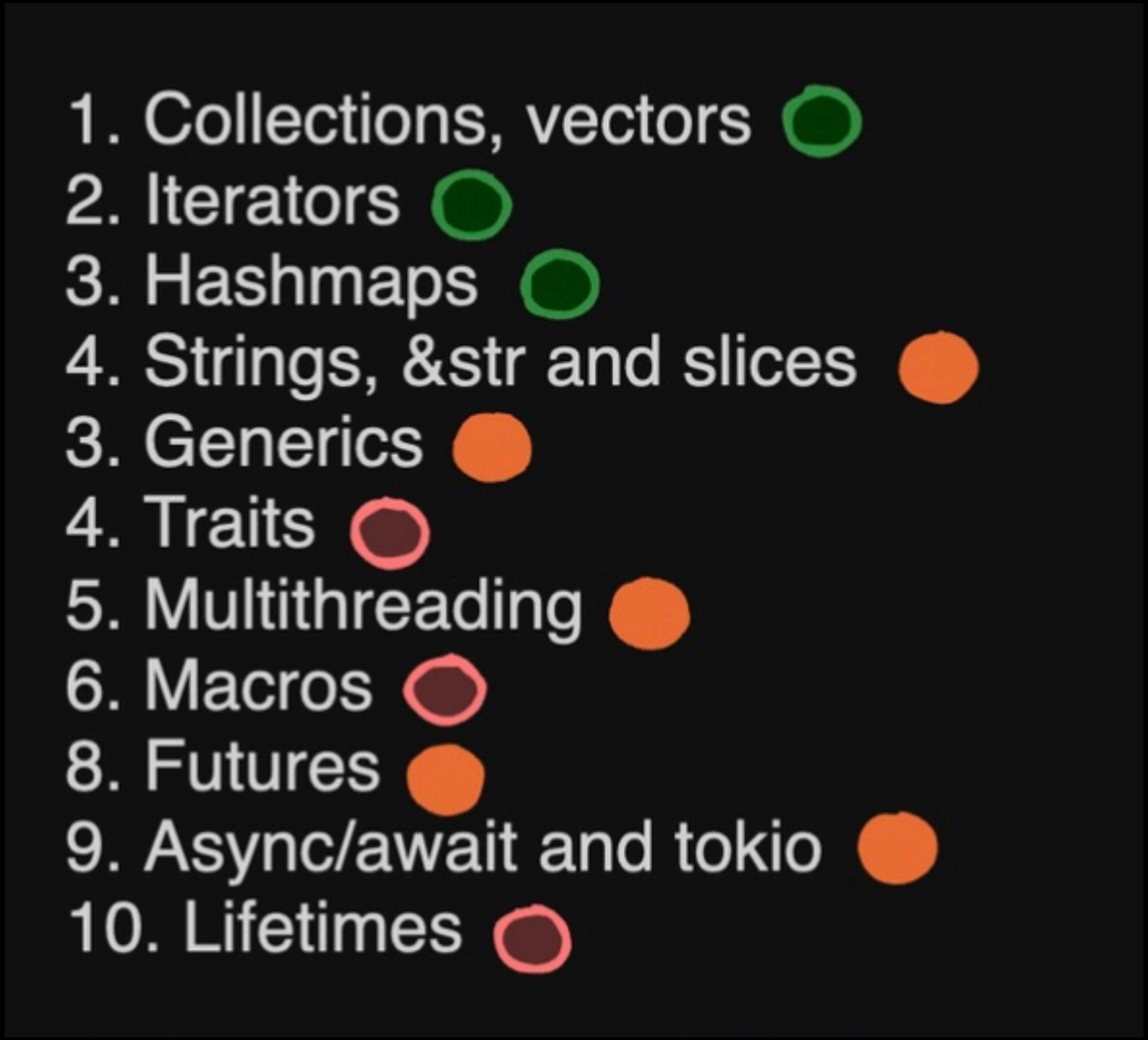
Advance Rust APIs



1. Collections, vectors 🟢
2. Iterators 🟢
3. Hashmaps 🟢
4. Strings, &str and slices 🟠
3. Generics 🟠
4. Traits 🔴
5. Multithreading 🟠
6. Macros 🔴
8. Futures 🟠
9. Async/await and tokio 🟠
10. Lifetimes 🔴

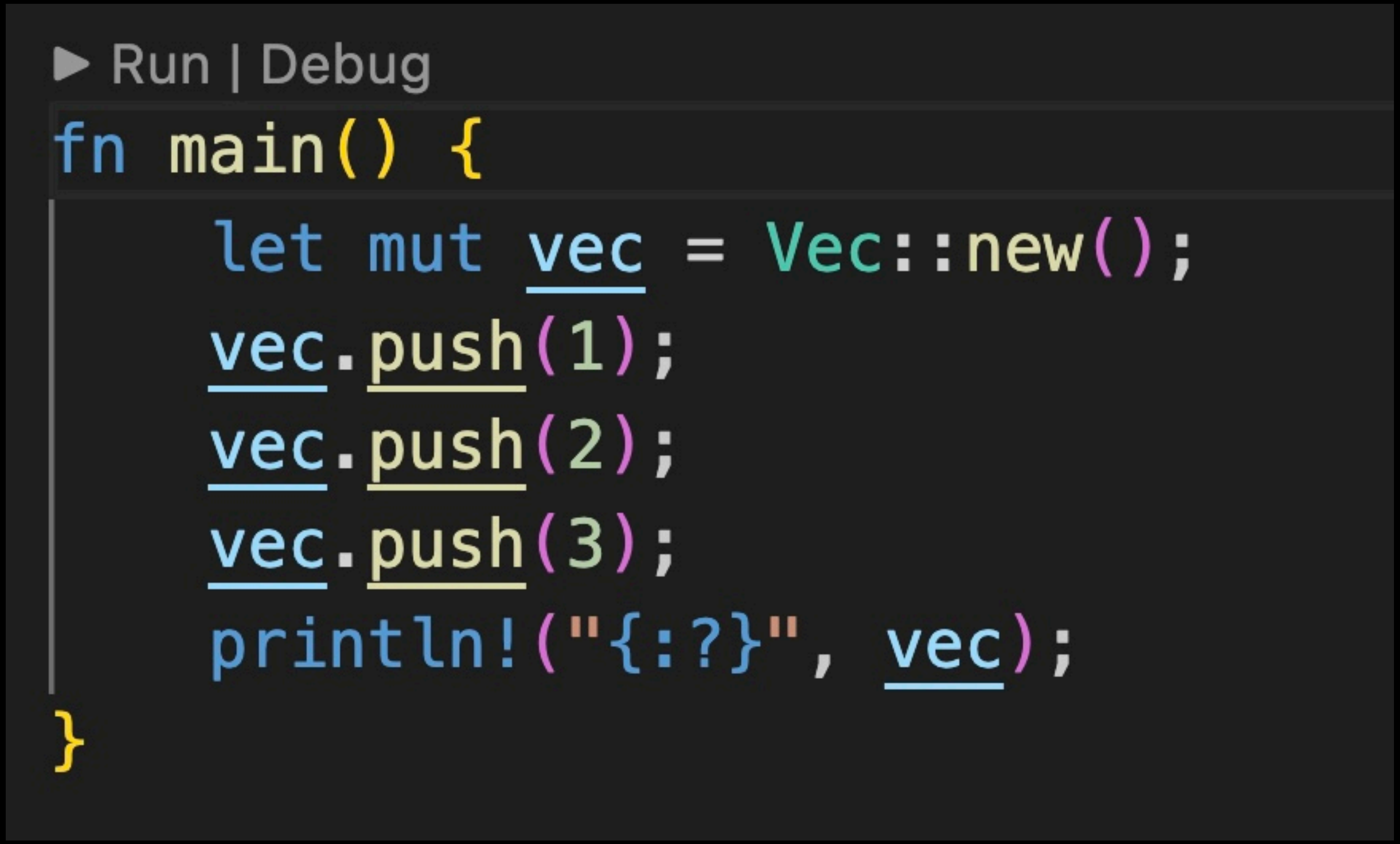
Collections

Rust’s standard library includes a number of very useful data structures called collections.
Most other data types represent one specific value, but collections can contain multiple values.
the data these collections point to is stored on the heap

Vectors

Vectors

Vectors allow you to store more than one value in a single data structure that puts all the values next to each other in memory.



```rust
fn main() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    vec.push(3);
    println!("{:?}", vec);
}
```

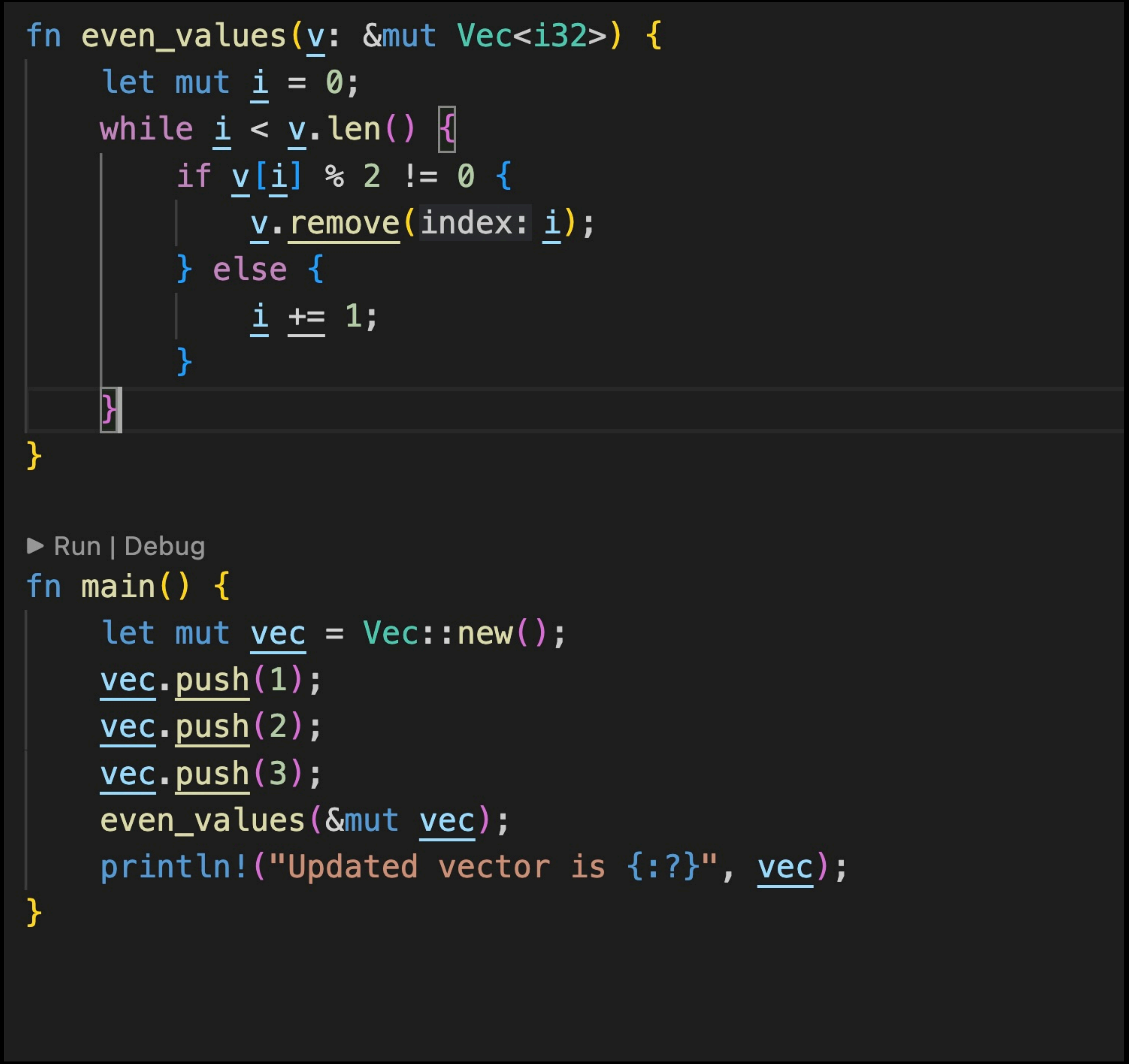
Vectors

Q: Write a function that takes a vector as an input and returns a vector with even values

Vectors

Q: Write a function that takes a vector as an input and returns a vector with even values

Approach #1

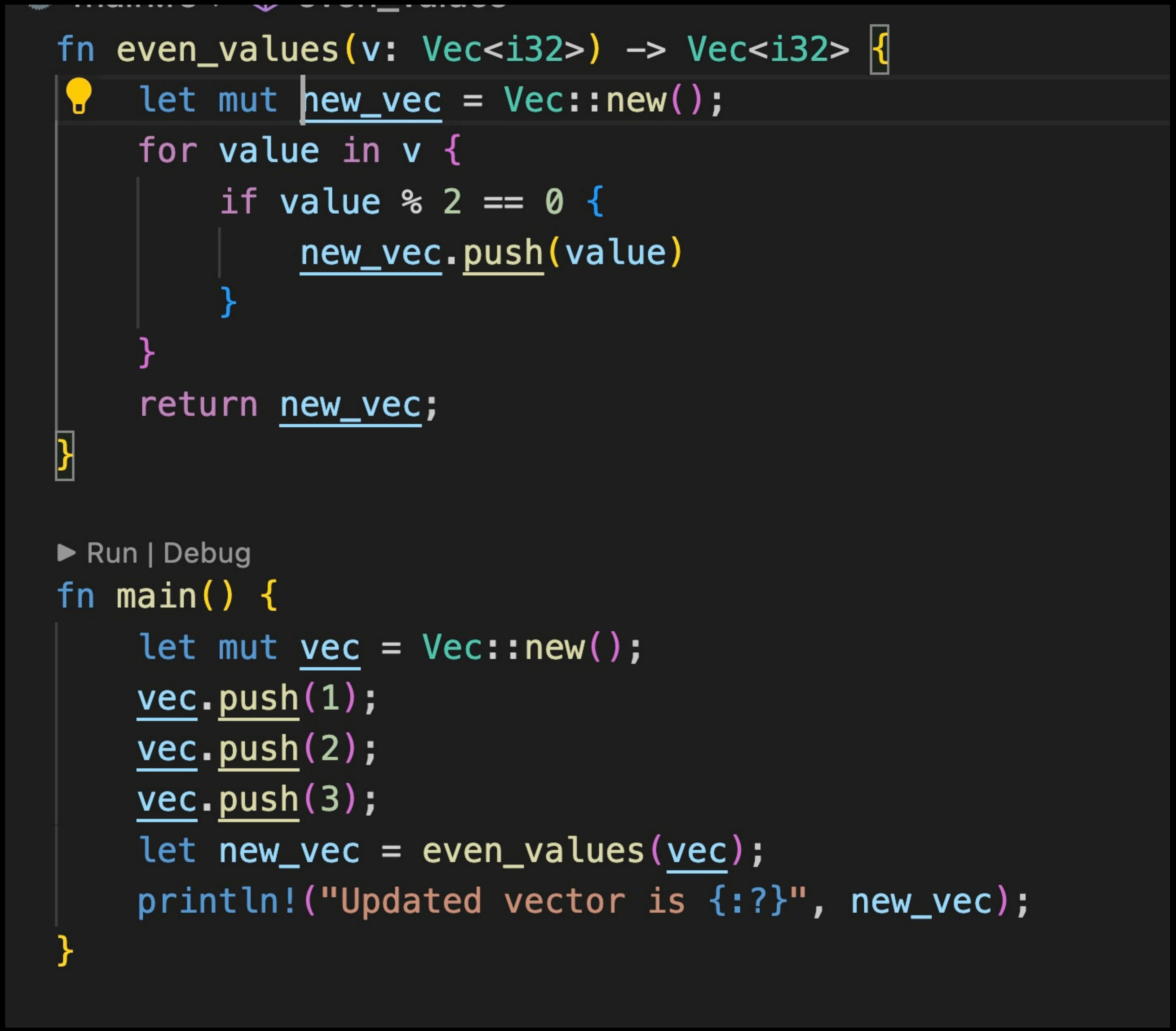


```
fn even_values(v: &mut Vec<i32>) {
    let mut i = 0;
    while i < v.len() {
        if v[i] % 2 != 0 {
            v.remove(index: i);
        } else {
            i += 1;
        }
    }
}

▶ Run | Debug
fn main() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    vec.push(3);
    even_values(&mut vec);
    println!("Updated vector is {:?}", vec);
}
```

Vectors

Q: Write a function that takes a vector as an input and returns a vector with even values

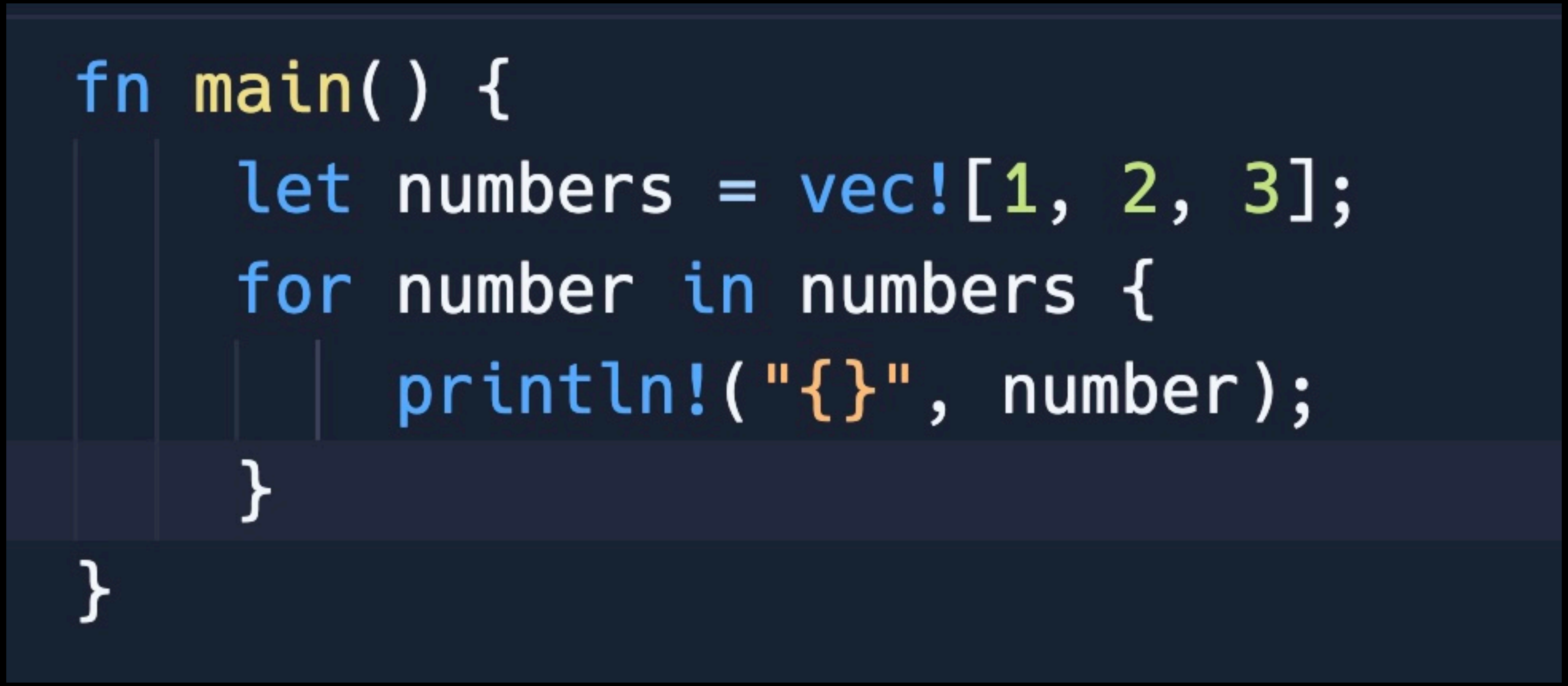
Approach #2



```rust
fn even_values(v: Vec<i32>) -> Vec<i32> {
    let mut new_vec = Vec::new();
    for value in v {
        if value % 2 == 0 {
            new_vec.push(value)
        }
    }
    return new_vec;
}

fn main() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    vec.push(3);
    let new_vec = even_values(vec);
    println!("Updated vector is {:?}", new_vec);
}
```

Vectors

Initialising using rust macros



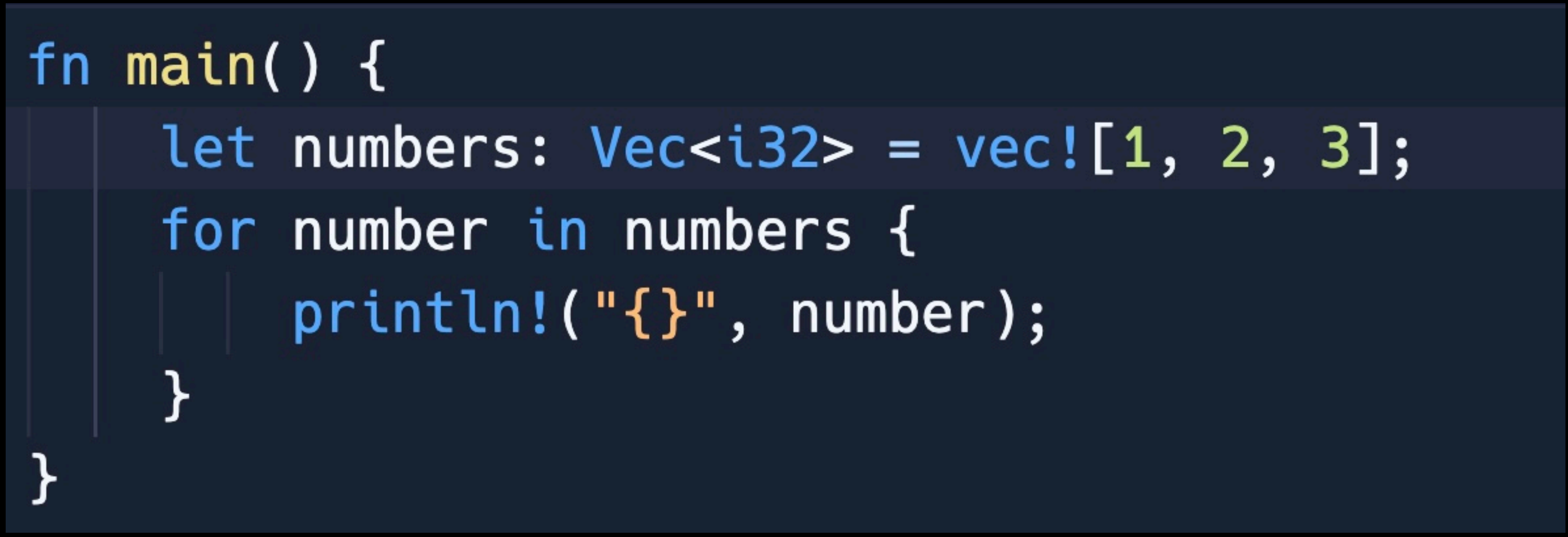
```rust
fn main() {
    let numbers = vec![1, 2, 3];
    for number in numbers {
        println!("{}", number);
    }
}
```

Vectors

Defining the type of the vector as a generic

Vectors

Explicitly giving type using generics



```rust
fn main() {
    let numbers: Vec<i32> = vec![1, 2, 3];
    for number in numbers {
        println!("{}", number);
    }
}
```

Hashmaps

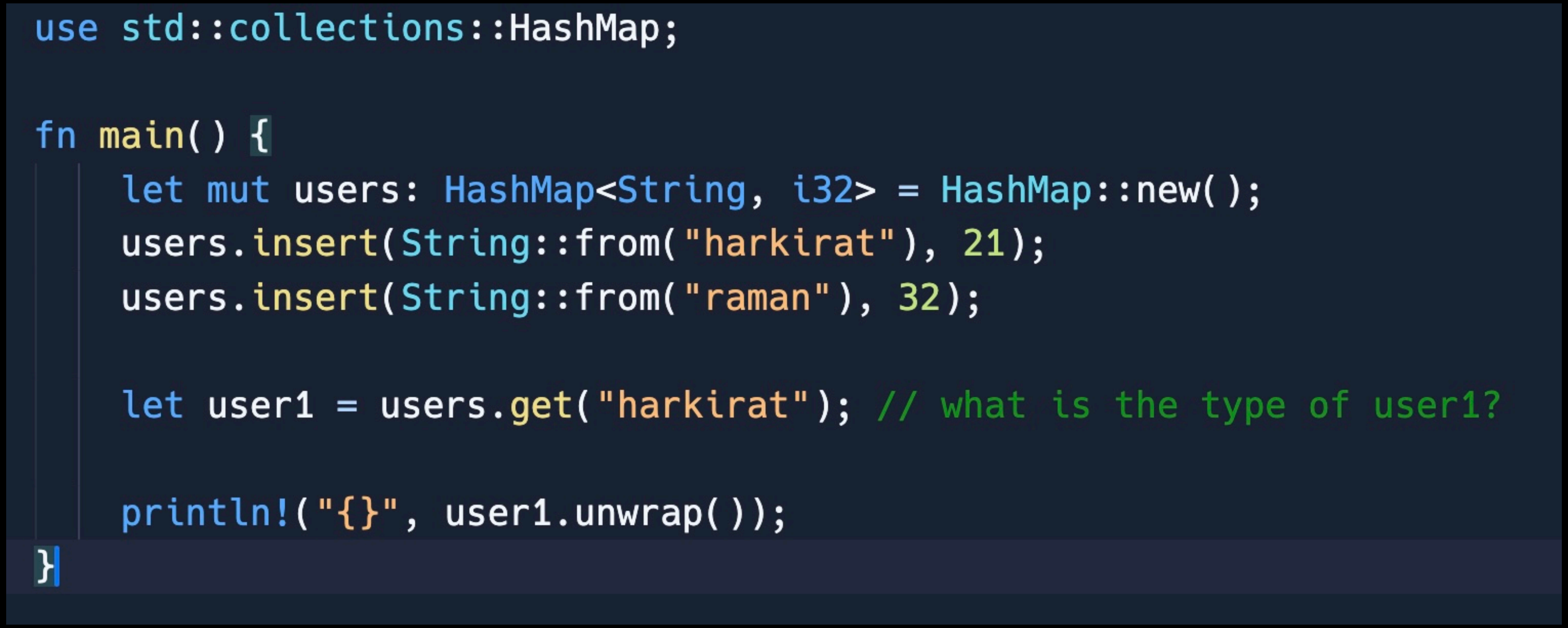
Hashmaps

Hashmaps stores a key value pair in rust.
Similar to objects in JS
Dict in Python
HashMaps in Java

Hashmaps

Methods -

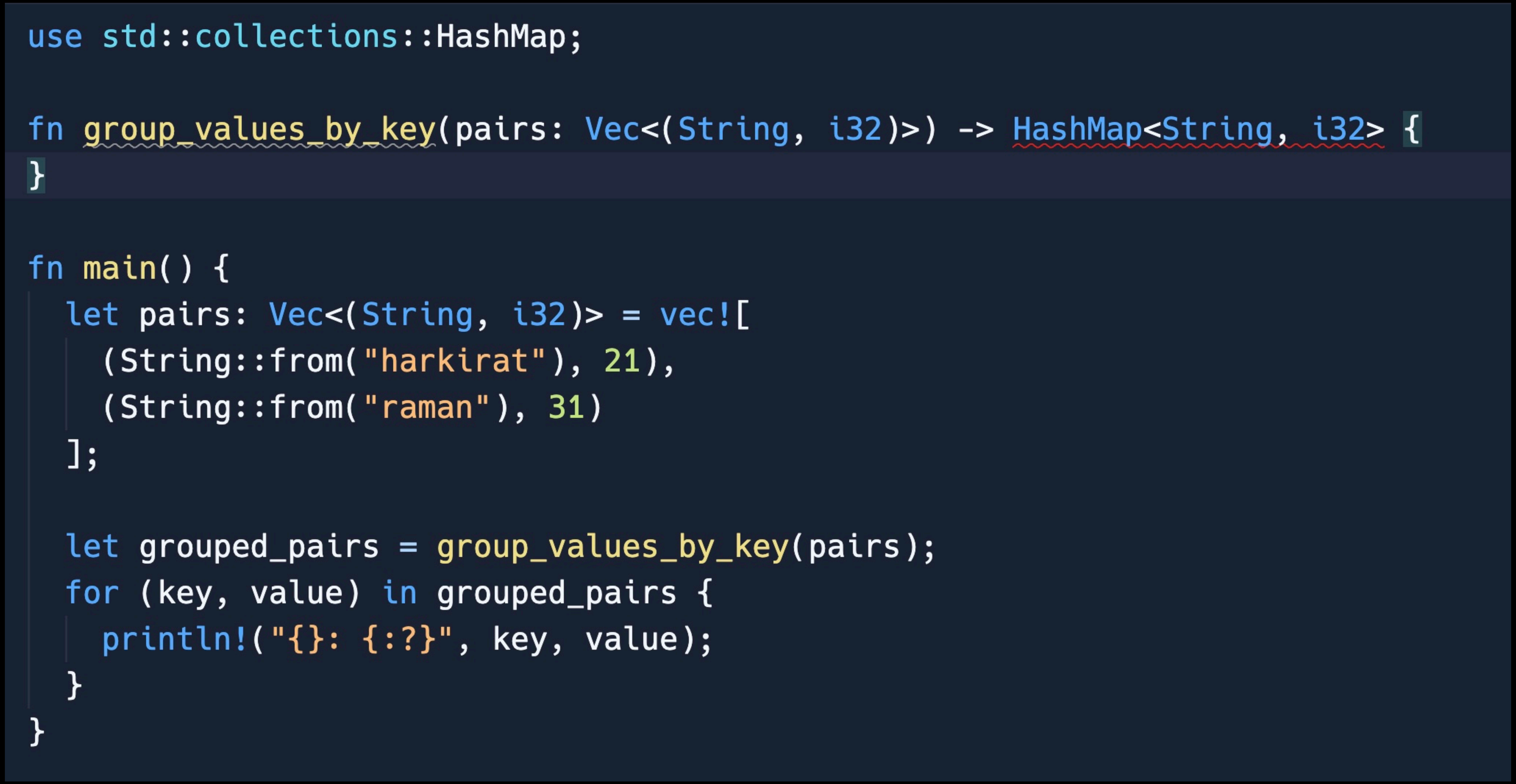
1. insert
2. get
3. remove
4. clear



```rust
use std::collections::HashMap;

fn main() {
    let mut users: HashMap<String, i32> = HashMap::new();
    users.insert(String::from("harkirat"), 21);
    users.insert(String::from("raman"), 32);

    let user1 = users.get("harkirat"); // what is the type of user1?

    println!("{}", user1.unwrap());
}
```

Hashmap

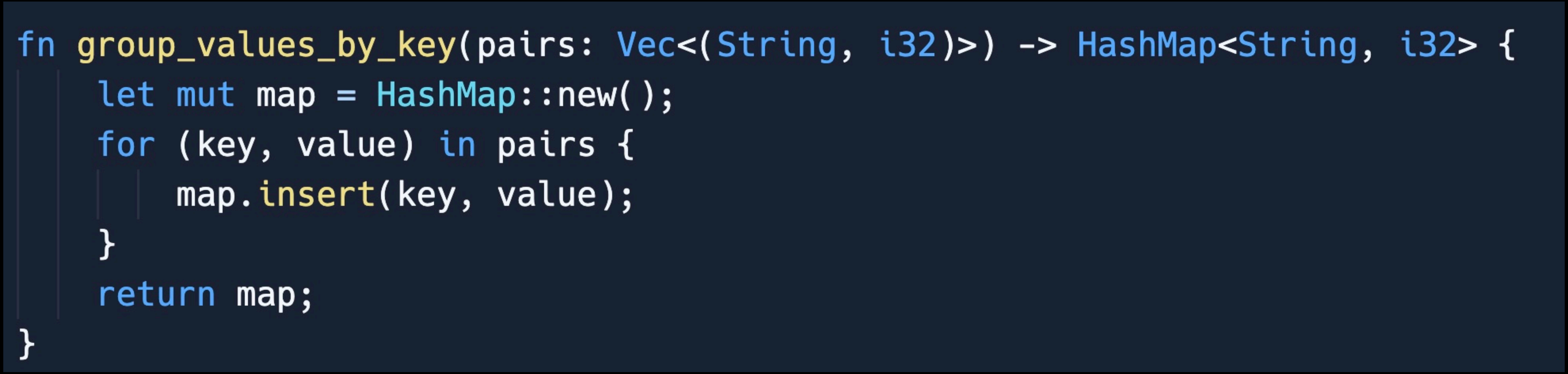
Q: Write a function that takes a vector of tuples (each tuple containing a key and a value) and returns a Hashmap where the keys are the unique keys from the input tuples and the values are vectors of all corresponding values associated with each key.



```rust
use std::collections::HashMap;

fn group_values_by_key(pairs: Vec<(String, i32)>) -> HashMap<String, i32> {
}

fn main() {
  let pairs: Vec<(String, i32)> = vec![
    (String::from("harkirat"), 21),
    (String::from("raman"), 31)
  ];

  let grouped_pairs = group_values_by_key(pairs);
  for (key, value) in grouped_pairs {
    println!("{}: {:?}", key, value);
  }
}
```

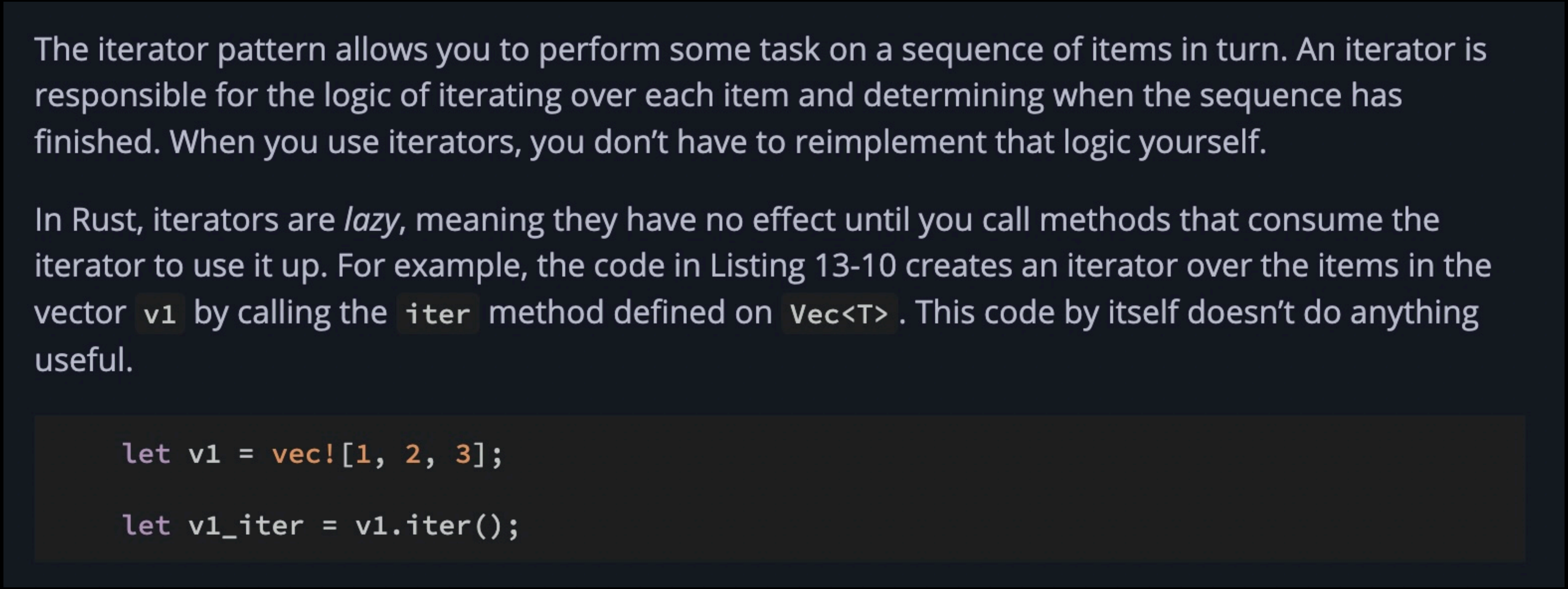
Hashmap

Q: Write a function that takes a vector of tuples (each tuple containing a key and a value) and returns a Hashmap where the keys are the unique keys from the input tuples and the values are vectors of all corresponding values associated with each key.



```
fn group_values_by_key(pairs: Vec<(String, i32)>) -> HashMap<String, i32> {
    let mut map = HashMap::new();
    for (key, value) in pairs {
        map.insert(key, value);
    }
    return map;
}
```

Iterators



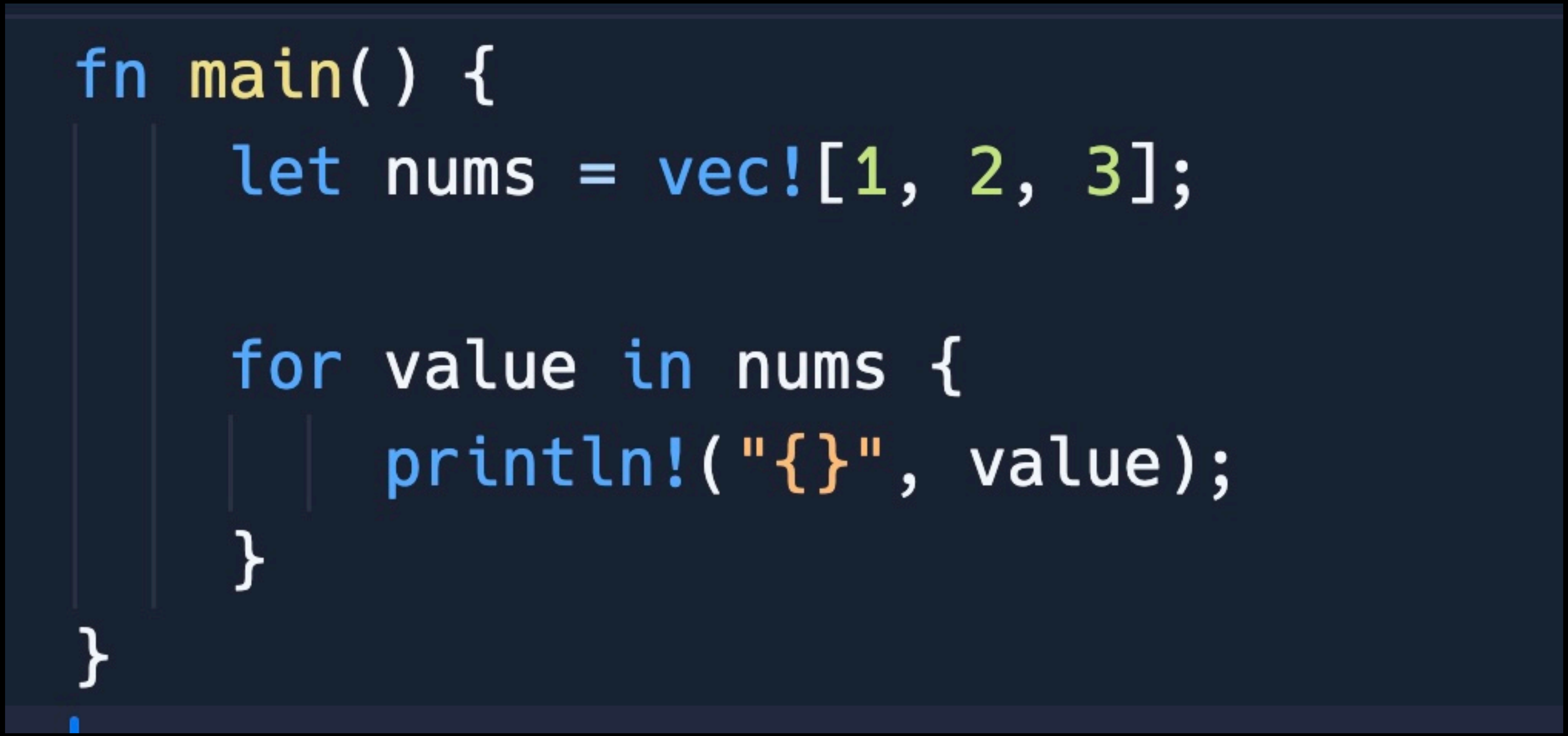
The iterator pattern allows you to perform some task on a sequence of items in turn. An iterator is responsible for the logic of iterating over each item and determining when the sequence has finished. When you use iterators, you don't have to reimplement that logic yourself.

In Rust, iterators are *lazy*, meaning they have no effect until you call methods that consume the iterator to use it up. For example, the code in Listing 13-10 creates an iterator over the items in the vector `v1` by calling the `iter` method defined on `Vec<T>`. This code by itself doesn't do anything useful.

```rust
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();
```

Iterators

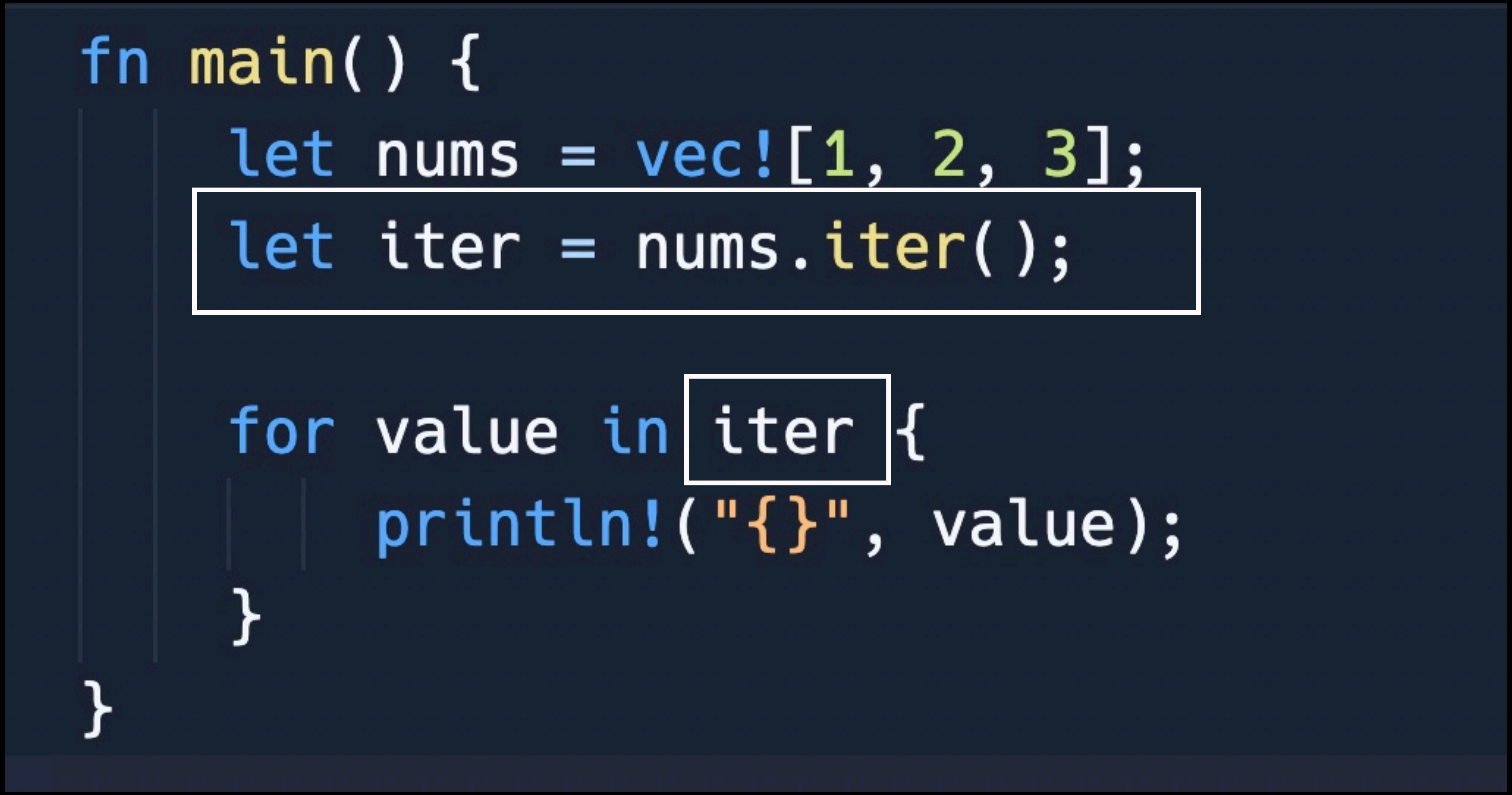
1. Iterating using for loops



```rust
fn main() {
    let nums = vec![1, 2, 3];

    for value in nums {
        println!("{}", value);
    }
}
```

Iterators

2. Iterating after creating an `iterator`



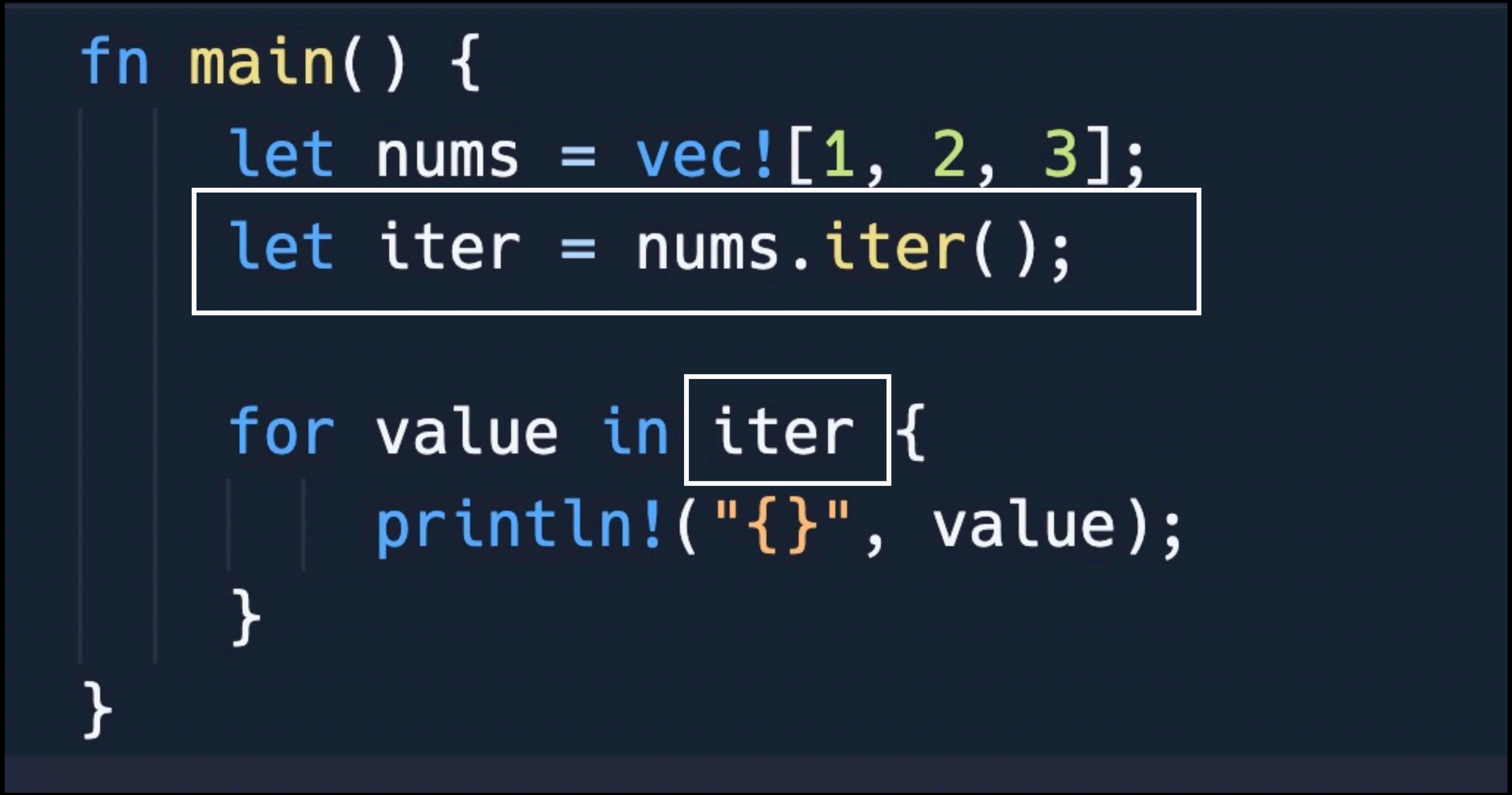
```
fn main() {
    let nums = vec![1, 2, 3];
    let iter = nums.iter();

    for value in iter {
        println!("{}", value);
    }
}
```

Iterators

2. Iterating after creating an `iterator`

The iter() method in Rust provides a way to iterate over the elements of a collection by borrowing them.

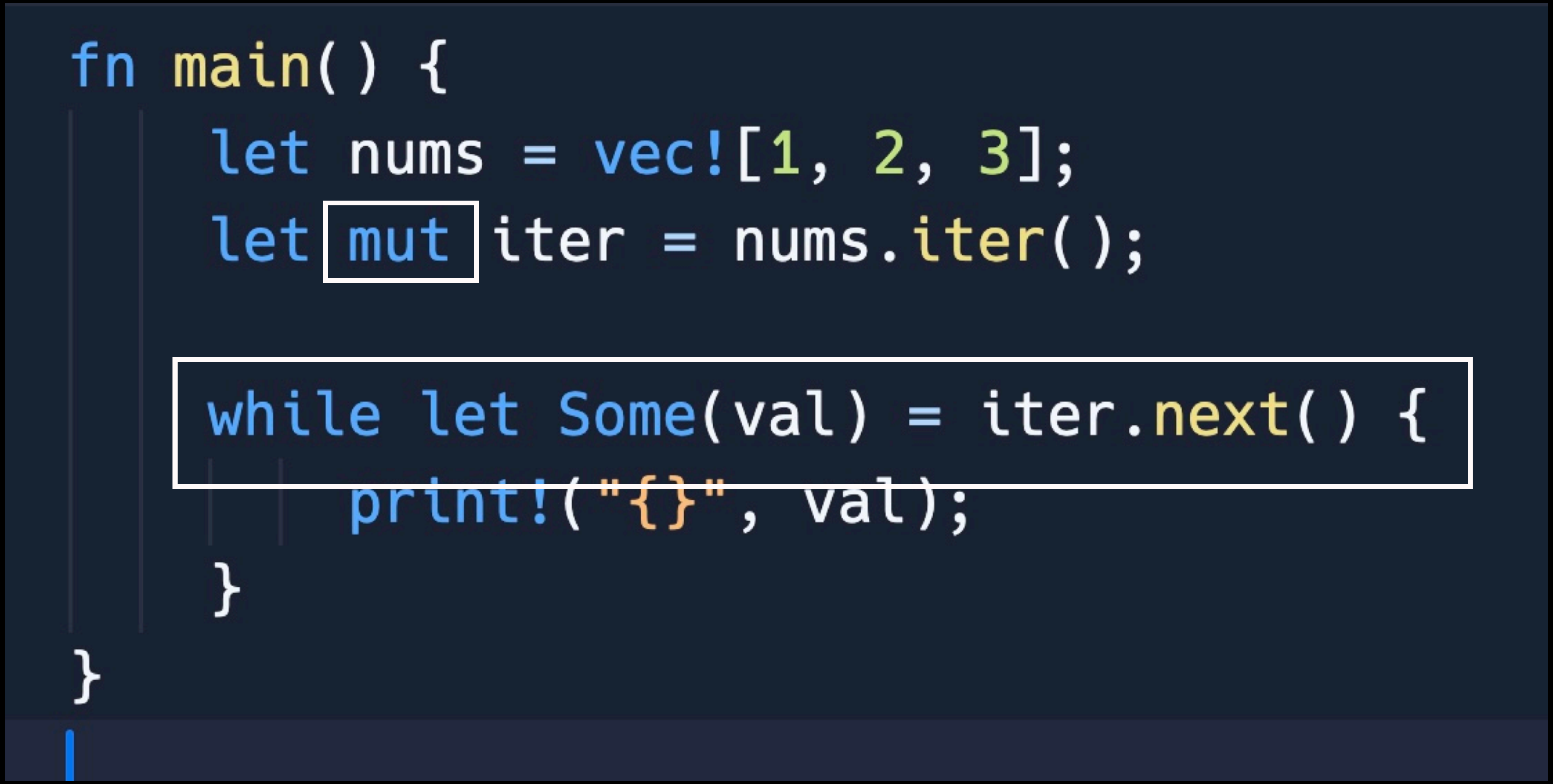
You can’t mutate the variables since we have an immutable reference to the internal elements



```rust
fn main() {
    let nums = vec![1, 2, 3];
    let iter = nums.iter();

    for value in iter {
        println!("{}", value);
    }
}
```

Iterators

3. Iterating using `.next`



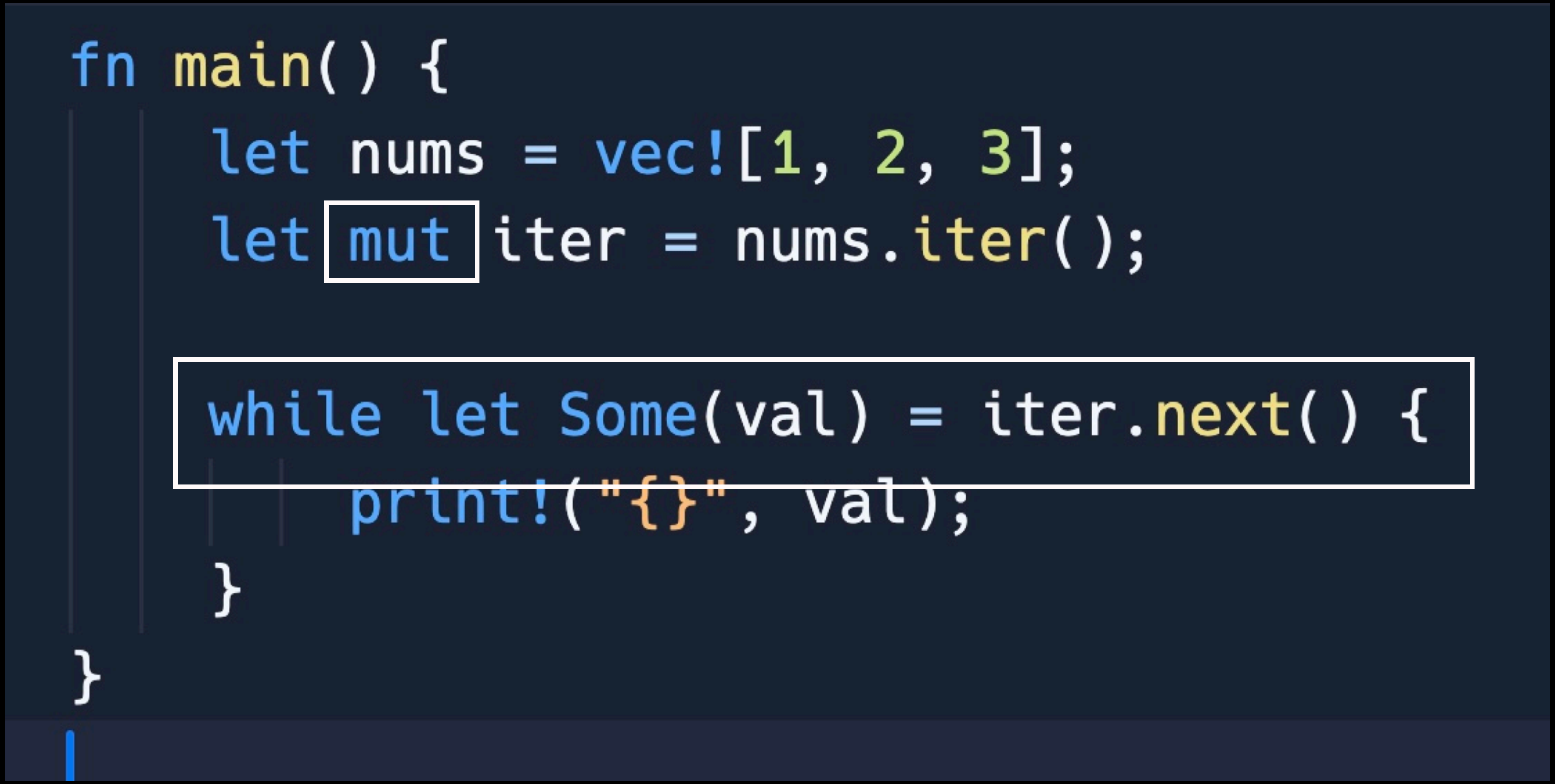
```rust
fn main() {
    let nums = vec![1, 2, 3];
    let mut iter = nums.iter();

    while let Some(val) = iter.next() {
        print!("{}", val);
    }
}
```

Iterators

3. Iterating using `.next`

You can't mutate the data here either

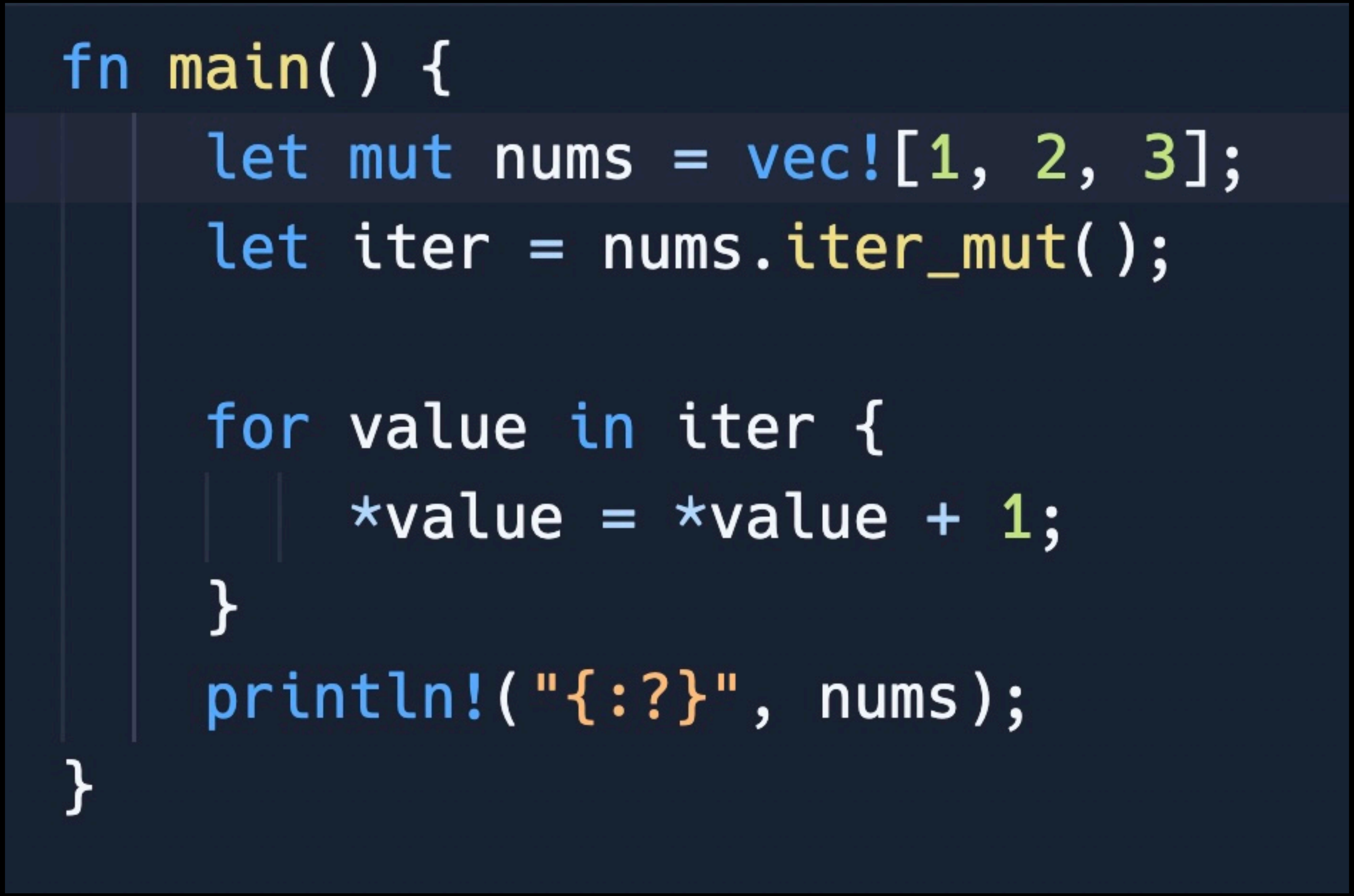
The `iterator` is mutable, but the inner elements (val) still is an immutable reference



```
fn main() {
    let nums = vec![1, 2, 3];
    let mut iter = nums.iter();

    while let Some(val) = iter.next() {
        print!("{}", val);
    }
}
```

Iterators

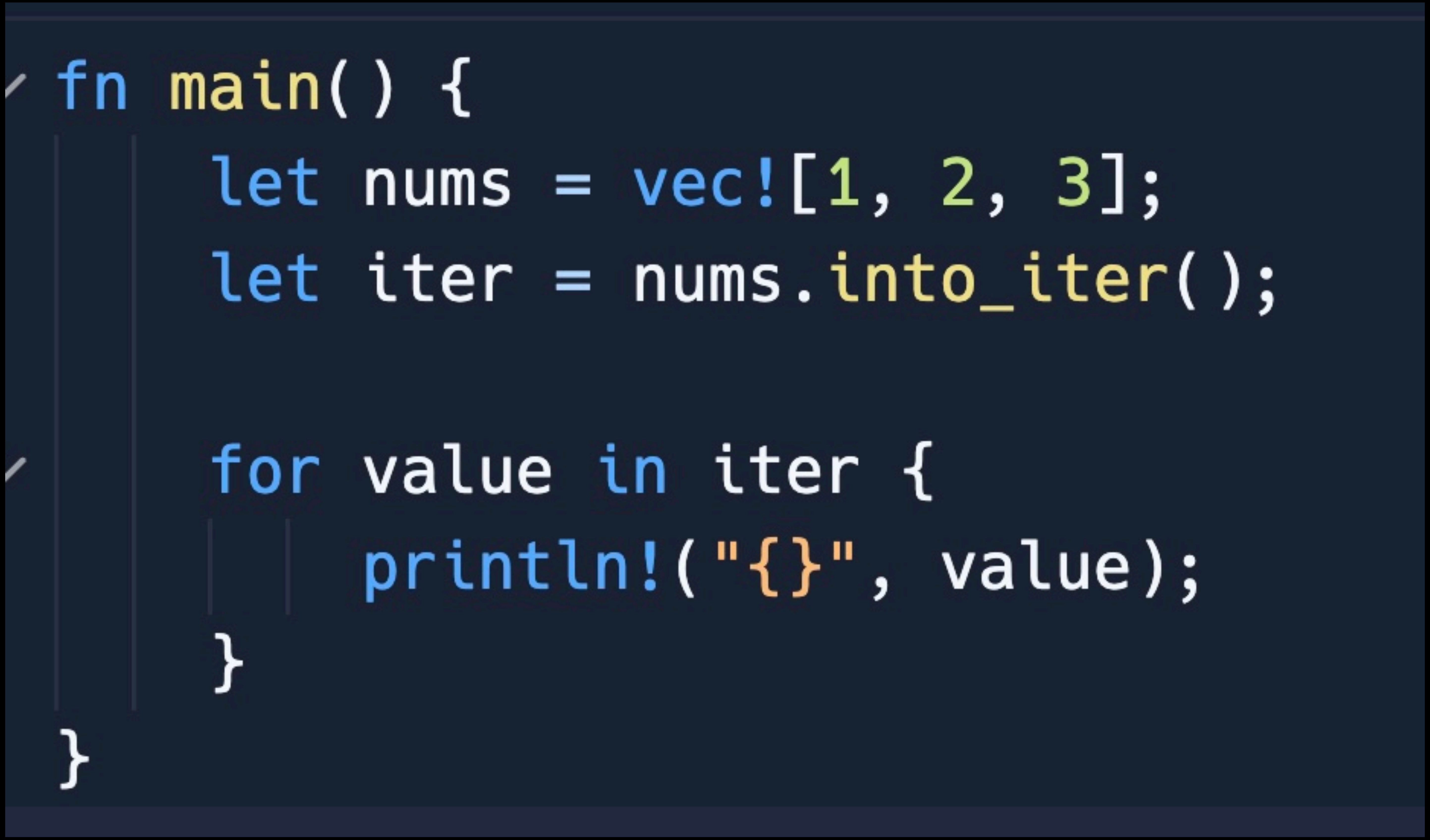
4. IterMut



```
fn main() {
    let mut nums = vec![1, 2, 3];
    let iter = nums.iter_mut();

    for value in iter {
        *value = *value + 1;
    }
    println!("{:?}", nums);
}
```

Iterators

5. IntoIter



```
fn main() {
    let nums = vec![1, 2, 3];
    let iter = nums.into_iter();

    for value in iter {
        println!("{}", value);
    }
}
```

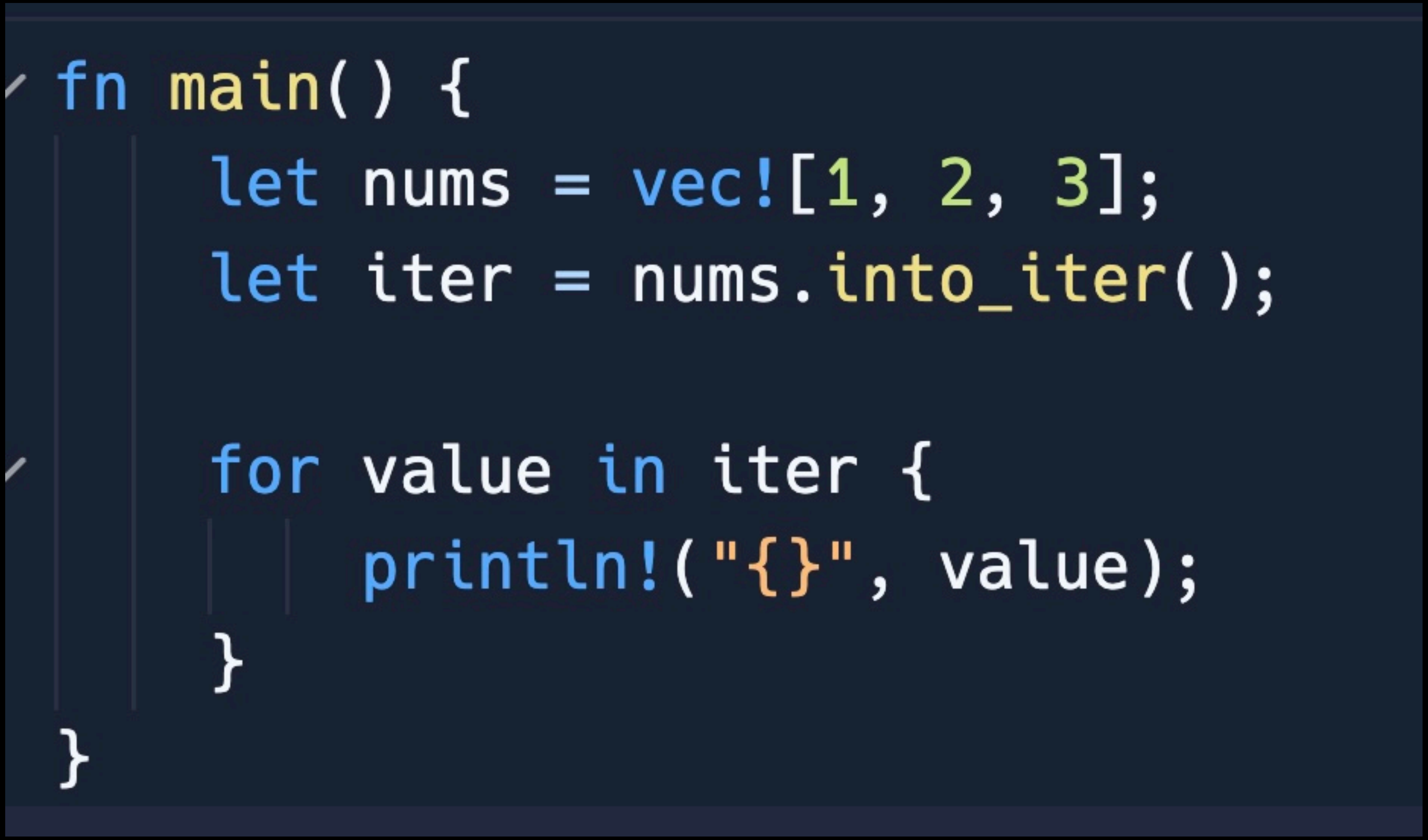
Iterators

5. IntoIter

The IntoIterator trait is used to convert a collection into an iterator that takes ownership of the collection.

