CSE227 – Graduate Computer Security

Software Security

UC San Diego

Housekeeping

General course things to know

- *Due by **1/17 (tomorrow!)** at 11:59*
  - Project intention form: https://forms.gle/3efhZJAmfG9Gv4xF8
  - #FinAid Canvas quiz: https://canvas.ucsd.edu/courses/61827/quizzes/199237
- Makeup office hours **tomorrow from 1 – 3pm PT in CSE 3248 for feedback on projects**
- Project specification released here: https://kumarde.com/cse227-wi25/cse227_project_spec.pdf
- Office hours updates
  - Deepak – 2ish – 3:30pm in CSE 3248
  - Tianyi: 11am – 12pm via Zoom (see Canvas)

Housekeeping – Comprehensive Exam

General course things to know

- By the end of the quarter **3/18:**
  - You must get at least a **B-** in the class
  - You must independently write up a document describing your specific contributions to the project with no help from any other student, including your own group
    - I will then independently verify these contributions
- I will provide more details about this around the midpoint check-in

Today

Today’s lecture – Software Security

Learning Objectives

- Recap the layout of computer memory, understand *why* it’s possible to conduct buffer overflow attacks
- Understand the basics of software vulnerabilities, buffer overflow attacks, and ROP
- Discuss some defenses against these attacks and why they might work or not work
- Discuss the landscape of software attacks more broadly and examine what we might do to make software “secure”

Notecard time

Instructions

- Write your **name** and **email** on the card, *legibly*

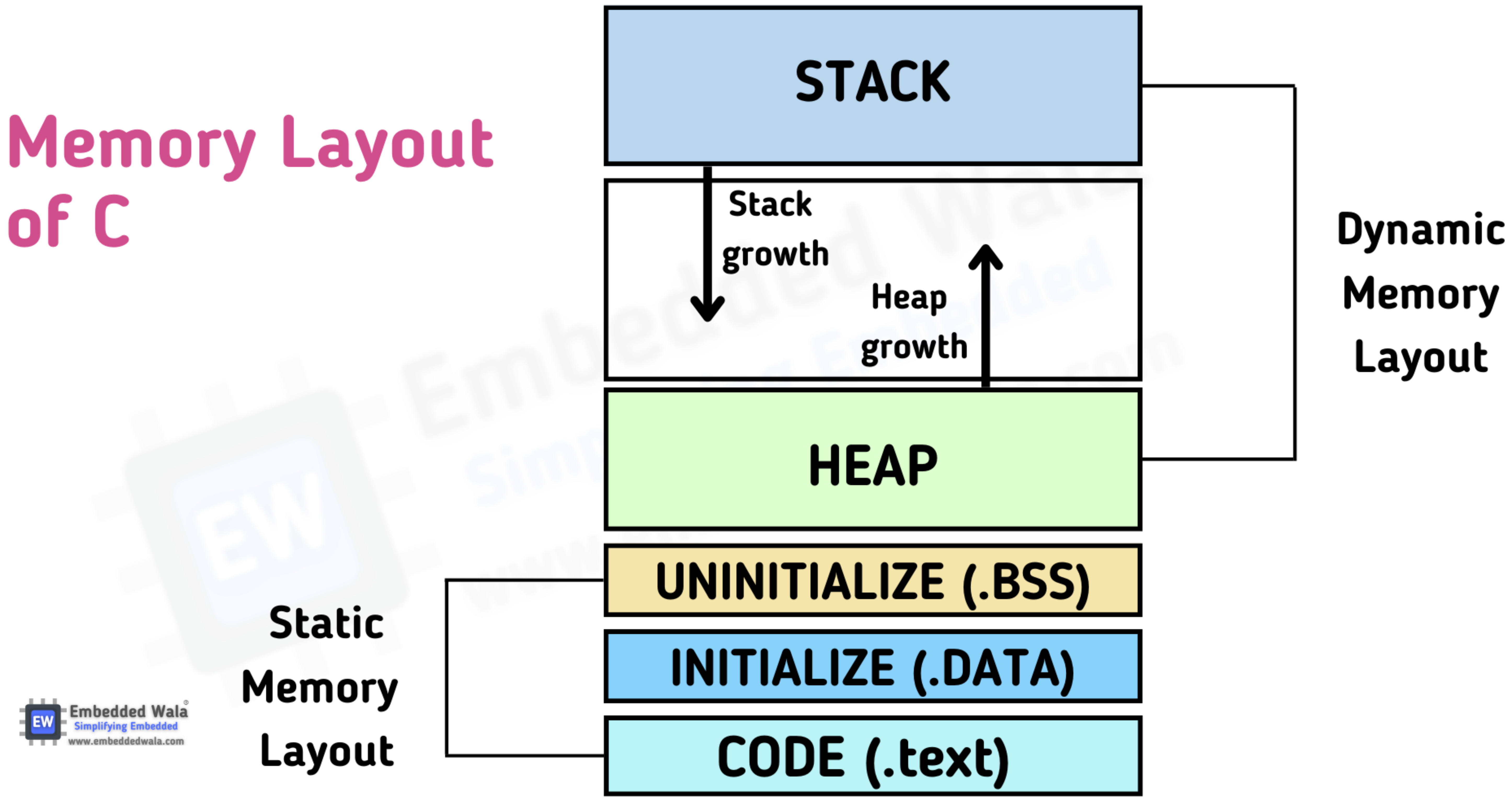
Preliminaries

What is computer memory?

What is computer memory?

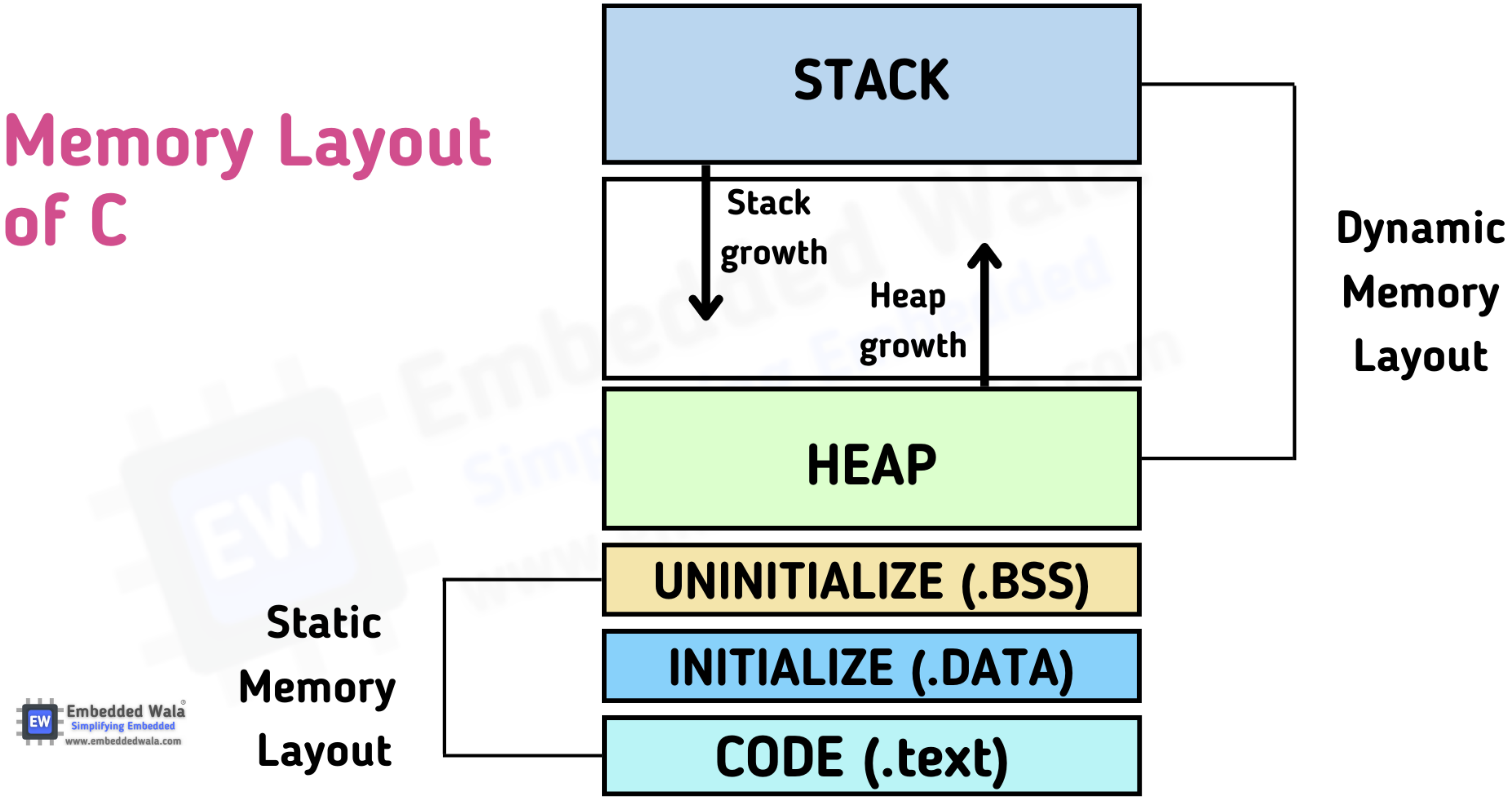
Computer Memory: Quick storage of information, like **data**, **program instructions** used to run computer programs.

Here's how a C program is laid out in memory (simplified)



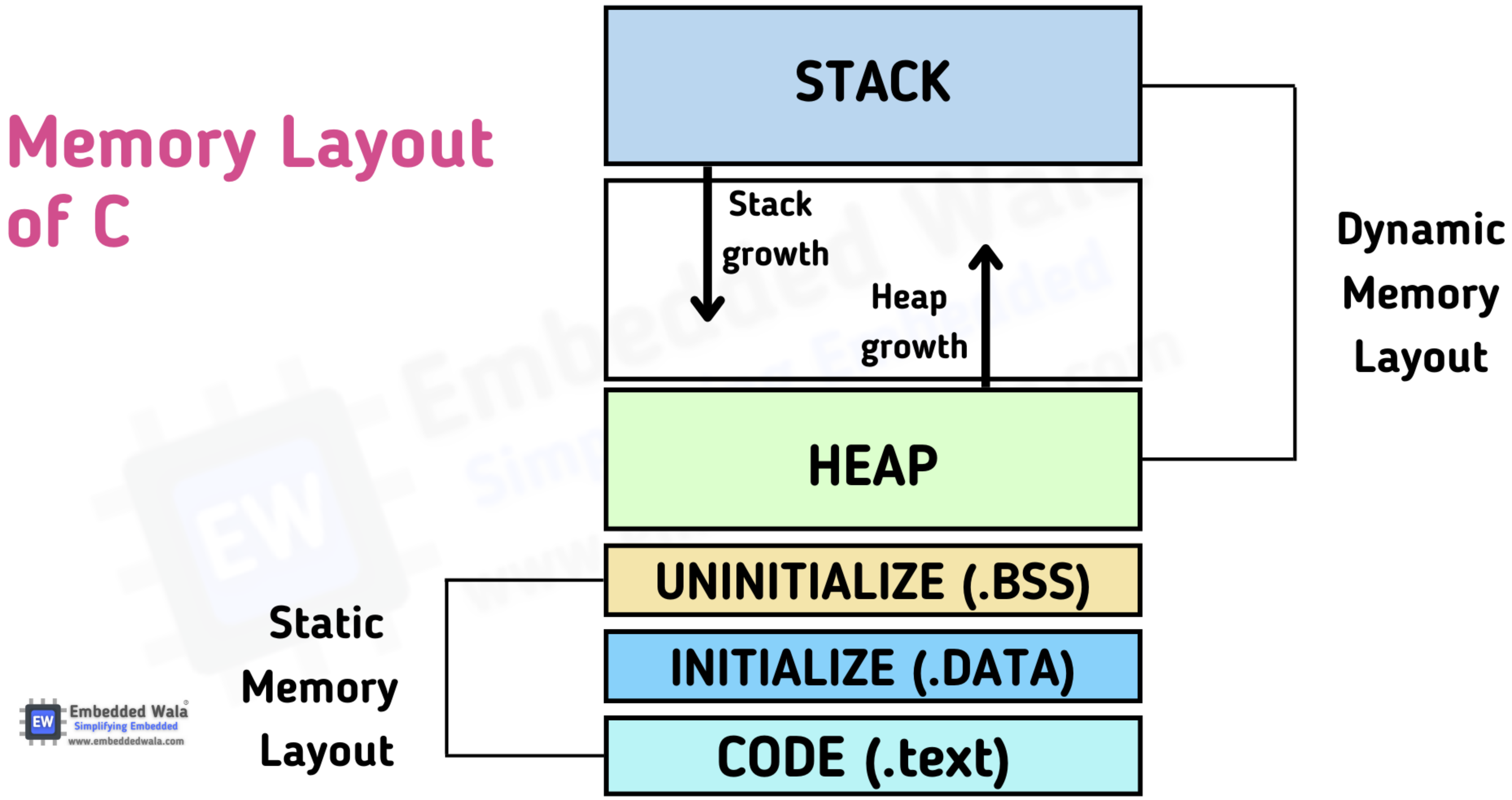
Here's how a C program is laid out in memory (simplified)

- What is the stack?



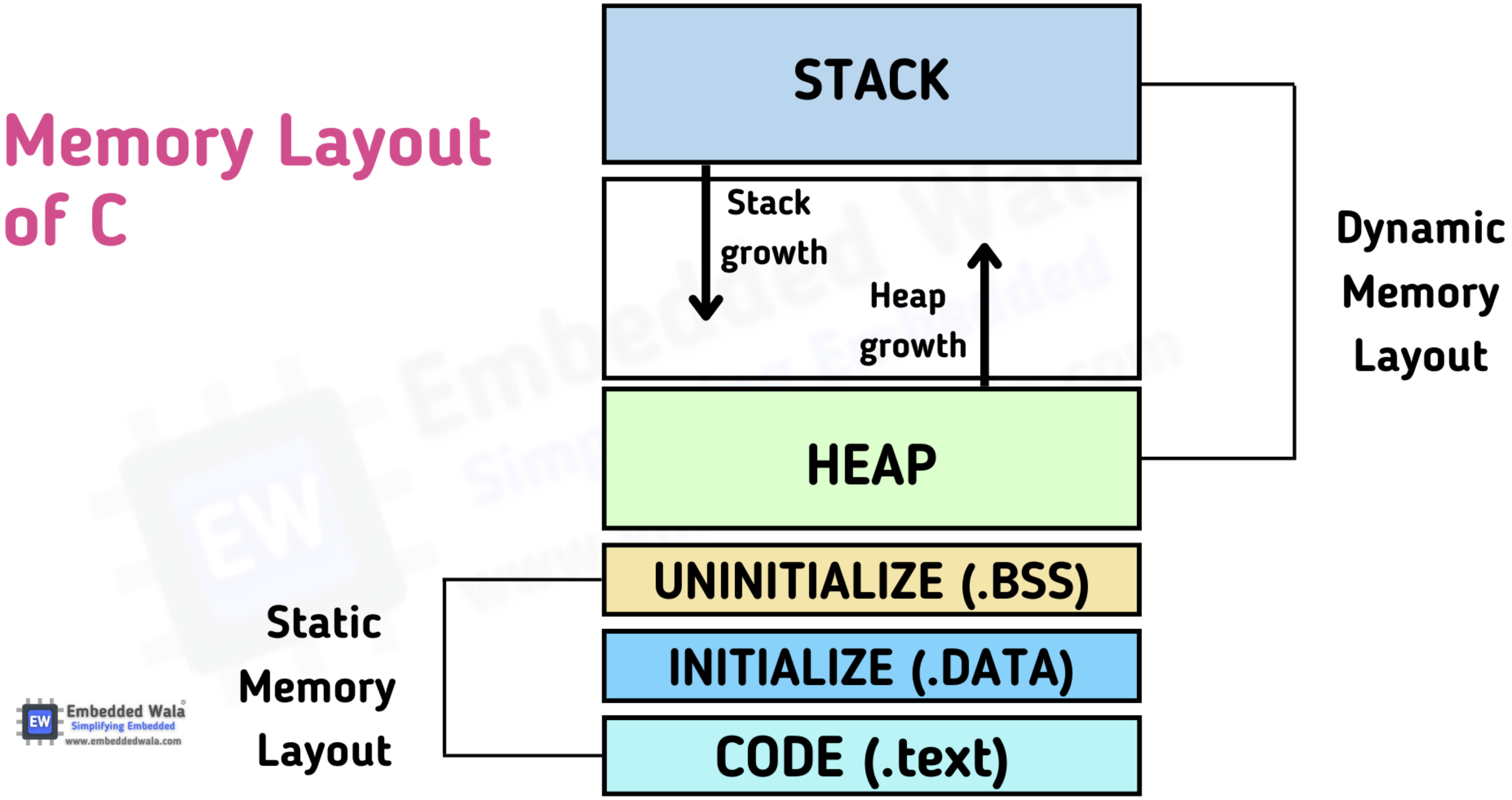
Here's how a C program is laid out in memory (simplified)

- What is the stack?
- What is the heap?



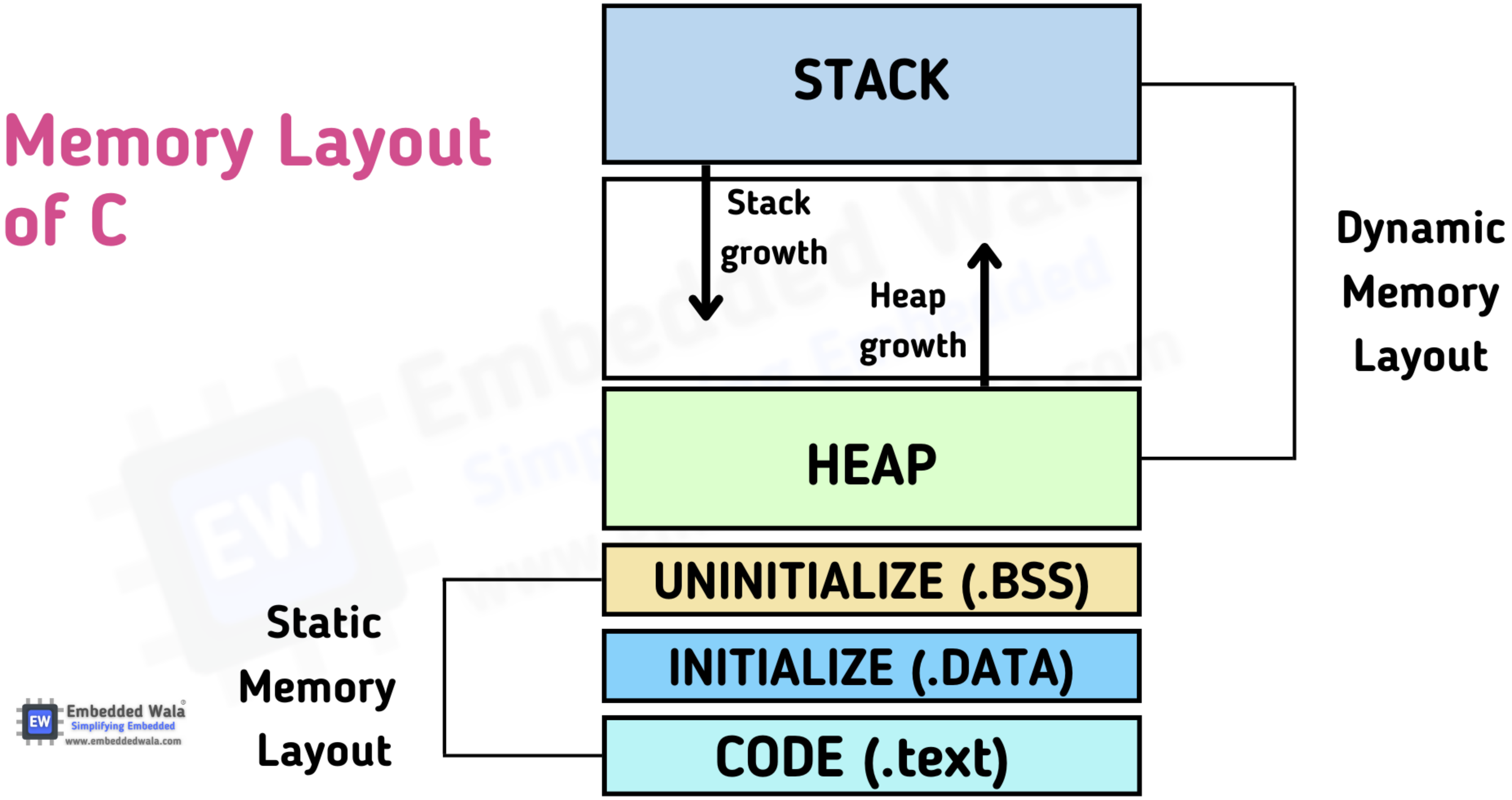
Here’s how a C program is laid out in memory (simplified)

- What is the stack?
- What is the heap?
- How are the stack and heap different?



