CS 241: Systems Programming
Lecture 11. Ownership in Rust

Fall 2025

Prof. Stephen Checkoway

Memory safety

Data in programs are stored in memory (RAM)

One reasonable way to think about RAM is as a giant array of bytes

All of the data (and the program code!) is stored somewhere in that array

When you create an i64 variable, 8 bytes of memory are allocated somewhere in the array for that variable

When you create a String, some bytes of memory hold the contents of the string, some bytes of memory hold a **pointer** to the contents, some other bytes hold the length of the string

Memory safety

It’s critical that it’s not possible to confuse which bytes are which

E.g., if our program can become confused about whether some memory is an i64 or is a pointer to our string contents, *anything could happen*!

- E.g., changing the i64 could cause the pointer to change and point at something that’s not a string or some region of memory that isn’t allocated at all

Memory safety is all about ensuring that it’s impossible for these sorts of errors to occur

Memory safety and Rust

Rust ensures that program are memory safe, e.g.,

- It's impossible to confuse a pointer with an integer
- It's impossible to access out-of-bounds data in an array/Vec

Most modern languages (Python, Java, Go, Haskell, Ruby, etc.) are memory-safe

Most systems languages (C and C++) are not!

- Memory safety errors are common and lead to real harm

Ownership

Rust ensures memory safety through a concept of ownership

These are rules that the rust compiler enforces to prevent **undefined behavior**

Stack frames

Variables live in a region of memory called the stack

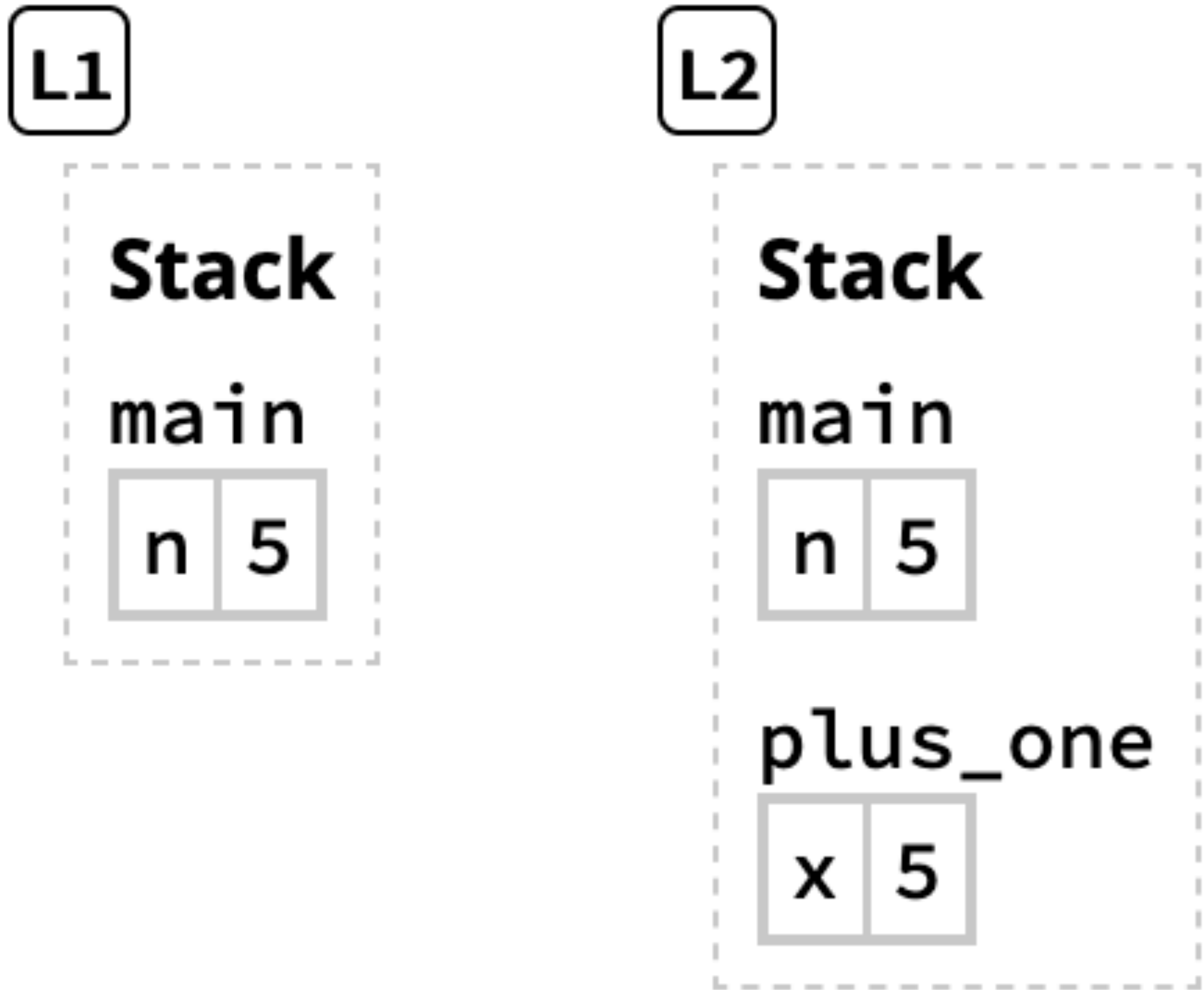
The stack is organized into **frames**

Local variables in functions live in a stack frame

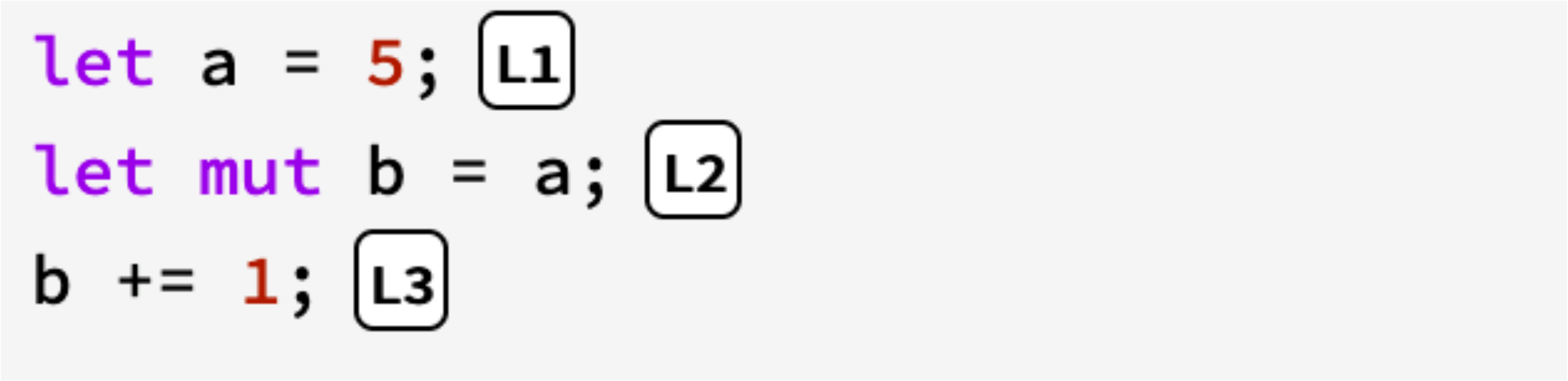
Each function that is called pushes a new frame onto the stack to hold its local variables