Useful when

1. You no longer need the original collection
2. When you need to squeeze performance benefits by transferring ownership (avoiding references)

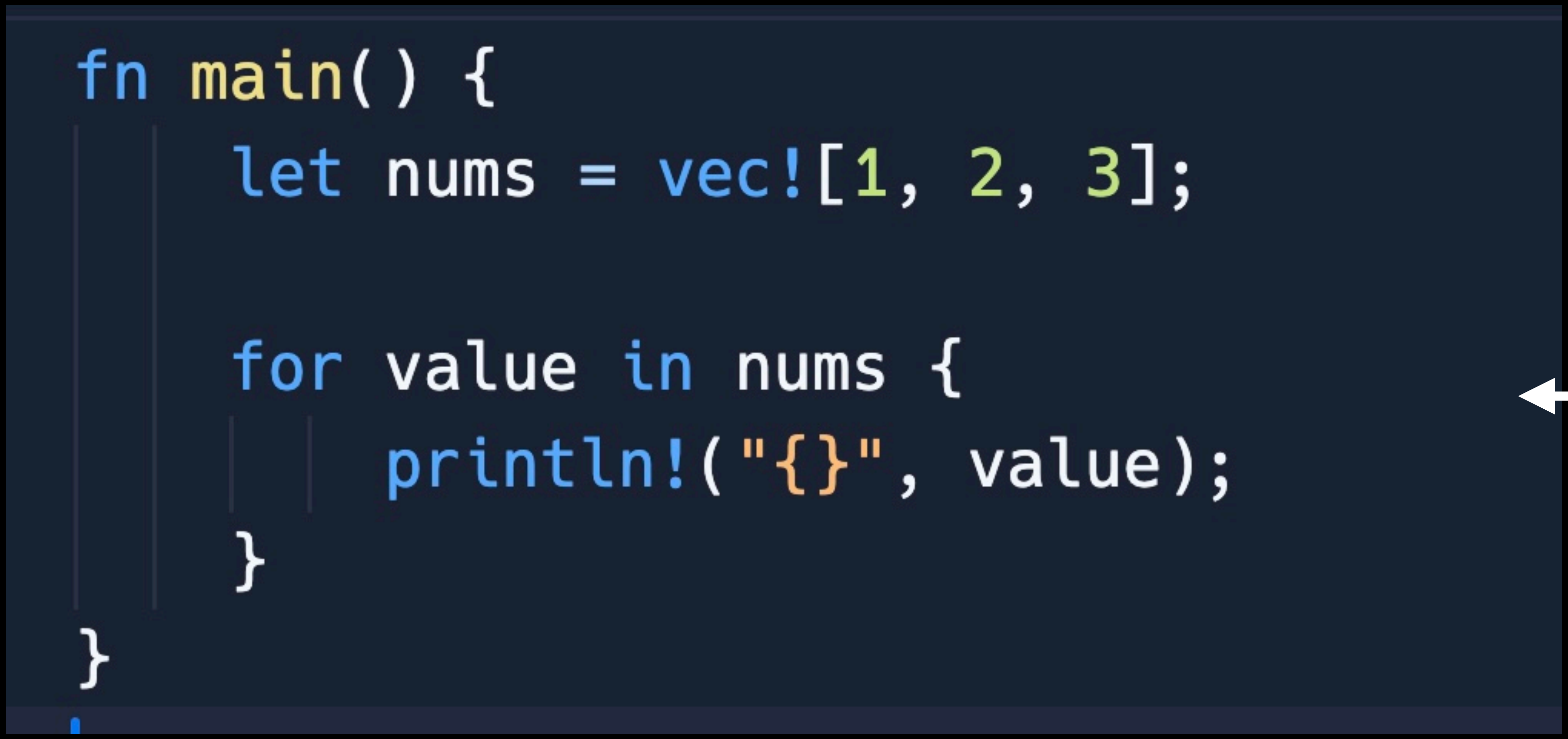


```rust
fn main() {
    let nums = vec![1, 2, 3];
    let iter = nums.into_iter();

    for value in iter {
        println!("{}", value);
    }
}
```

Iterators

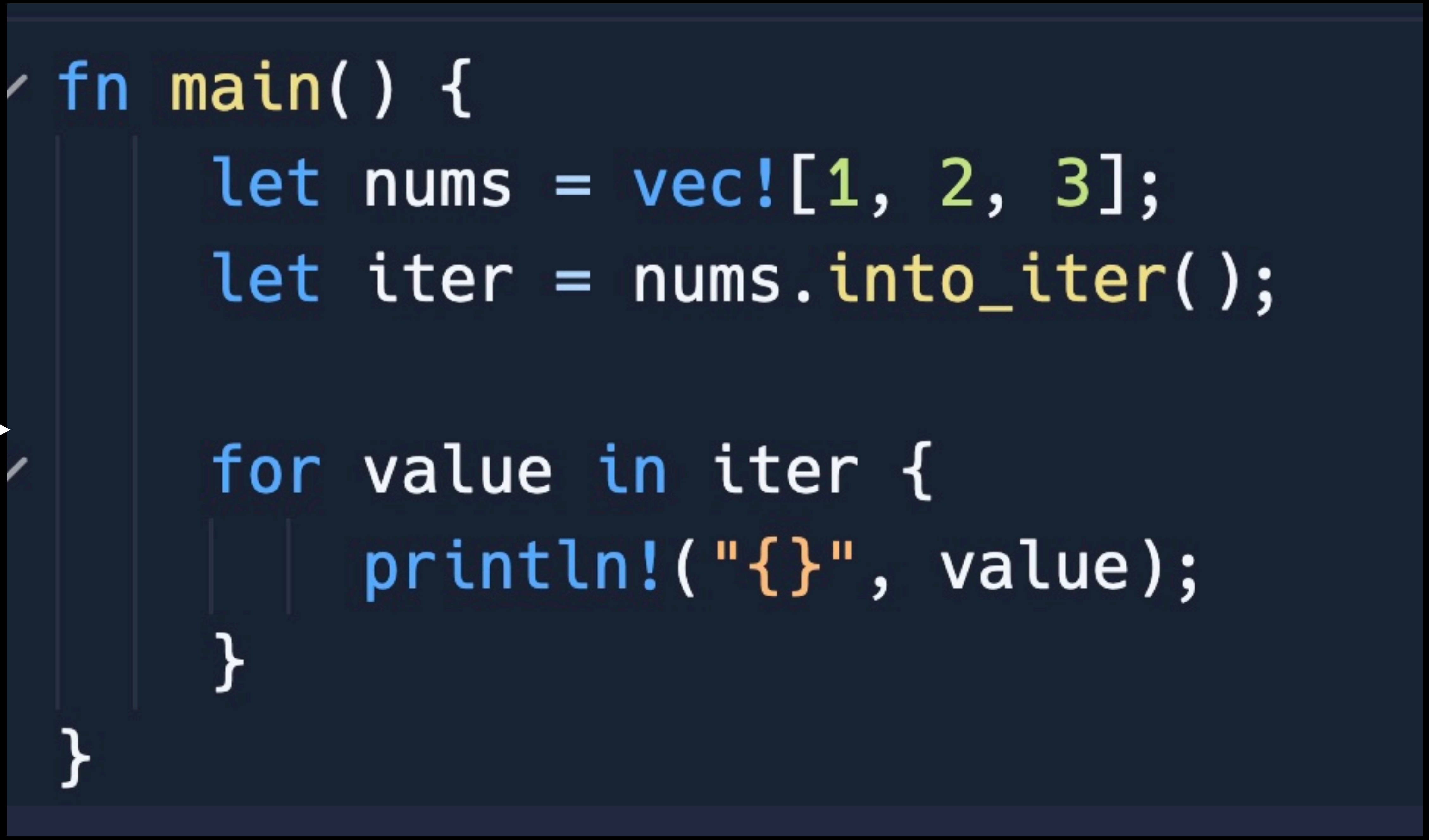
5. IntoIter

The for syntax when applied directly on the collection uses into_iter under the hood



```rust
fn main() {
    let nums = vec![1, 2, 3];

    for value in nums {
        println!("{}", value);
    }
}
```

same



```rust
fn main() {
    let nums = vec![1, 2, 3];
    let iter = nums.into_iter();

    for value in iter {
        println!("{}", value);
    }
}
```

Iterators

Which to chose?

Iter

If you want immutable references to the inner variables and don't want to transfer ownership

IterMut

If you want mutable references to the inner variables and don't want to transfer ownership

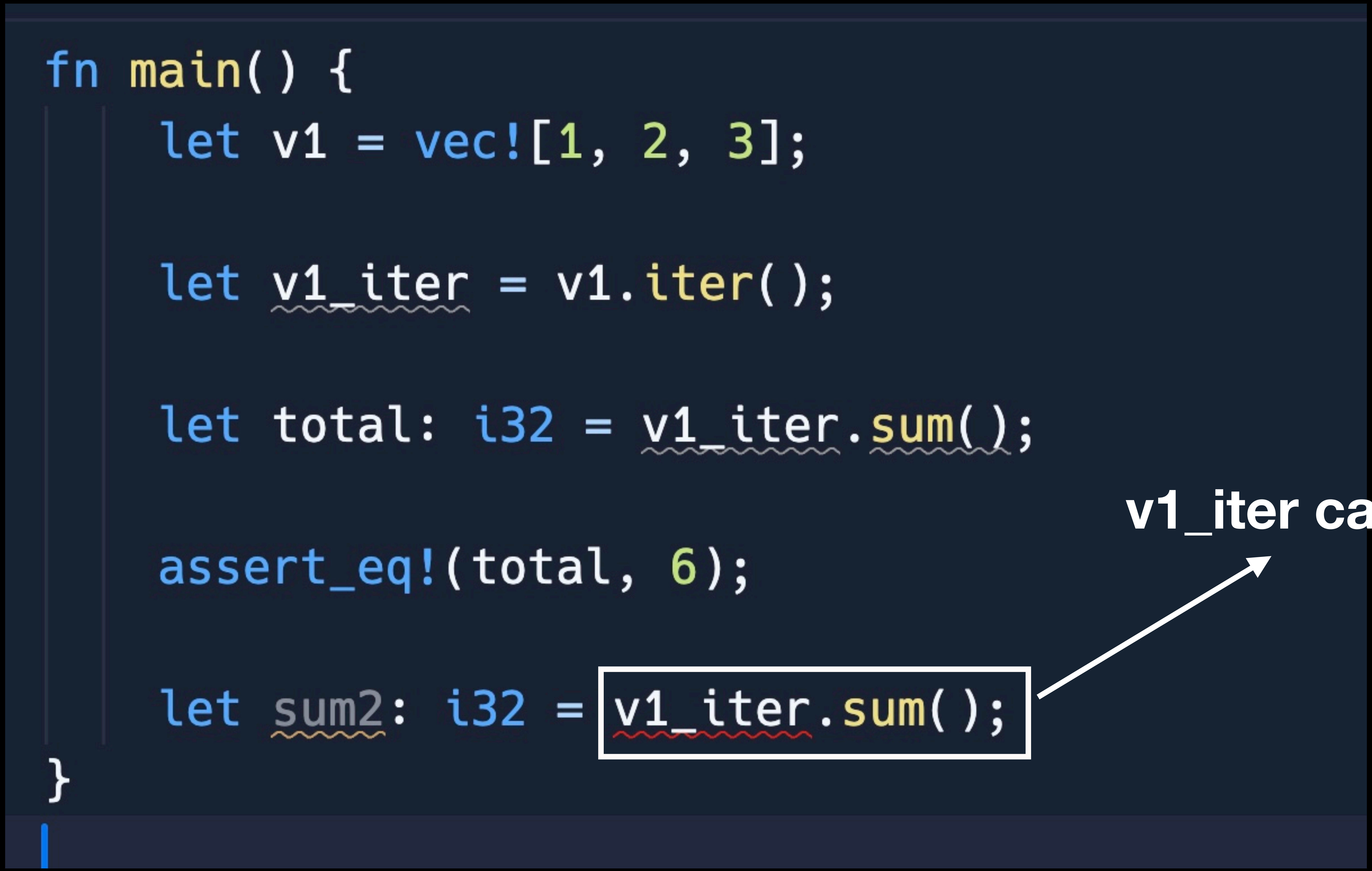
IterInto

If you want to move the variable into the iterator and don't want to use it afterwards

Iterators

Consuming adapters

Methods that call next are called consuming adaptors, because calling them uses up the iterator.



```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);

    let sum2: i32 = v1_iter.sum();
}
```

v1_iter can’t be used again

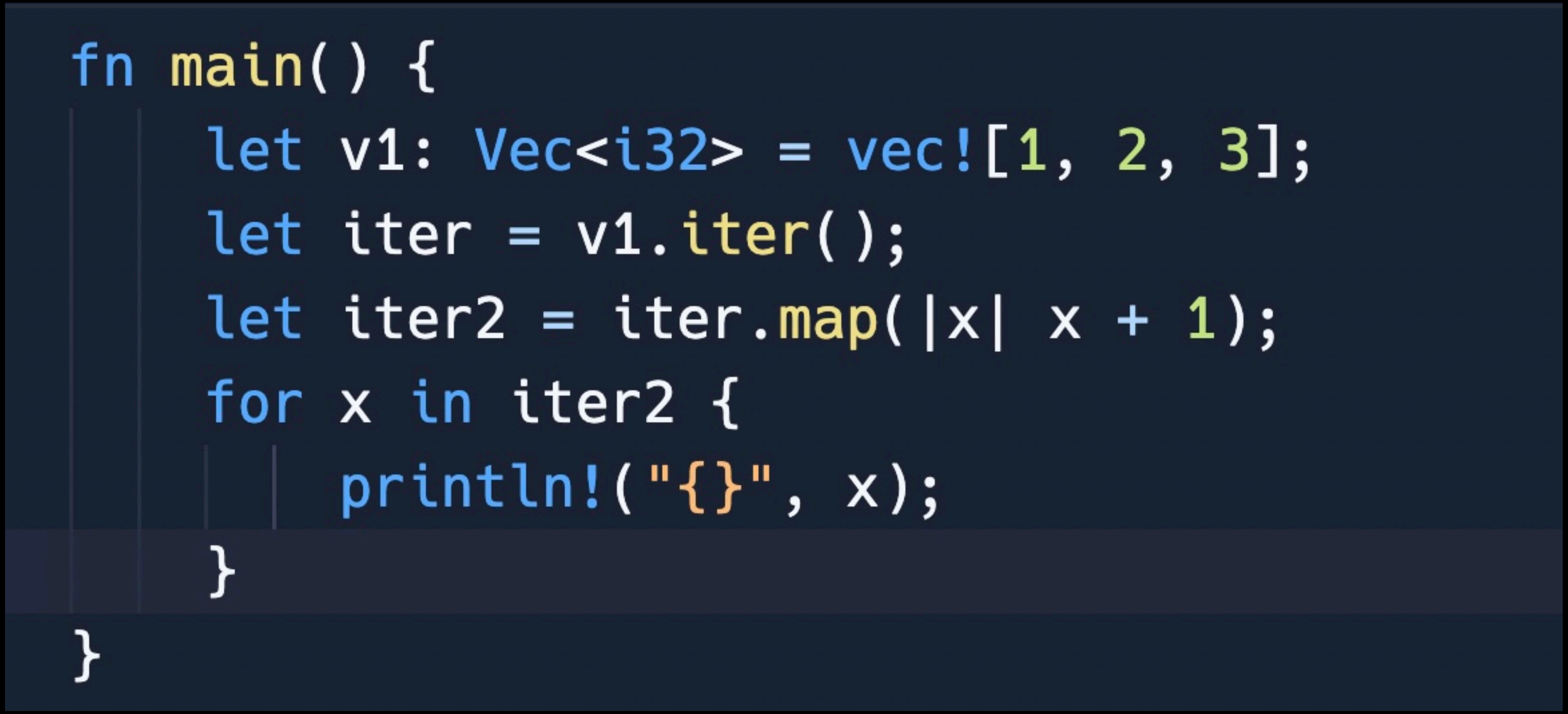
Iterators

Iterator adaptors are methods defined on the Iterator trait that don’t consume the iterator. Instead, they produce different iterators by changing some aspect of the original iterator.

Iterators

Iterator adaptors are methods defined on the Iterator trait that don't consume the iterator. Instead, they produce different iterators by changing some aspect of the original iterator.

1. **map**

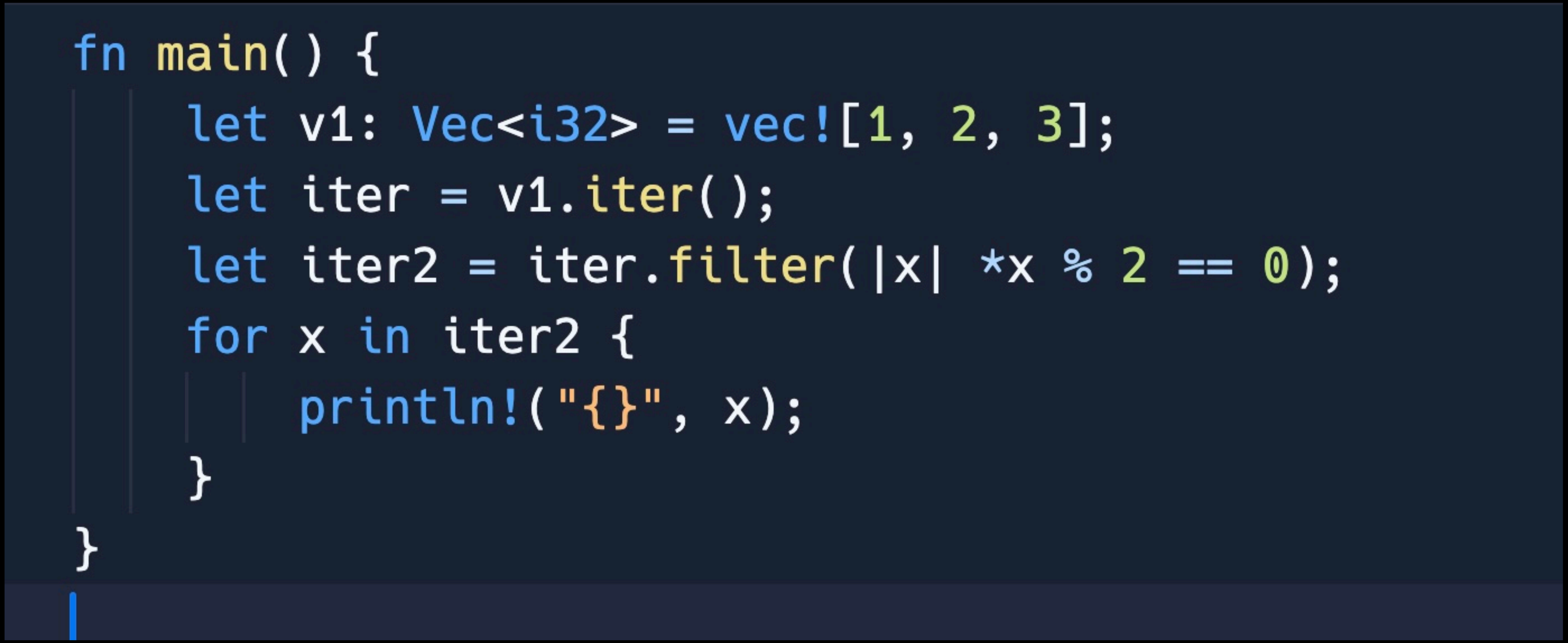


```
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let iter = v1.iter();
    let iter2 = iter.map(|x| x + 1);
    for x in iter2 {
        println!("{}", x);
    }
}
```

Iterators

Iterator adaptors are methods defined on the Iterator trait that don't consume the iterator. Instead, they produce different iterators by changing some aspect of the original iterator.

1. map
2. filter



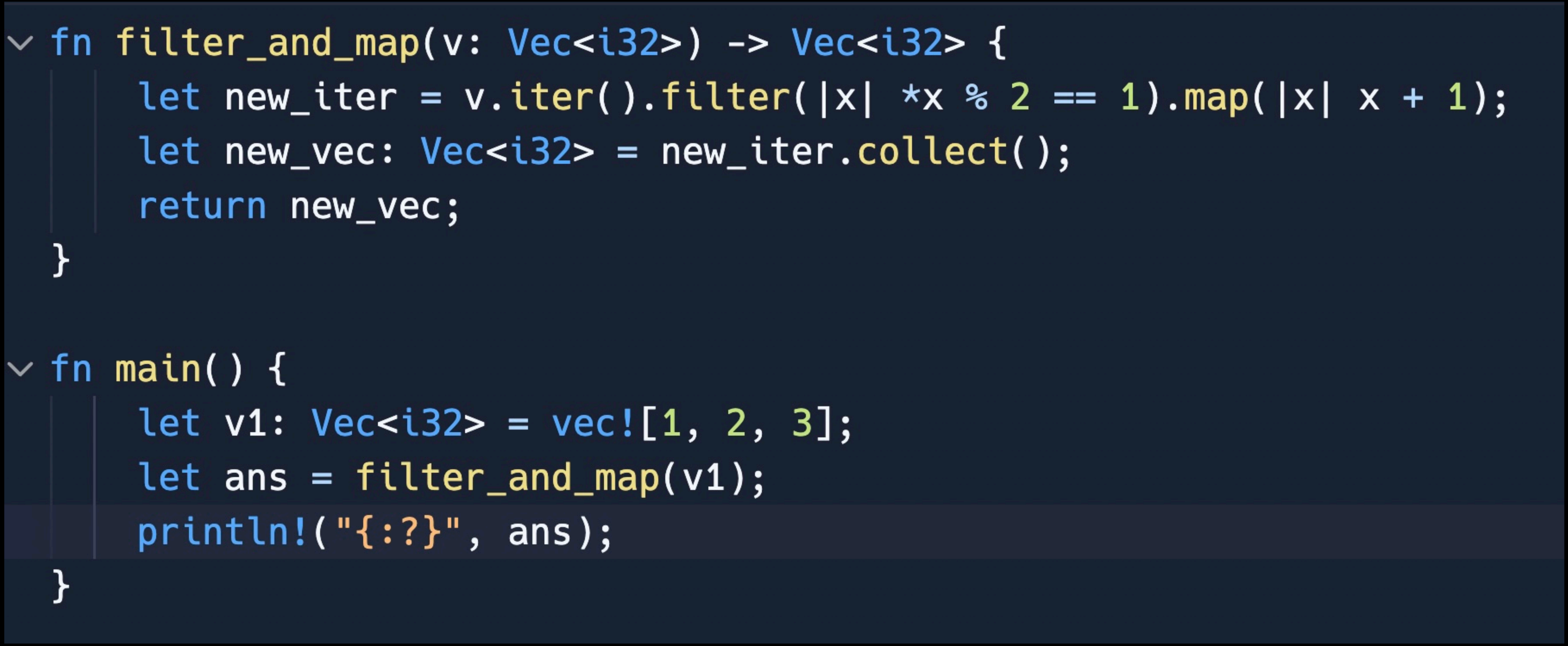
```
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let iter = v1.iter();
    let iter2 = iter.filter(|x| *x % 2 == 0);
    for x in iter2 {
        println!("{}", x);
    }
}
```

Iterators

Assignment -
Write the logic to first filter all odd values then double each value and create a new vector

Iterators

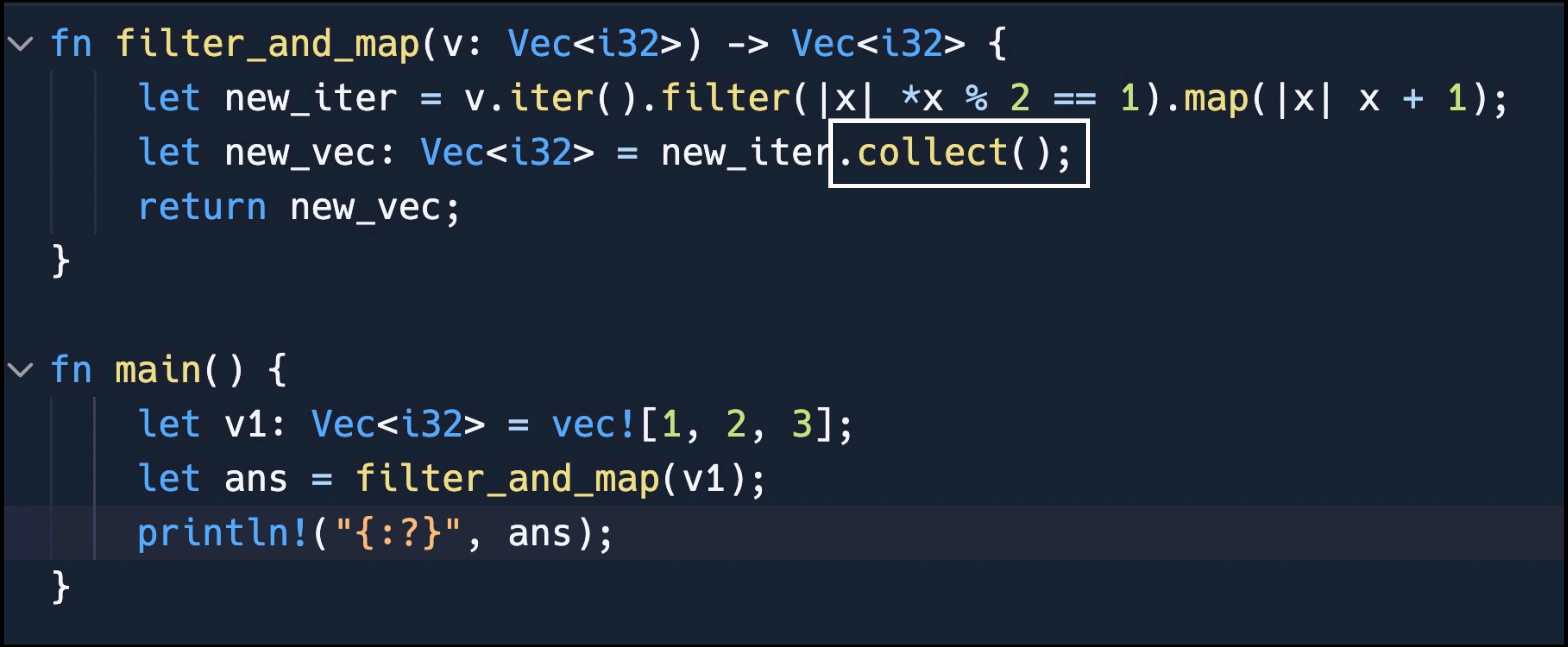
Assignment -
Write the logic to first filter all odd values then double each value and create a new vector



```rust
fn filter_and_map(v: Vec<i32>) -> Vec<i32> {
    let new_iter = v.iter().filter(|x| *x % 2 == 1).map(|x| x + 1);
    let new_vec: Vec<i32> = new_iter.collect();
    return new_vec;
}

fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let ans = filter_and_map(v1);
    println!("{:?}", ans);
}
```

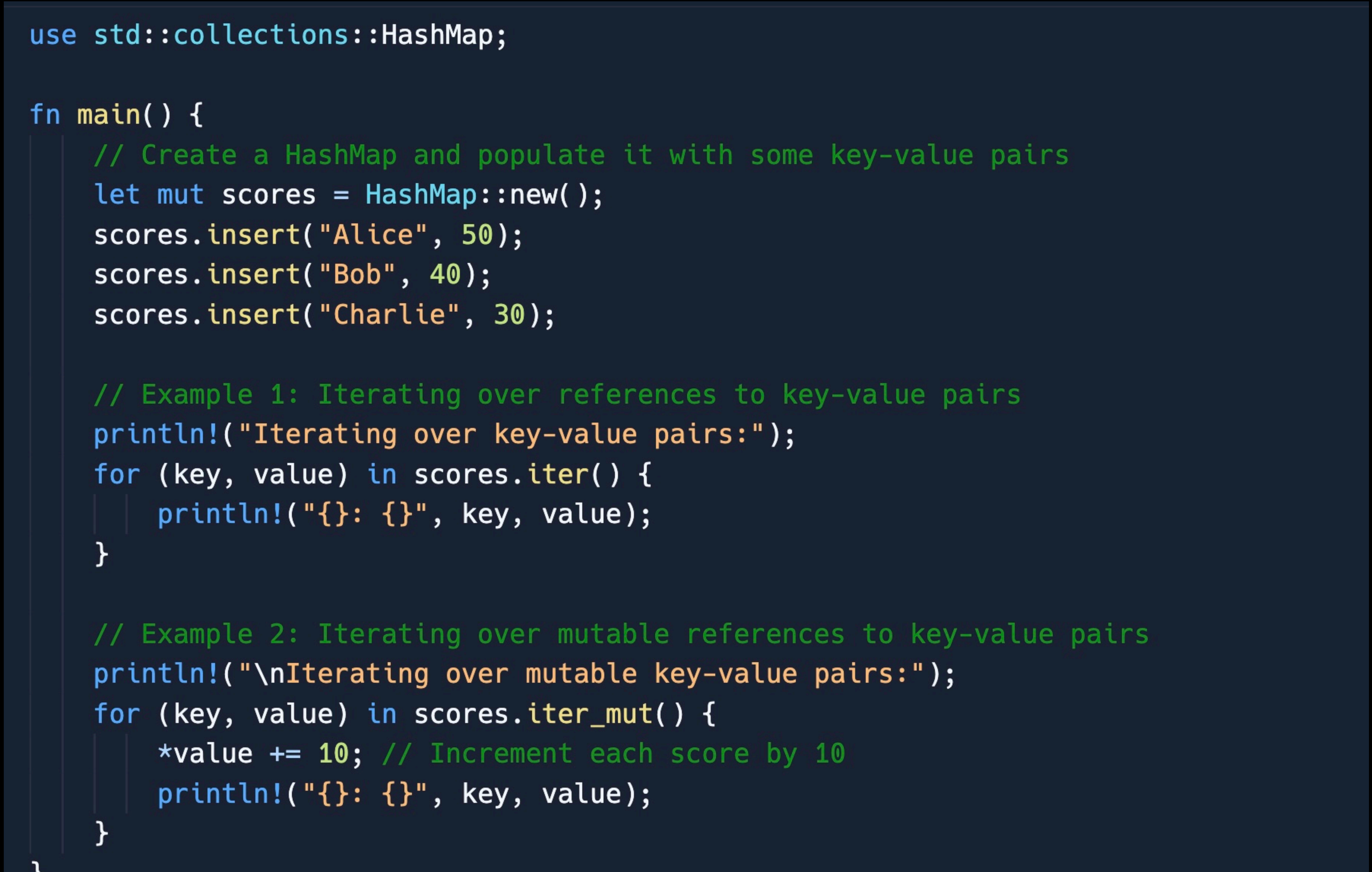
Iterators

Assignment -
Write the logic to first filter all odd values then double each value and create a new vector



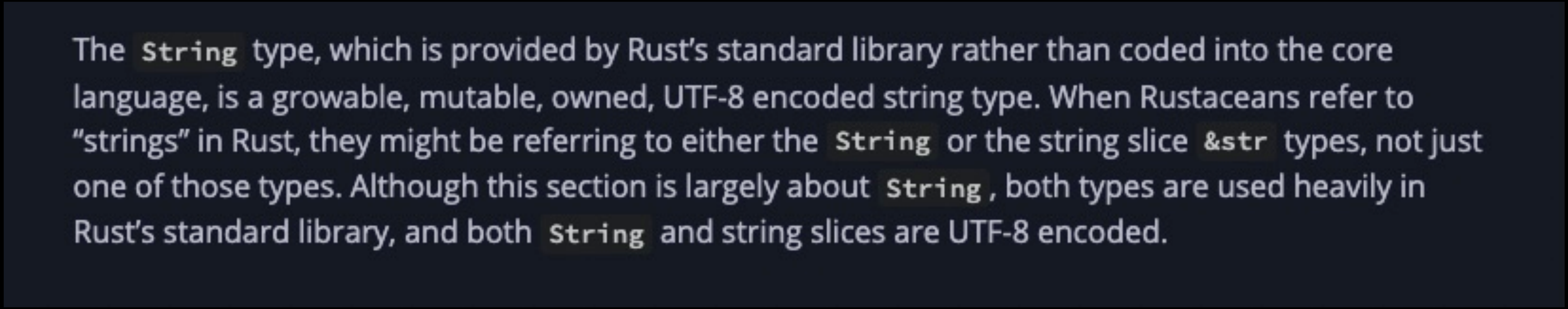
```rust
fn filter_and_map(v: Vec<i32>) -> Vec<i32> {
    let new_iter = v.iter().filter(|x| *x % 2 == 1).map(|x| x + 1);
    let new_vec: Vec<i32> = new_iter.collect();
    return new_vec;
}

fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let ans = filter_and_map(v1);
    println!("{:?}", ans);
}
```

Iterators in Hashmaps

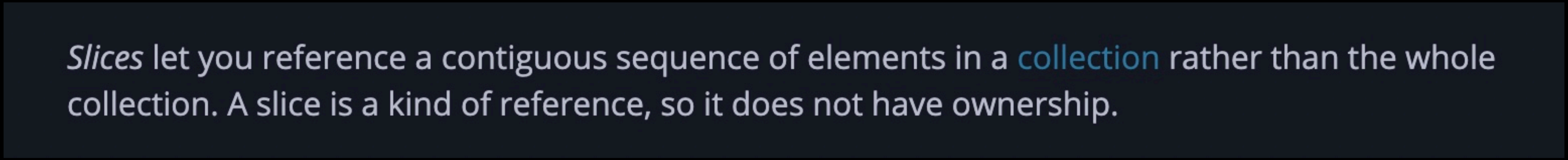


```rust
use std::collections::HashMap;

fn main() {
    // Create a HashMap and populate it with some key-value pairs
    let mut scores = HashMap::new();
    scores.insert("Alice", 50);
    scores.insert("Bob", 40);
    scores.insert("Charlie", 30);

    // Example 1: Iterating over references to key-value pairs
    println!("Iterating over key-value pairs:");
    for (key, value) in scores.iter() {
        println!("{}: {}", key, value);
    }

    // Example 2: Iterating over mutable references to key-value pairs
    println!("\nIterating over mutable key-value pairs:");
    for (key, value) in scores.iter_mut() {
        *value += 10; // Increment each score by 10
        println!("{}: {}", key, value);
    }
}
```

Strings vs slices



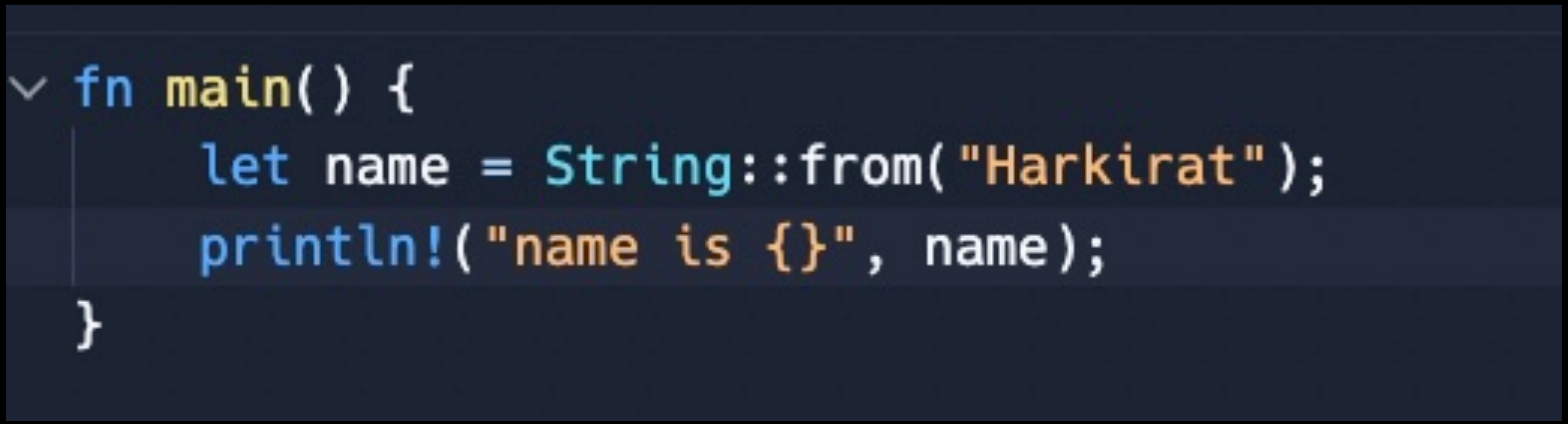
The `String` type, which is provided by Rust’s standard library rather than coded into the core language, is a growable, mutable, owned, UTF-8 encoded string type. When Rustaceans refer to “strings” in Rust, they might be referring to either the `String` or the string slice `&str` types, not just one of those types. Although this section is largely about `String`, both types are used heavily in Rust’s standard library, and both `String` and string slices are UTF-8 encoded.



Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection. A slice is a kind of reference, so it does not have ownership.

Strings

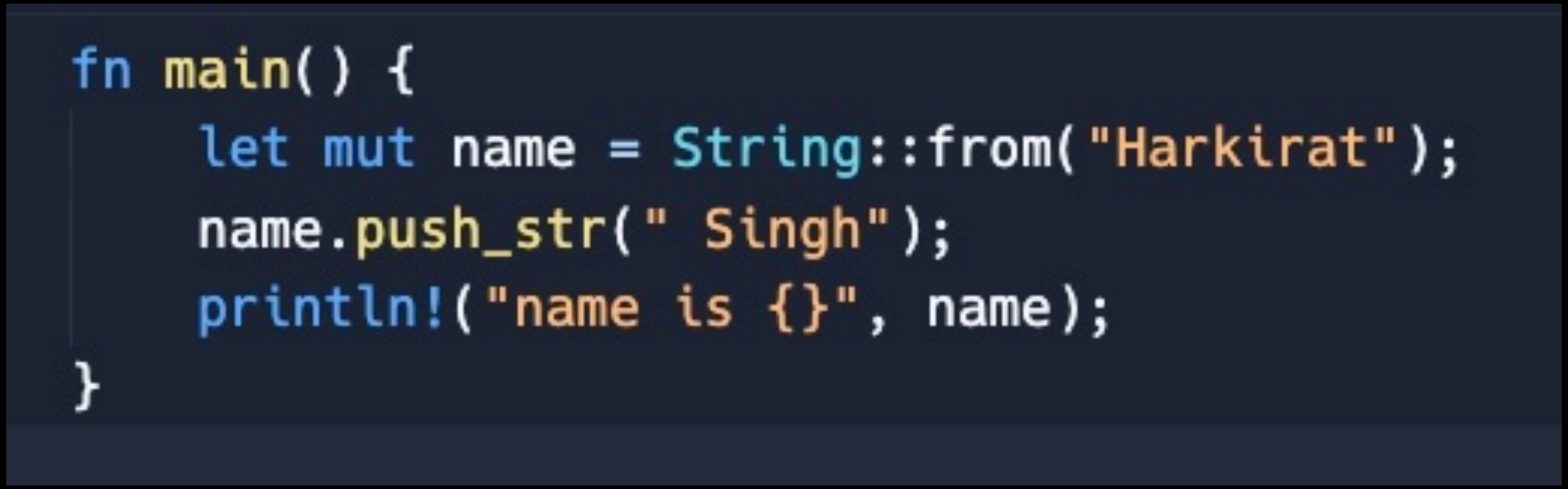
1. Creating a string



```
fn main() {
    let name = String::from("Harkirat");
    println!("name is {}", name);
}
```

Strings

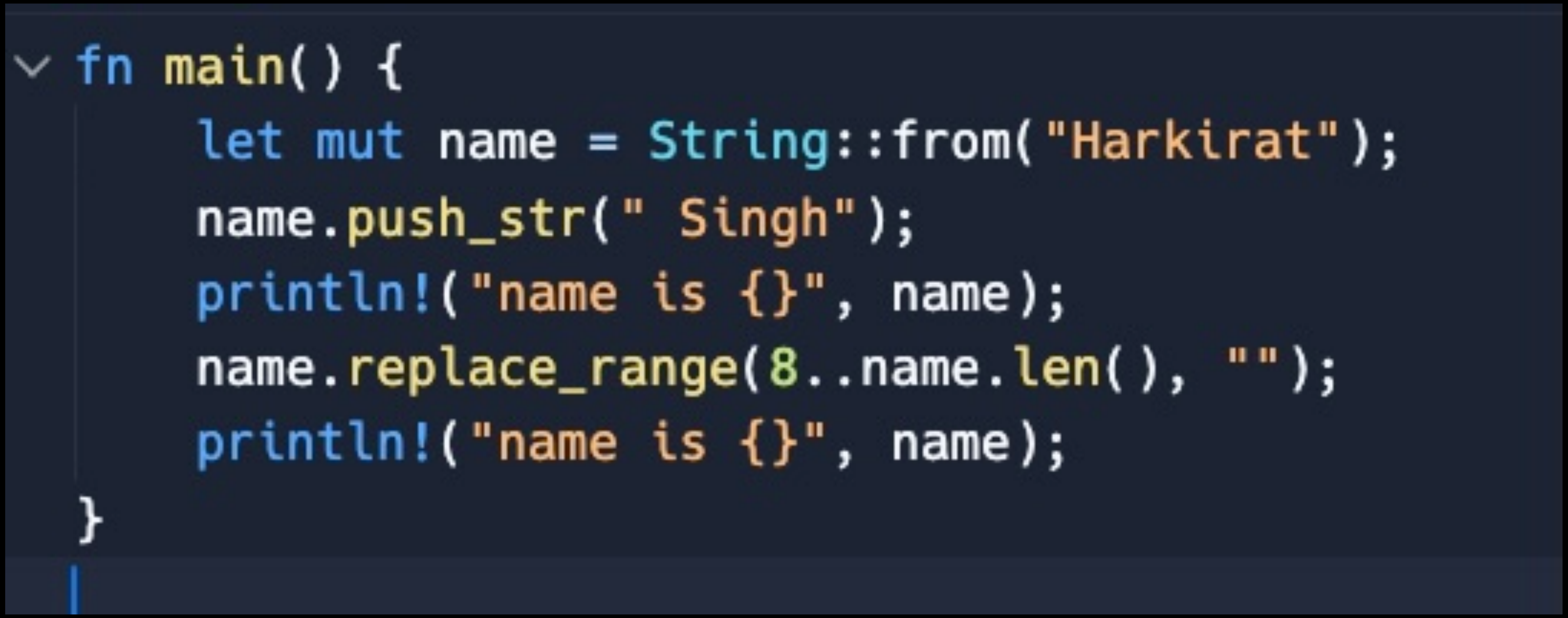
1. Creating a string
2. Mutating a string



```rust
fn main() {
    let mut name = String::from("Harkirat");
    name.push_str(" Singh");
    println!("name is {}", name);
}
```

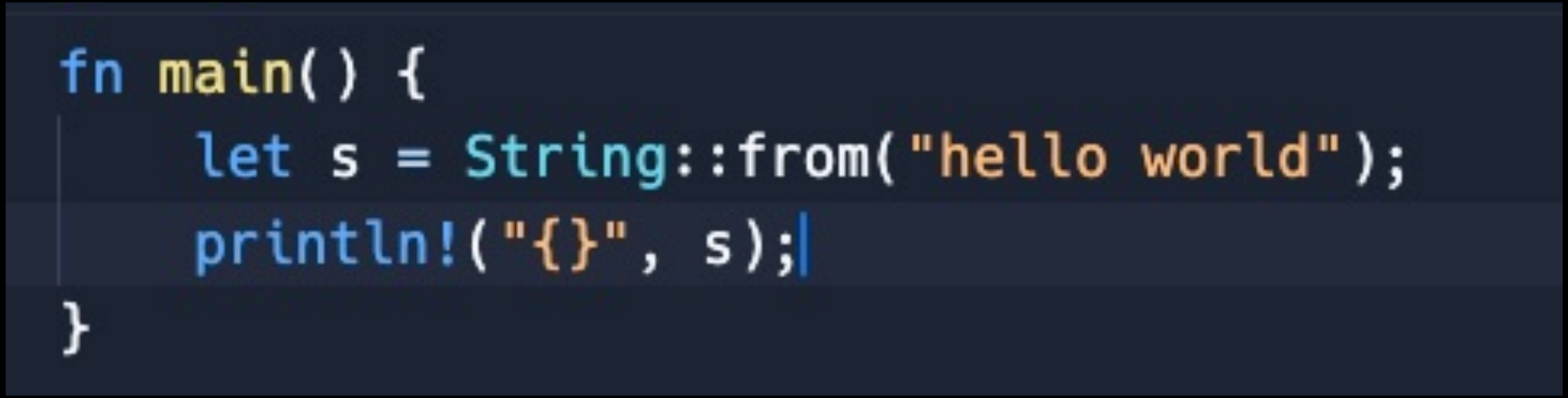
Strings

1. Creating a string
2. Mutating a string
3. Deleting from a string

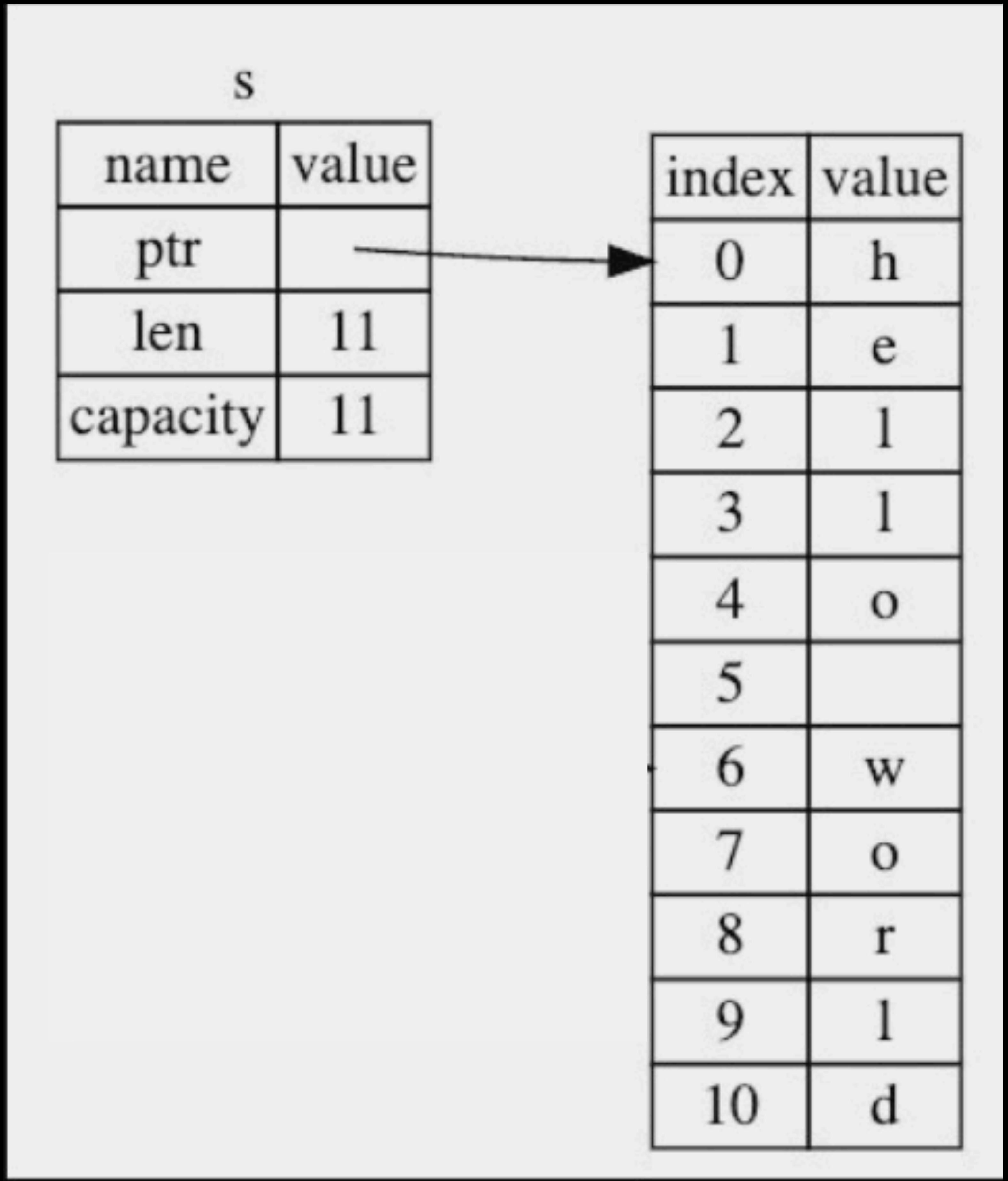


```rust
fn main() {
    let mut name = String::from("Harkirat");
    name.push_str(" Singh");
    println!("name is {}", name);
    name.replace_range(8..name.len(), "");
    println!("name is {}", name);
}
```

Strings



```
fn main() {
    let s = String::from("hello world");
    println!("{}", s);
}
```

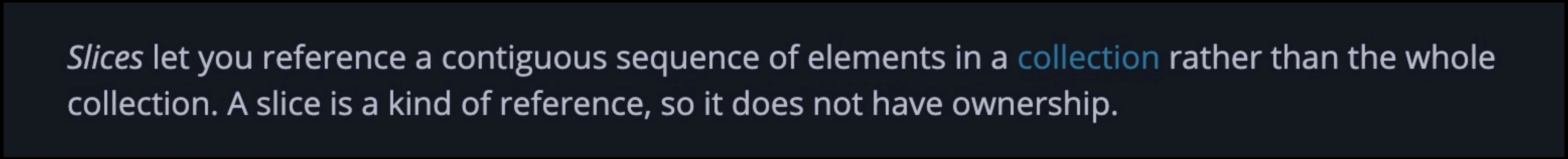


s

name	value
ptr	
len	11
capacity	11

index	value
0	h
1	e
2	l
3	l
4	o
5	
6	w
7	o
8	r
9	l
10	d

Slices



Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection. A slice is a kind of reference, so it does not have ownership.

Slices

Lets understand why we need slices
With a common example (from the rust book) of how it provides memory safety

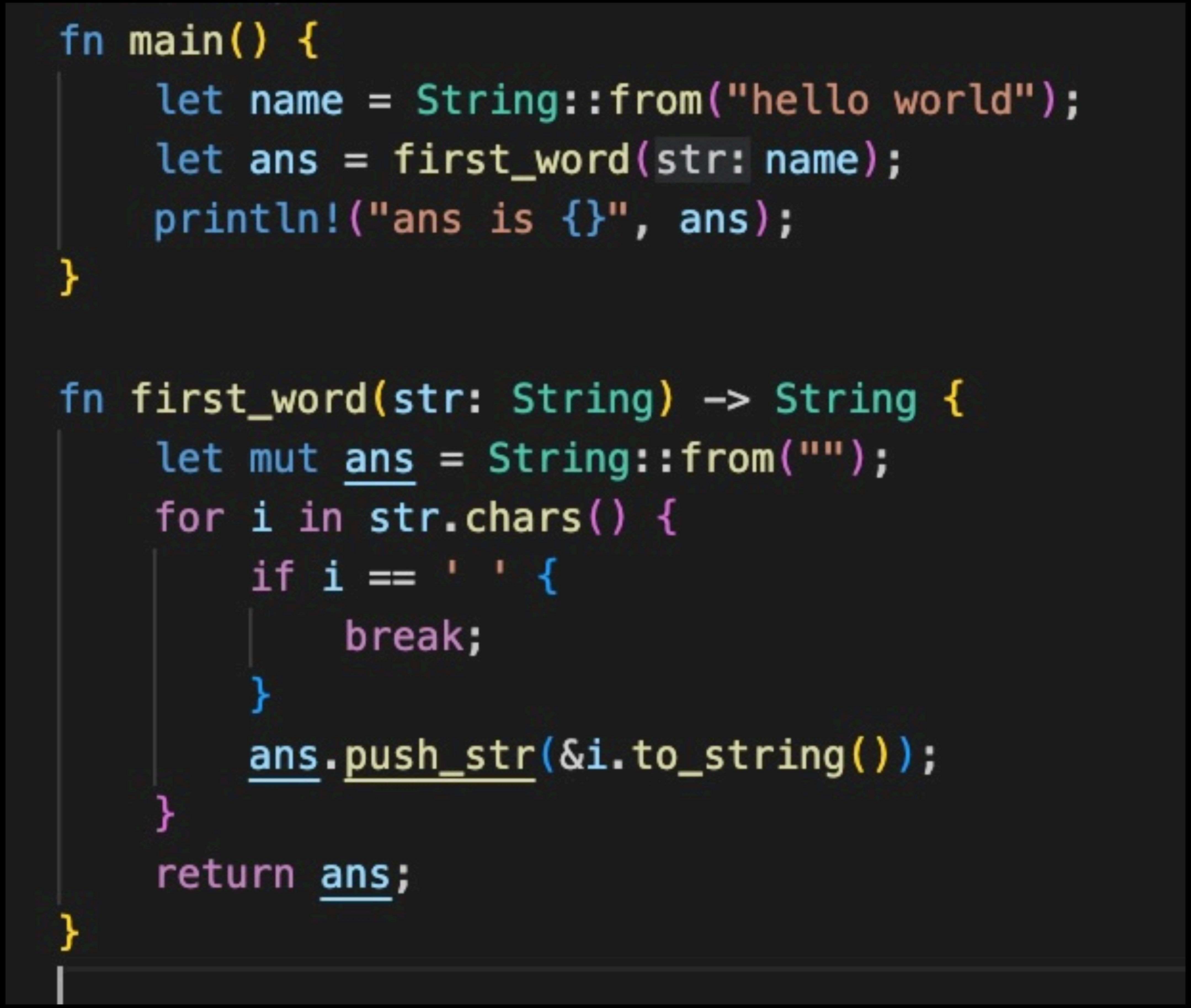
Slices

Q: Write a function that takes a string as an input
And returns the first word from it

Slices

Q: Write a function that takes a string as an input
And returns the first word from it

Approach #1 - Return a new string



```
fn main() {
    let name = String::from("hello world");
    let ans = first_word(str: name);
    println!("ans is {}", ans);
}

fn first_word(str: String) -> String {
    let mut ans = String::from("");
    for i in str.chars() {
        if i == ' ' {
            break;
        }
        ans.push_str(&i.to_string());
    }
    return ans;
}
```

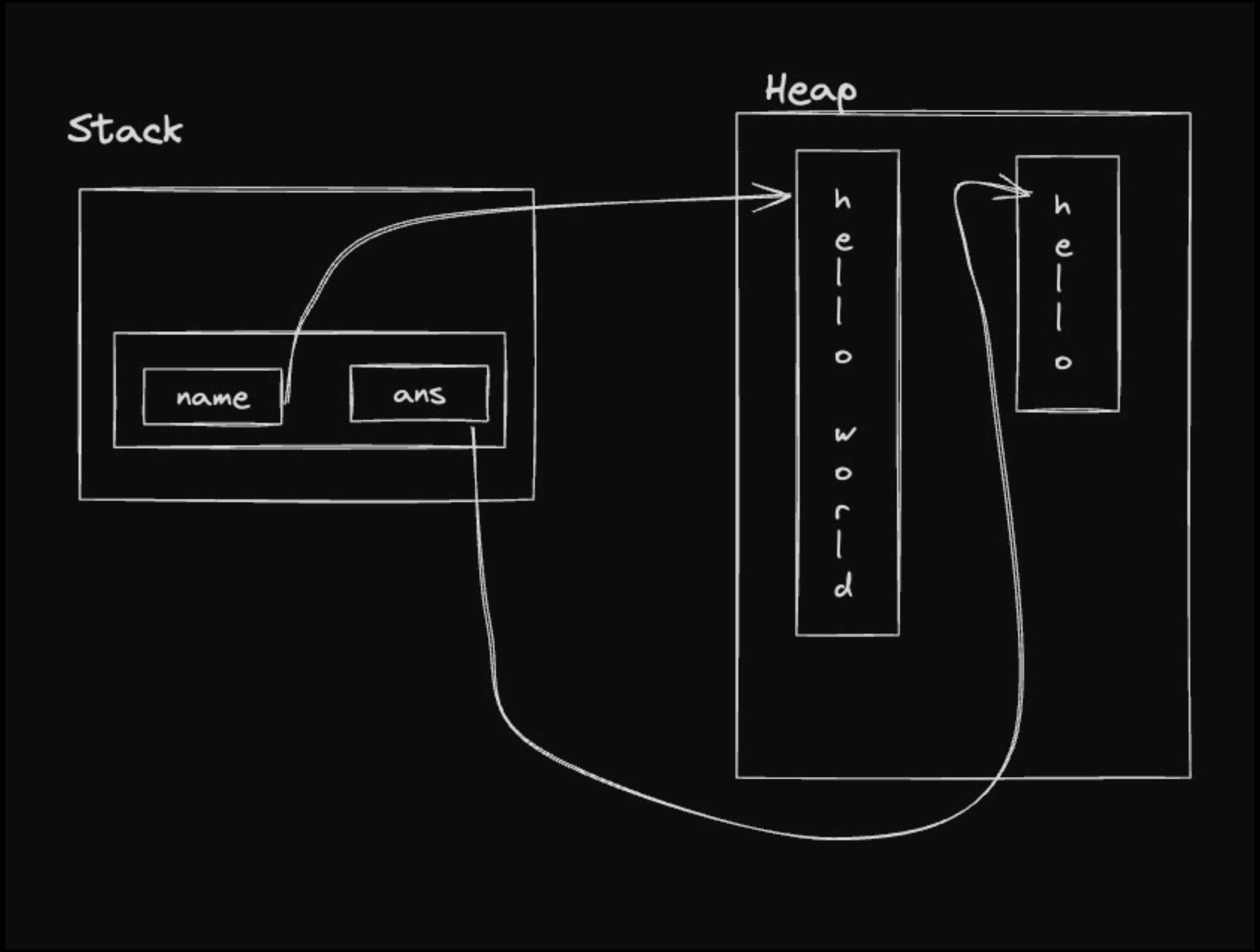
Slices

**Q: Write a function that takes a string as an input
And returns the first word from it**

Approach #1 - Return a new string

Problem -

1. **We take up double the memory**
2. **If the `name` string gets `cleared` , `ans` still has `hello` as the value in it**



Slices

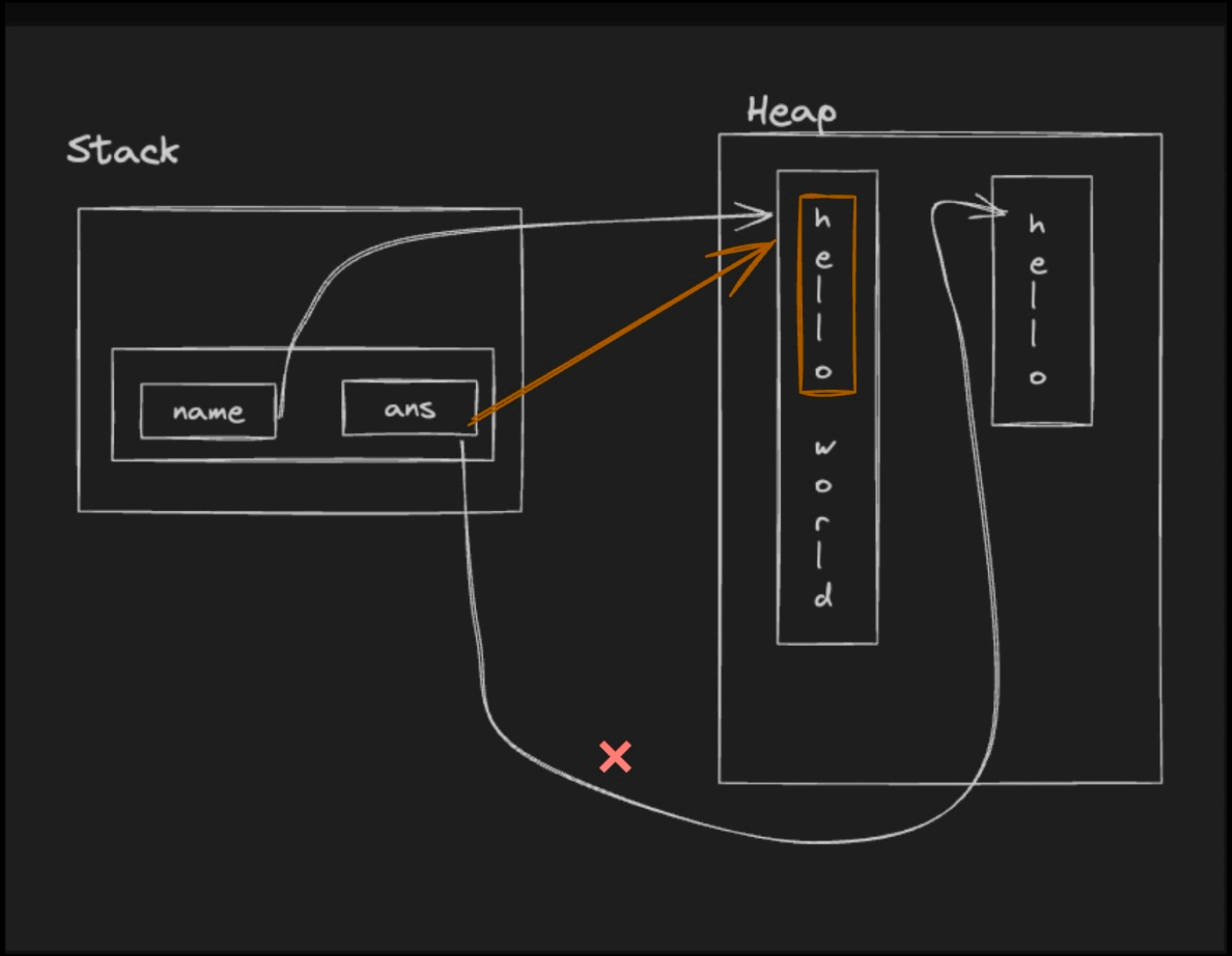
Q: Write a function that takes a string as an input
And returns the first word from it

Approach #1 - Return a new string

Problem -

1. We take up double the memory
2. If the `name` string gets `cleared` ,
   `ans` still has `hello` as the value in it

What we want is a `view` into the original string
And not copy it over



Slices

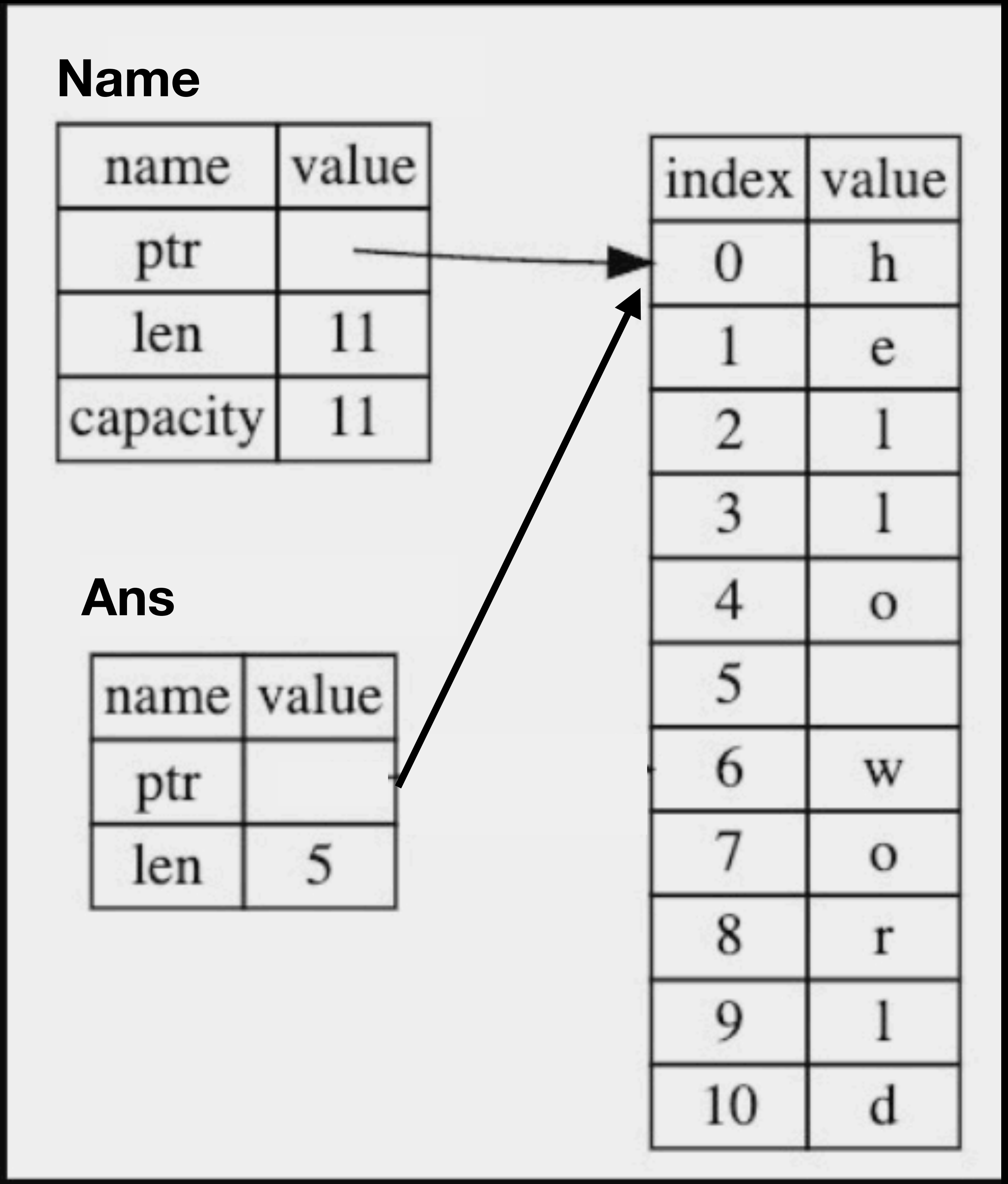
Q: Write a function that takes a string as an input
And returns the first word from it

Approach #1 - Return a new string

Problem -

1. We take up double the memory
2. If the `name` string gets `cleared` ,
   `ans` still has `hello` as the value in it

What we want is a `view` into the original string
And not copy it over



Name

name	value
ptr	
len	11
capacity	11

Ans

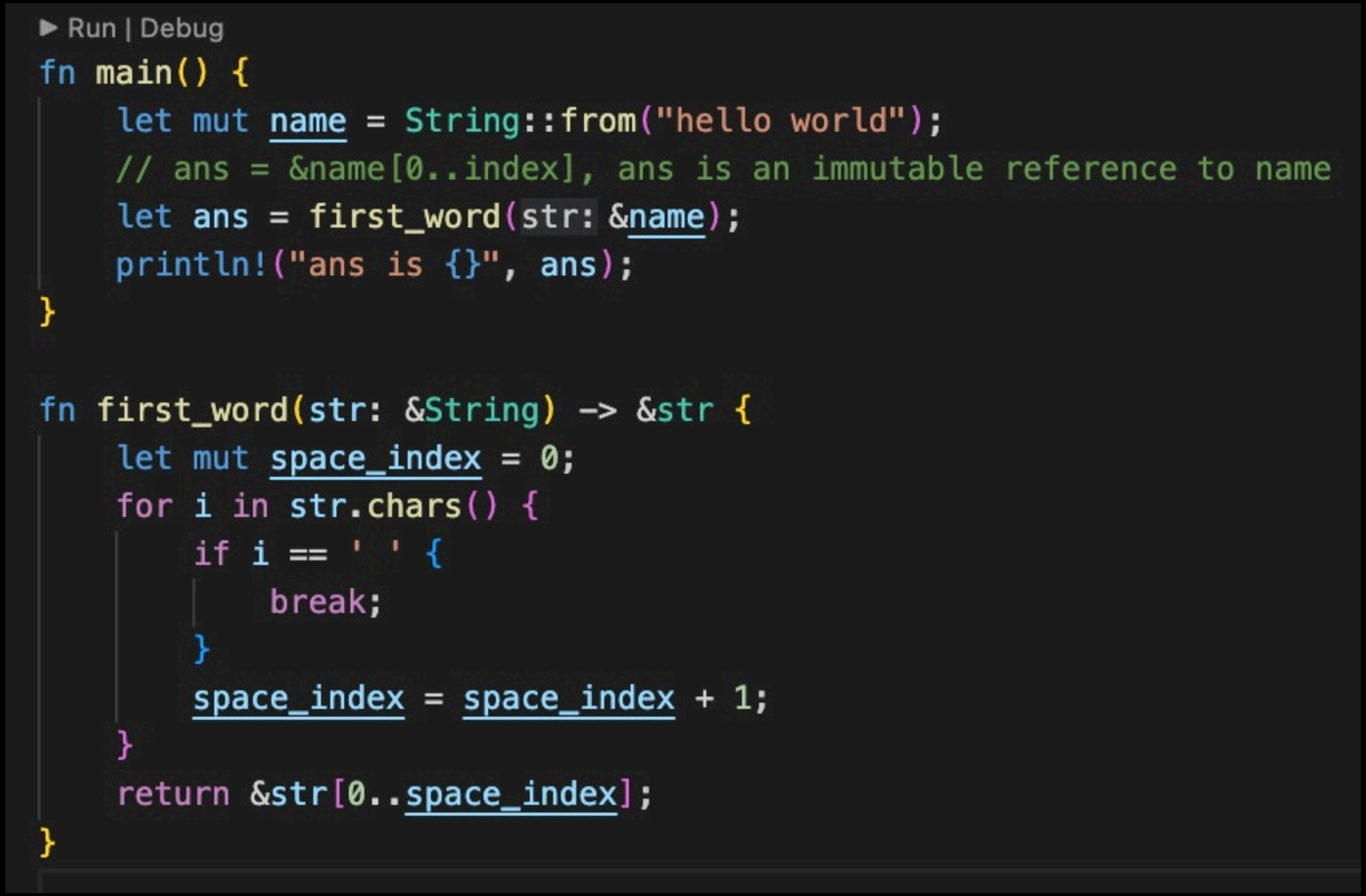
name	value
ptr	
len	5

index	value
0	h
1	e
2	l
3	l
4	o
5	
6	w
7	o
8	r
9	l
10	d

Slices

Q: Write a function that takes a string as an input
And returns the first word from it