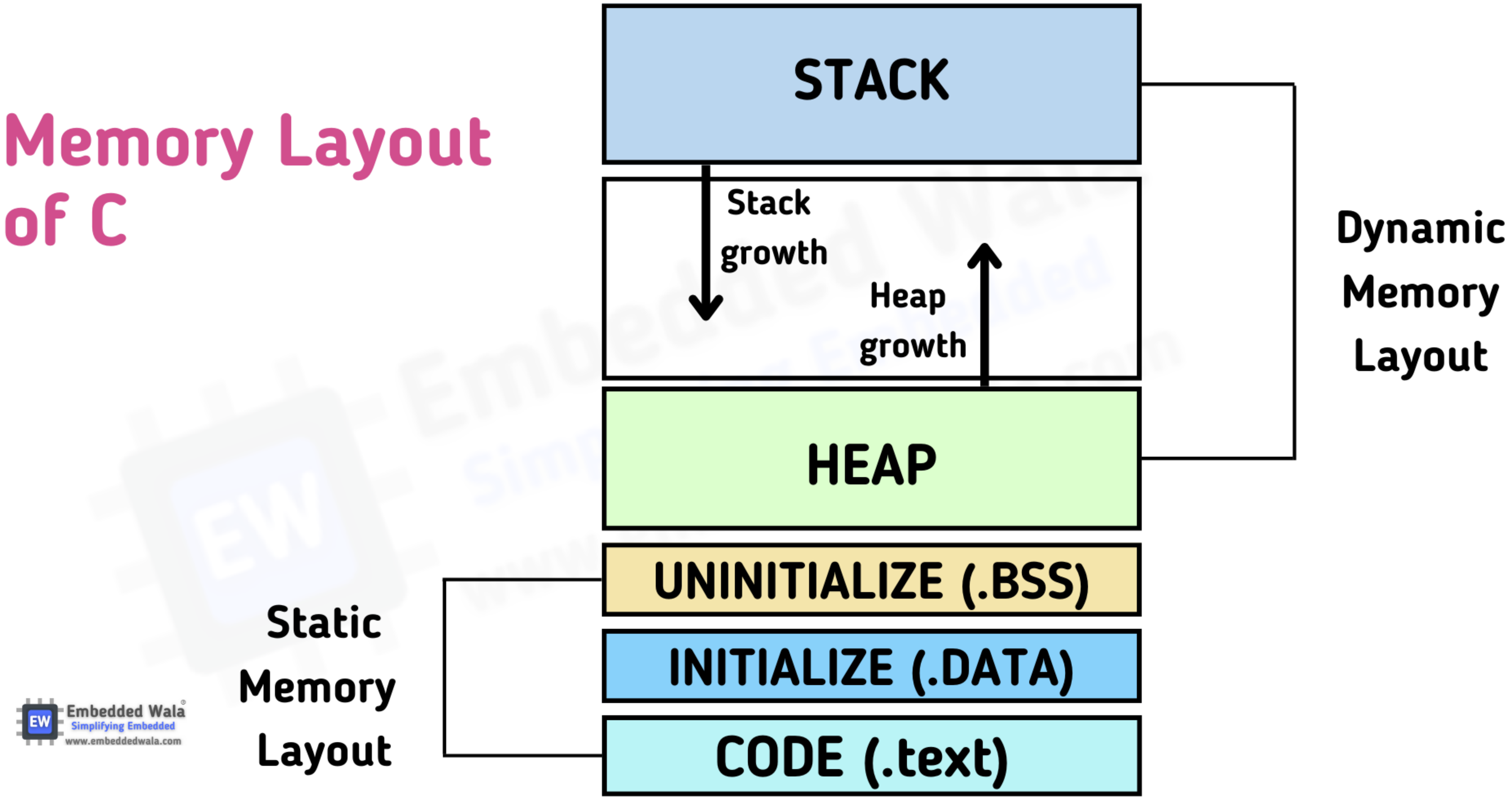
Here’s how a C program is laid out in memory (simplified)

- What is the stack?
- What is the heap?
- How are the stack and heap different?
- What is the .bss segment?



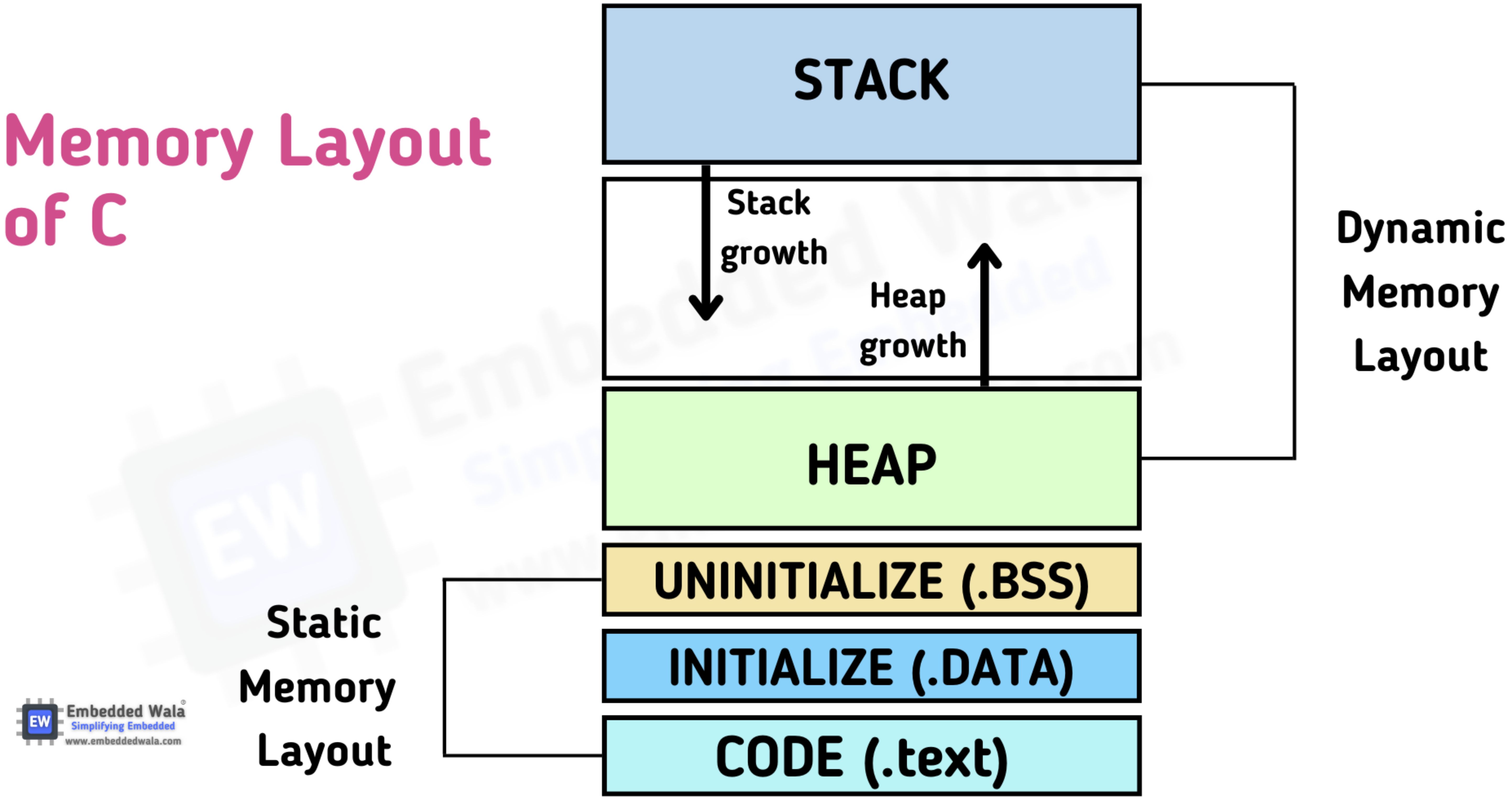
Here’s how a C program is laid out in memory (simplified)

- What is the stack?
- What is the heap?
- How are the stack and heap different?
- What is the .bss segment?
- What is the .data segment?



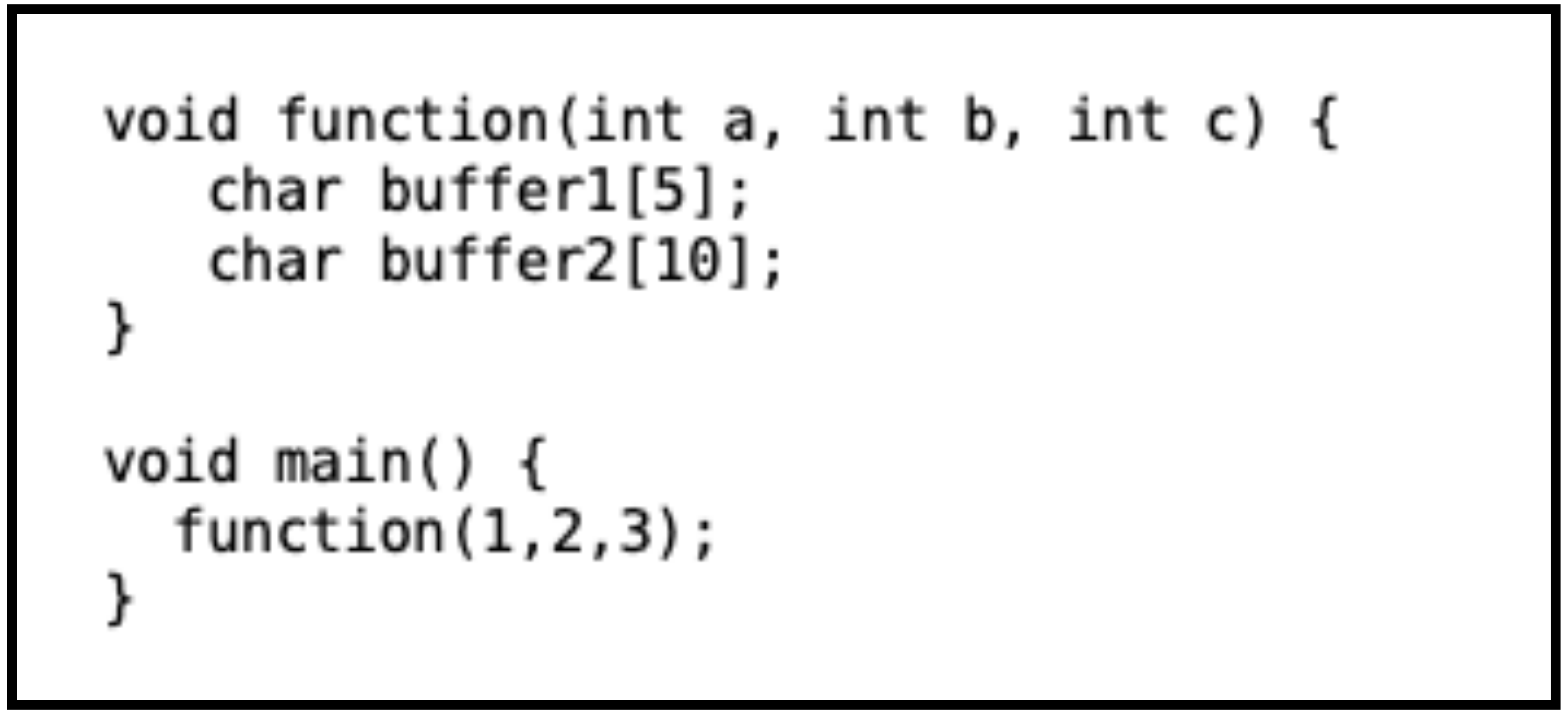
Here's how a C program is laid out in memory (simplified)

- What is the stack?
- What is the heap?
- How are the stack and heap different?
- What is the .bss segment?
- What is the .data segment?
- What is the .text segment?



C Arrays

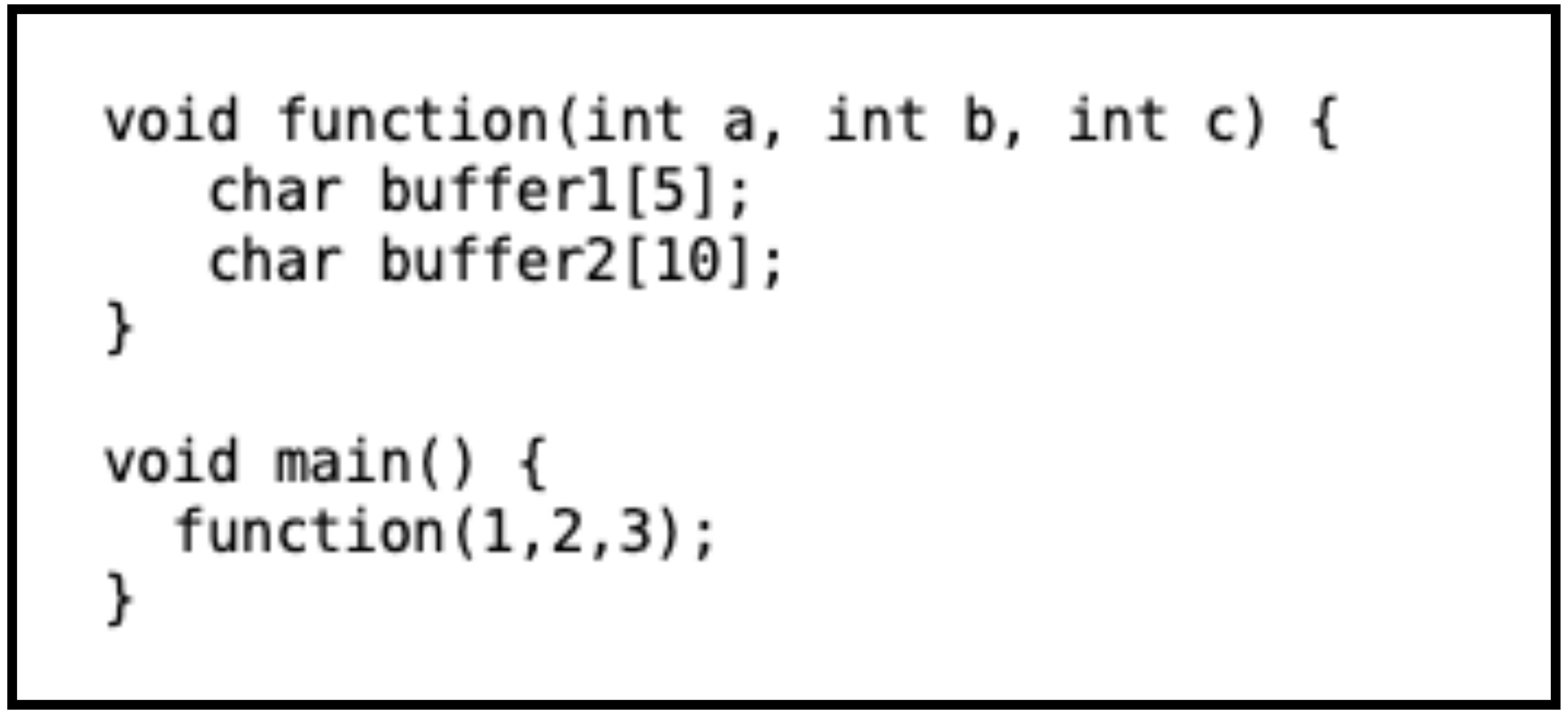
- What is an array?



```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

C Arrays

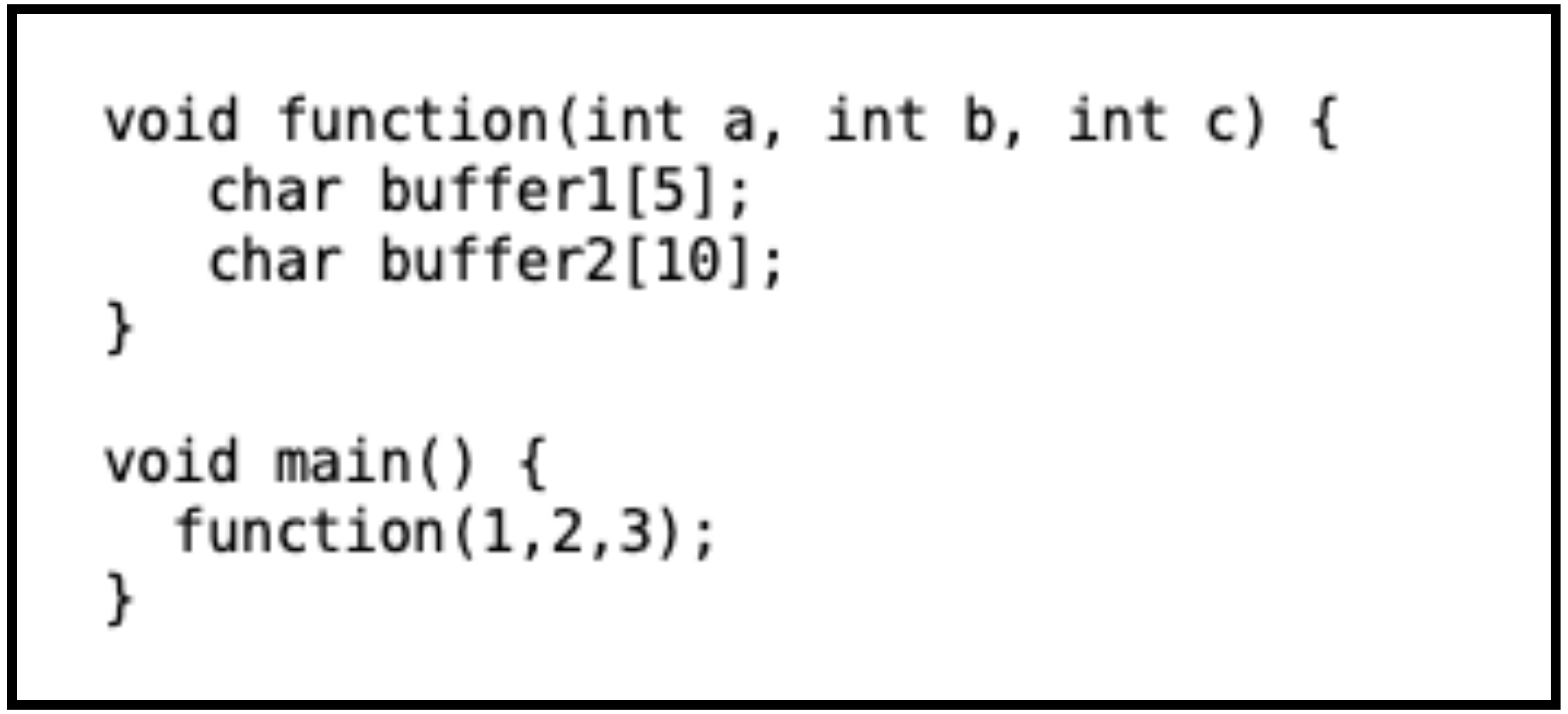
- What is an array?
- How much memory is allocated for these char buffers? Assume a 32-bit machine w/ 4-byte word size



