RUSTikales Rust for advanced coders

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Slices

1. Recap

1. Recap

- References by default are very unsafe

1. Recap

- References by default are very unsafe
- Static analysis required to guarantee memory safety

1. Recap

- References by default are very unsafe
- Static analysis required to guarantee memory safety
- Borrow Checker
  - Reference mutably borrowed twice+ → Not allowed, illegal
  - Reference mutably borrowed once → No other borrows allowed
  - Reference immutably borrowed → Only other immutable borrows allowed
  - Reference may not outlive borrowed data

1. Recap

- References by default are very unsafe
- Static analysis required to guarantee memory safety
- Borrow Checker
- Borrow Checker evaluates Lifetimes of all values

1. Recap

- References by default are very unsafe
- Static analysis required to guarantee memory safety
- Borrow Checker
- Borrow Checker evaluates Lifetimes of all values
- Lifetimes are complicated…
  - Variable → The scope they're defined in
  - Value → Dropped when owner dropped
  - Reference → Well…

1. Recap

- References by default are very unsafe
- Static analysis required to guarantee memory safety
- Borrow Checker
- Borrow Checker evaluates Lifetimes of all values
- Lifetimes are complicated…
- Lifetime of a reference → Not only when, but where

1. Recap

- References by default are very unsafe
- Static analysis required to guarantee memory safety
- Borrow Checker
- Borrow Checker evaluates Lifetimes of all values
- Lifetimes are complicated…
- Lifetime of a reference → Not only when, but where
  - A region of code in which the reference must be valid
    - Between assigning and using the reference
    - Can have gaps
    - Non-Lexical Lifetime → Not limited to scopes