Approach #1 (with slices)

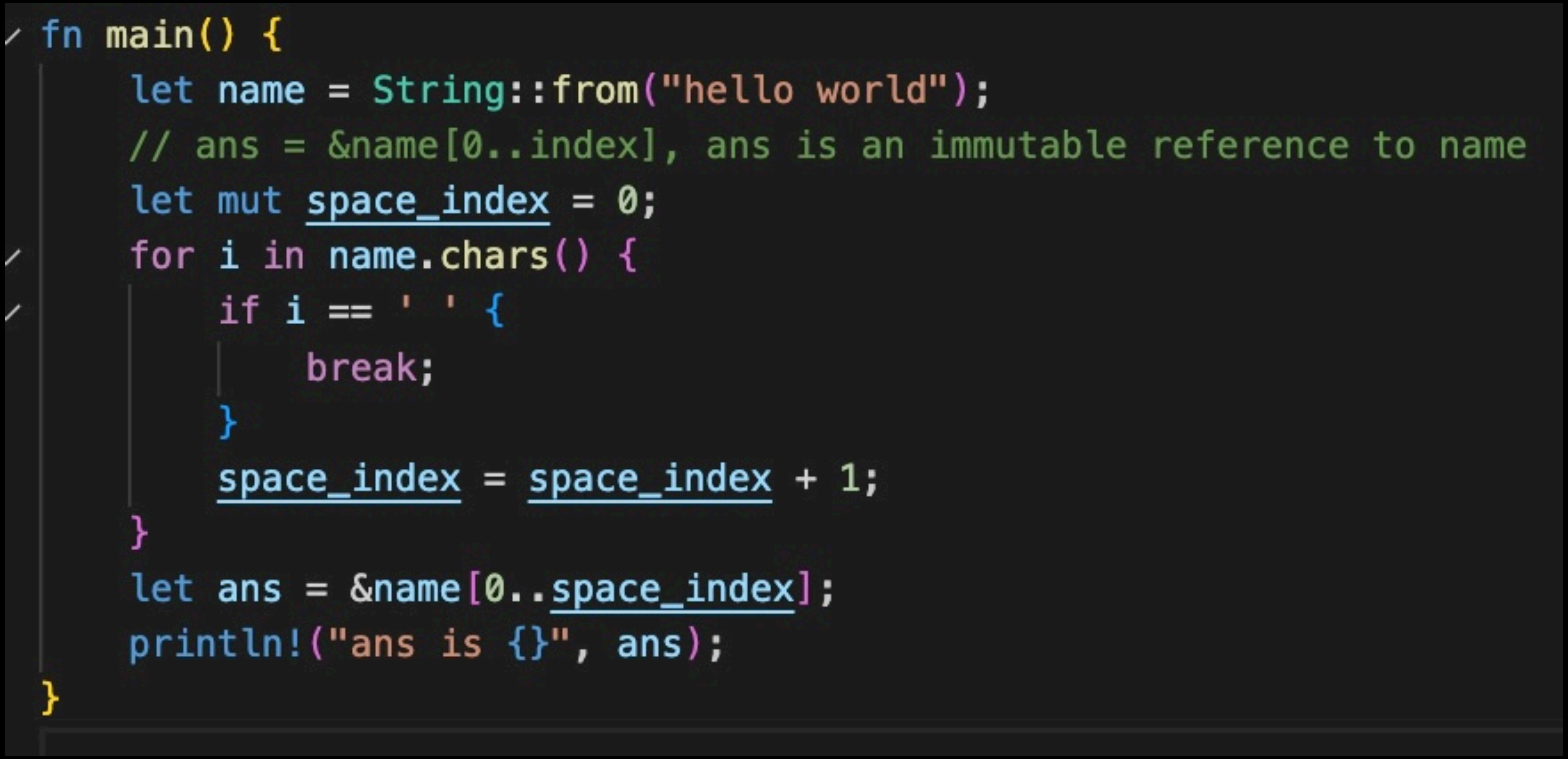


```
▶ Run | Debug
fn main() {
    let mut name = String::from("hello world");
    // ans = &name[0..index], ans is an immutable reference to name
    let ans = first_word(str: &name);
    println!("ans is {}", ans);
}

fn first_word(str: &String) -> &str {
    let mut space_index = 0;
    for i in str.chars() {
        if i == ' ' {
            break;
        }
        space_index = space_index + 1;
    }
    return &str[0..space_index];
}
```

Slices

**Q: Write a function that takes a string as an input
And returns the first word from it**

Approach #2 (with slices)

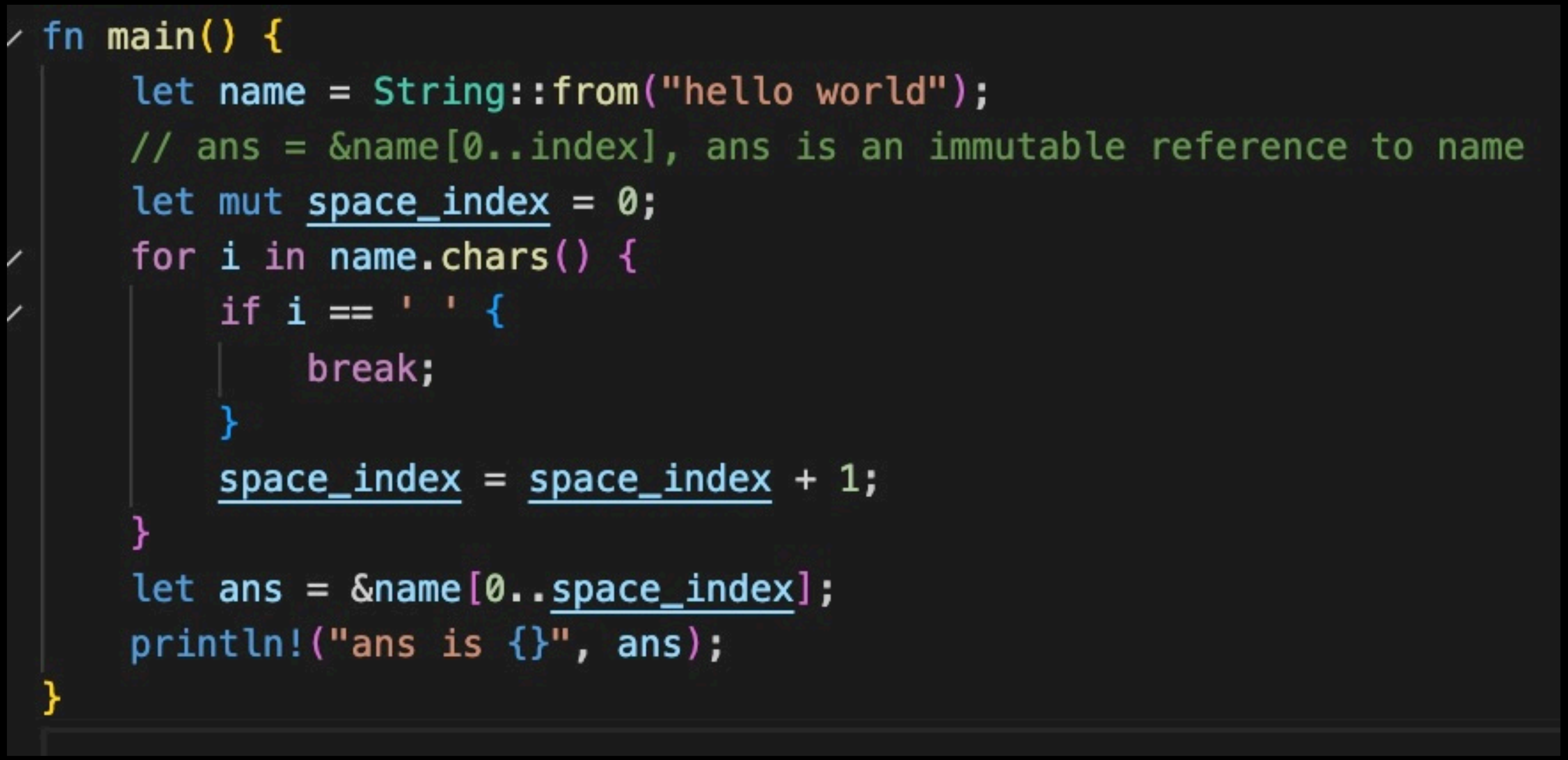


```rust
fn main() {
    let name = String::from("hello world");
    // ans = &name[0..index], ans is an immutable reference to name
    let mut space_index = 0;
    for i in name.chars() {
        if i == ' ' {
            break;
        }
        space_index = space_index + 1;
    }
    let ans = &name[0..space_index];
    println!("ans is {}", ans);
}
```

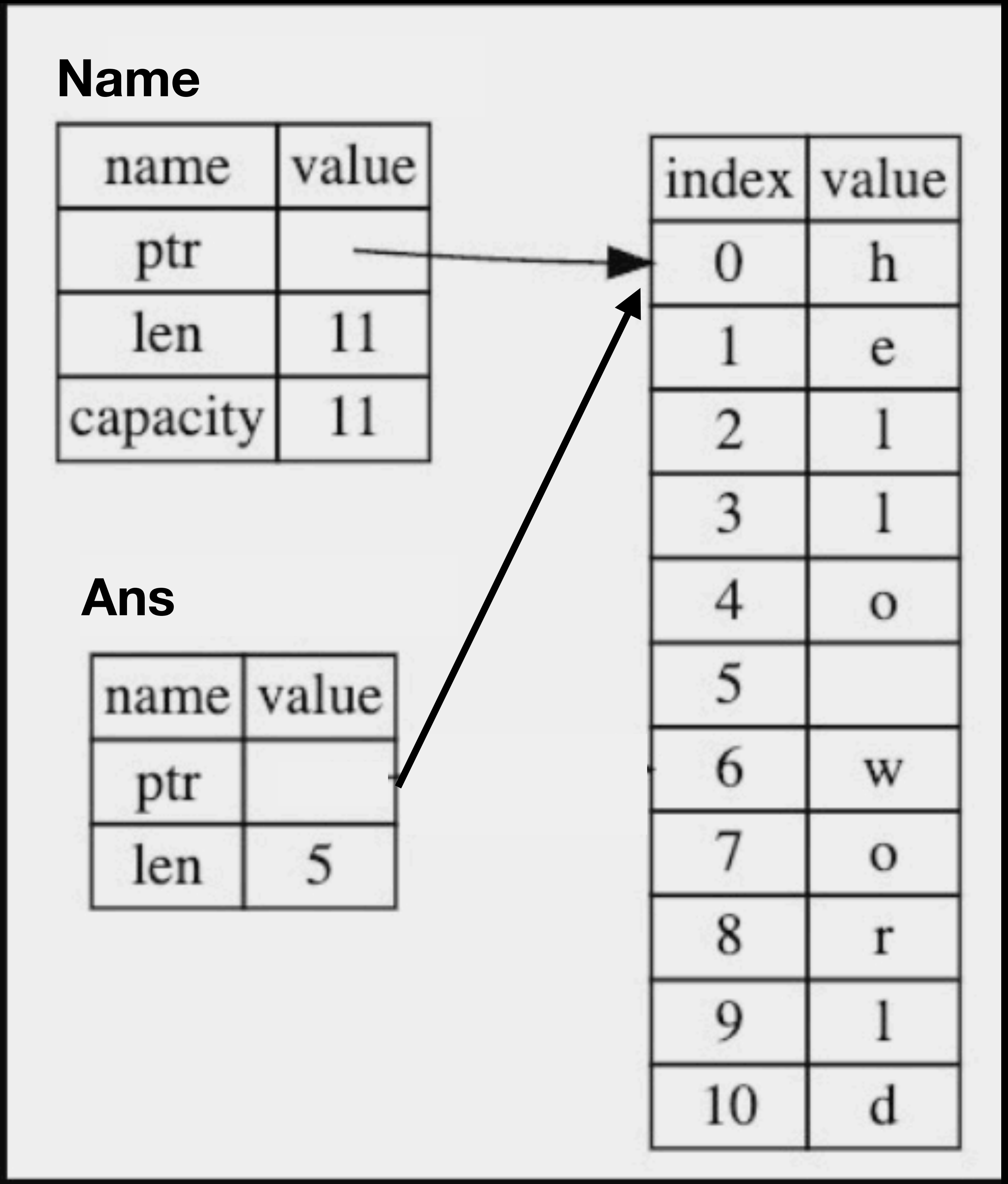
Slices

Q: Write a function that takes a string as an input
And returns the first word from it

Approach #2 (with slices) (alt)



```
fn main() {
    let name = String::from("hello world");
    // ans = &name[0..index], ans is an immutable reference to name
    let mut space_index = 0;
    for i in name.chars() {
        if i == ' ' {
            break;
        }
        space_index = space_index + 1;
    }
    let ans = &name[0..space_index];
    println!("ans is {}", ans);
}
```



Name

name	value
ptr	
len	11
capacity	11

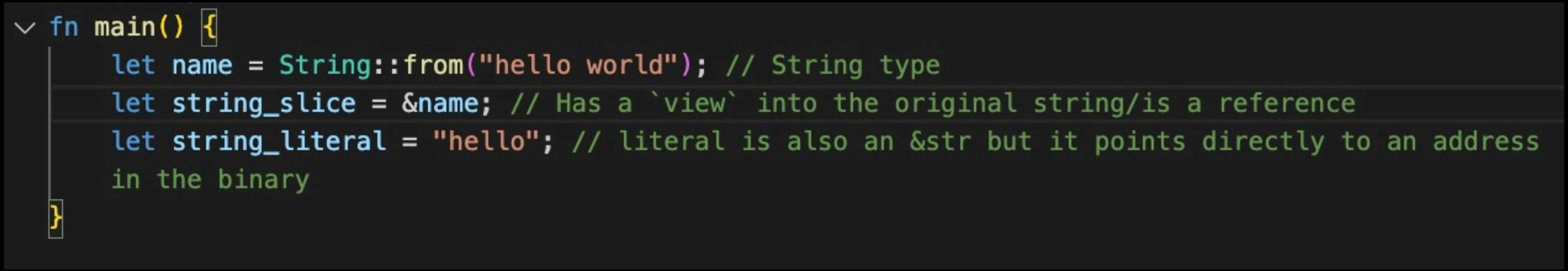
Ans

name	value
ptr	
len	5

index	value
0	h
1	e
2	l
3	l
4	o
5	
6	w
7	o
8	r
9	l
10	d

Slices

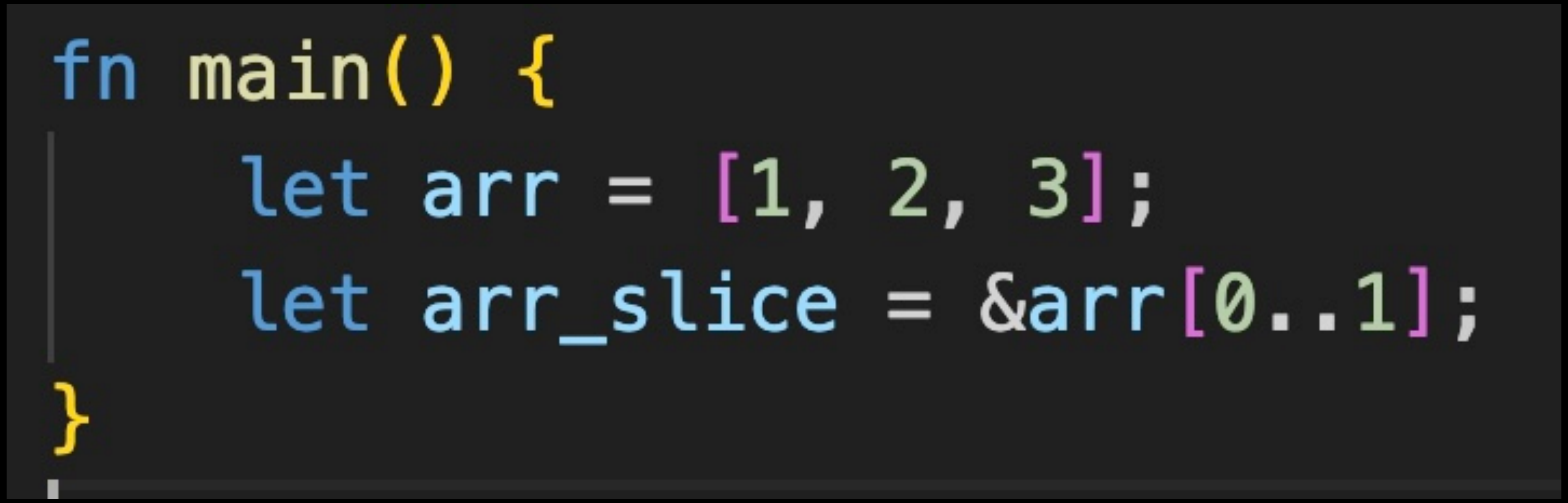
3 types of commonly used strings (there are actually much more)



```
fn main() {
    let name = String::from("hello world"); // String type
    let string_slice = &name; // Has a `view` into the original string/is a reference
    let string_literal = "hello"; // literal is also an &str but it points directly to an address
    in the binary
}
```

Slices

Slices can also be applied to other collections like vectors/arrays



```rust
fn main() {
    let arr = [1, 2, 3];
    let arr_slice = &arr[0..1];
}
```

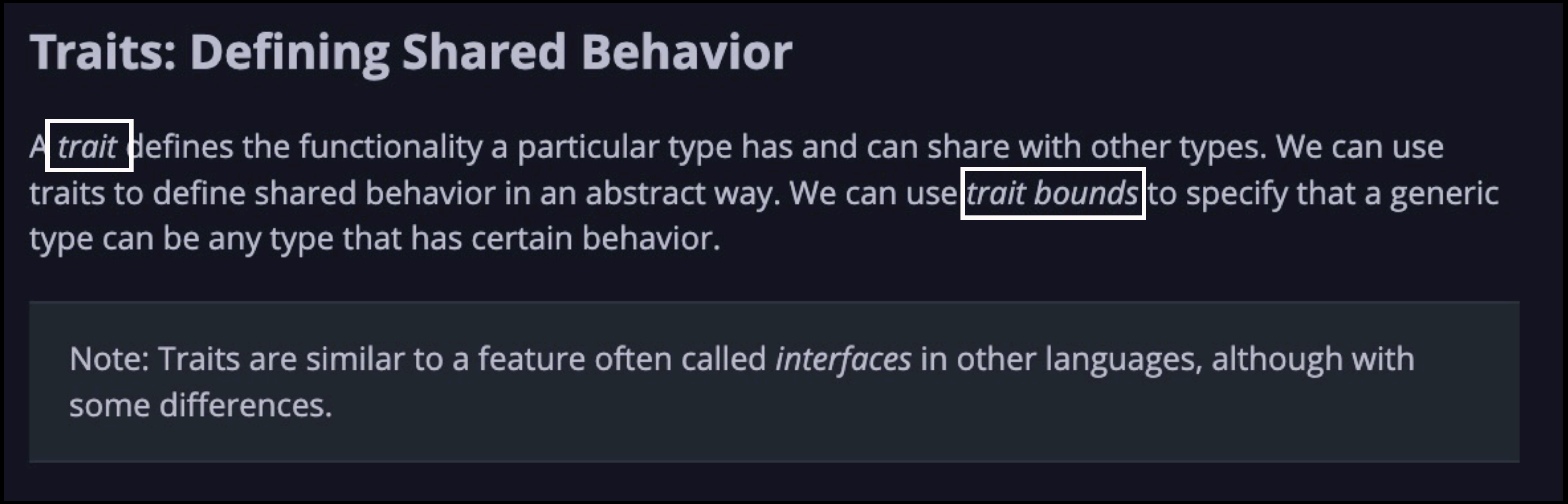
Generics

From the live meet-up

Impl blocks

From the live meet-up

Traits

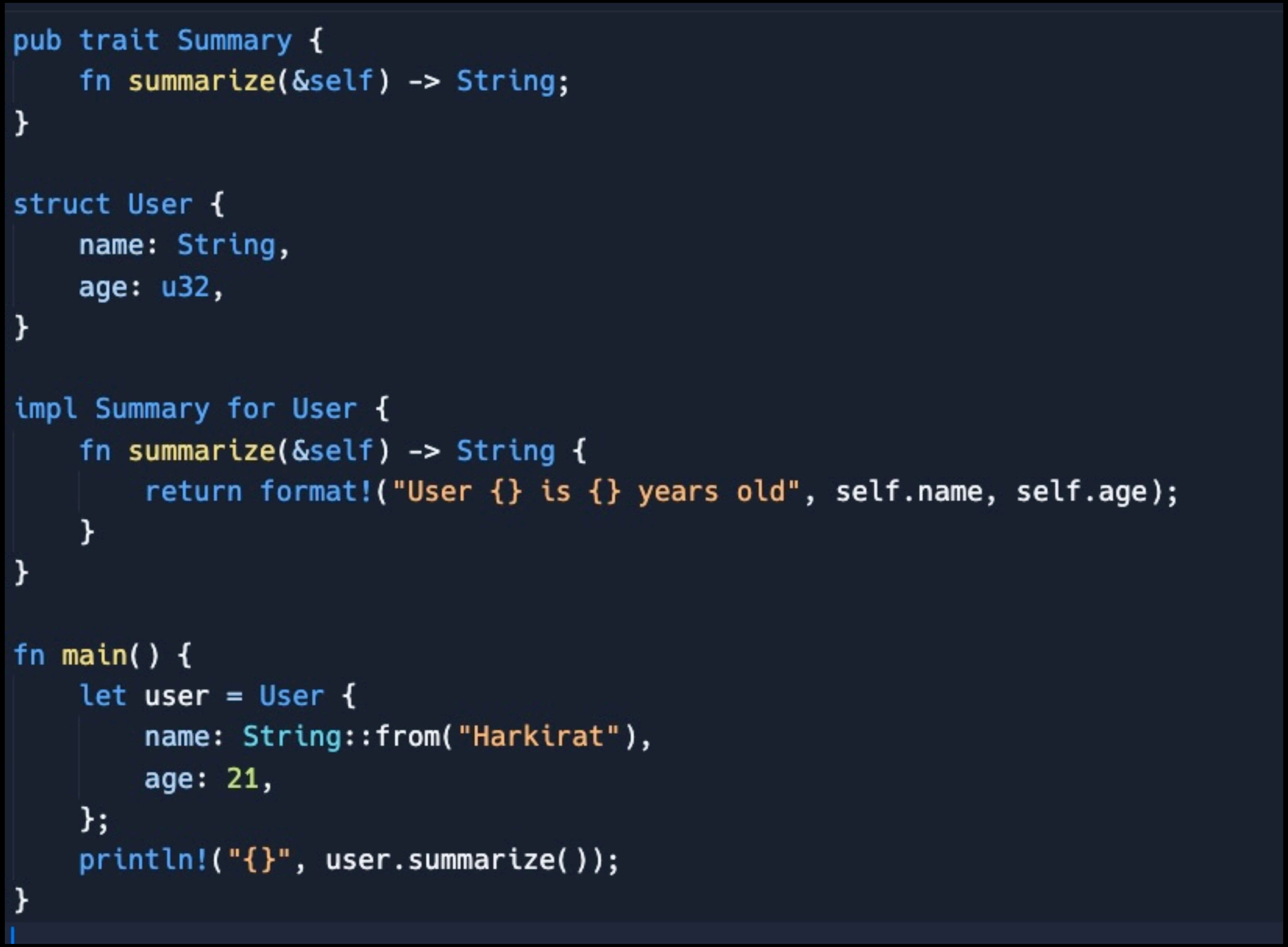


Traits: Defining Shared Behavior

A *trait* defines the functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use *trait bounds* to specify that a generic type can be any type that has certain behavior.

> Note: Traits are similar to a feature often called *interfaces* in other languages, although with some differences.

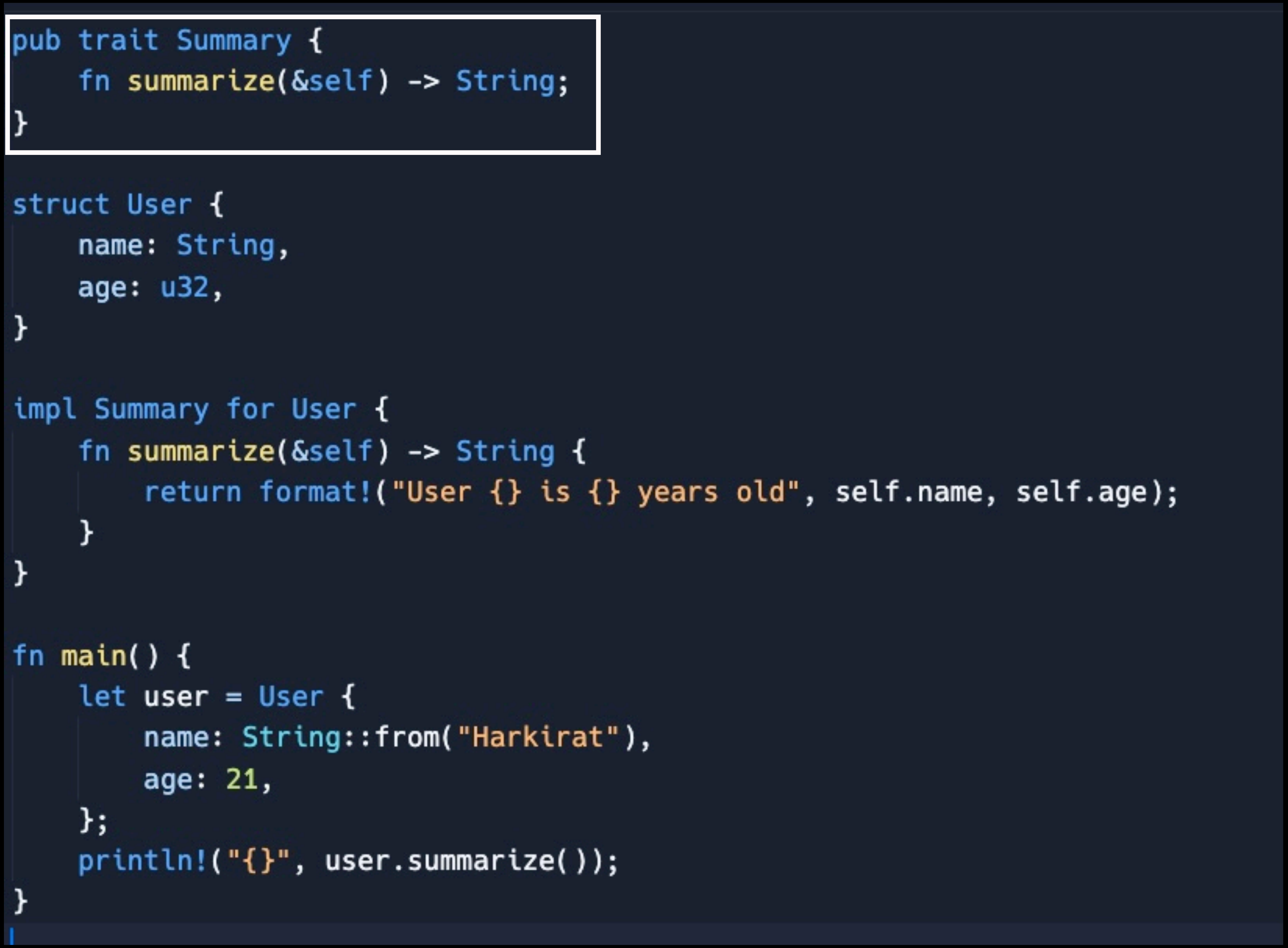
Traits



```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {
    fn summarize(&self) -> String {
        return format!("User {} is {} years old", self.name, self.age);
    }
}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    println!("{}", user.summarize());
}
```

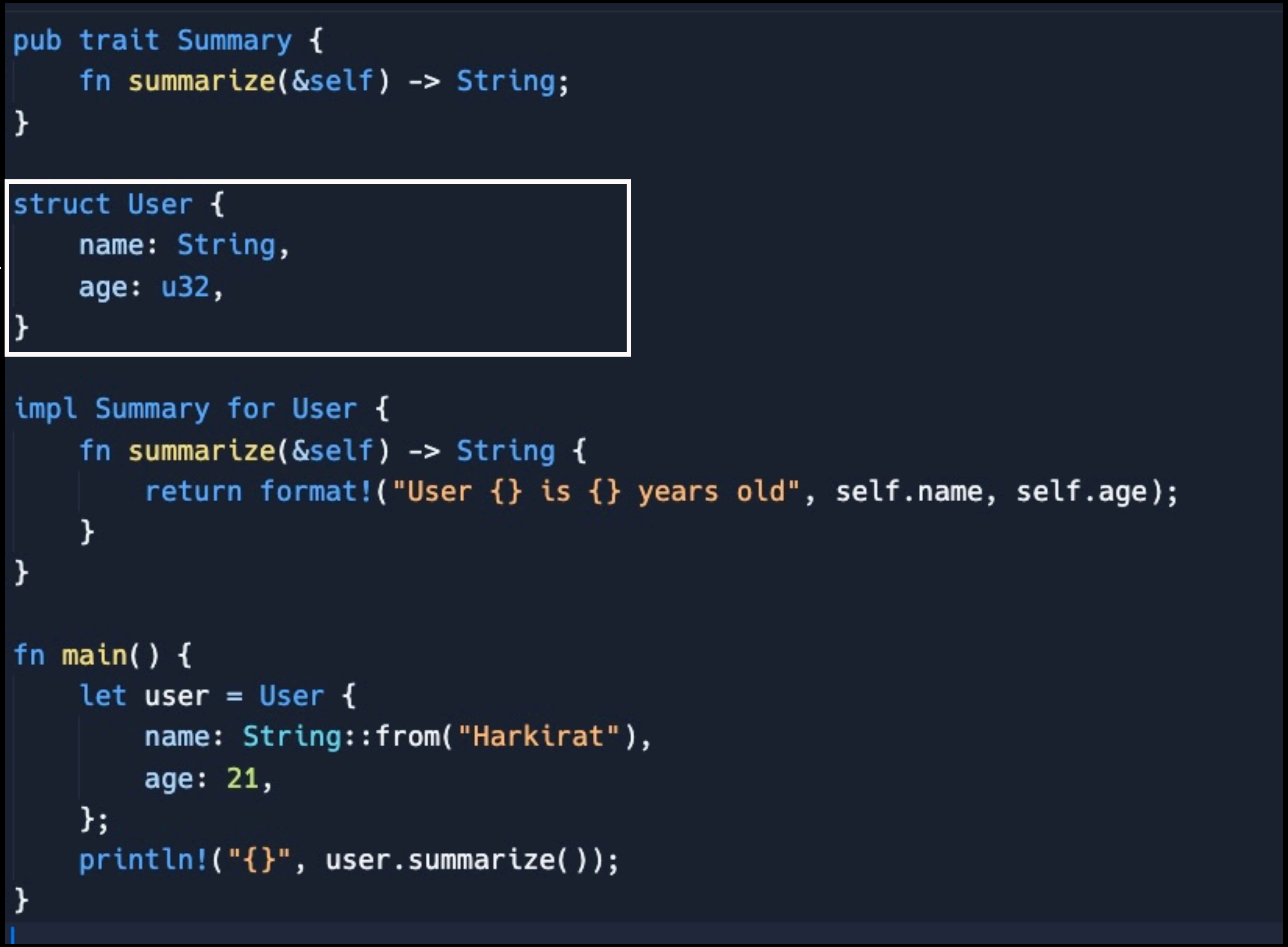
Traits

Defining the trait



```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {
    fn summarize(&self) -> String {
        return format!("User {} is {} years old", self.name, self.age);
    }
}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    println!("{}", user.summarize());
}
```

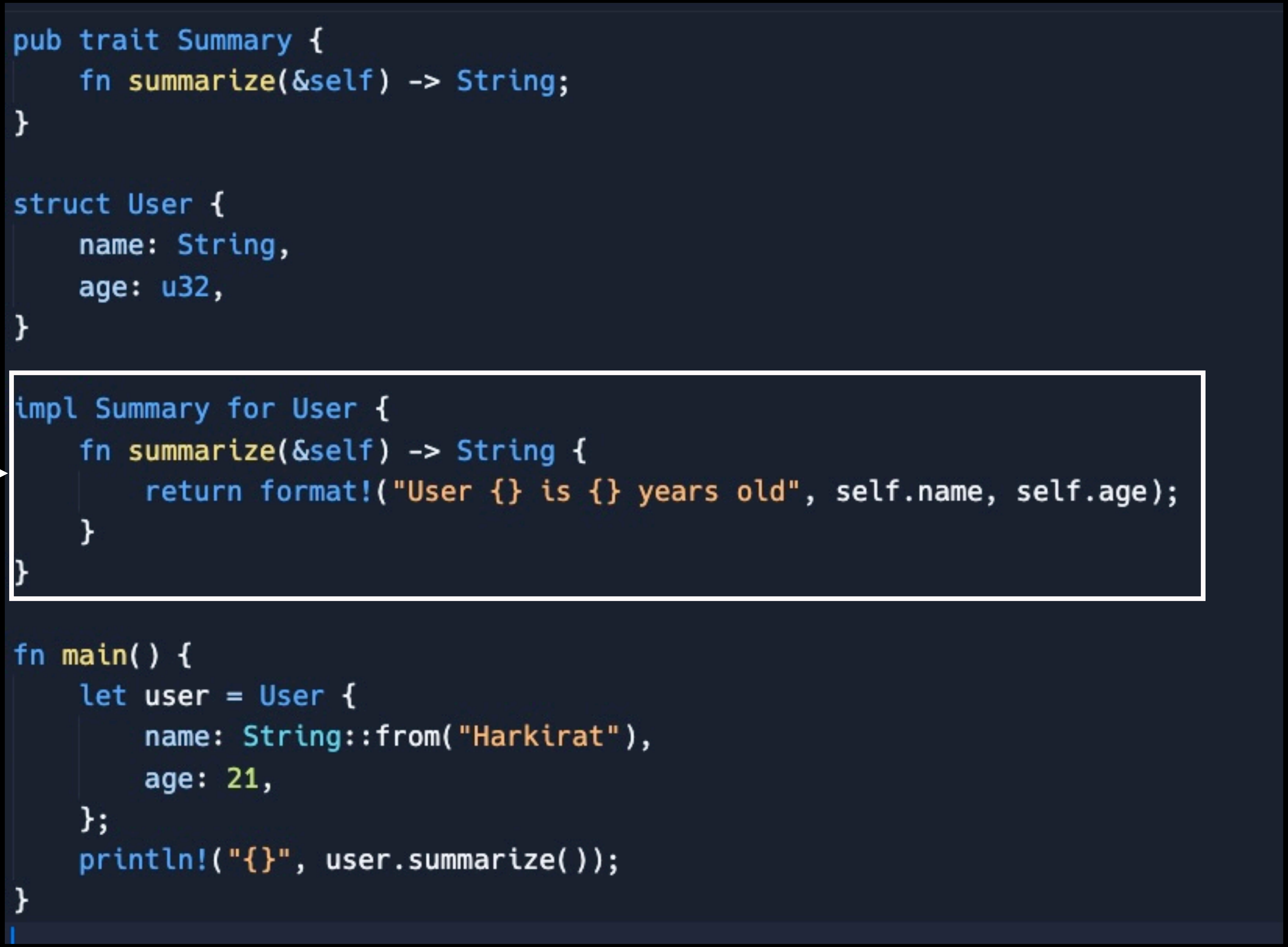
Traits



```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {
    fn summarize(&self) -> String {
        return format!("User {} is {} years old", self.name, self.age);
    }
}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    println!("{}", user.summarize());
}
```

Defining a struct

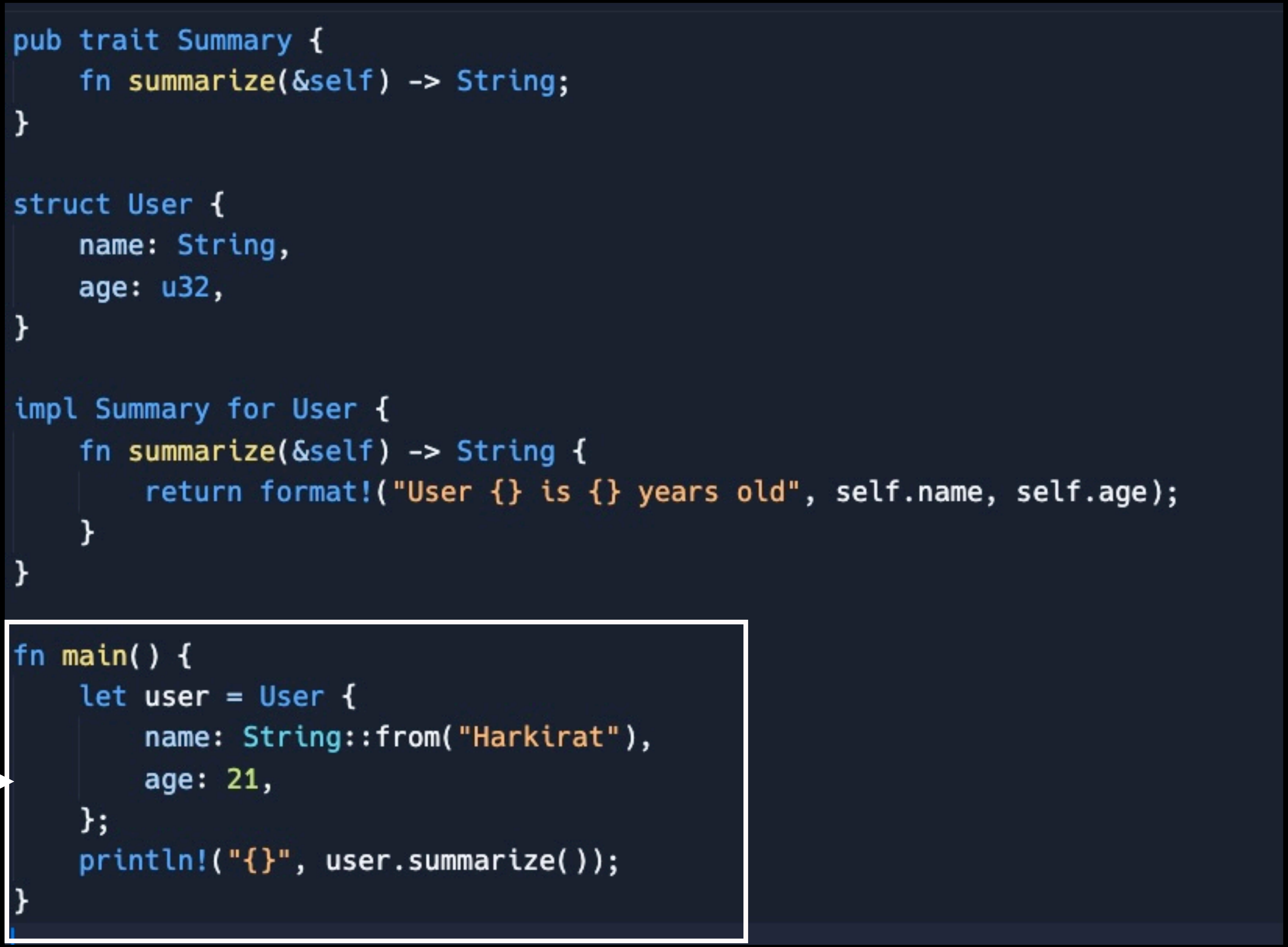
Traits



```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {
    fn summarize(&self) -> String {
        return format!("User {} is {} years old", self.name, self.age);
    }
}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    println!("{}", user.summarize());
}
```

Implementing a Trait on the struct

Traits

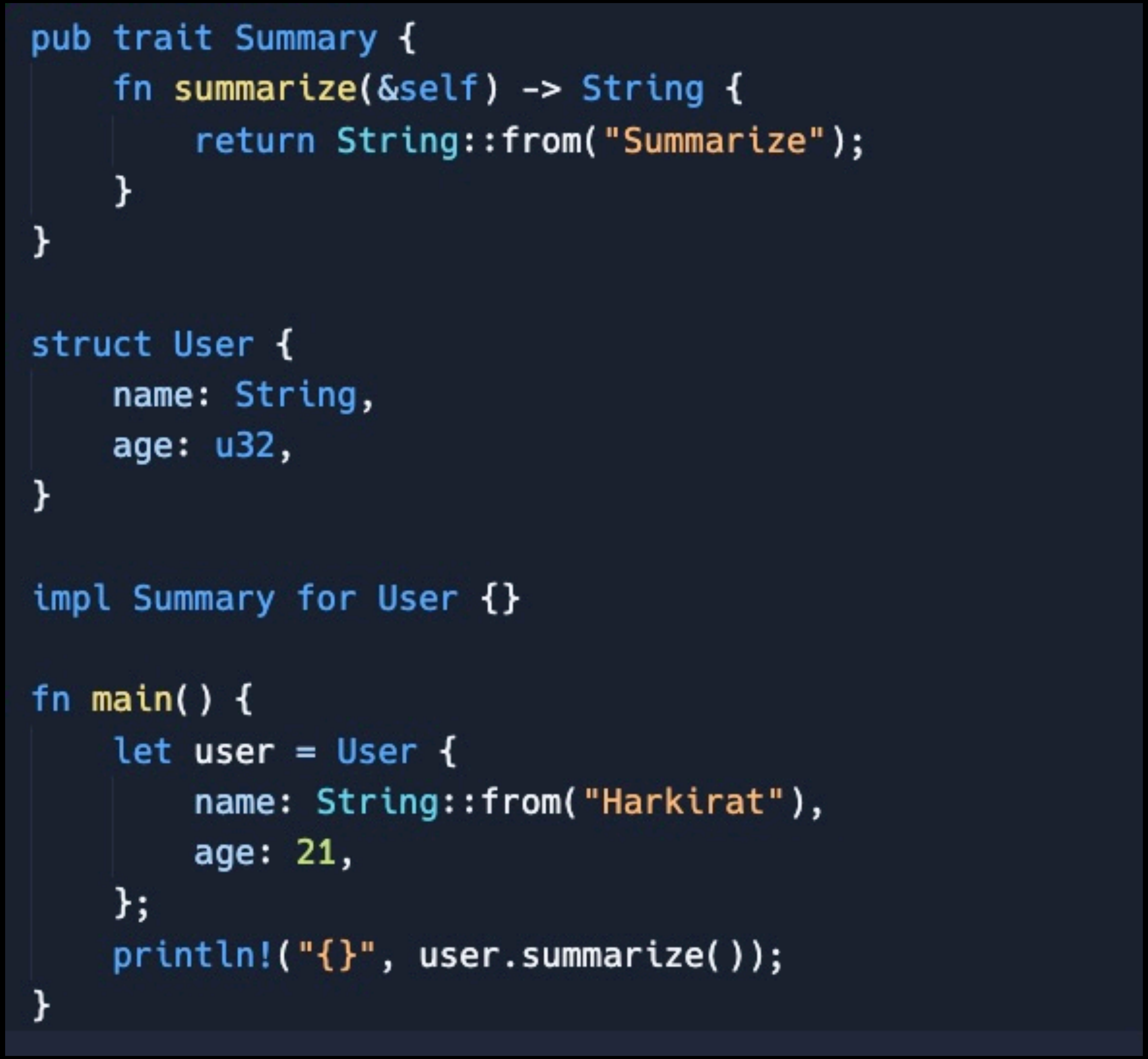


```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {
    fn summarize(&self) -> String {
        return format!("User {} is {} years old", self.name, self.age);
    }
}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    println!("{}", user.summarize());
}
```

Defining the struct and using its function

Traits

Default implementations

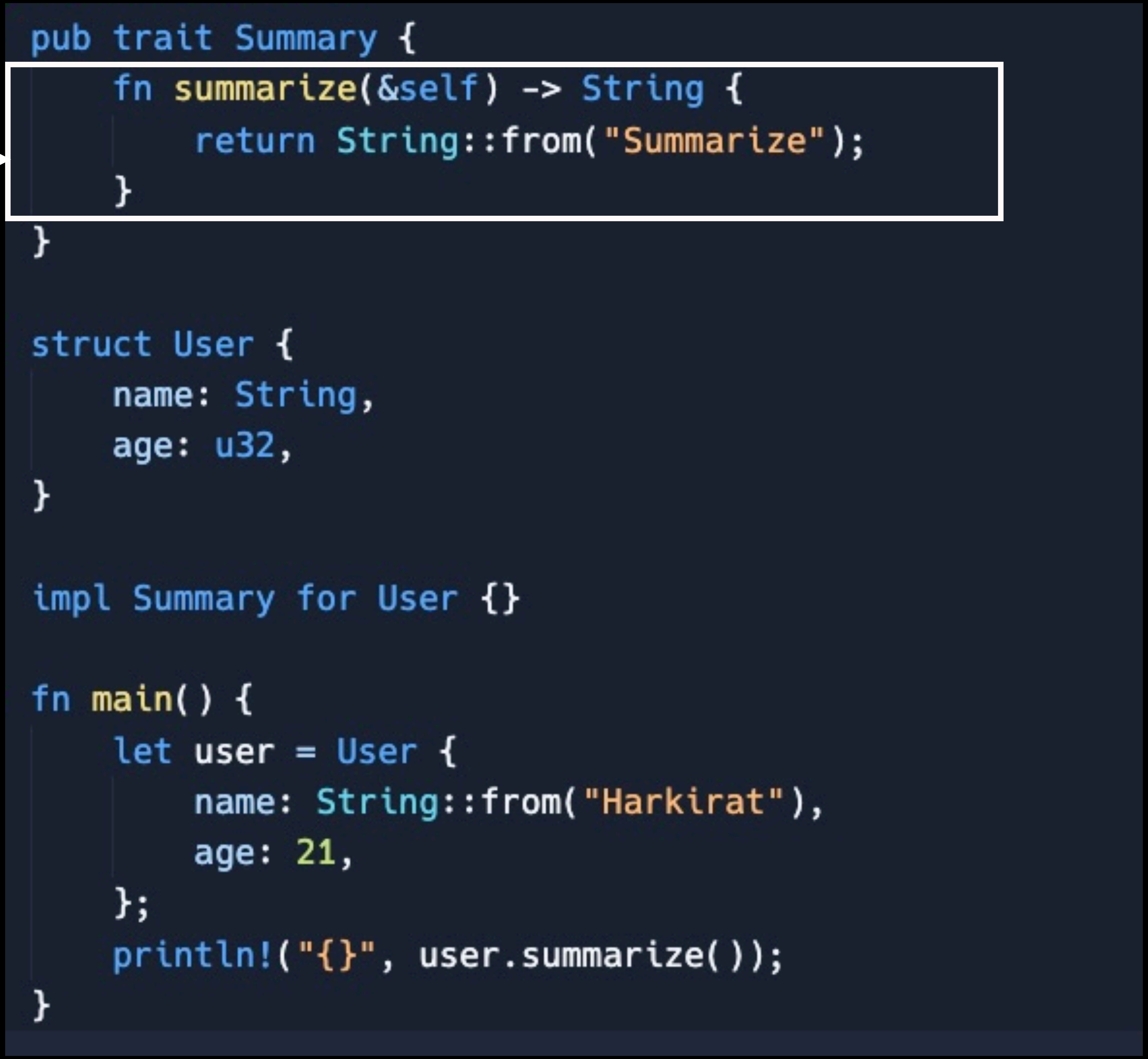


```rust
pub trait Summary {
    fn summarize(&self) -> String {
        return String::from("Summarize");
    }
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    println!("{}", user.summarize());
}
```

Traits

Default implementations

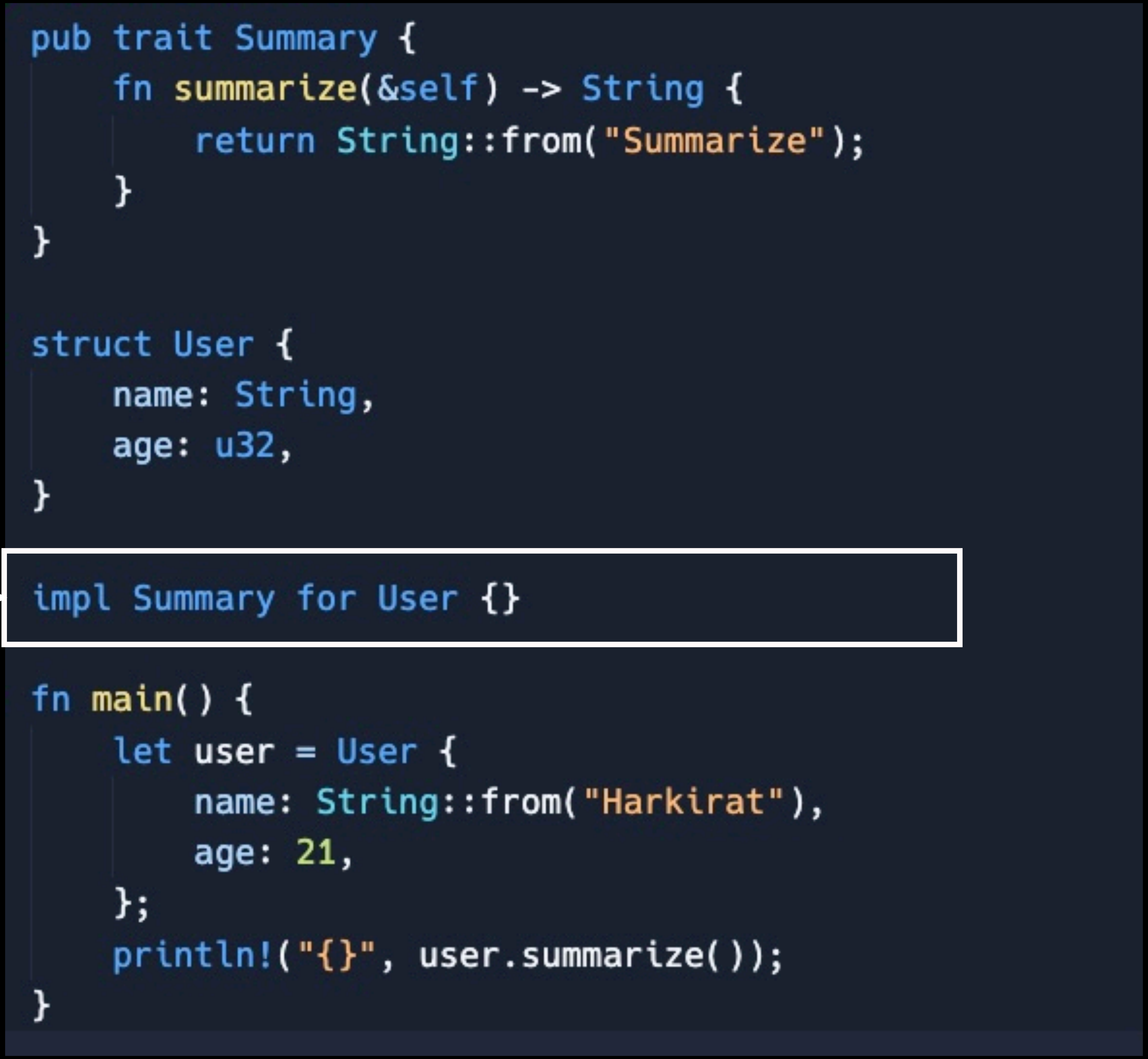
Defining the
`default implementation`



```rust
pub trait Summary {
    fn summarize(&self) -> String {
        return String::from("Summarize");
    }
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    println!("{}", user.summarize());
}
```

Traits

Default implementations

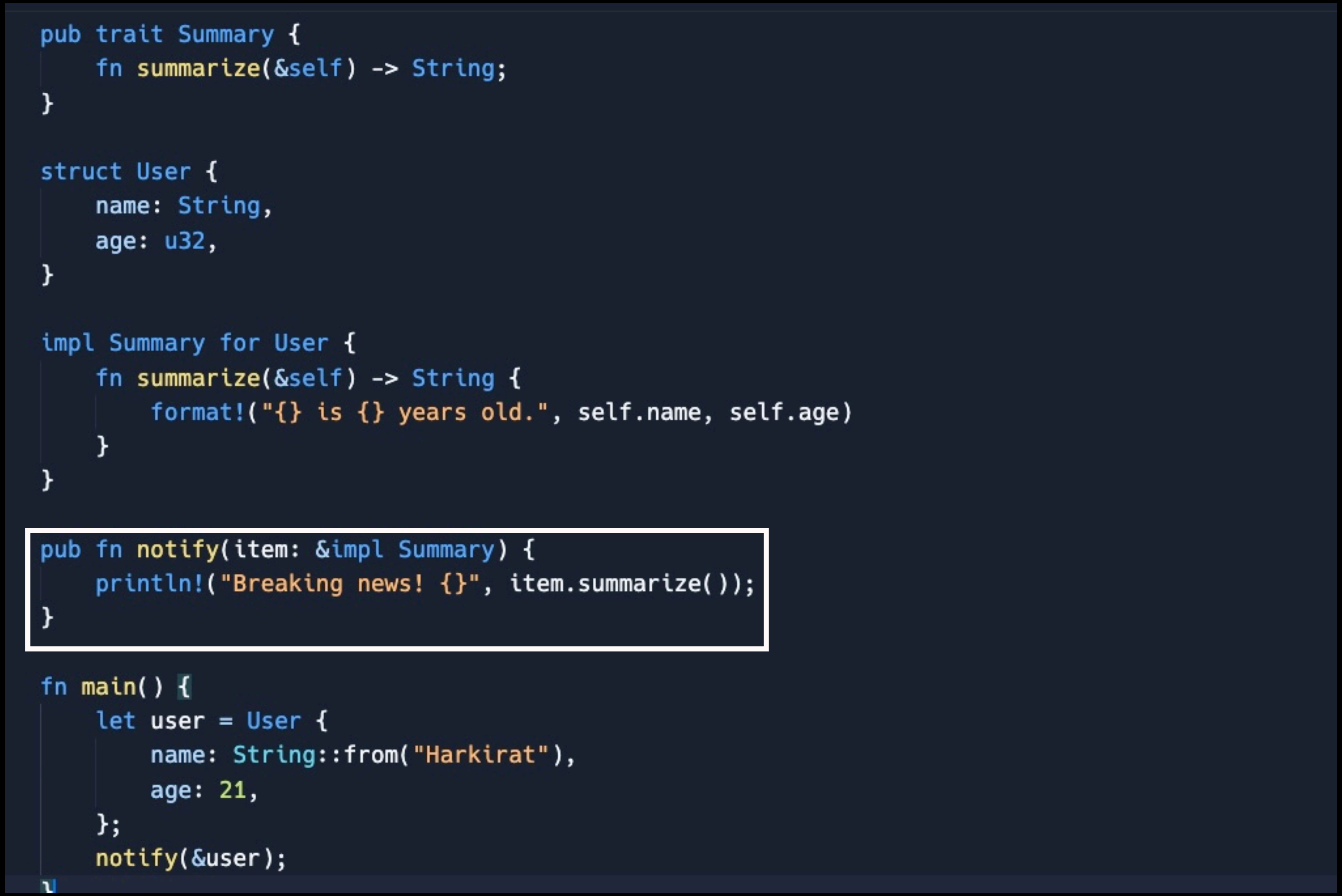


```rust
pub trait Summary {
    fn summarize(&self) -> String {
        return String::from("Summarize");
    }
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    println!("{}", user.summarize());
}
```

Doesn’t error out →

Traits

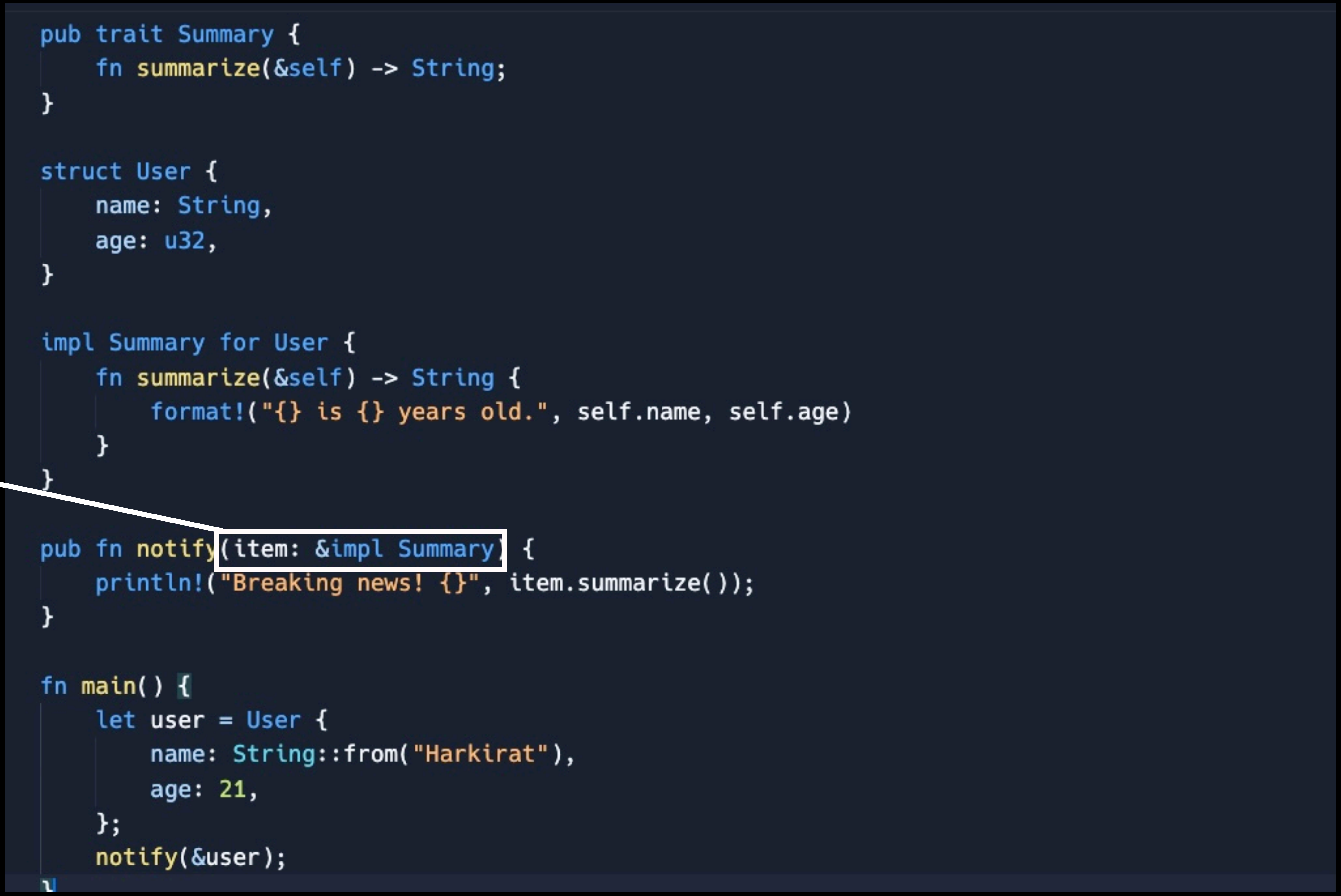
Traits as parameters



```
pub trait Summary {
    fn summarize(&self) -> String;
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {
    fn summarize(&self) -> String {
        format!("{} is {} years old.", self.name, self.age)
    }
}

pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    notify(&user);
}
```

Traits

Traits as parameters

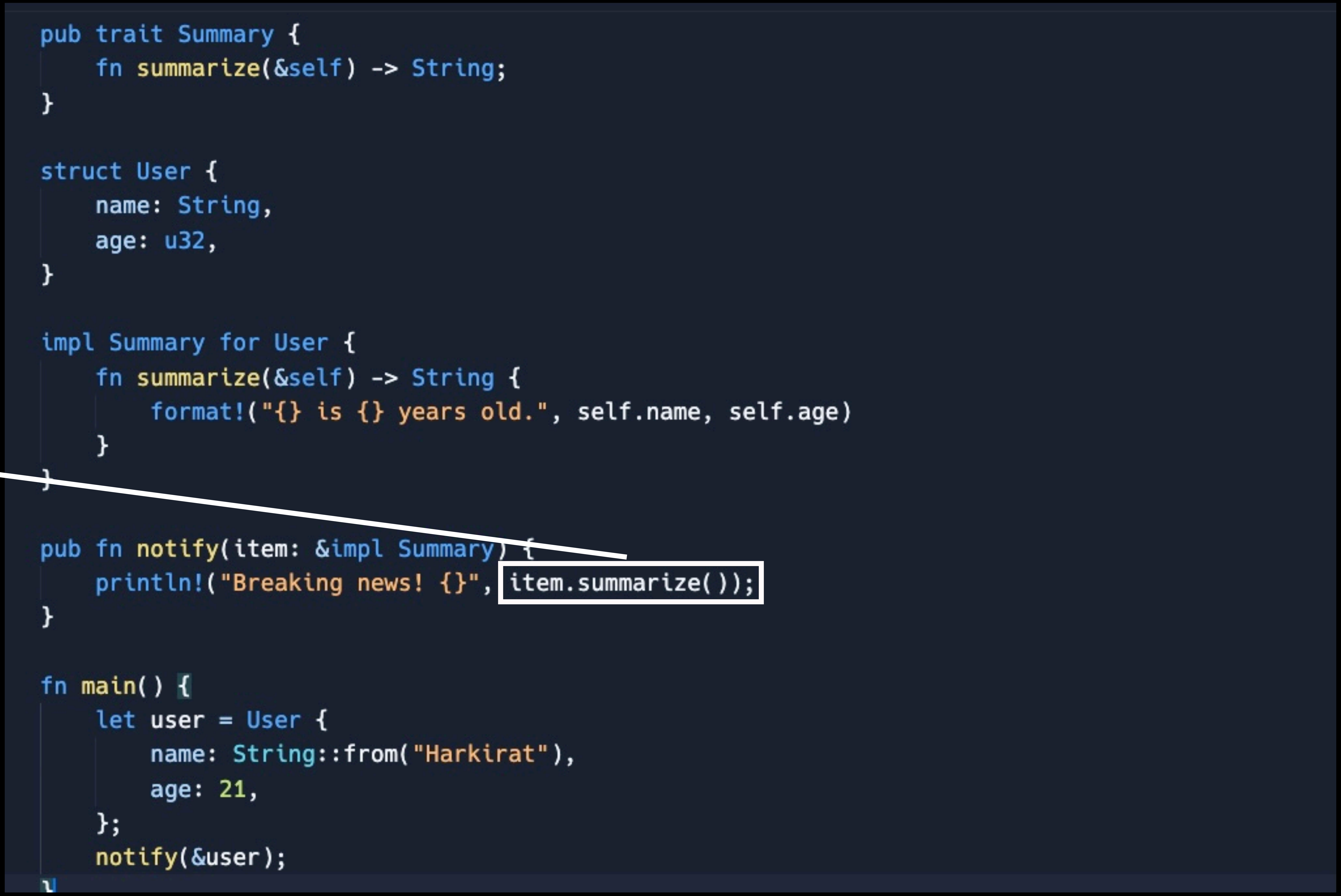


```
pub trait Summary {
    fn summarize(&self) -> String;
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {
    fn summarize(&self) -> String {
        format!("{} is {} years old.", self.name, self.age)
    }
}

pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    notify(&user);
}
```

Trait as an argument

Traits

Traits as parameters

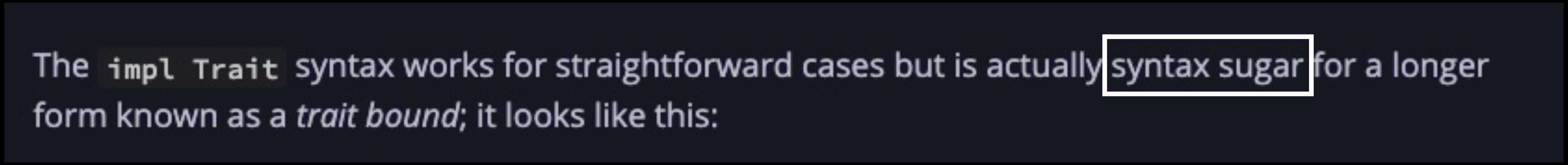


```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

struct User {
    name: String,
    age: u32,
}

impl Summary for User {
    fn summarize(&self) -> String {
        format!("{} is {} years old.", self.name, self.age)
    }
}

pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

fn main() {
    let user = User {
        name: String::from("Harkirat"),
        age: 21,
    };
    notify(&user);
}
```

Can call summarise on it

Traits

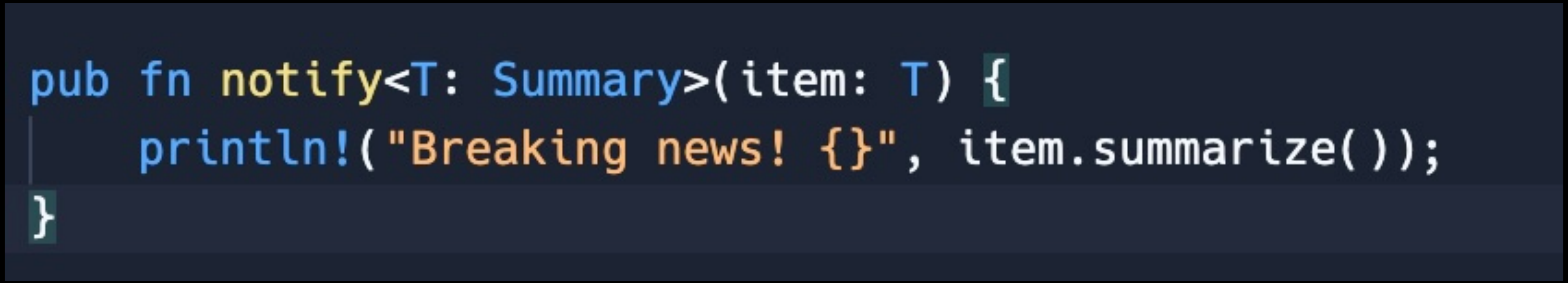
Trait bound syntax



The `impl Trait` syntax works for straightforward cases but is actually syntax sugar for a longer form known as a *trait bound*; it looks like this:

Traits

Trait bound syntax

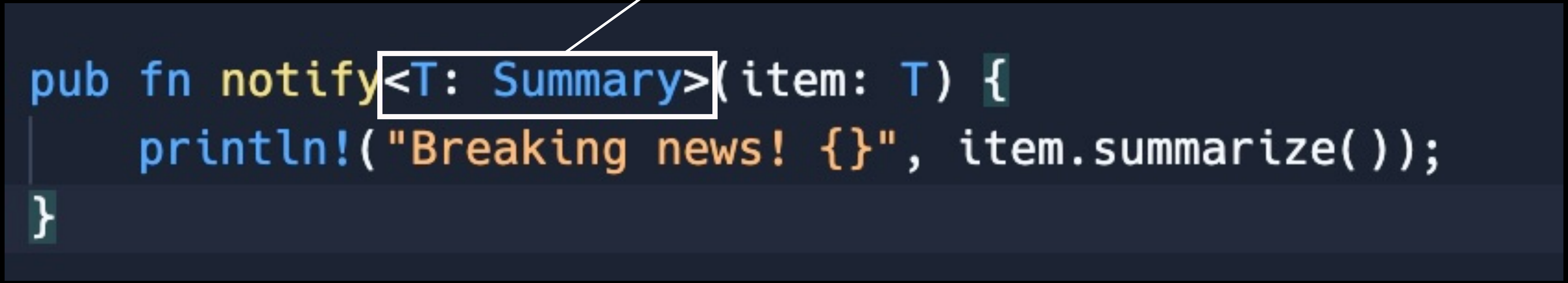


```
pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

Traits

Trait bound syntax

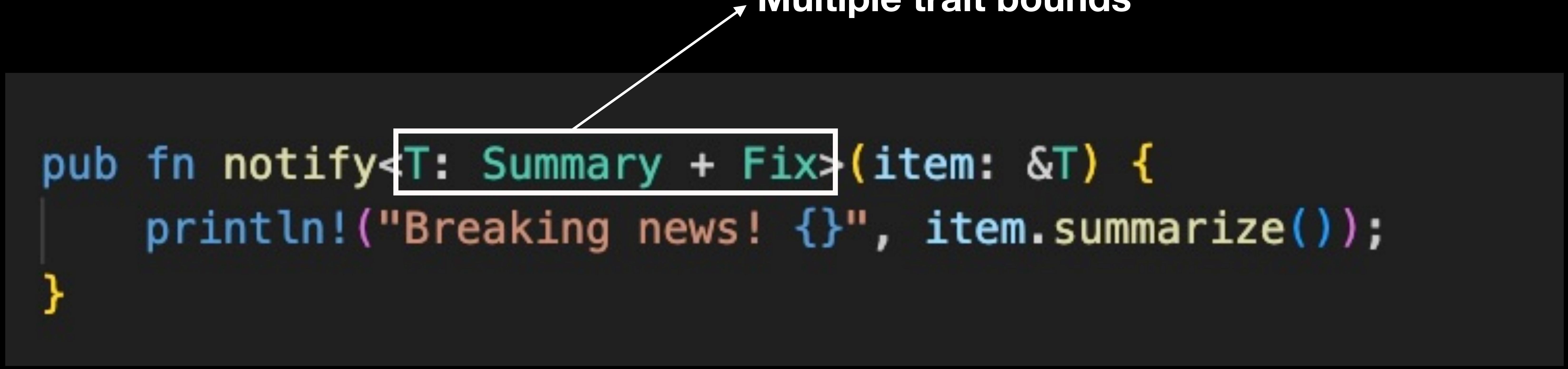
The generic is bound to the trait



```rust
pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

Traits

Multiple Trait bounds



Multiple trait bounds

```
pub fn notify<T: Summary + Fix>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

Lifetimes

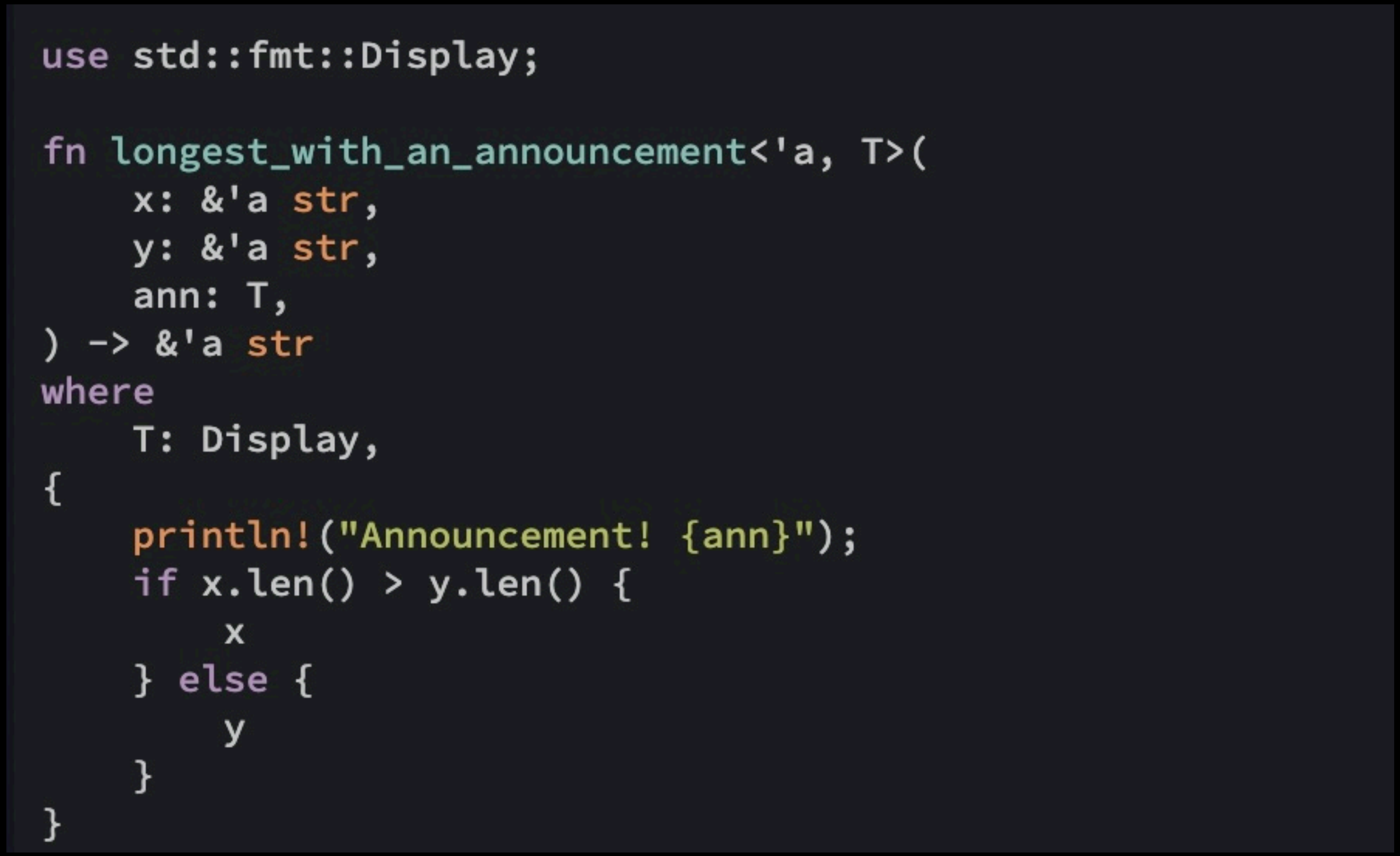
Lifetimes are hard to digest.

Takes a lot of time to understand why they are needed

Lot of times the compiler will help you and guide you in the right direction

Lifetimes

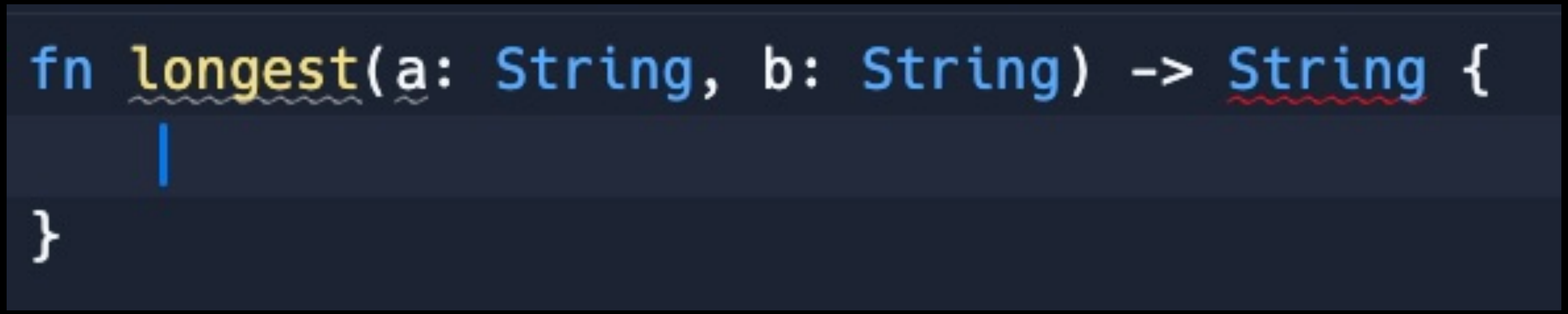
Goal is by the end of this section
You can understand the following syntax



```rust
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {ann}");
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes

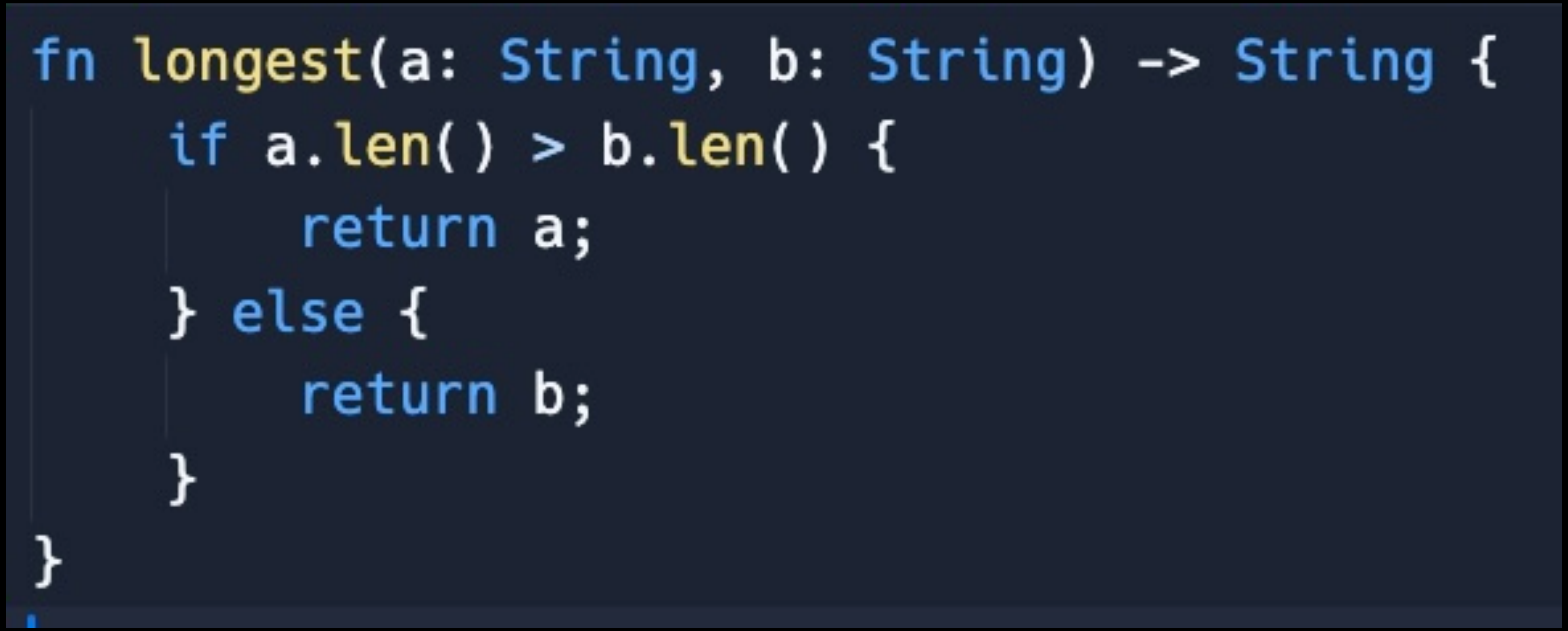
Q - Write a function that takes two strings as an input
And returns the bigger amongst them



```rust
fn longest(a: String, b: String) -> String {
    
}
```

Lifetimes

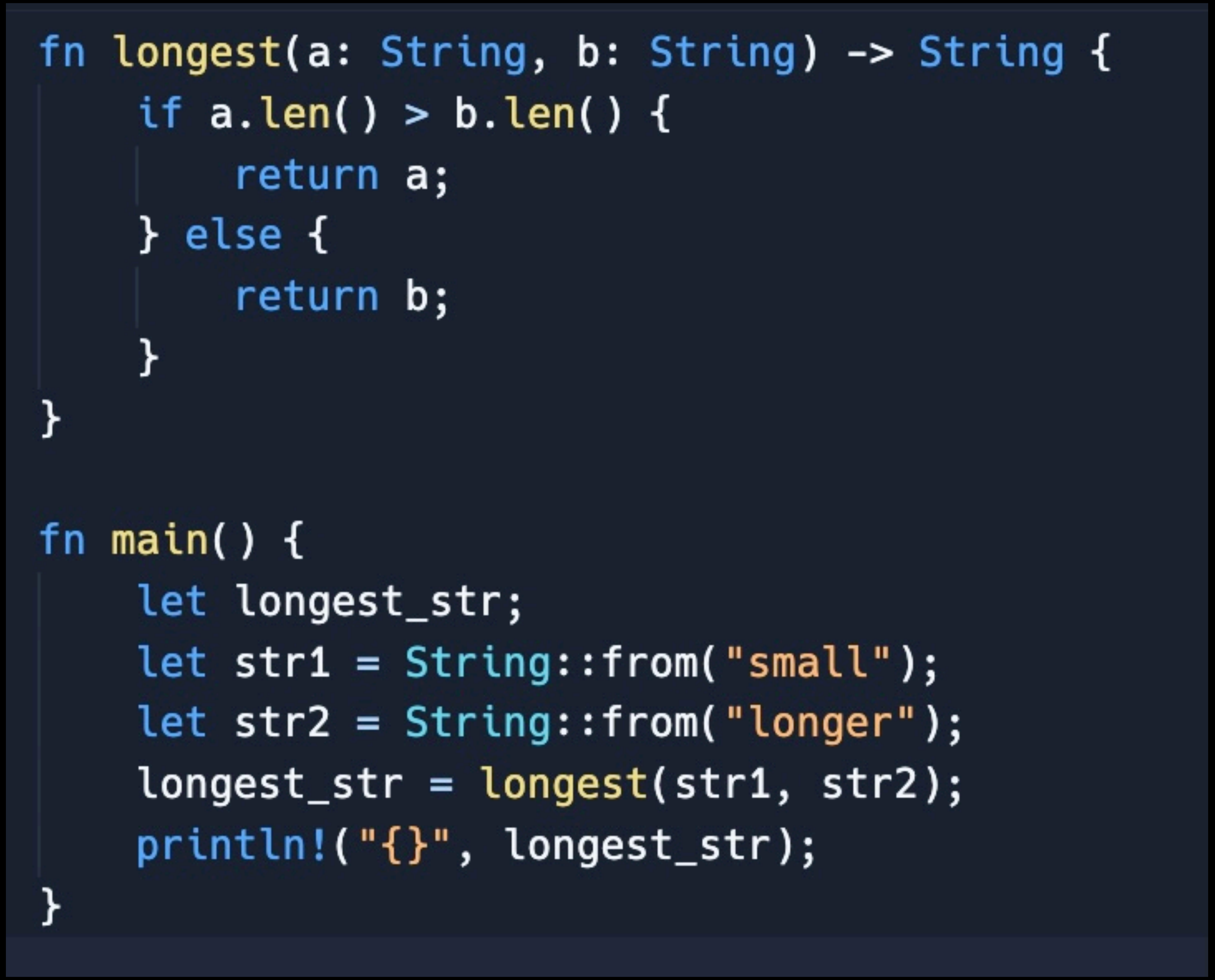
**Q - Write a function that takes two strings as an input
And returns the bigger amongst them**



```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}
```

Lifetimes

Q - Write a function that takes two strings as an input
And returns the bigger amongst them

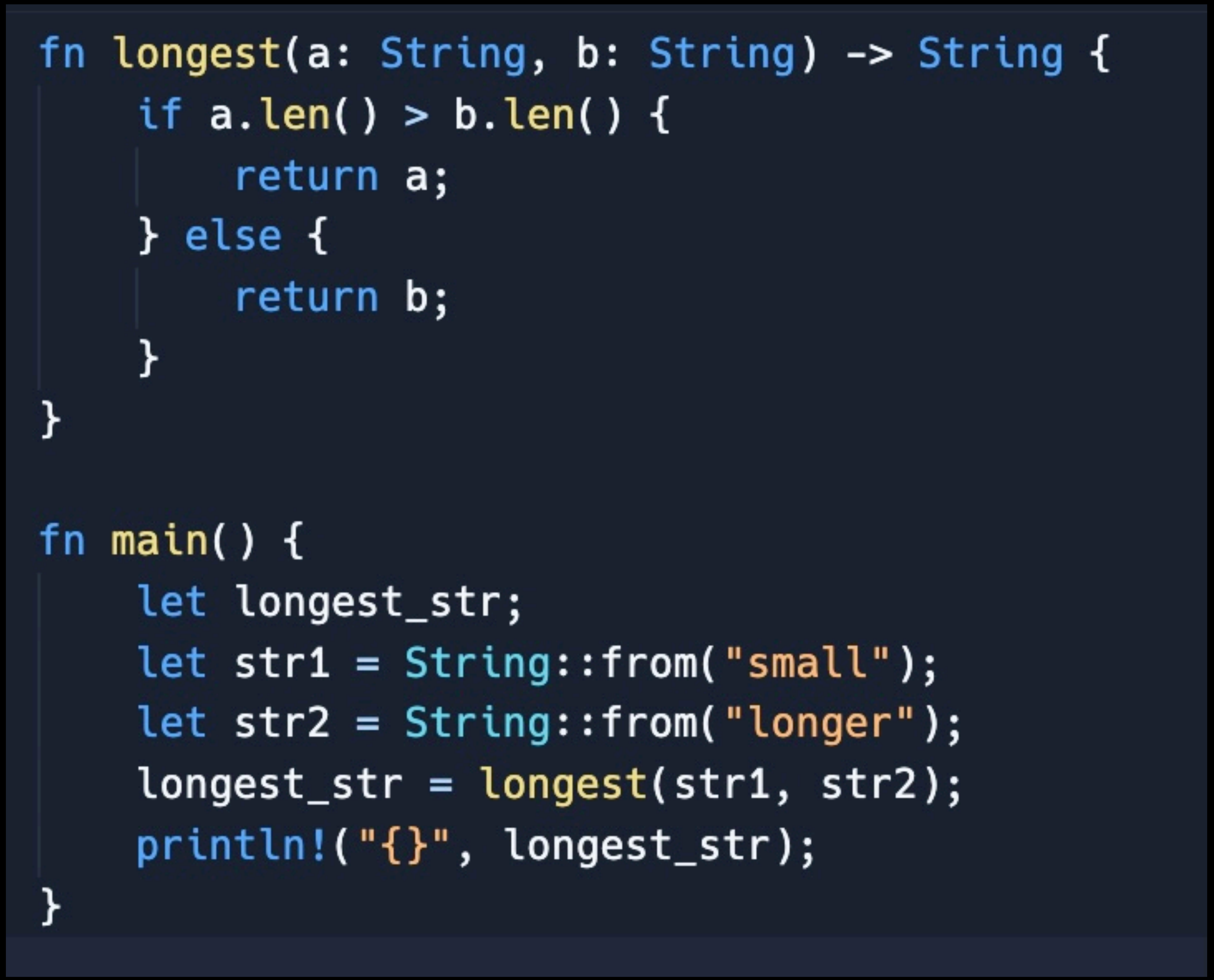


```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(str1, str2);
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two strings as an input
And returns the bigger amongst them

longest_str is defined →

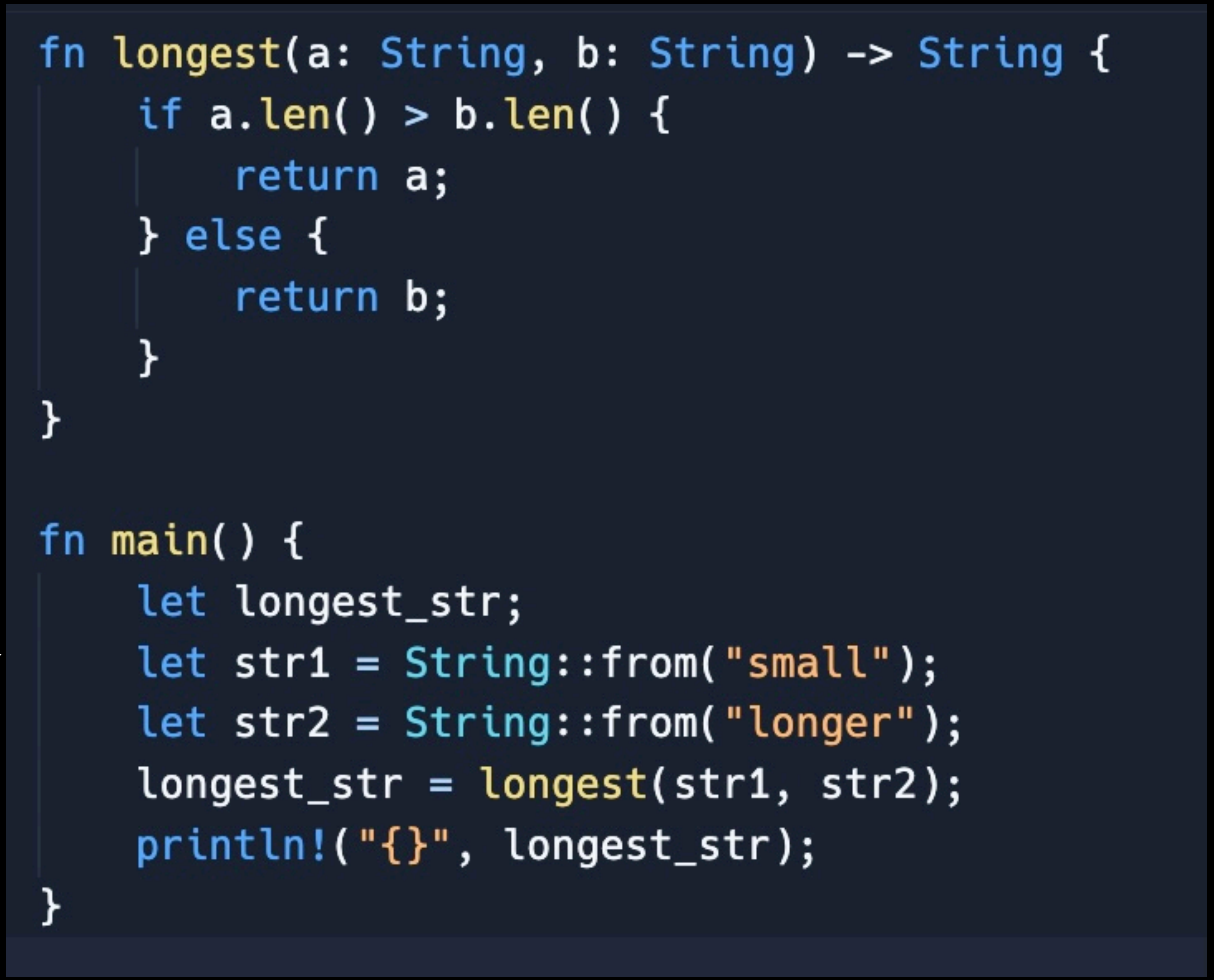


```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(str1, str2);
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two strings as an input
And returns the bigger amongst them

str1 is defined

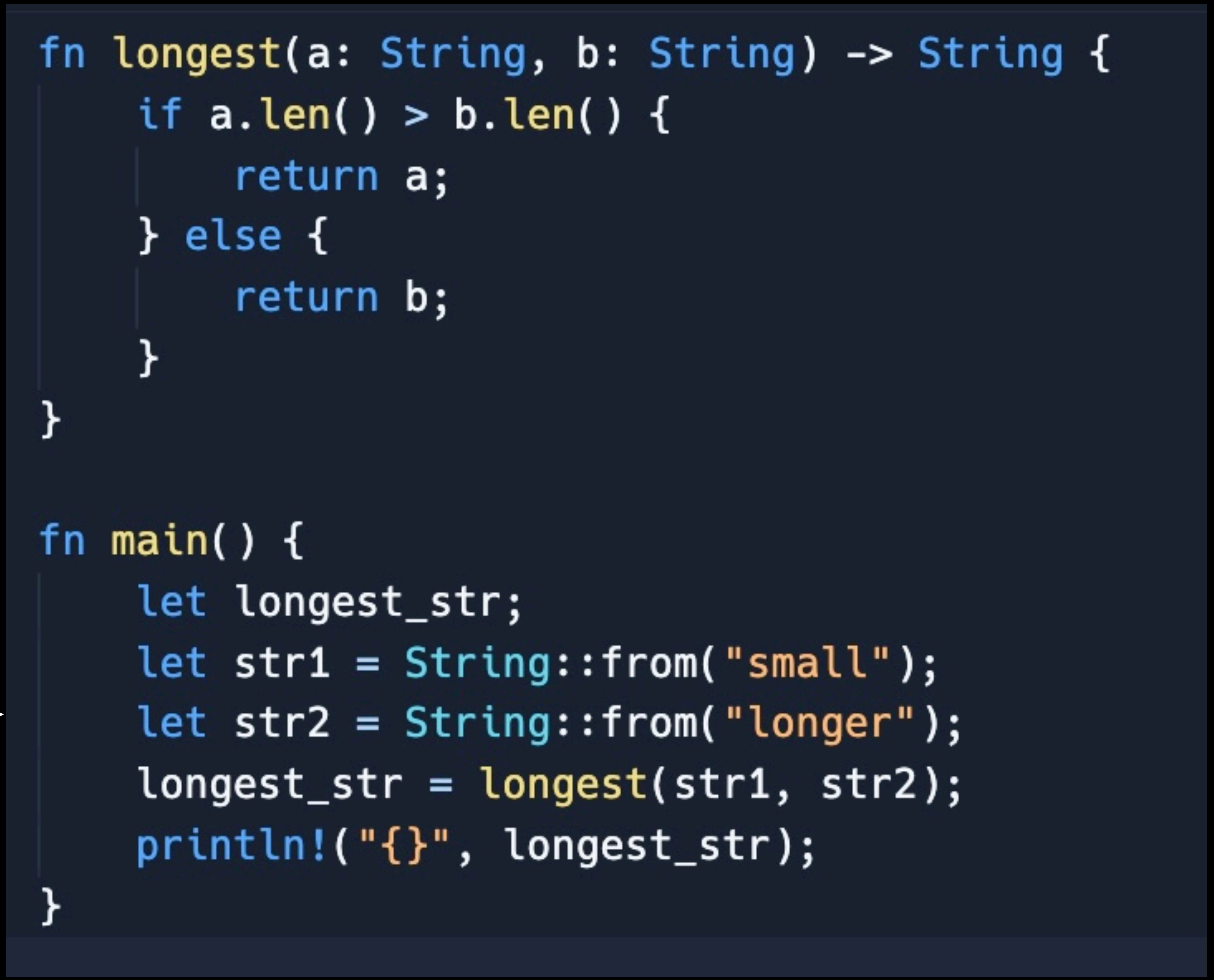


```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(str1, str2);
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two strings as an input
And returns the bigger amongst them

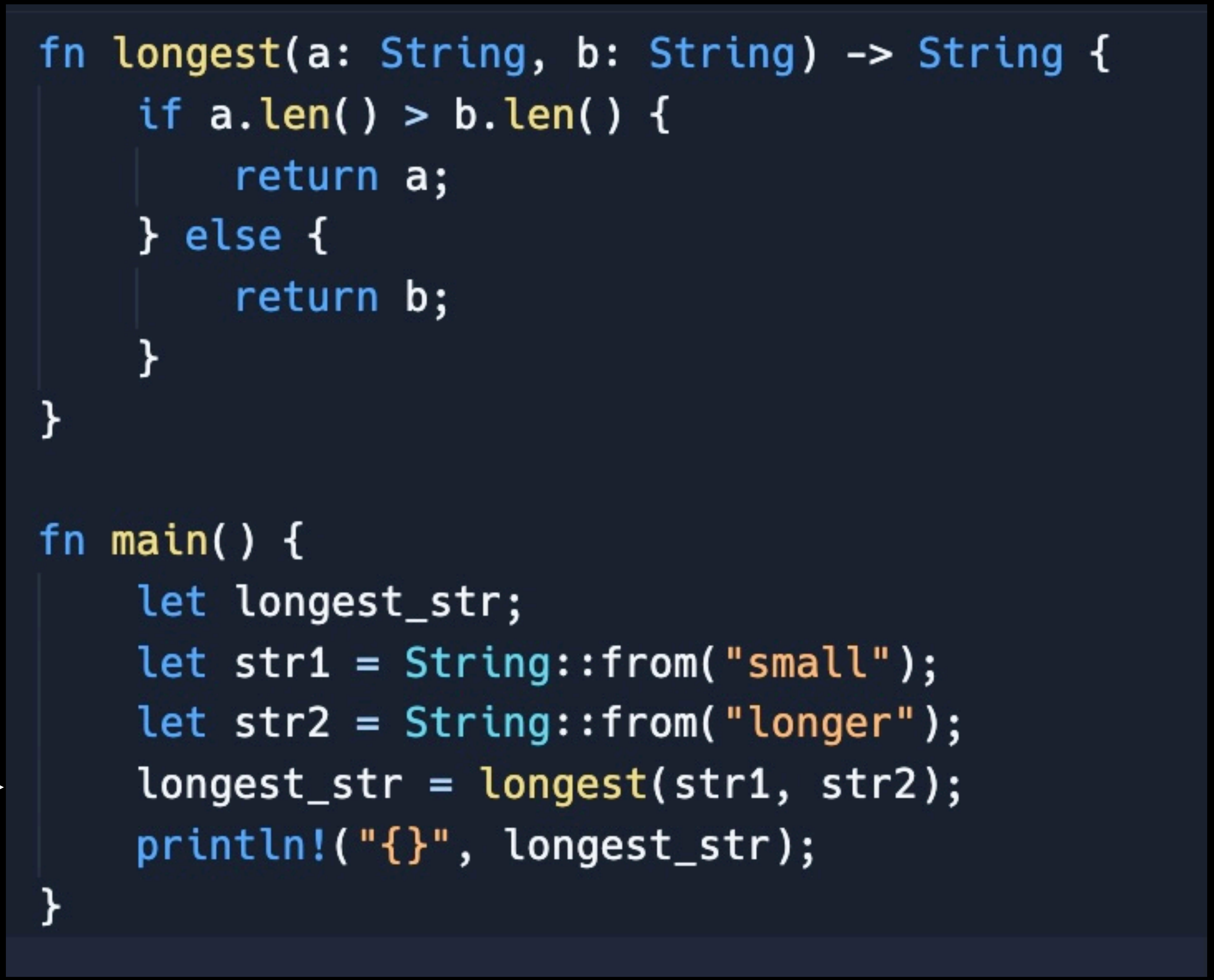
Str2 is defined



```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(str1, str2);
    println!("{}", longest_str);
}
```

Lifetimes

**Q - Write a function that takes two strings as an input
And returns the bigger amongst them**



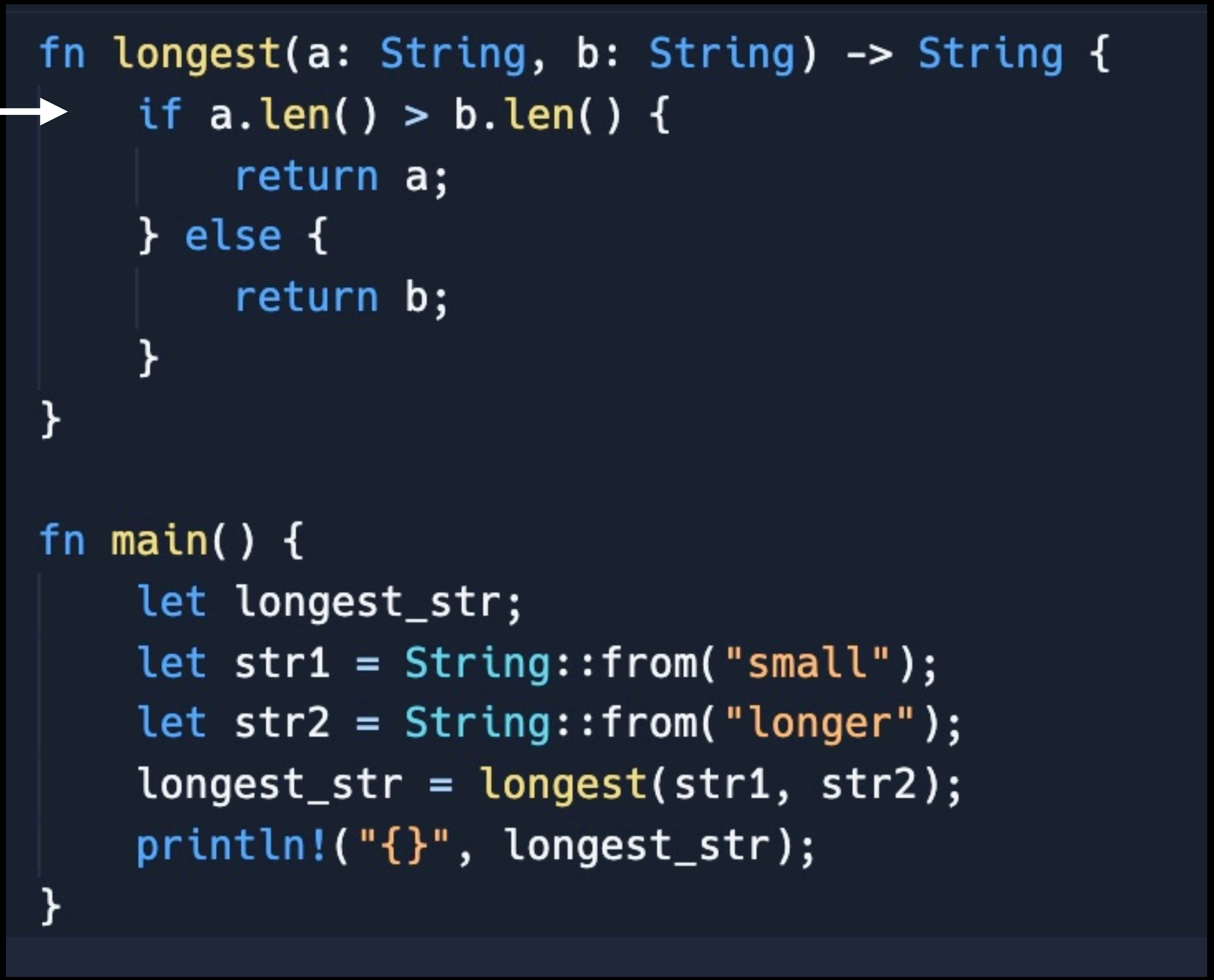
```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(str1, str2);
    println!("{}", longest_str);
}
```

str1 and str2 both get moved →

Lifetimes

Q - Write a function that takes two strings as an input
And returns the bigger amongst them

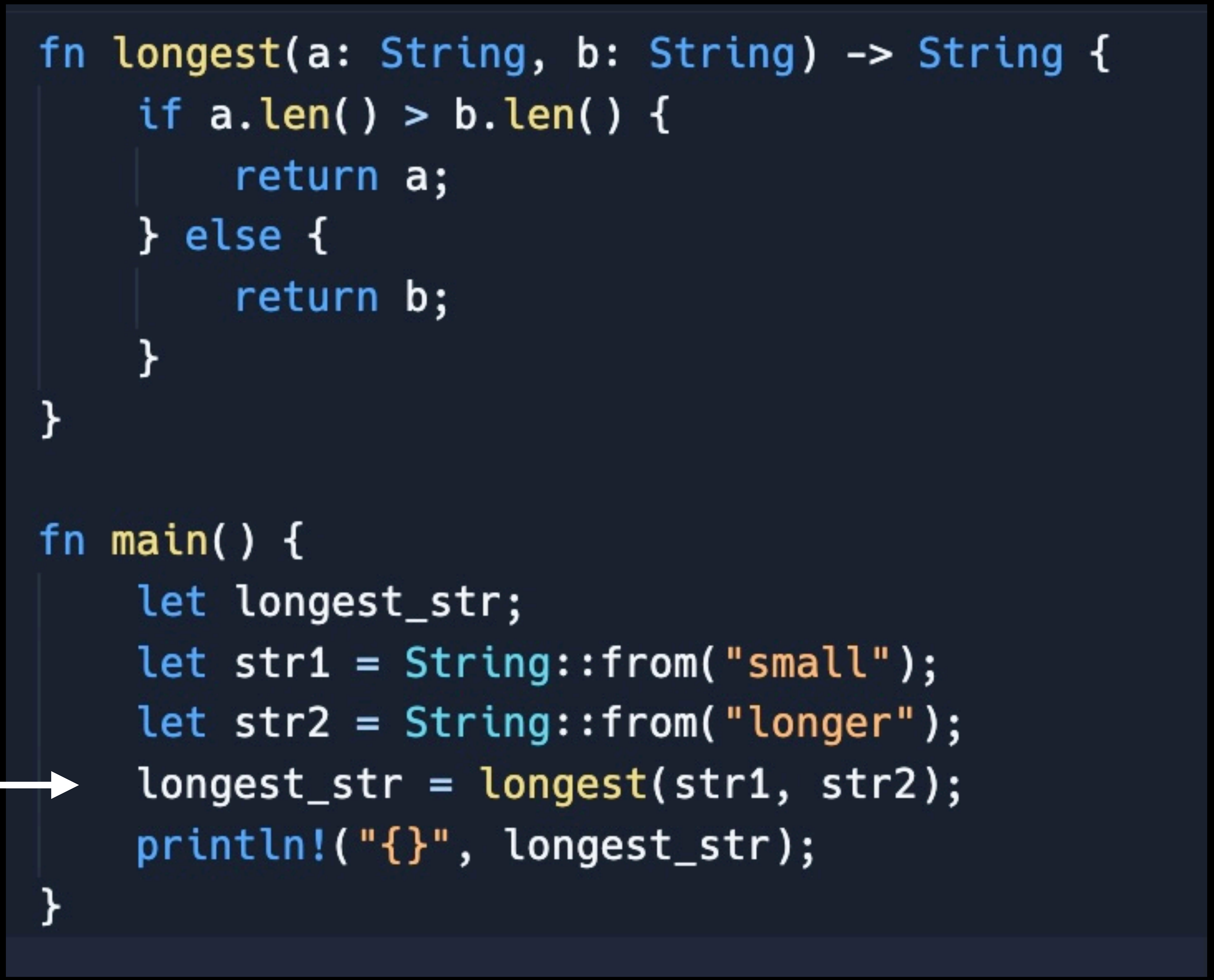
Check



```
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(str1, str2);
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two strings as an input
And returns the bigger amongst them

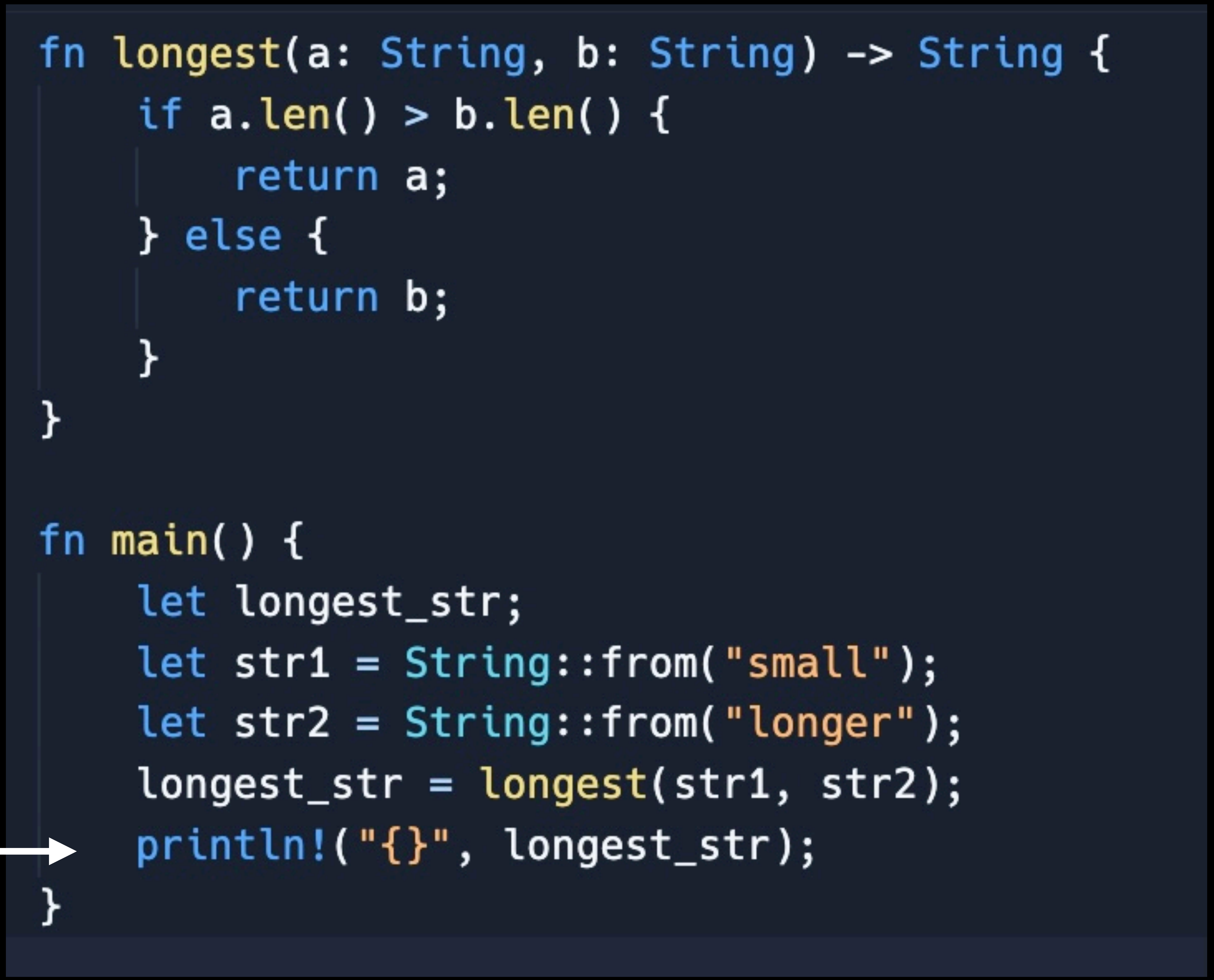


```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(str1, str2);
    println!("{}", longest_str);
}
```

str2 moves to longest_str →

Lifetimes

Q - Write a function that takes two strings as an input
And returns the bigger amongst them



```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(str1, str2);
    println!("{}", longest_str);
}
```

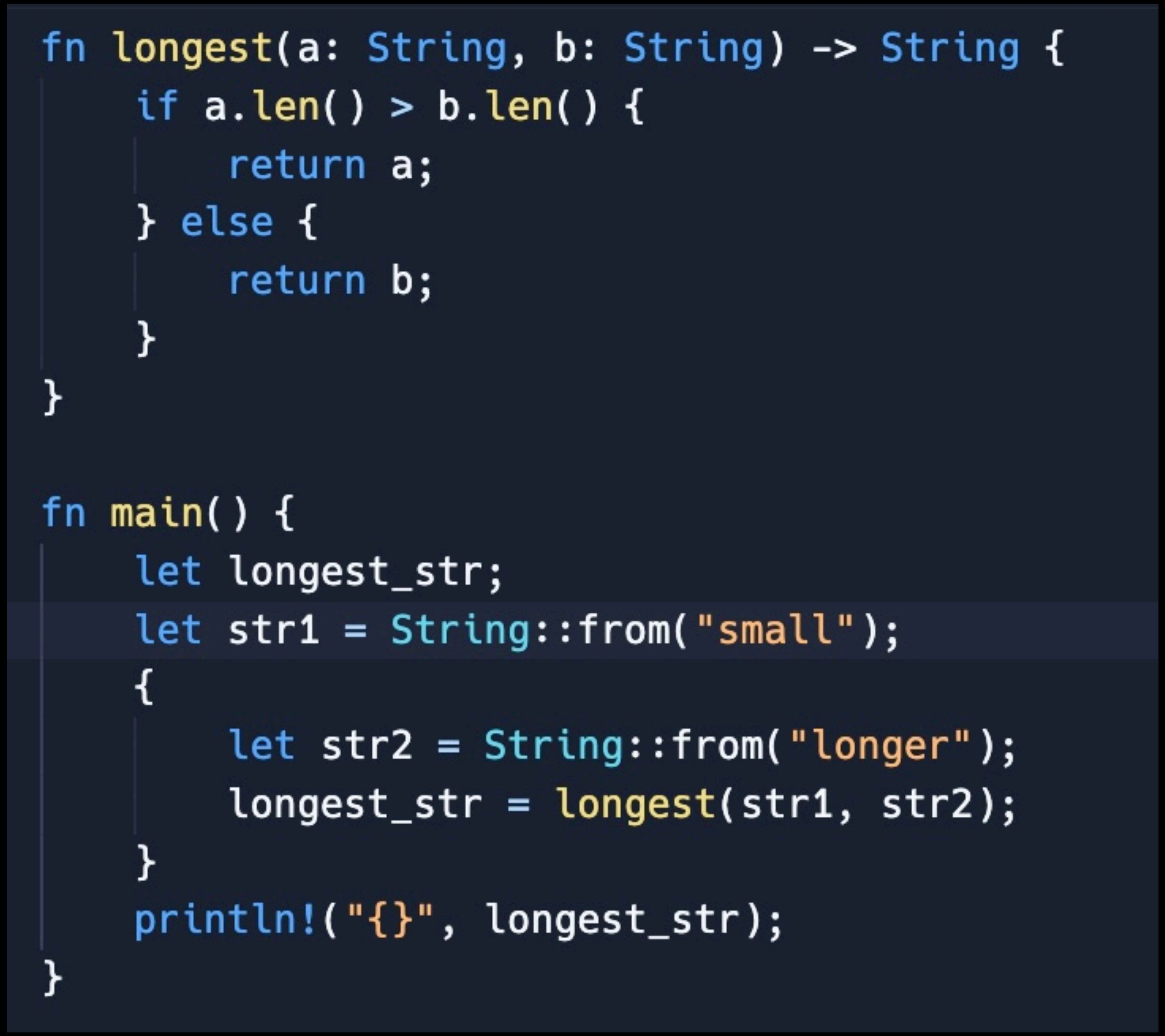
longest_str is valid →

Lifetimes

Now lets change the syntax a bit

Lifetimes

Do you think this is valid syntax?

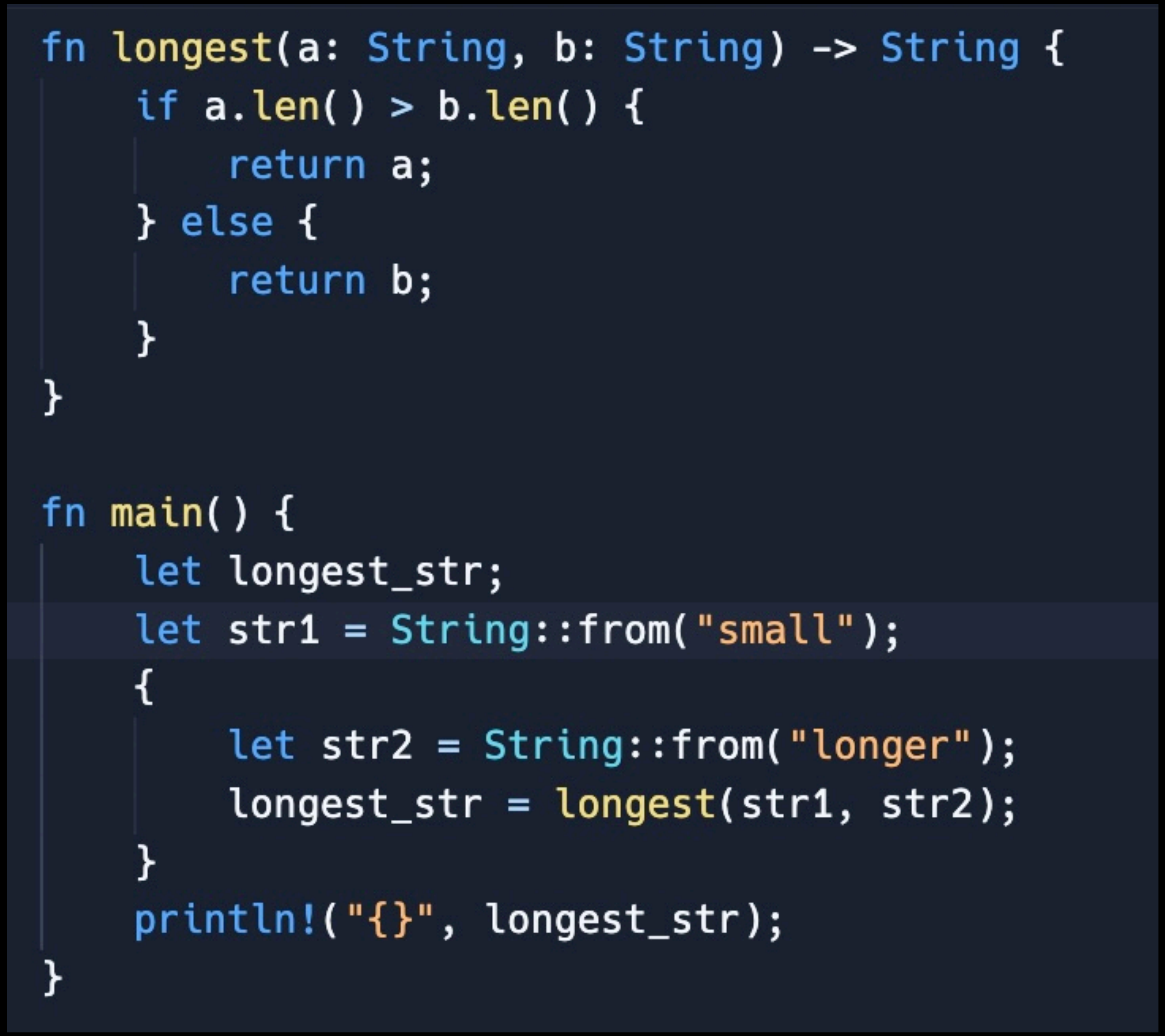


```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

Do you think this is valid syntax?

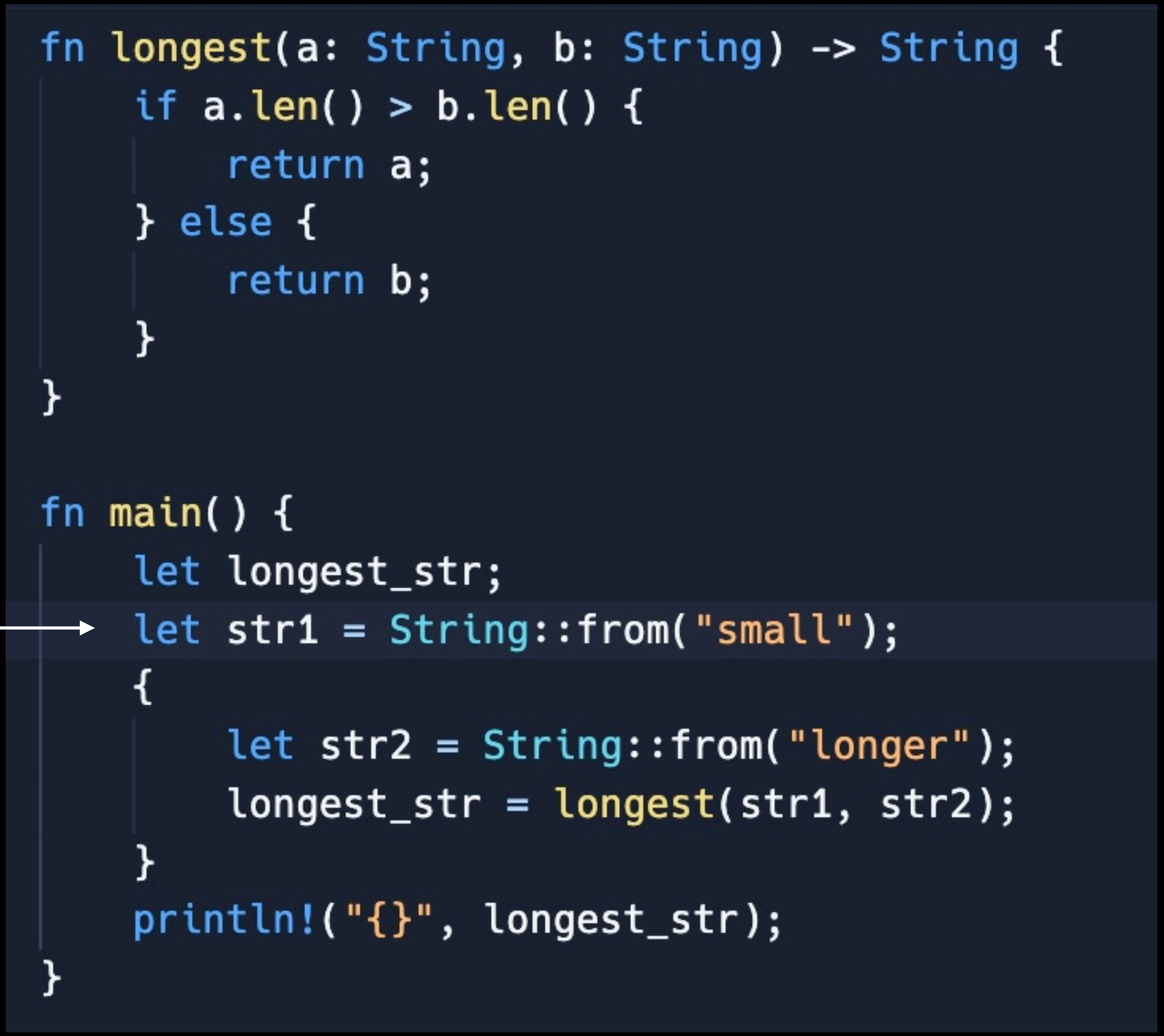
Yes



```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

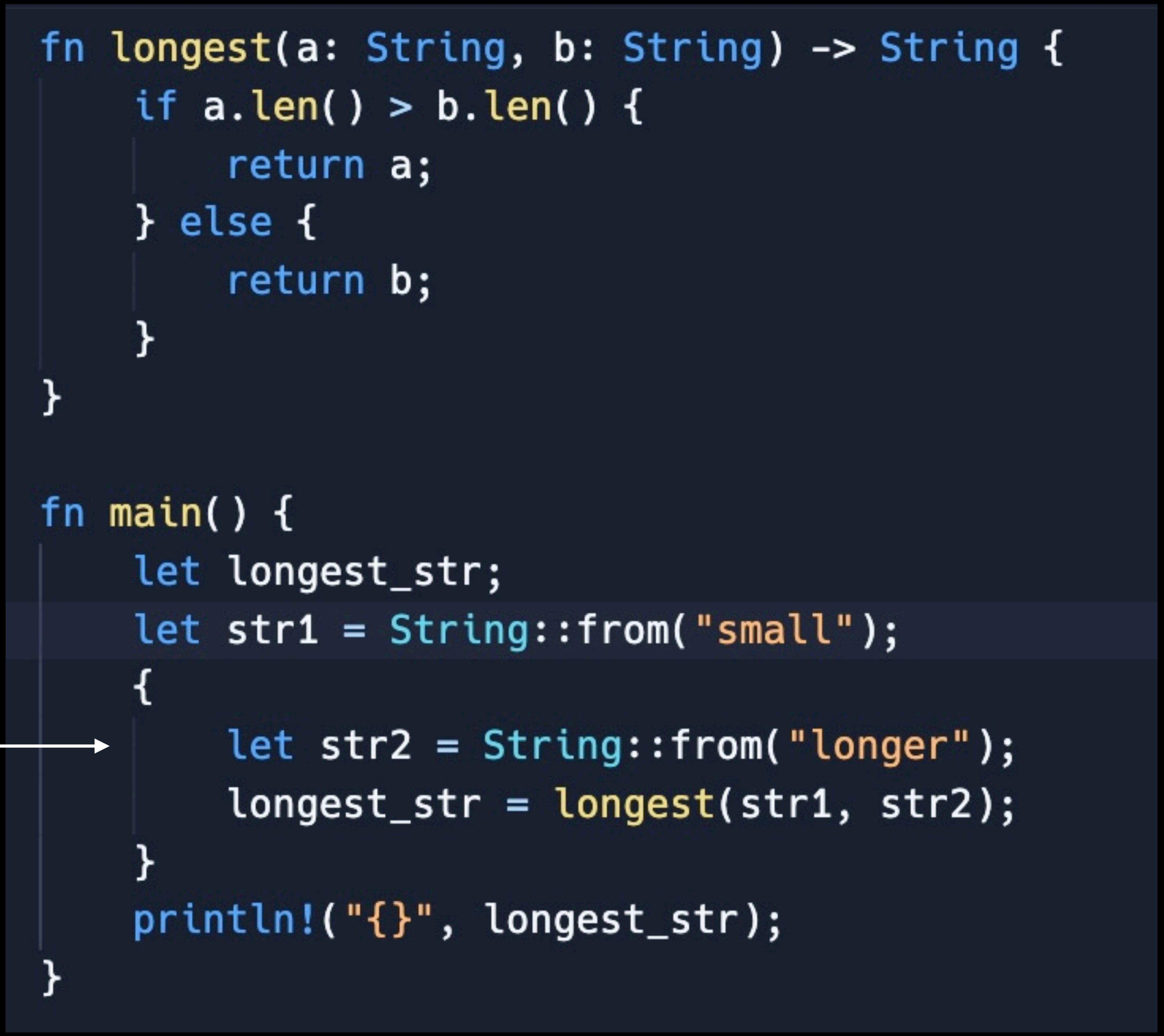
str1 owns the string



```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

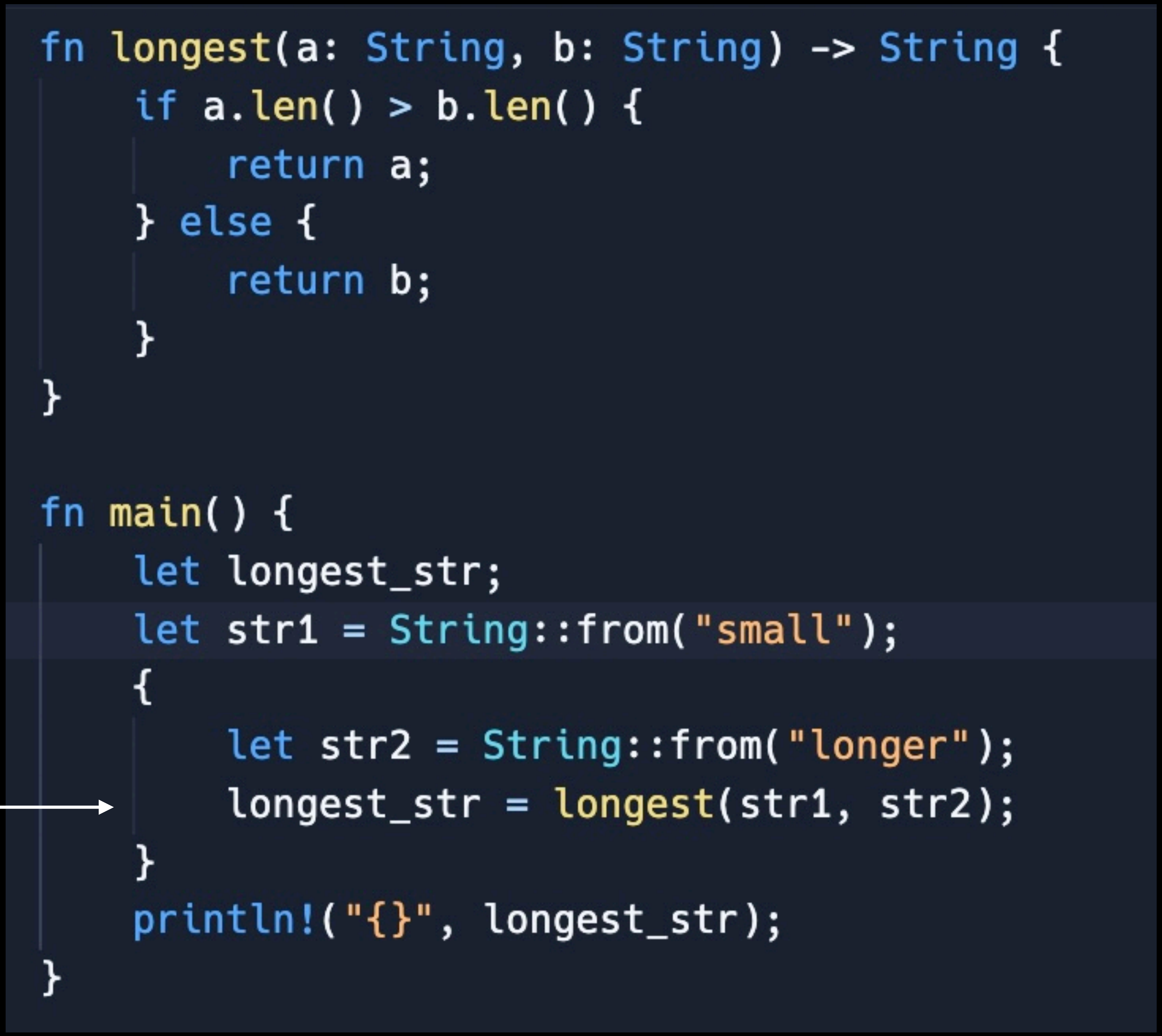
Lifetimes

str2 owns the string →



```
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

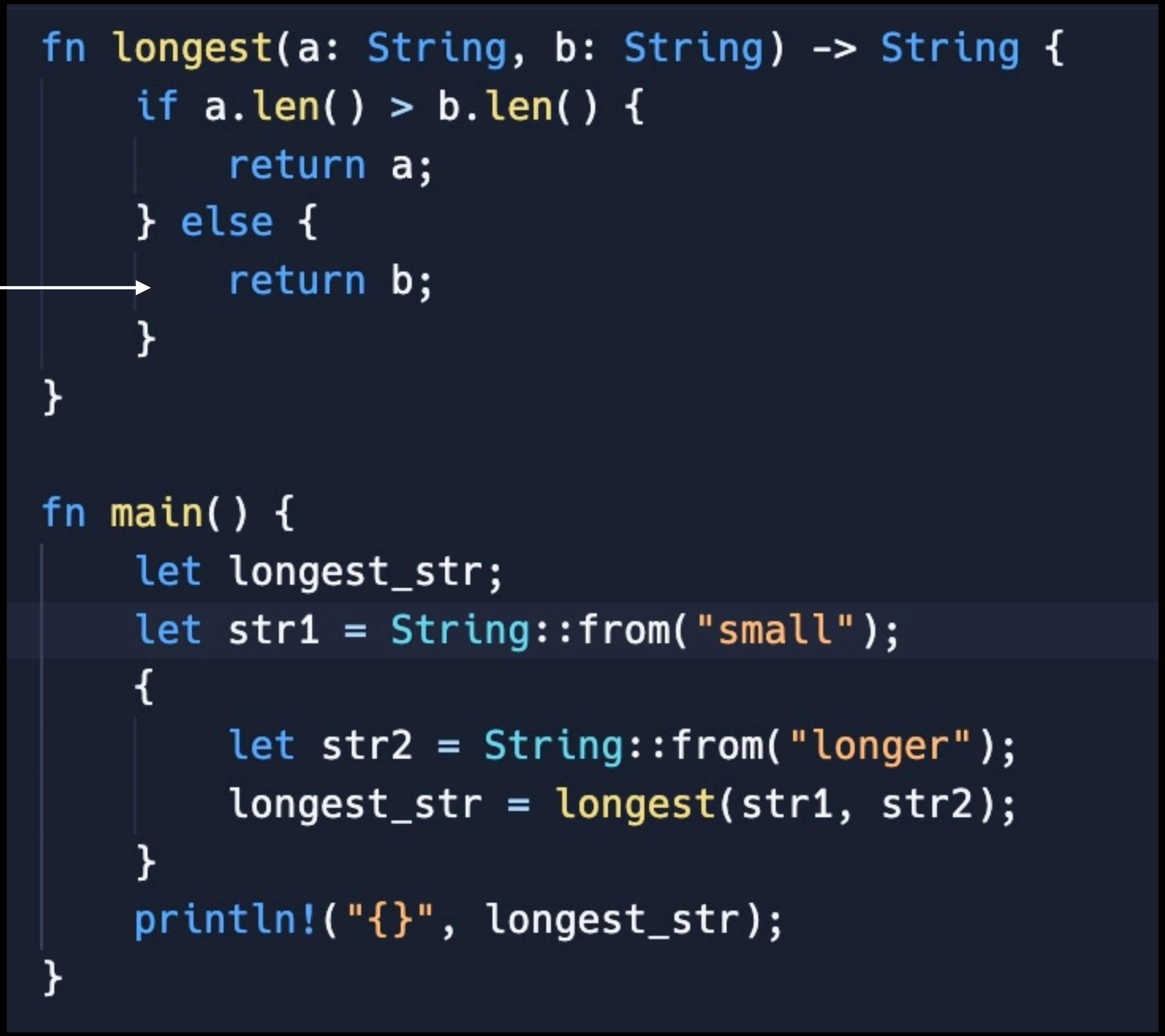


```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

Both str1 and str2 get moved

Lifetimes

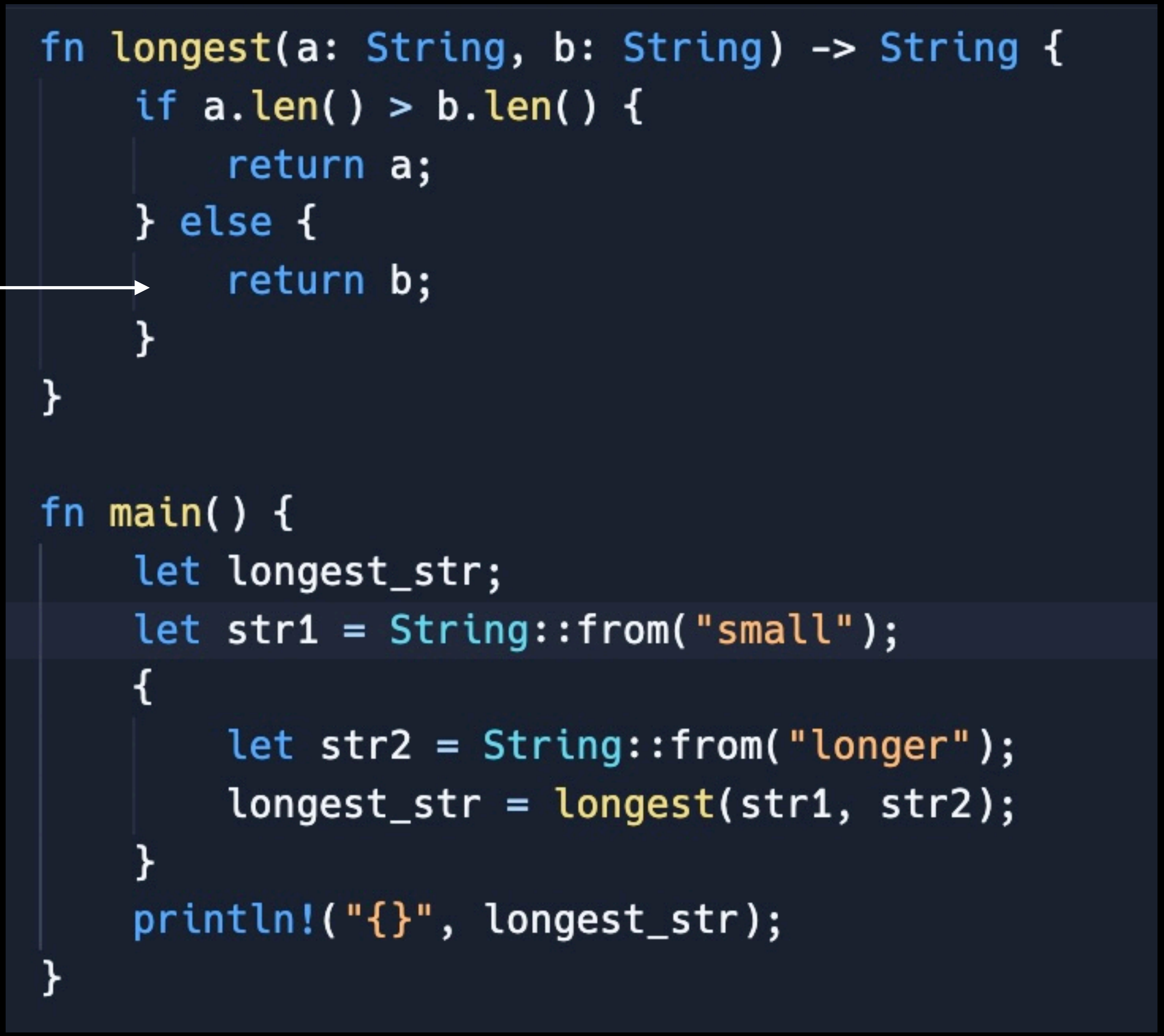
B (str1) gets moved back