```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

C Arrays

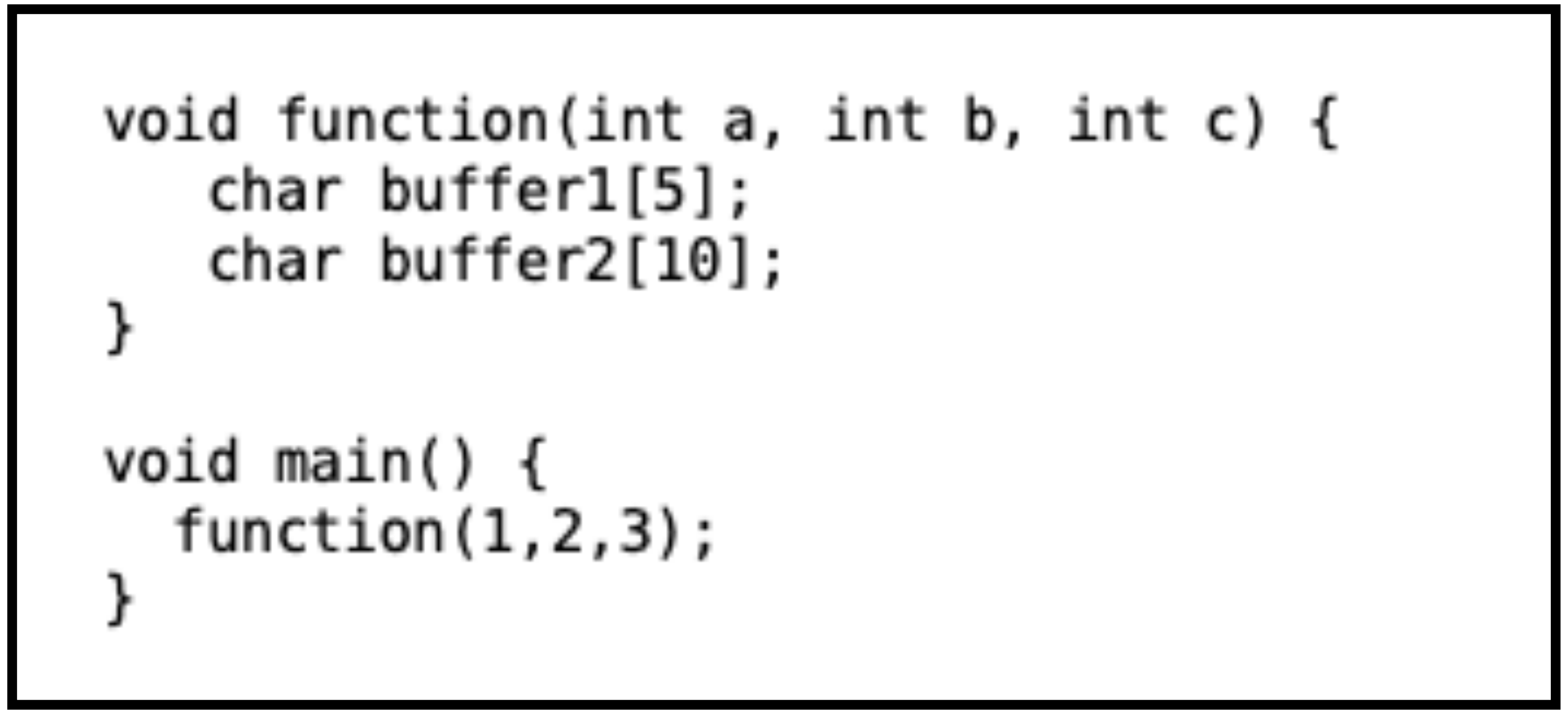
- What is an array?
- How much memory is allocated for these char buffers? Assume a 32-bit machine w/ 4-byte word size
- Is this memory allocated on the stack or the heap?



```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

C Arrays

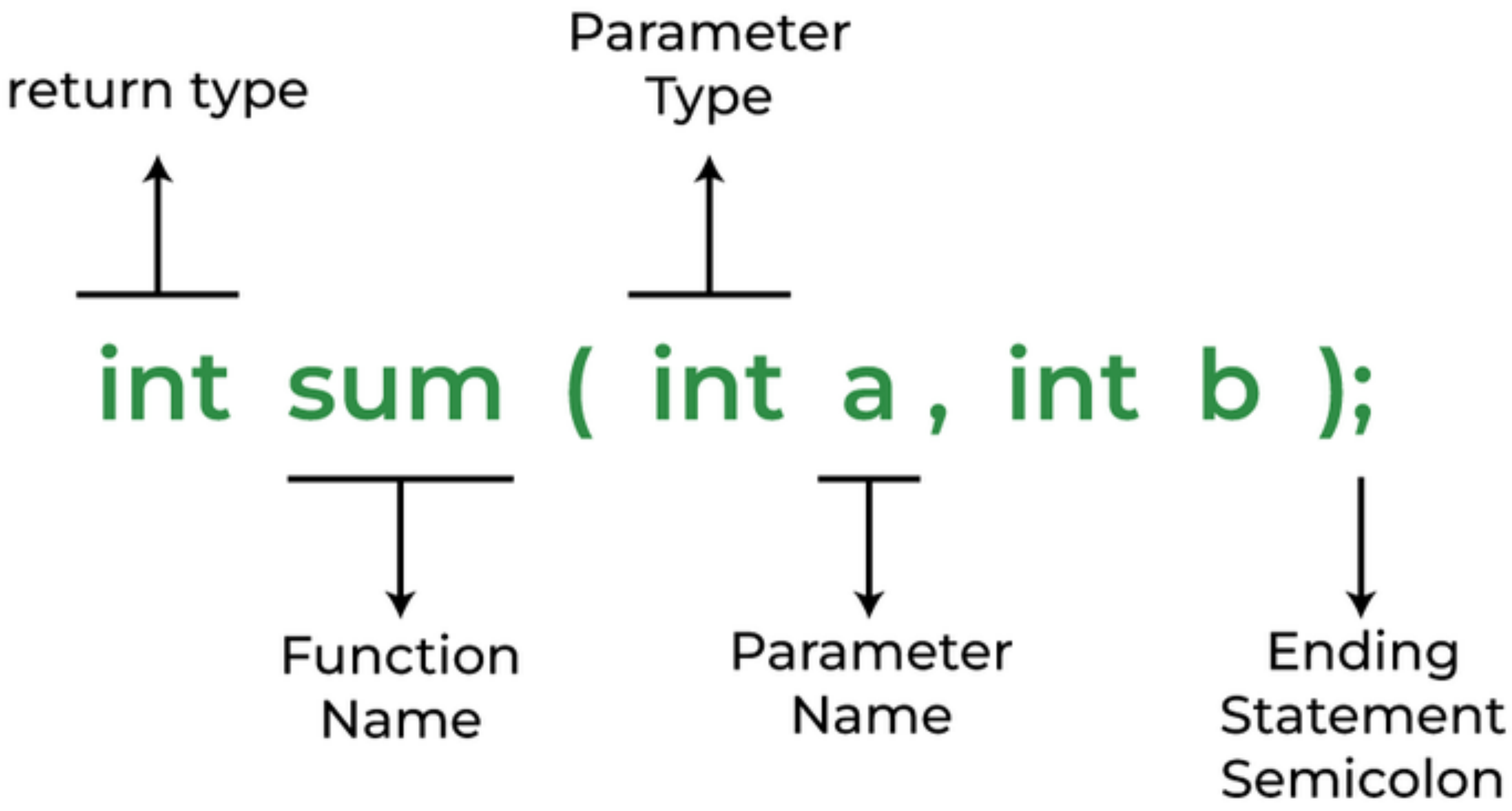
- What is an array?
- How much memory is allocated for these char buffers? Assume a 32-bit machine w/ 4-byte word size
- Is this memory allocated on the stack or the heap?
- Will the program throw an error if you write beyond the buffer?
  - Why or why not?



```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

What is a function in C?

What is a function in C?



```
int sum ( int a , int b );
```

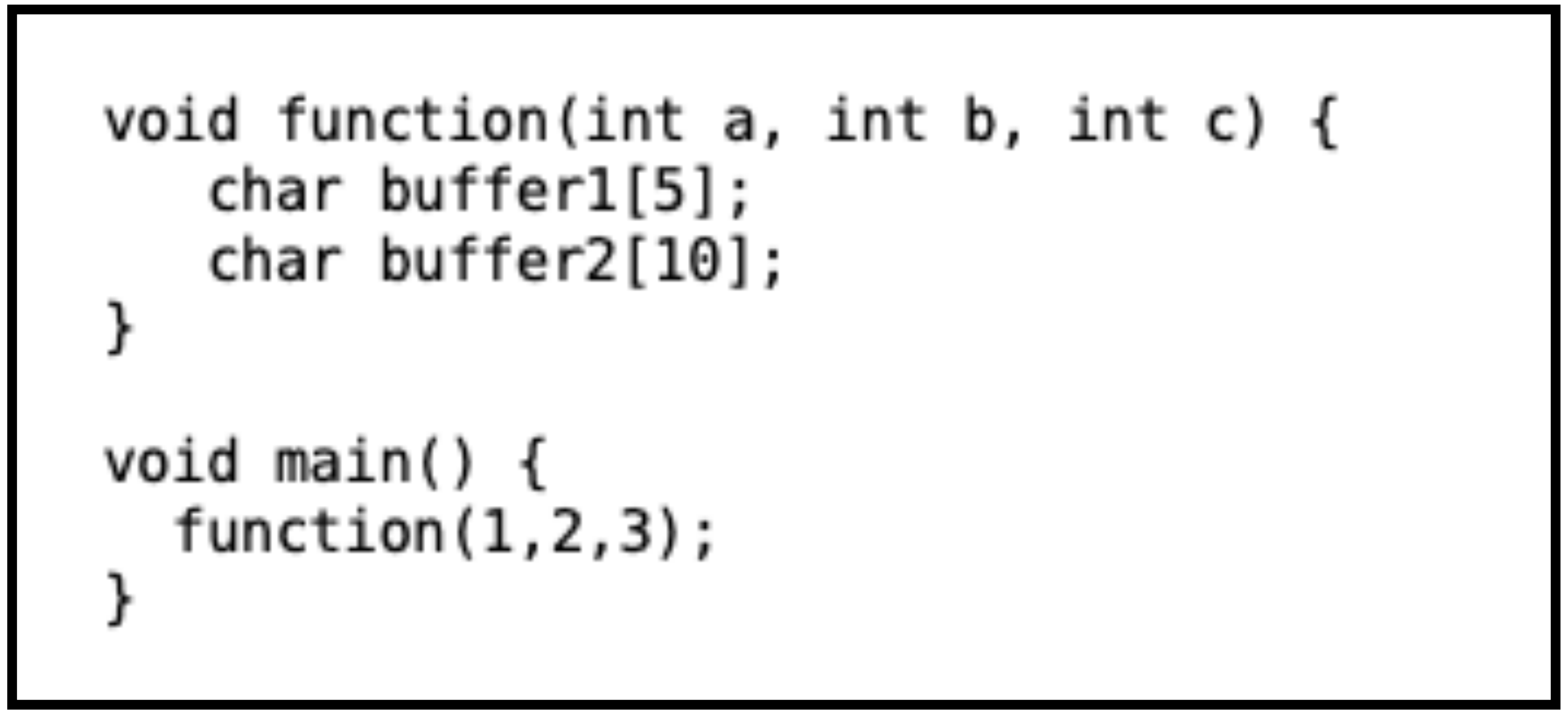
Why are we talking about C?

Why are we talking about C?

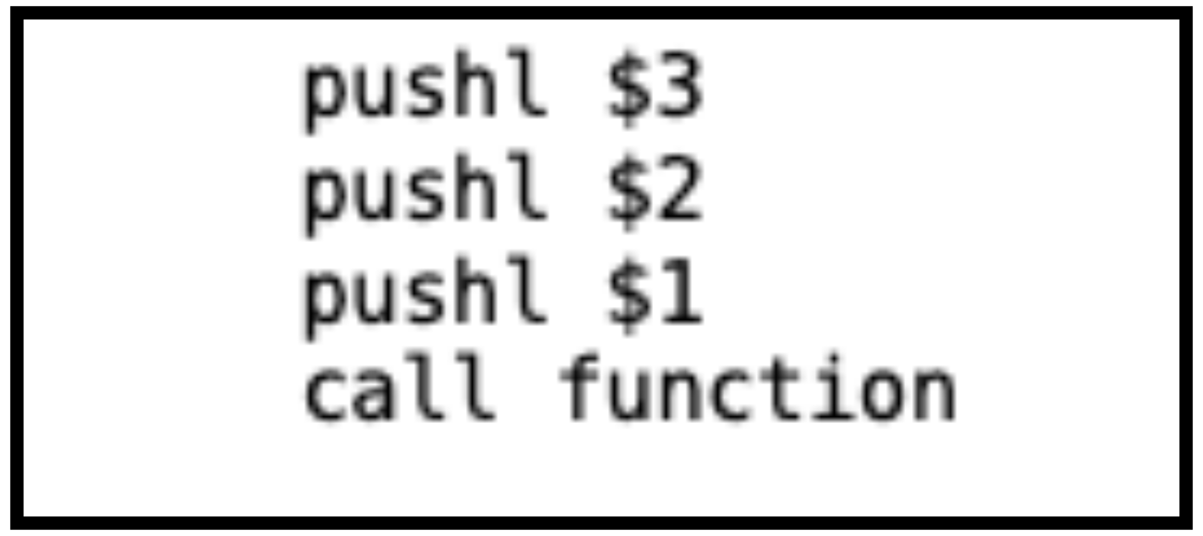
What is the relationship between a function and the stack?

What is the relationship between a function and the stack?

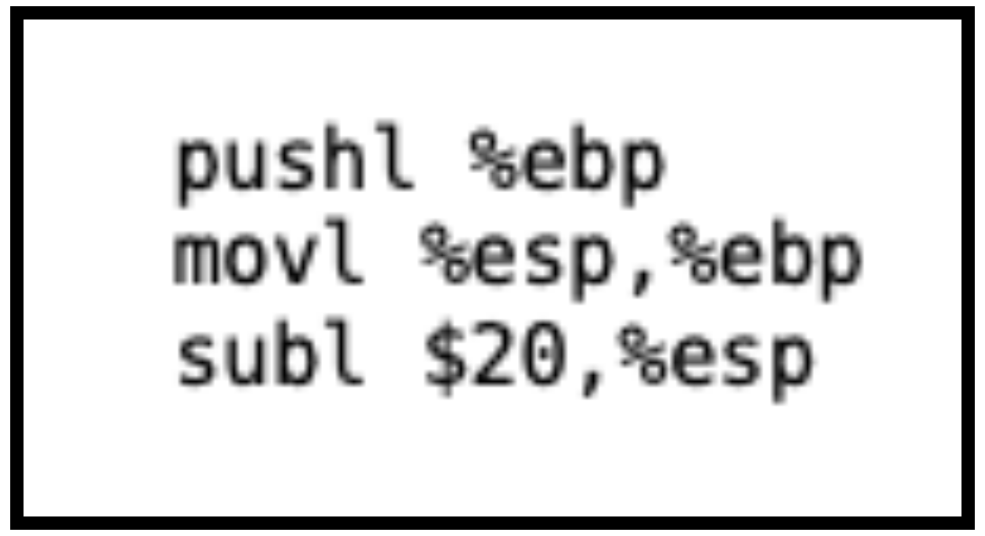
- We implement function calls via the stack —> using **push** and **pop** to keep track of where in the function we are
- Example:



```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```



```
pushl $3
pushl $2
pushl $1
call function
```



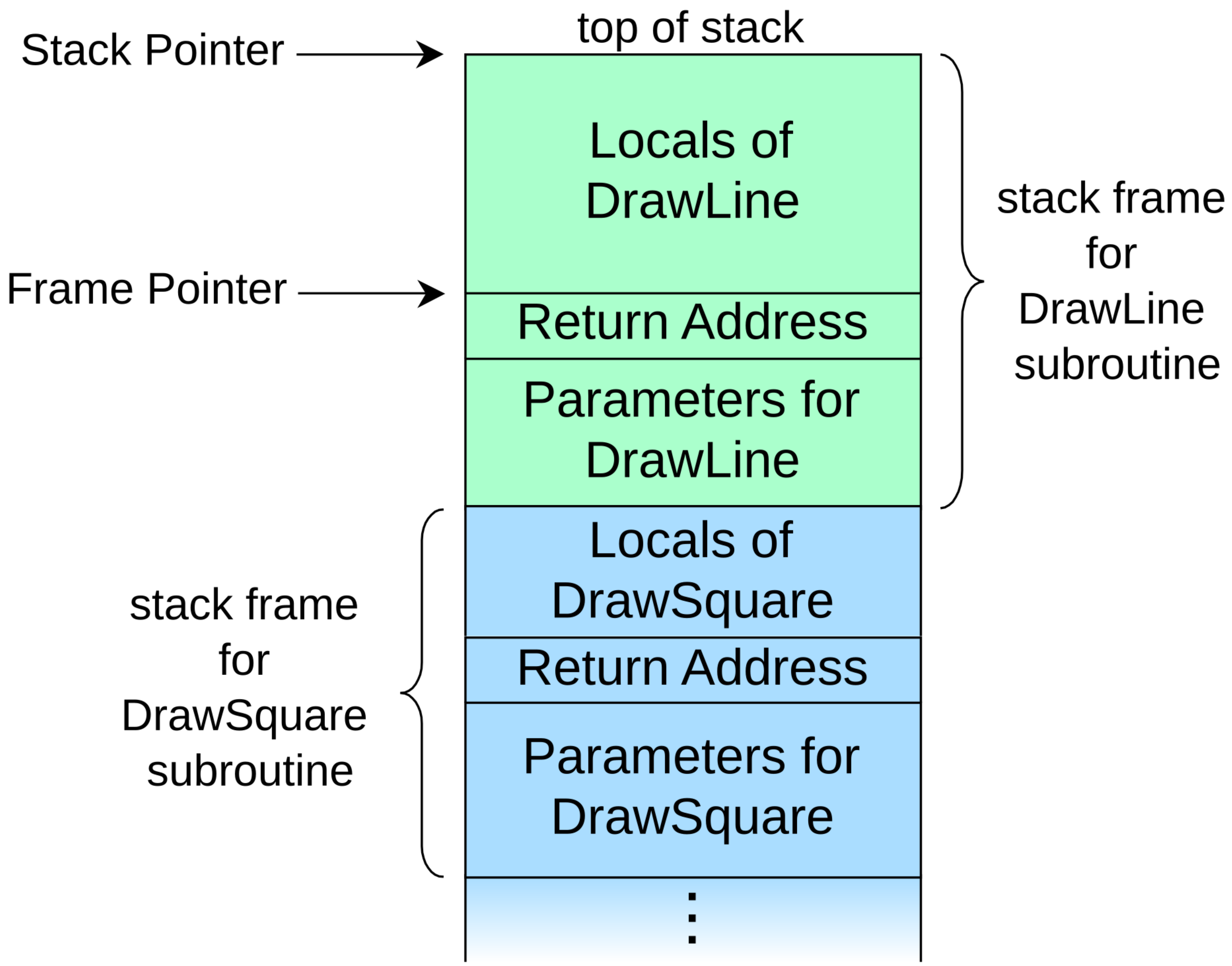
```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

Stack Frame Organization

- What is a stack frame?
- What is a return address?
- Where does a return address go in a stack frame?

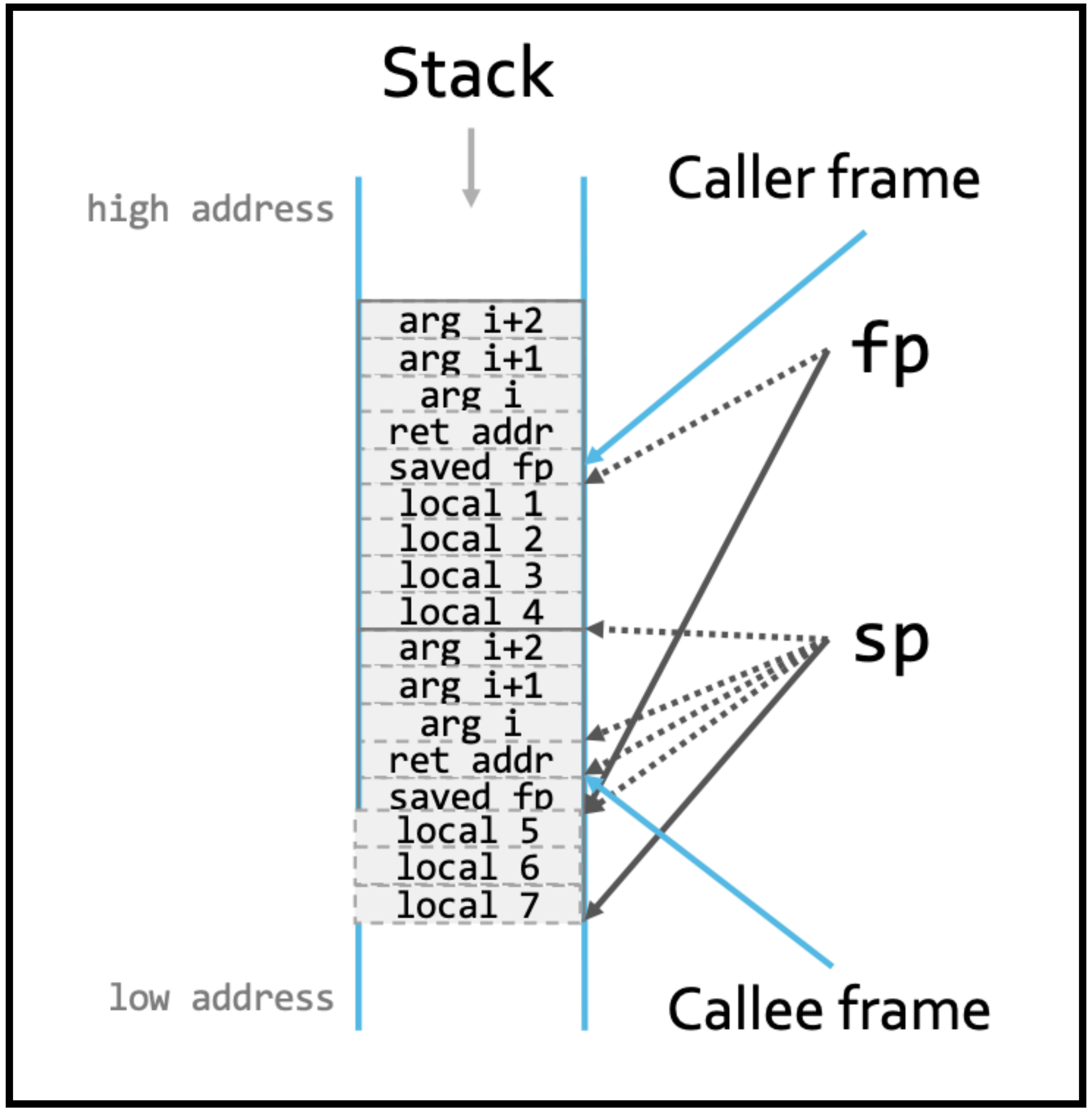
Stack Frame Organization

- Stacks are divided into **frames**
  - Each frame stores locals + args to called functions
- **call** will push the return address (e.g., where you were previously) onto the stack
- **Stack pointer** points to the top of the stack (%esp register in x86)
  - x86: stack grows down (from high to low addresses)
- **Frame pointer** points to the caller's frame on the stack (%ebp in x86)