Each function that returns pops its stack frame off the stack

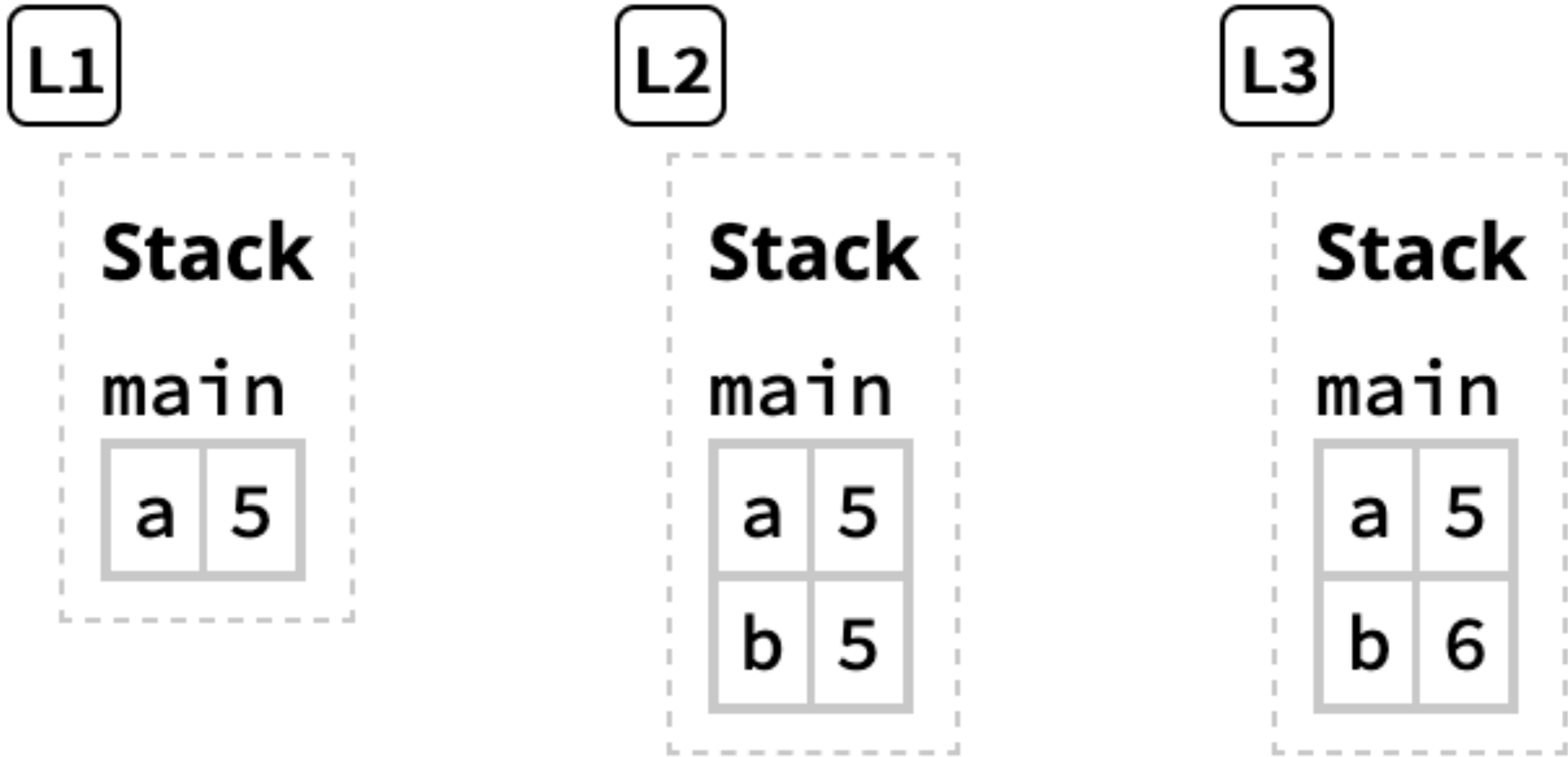
```
fn main() {
    let n = 5; L1
    let y = plus_one(n); L3
    println!("The value of y is: {y}");
}

fn plus_one(x: i32) -> i32 {
    L2 x + 1
}
```



Every variable gets its own slot



```
let a = 5; L1
let mut b = a; L2
b += 1; L3
```



L1

Stack

main

a	5

L2

Stack

main

a	5
b	5

L3

Stack

main

a	5
b	6

```
fn foo() {
    // What does the stack look
    // like in this function…
}

fn bar() {
    foo();
}

fn main() {
    foo();
    bar();
    foo(); // …when called here?
}
```

A. main

B. foo

C. main
   foo

D. main
   bar
   foo

E. main
   foo
   bar
   foo
   foo

Local variables

Local variables in functions live on the stack (in a stack frame)

When the function returns, variables in the stack frame for the function are **dropped**

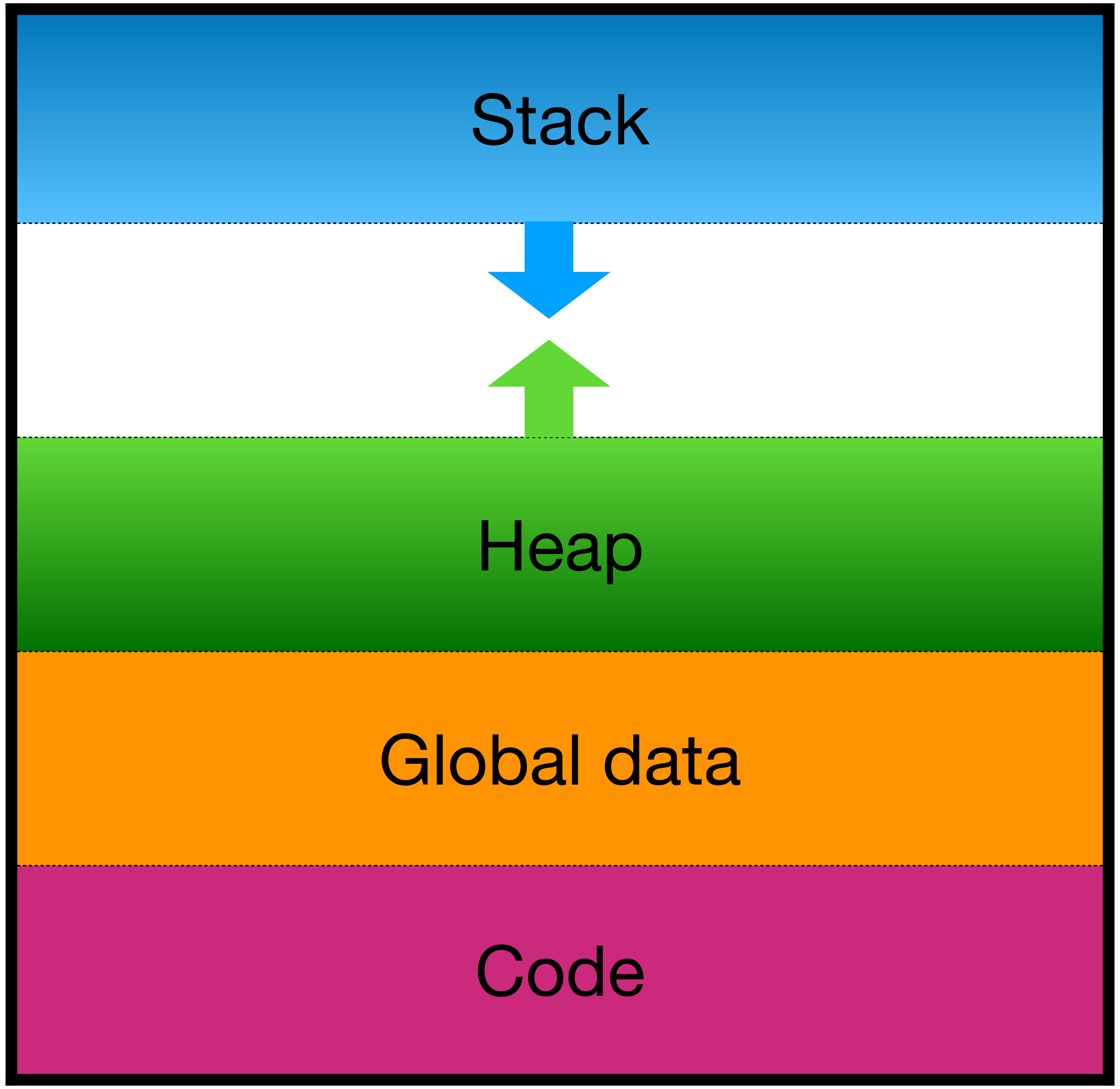
Once a variable is dropped, it can no longer be accessed

Returning a value from a function copies it into the stack frame of the function being returned to