```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

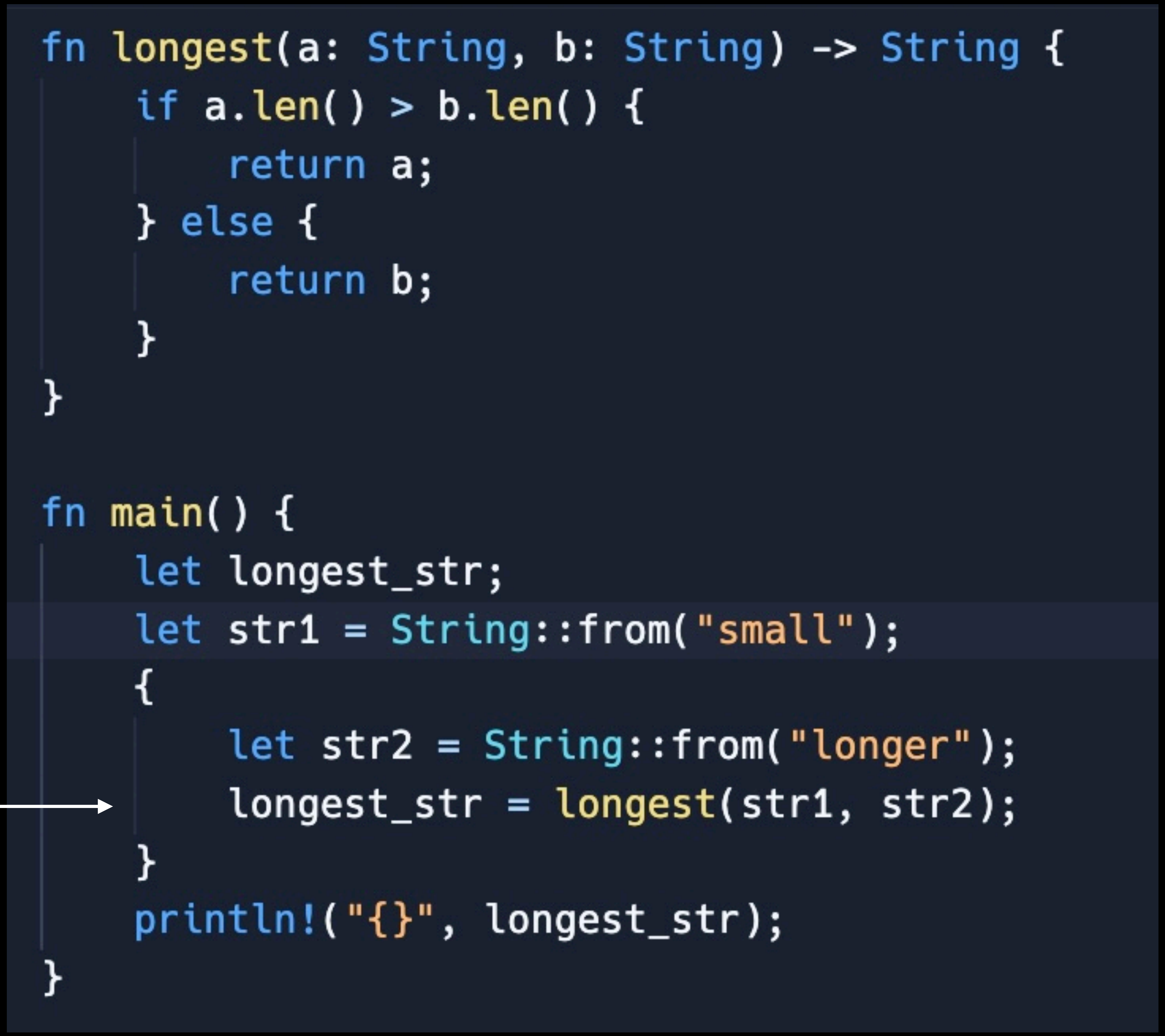
Lifetimes

B (str1) gets moved back



```
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

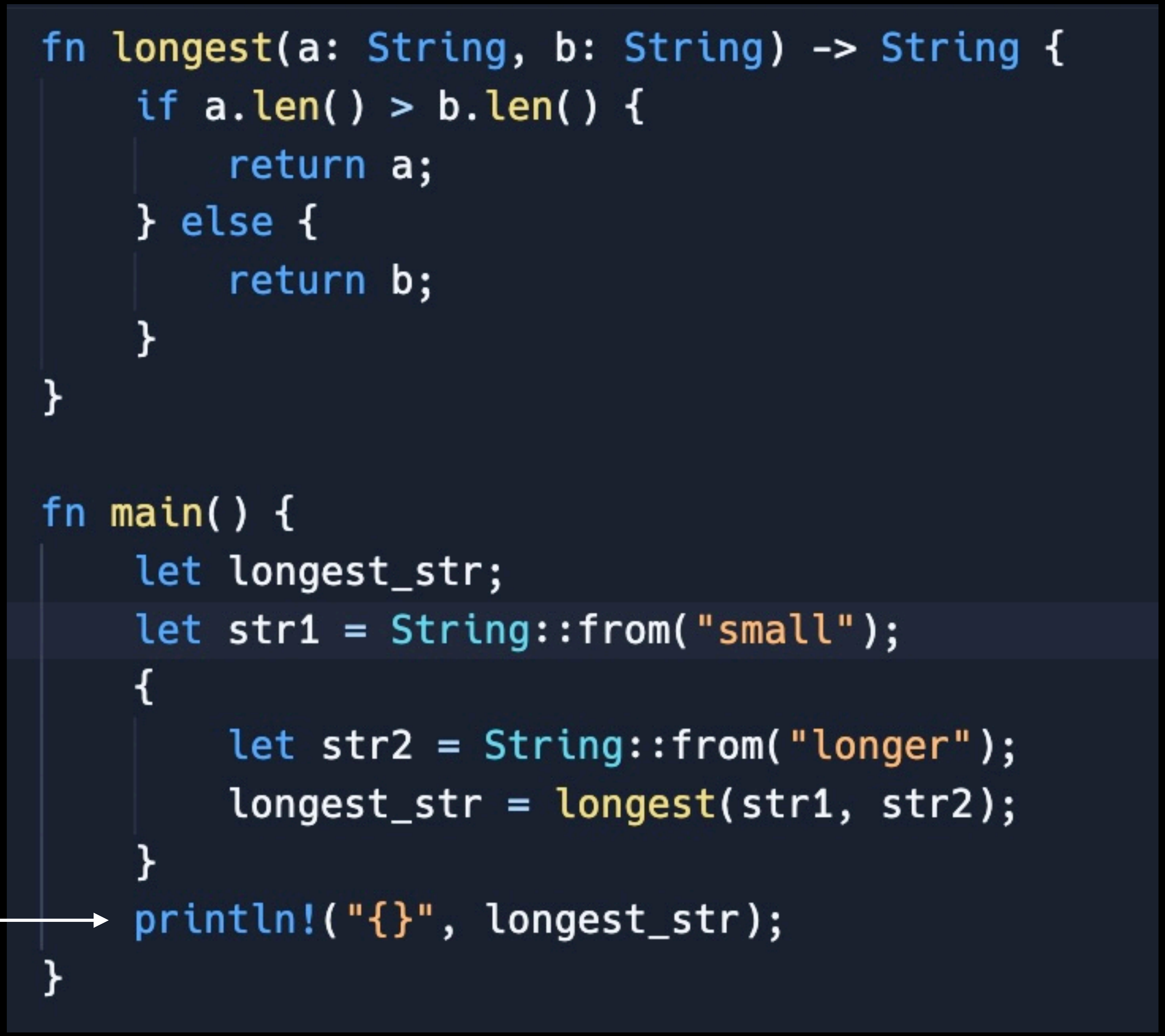
Lifetimes



```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

longest_str owns the string →

Lifetimes



```rust
fn longest(a: String, b: String) -> String {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(str1, str2);
    }
    println!("{}", longest_str);
}
```

longest_str owns gets printed →

Lifetimes

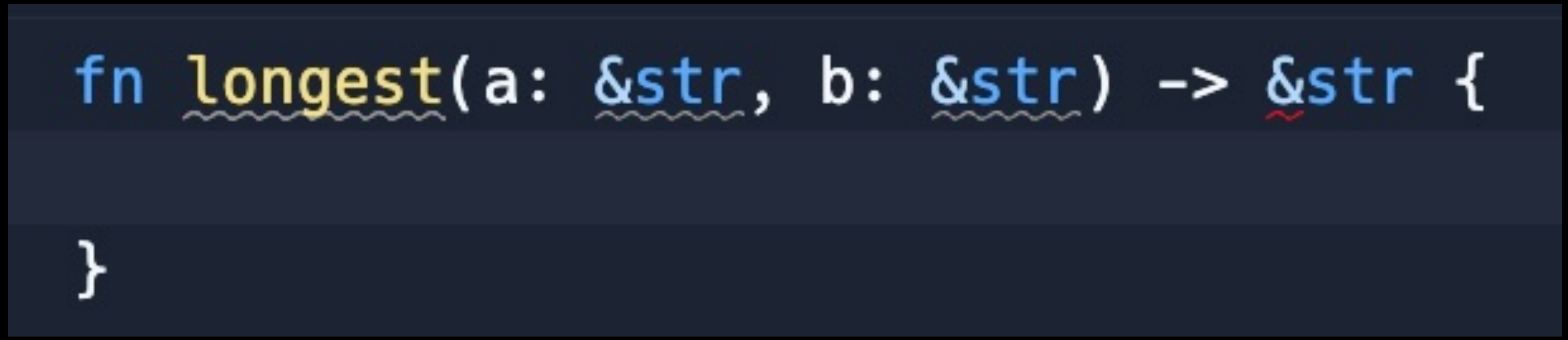
What we’ve done until now has nothing to do with lifetimes

Lifetimes

What we’ve done until now has nothing to do with lifetimes
Lets change the function signature a bit

Lifetimes

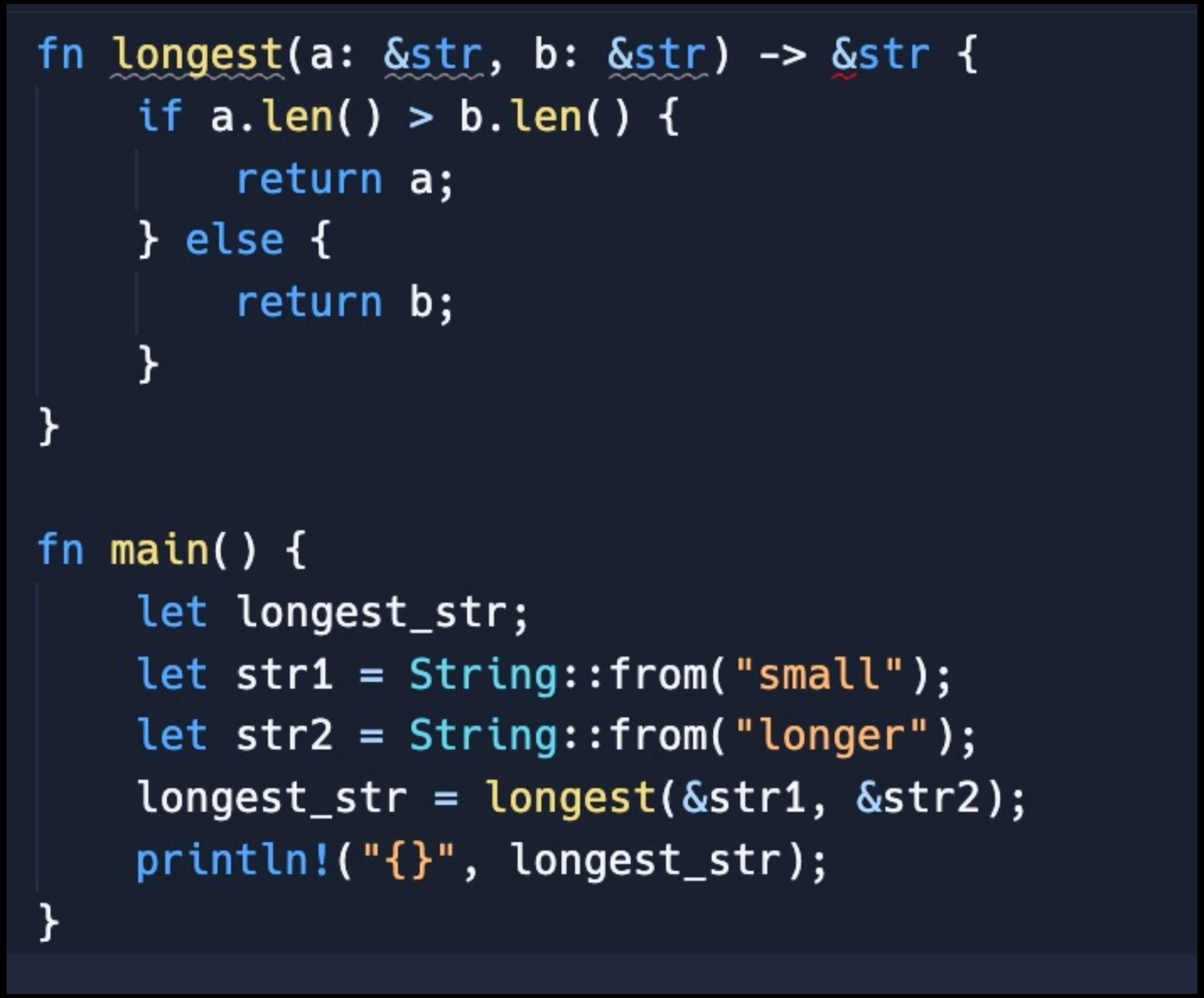
Q - Write a function that takes two string references as an input
And returns the bigger amongst them



```
fn longest(a: &str, b: &str) -> &str {

}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them

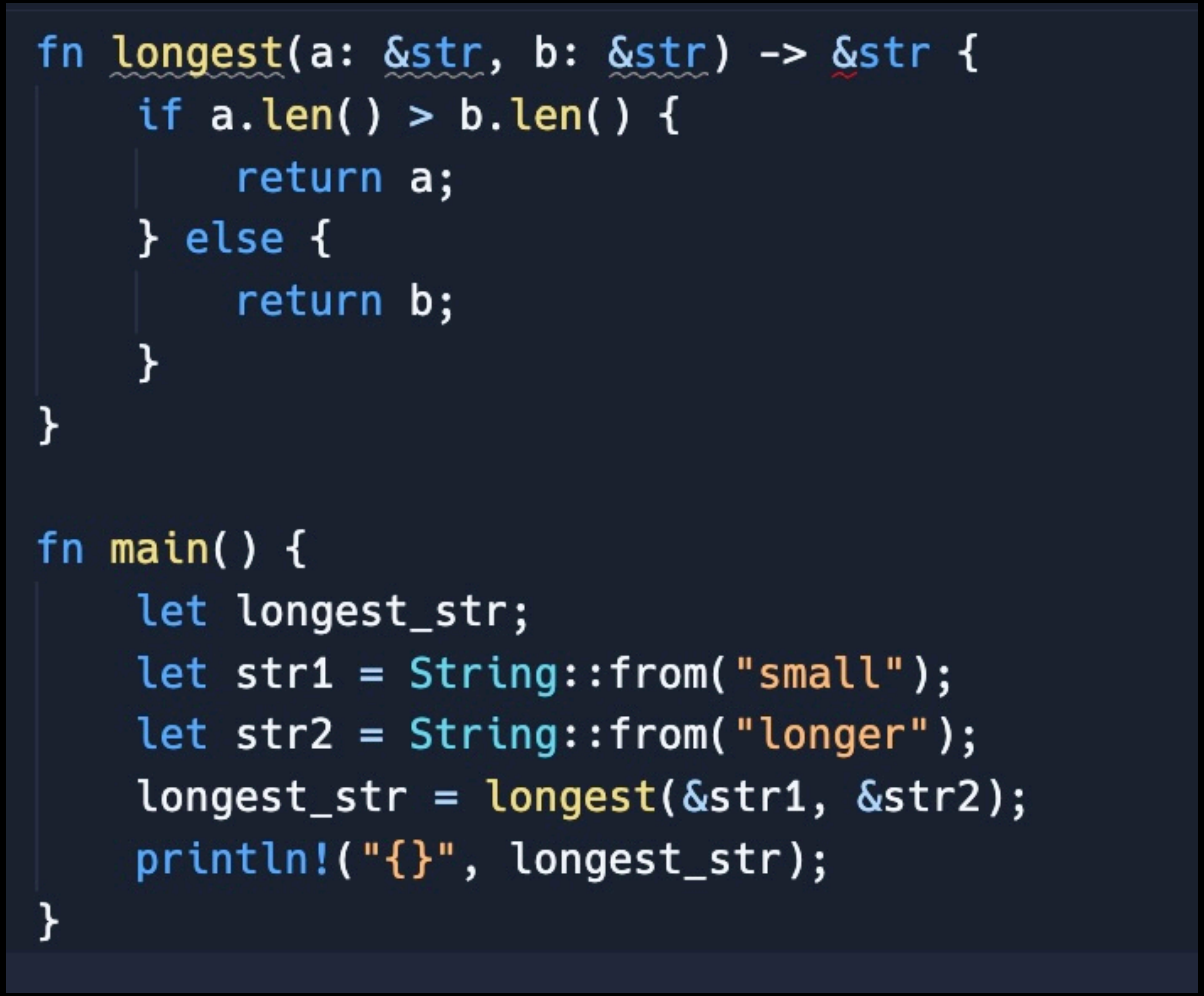


```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(&str1, &str2);
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them

Do you think this code is valid?



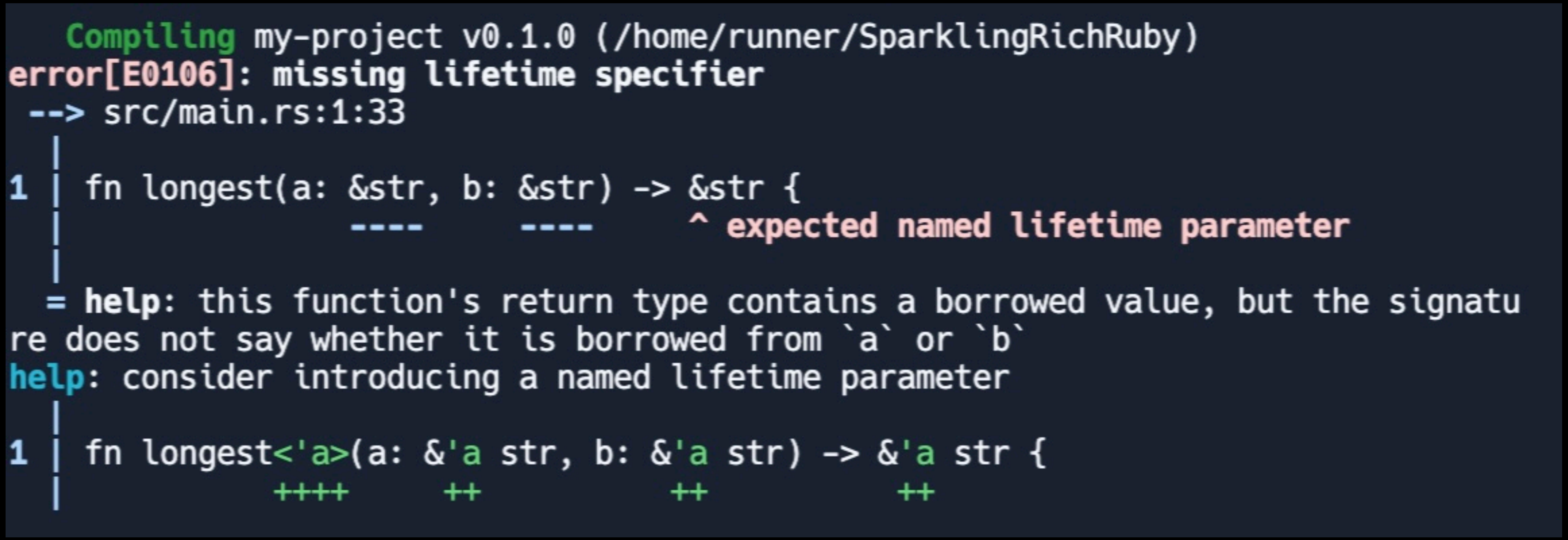
```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(&str1, &str2);
    println!("{}", longest_str);
}
```

Lifetimes

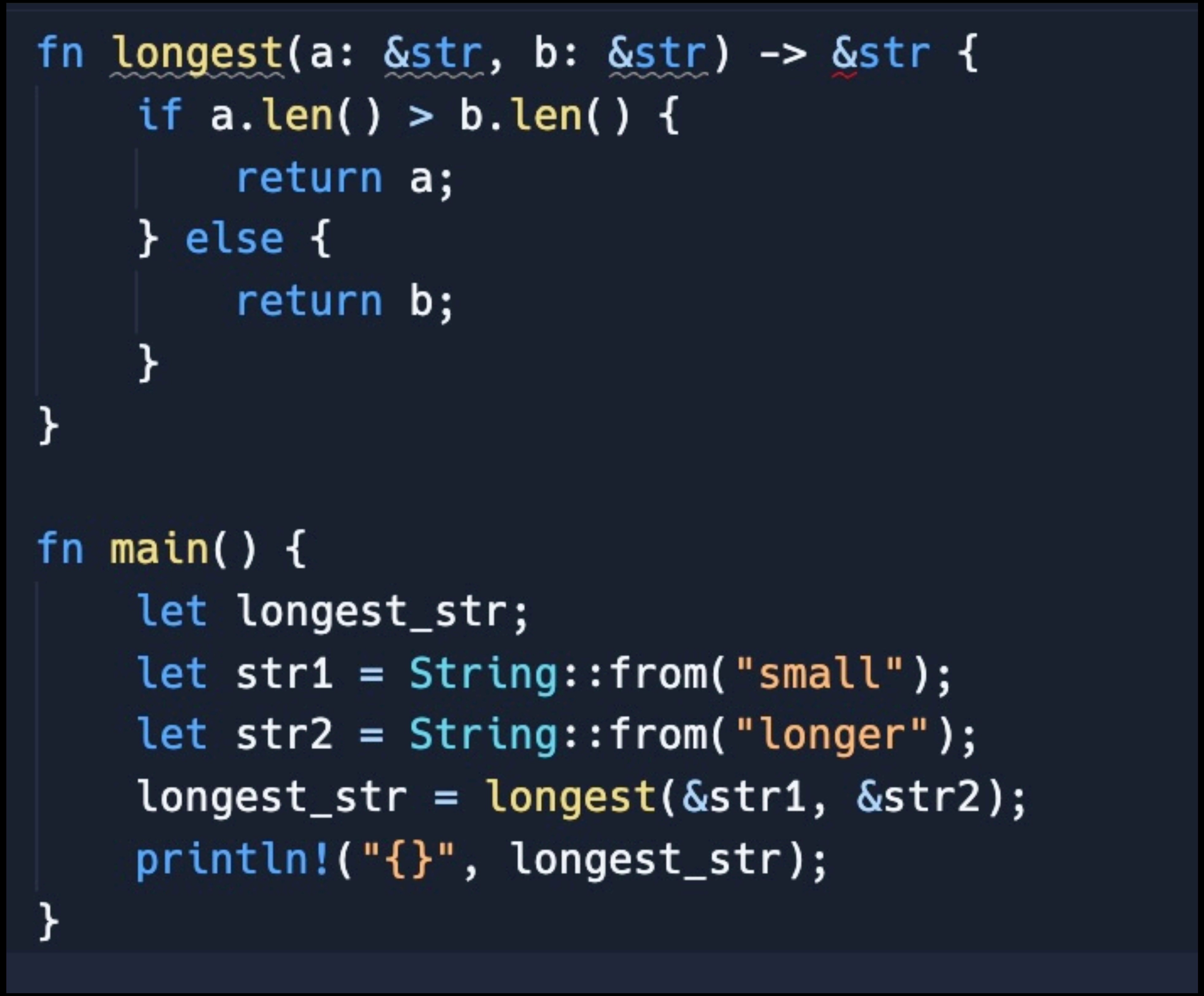
Q - Write a function that takes two string references as an input
And returns the bigger amongst them

Do you think this code is valid?

No



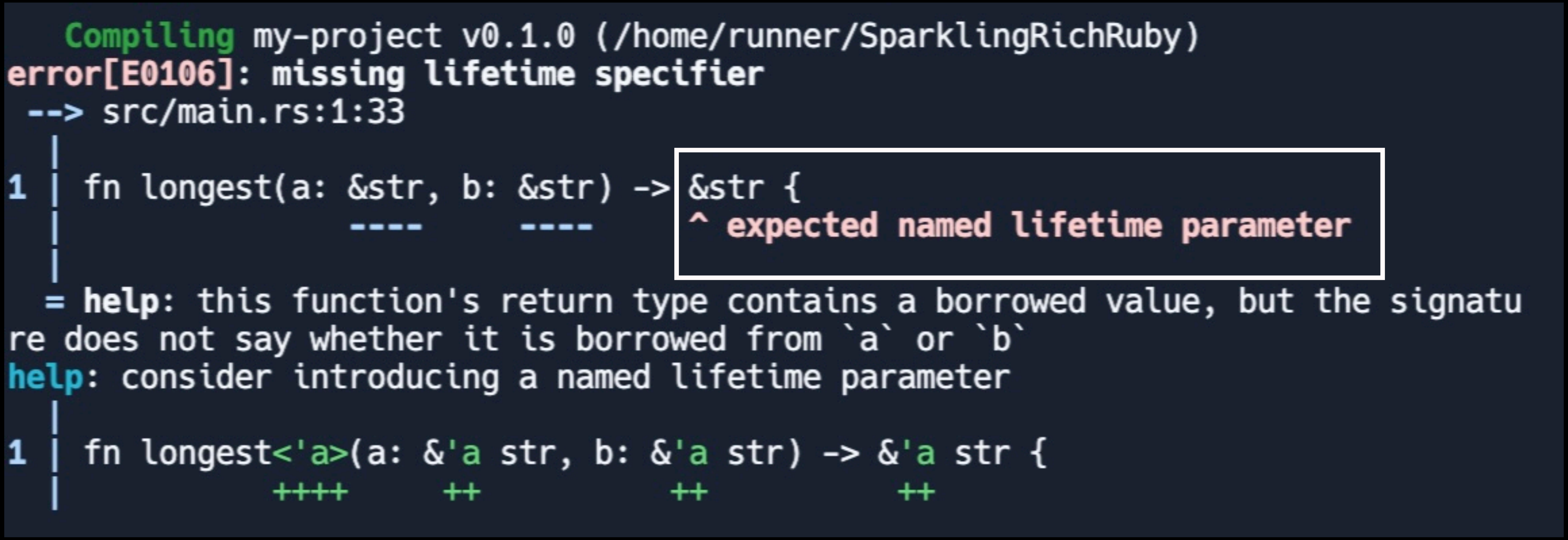
```
   Compiling my-project v0.1.0 (/home/runner/SparklingRichRuby)
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(a: &str, b: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signatu
re does not say whether it is borrowed from `a` or `b`
help: consider introducing a named lifetime parameter
  |
1 | fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```



```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    let str2 = String::from("longer");
    longest_str = longest(&str1, &str2);
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them

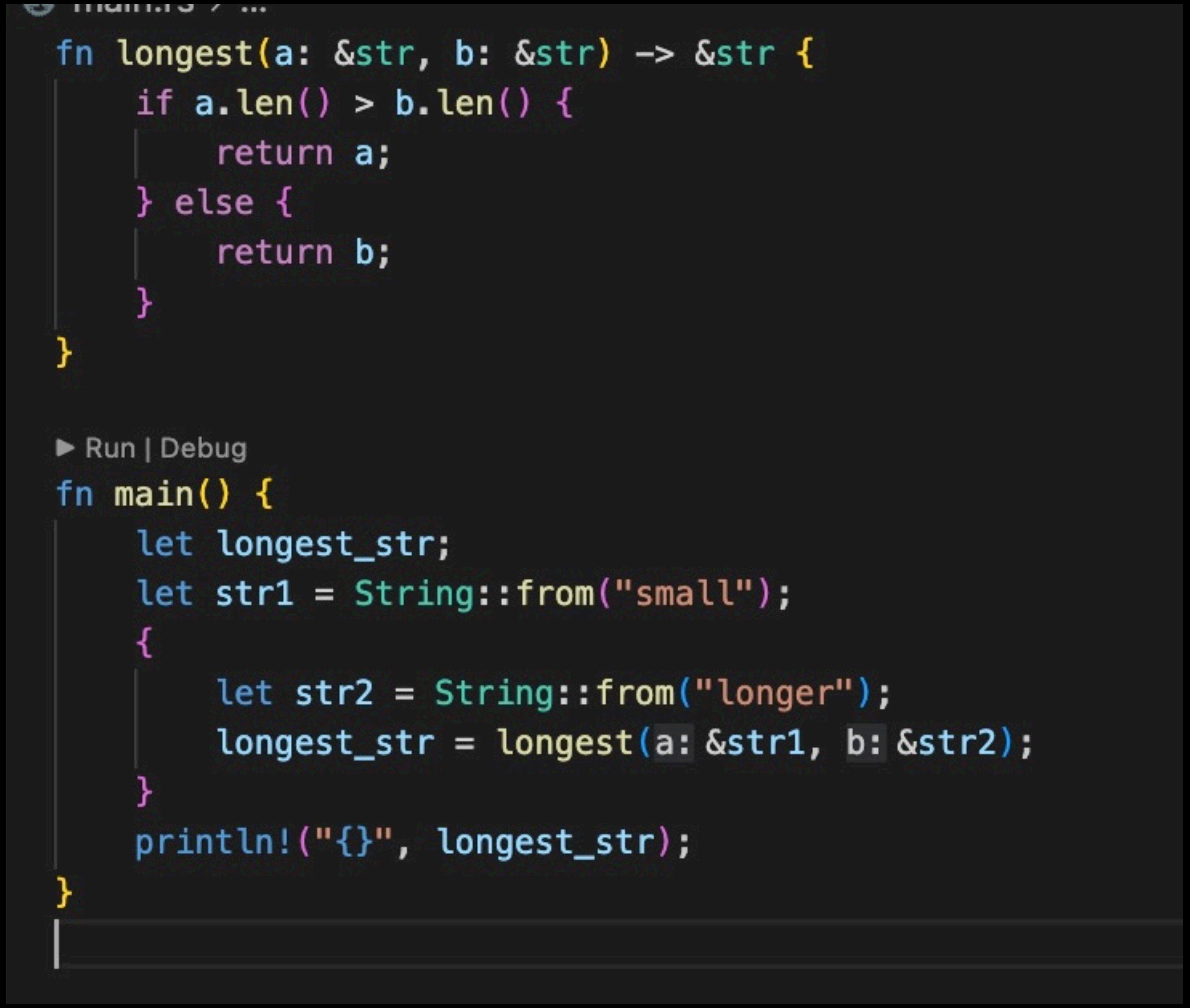


```
   Compiling my-project v0.1.0 (/home/runner/SparklingRichRuby)
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(a: &str, b: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signatu
re does not say whether it is borrowed from `a` or `b`
help: consider introducing a named lifetime parameter
  |
1 | fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```

Lifetimes

**Q - Write a function that takes two string references as an input
And returns the bigger amongst them**

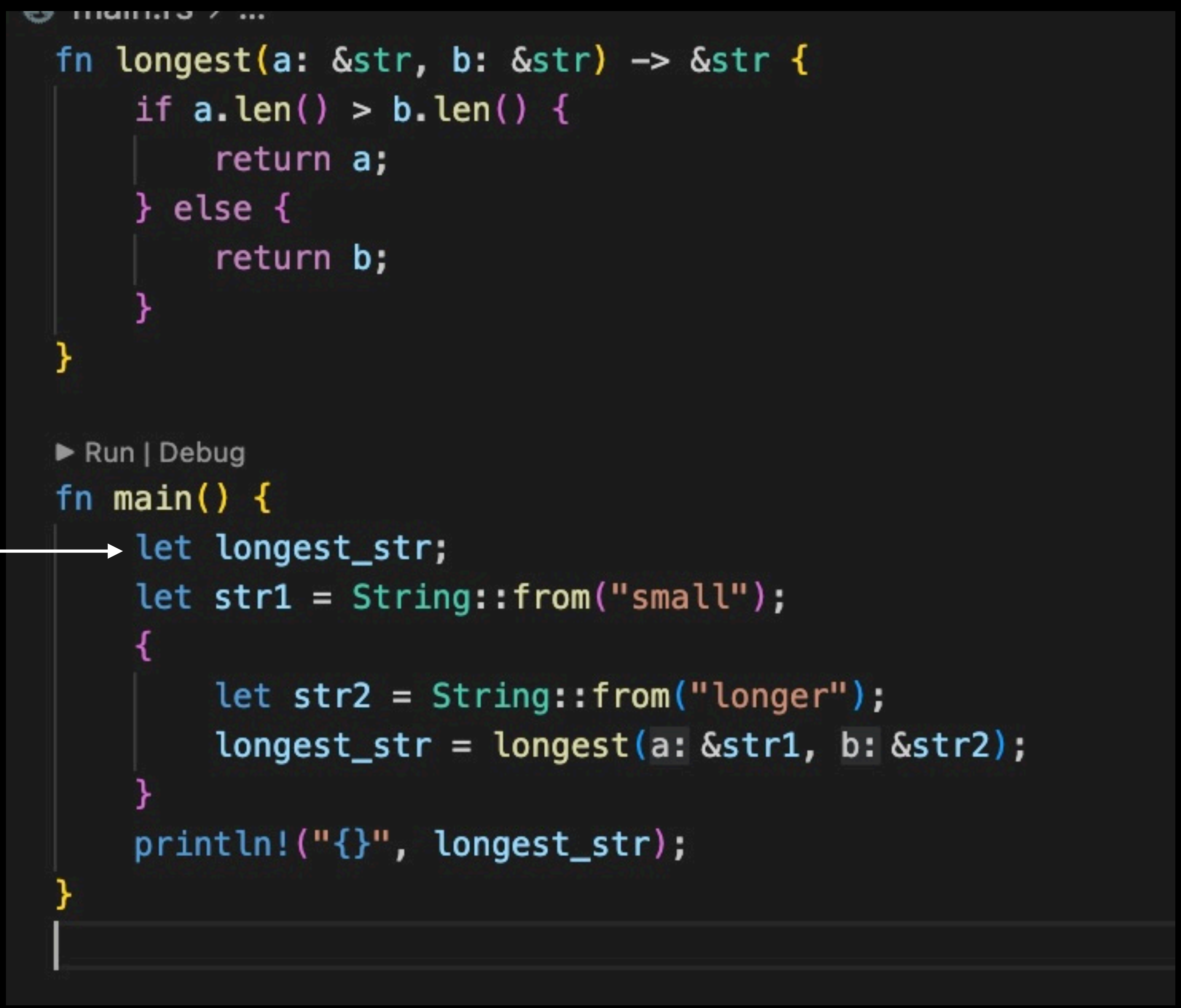
**Can you look at the code and tell
Why the compiler complaining is a good thing?**



```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them



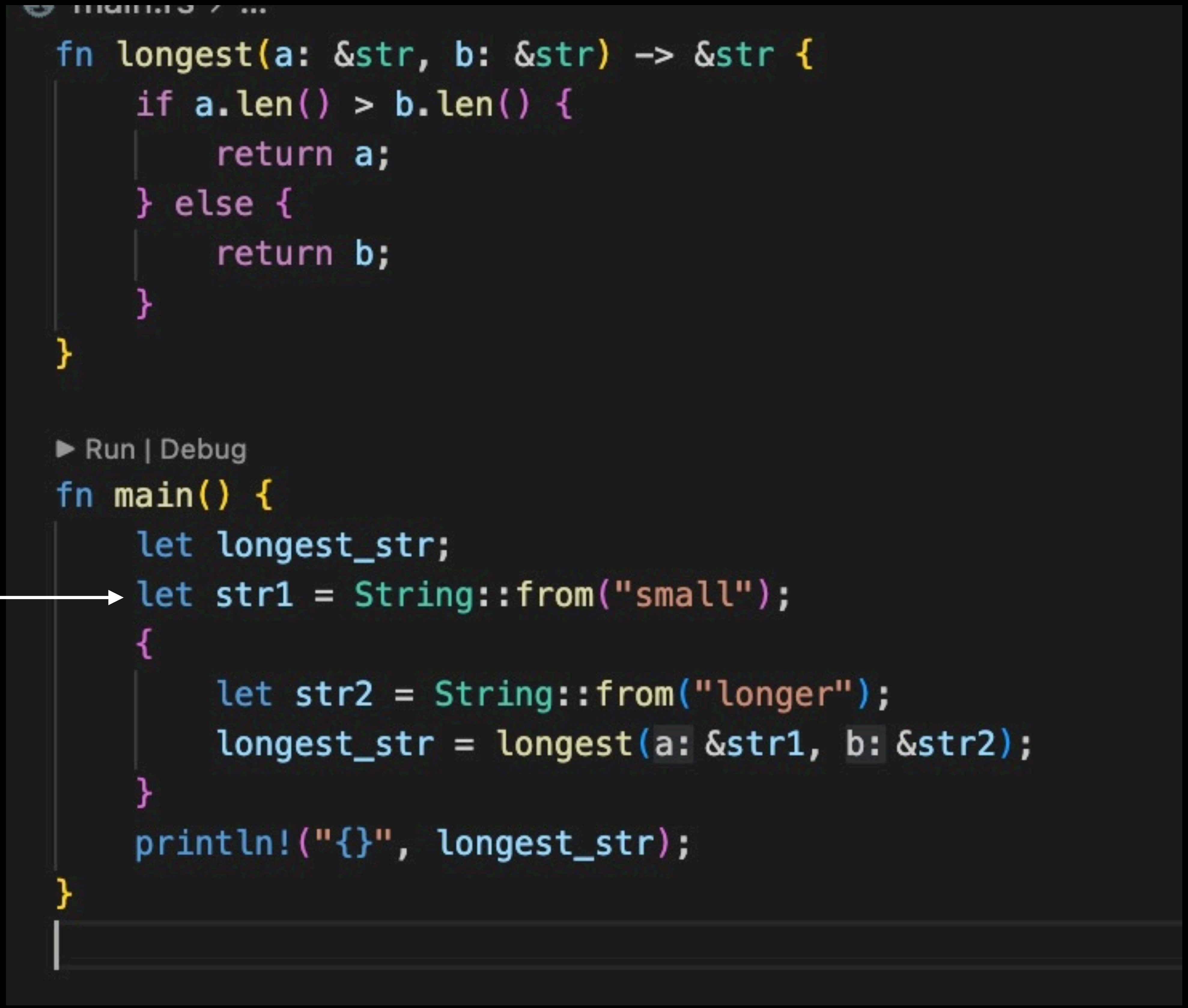
```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Points to nothing → `let longest_str;`

Lifetimes

**Q - Write a function that takes two string references as an input
And returns the bigger amongst them**

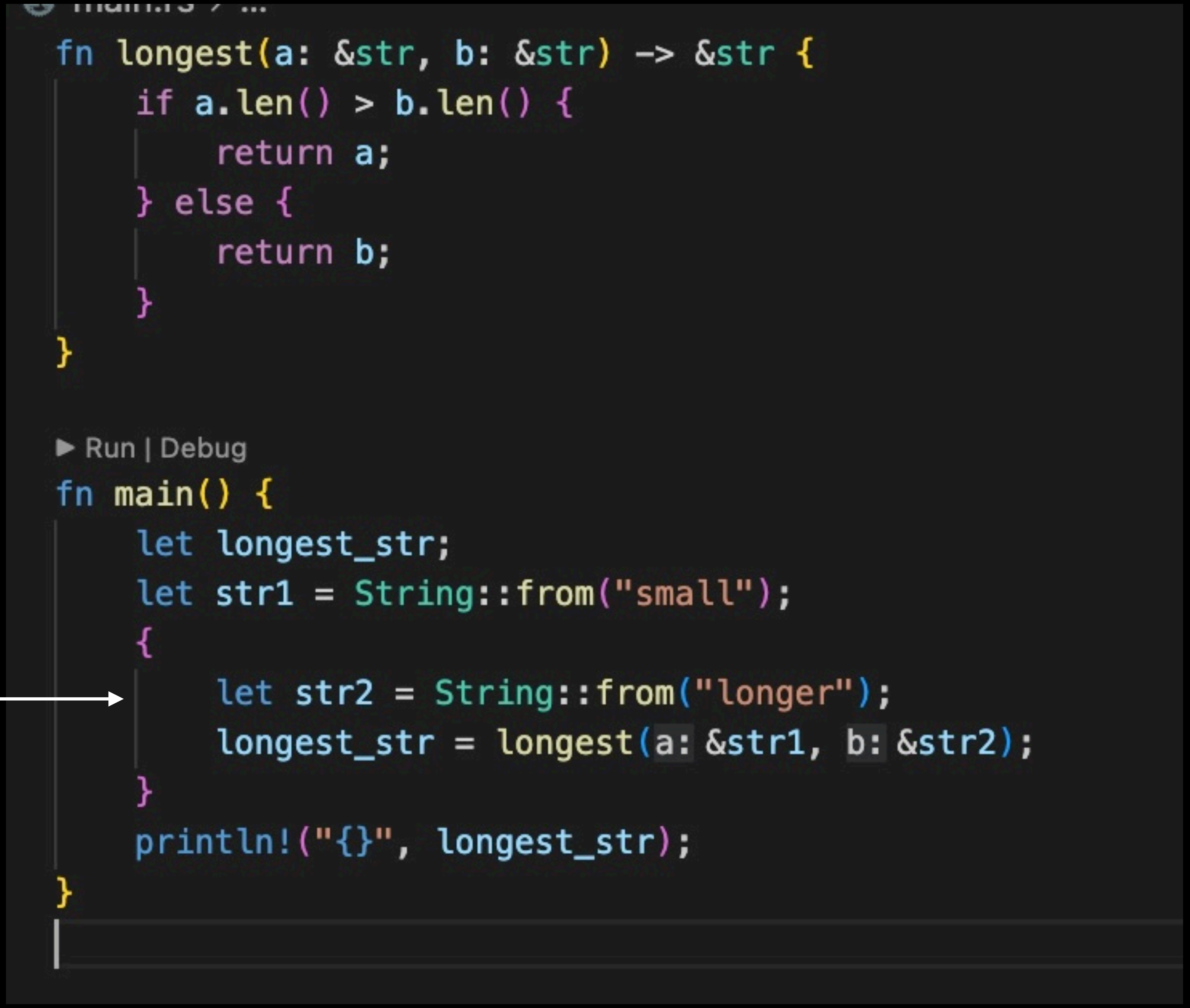
Str1 owns the string



```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them

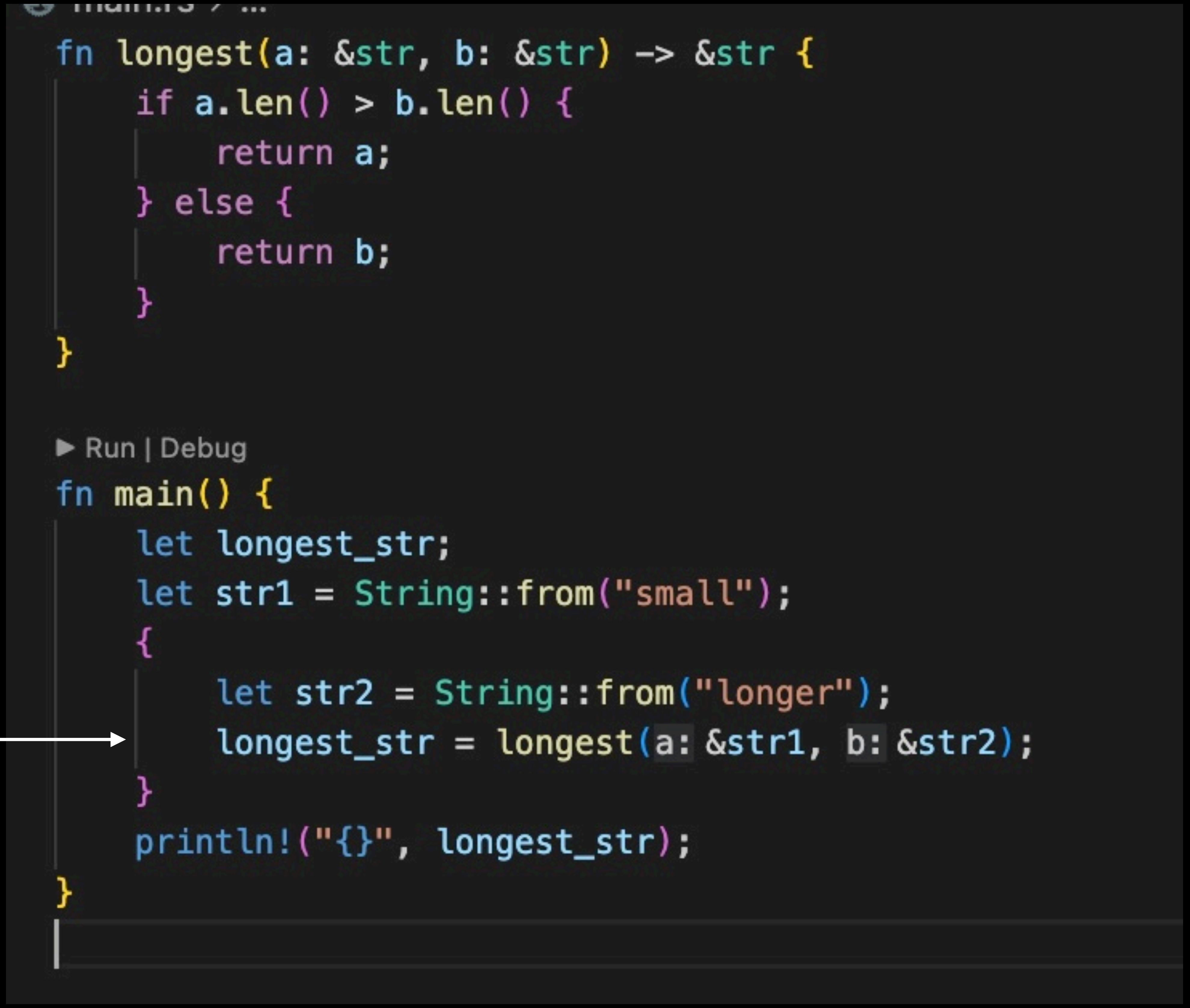


```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Str2 owns the string →

Lifetimes

**Q - Write a function that takes two string references as an input
And returns the bigger amongst them**



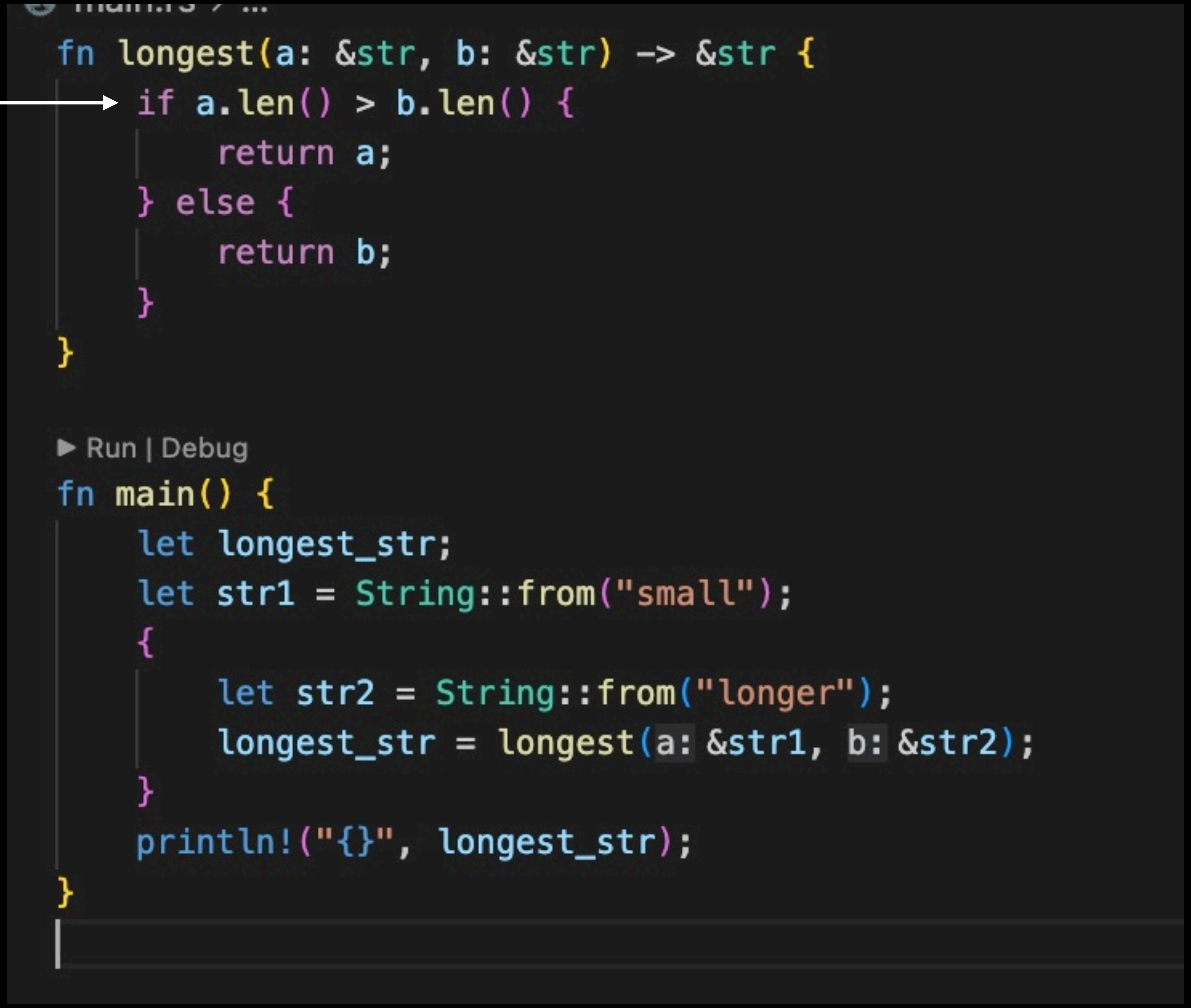
```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Str1 and str2 get borrowed →

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them

Condition checks →

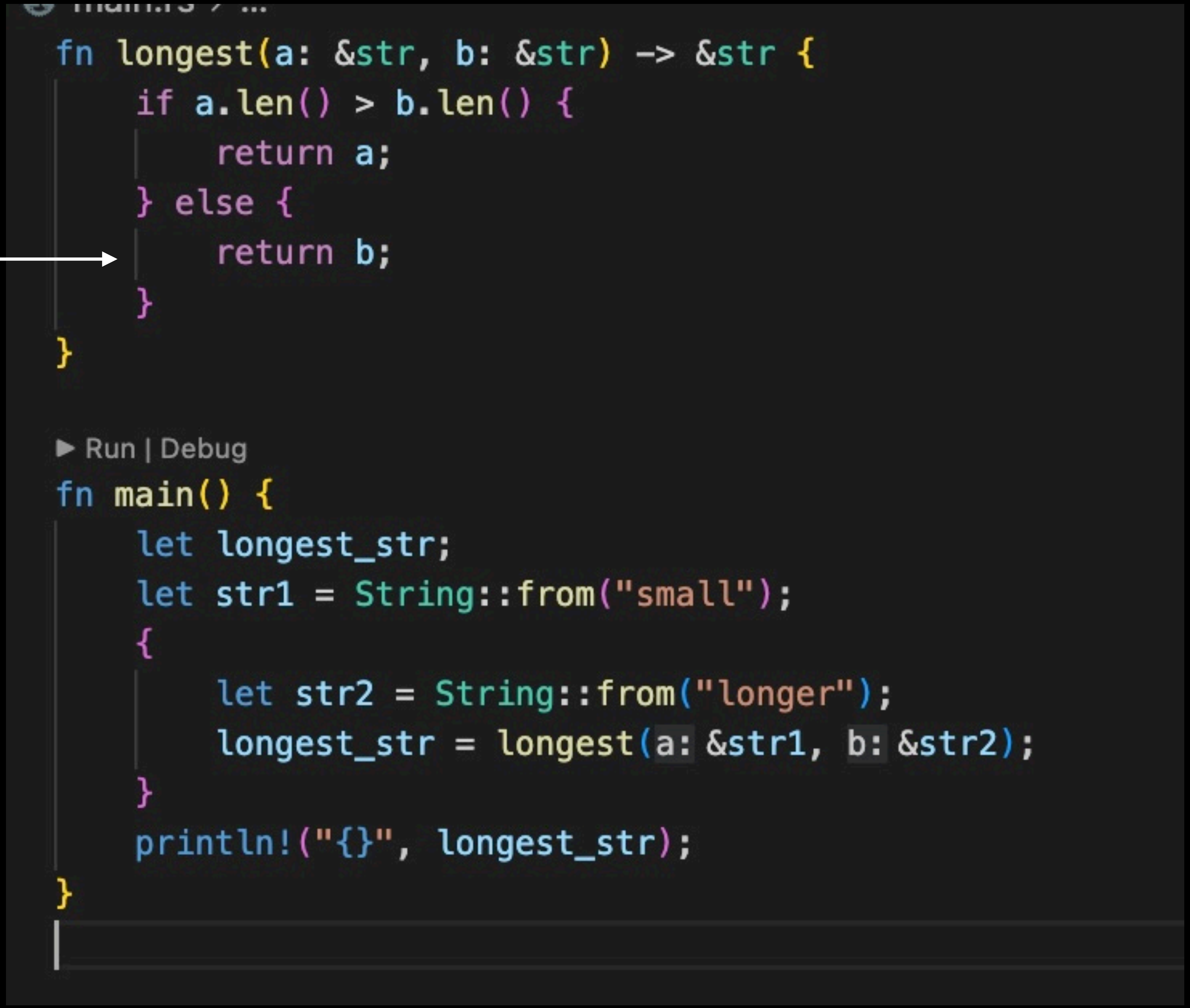


```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them

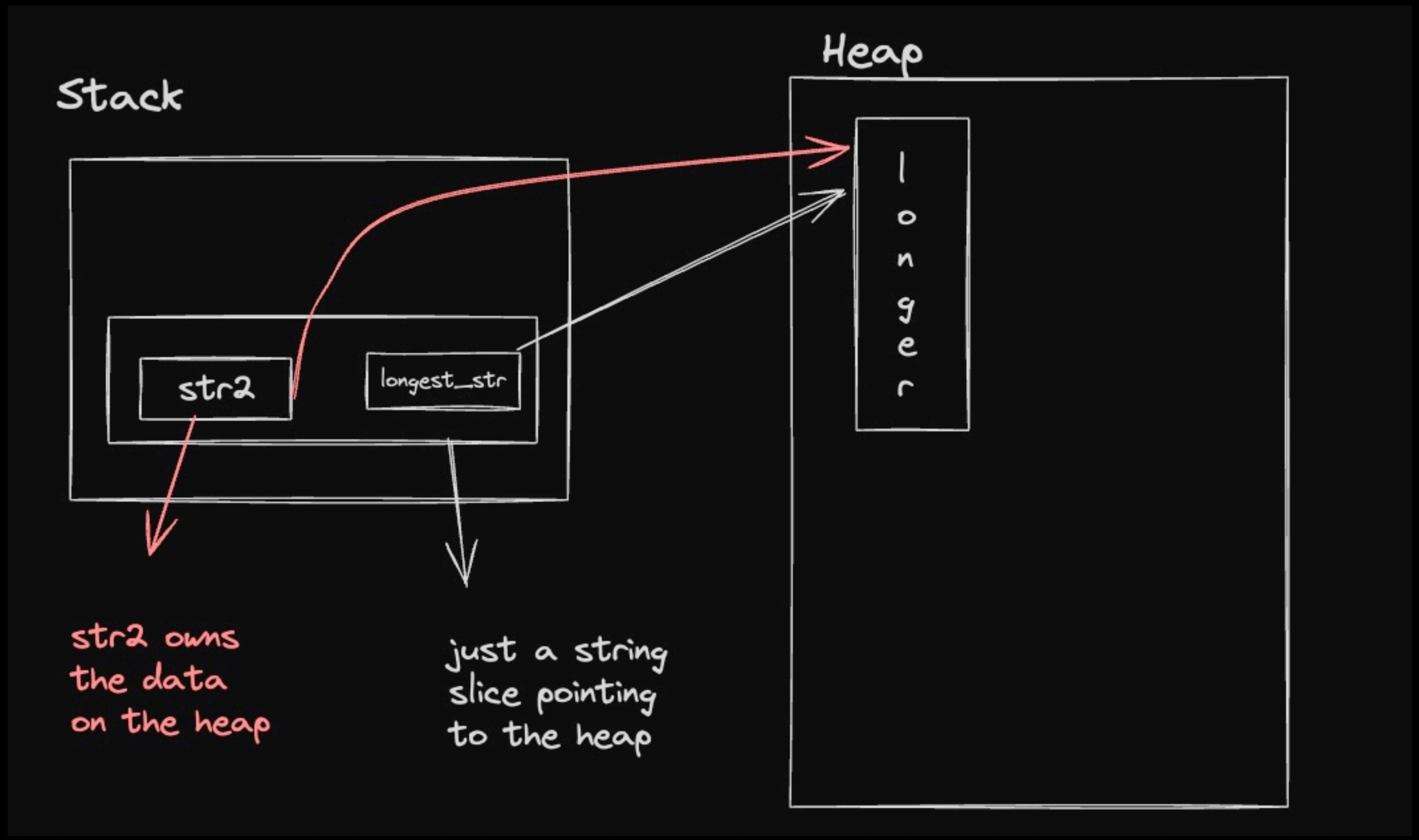
Reference to b gets returned



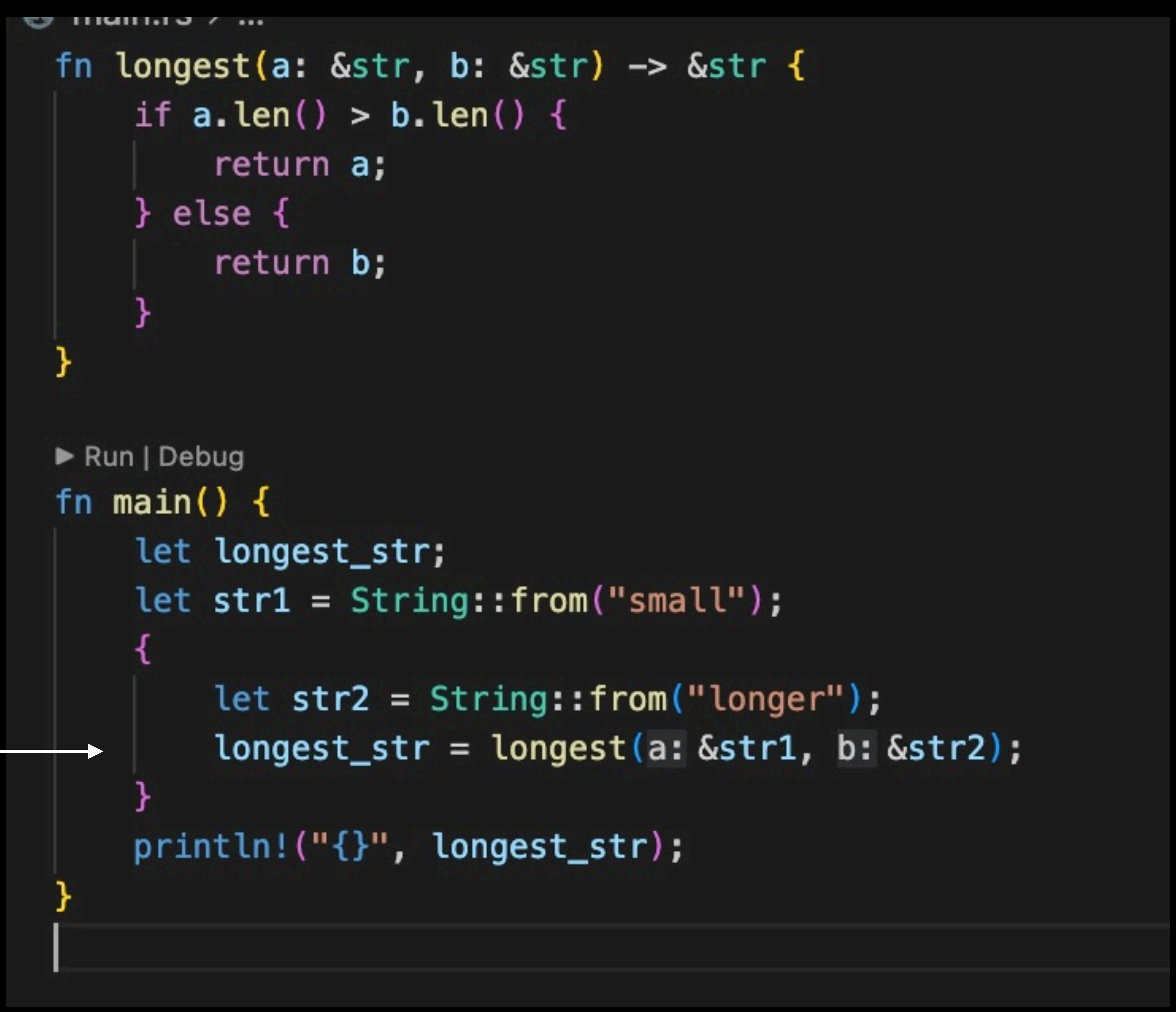
```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them



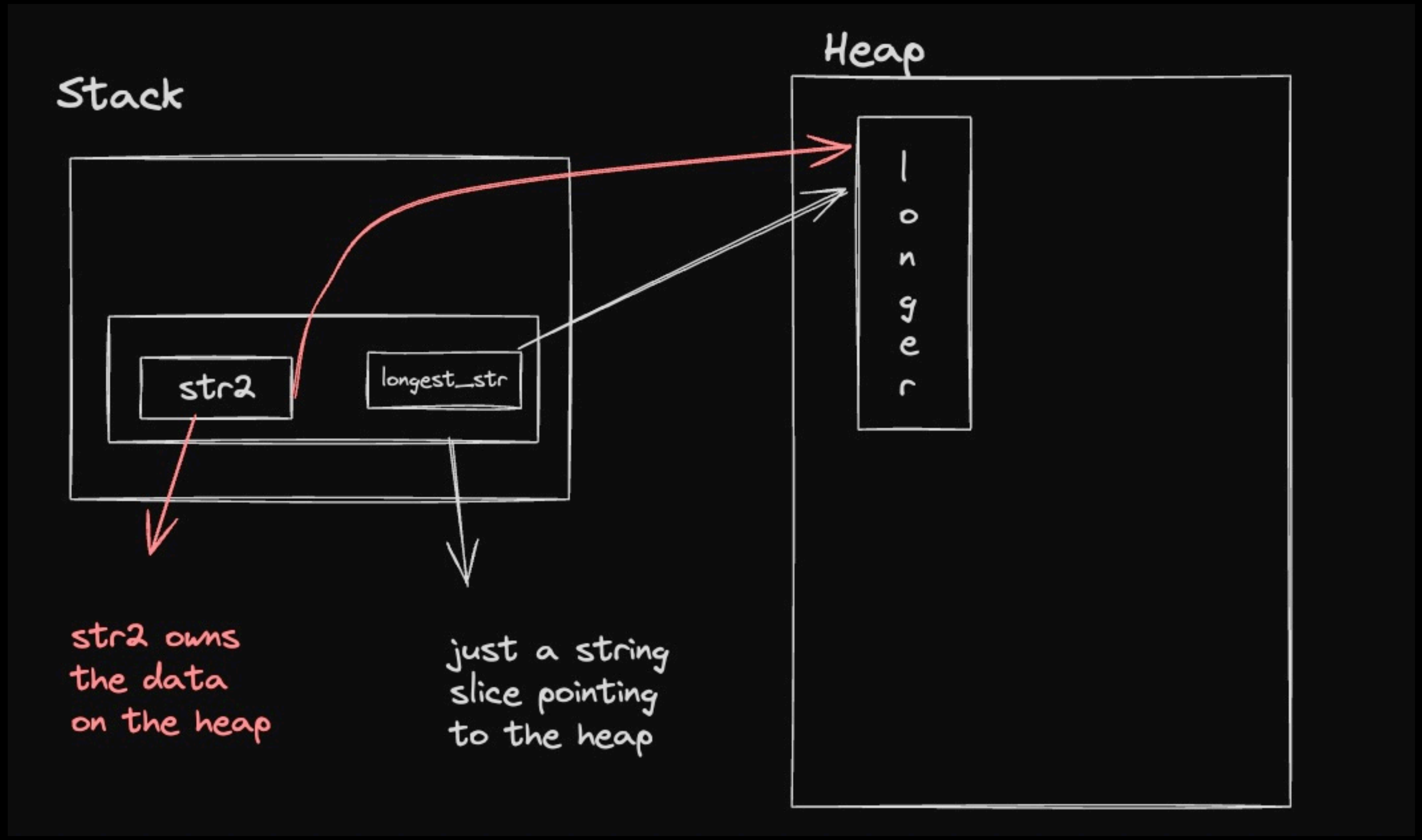
longest_string is a string slice
Pointing to str2