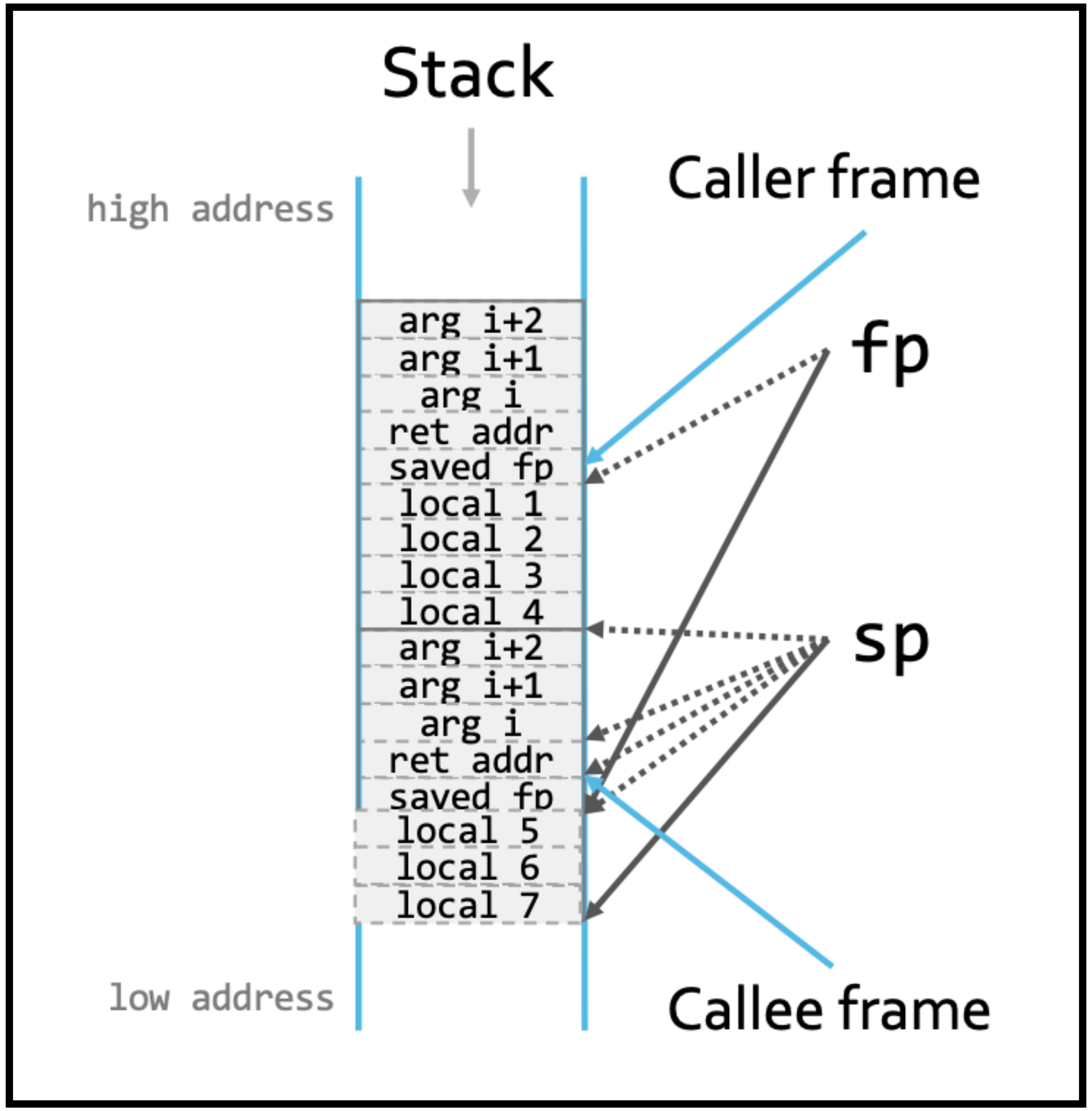
Understanding Function Calls

- What is the caller and what is the callee?



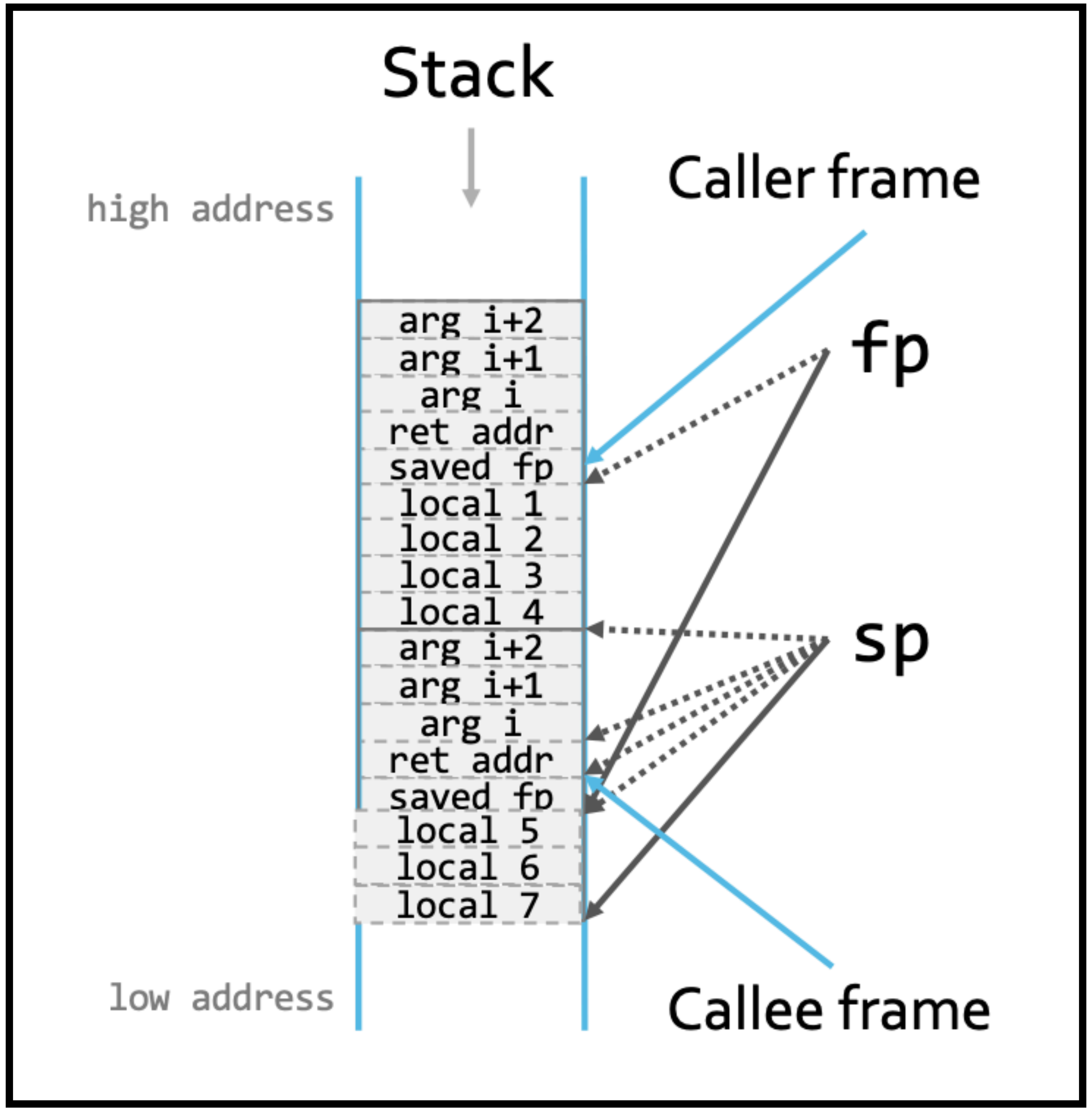
Understanding Function Calls

- What is the caller and what is the callee?
    - Both are functions! Even *main* is a function.



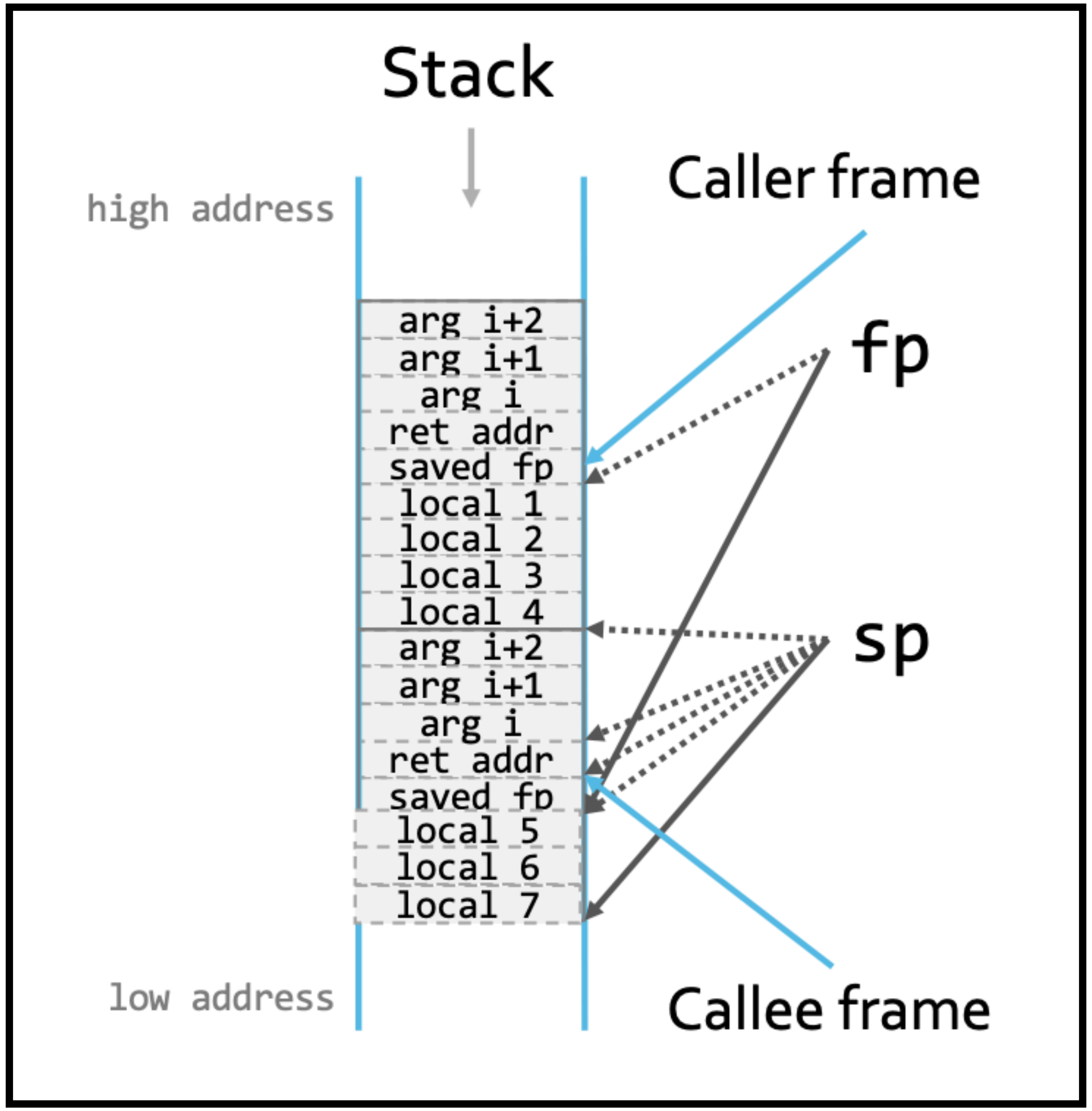
Understanding Function Calls

- What is the caller and what is the callee?
    - Both are functions! Even *main* is a function.
- What are the responsibilities of the caller?



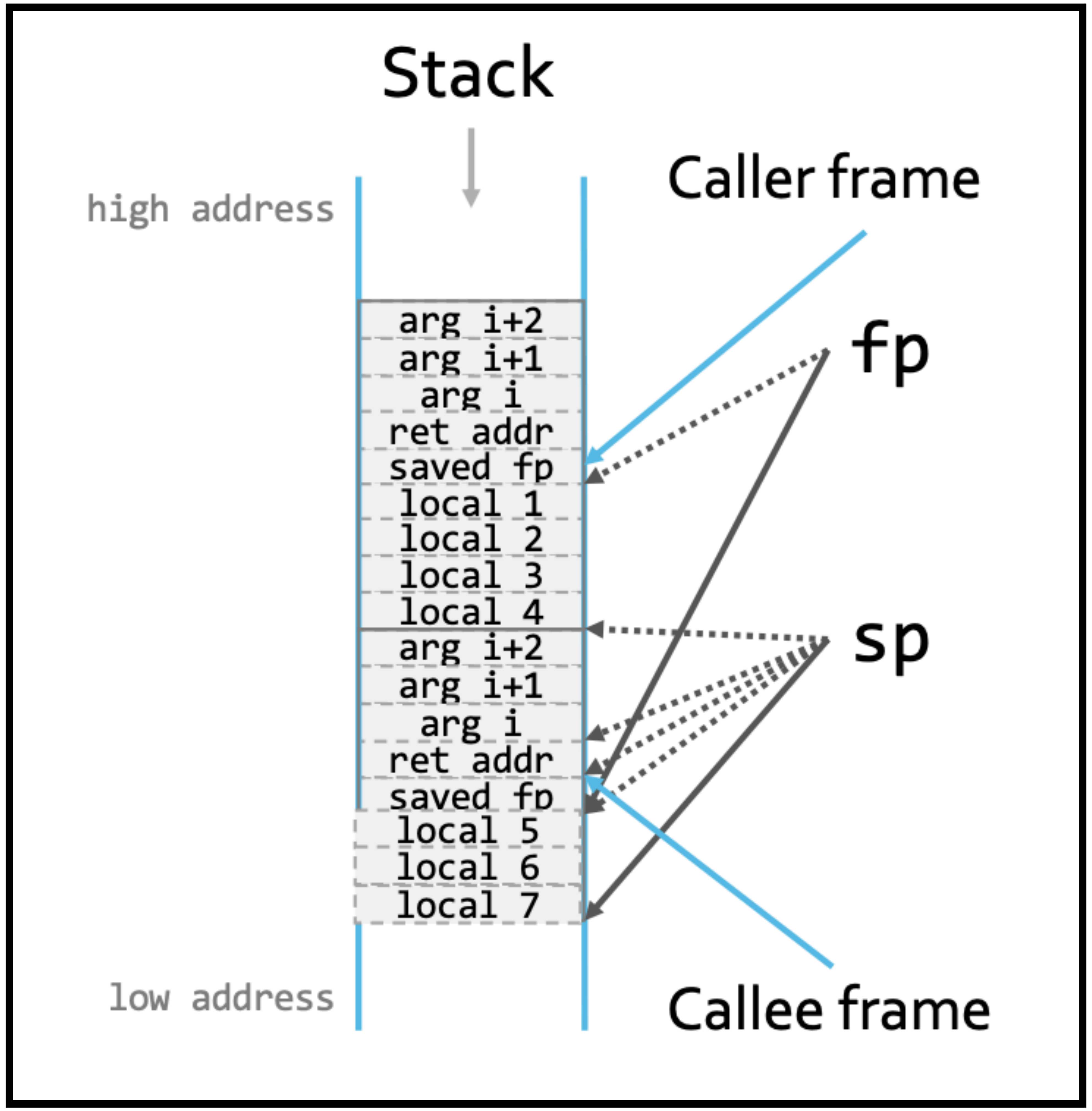
Understanding Function Calls

- What is the caller and what is the callee?
  - Both are functions! Even *main* is a function.
- What are the responsibilities of the caller?
  - Push arguments, save return address, call new function



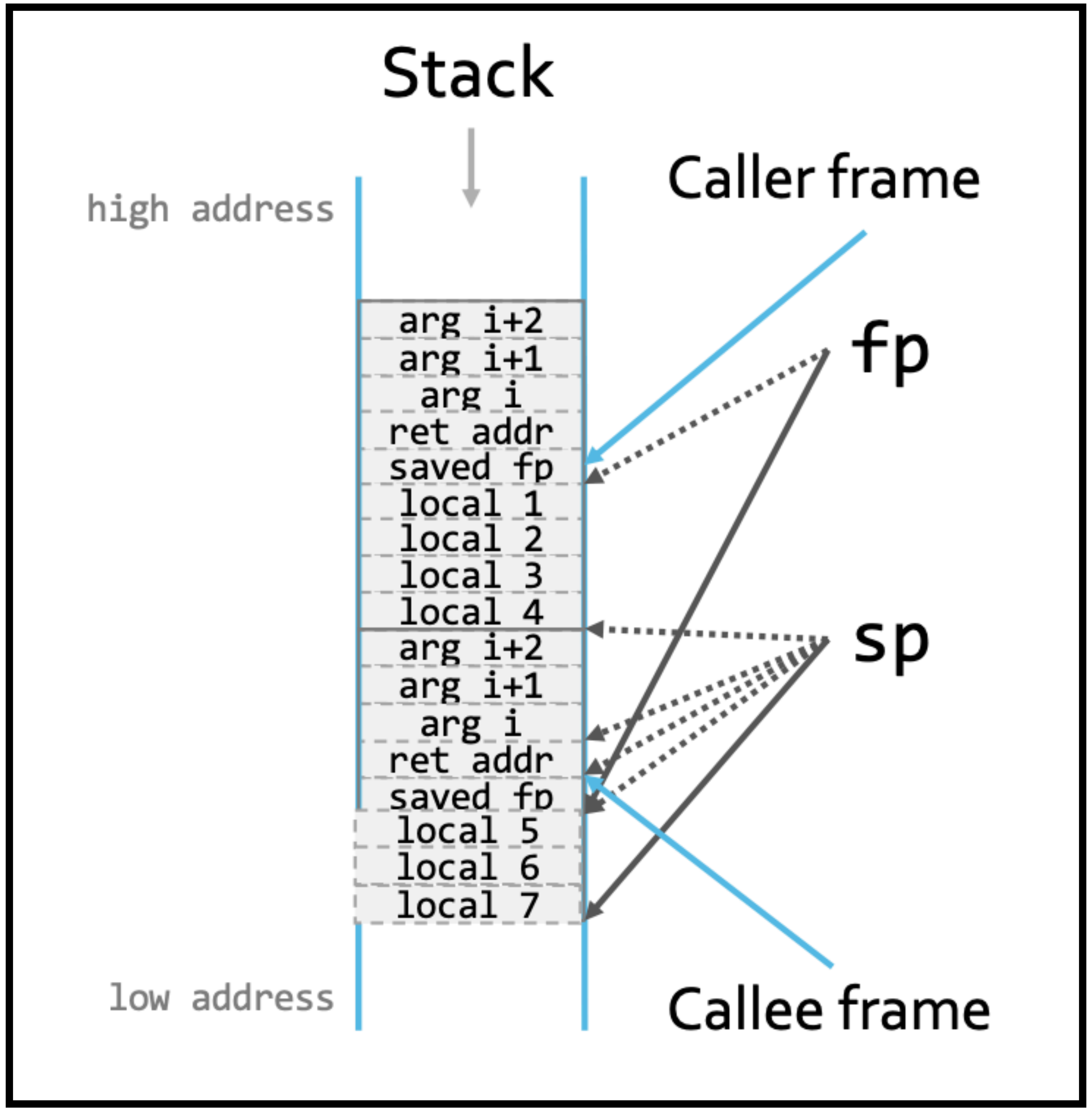
Understanding Function Calls

- What is the caller and what is the callee?
  - Both are functions! Even *main* is a function.
- What are the responsibilities of the caller?
  - Push arguments, save return address, call new function
- What is the responsibility of the callee?