Other than returning a variable, there's no way for the *variable* to live longer than the function (data can live longer as we're about to see)

Memory layout (simplified)

High memory address



Low memory address

Stack grows down

Heap grows up

Global data and
Code are fixed size

Heap

Data in the heap lives longer than an individual function

Strings and Vecs store their contents on the heap

A String or Vec variable holds a **pointer** to the contents

Any data type that needs to hold a variable amount of data works the same way:

- Contents in the heap
- Pointers to the contents

Pointers

A pointer says where data can be located in memory

At a hardware level, a pointer is nothing more than an index into memory where the data can be found

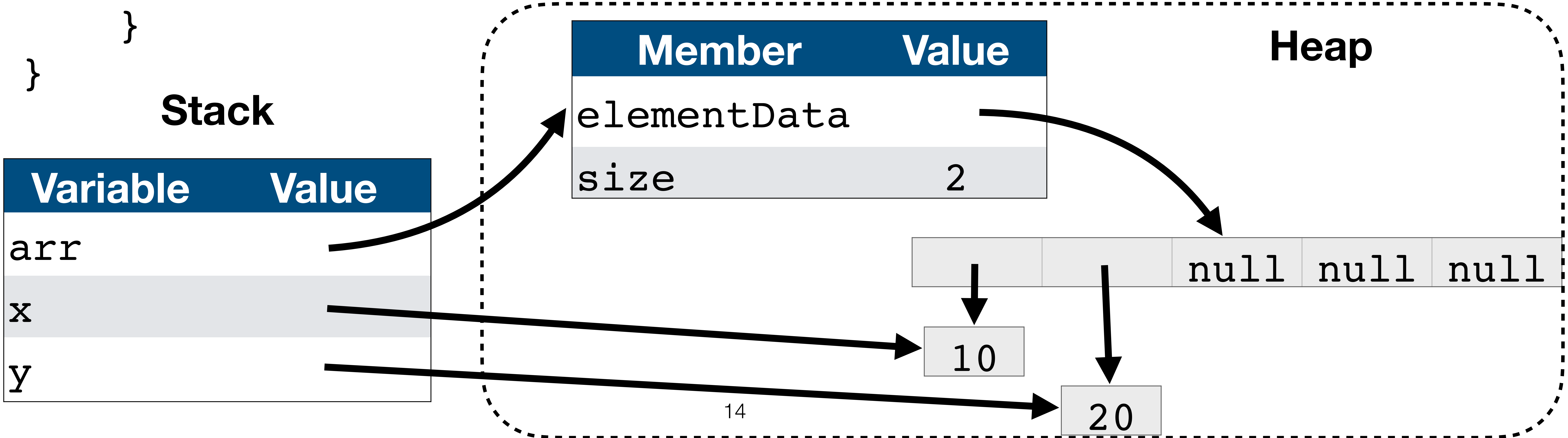
In Java, every Object lives in the heap and is accessed via a pointer

- The variables are pointers

In Rust, objects can live on the stack or in the heap

- Many objects (like String and Vec) contain pointers to heap memory

```
import java.util.ArrayList;

public class A {
    public static void main(String[] args) {
        ArrayList<Integer> arr = new ArrayList<Integer>();
        Integer x = new Integer(10);
        Integer y = new Integer(20);
        arr.add(x);
        arr.add(y);
        System.out.println(arr.get(1));
    }
}
```



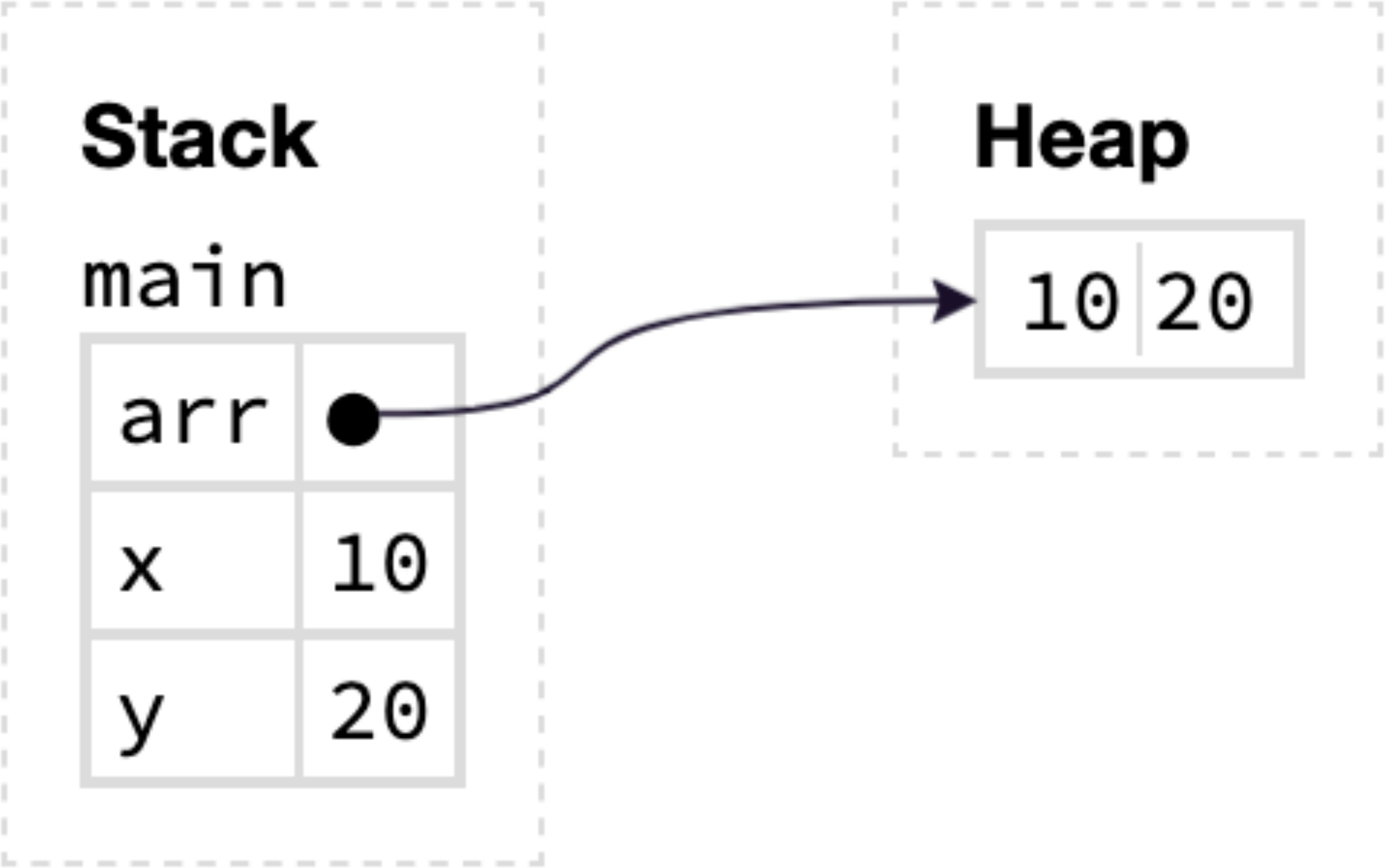
Stack

Variable	Value
arr	
x	
y	

Heap

Member	Value
elementData	
size	2

```rust
fn main() {
    let mut arr: Vec<i32> = Vec::new();
    let x = 10;
    let y = 20;
    arr.push(x);
    arr.push(y);
    println!("{arr:?}");
}
```