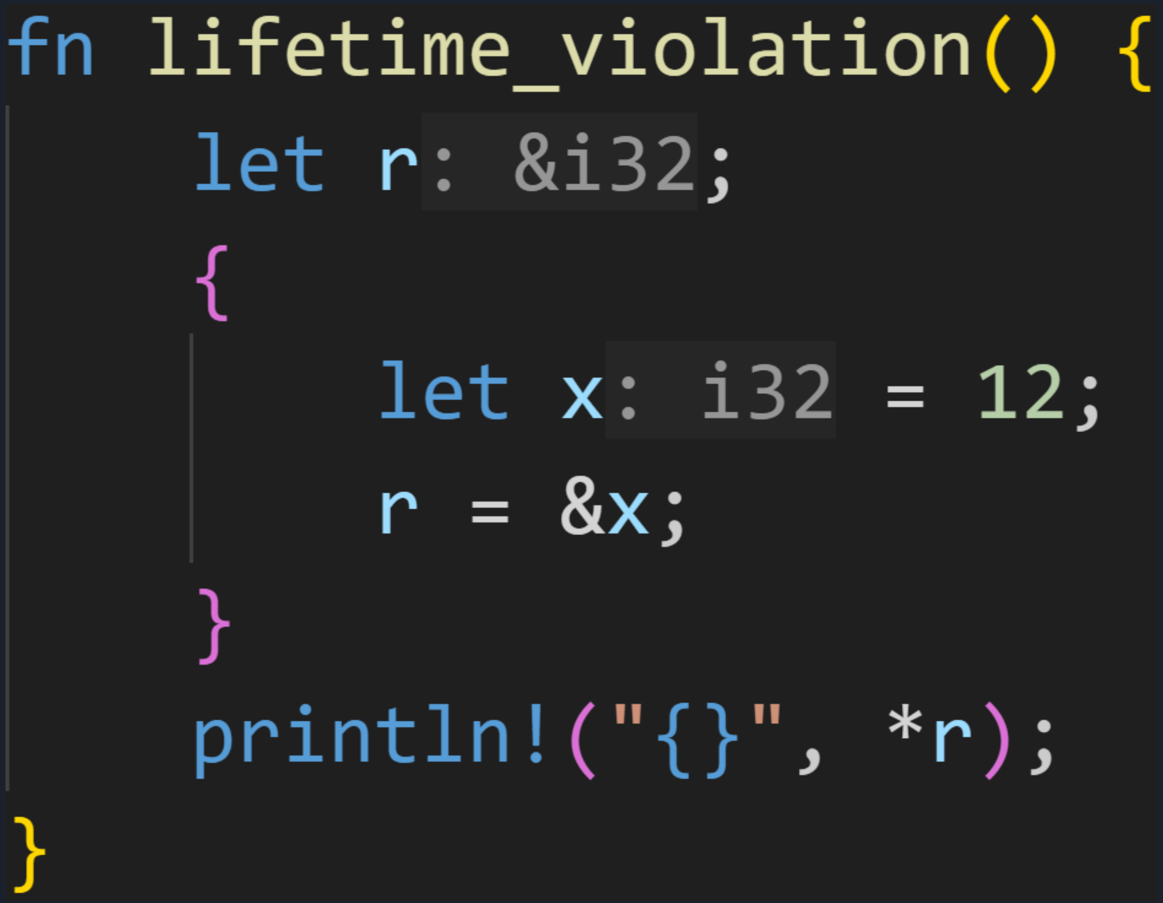
1. Recap

- References by default are very unsafe
- Static analysis required to guarantee memory safety
- Borrow Checker
- Borrow Checker evaluates Lifetimes of all values
- Lifetimes are complicated…
- Lifetime of a reference → Not only when, but where
  - A region of code in which the reference must be valid
  - A region of memory where the reference may point into
    - When using the reference, the original value must be alive

1. Recap

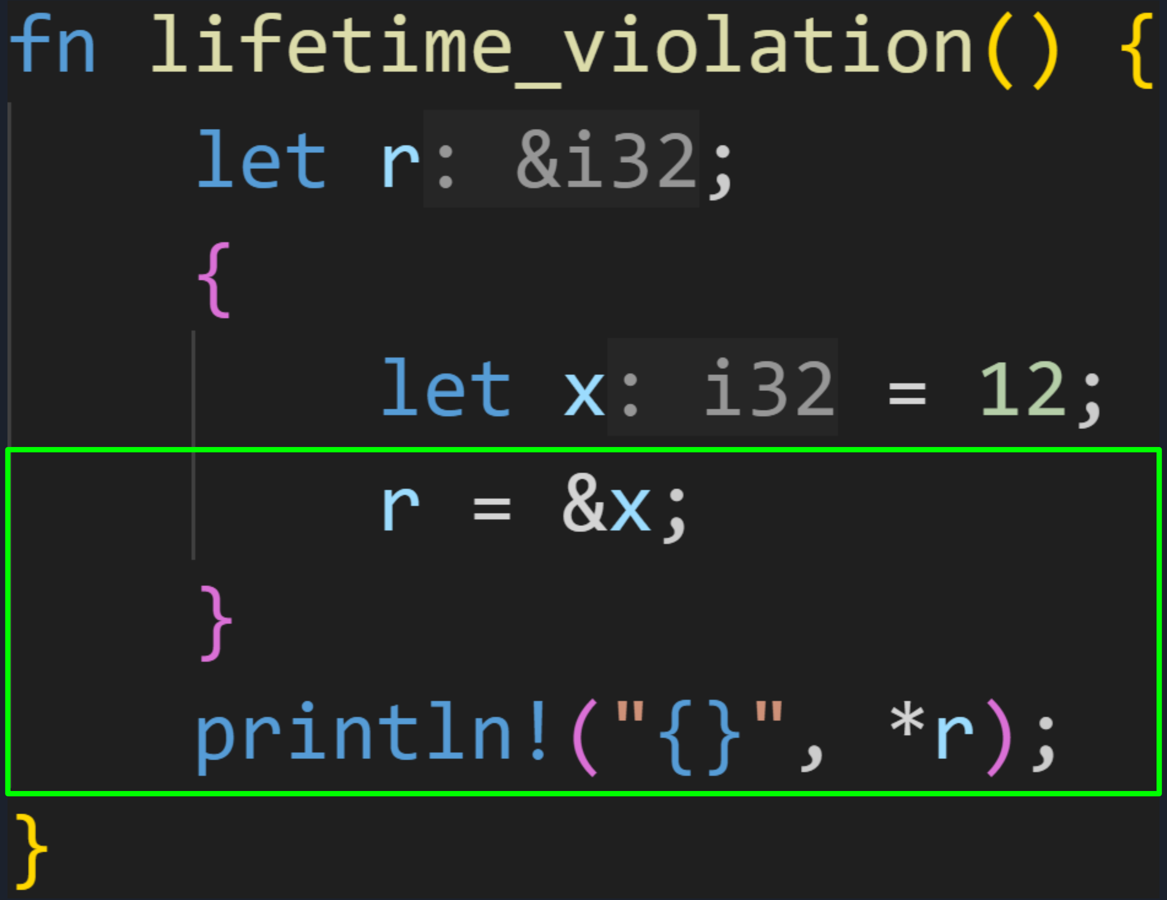
- References by default are very unsafe
- Static analysis required to guarantee memory safety
- Borrow Checker
- Borrow Checker evaluates Lifetimes of all values
- Lifetimes are complicated…
- Lifetime of a reference → Not only when, but where
- Compiler is very good at figuring out lifetimes
  - Lifetime Elision
  - Sometimes, we have to specify them ourselves → Named lifetime parameters