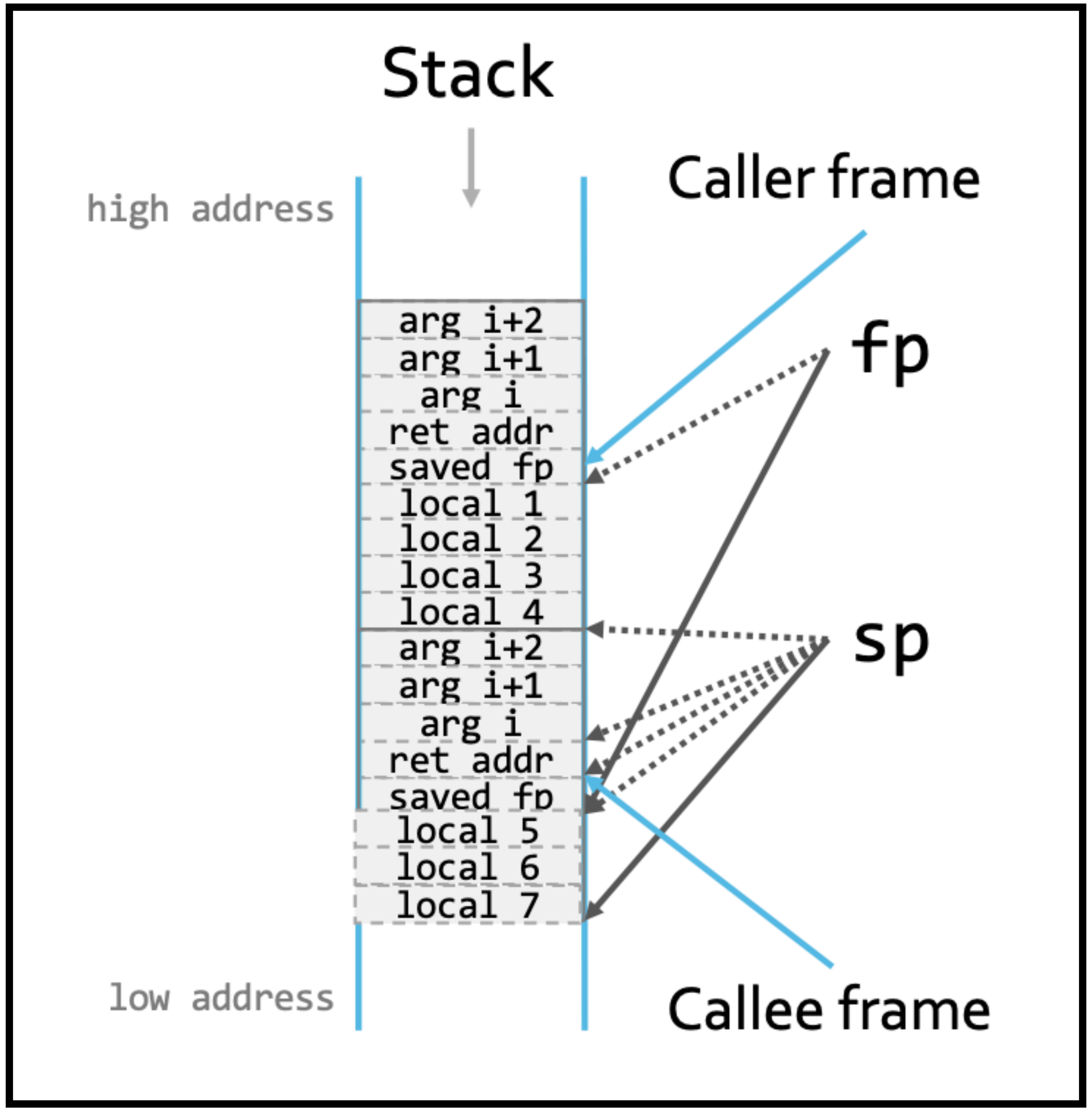
Understanding Function Calls

- What is the caller and what is the callee?
    - Both are functions! Even *main* is a function.
- What are the responsibilities of the caller?
    - Push arguments, save return address, call new function
- What is the responsibility of the callee?
    - Save old FP, set FP = SP, allocate stack space for local storage



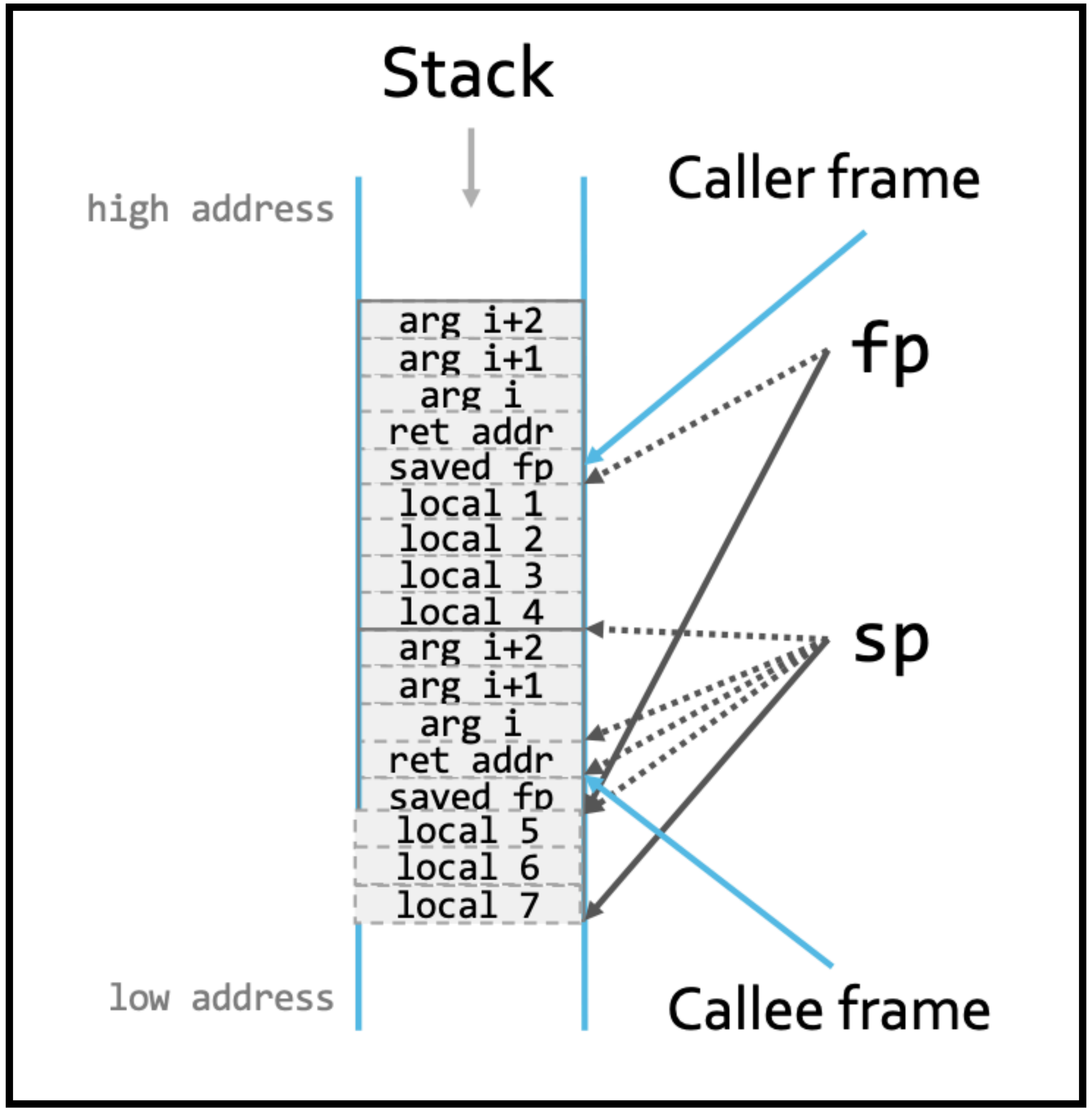
Understanding Function Returns

- What does the callee do when returning?



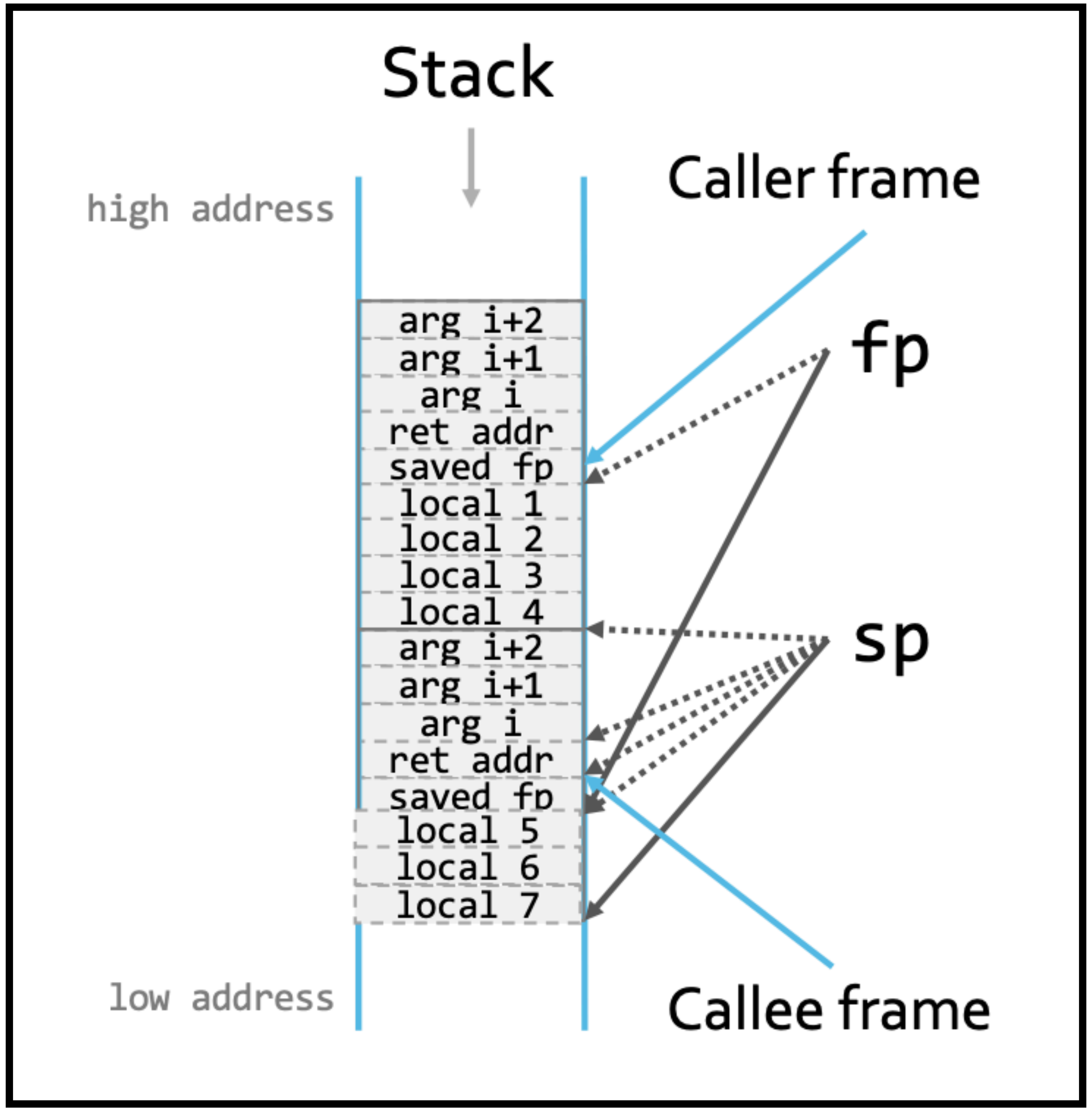
Understanding Function Returns

- What does the callee do when returning?
  - Pop local storage
  - Set SP = FP
  - Pop frame pointer
  - Pop return address and **ret**



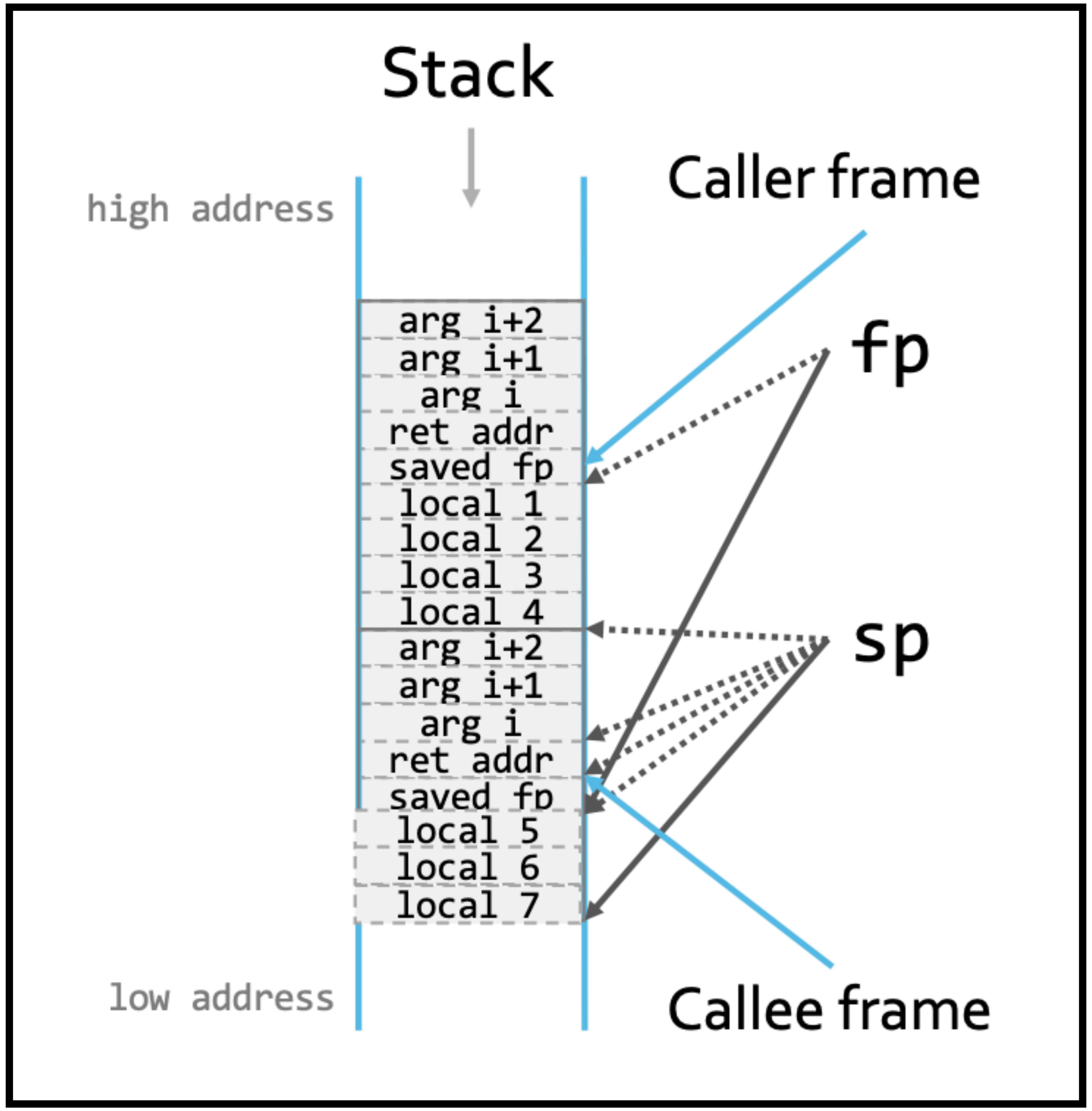
Understanding Function Returns

- What does the callee do when returning?
  - Pop local storage
  - Set SP = FP
  - Pop frame pointer
  - Pop return address and **ret**
- What does the caller do when returning?



Understanding Function Returns

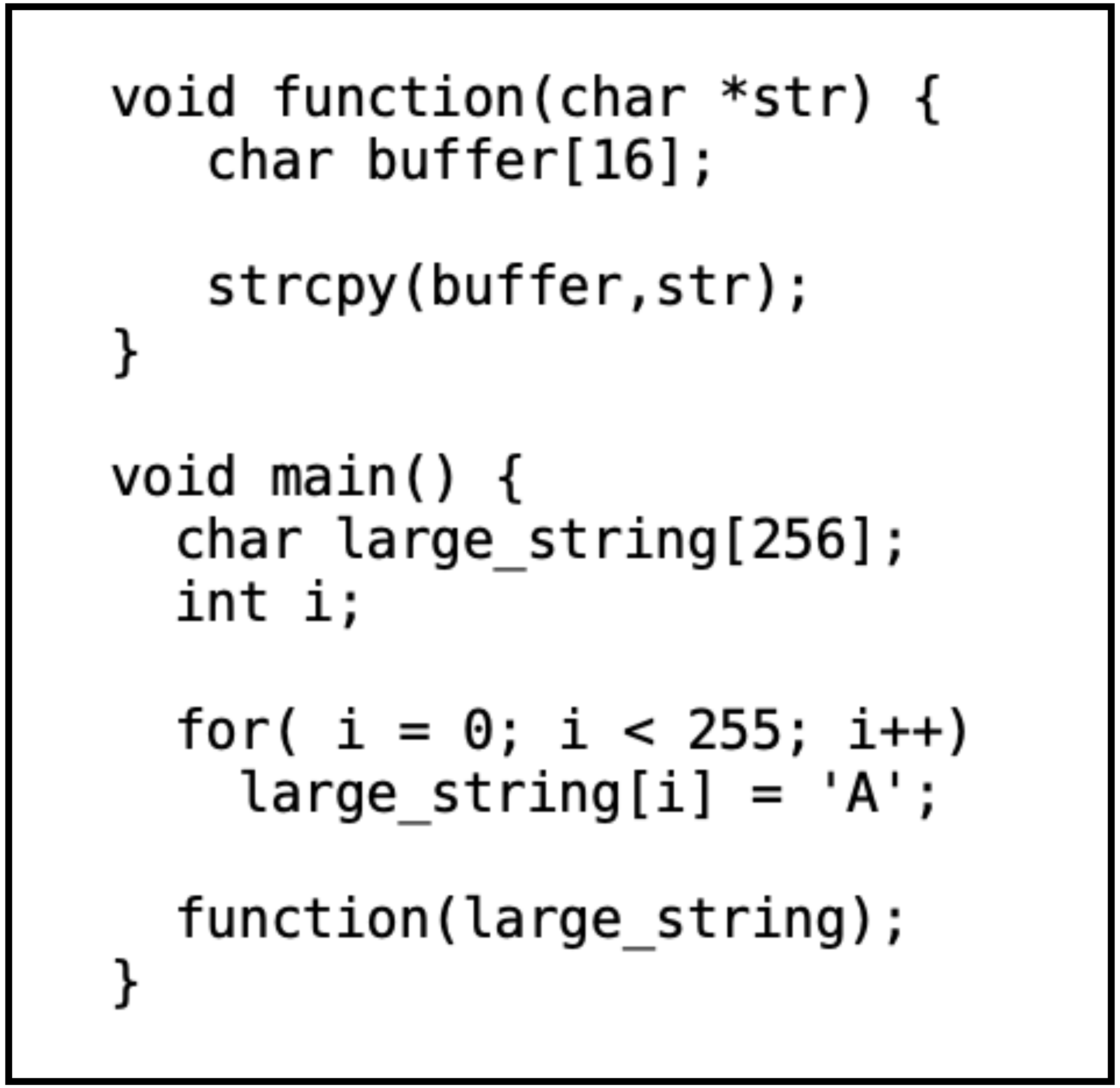
- What does the callee do when returning?
  - Pop local storage
  - Set SP = FP
  - Pop frame pointer
  - Pop return address and **ret**
- What does the caller do when returning?
  - Pop arguments and continue



Any questions?

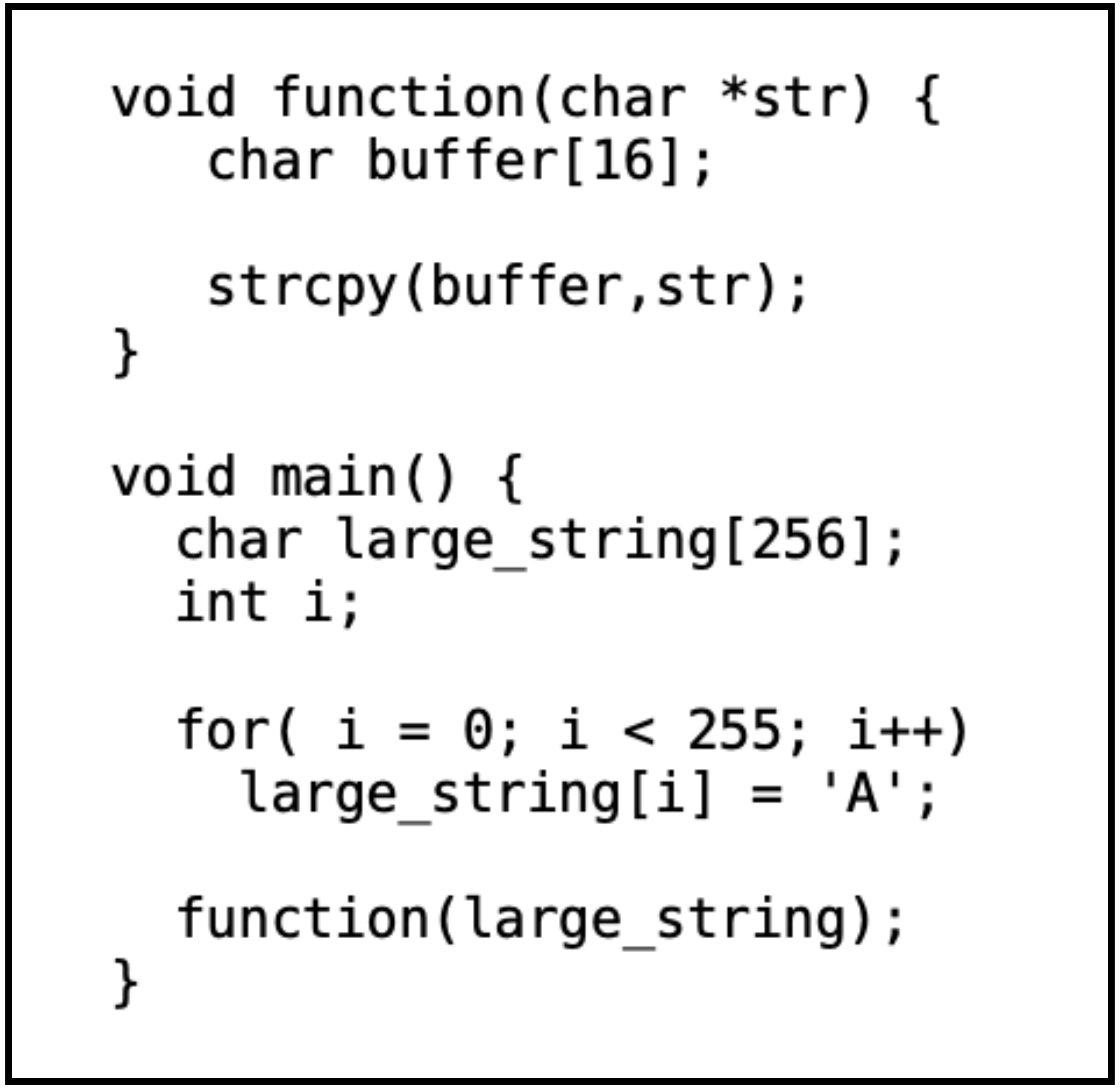
Smashing the Stack

What does this function do?



