- Then, return address back to `foo()`.
- Then local variables



What happens on the stack when `bar()` returns?

# Stack

- `bar bar()` returns, the stack would have `foo()`'s stack frame (and its caller).