1. Recap



```
fn lifetime_violation() {
    let r: &i32;
    {
        let x: i32 = 12;
        r = &x;
    }
    println!("{}", *r);
}
```

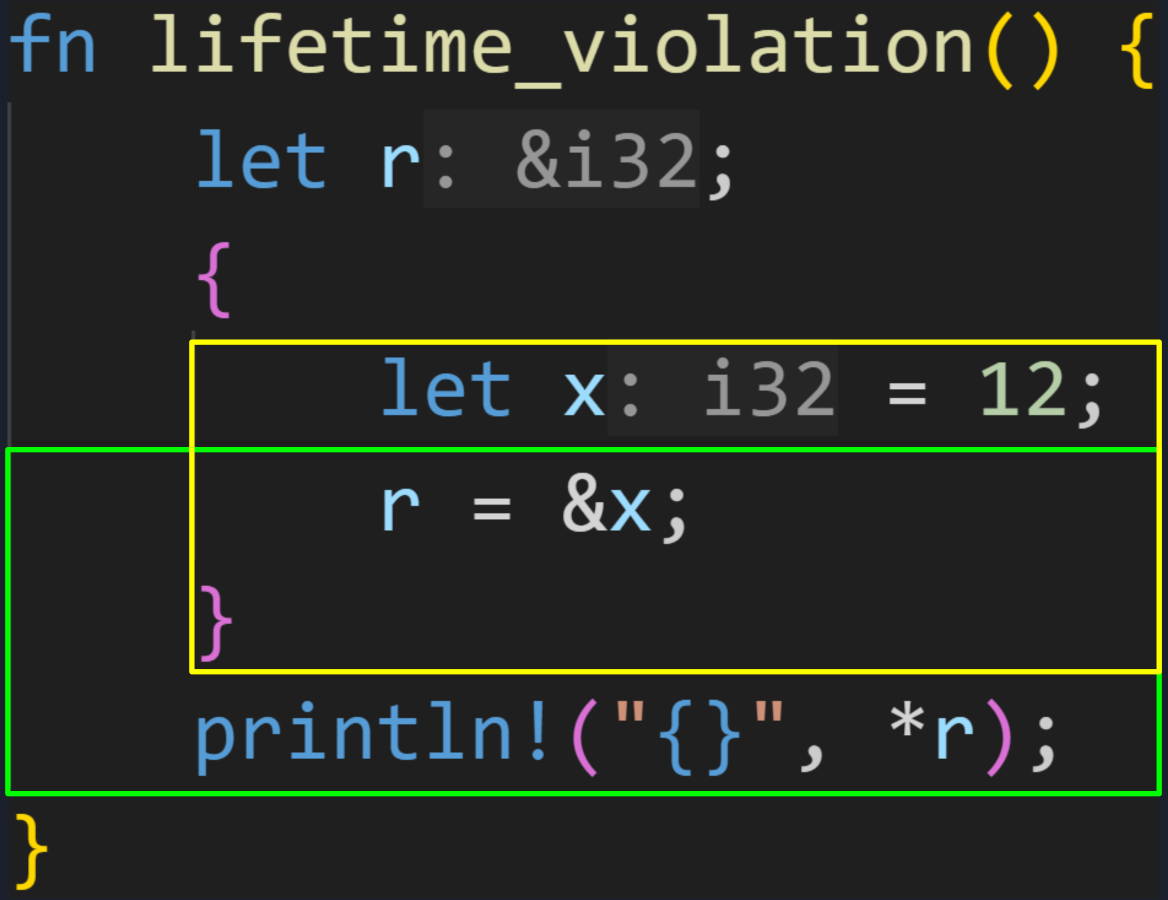
1. Recap



```rust
fn lifetime_violation() {
    let r: &i32;
    {
        let x: i32 = 12;
        r = &x;
    }
    println!("{}", *r);
}
```