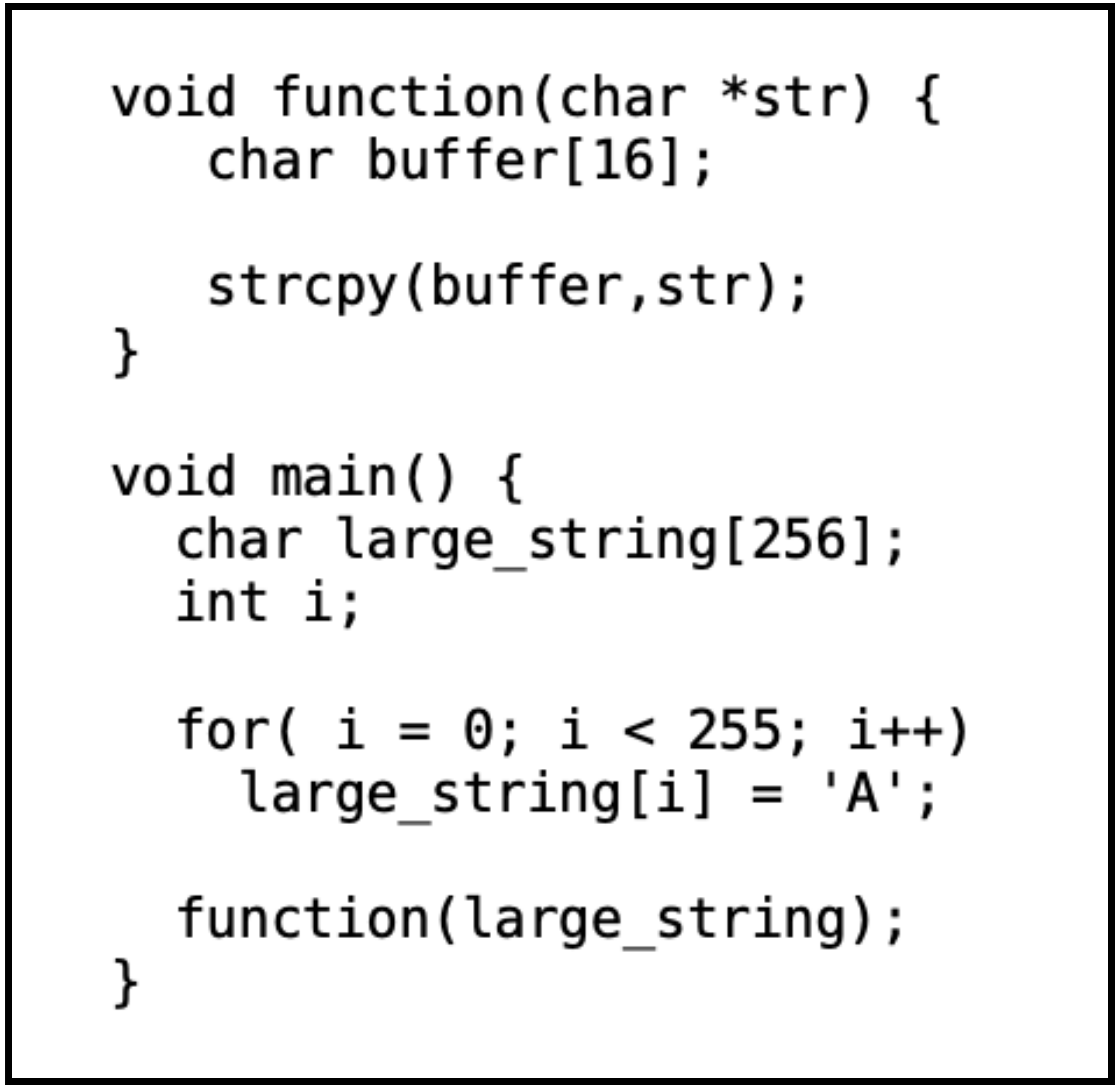
```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

What’s wrong with this function?



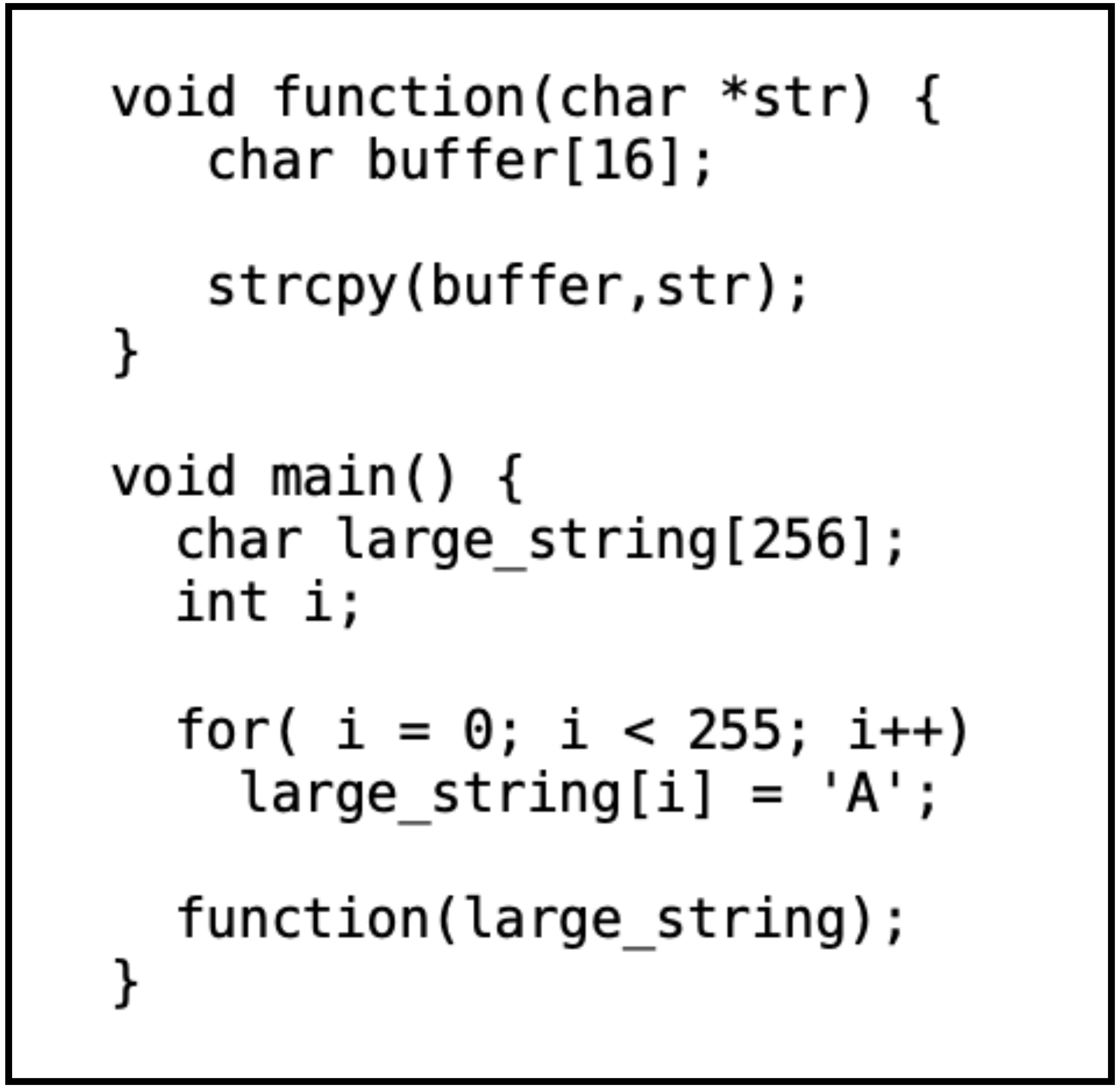
```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

Where is the return address on the stack?



```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

What is the return address written to?



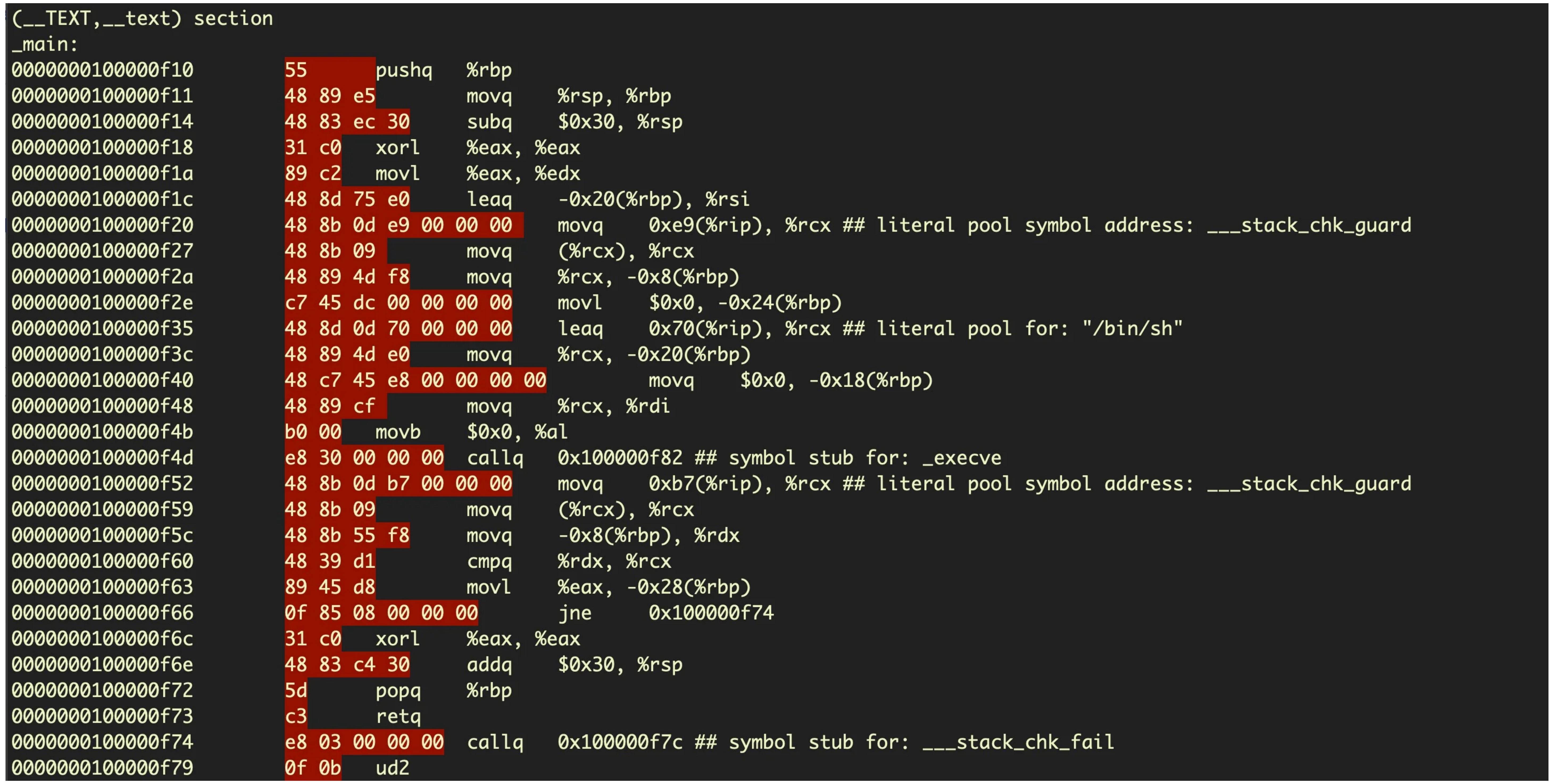
```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

Return Address: 0x41414141

What is shellcode?

What is shellcode?

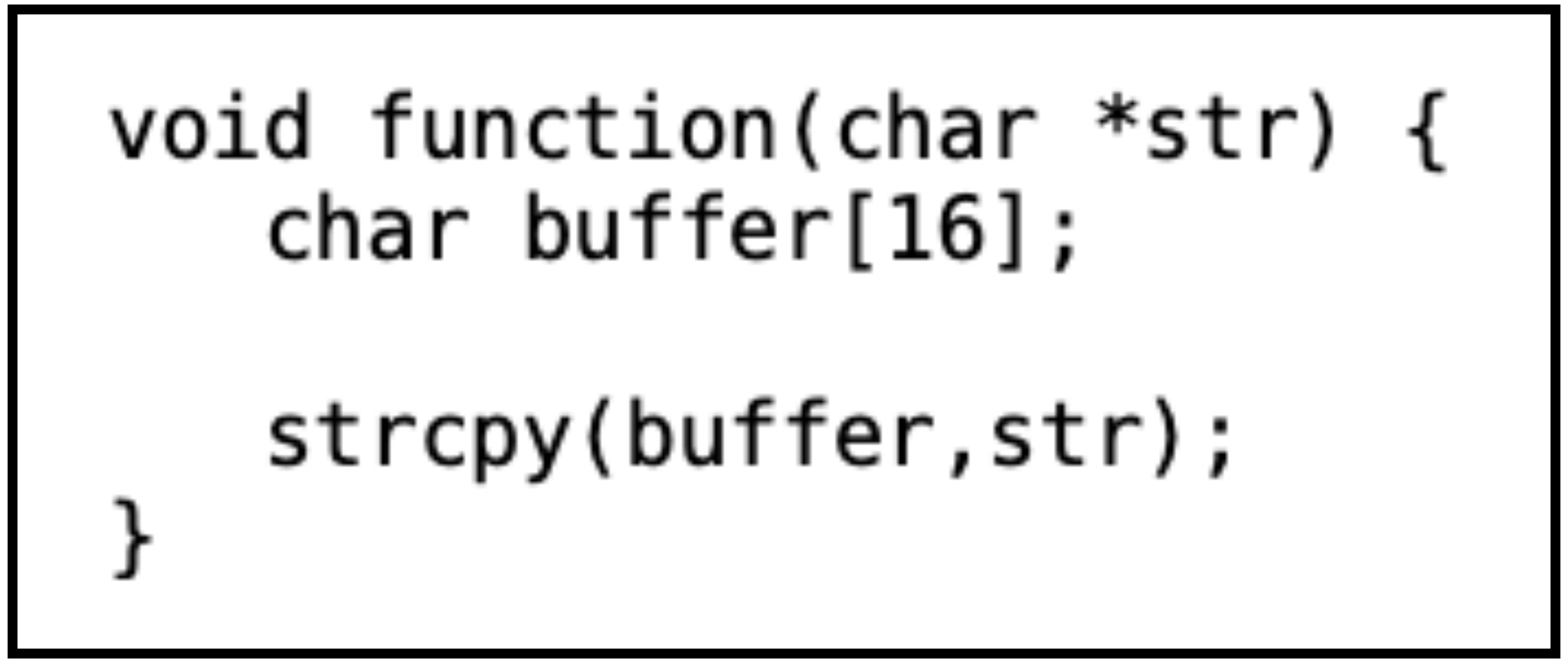
```c
#include <stdio.h>
int main()
{
    char *args[2];
    args[0] = "/bin/sh";
    args[1] = NULL;
    execve("/bin/sh", args, NULL);
    return 0;
}
```



```
(__TEXT,__text) section
_main:
0000000100000f10        55      pushq   %rbp
0000000100000f11        48 89 e5        movq    %rsp, %rbp
0000000100000f14        48 83 ec 30     subq    $0x30, %rsp
0000000100000f18        31 c0   xorl    %eax, %eax
0000000100000f1a        89 c2   movl    %eax, %edx
0000000100000f1c        48 8d 75 e0     leaq    -0x20(%rbp), %rsi
0000000100000f20        48 8b 0d e9 00 00 00    movq    0xe9(%rip), %rcx ## literal pool symbol address: ___stack_chk_guard
0000000100000f27        48 8b 09        movq    (%rcx), %rcx
0000000100000f2a        48 89 4d f8     movq    %rcx, -0x8(%rbp)
0000000100000f2e        c7 45 dc 00 00 00 00    movl    $0x0, -0x24(%rbp)
0000000100000f35        48 8d 0d 70 00 00 00    leaq    0x70(%rip), %rcx ## literal pool for: "/bin/sh"
0000000100000f3c        48 89 4d e0     movq    %rcx, -0x20(%rbp)
0000000100000f40        48 c7 45 e8 00 00 00 00         movq    $0x0, -0x18(%rbp)
0000000100000f48        48 89 cf        movq    %rcx, %rdi
0000000100000f4b        b0 00   movb    $0x0, %al
0000000100000f4d        e8 30 00 00 00  callq   0x100000f82 ## symbol stub for: _execve
0000000100000f52        48 8b 0d b7 00 00 00    movq    0xb7(%rip), %rcx ## literal pool symbol address: ___stack_chk_guard
0000000100000f59        48 8b 09        movq    (%rcx), %rcx
0000000100000f5c        48 8b 55 f8     movq    -0x8(%rbp), %rdx
0000000100000f60        48 39 d1        cmpq    %rdx, %rcx
0000000100000f63        89 45 d8        movl    %eax, -0x28(%rbp)
0000000100000f66        0f 85 08 00 00 00       jne     0x100000f74
0000000100000f6c        31 c0   xorl    %eax, %eax
0000000100000f6e        48 83 c4 30     addq    $0x30, %rsp
0000000100000f72        5d      popq    %rbp
0000000100000f73        c3      retq
0000000100000f74        e8 03 00 00 00  callq   0x100000f7c ## symbol stub for: ___stack_chk_fail
0000000100000f79        0f 0b   ud2
```

Executing shellcode in vulnerable code

- Let's say I have some shellcode instructions and the function to the right. How might I execute the shellcode?



```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}
```

Smashing the Stack for Fun and Profit

- *Attacker controlled buffer* can be overrun to overwrite *return address* to jump to any other point in the stack
- If that point in the stack has *valid instructions*, the CPU will start running from there
  - E.g., *shellcode*
- You can overwrite lots of things
  - Another local variable, saved frame pointer, function arguments, even **deeper stack frames,** exception control data…. **anything that is valid to write to on the stack!**

Why does this happen?

Why does this happen?

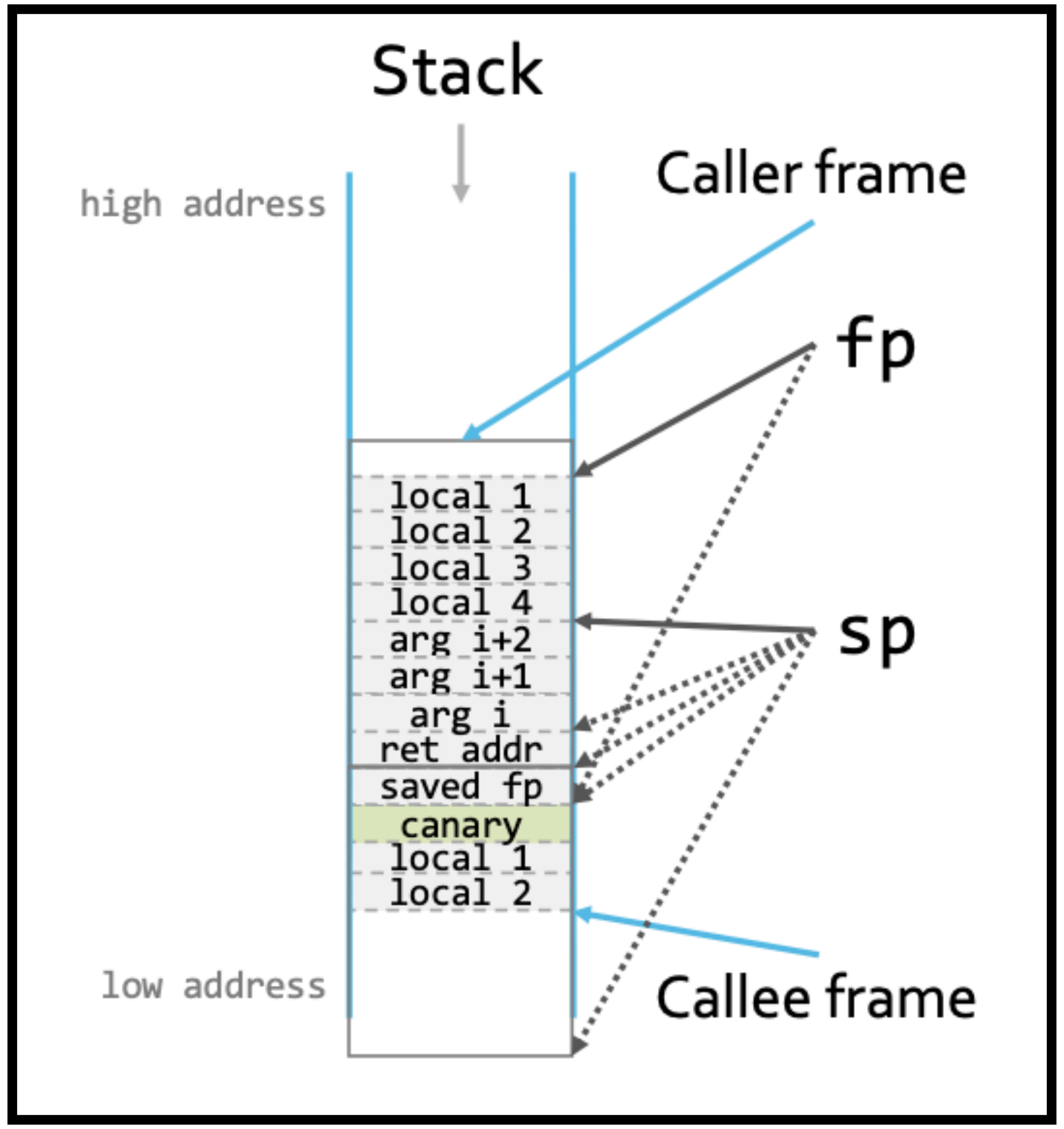
- The C language is *weakly typed*
  - Allows writing arbitrary values to arbitrary locations in memory (e.g., all arrays are the same under the hood, it's just bytes)
- Control flow is dynamic and based on *memory*
  - Return addresses, function pointers, jump tables
  - If you overwrite these you can change control flow
- The processor **doesn't know the difference between code and data**
  - This is a *common issue* in computer security, not just software security
  - Where else?