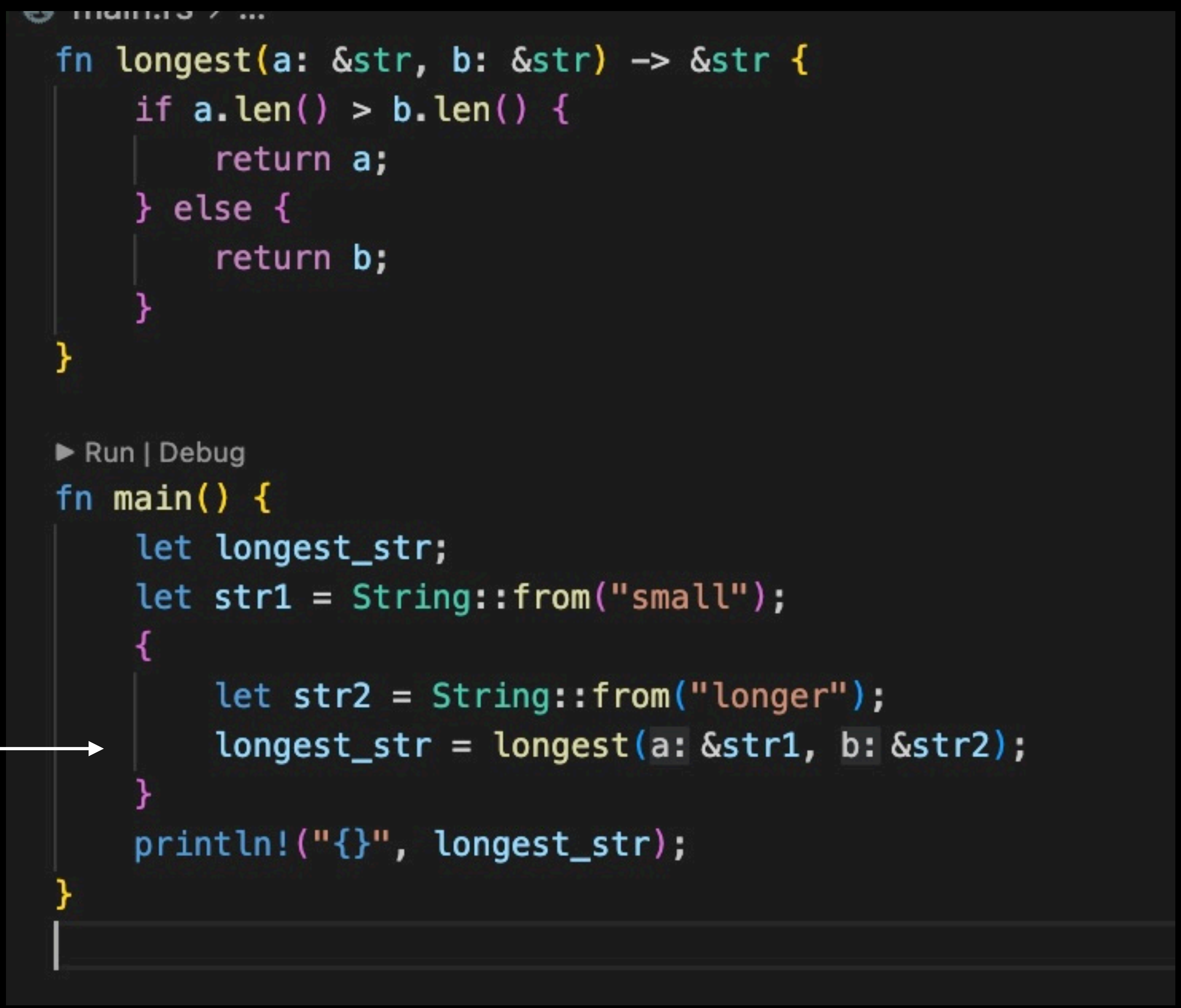
```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

▶ Run | Debug
fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them


Stack
Heap
str2
longest_str
l o n g e r
str2 owns the data on the heap
just a string slice pointing to the heap

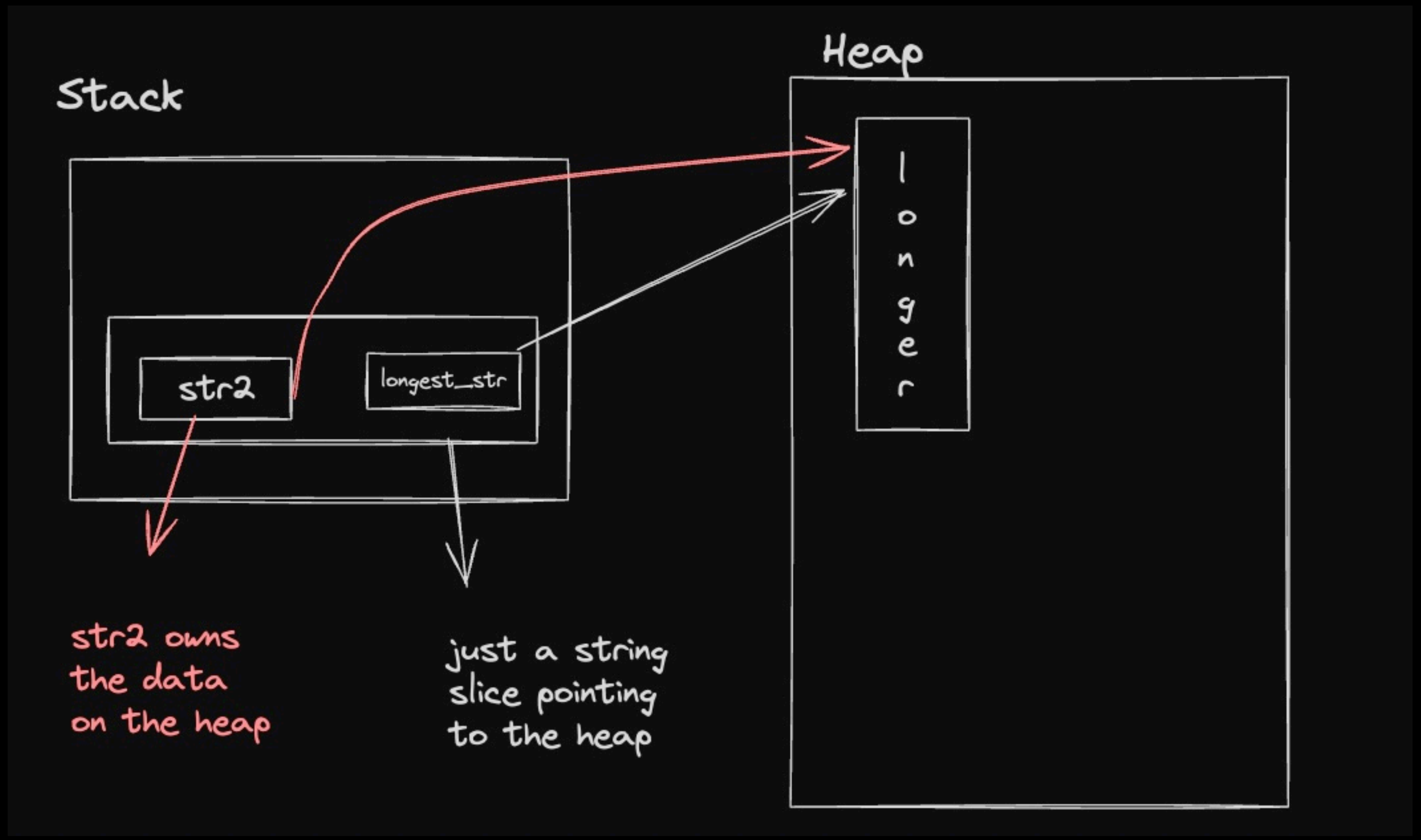
longest_string is a string slice
Pointing to str2

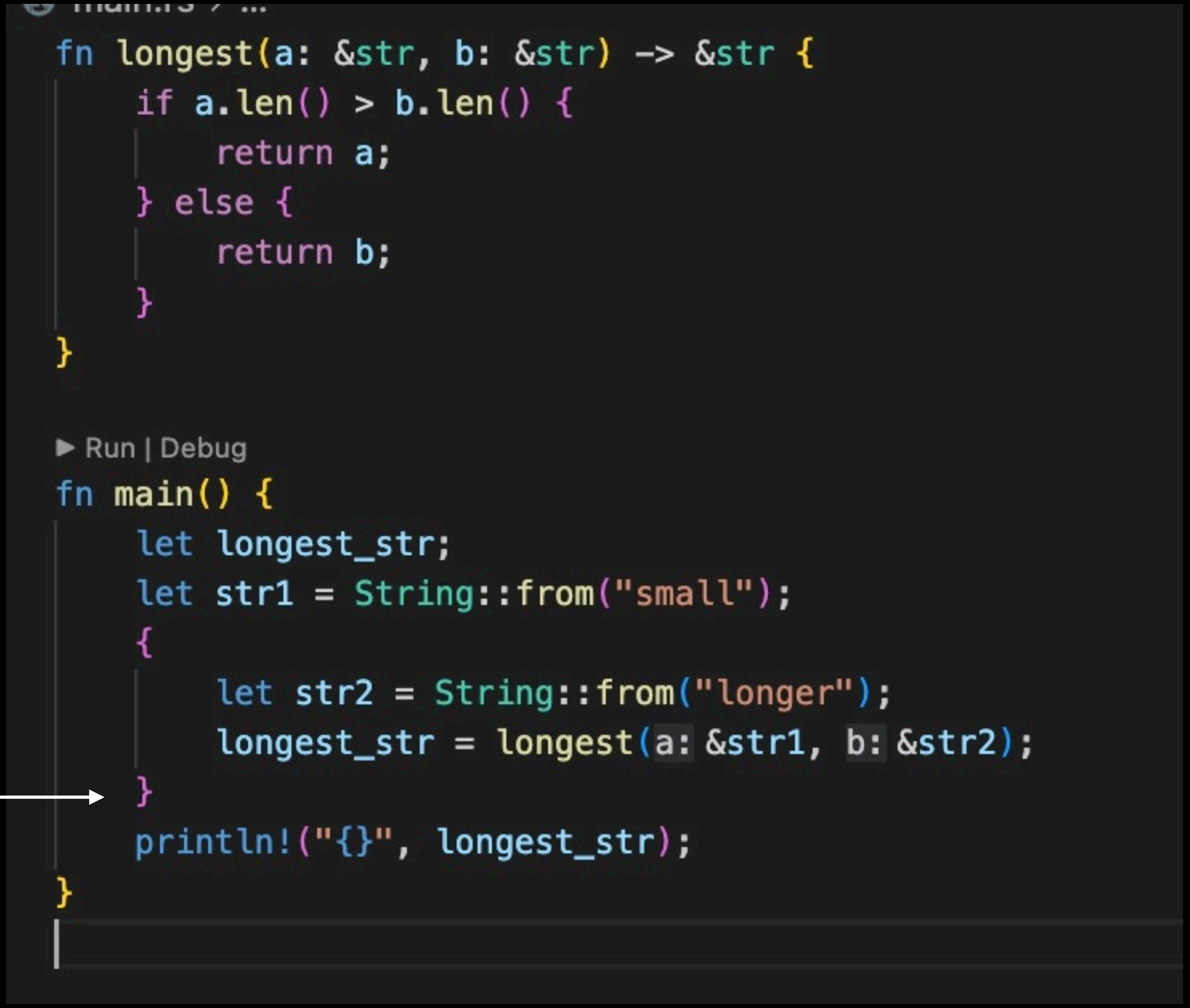


```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them


Heap
Stack
str2
longest_str
l o n g e r
str2 owns the data on the heap
just a string slice pointing to the heap

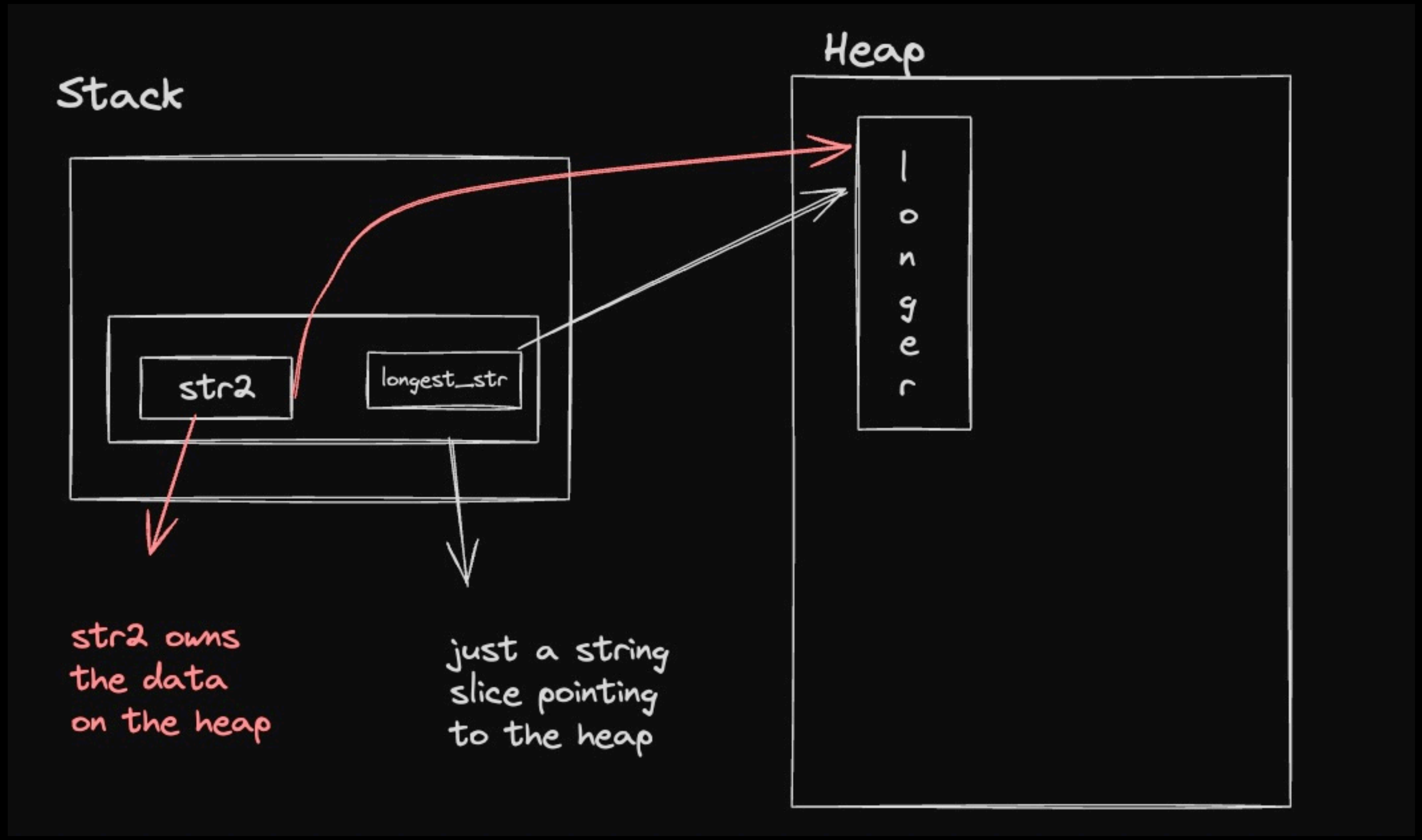


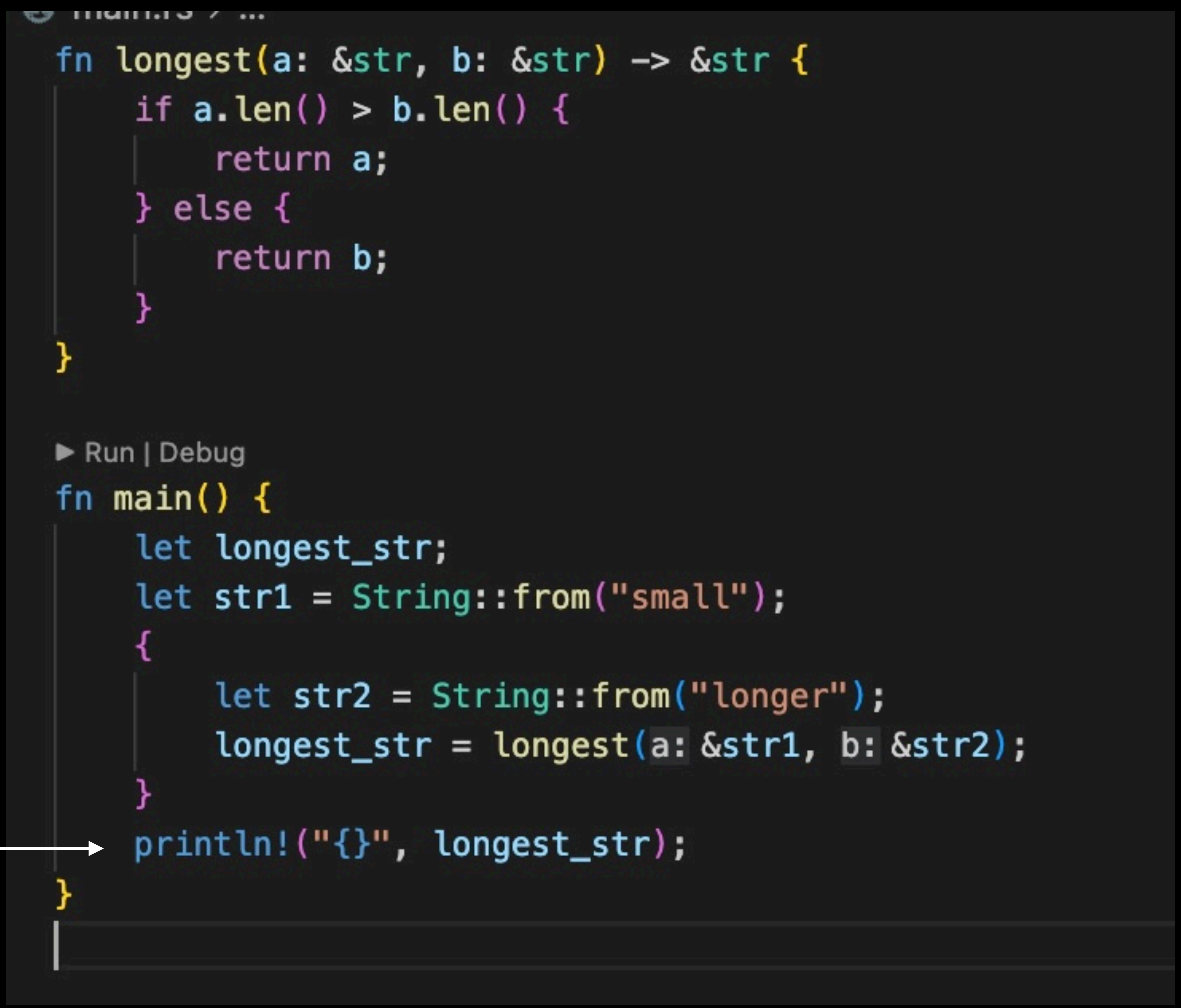
```rust
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

str2 goes out of scope
Data gets removed from the heap

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them


Stack
Heap
str2
longest_str
l
o
n
g
e
r
str2 owns the data on the heap
just a string slice pointing to the heap



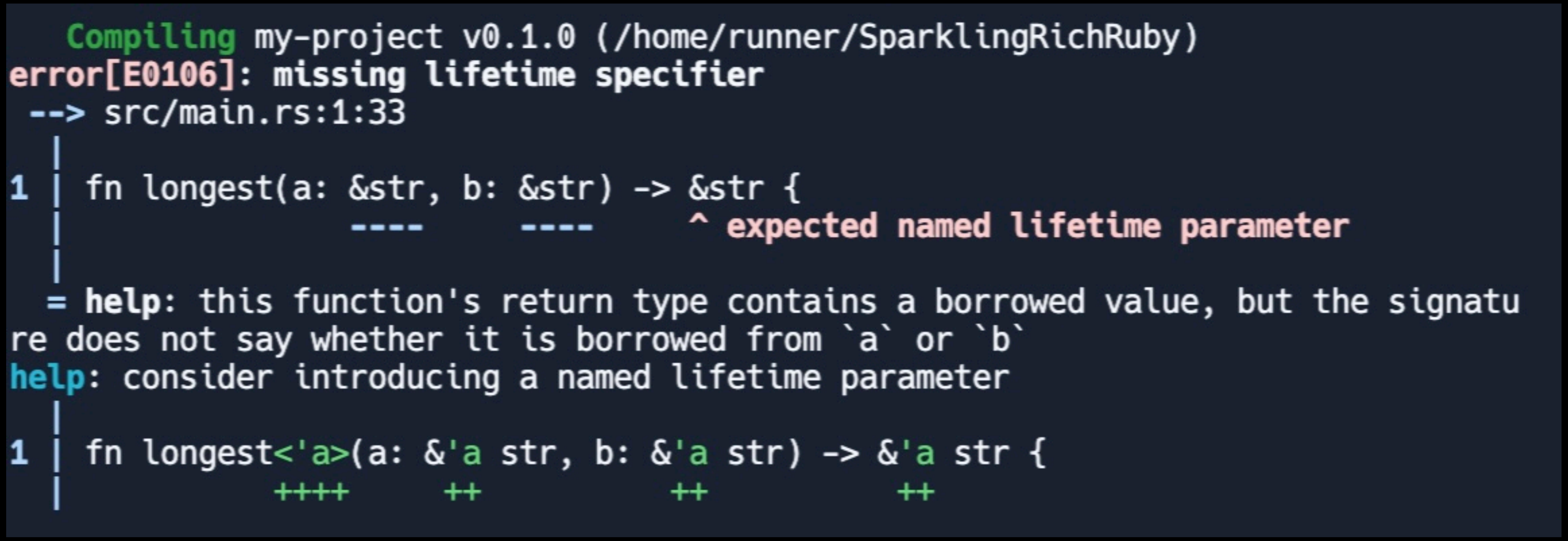
```
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(a: &str1, b: &str2);
    }
    println!("{}", longest_str);
}
```

longest_str is a dangling pointer →

Lifetimes

Q - Write a function that takes two string references as an input
And returns the bigger amongst them

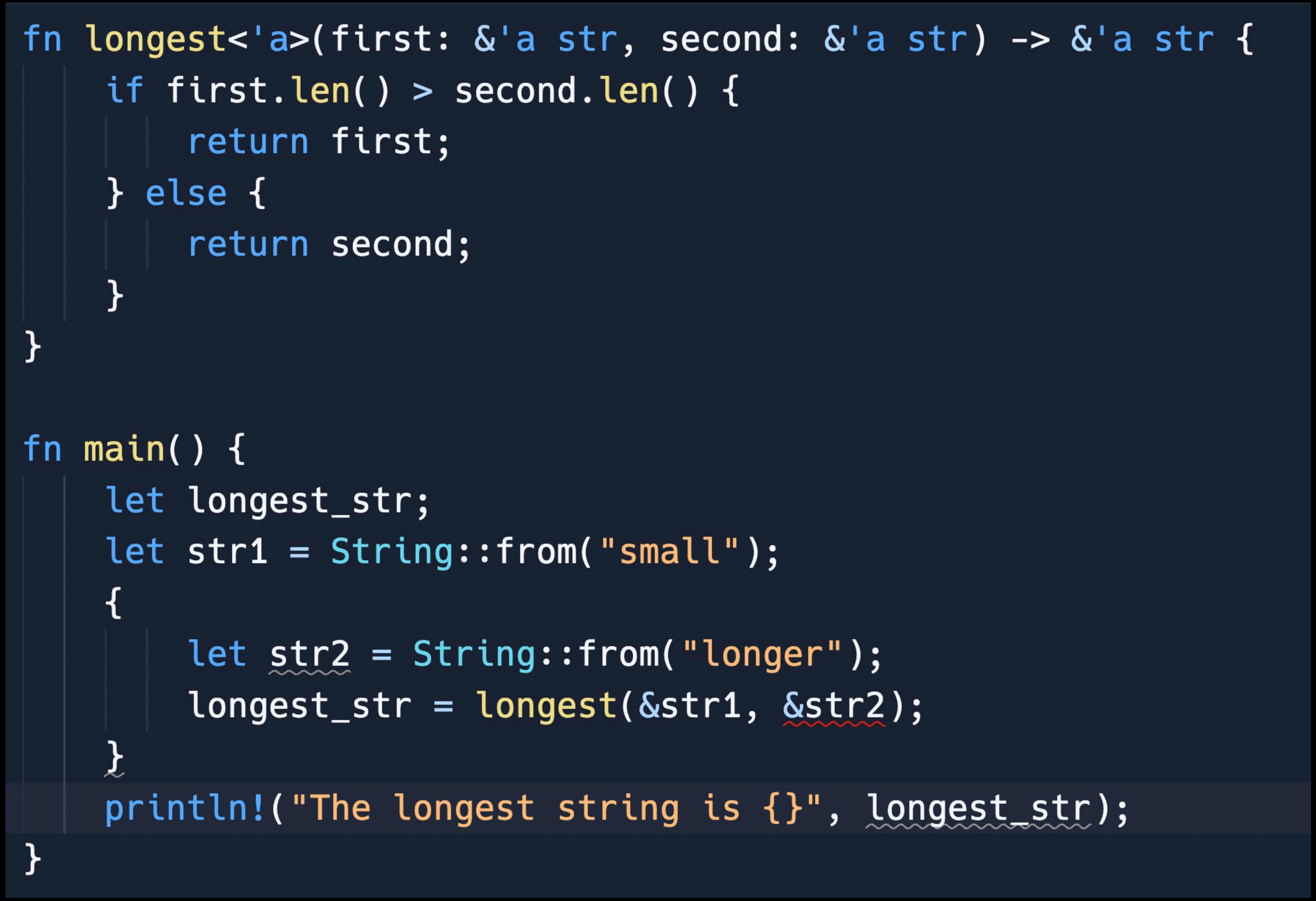
How to fix the error?



```
   Compiling my-project v0.1.0 (/home/runner/SparklingRichRuby)
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(a: &str, b: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signatu
re does not say whether it is borrowed from `a` or `b`
help: consider introducing a named lifetime parameter
  |
1 | fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```

Lifetimes

How to fix the error? - Specify lifetimes

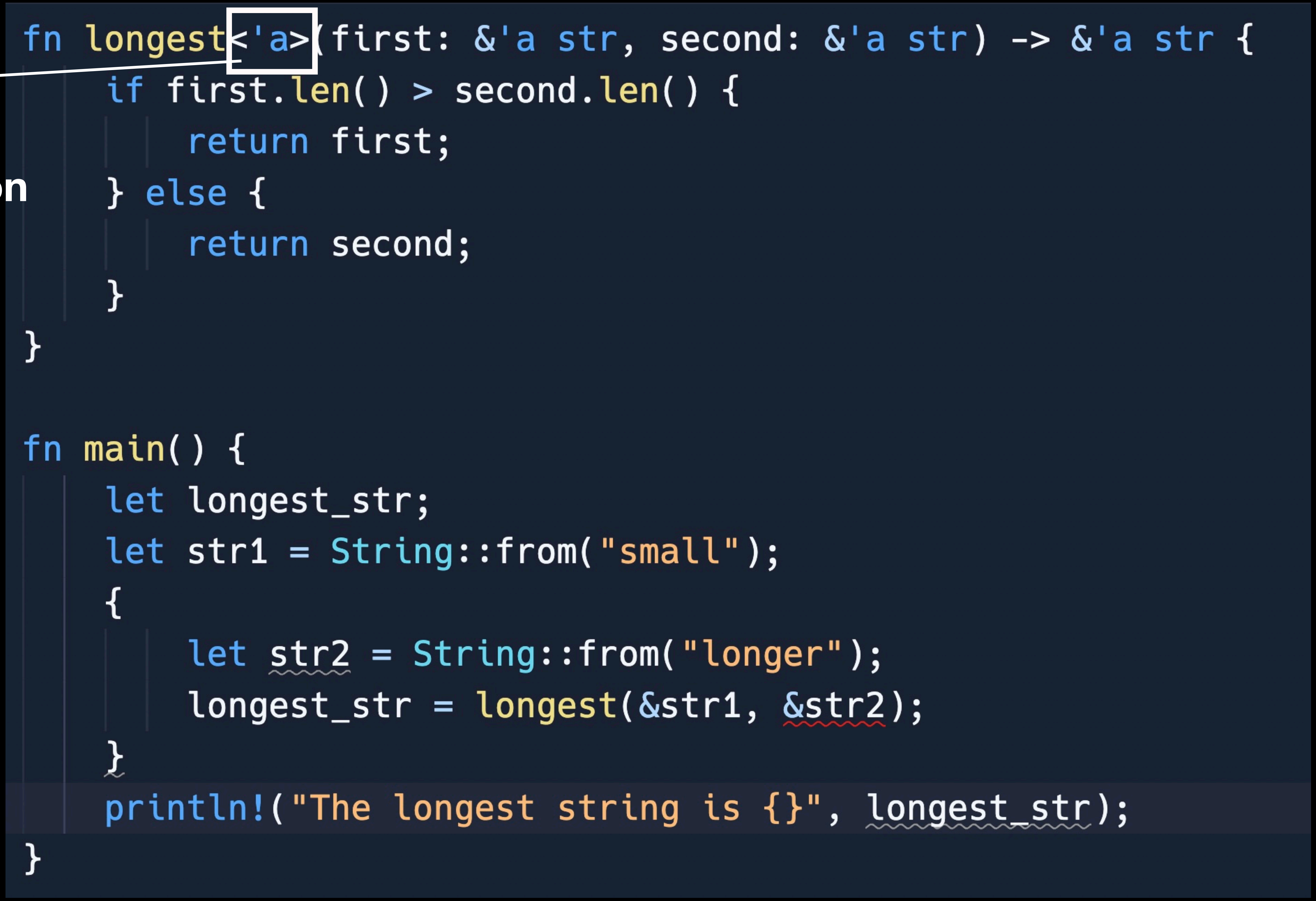


```rust
fn longest<'a>(first: &'a str, second: &'a str) -> &'a str {
    if first.len() > second.len() {
        return first;
    } else {
        return second;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(&str1, &str2);
    }
    println!("The longest string is {}", longest_str);
}
```

Lifetimes

How to fix the error? - Specify lifetimes

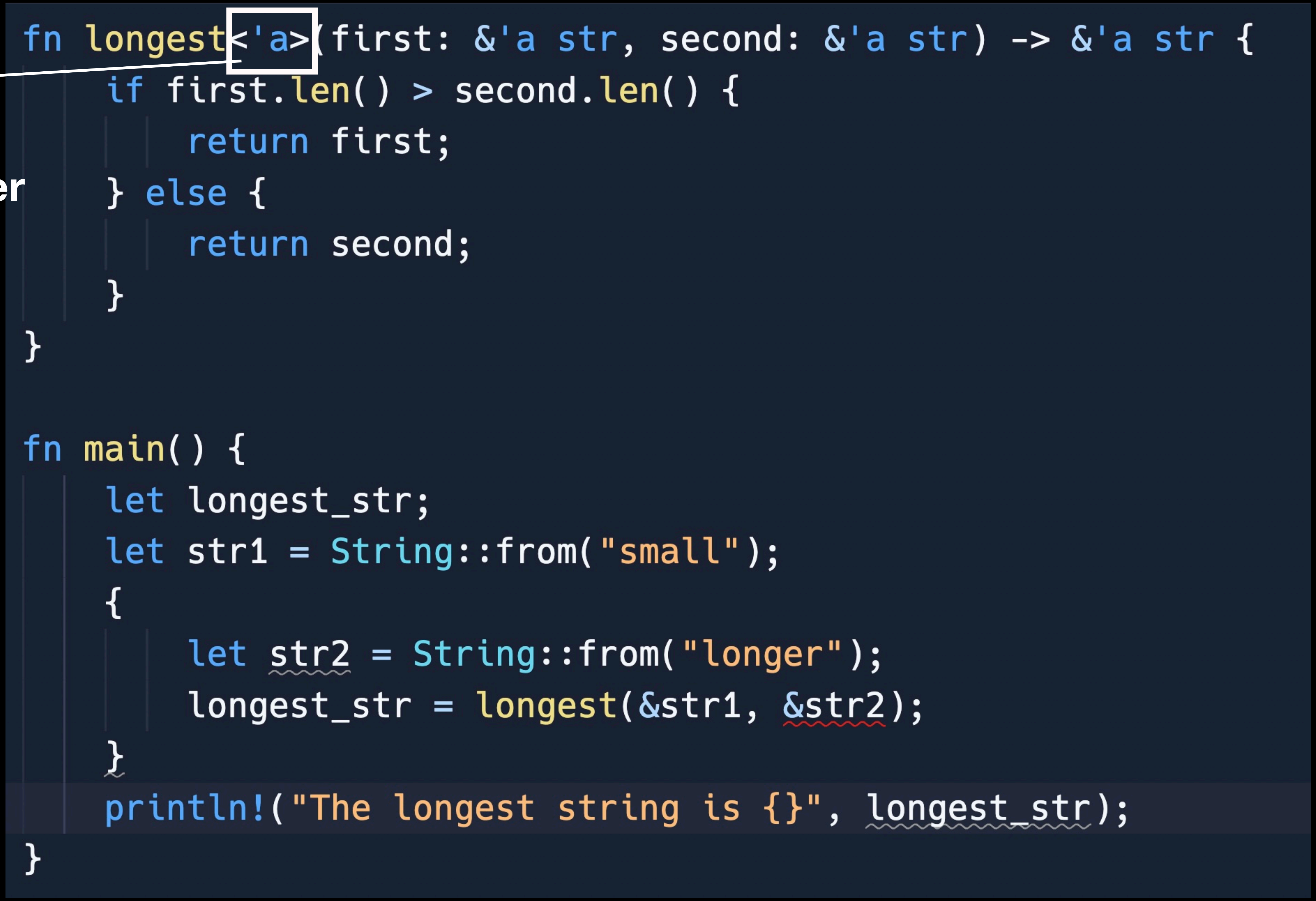
Very similar to a generic
Called a generic lifetime annotation



```rust
fn longest<'a>(first: &'a str, second: &'a str) -> &'a str {
    if first.len() > second.len() {
        return first;
    } else {
        return second;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(&str1, &str2);
    }
    println!("The longest string is {}", longest_str);
}
```

Lifetimes

How to fix the error? - Specify lifetimes

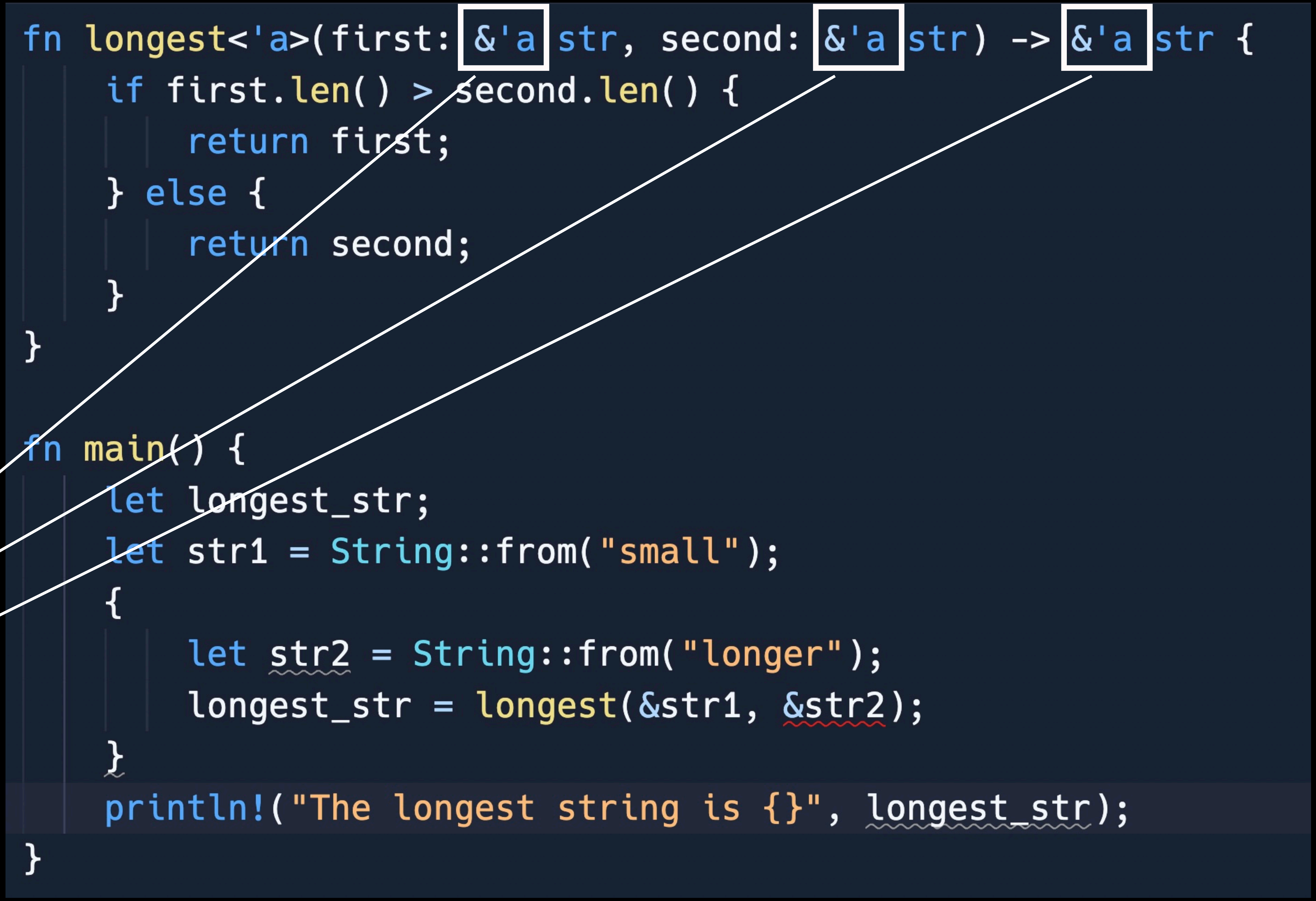


```
fn longest<'a>(first: &'a str, second: &'a str) -> &'a str {
    if first.len() > second.len() {
        return first;
    } else {
        return second;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(&str1, &str2);
    }
    println!("The longest string is {}", longest_str);
}
```

Very similar to a generic
Called a generic lifetime parameter

Lifetimes

How to fix the error? - Specify lifetimes

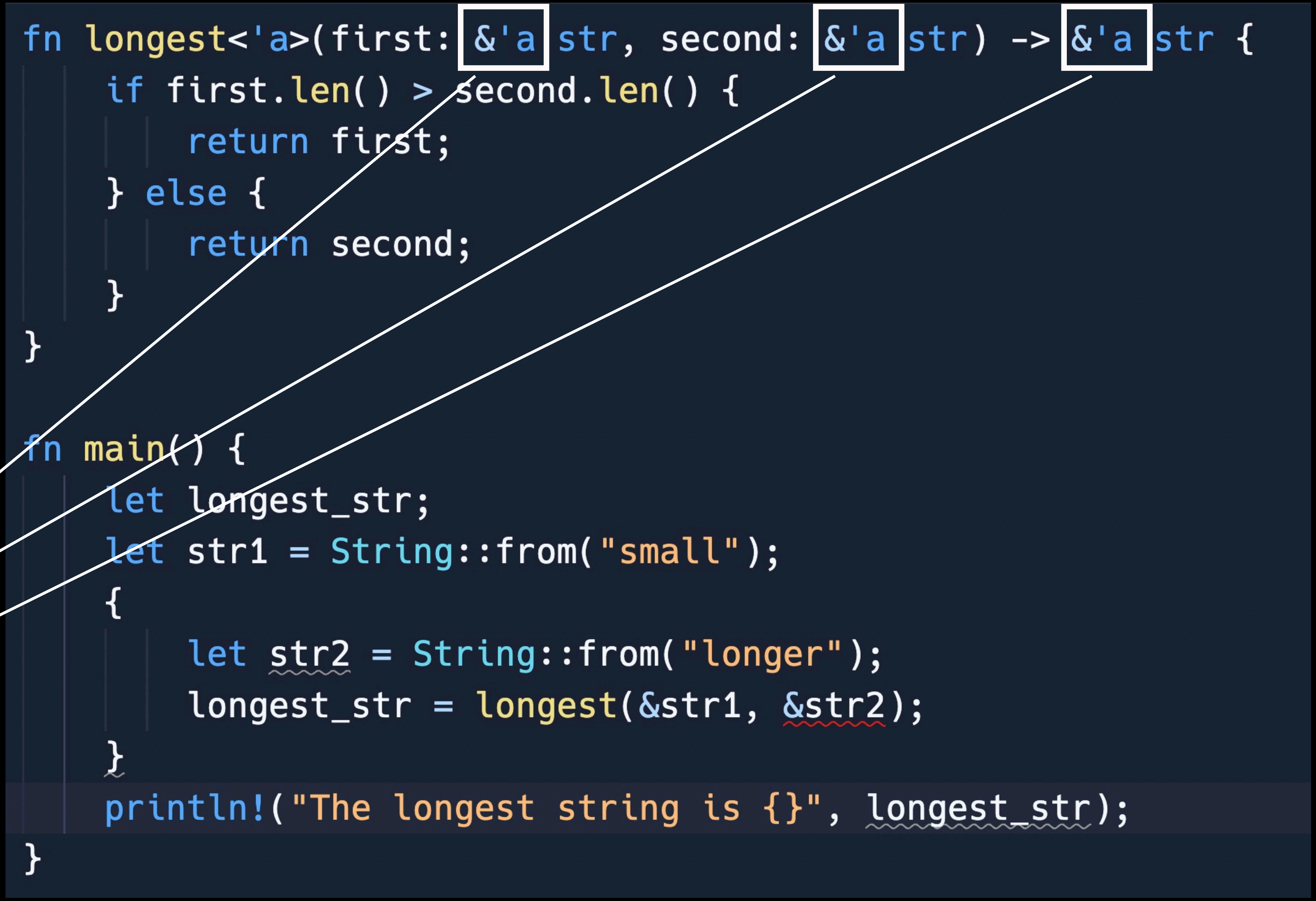


```
fn longest<'a>(first: &'a str, second: &'a str) -> &'a str {
    if first.len() > second.len() {
        return first;
    } else {
        return second;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(&str1, &str2);
    }
    println!("The longest string is {}", longest_str);
}
```

Describe a relationship b/w the lifetimes of input args and output args

Lifetimes

How to fix the error? - Specify lifetimes



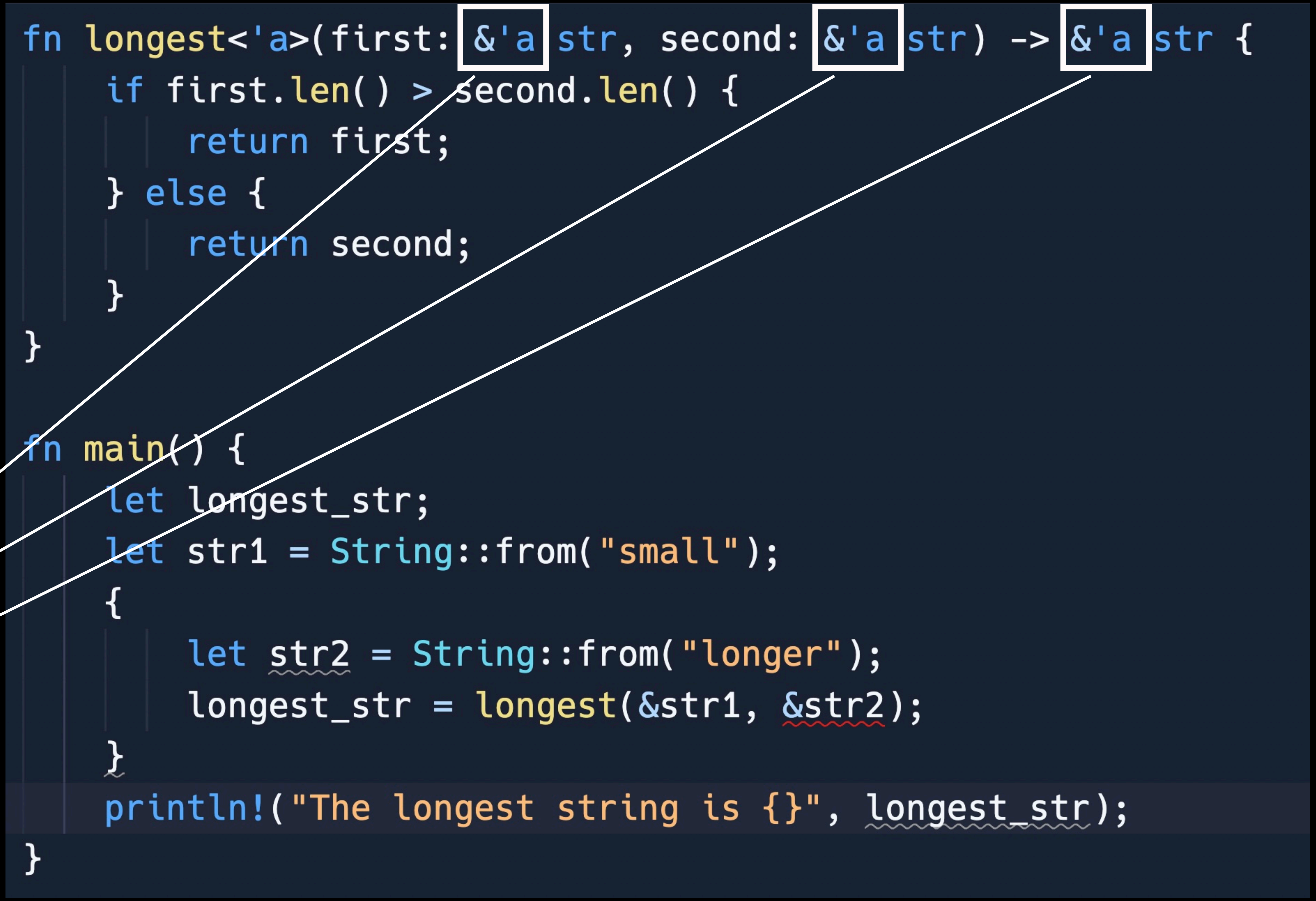
```rust
fn longest<'a>(first: &'a str, second: &'a str) -> &'a str {
    if first.len() > second.len() {
        return first;
    } else {
        return second;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(&str1, &str2);
    }
    println!("The longest string is {}", longest_str);
}
```

Describe a relationship b/w the lifetimes of input args and output args

It says that the `return type` will be valid as long as both the arguments are valid

Lifetimes

How to fix the error? - Specify lifetimes



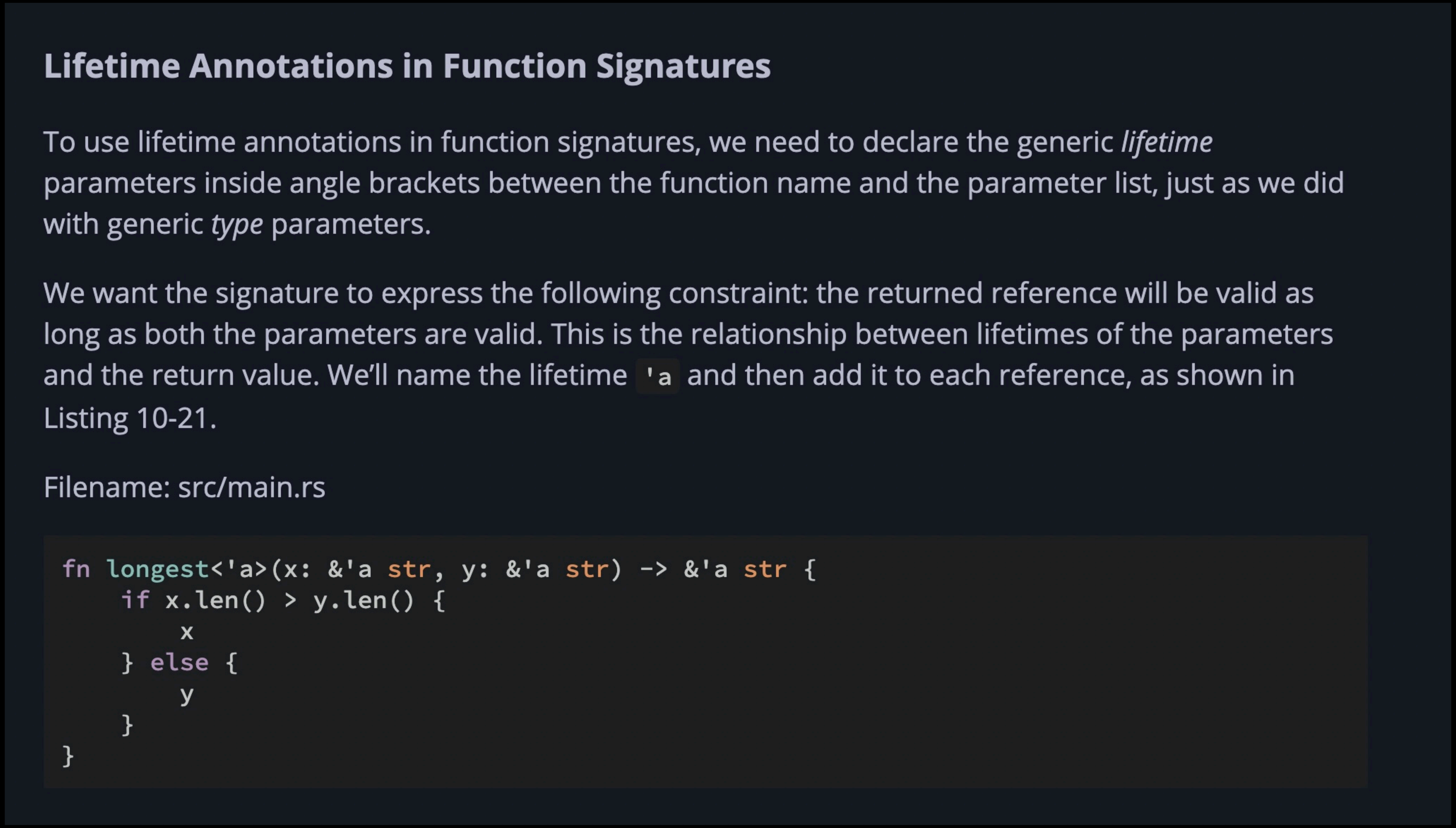
```
fn longest<'a>(first: &'a str, second: &'a str) -> &'a str {
    if first.len() > second.len() {
        return first;
    } else {
        return second;
    }
}

fn main() {
    let longest_str;
    let str1 = String::from("small");
    {
        let str2 = String::from("longer");
        longest_str = longest(&str1, &str2);
    }
    println!("The longest string is {}", longest_str);
}
```

Describe a relationship b/w the lifetimes of input args and output args

It says that the `return type` will be valid as long as both the arguments are valid

Or more technically, the shorter lifetimes is what the return type will have

Lifetimes



Lifetime Annotations in Function Signatures

To use lifetime annotations in function signatures, we need to declare the generic *lifetime* parameters inside angle brackets between the function name and the parameter list, just as we did with generic *type* parameters.

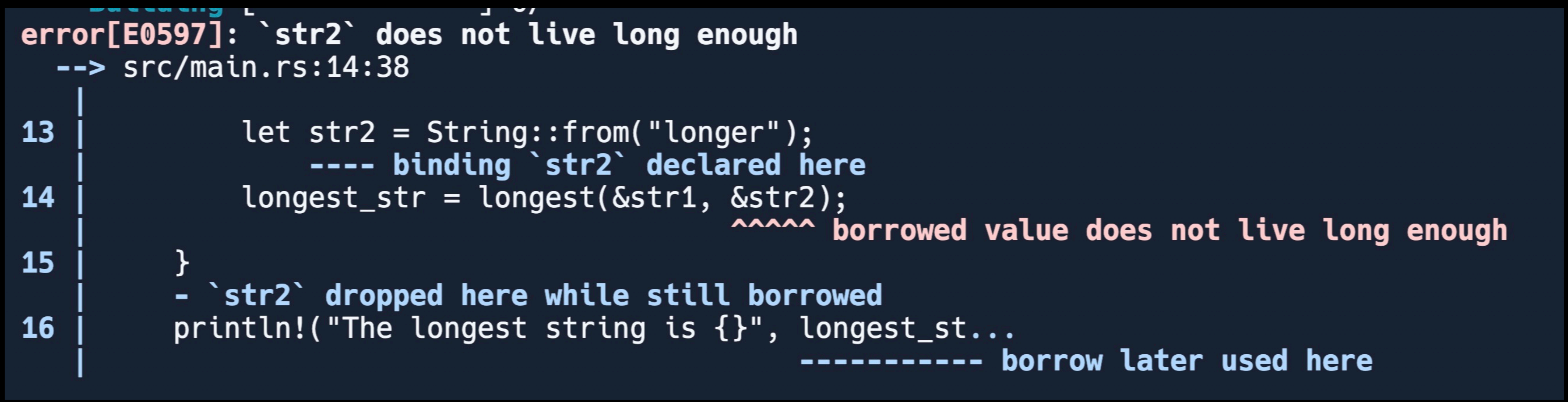
We want the signature to express the following constraint: the returned reference will be valid as long as both the parameters are valid. This is the relationship between lifetimes of the parameters and the return value. We'll name the lifetime `'a` and then add it to each reference, as shown in Listing 10-21.

Filename: src/main.rs

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes

Now if you run this code, it fails with a better error

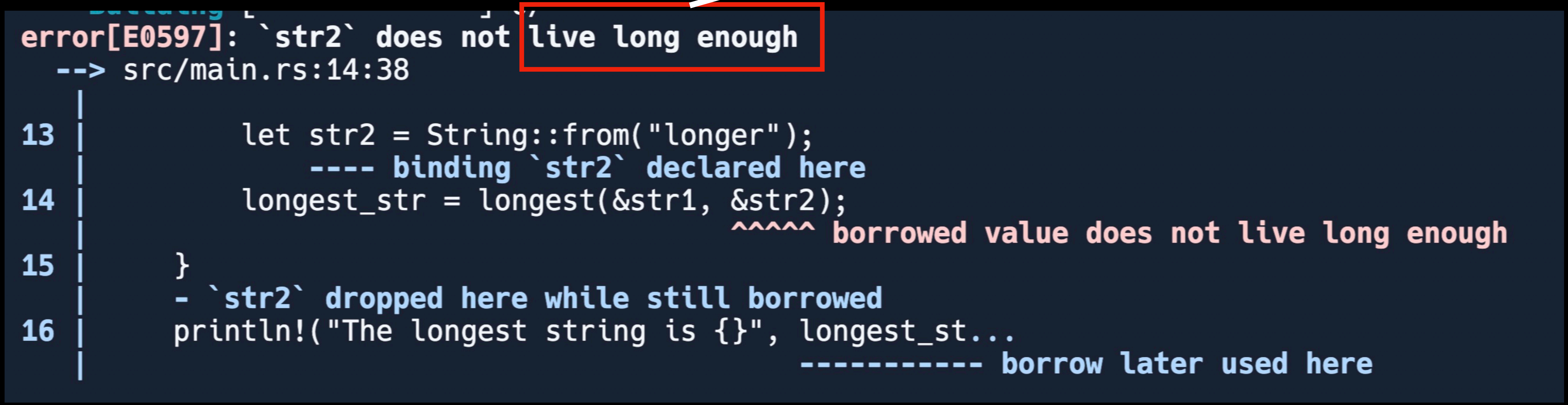


```
error[E0597]: `str2` does not live long enough
  --> src/main.rs:14:38
   |
13 |         let str2 = String::from("longer");
   |             ---- binding `str2` declared here
14 |         longest_str = longest(&str1, &str2);
   |                                      ^^^^^ borrowed value does not live long enough
15 |     }
   |     - `str2` dropped here while still borrowed
16 |     println!("The longest string is {}", longest_st...
   |                                          ----------- borrow later used here
```

Lifetimes

Now if you run this code,
it fails with a better error

Live long enough = lifetime of return value ends



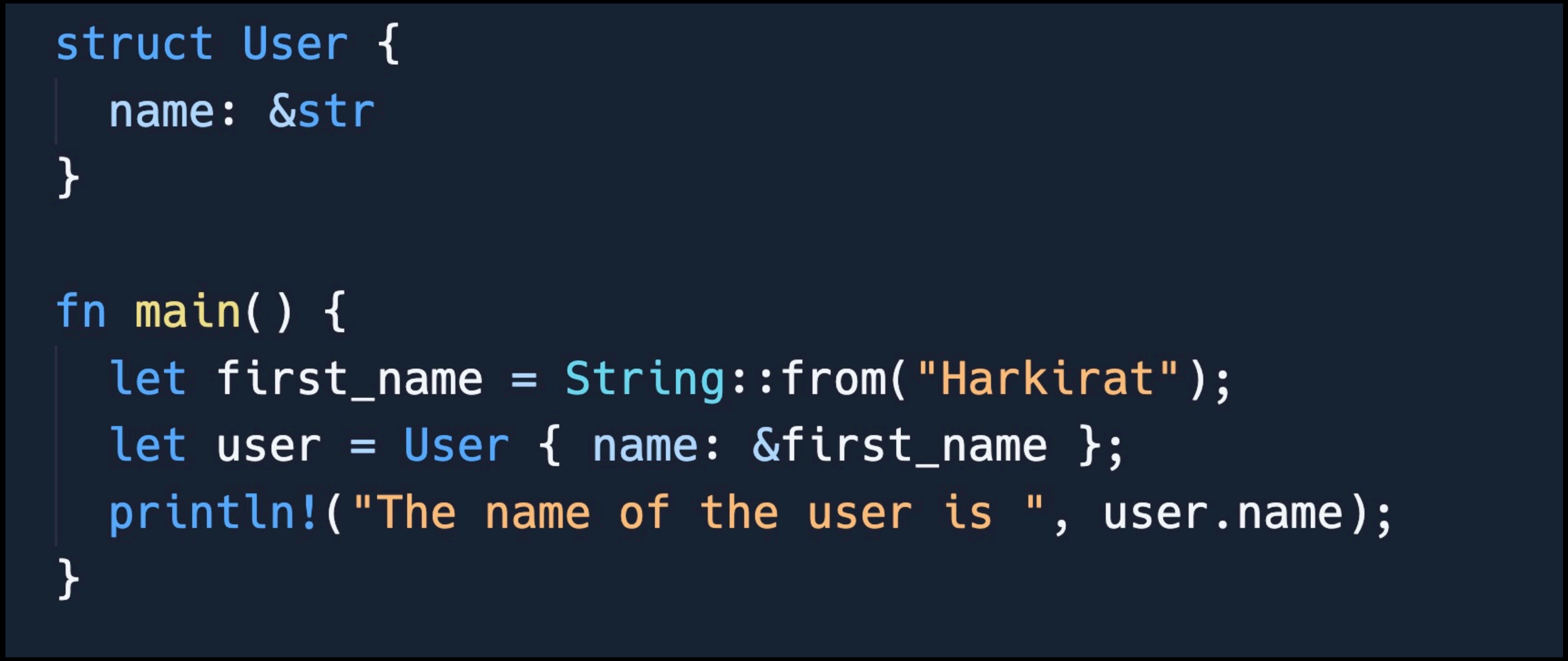
```
error[E0597]: `str2` does not live long enough
  --> src/main.rs:14:38
   |
13 |         let str2 = String::from("longer");
   |             ---- binding `str2` declared here
14 |         longest_str = longest(&str1, &str2);
   |                                      ^^^^^ borrowed value does not live long enough
15 |     }
   |     - `str2` dropped here while still borrowed
16 |     println!("The longest string is {}", longest_st...
   |                                          ----------- borrow later used here
```

Structs with lifetimes

Until now, we haven’t used references inside a struct
Lets try that

Structs with lifetimes

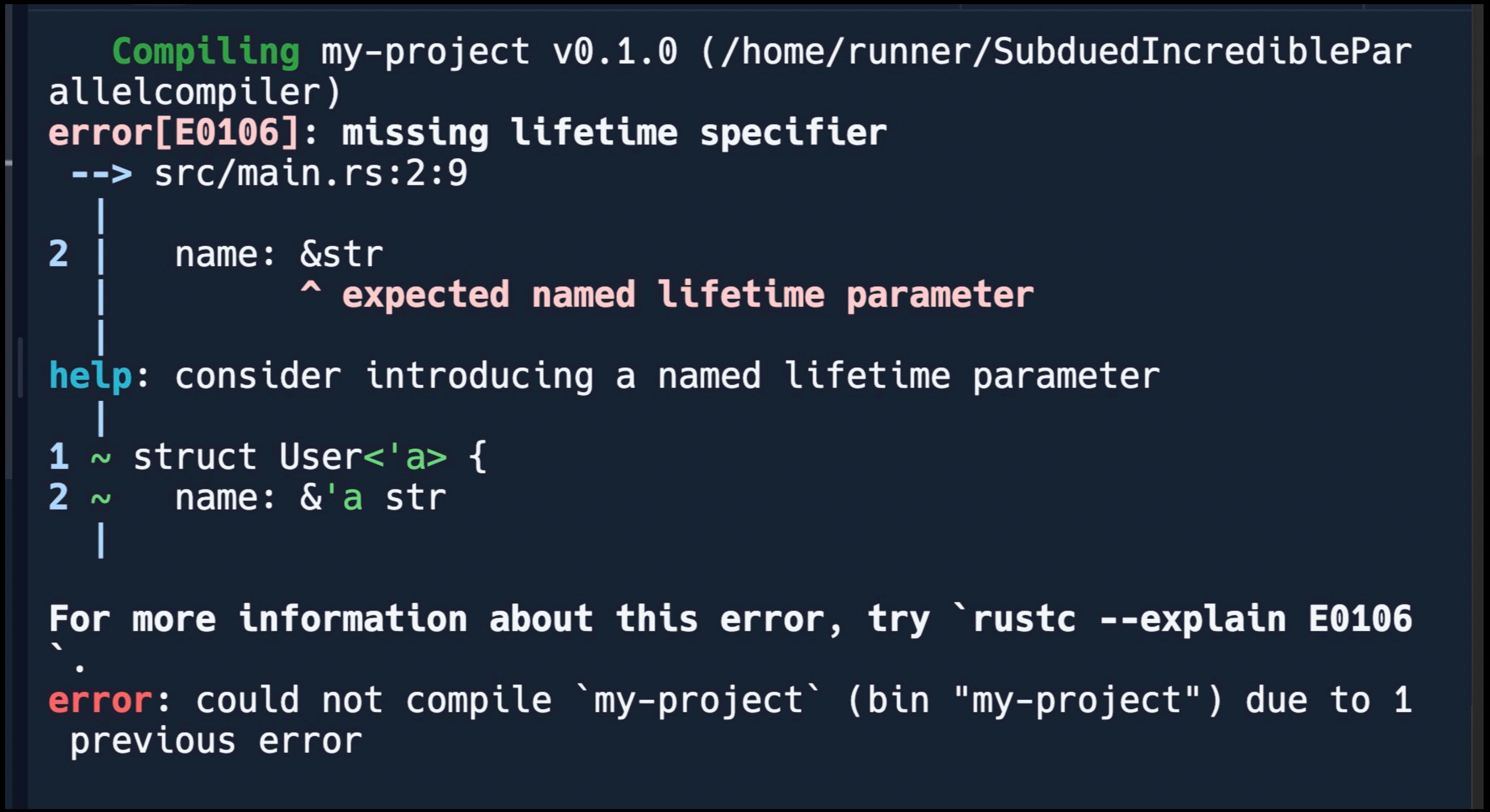
Until now, we haven't used references inside a struct
Lets try that



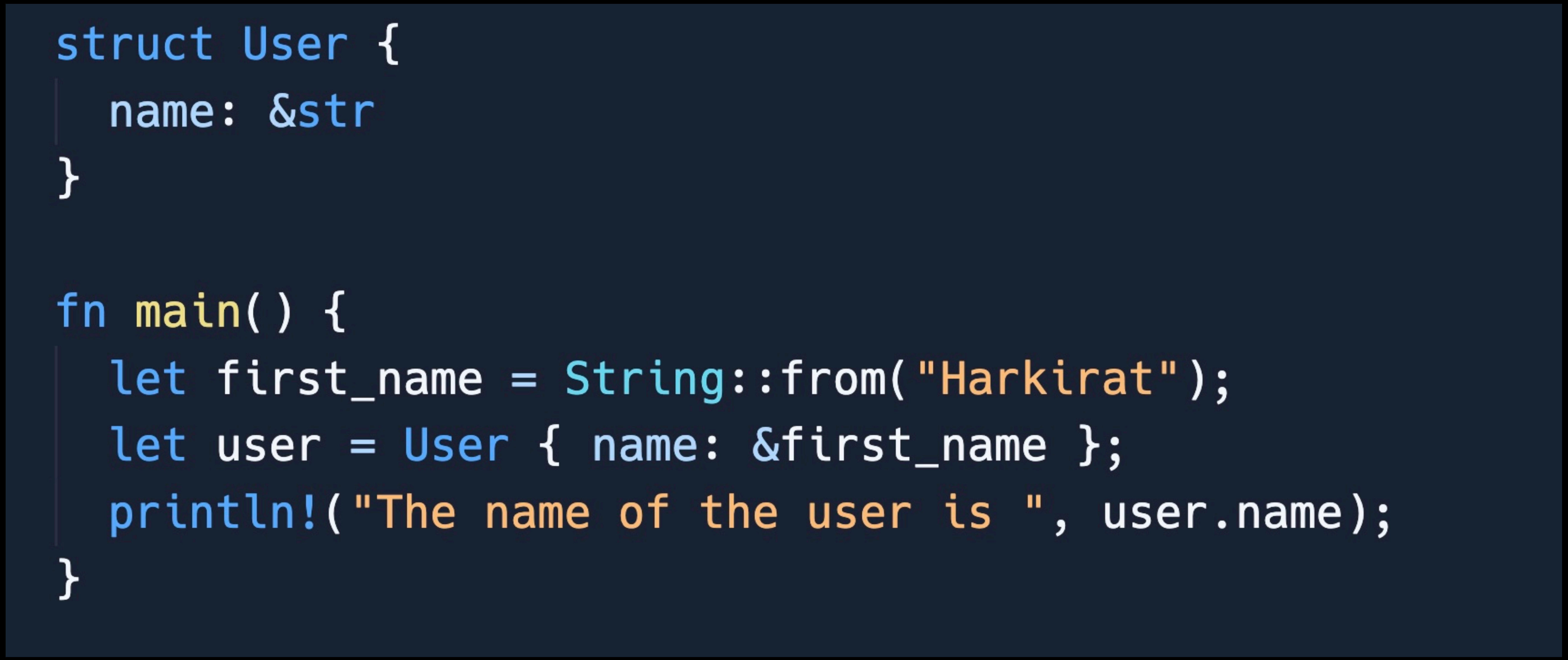
```rust
struct User {
  name: &str
}

fn main() {
  let first_name = String::from("Harkirat");
  let user = User { name: &first_name };
  println!("The name of the user is ", user.name);
}
```

Structs with lifetimes

Until now, we haven’t used references inside a struct
Lets try that



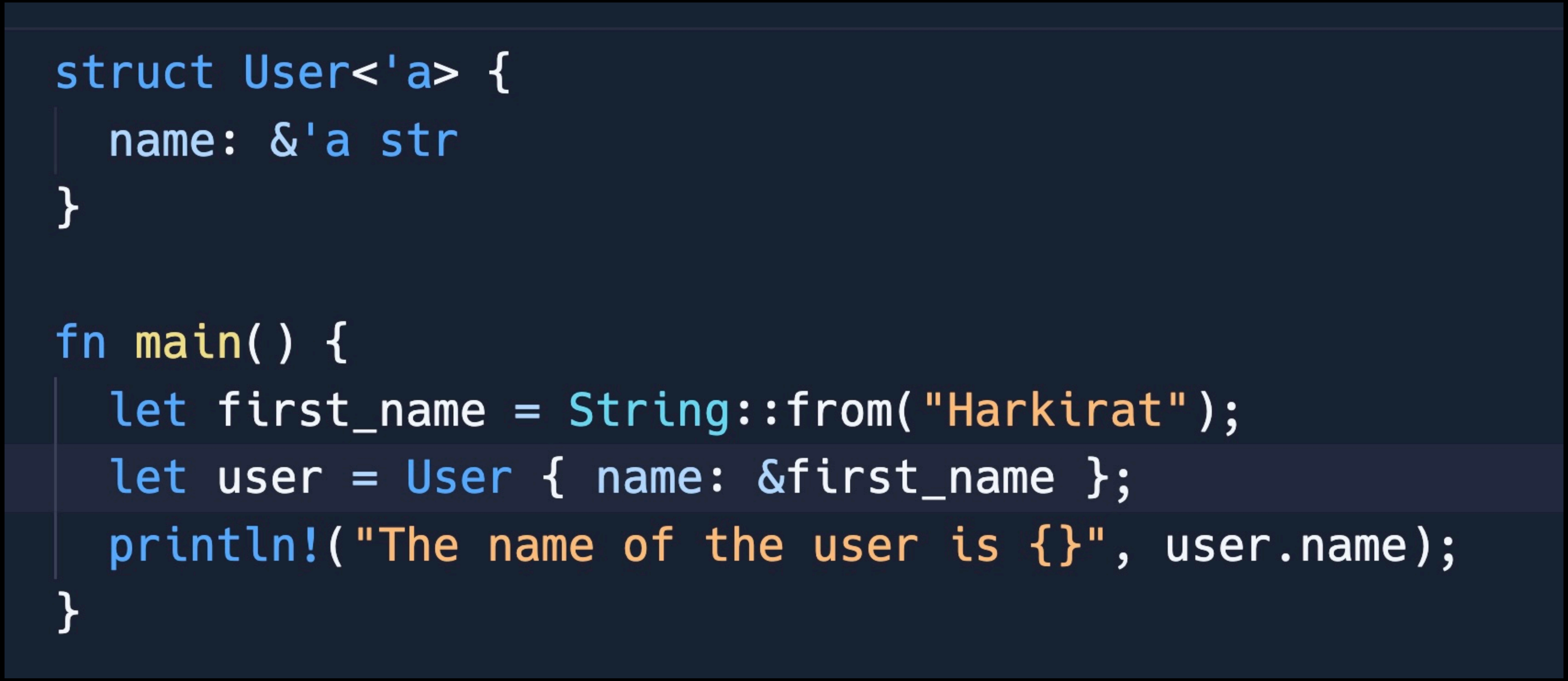
```
   Compiling my-project v0.1.0 (/home/runner/SubduedIncrediblePar
allelcompiler)
error[E0106]: missing lifetime specifier
 --> src/main.rs:2:9
  |
2 |   name: &str
  |         ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 ~   name: &'a str
  |

For more information about this error, try `rustc --explain E0106
`.
error: could not compile `my-project` (bin "my-project") due to 1
 previous error
```



```rust
struct User {
  name: &str
}

fn main() {
  let first_name = String::from("Harkirat");
  let user = User { name: &first_name };
  println!("The name of the user is ", user.name);
}
```

Structs with lifetimes

Until now, we haven’t used references inside a struct
Lets try that



```
struct User<'a> {
  name: &'a str
}

fn main() {
  let first_name = String::from("Harkirat");
  let user = User { name: &first_name };
  println!("The name of the user is {}", user.name);
}
```

Structs with lifetimes

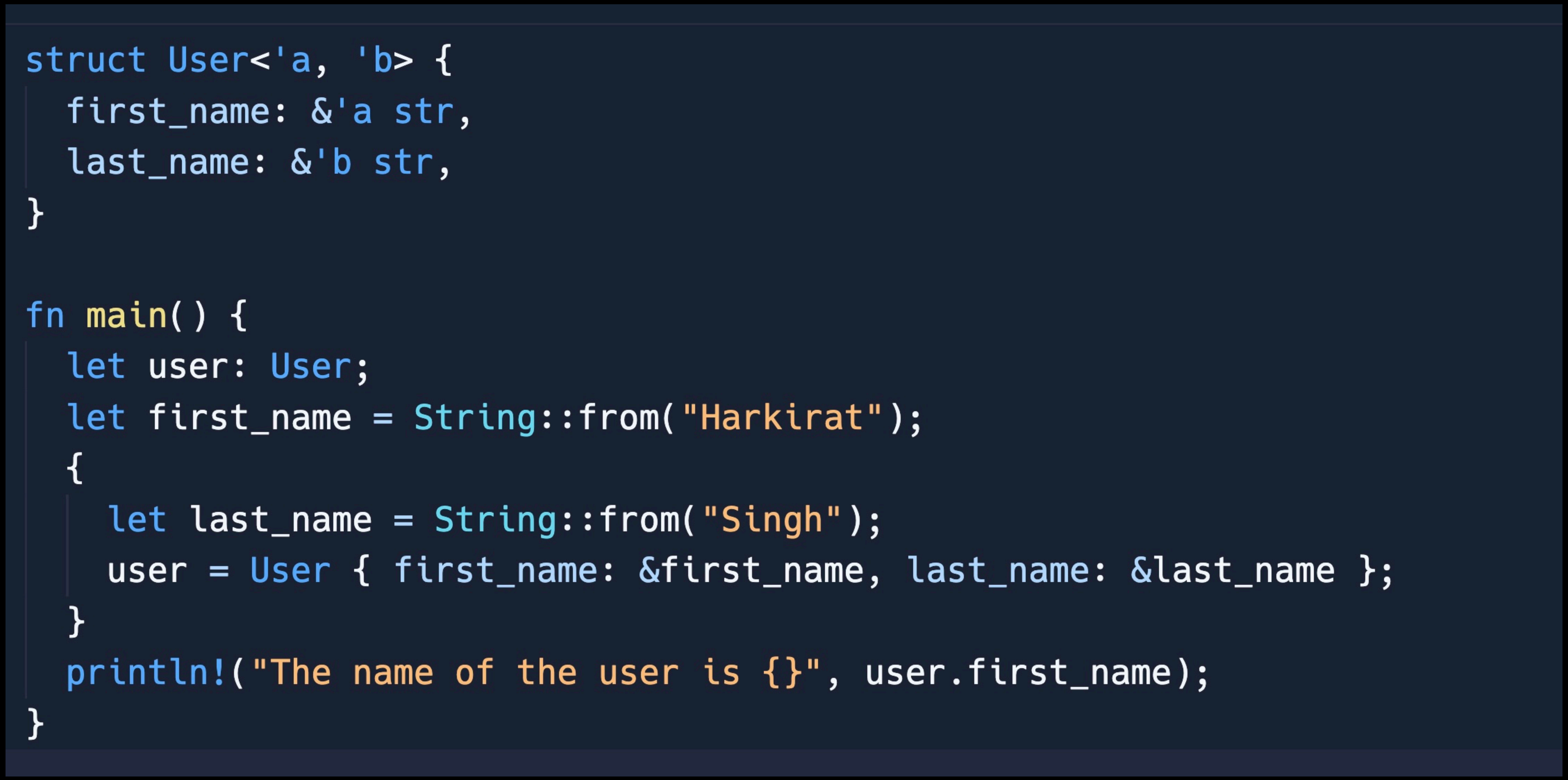
Why do you need structs with references to have a lifetime parameter?