Lifetime of &x

1. Recap

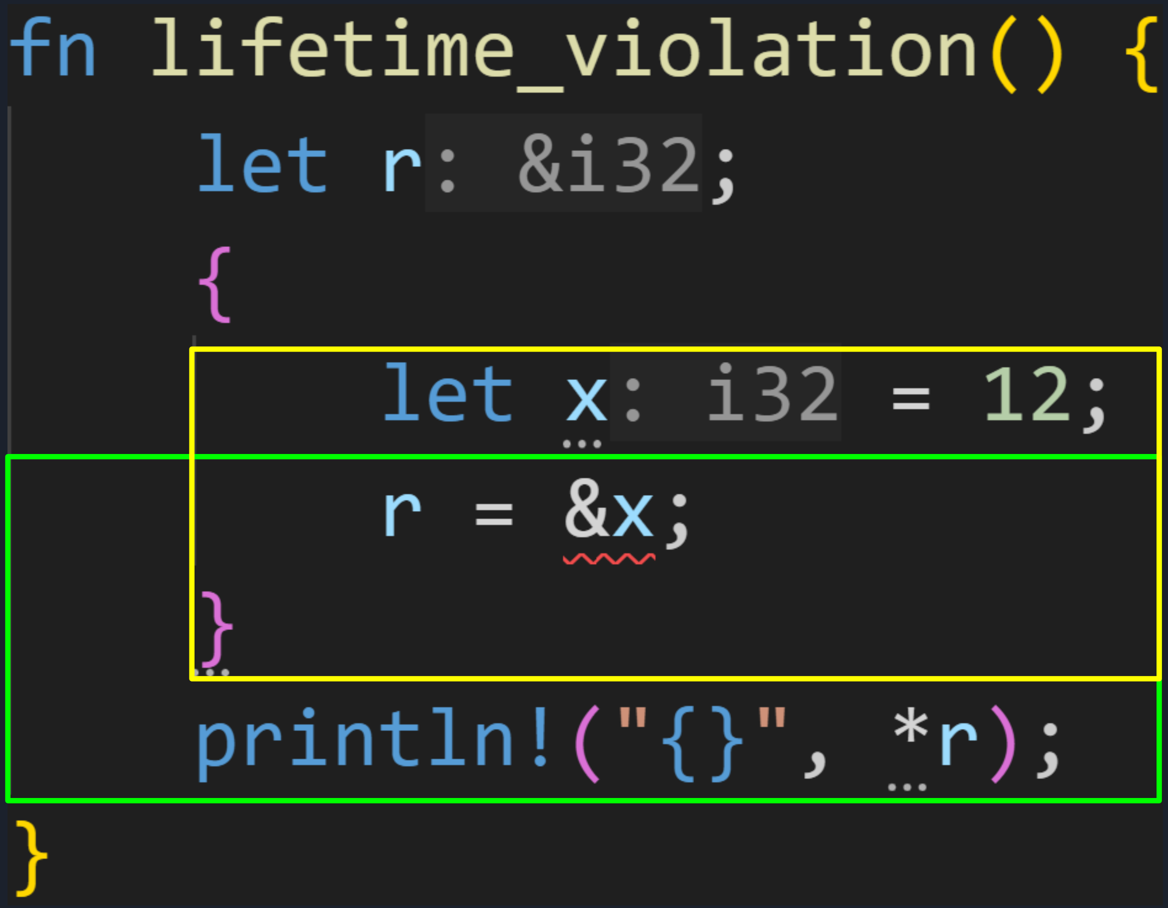


```rust
fn lifetime_violation() {
    let r: &i32;
    {
        let x: i32 = 12;
        r = &x;
    }
    println!("{}", *r);
}
```