5-Minute exercise: Defenses against buffer overflows

- Can we *detect* the overwriting of the return address? How?

One idea: Canaries

- Can we *detect* the overwriting of the return address? How?
  - Use a **canary** – a value the callee pushes before the return address and *check* to make sure it aligns with what you're expecting
  - When returning, the callee checks canary against a global "gold" copy stored as a constant (not on the stack)



Stack Canary Limitations

- What **assumptions** am I making about stack canaries that make them useful?

Stack Canary Limitations

- What **assumptions** am I making about stack canaries that make them useful?
  - **Assumption:** impossible to subvert control flow without corrupting the canary

Stack Canary Limitations

- What **assumptions** am I making about stack canaries that make them useful?
  - **Assumption:** impossible to subvert control flow without corrupting the canary
- Can we overwrite the canary with a valid canary value?

Stack Canary Limitations

- What **assumptions** am I making about stack canaries that make them useful?
  - **Assumption:** impossible to subvert control flow without corrupting the canary
- Can we overwrite the canary with a valid canary value?
  - Sure, if you can read or guess the value

Stack Canary Limitations

- What **assumptions** am I making about stack canaries that make them useful?
  - **Assumption:** impossible to subvert control flow without corrupting the canary
- Can we overwrite the canary with a valid canary value?
  - Sure, if you can read or guess the value
- Do I always need to overwrite the canary?

Stack Canary Limitations

- What **assumptions** am I making about stack canaries that make them useful?
  - **Assumption:** impossible to subvert control flow without corrupting the canary
- Can we overwrite the canary with a valid canary value?
  - Sure, if you can read or guess the value
- Do I always need to overwrite the canary?
  - No, what if the function uses **pointers**? What if you can overwrite the address of a data pointer to point directly at the saved return address? Then writes through that pointer will modify the return address without touching the canary.

Break Time + Attendance

Codeword:

Stacking-Pancakes

https://tinyurl.com/cse227-attend

The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86)

Defenses against code vs. data

- W^X (W *xor* X)
  - Memory protection policy whereby every *page* in an address space is either writeable or executable but **not both**
  - Why does this prevent the attacks we discovered previously?

Return-to-libc

- What is a return-to-libc attack?

Return-to-libc

- What is a return-to-libc attack? *Return control to system functions to execute shellcode*

Return-to-libc

- What is a return-to-libc attack? *Return control to system functions to execute shellcode*
- What are some issues with the return-to-libc attack that make it hard to exploit (in theory?)

Return-to-libc

- What is a return-to-libc attack? *Return control to system functions to execute shellcode*
- What are some issues with the return-to-libc attack that make it hard to exploit (in theory?)
  - “Straight line limited” – means you can only enter into one libc function after another
  - “Removal limited” – if you remove libc function that aren’t useful, you can seriously hamper attackers

Return-to-libc

- What is a return-to-libc attack? *Return control to system functions to execute shellcode*

- What are some issues with the return-to-libc attack that make it hard to exploit (in theory?)

    - “Straight line limited” – means you can only enter into one libc function after another

    - “Removal limited” – if you remove libc function that aren’t useful, you can seriously hamper attackers

- This paper: **Those assumptions are wrong, you don’t even need functions!**

Return-Oriented Programming

- This paper demonstrates that you don't even need function calls, but all you need are *micro sequences of instructions* to mess with control flow of a program
- What is the fundamental insight about x86 that enables this attack?

Return-Oriented Programming

- This paper demonstrates that you don’t even need function calls, but all you need are *micro sequences of instructions* to mess with control flow of a program
- What is the fundamental insight about x86 that enables this attack?
  - x86 instructions are **ambiguous** and **dense**, so shifting by a single byte often leads to interesting strings of instructions
  - All you need is **ret** to chain gadgets together

Return-Oriented Programming

- This paper demonstrates that you don't even need function calls, but all you need are *micro sequences of instructions* to mess with control flow of a program
- What is the fundamental insight about x86 that enables this attack?
    - x86 instructions are **ambiguous** and **dense**, so shifting by a single byte often leads to interesting strings of instructions
    - All you need is **ret** to chain gadgets together
- Is this true in all architectures?

Return-Oriented Programming

- What is return-oriented programming?
- How do you execute return-oriented programming?

Return-Oriented Programming

- What is return-oriented programming?
- How do you execute return-oriented programming?
  - Processor executes a ret with %esp (stack pointer) pointing to the bottom word of the gadget, serves as a sort of “instruction pointer”

Gadgets Galore

- What is a *useful* gadget in this paper?

Gadgets Galore

- What is a *useful* gadget in this paper?
    - Anything that ends w/ ret and doesn’t alter control flow is good (because we can set up the stack the way we like!)

Gadgets Galore

- What is a *useful* gadget in this paper?
  - Anything that ends w/ ret and doesn’t alter control flow is good (because we can set up the stack the way we like!)
- Why do these gadgets exist?

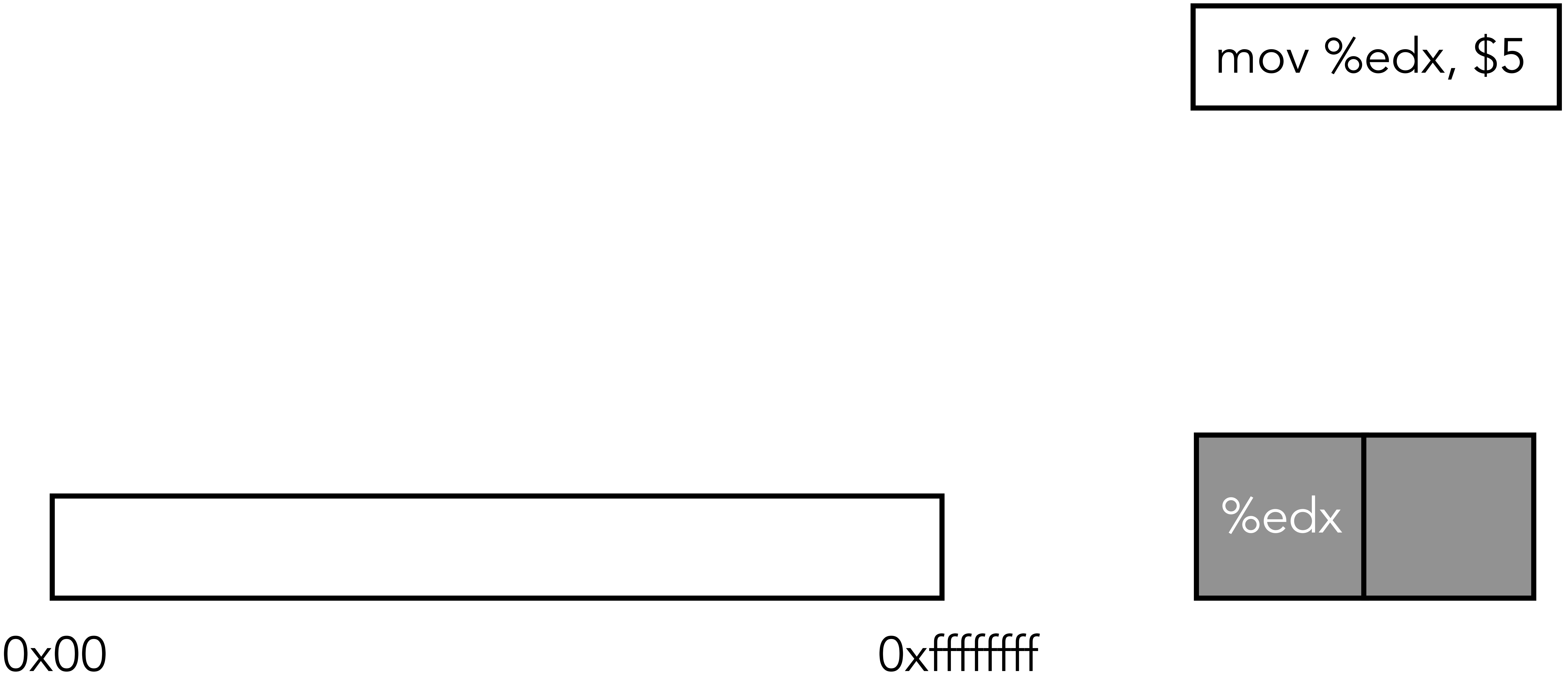
Gadgets Galore

- What is a *useful* gadget in this paper?
  - Anything that ends w/ ret and doesn't alter control flow is good (because we can set up the stack the way we like!)
- Why do these gadgets exist?
  - Compilers commonly add them at the end of a function! Very hard to avoid.

Gadgets Galore

- What is a *useful* gadget in this paper?
  - Anything that ends w/ ret and doesn’t alter control flow is good (because we can set up the stack the way we like!)
- Why do these gadgets exist?
  - Compilers commonly add them at the end of a function! Very hard to avoid.
- Can build arbitrary new bad programs that are made completely out of “known good” instructions (e.g., libc)

Simple example



`mov %edx, $5`

%edx

0x00

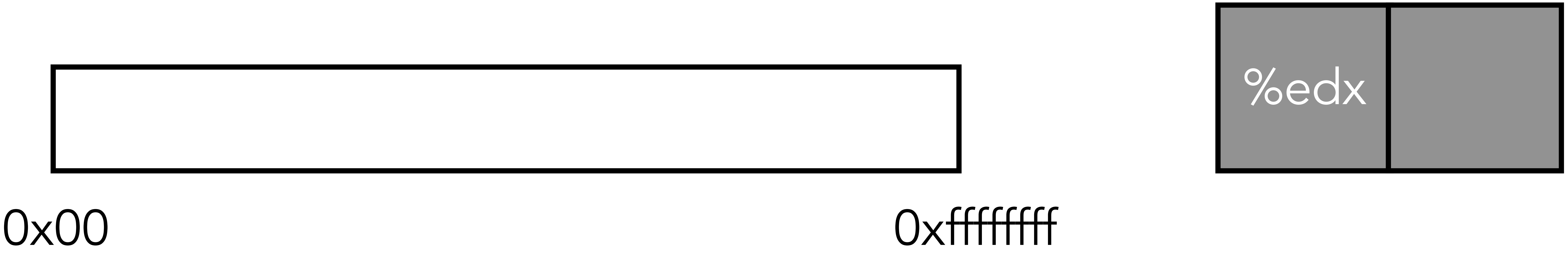
0xffffffff

stolen w/ love from UMD

Simple example

What does this piece of assembly do?

```
mov %edx, $5
```

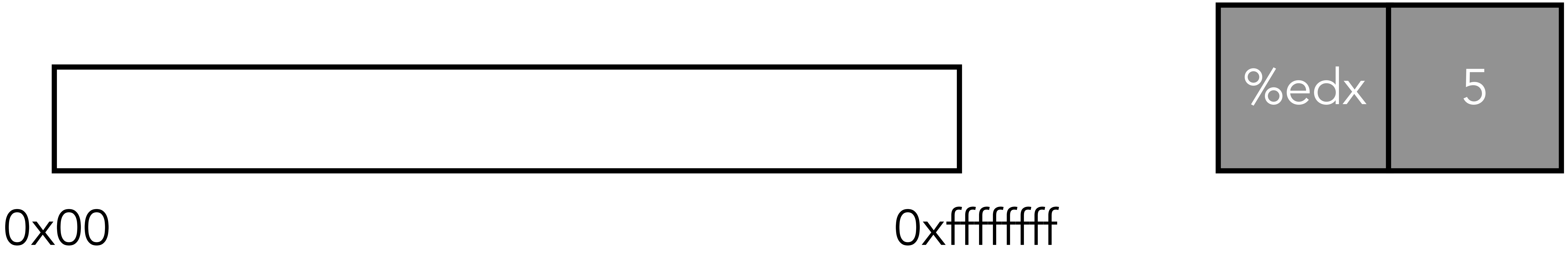


stolen w/ love from UMD

Simple example

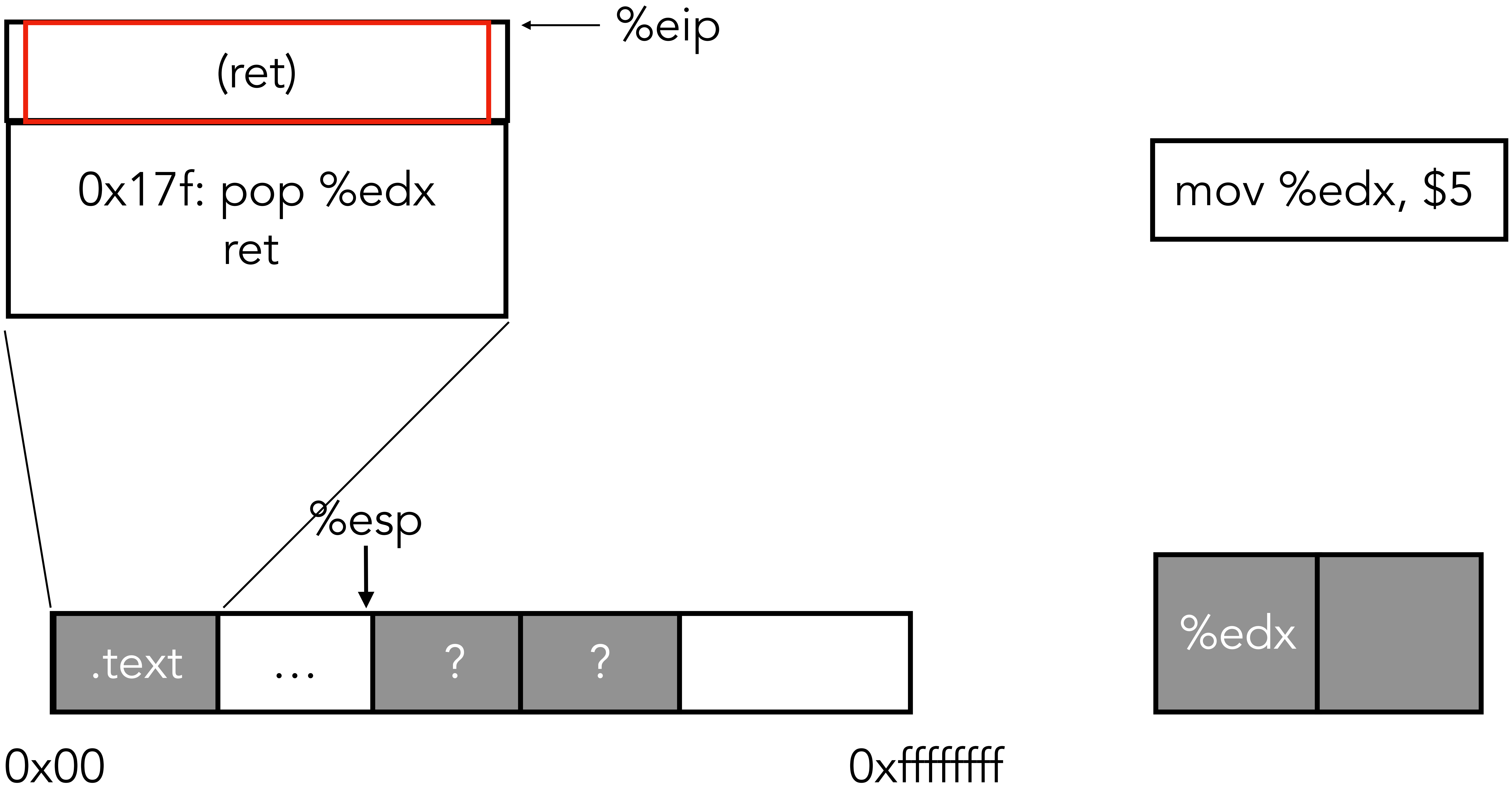
What does this piece of assembly do?

```
mov %edx, $5
```