Structs with lifetimes

Why do you need structs with references to have a lifetime parameter?

So we know how long the `struct` can live

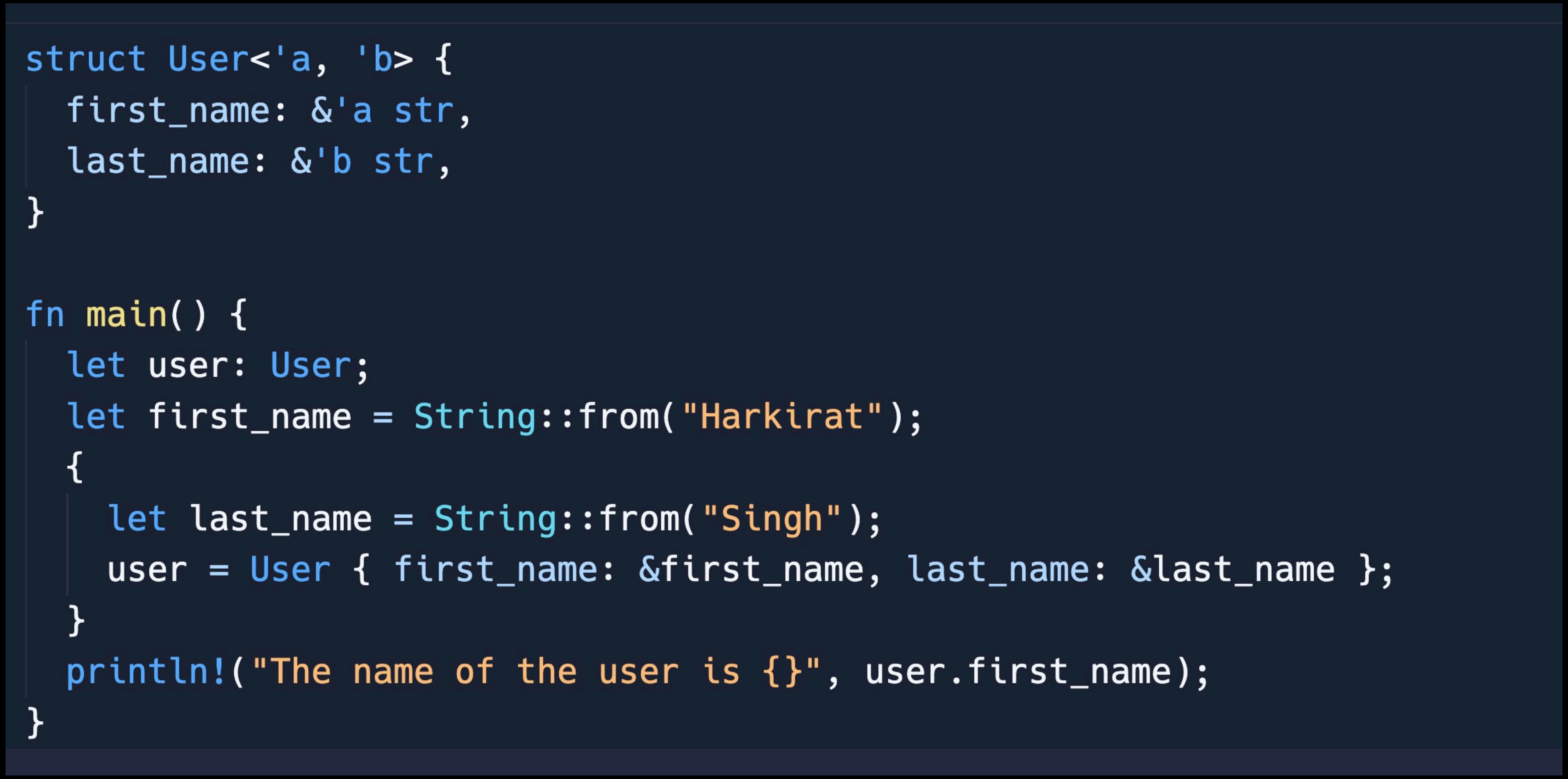
Structs with lifetimes



```rust
struct User<'a, 'b> {
  first_name: &'a str,
  last_name: &'b str,
}

fn main() {
  let user: User;
  let first_name = String::from("Harkirat");
  {
    let last_name = String::from("Singh");
    user = User { first_name: &first_name, last_name: &last_name };
  }
  println!("The name of the user is {}", user.first_name);
}
```

Structs with lifetimes

This code doesn’t compile
Because

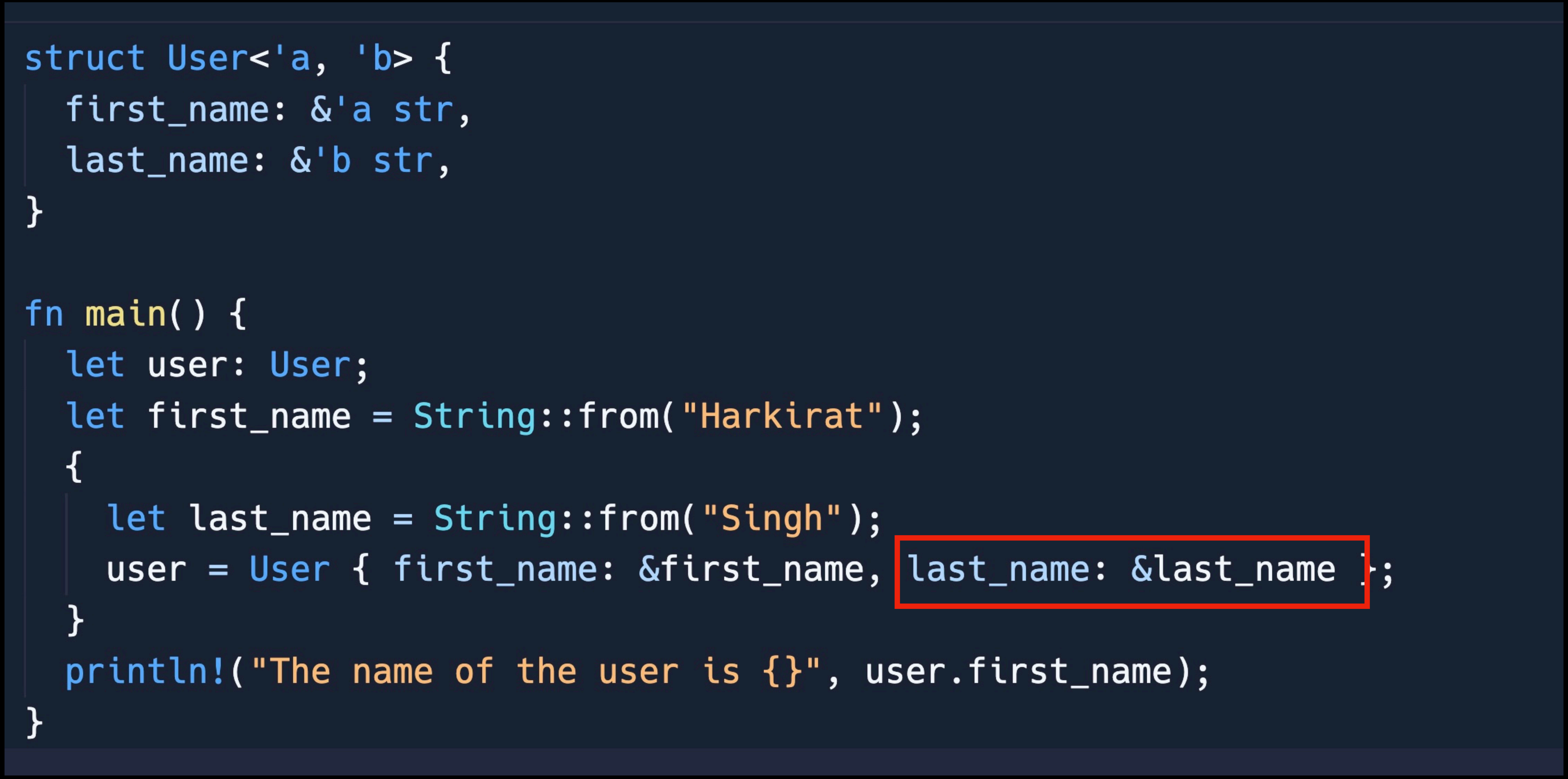


```rust
struct User<'a, 'b> {
  first_name: &'a str,
  last_name: &'b str,
}

fn main() {
  let user: User;
  let first_name = String::from("Harkirat");
  {
    let last_name = String::from("Singh");
    user = User { first_name: &first_name, last_name: &last_name };
  }
  println!("The name of the user is {}", user.first_name);
}
```

Structs with lifetimes

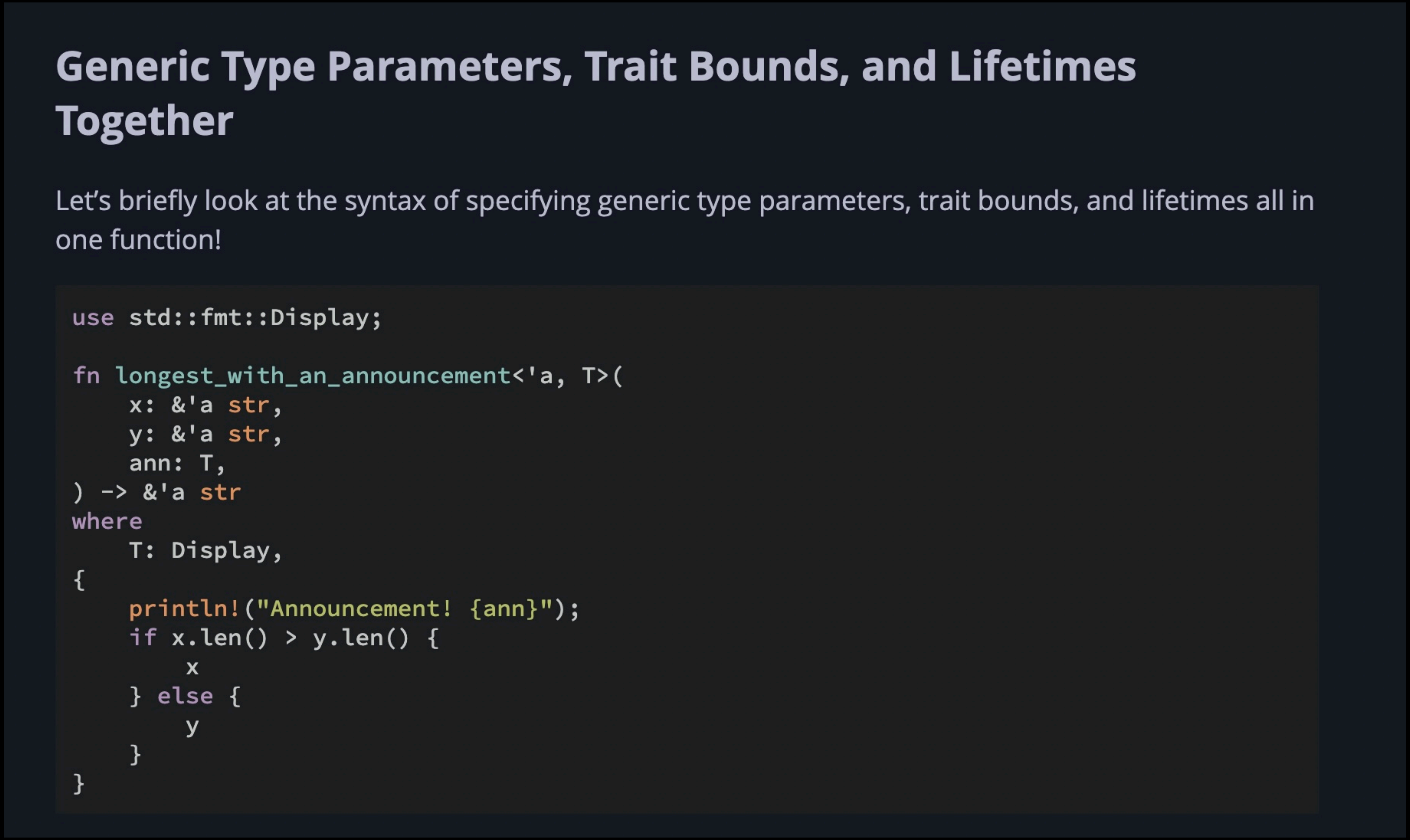
This code doesn't compile
Because

last_name doesn't live long enough



```
struct User<'a, 'b> {
  first_name: &'a str,
  last_name: &'b str,
}

fn main() {
  let user: User;
  let first_name = String::from("Harkirat");
  {
    let last_name = String::from("Singh");
    user = User { first_name: &first_name, last_name: &last_name };
  }
  println!("The name of the user is {}", user.first_name);
}
```

Structs with lifetimes

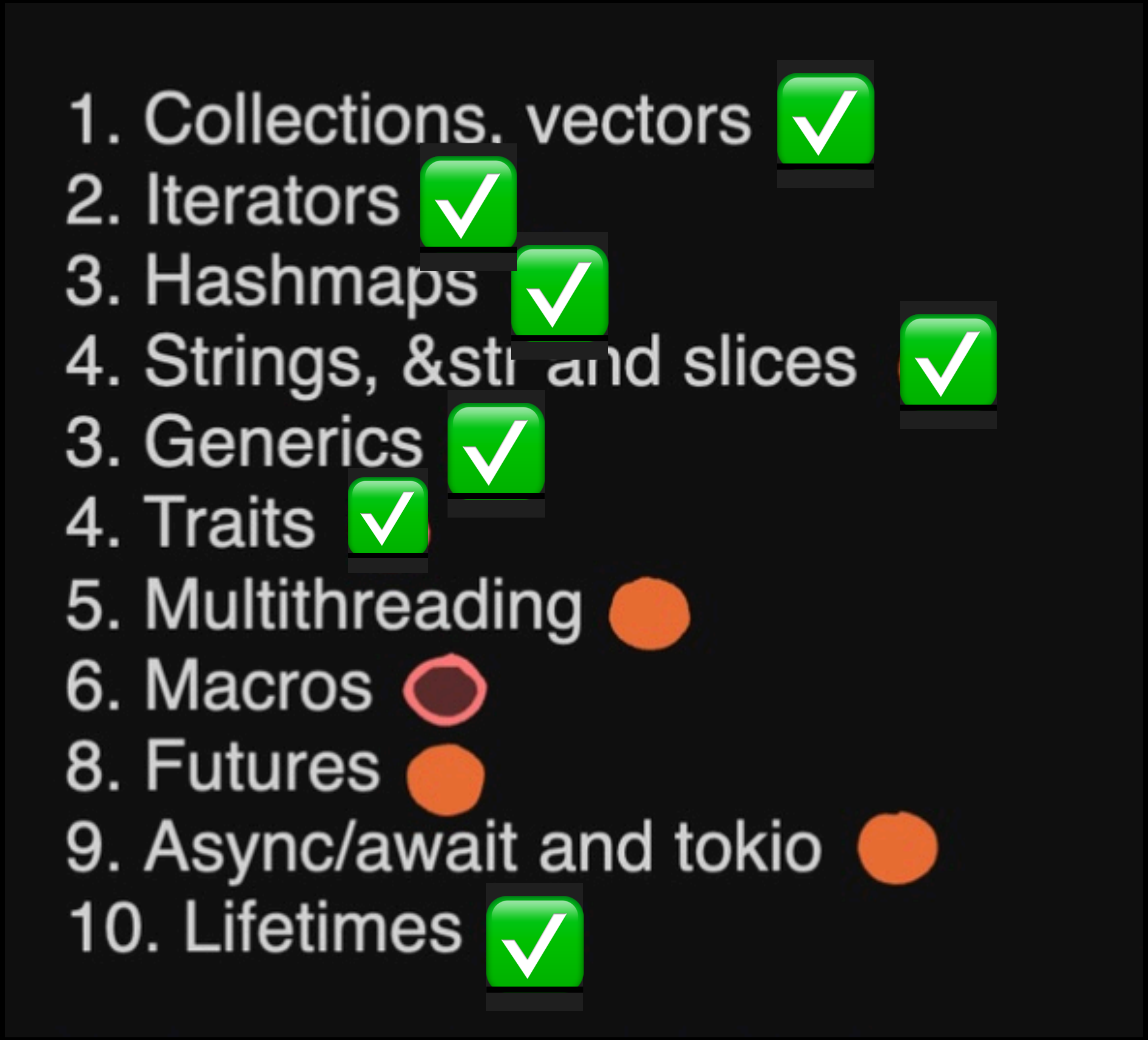


Generic Type Parameters, Trait Bounds, and Lifetimes Together

Let’s briefly look at the syntax of specifying generic type parameters, trait bounds, and lifetimes all in one function!

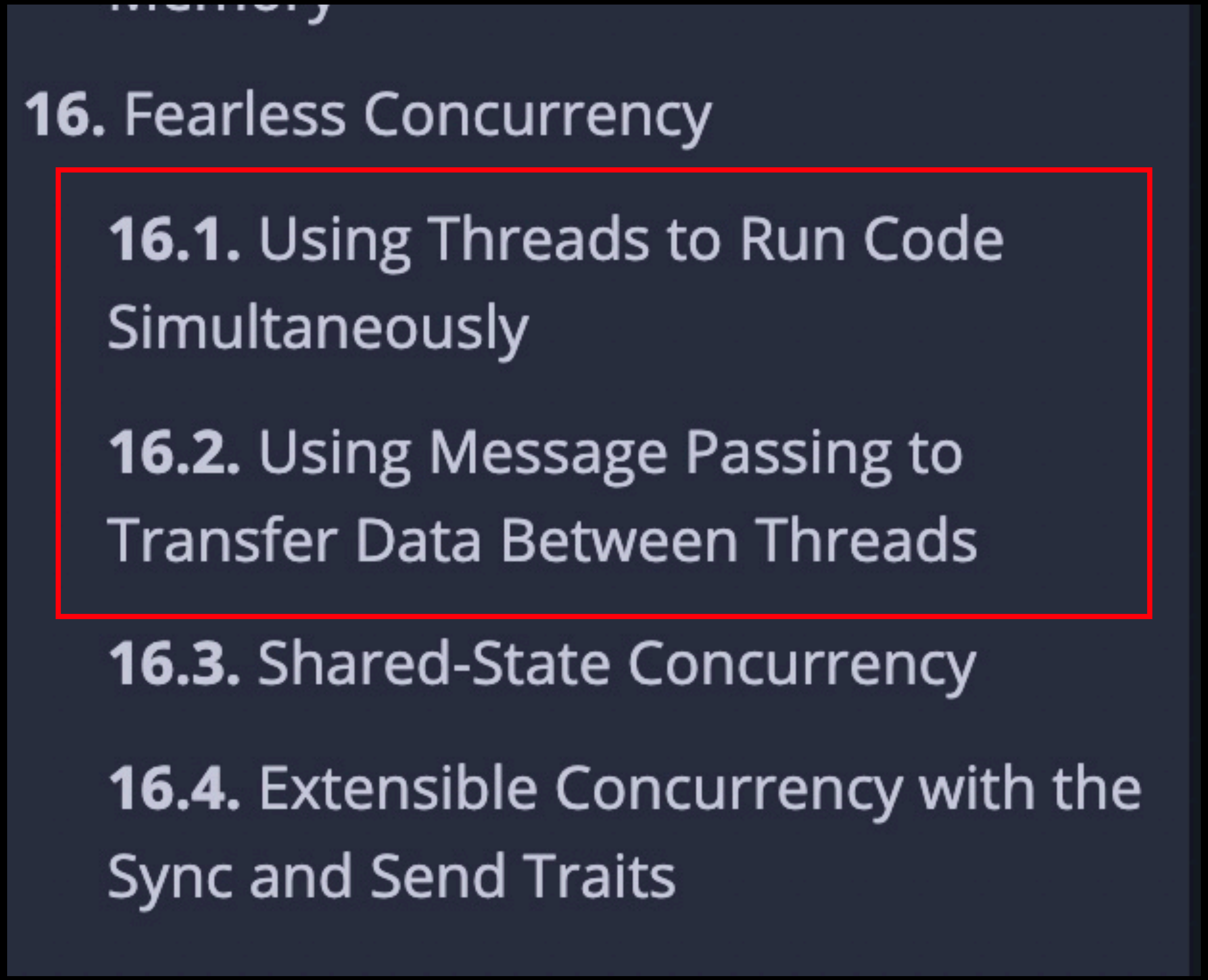
```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {ann}");
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Checkpoint



1. Collections, vectors ✅
2. Iterators ✅
3. Hashmaps ✅
4. Strings, &str and slices ✅
3. Generics ✅
4. Traits ✅
5. Multithreading ●
6. Macros ●
8. Futures ●
9. Async/await and tokio ●
10. Lifetimes ✅

Multithreading



Memory

16. Fearless Concurrency

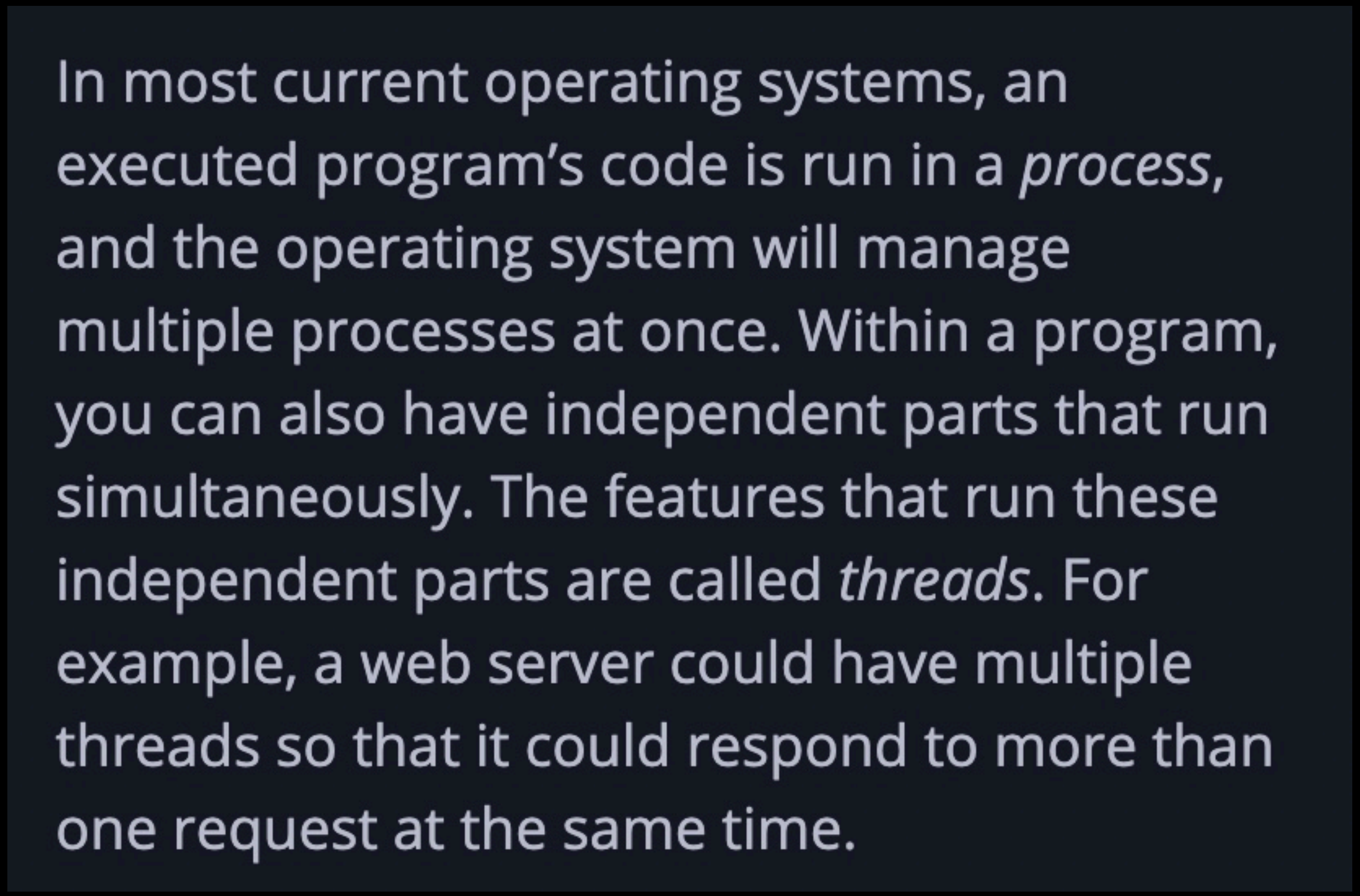
16.1. Using Threads to Run Code Simultaneously

16.2. Using Message Passing to Transfer Data Between Threads

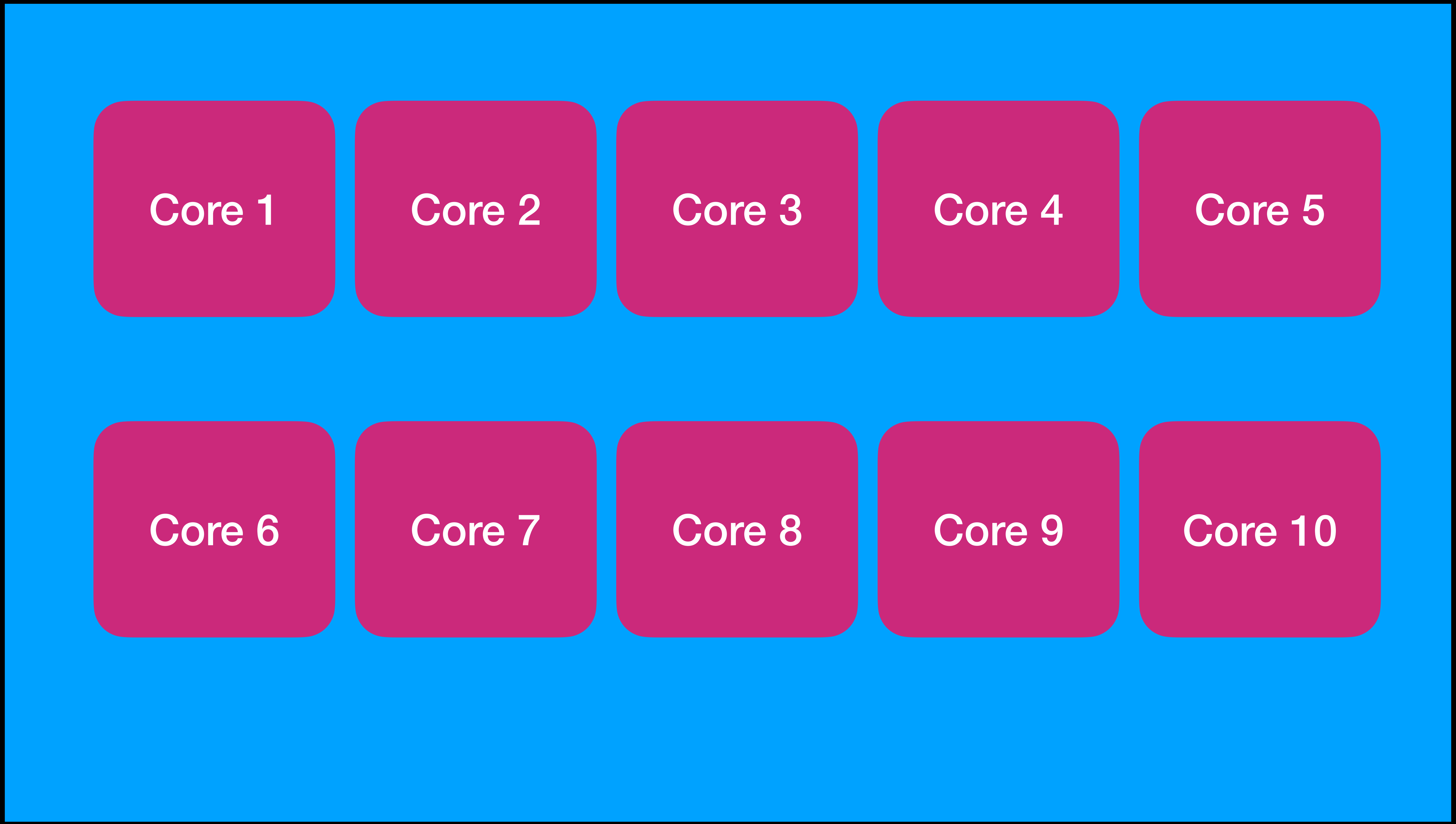
16.3. Shared-State Concurrency

16.4. Extensible Concurrency with the Sync and Send Traits

Multithreading

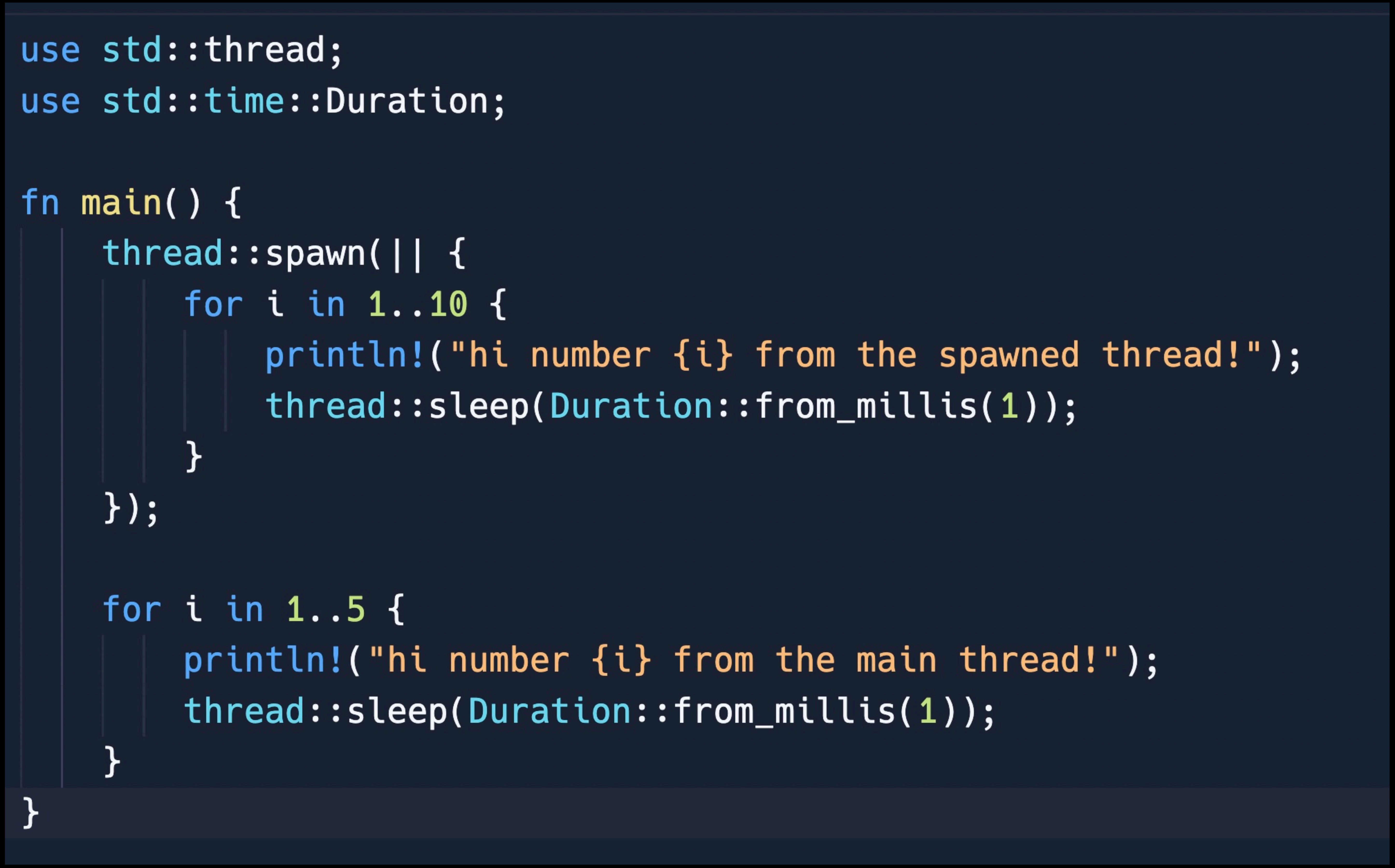


In most current operating systems, an executed program's code is run in a *process*, and the operating system will manage multiple processes at once. Within a program, you can also have independent parts that run simultaneously. The features that run these independent parts are called *threads*. For example, a web server could have multiple threads so that it could respond to more than one request at the same time.

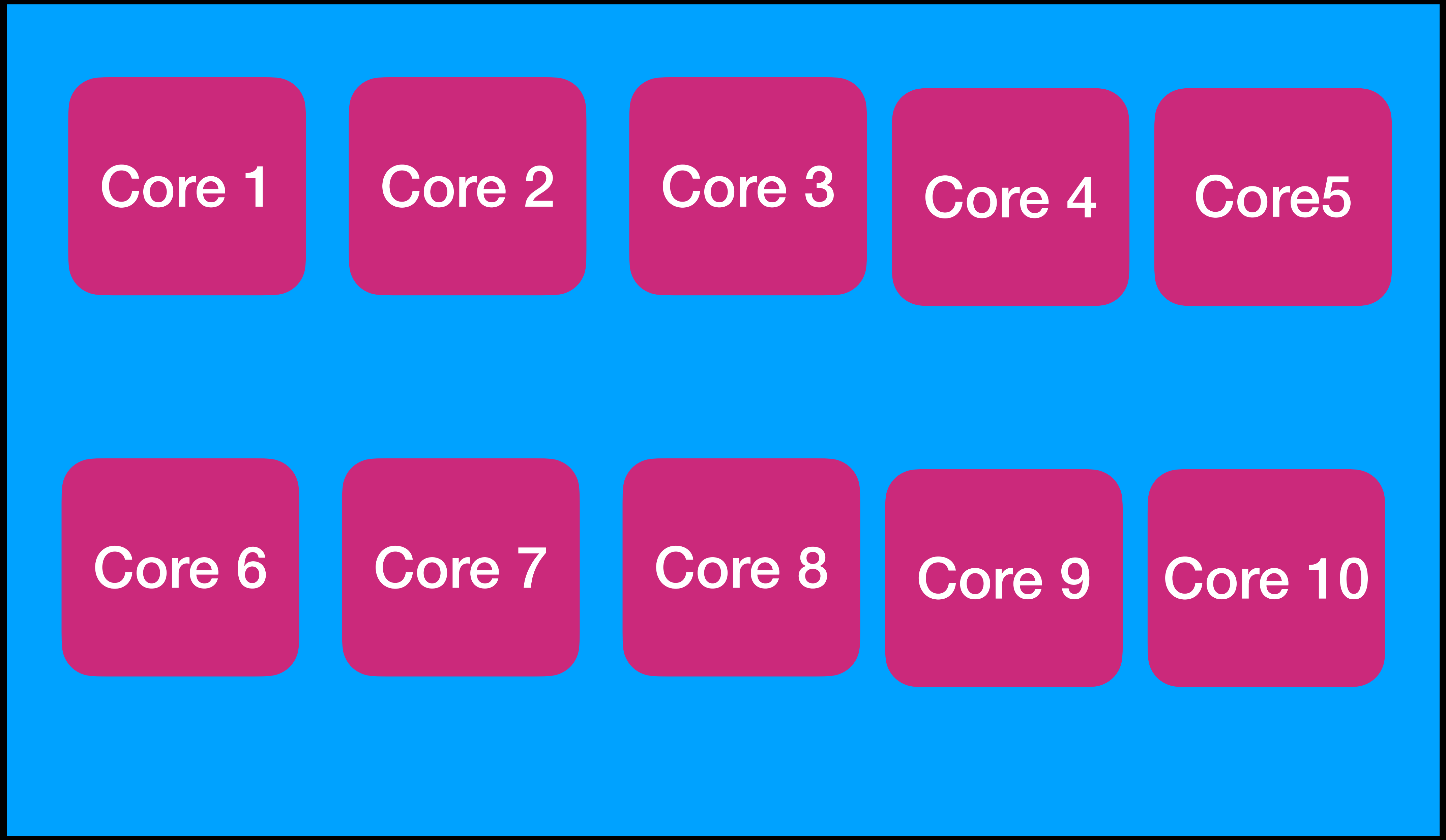


Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
Core 9
Core 10

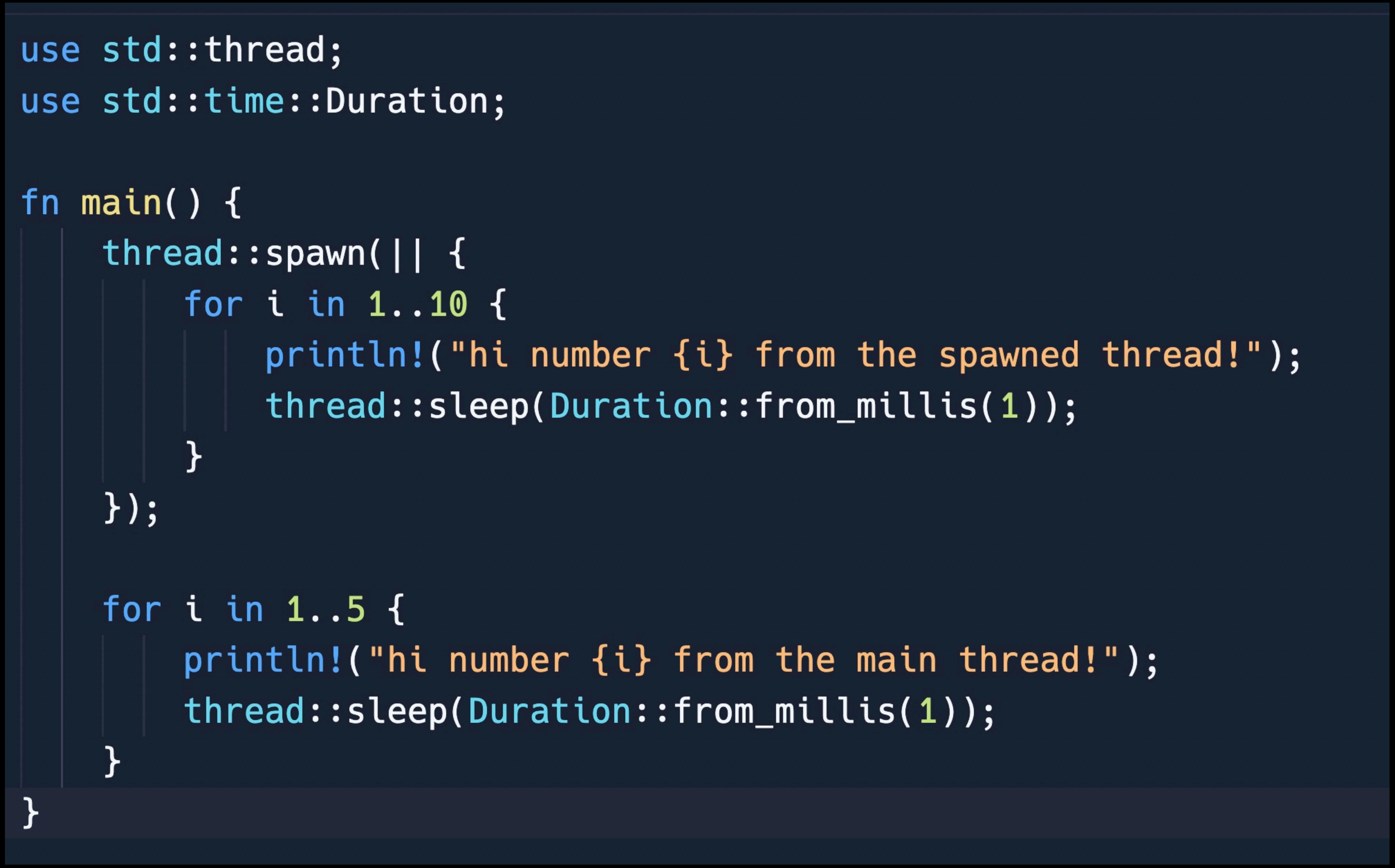
Multithreading



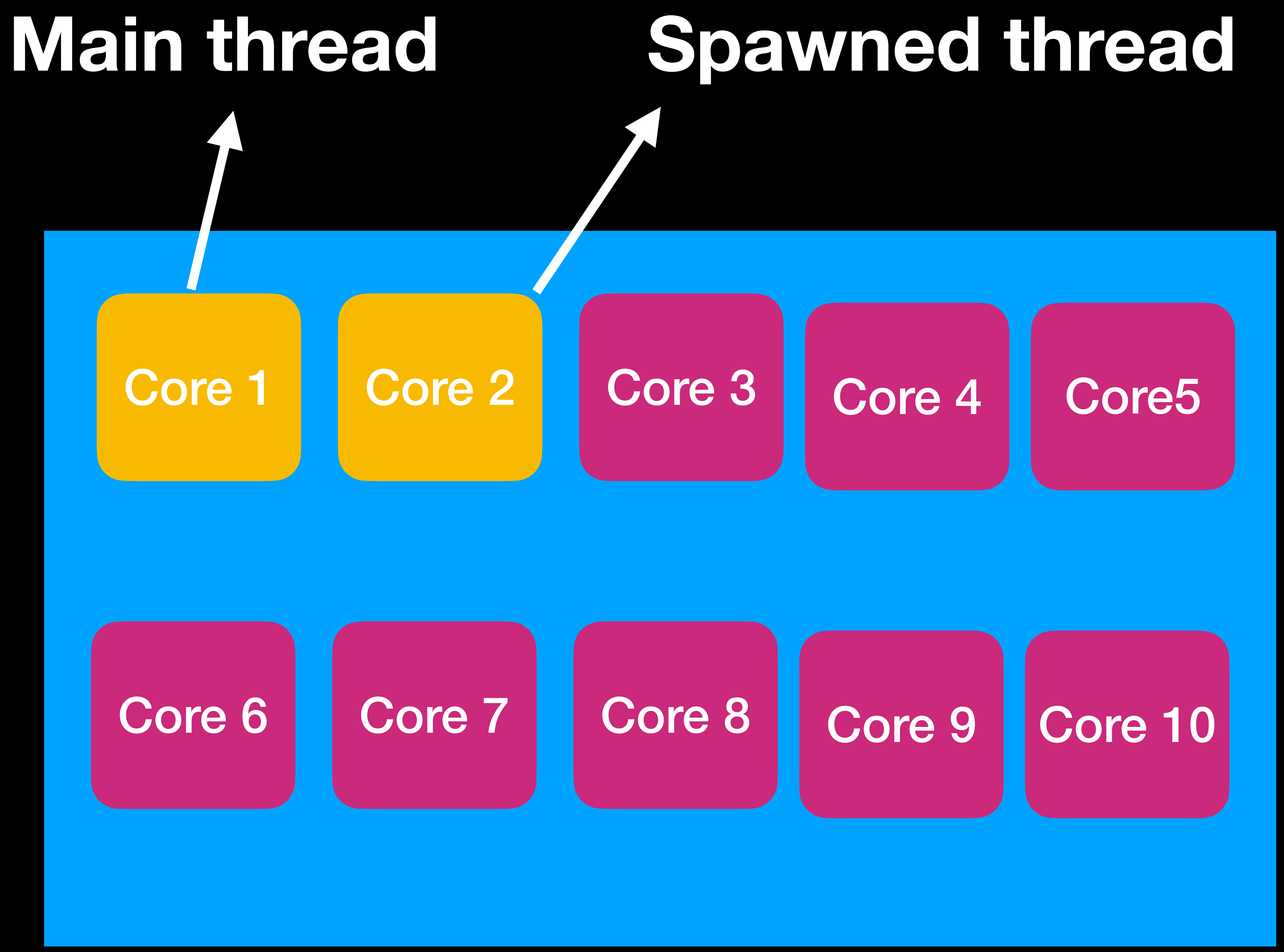
```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```



Multithreading



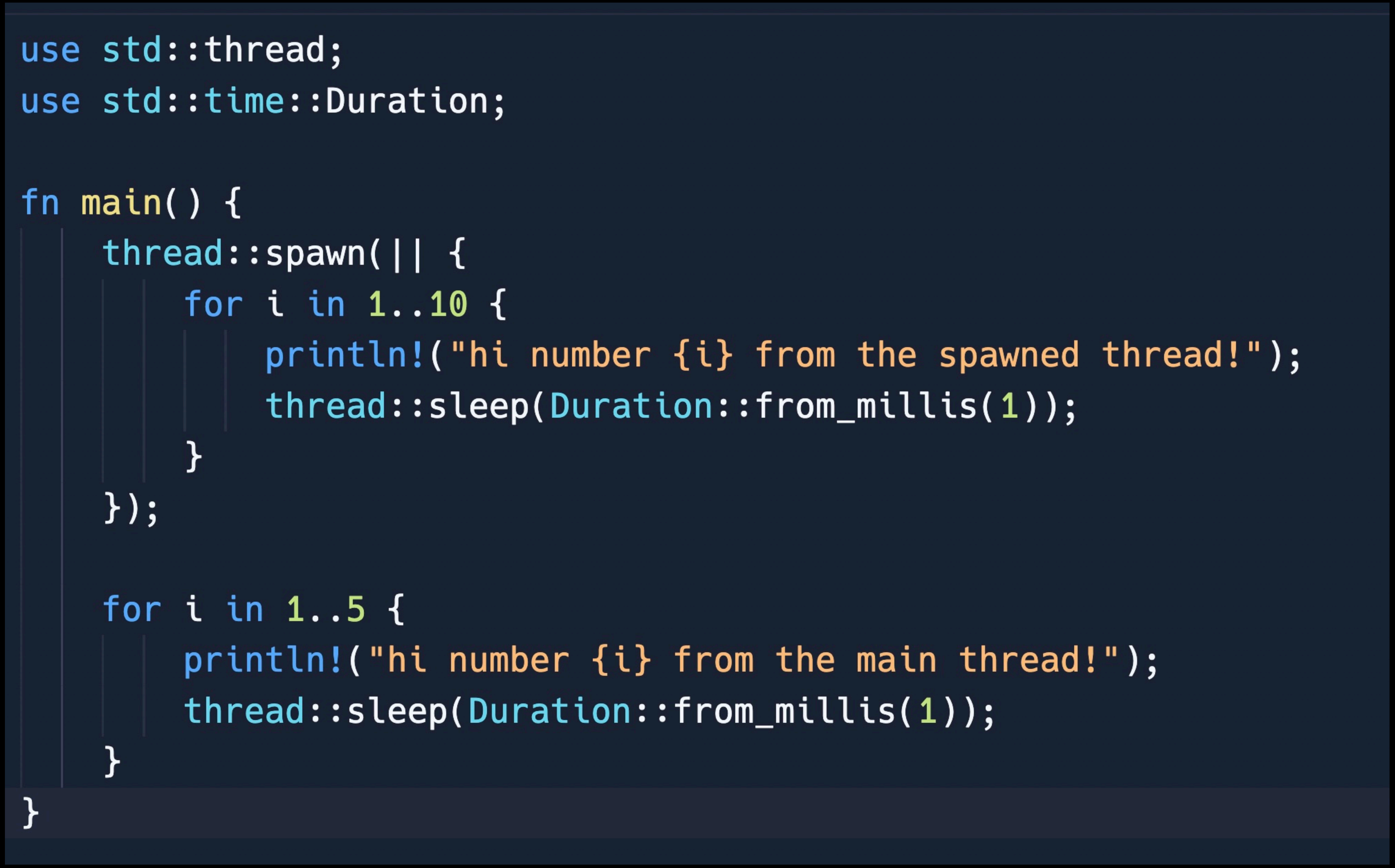
```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```



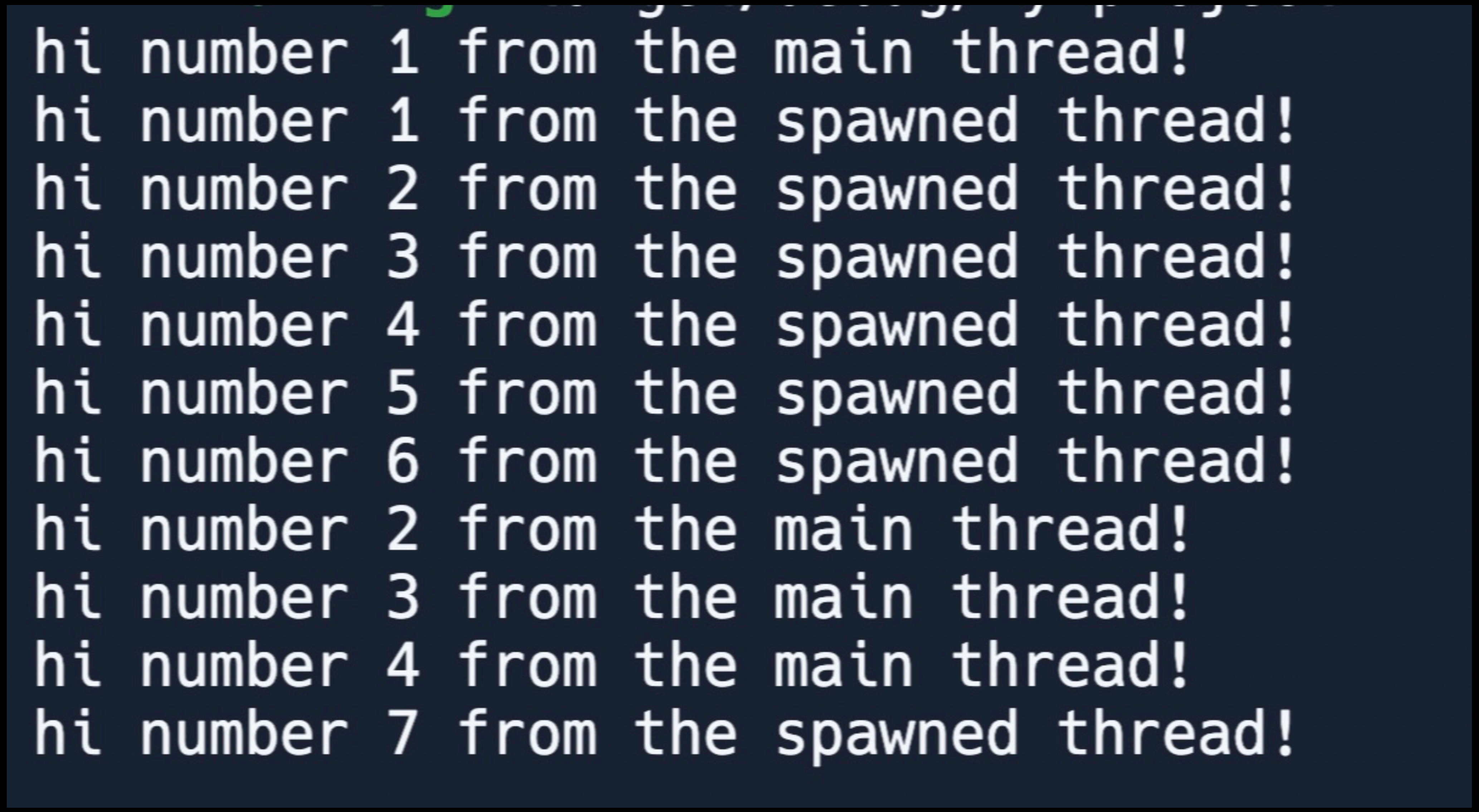
Main thread

Spawned thread

Multithreading

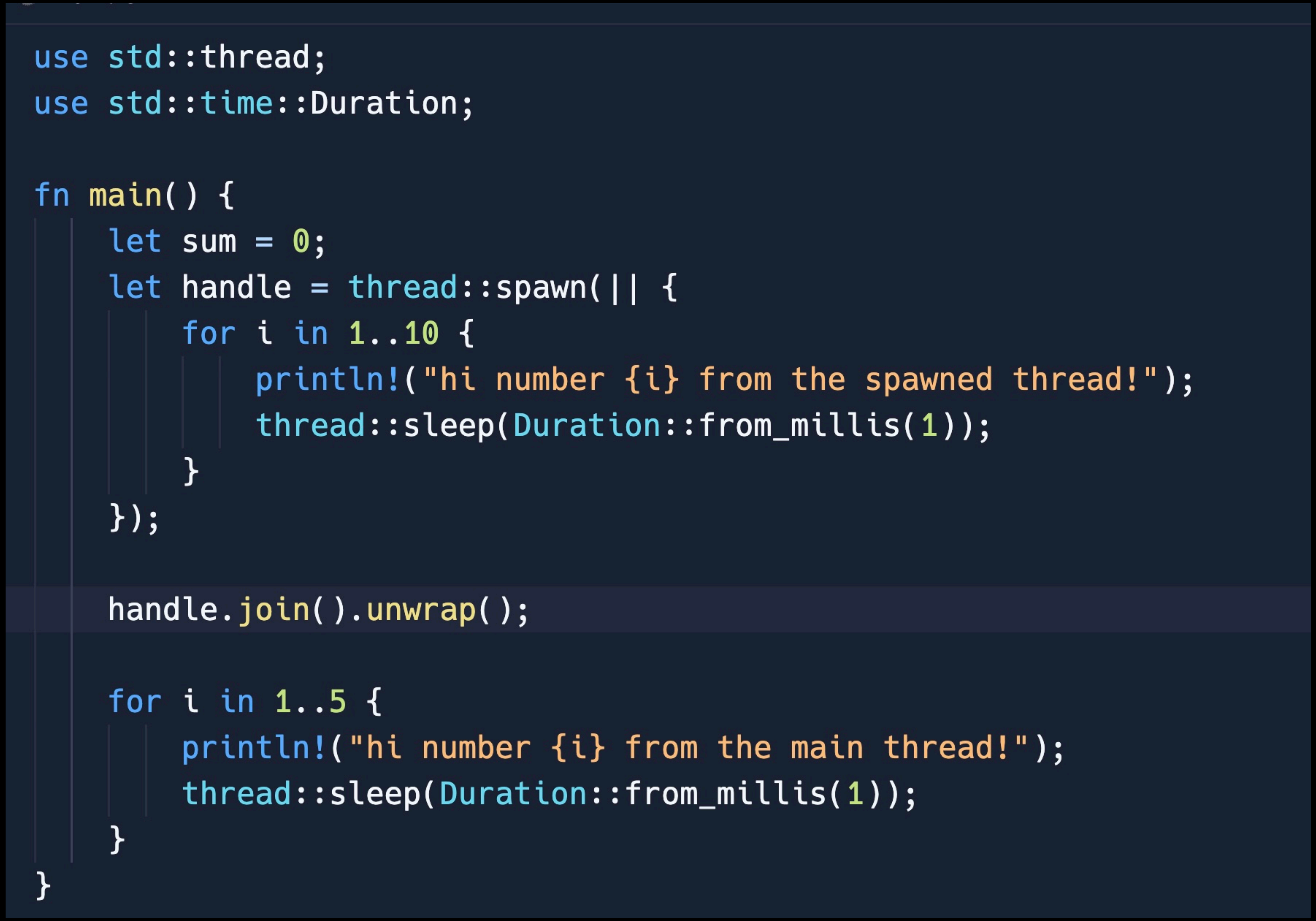


```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```



```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
hi number 7 from the spawned thread!
```

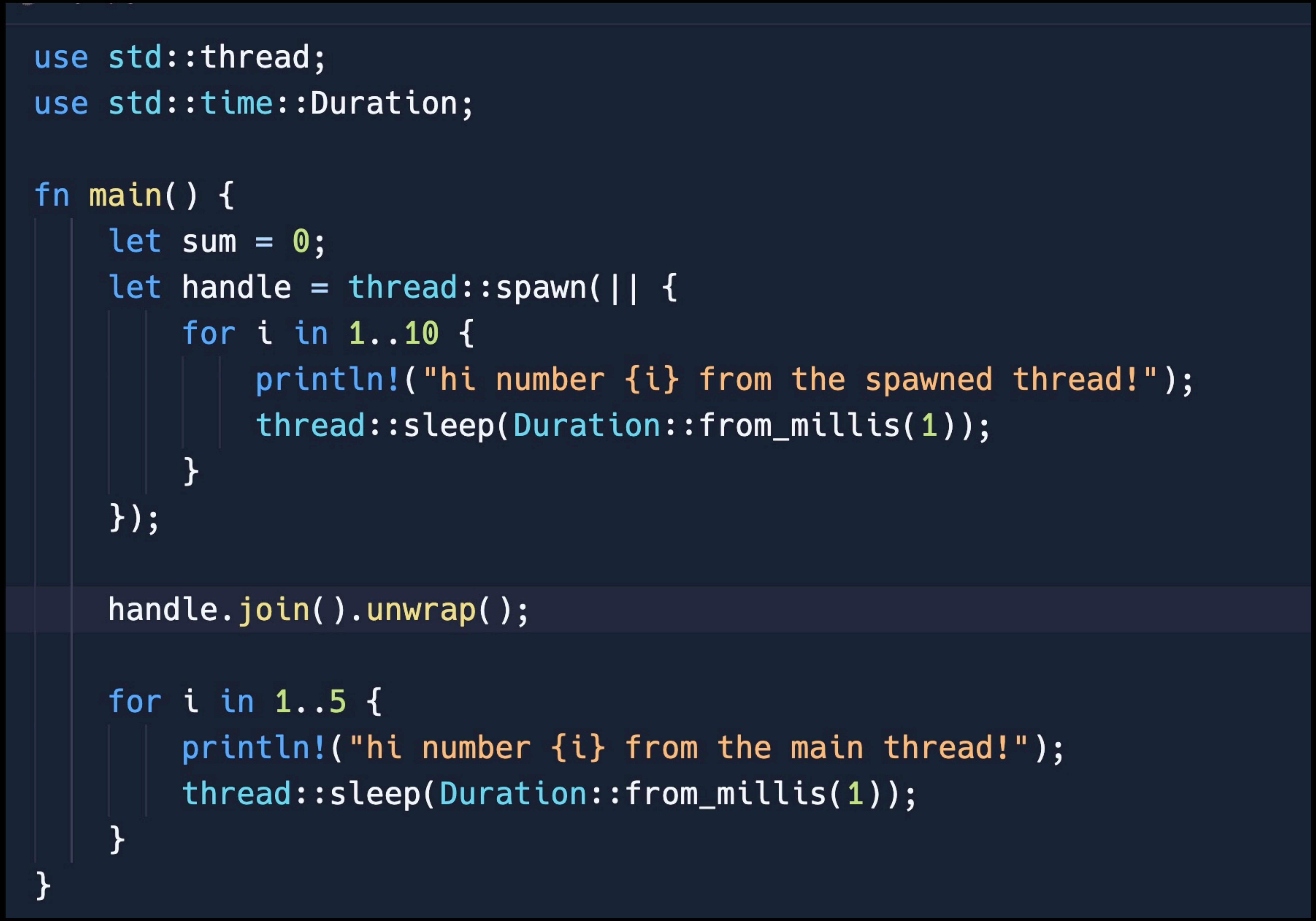
Multithreading



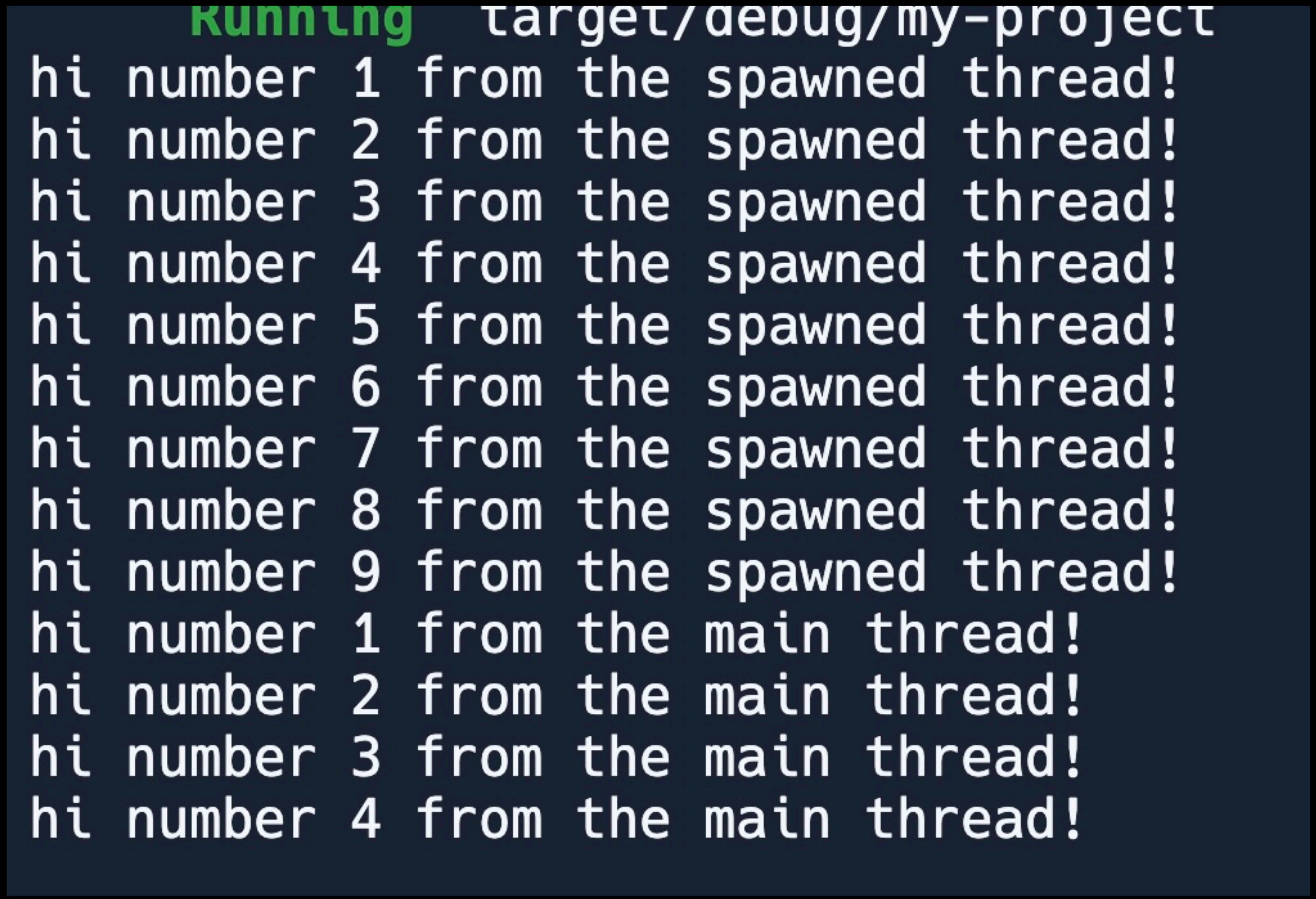
```rust
use std::thread;
use std::time::Duration;

fn main() {
    let sum = 0;
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```

Awaiting the thread to finish
Before running the iteration on the main thread

Multithreading

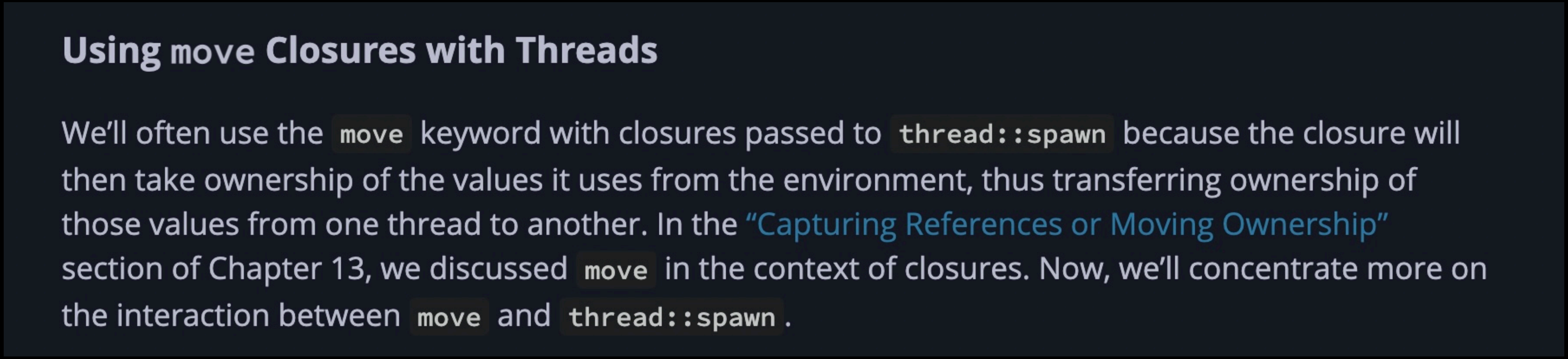