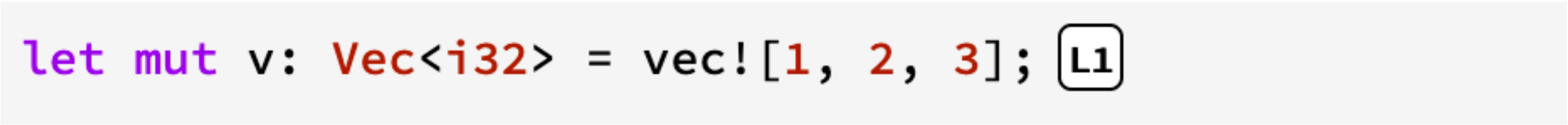
Stack

main

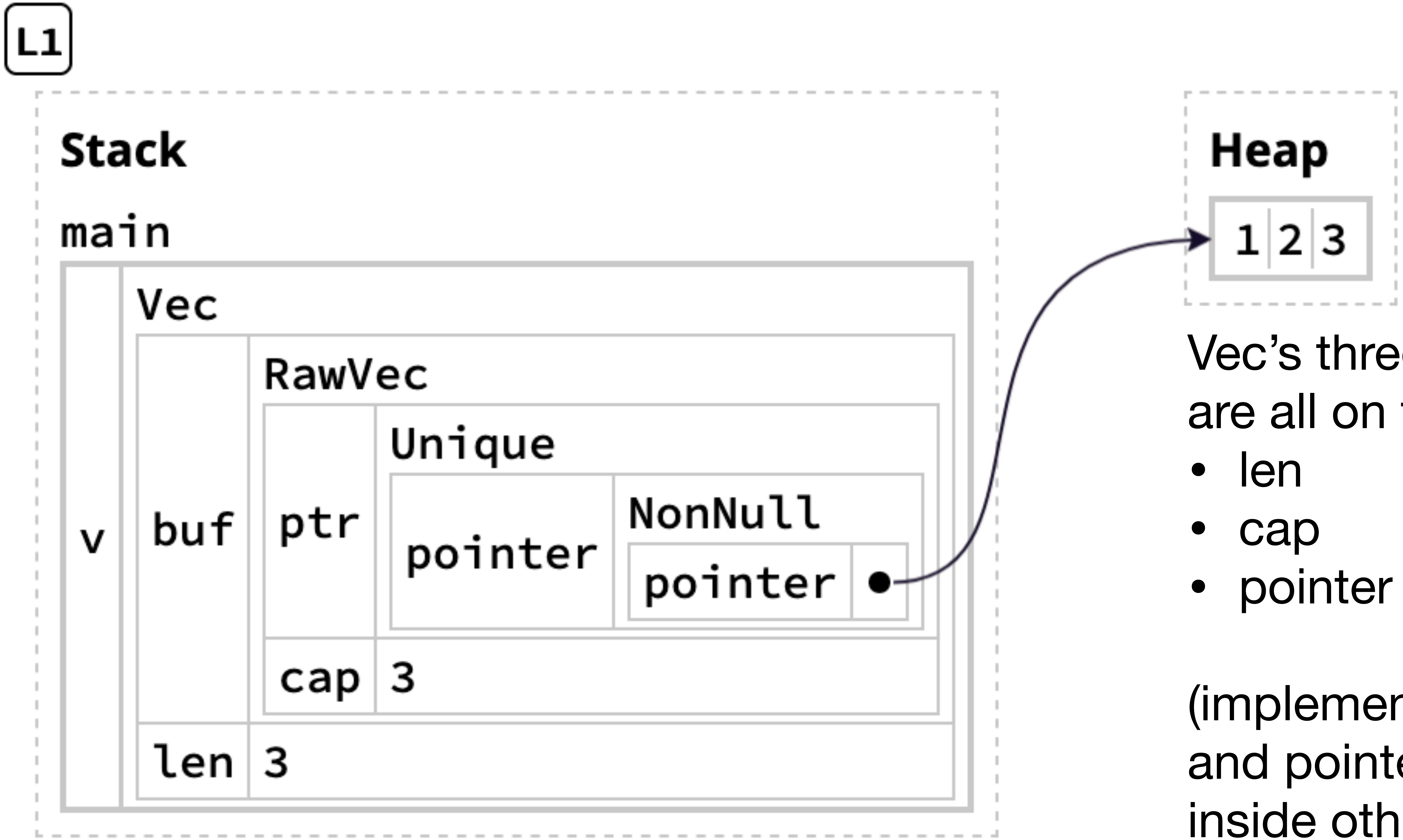
arr	●
x	10
y	20

Heap

10	20



```
let mut v: Vec<i32> = vec![1, 2, 3]; L1
```



Vec’s three members are all on the stack

- len
- cap
- pointer

(implementation detail: cap and pointer are nested inside other structures, but still on the stack)

Which of the following statements are true?
1. A local variable in a function can outlive the function.
2. Data in the heap can outlive the function that created it.
3. Variable-length data (usually) live in the heap.
4. Data on the heap is accessed using pointers

A. 1 and 2

B. 1, 2, and 3,

C. 3, and 4,

D. 2, 3, and 4

E. 1, 2, 3, 4, and 5

Boxes—an owning pointer

We can store data in the heap by putting it in a Box

```
let b: Box<[i64; 1000]> = Box::new([42; 1000]);
let b2: Box<usize> = Box::new(37);
```

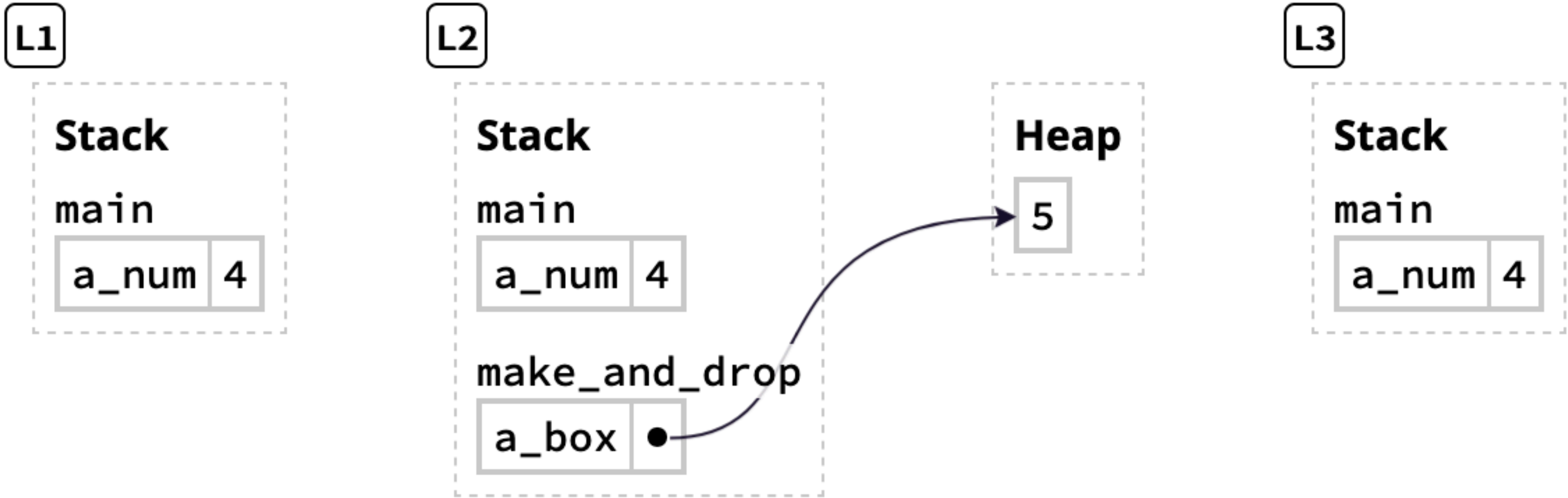
A Box is a type of pointer that always points to valid data in the heap

The Box owns the data it points to

When a Box variable is dropped (e.g., because the function whose frame contains the variable returns), the data in the heap is **freed**