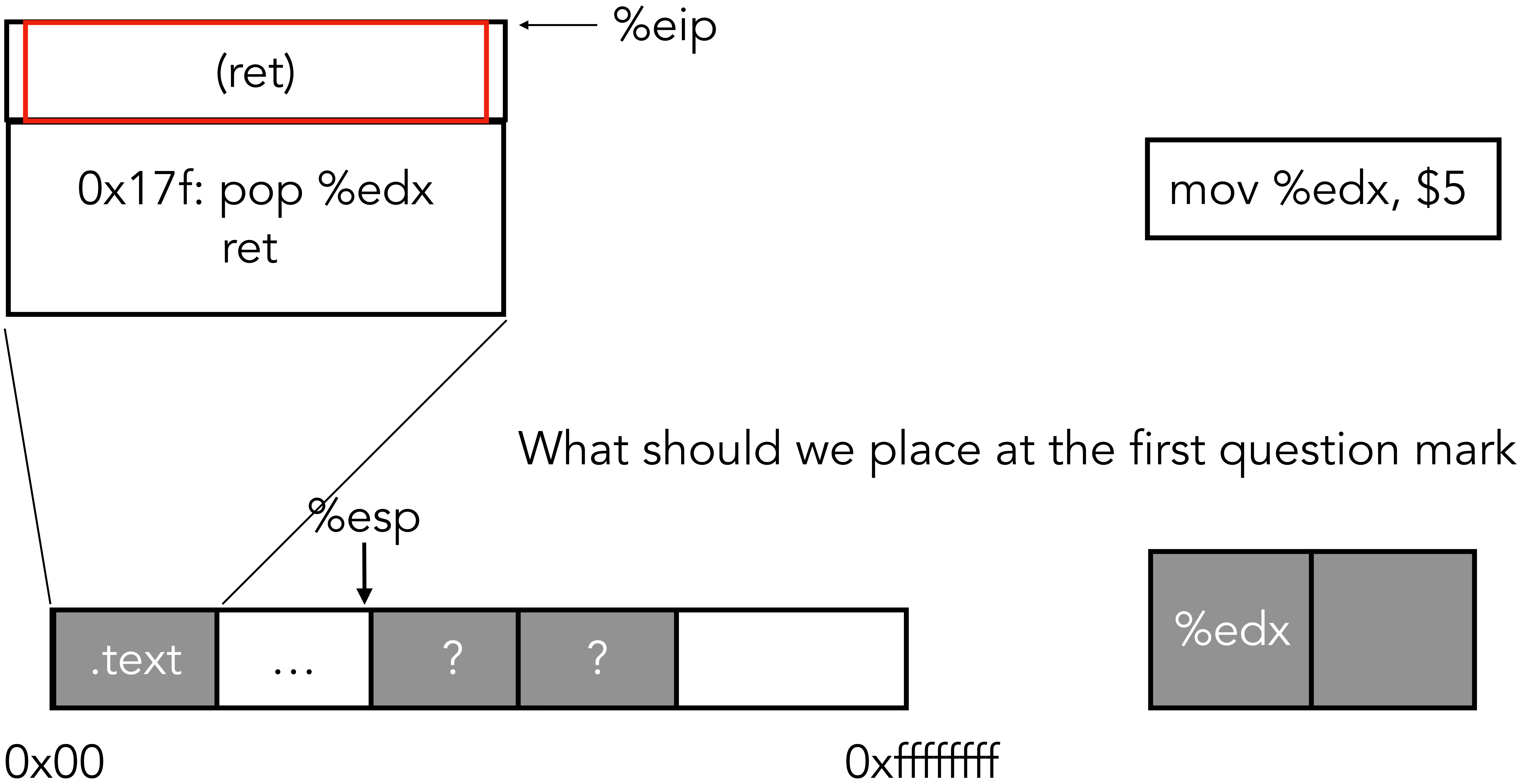
stolen w/ love from UMD

Simple example



stolen w/ love from UMD

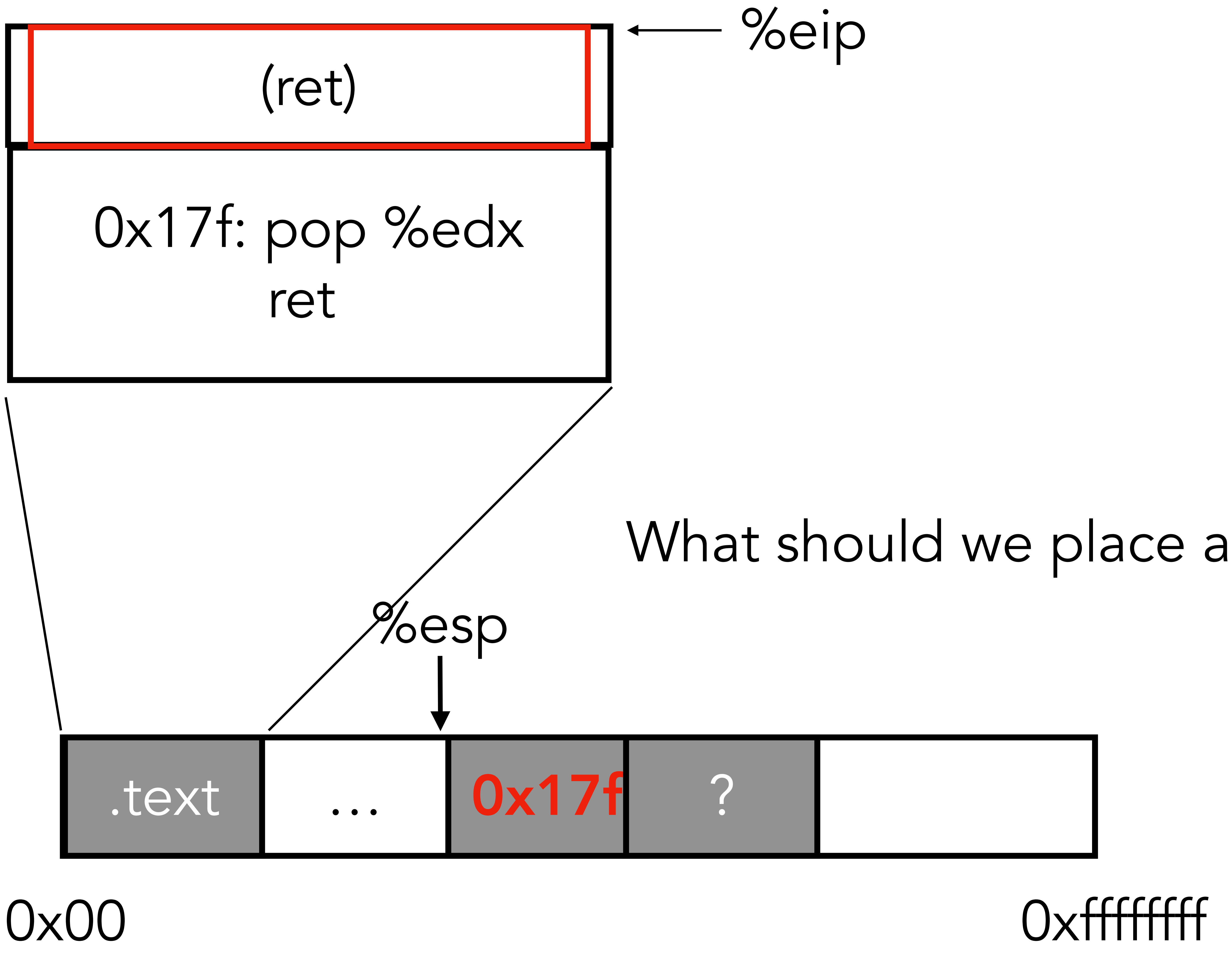
Simple example



What should we place at the first question mark?

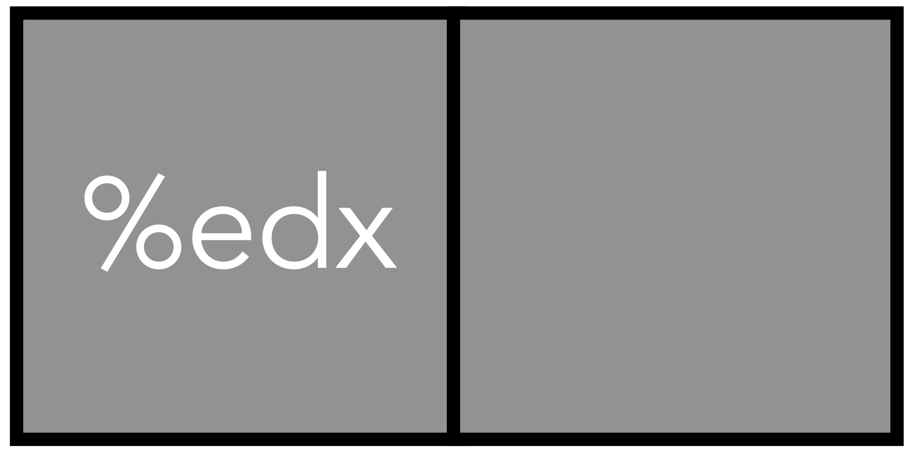
stolen w/ love from UMD

Simple example



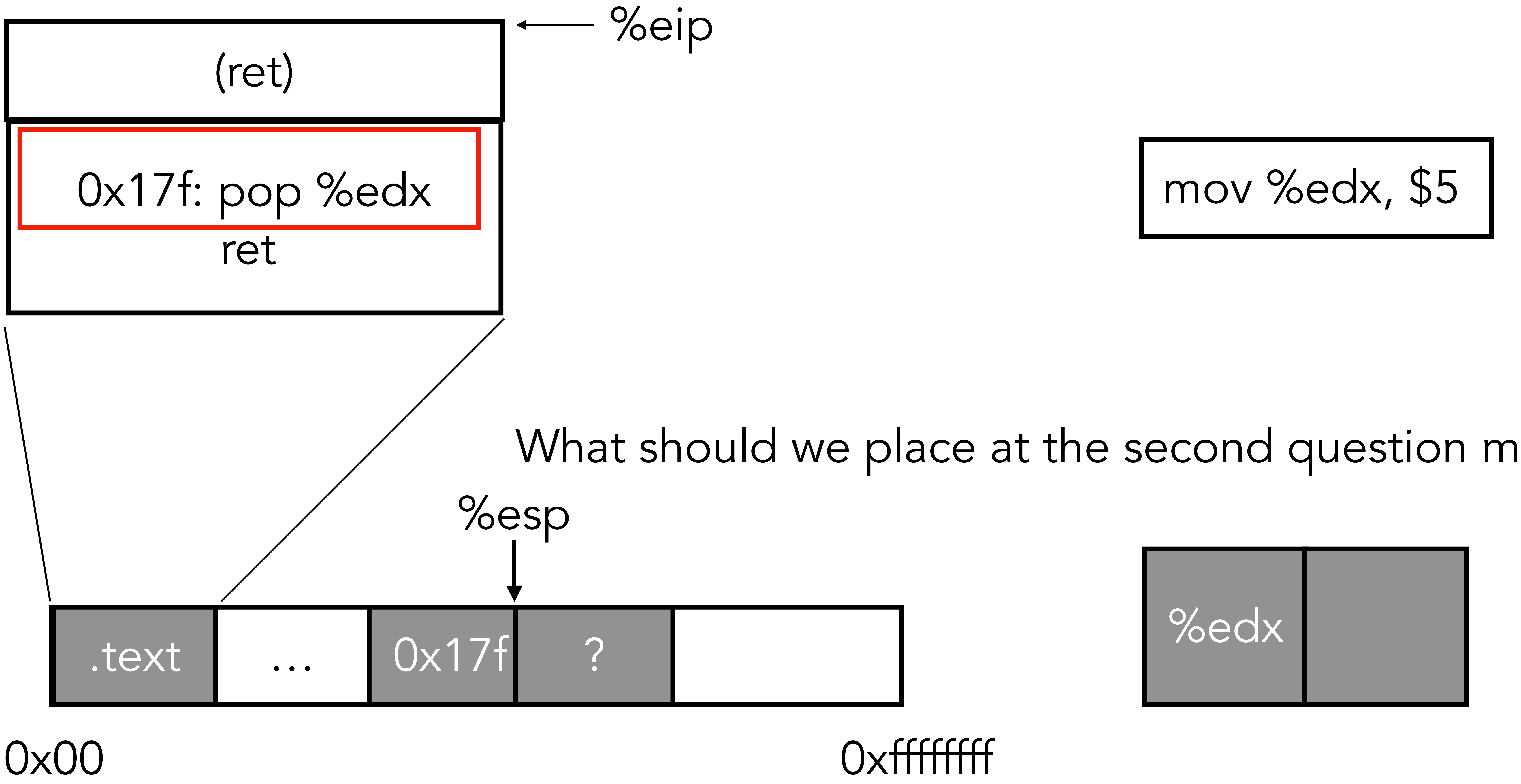
mov %edx, $5

What should we place at the first question mark?



stolen w/ love from UMD

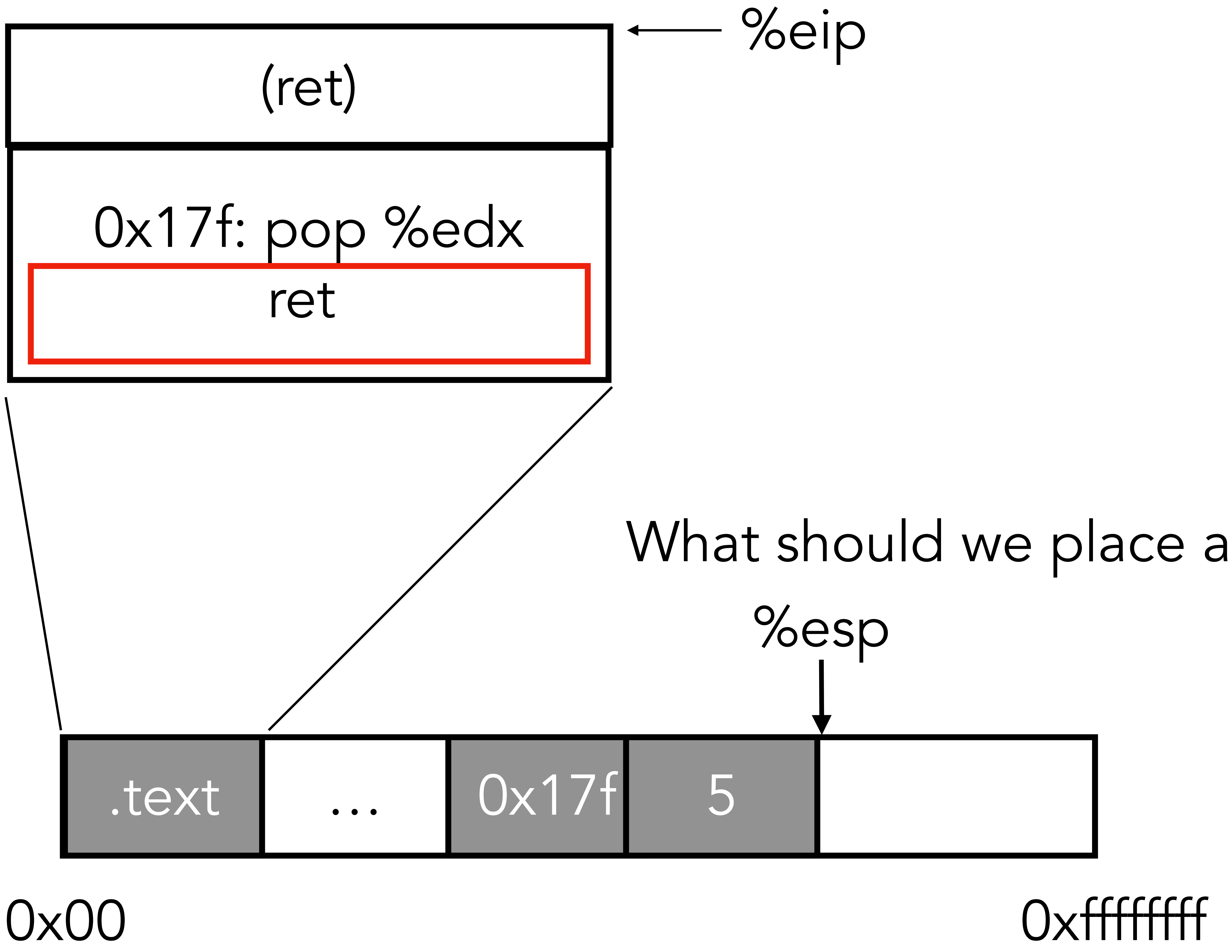
Simple example



What should we place at the second question mark?

stolen w/ love from UMD

Simple example

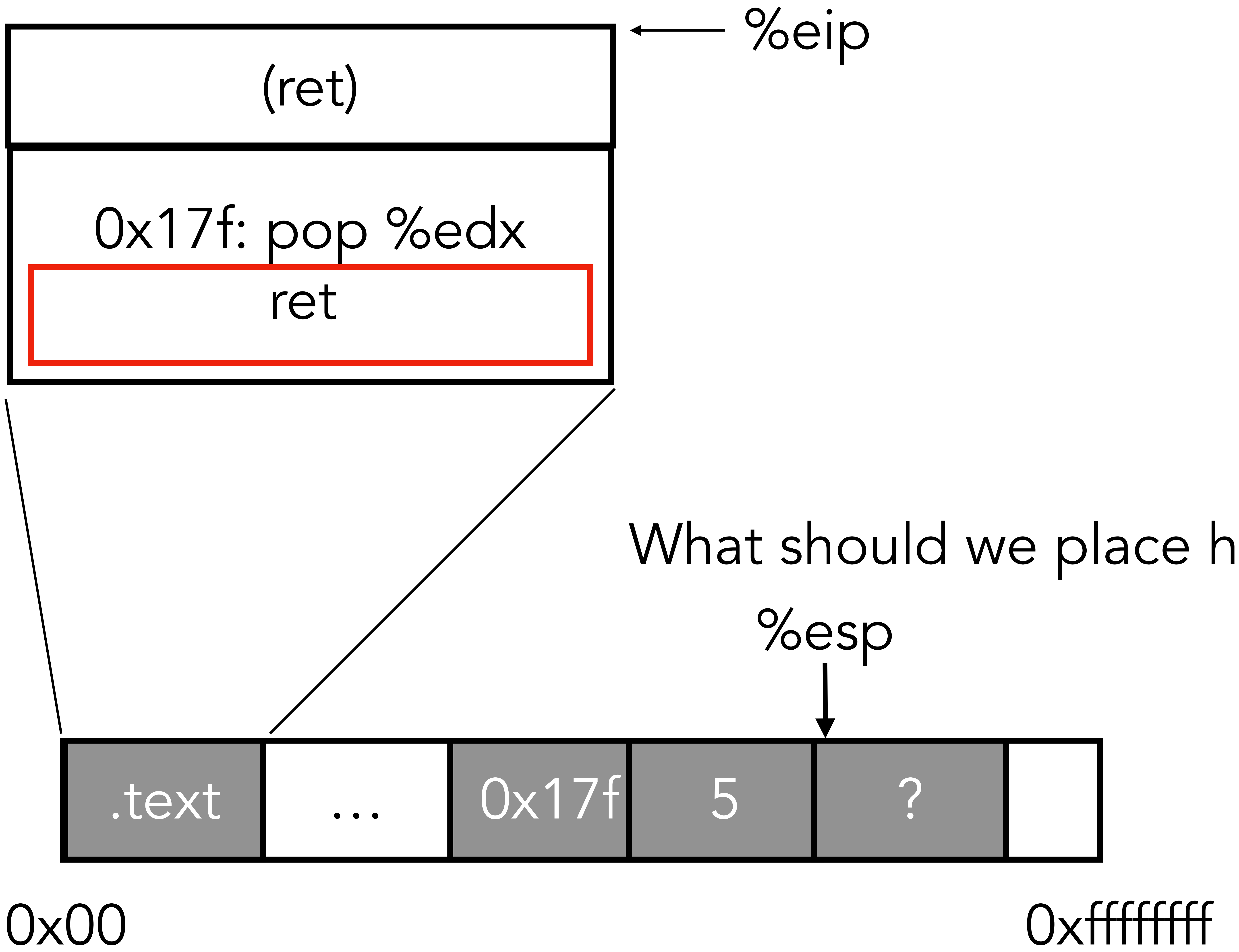


mov %edx, $5

What should we place at the second question mark?

stolen w/ love from UMD

Simple example



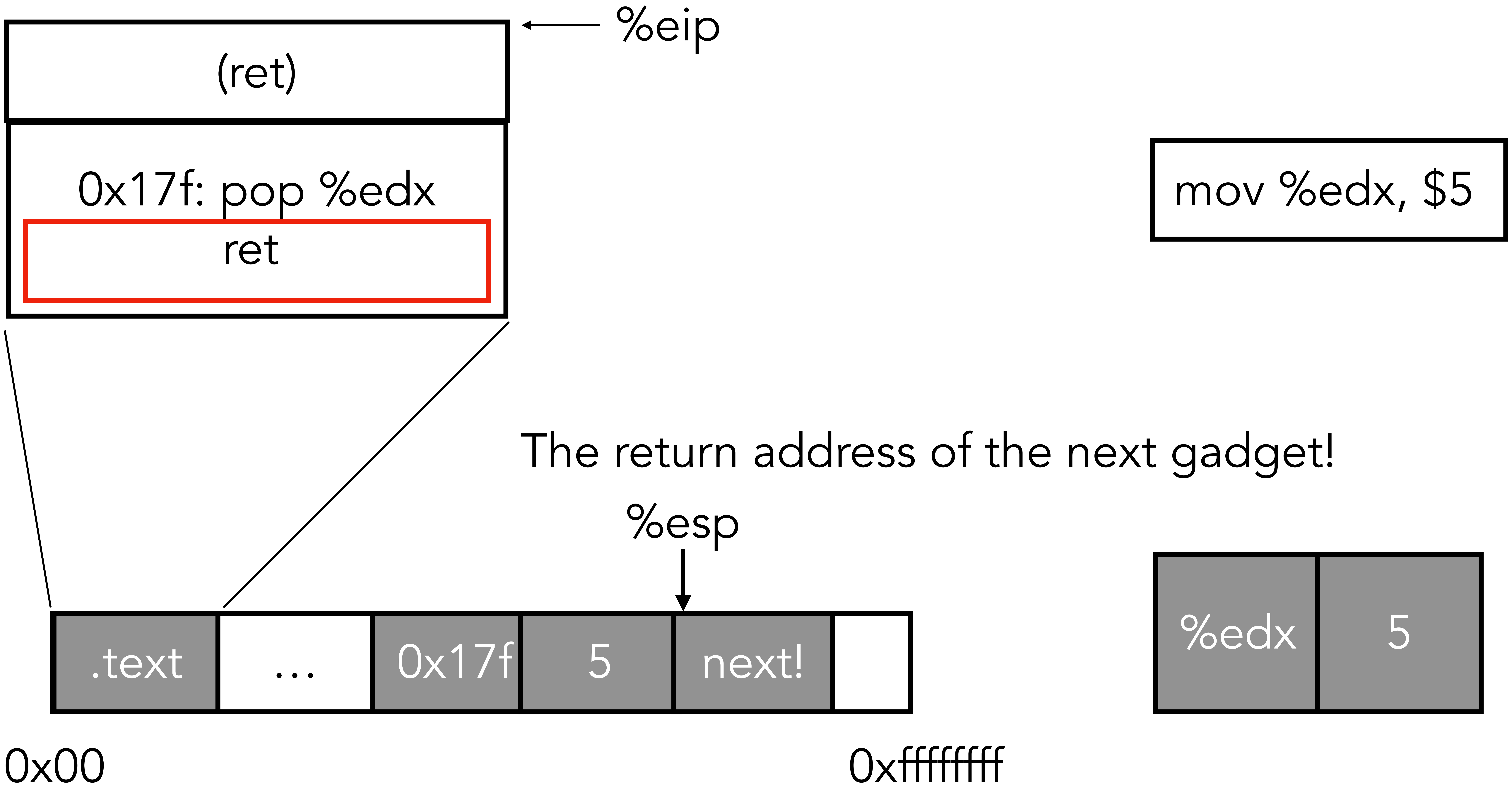
What should we place here?

mov %edx, $5

%edx	5

stolen w/ love from UMD

Simple example



The return address of the next gadget!

stolen w/ love from UMD

Making ROP Hard

- What are some assumptions made about the *location* of libc functions that make ROP possible?

Making ROP Hard

- What are some assumptions made about the *location* of libc functions that make ROP possible?
  - libc is in a fixed location: **not true with Address Space Layout Randomization (ASLR)**

Making ROP Hard

- What are some assumptions made about the *location* of libc functions that make ROP possible?
  - libc is in a fixed location: **not true with Address Space Layout Randomization (ASLR)**
- Control flow integrity (CFI)
  - Check at run-time if the execution path is allowed by the original program
  - Insert “tags” before each branch target when branching, and first check the target’s tag matches expectation
  - Like stack canaries, but for *control flow* rather than *data protection*

Return-Oriented Programming

is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from .text segments

Discussion

What about these attacks *surprised* you?

What do these attacks teach us about *trust*?

Code vs. Data is a fundamental security issue. Why?

For next time…

- Make sure you submit your project intention form! Due tomorrow, 1/17
- Read two side channels papers (course webpage has been updated post illness) and be ready to discuss them
- Come chat with me about your projects, if you want them to be good :)