Lifetime of x

Lifetime of &x

1. Recap



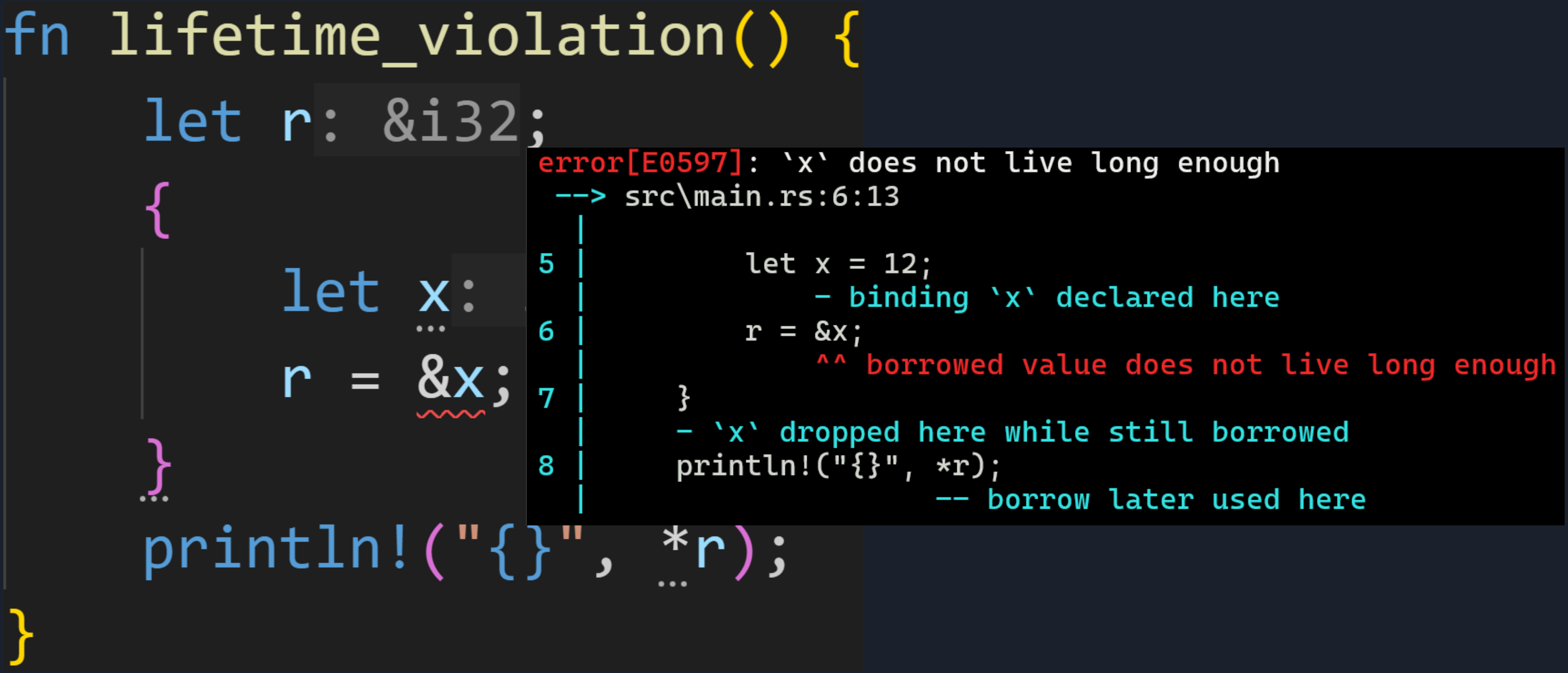
```rust
fn lifetime_violation() {
    let r: &i32;
    {
        let x: i32 = 12;
        r = &x;
    }
    println!("{}", *r);
}
```

Lifetime of x

Lifetime of &x

&x outlives x, this is not allowed!