Once data has been freed, it is no longer accessible

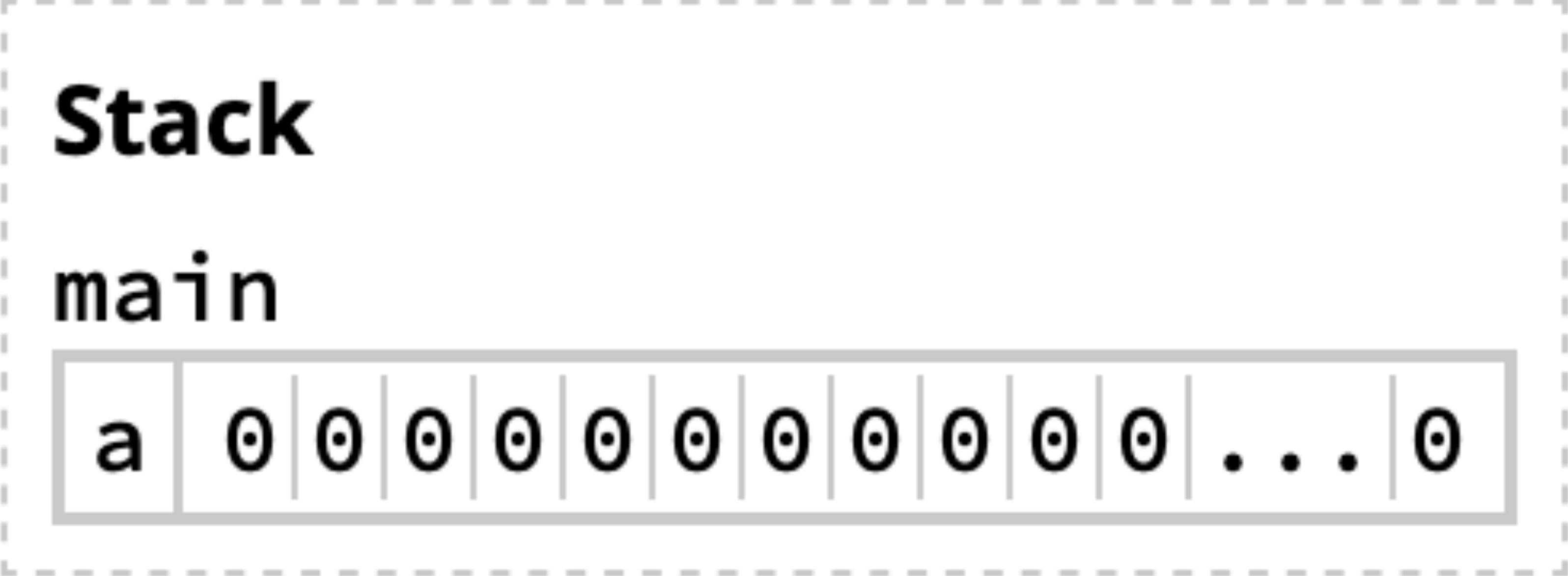
```
fn main() {
    let a_num = 4; L1
    make_and_drop(); L3
}

fn make_and_drop() {
    let a_box = Box::new(5); L2
}
```