```rust
use std::thread;
use std::time::Duration;

fn main() {
    let sum = 0;
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```



```
     Running  target/debug/my-project
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

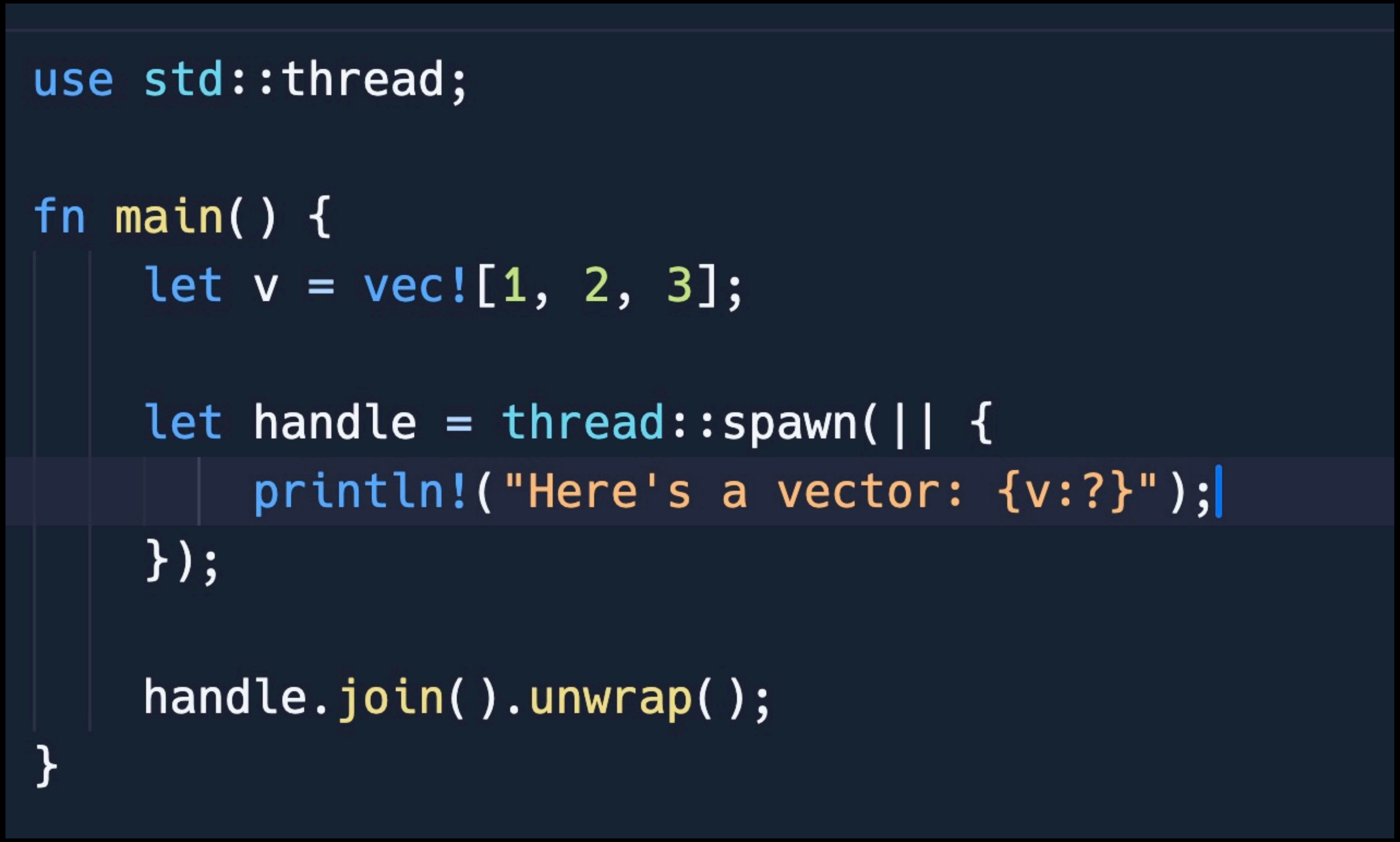
Multithreading



Using `move` Closures with Threads

We'll often use the `move` keyword with closures passed to `thread::spawn` because the closure will then take ownership of the values it uses from the environment, thus transferring ownership of those values from one thread to another. In the "Capturing References or Moving Ownership" section of Chapter 13, we discussed `move` in the context of closures. Now, we'll concentrate more on the interaction between `move` and `thread::spawn`.

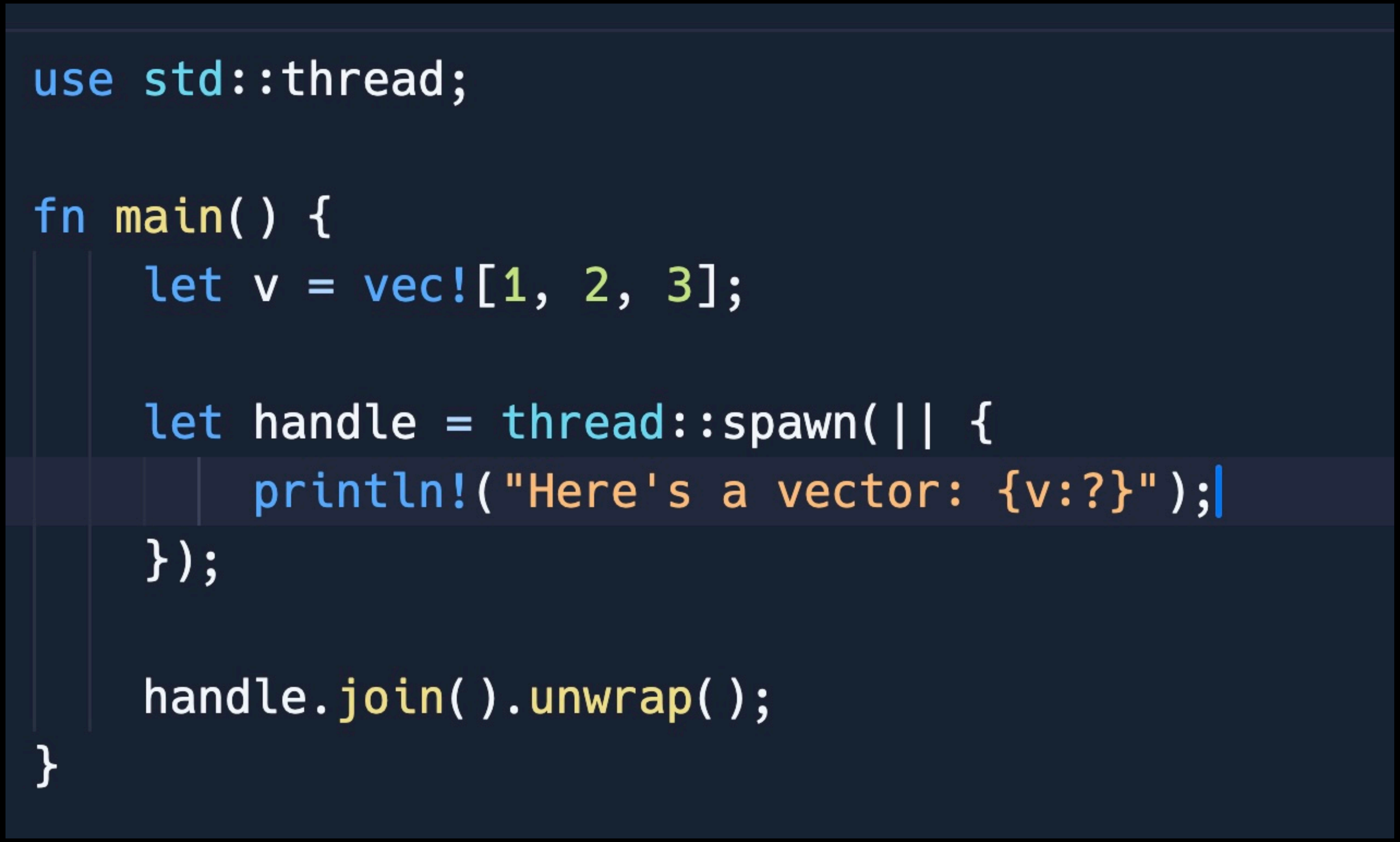
Multithreading



```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {v:?}");
    });

    handle.join().unwrap();
}
```

This code doesn’t compile
Because `v` could go out of scope before the thread starts

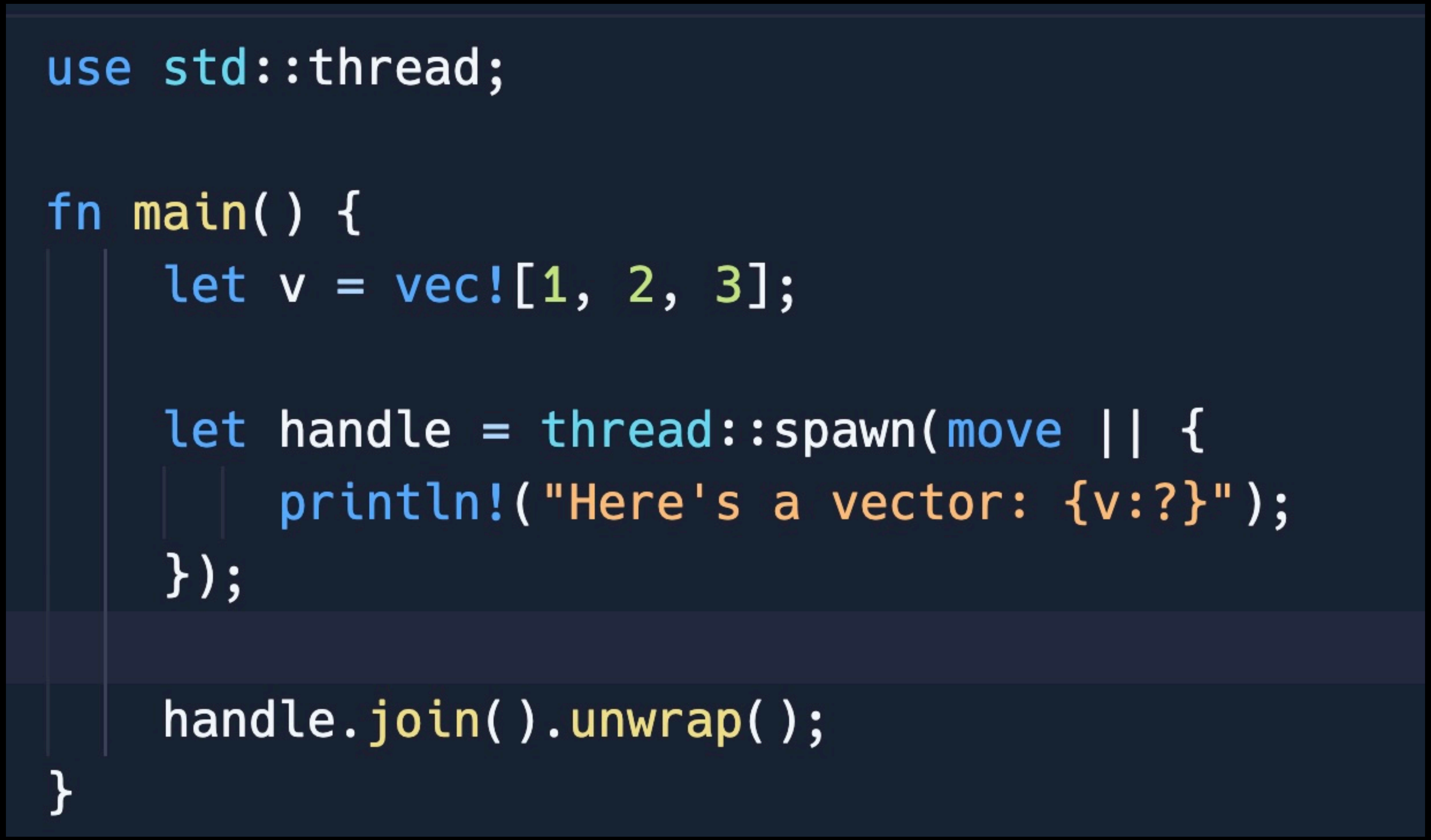
Multithreading



```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {v:?}");
    });

    handle.join().unwrap();
}
```

This code doesn’t compile
Because `v` could go out of scope before the thread starts

Multithreading

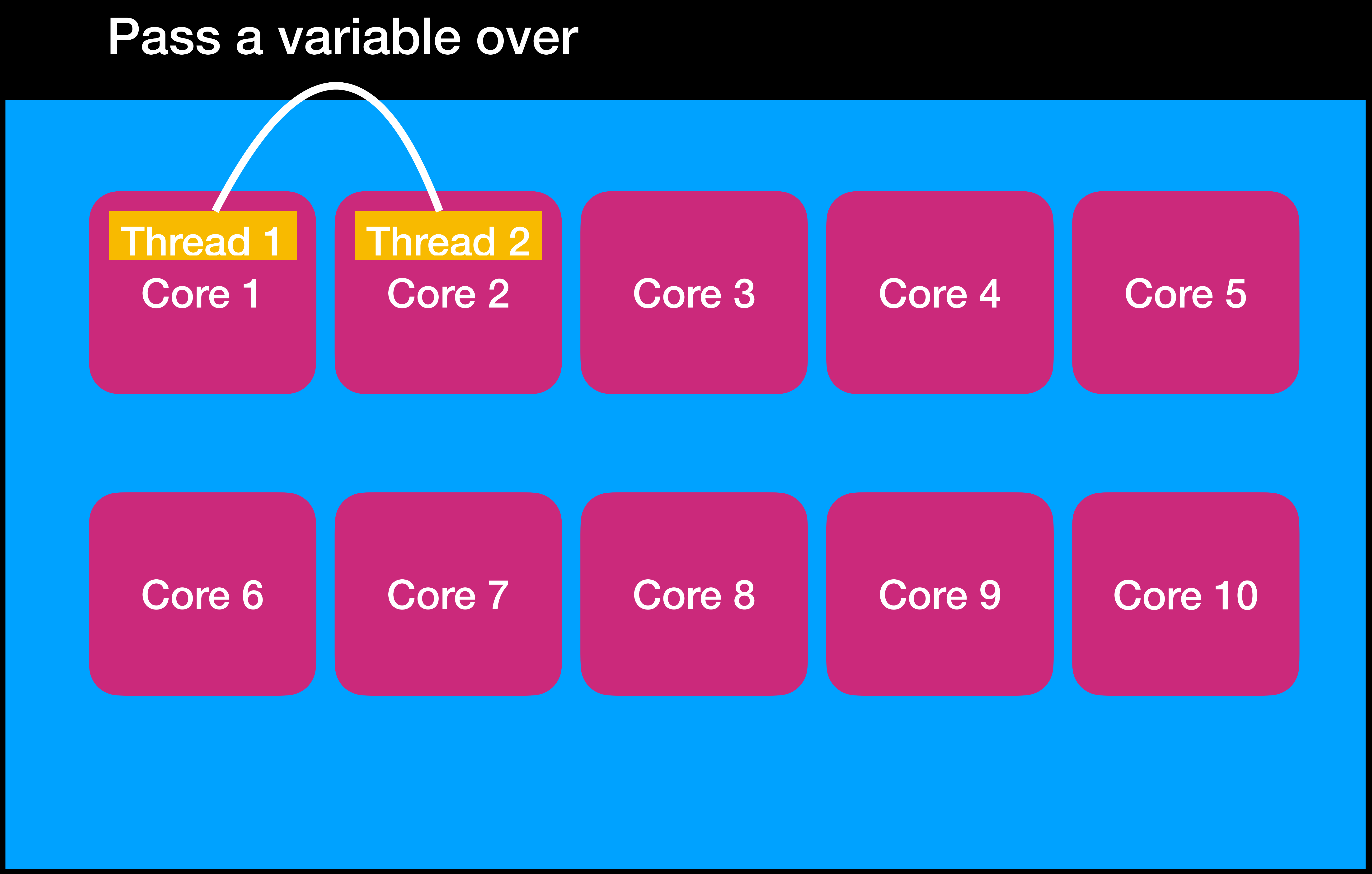


```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {v:?}");
    });

    handle.join().unwrap();
}
```

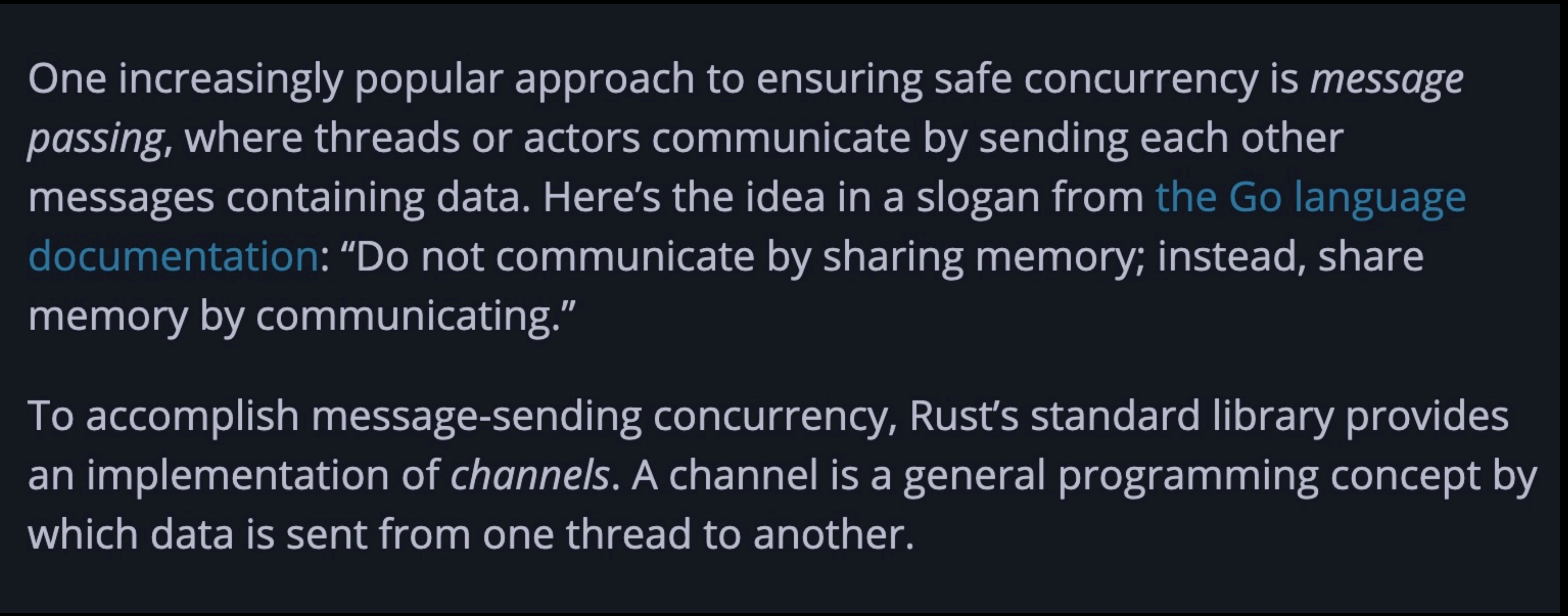
This code does compile
Because we `move` v to the spawned thread

It can not be used in the main thread anymore

Message passing



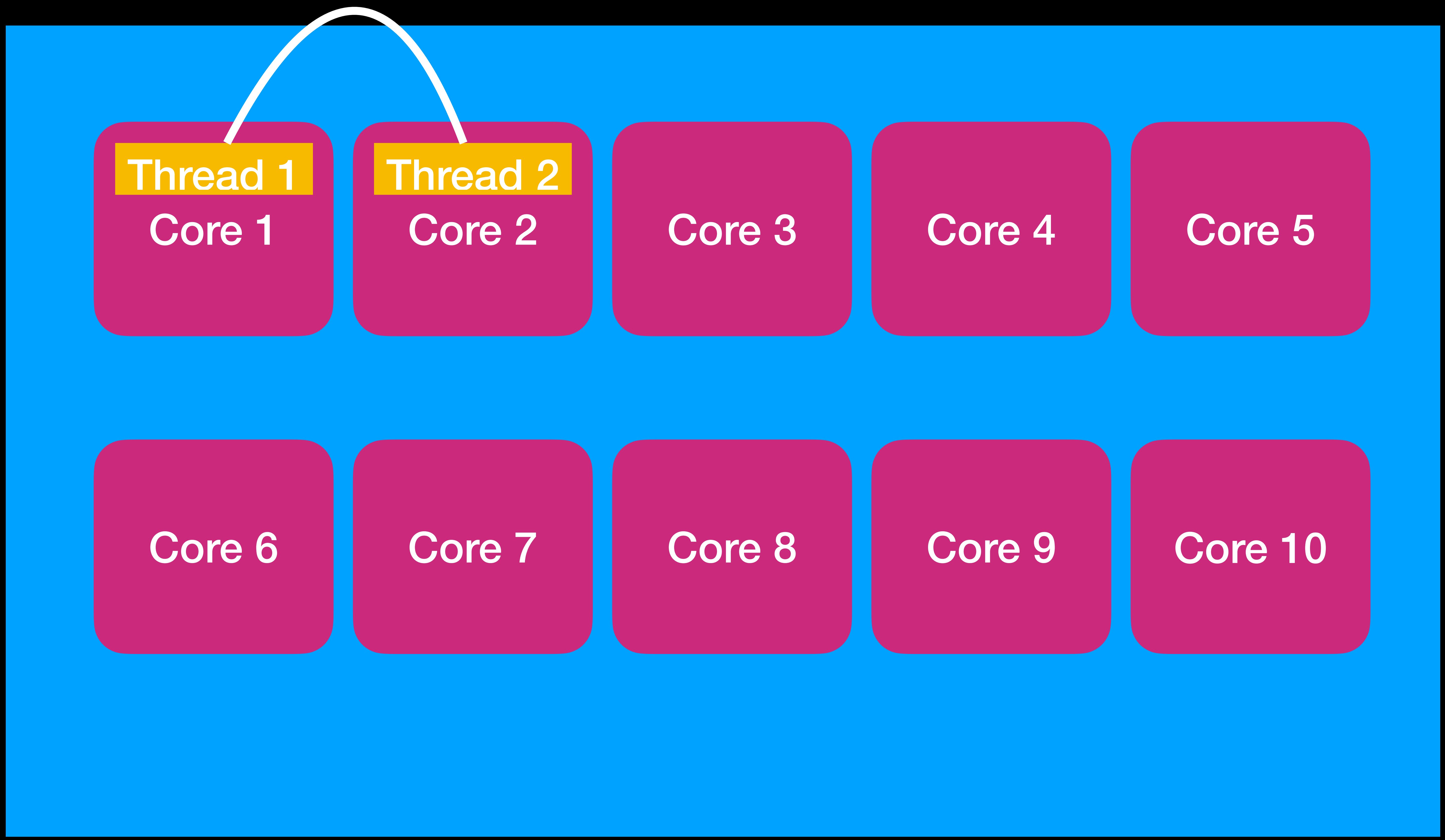
Message passing



One increasingly popular approach to ensuring safe concurrency is *message passing*, where threads or actors communicate by sending each other messages containing data. Here's the idea in a slogan from the Go language documentation: "Do not communicate by sharing memory; instead, share memory by communicating."

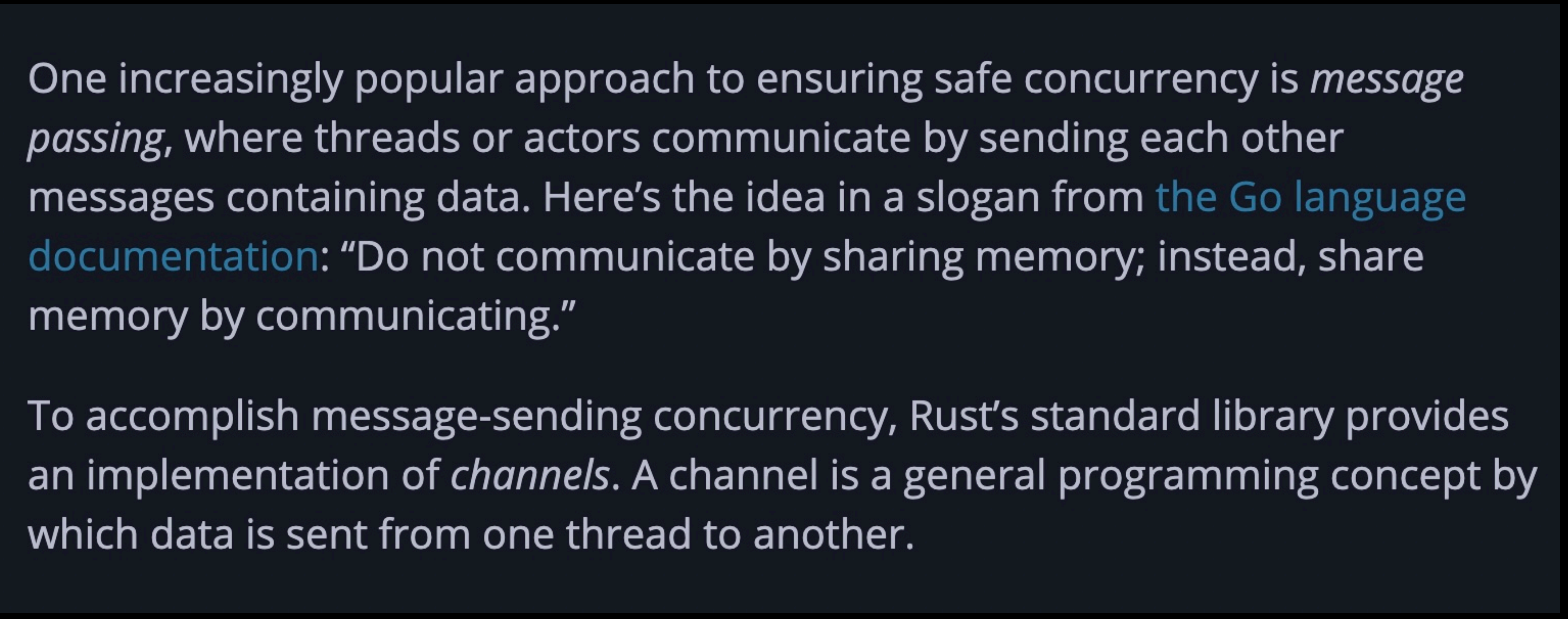
To accomplish message-sending concurrency, Rust's standard library provides an implementation of *channels*. A channel is a general programming concept by which data is sent from one thread to another.

Pass a variable over



Use case?
One thread reading data from redis ,
other thread processing it

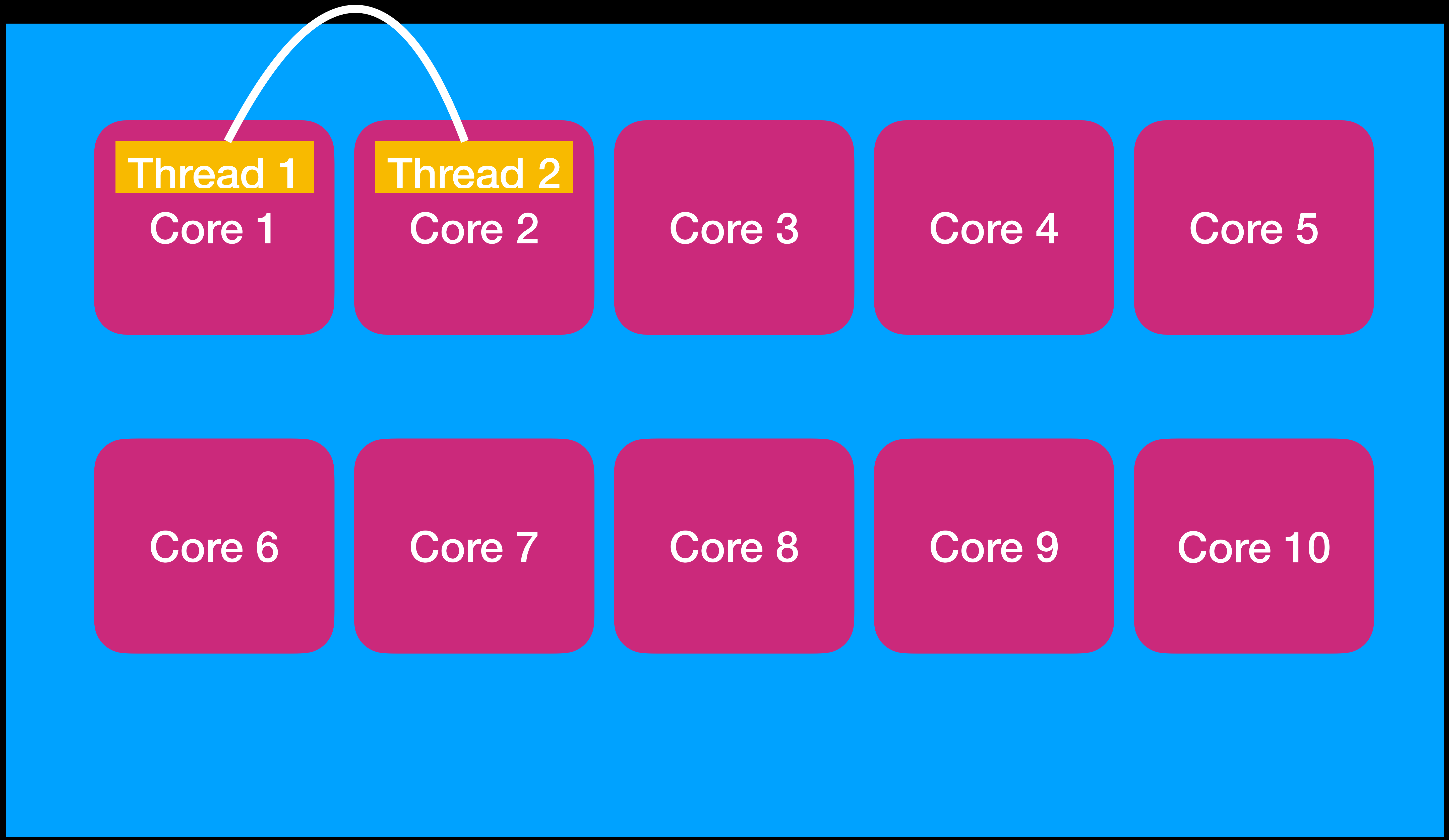
Message passing



One increasingly popular approach to ensuring safe concurrency is *message passing*, where threads or actors communicate by sending each other messages containing data. Here's the idea in a slogan from the Go language documentation: "Do not communicate by sharing memory; instead, share memory by communicating."

To accomplish message-sending concurrency, Rust's standard library provides an implementation of *channels*. A channel is a general programming concept by which data is sent from one thread to another.

Pass a variable over



Thread 1
Core 1

Thread 2
Core 2

Core 3

Core 4

Core 5

Core 6

Core 7

Core 8

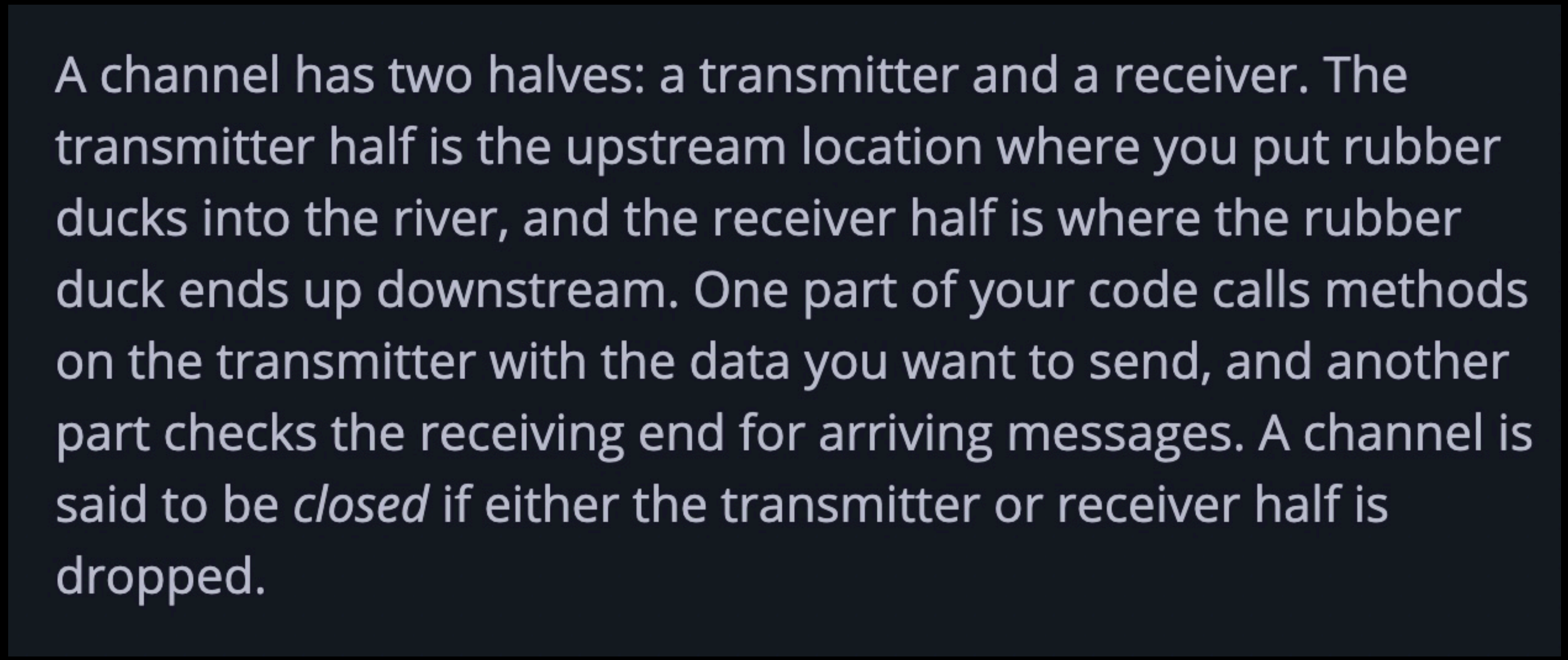
Core 9

Core 10

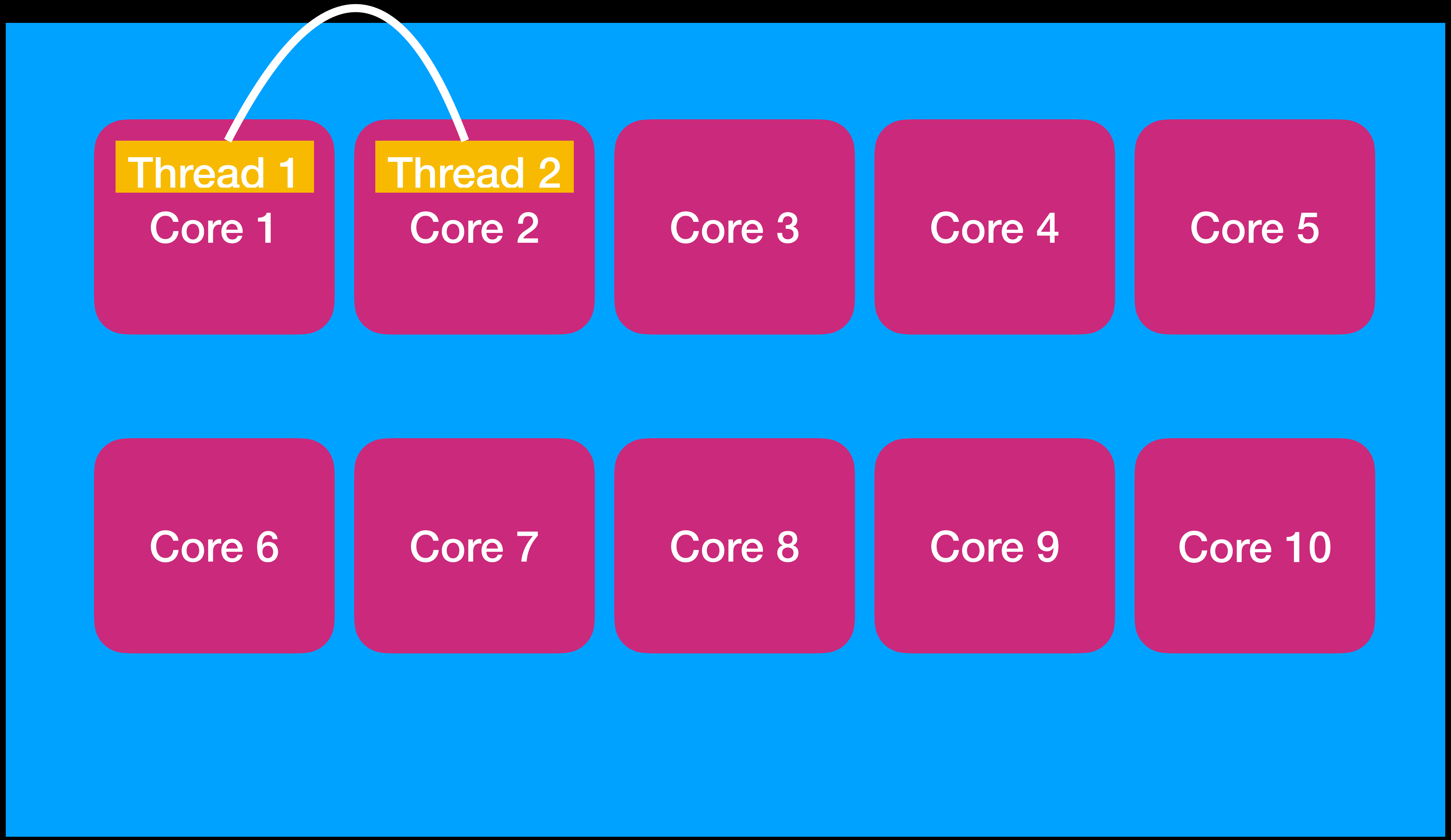
Use case?
One thread reading data from redis ,
other thread processing it

Message passing

Channels

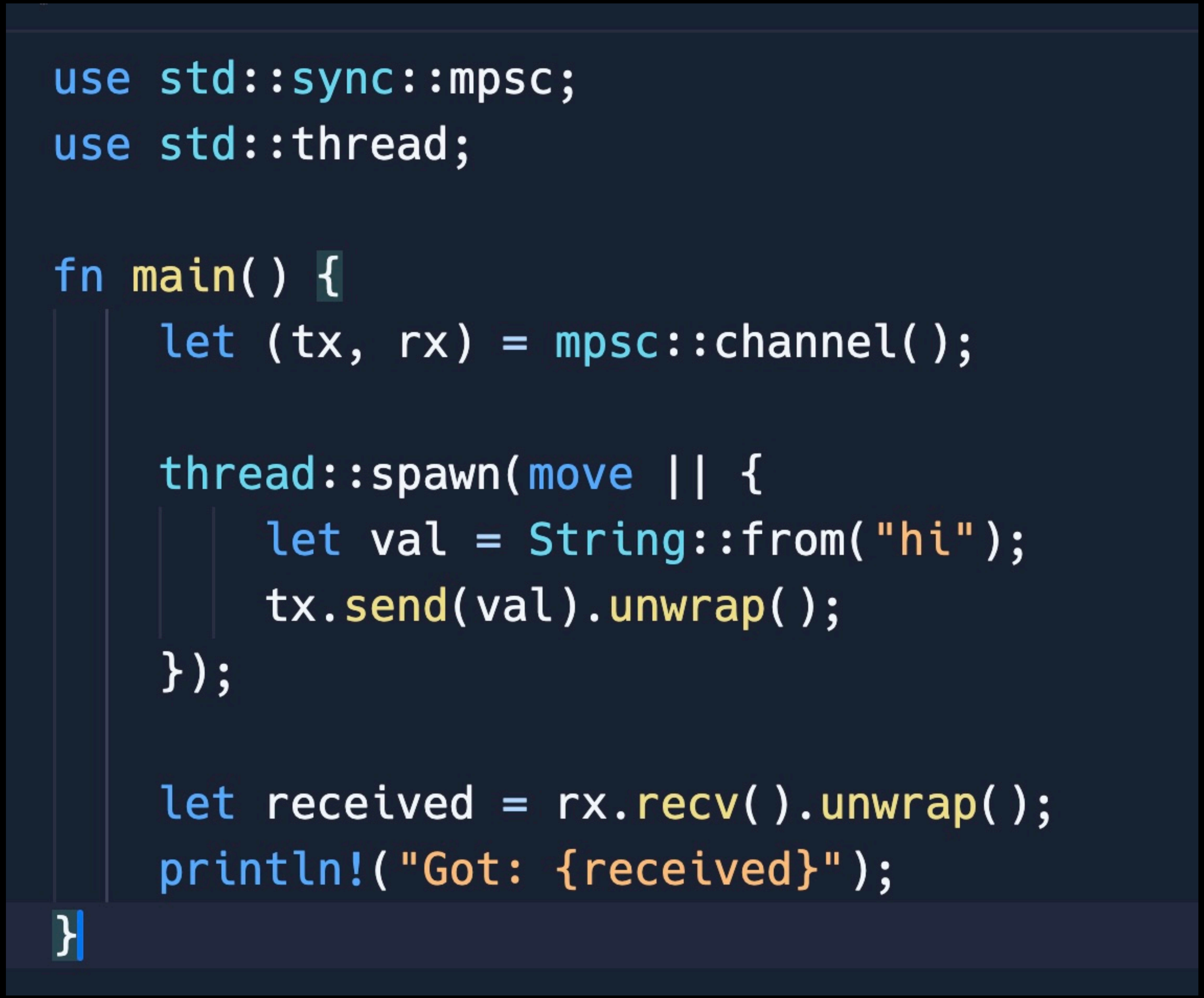


A channel has two halves: a transmitter and a receiver. The transmitter half is the upstream location where you put rubber ducks into the river, and the receiver half is where the rubber duck ends up downstream. One part of your code calls methods on the transmitter with the data you want to send, and another part checks the receiving end for arriving messages. A channel is said to be *closed* if either the transmitter or receiver half is dropped.

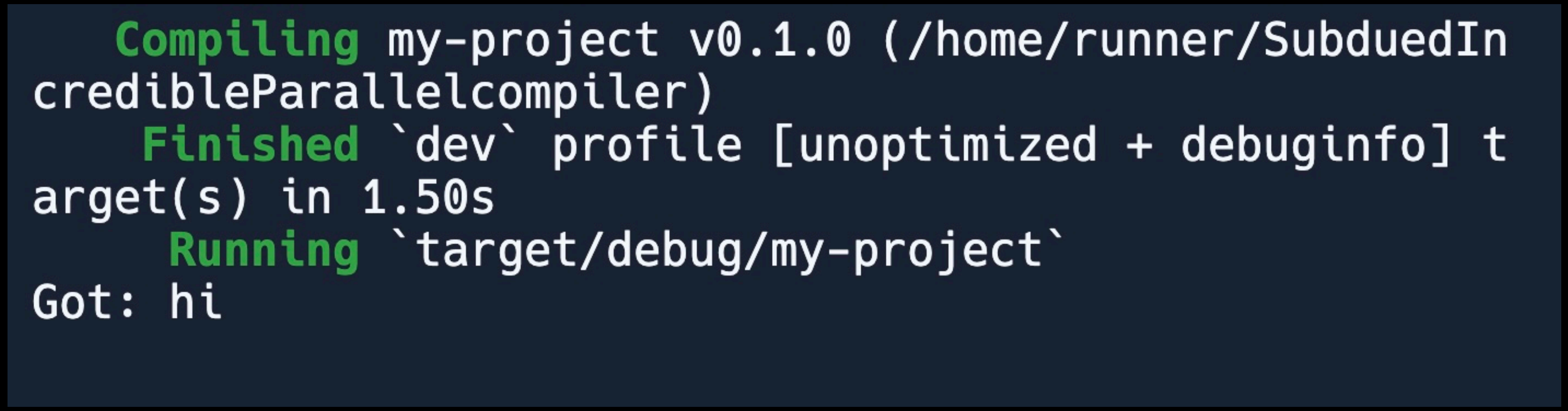


Message passing

Channels



```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {received}");
}
```



```
   Compiling my-project v0.1.0 (/home/runner/SubduedIn
crediblePArallelcompiler)
    Finished `dev` profile [unoptimized + debuginfo] t
arget(s) in 1.50s
     Running `target/debug/my-project`
Got: hi
```

Message passing

Channels

Can you write the code that finds the sum from 1 - 10^8?

Message passing

Channels

Can you write the code that finds the sum from 1 - 10^8?

Use threads to make sure you use all cores of your machine

Message passing

Channels

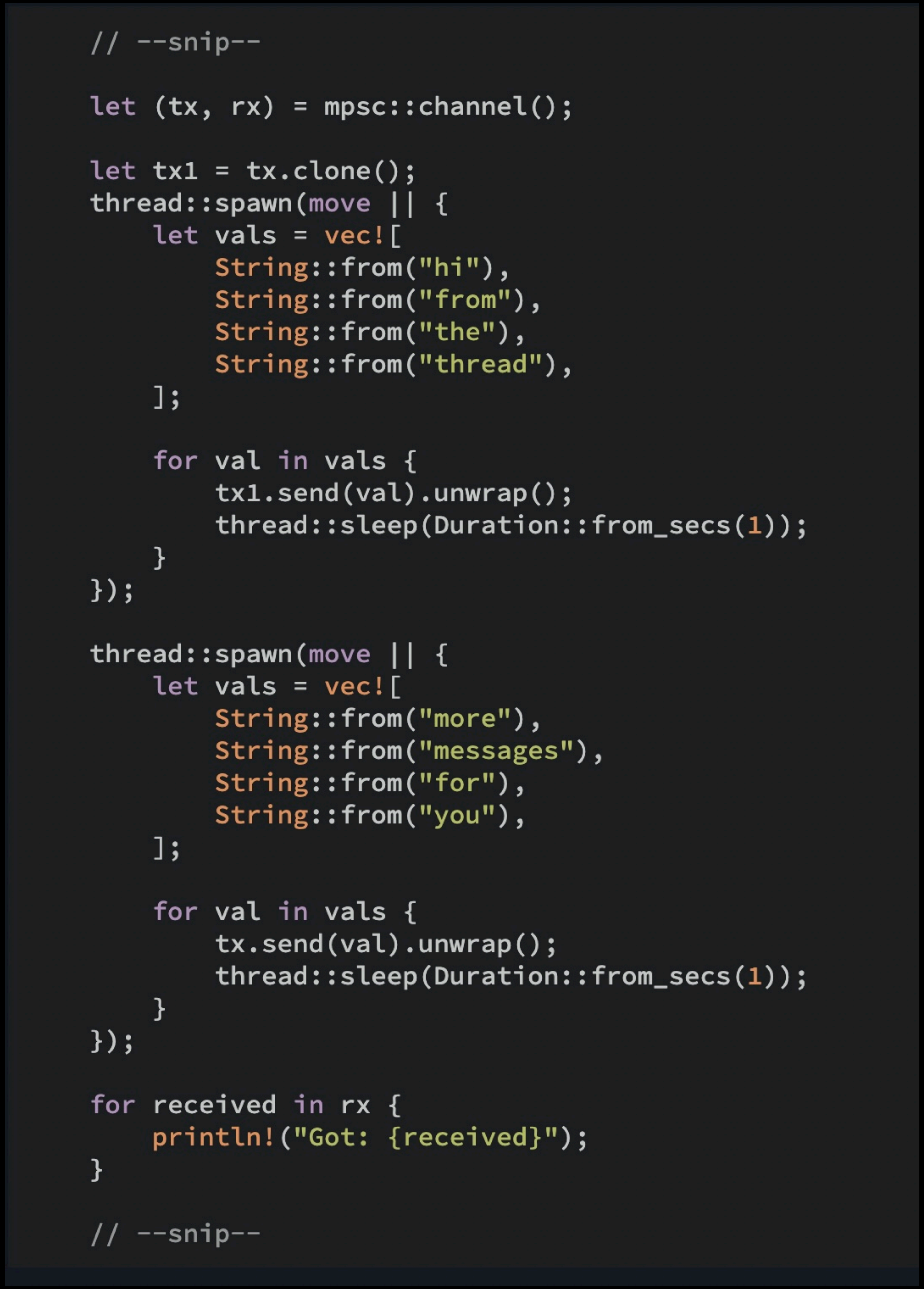
Can you write the code that finds the sum from 1 - 10^8?

Use threads to make sure you use all cores of your machine

Remember the name says `multiple producer single consumer`

Message passing

Hint -
You can clone a producer before moving it to a thread



```rust
// --snip--

let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("Got: {received}");
}

// --snip--
```

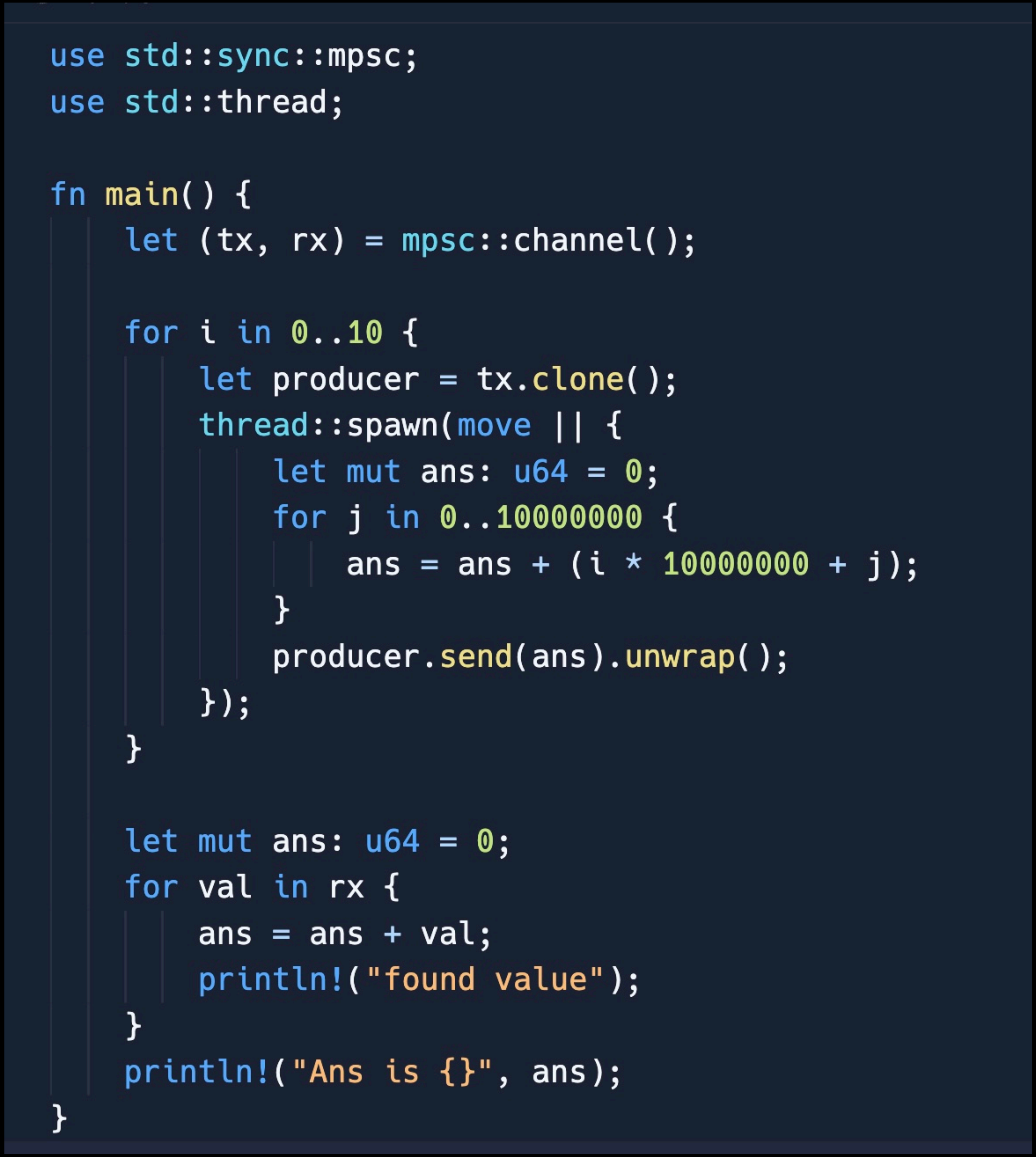
Message passing

Channels

Can you write the code that finds the sum from 1 - 10^8?

Message passing

This code almost works

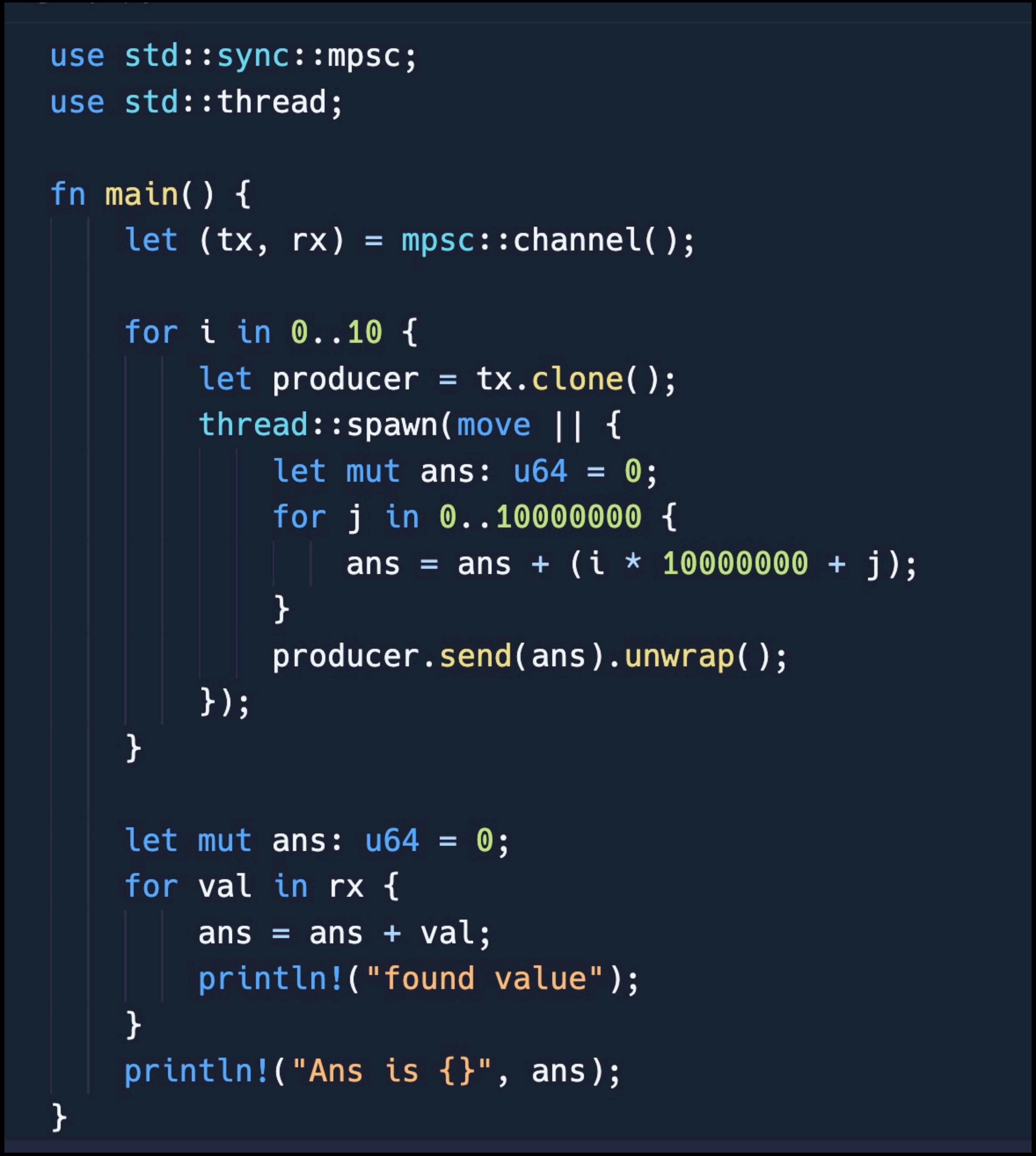


```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let producer = tx.clone();
        thread::spawn(move || {
            let mut ans: u64 = 0;
            for j in 0..10000000 {
                ans = ans + (i * 10000000 + j);
            }
            producer.send(ans).unwrap();
        });
    }

    let mut ans: u64 = 0;
    for val in rx {
        ans = ans + val;
        println!("found value");
    }
    println!("Ans is {}", ans);
}
```

Message passing

This code almost works

Can you guess what goes wrong here?

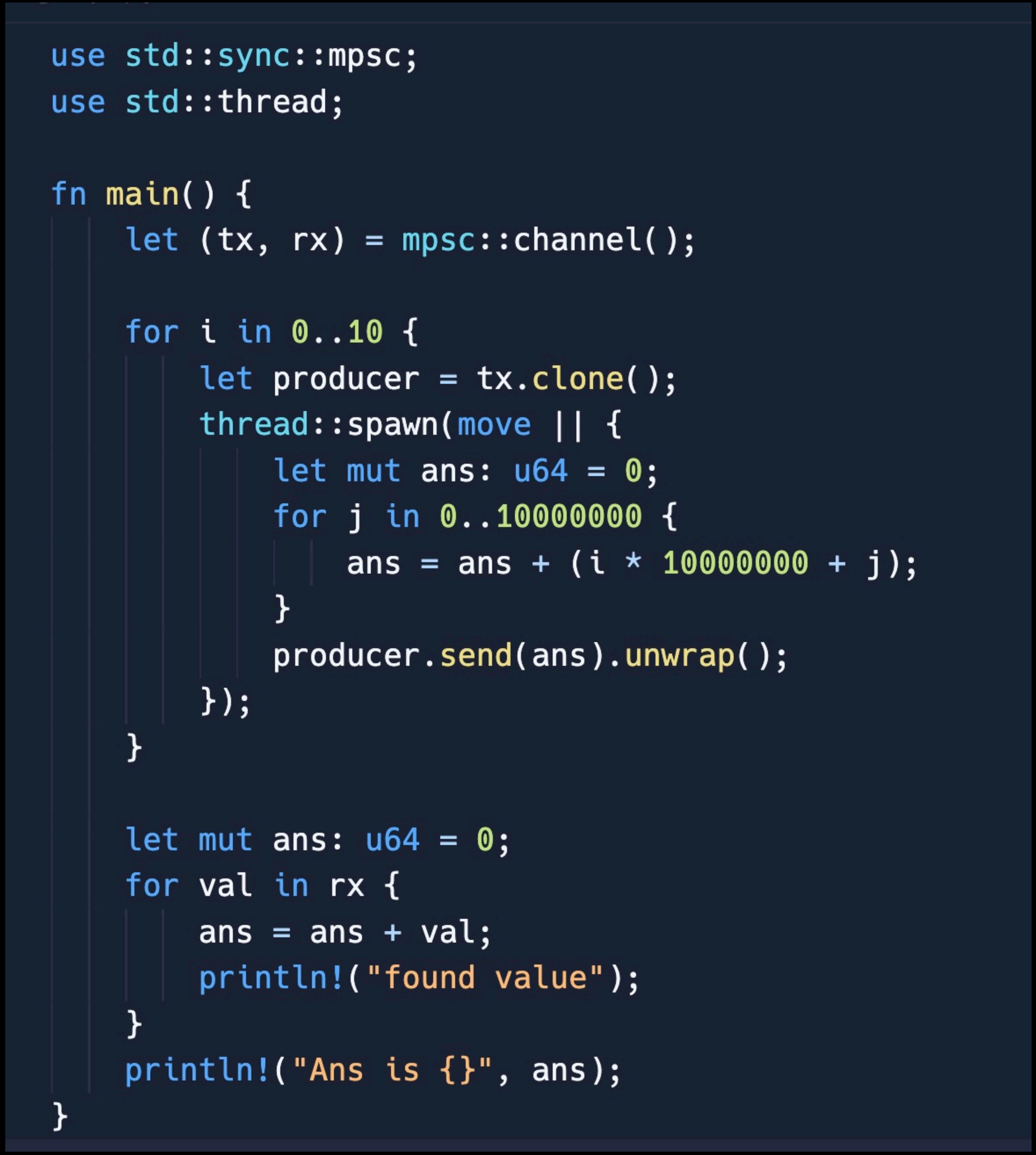


```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let producer = tx.clone();
        thread::spawn(move || {
            let mut ans: u64 = 0;
            for j in 0..10000000 {
                ans = ans + (i * 10000000 + j);
            }
            producer.send(ans).unwrap();
        });
    }

    let mut ans: u64 = 0;
    for val in rx {
        ans = ans + val;
        println!("found value");
    }
    println!("Ans is {}", ans);
}
```

Message passing

This code almost works
Can you guess what goes wrong here?

The original tx variable never drops, and so the for loop keeps on waiting for data from it

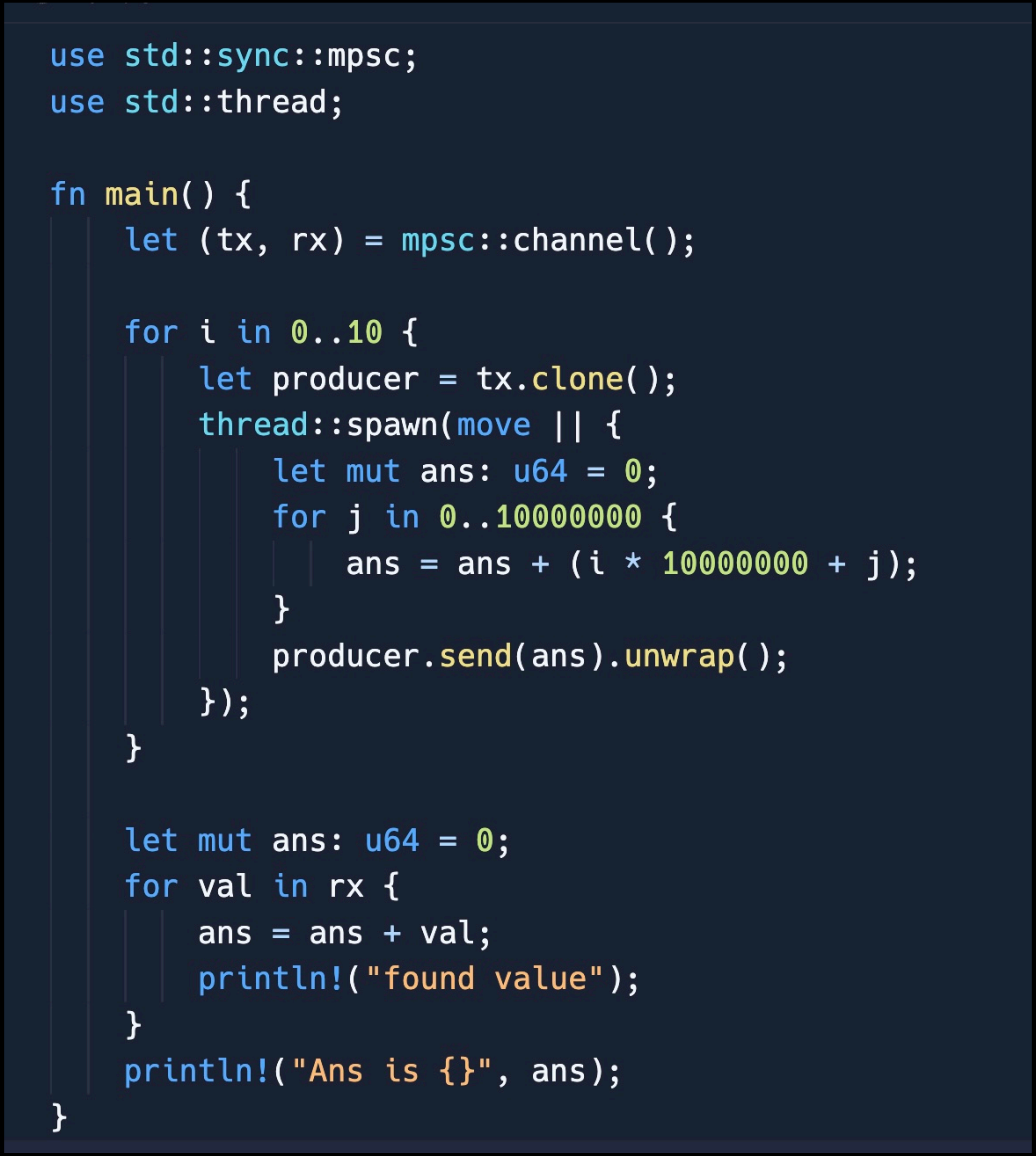


```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let producer = tx.clone();
        thread::spawn(move || {
            let mut ans: u64 = 0;
            for j in 0..10000000 {
                ans = ans + (i * 10000000 + j);
            }
            producer.send(ans).unwrap();
        });
    }

    let mut ans: u64 = 0;
    for val in rx {
        ans = ans + val;
        println!("found value");
    }
    println!("Ans is {}", ans);
}
```

Message passing

This code almost works

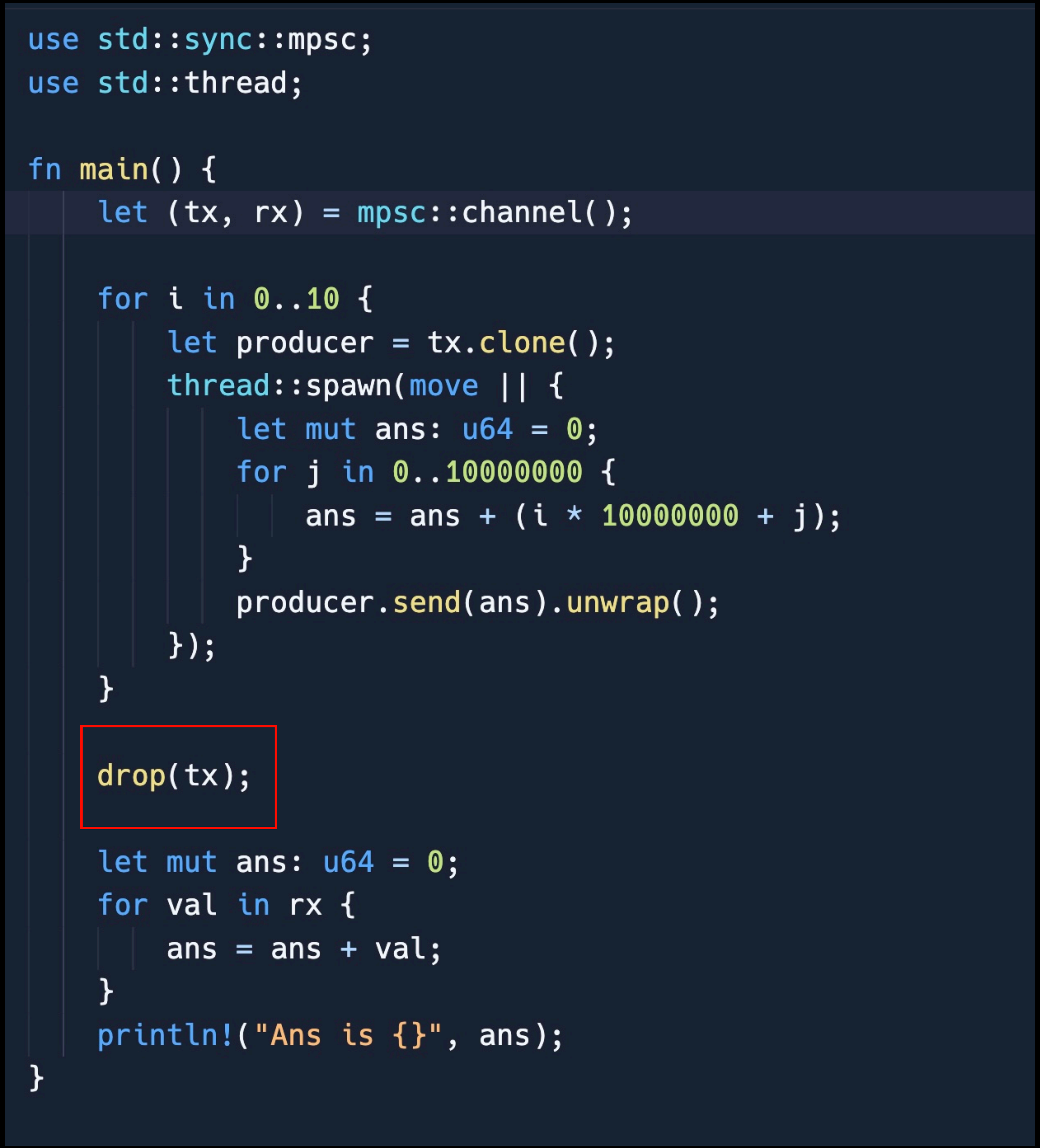
Can you guess what goes wrong here?



```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let producer = tx.clone();
        thread::spawn(move || {
            let mut ans: u64 = 0;
            for j in 0..10000000 {
                ans = ans + (i * 10000000 + j);
            }
            producer.send(ans).unwrap();
        });
    }

    let mut ans: u64 = 0;
    for val in rx {
        ans = ans + val;
        println!("found value");
    }
    println!("Ans is {}", ans);
}
```

Message passing

Right solution



```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let producer = tx.clone();
        thread::spawn(move || {
            let mut ans: u64 = 0;
            for j in 0..10000000 {
                ans = ans + (i * 10000000 + j);
            }
            producer.send(ans).unwrap();
        });
    }

    drop(tx);

    let mut ans: u64 = 0;
    for val in rx {
        ans = ans + val;
    }
    println!("Ans is {}", ans);
}
```

Async rust