Data on the stack vs. heap

```
let a = [0; 1_000_000]; L1
let b = a; L2
```

L1

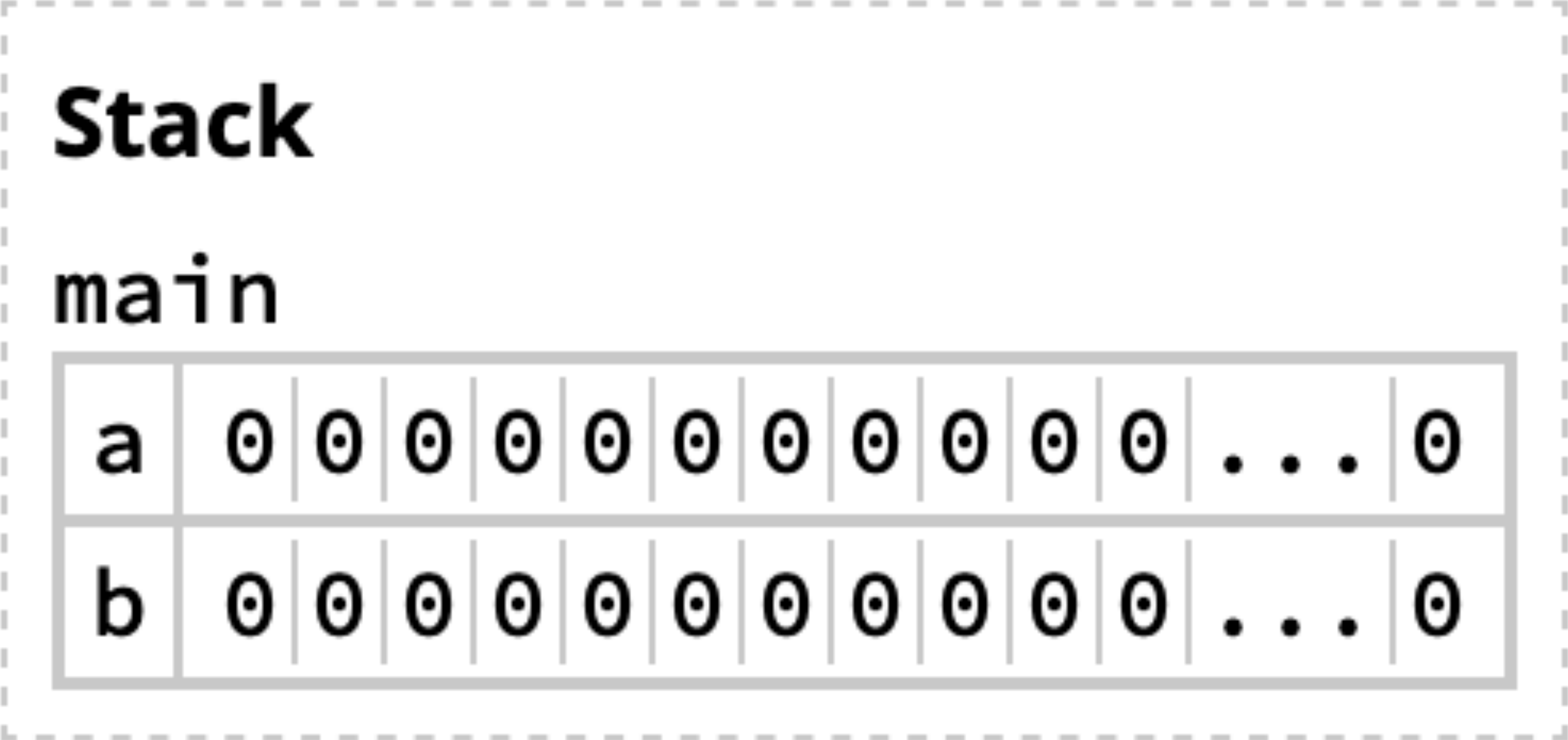


Stack

main

a 0 0 0 0 0 0 0 0 0 0 0 ... 0

L2



Stack

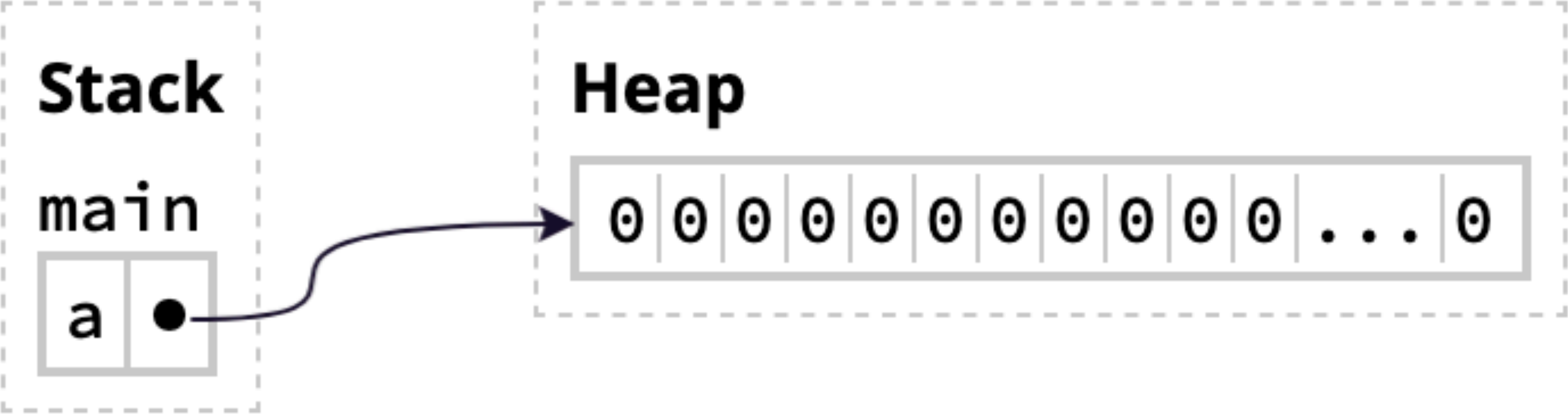
main

a 0 0 0 0 0 0 0 0 0 0 0 ... 0

b 0 0 0 0 0 0 0 0 0 0 0 ... 0

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```

L1



Stack

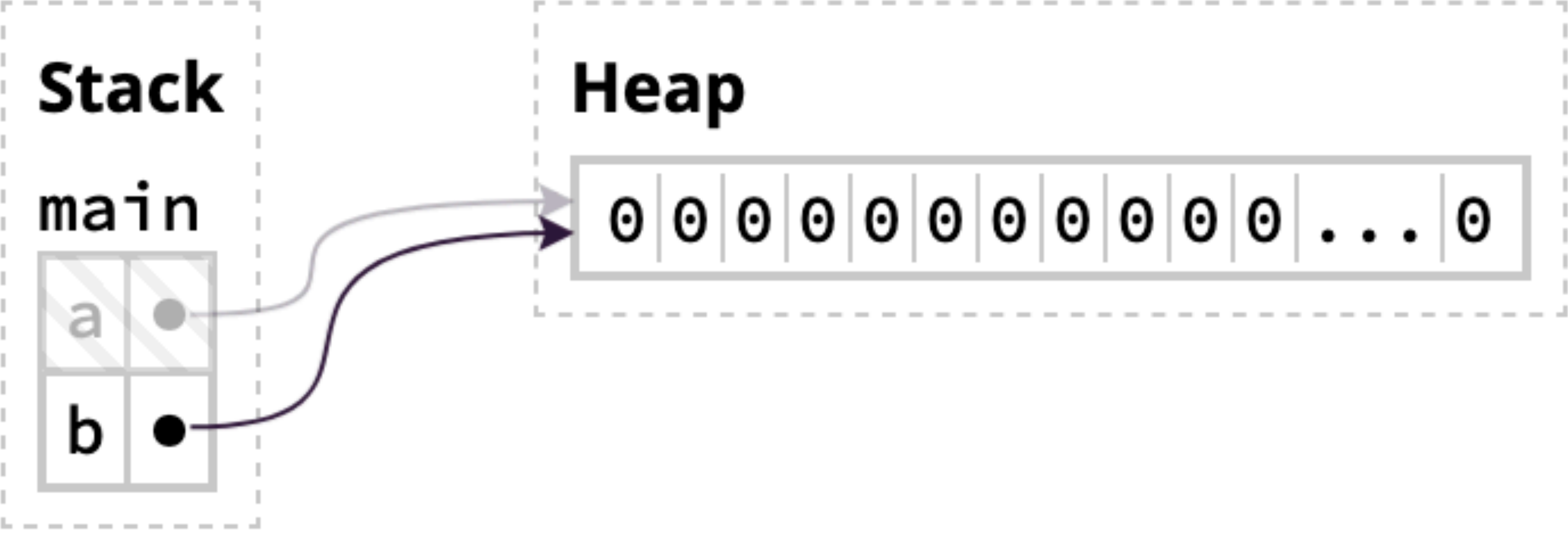
main

a ●

Heap

0 0 0 0 0 0 0 0 0 0 0 ... 0

L2



Stack

main

a ●

b ●

Heap

0 0 0 0 0 0 0 0 0 0 0 ... 0

No manual memory management

Languages like C and C++ let programmers allocate and free heap memory

- malloc(n) allocates n bytes of heap memory and returns a pointer to it
- free(p) frees the memory pointed to by the pointer p
- This is a **massive** source of security vulnerabilities

Rust doesn’t permit manual memory management

- Once you allocate a Box, the data remains valid and accessible until the Box is dropped
- Once the Box is dropped, the data is freed and no longer accessible

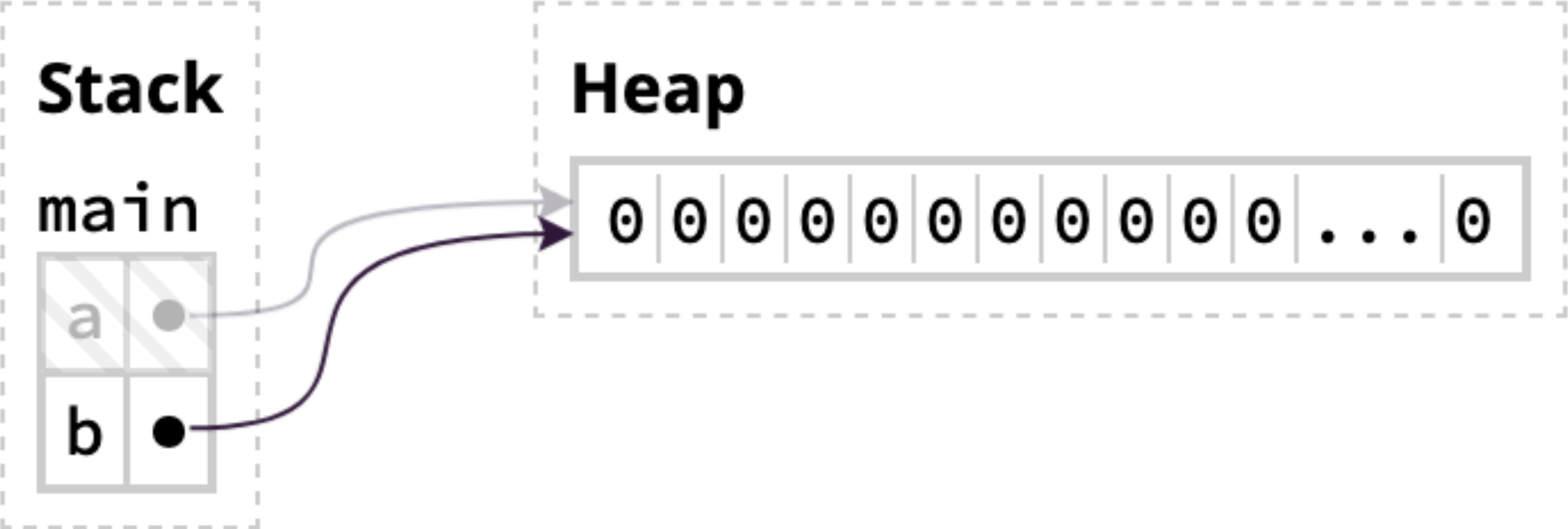
No double frees

We need to reconcile two facts

- When a Box is dropped (e.g., because the function returns), the heap memory is freed
- When we assign a Box to a new variable, the new variable points to the same heap memory

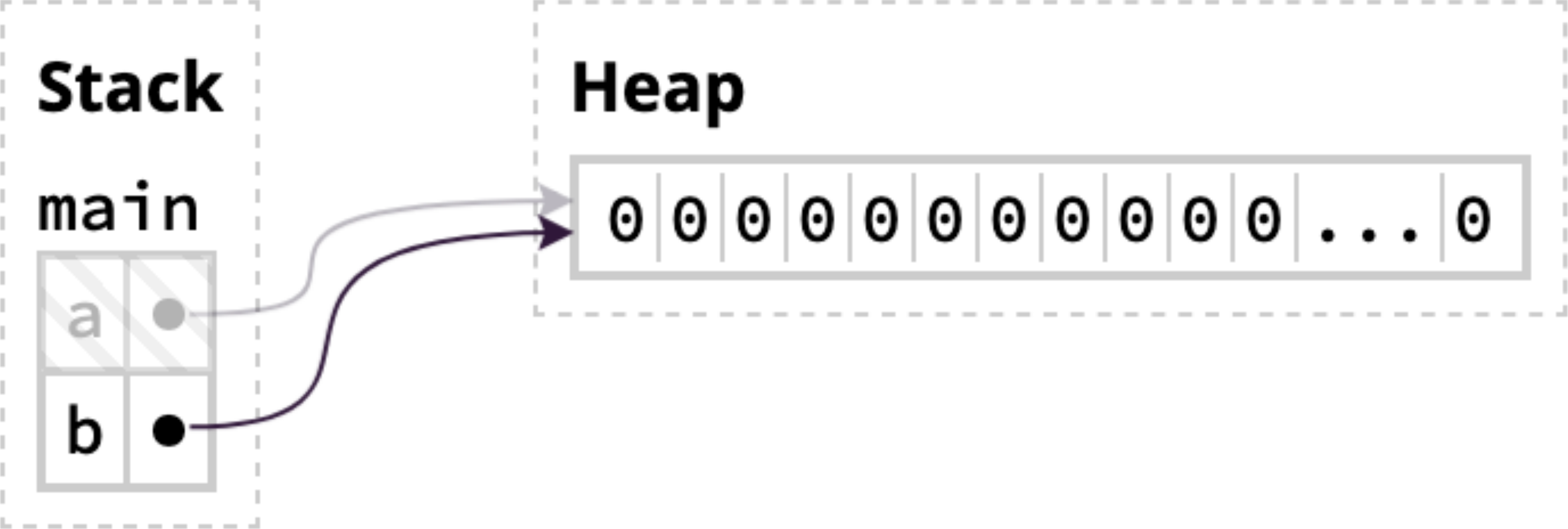
```
let a = Box::new([0; 1_000_000]);
let b = a;
```

When main returns, it seems like both a and b will be dropped and the heap memory will be freed twice!
Undefined behavior!



No double frees due to ownership!

```
let a = Box::new([0; 1_000_000]);
let b = a;
```



Double frees don’t happen because the box was **moved**, not copied

After moving data, it can no longer be accessed by the old name

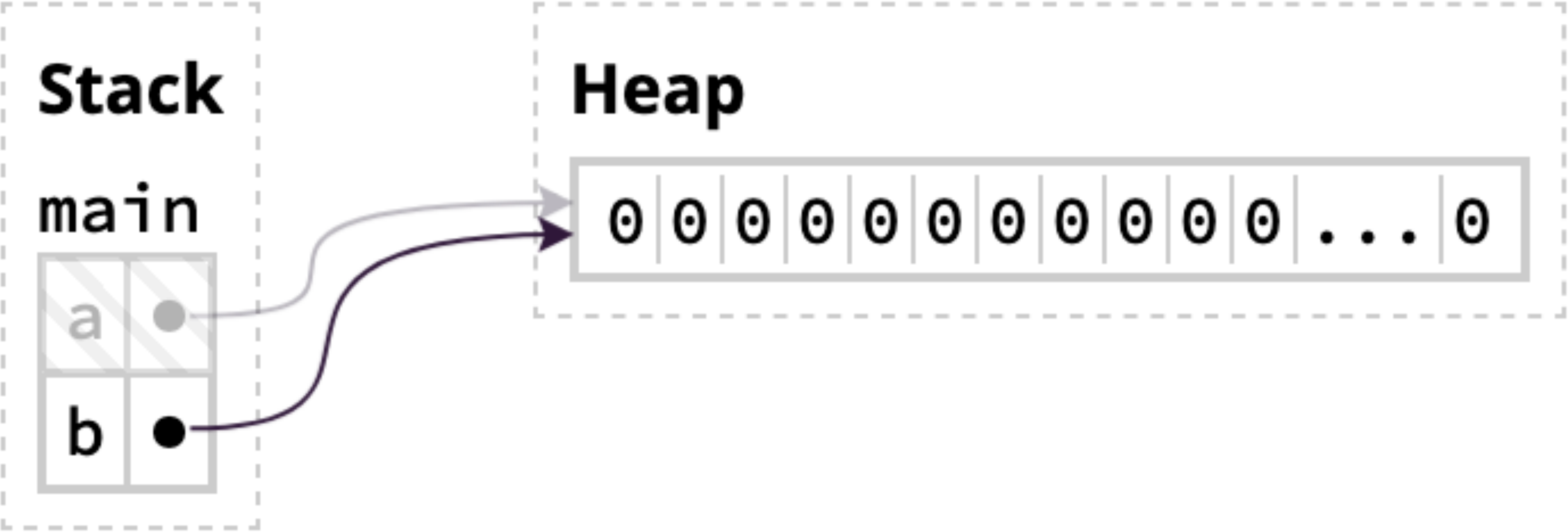
We say that b **owns** the Box

Box deallocation principle

If a variable owns a Box, when Rust deallocates the variable's frame, then Rust deallocates the Box's heap memory

In the example,
`let b = a;`
moved the ownership of the Box from a to b

Therefore the heap memory is only freed once



No use-after-free

A common vulnerability in C and C++ code is

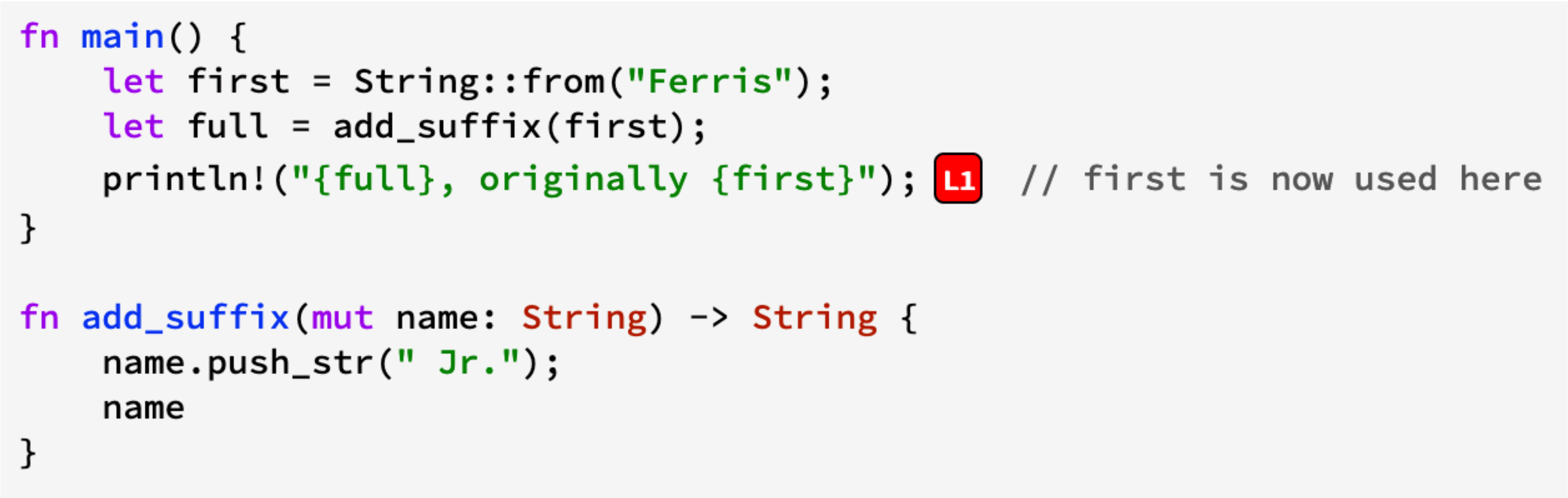
- Allocate some heap memory
- Free the allocated memory
- Use the freed memory; this is **undefined behavior!**

In Rust, that might look something like

```
let b = Box::new(10);
drop(b); // Frees the allocated memory
println!("{b}");
```

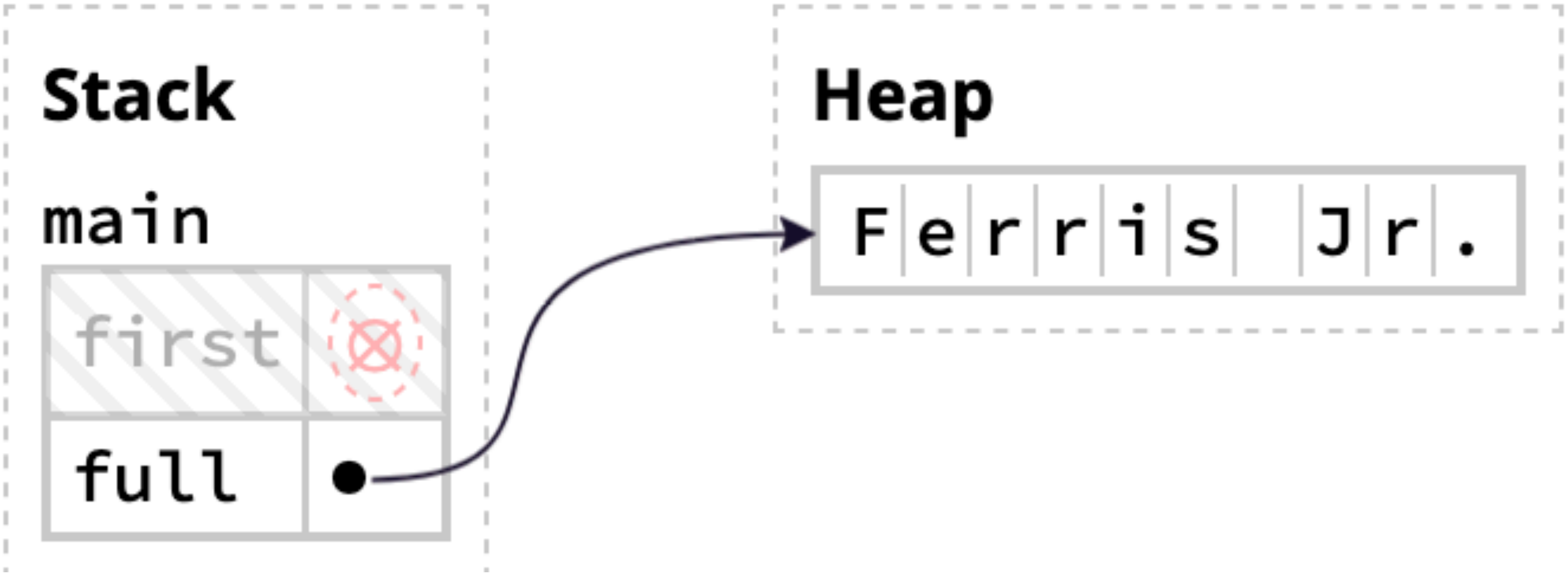
Rust gives a compile time error

Cannot use a variable after moving it



```
fn main() {
    let first = String::from("Ferris");
    let full = add_suffix(first);
    println!("{full}, originally {first}"); L1  // first is now used here
}

fn add_suffix(mut name: String) -> String {
    name.push_str(" Jr.");
    name
}
```

L1 undefined behavior: pointer used after its pointee is freed



Appending the string “ Jr.” causes the string to be reallocated

If we could continue to access first, it would point to freed memory! **Undefined behavior!**

Cloning

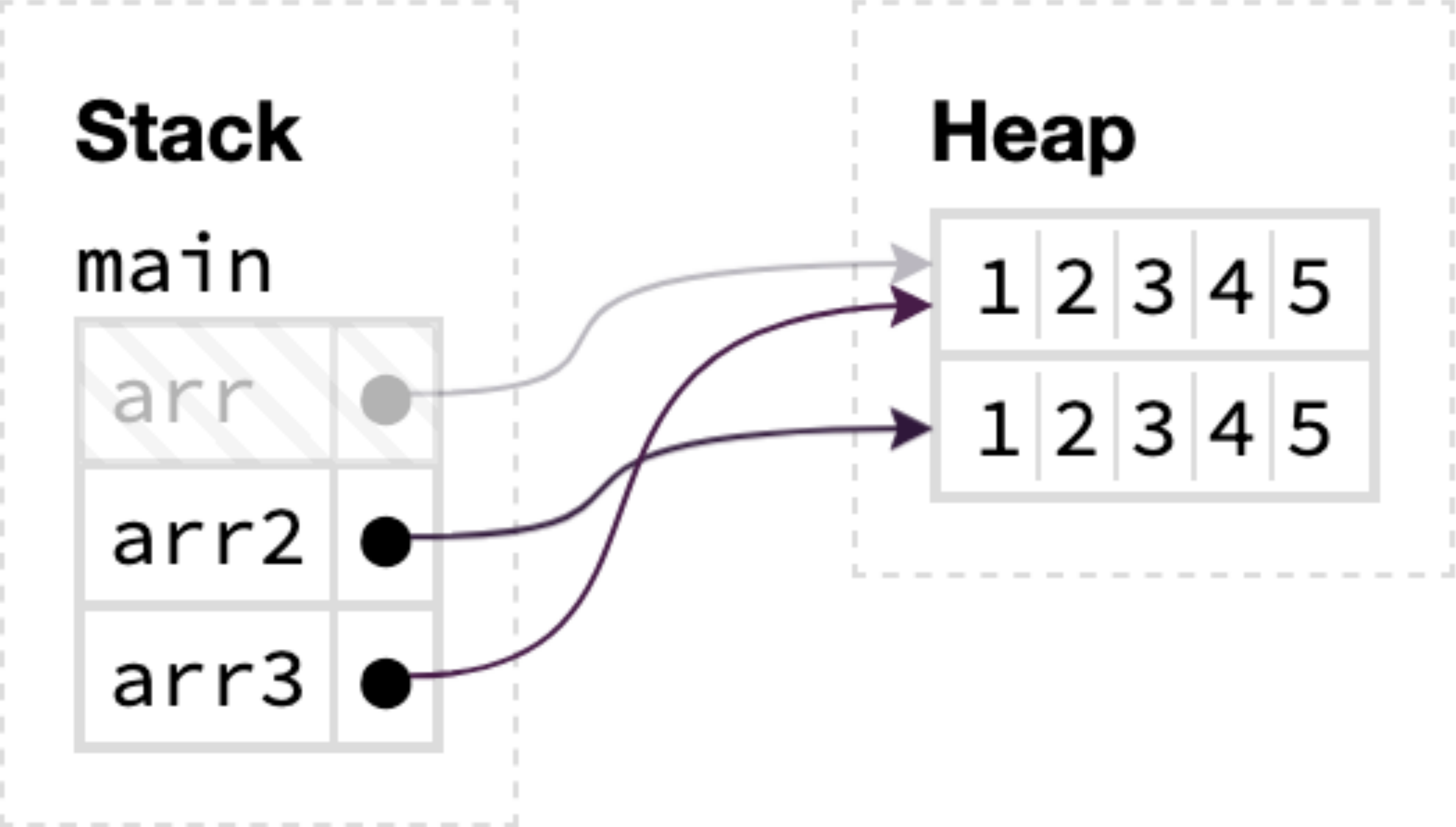
Primitive types like numeric types (i32, u64, usize, etc.) can be copied

Types that involve pointers (Box, String, Vec, etc.) cannot; they are moved

If we want to make a deep copy of a type, we can use the clone() method

Cloning

```
fn main() {
    let arr = vec![1, 2, 3, 4, 5];
    let arr2 = arr.clone();
    let arr3 = arr;
}
```



```
fn foo(s: String) { /* ... */ }

fn main() {
    let clickers = String::from("Clickers!");
    foo(XXX); // <-- Here
    println!("{clickers}");
}
```

What should we replace XXX with to pass the clickers string to `foo()`?

A. clickers

B. &clickers

C. clickers.clone()

D. clone(clickers)

E. More than one of the above

Collections

Collections like String, Vec, and HashMap use a Box internally*

When the String or Vec variable is dropped, the contents is freed

When a collection is passed as an argument to a function or returned from a function, only the pointer needs to be copied, not the contents

```
fn make_evens(num_evens: u32) -> Vec<u32> {
    let mut result = Vec::new();
    for num in 0..num_evens {
        result.push(num * 2);
    }
    result
}
```

* It’s not actually a Box, but it behaves similarly