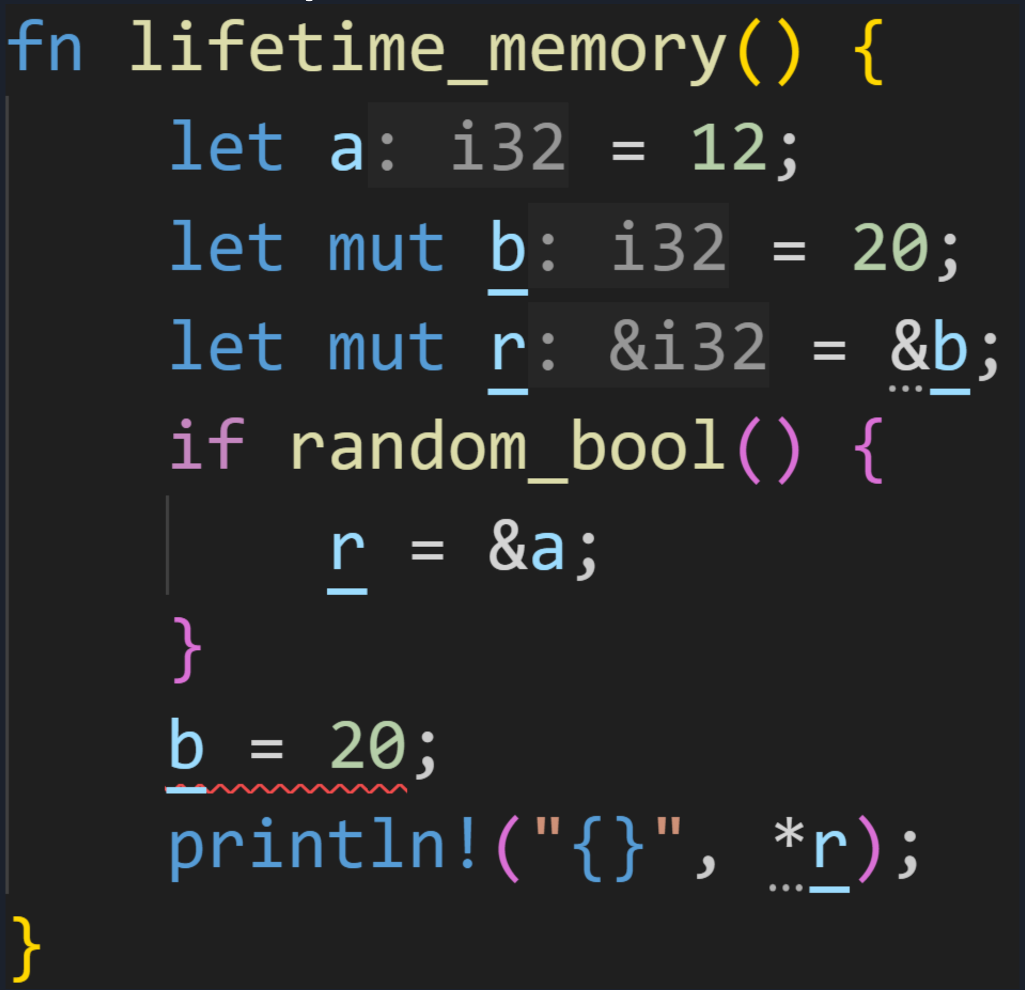
1. Recap



```
fn lifetime_violation() {
    let r: &i32;
    {
        let x:
        r = &x;
    }
    println!("{}", *r);
}
```

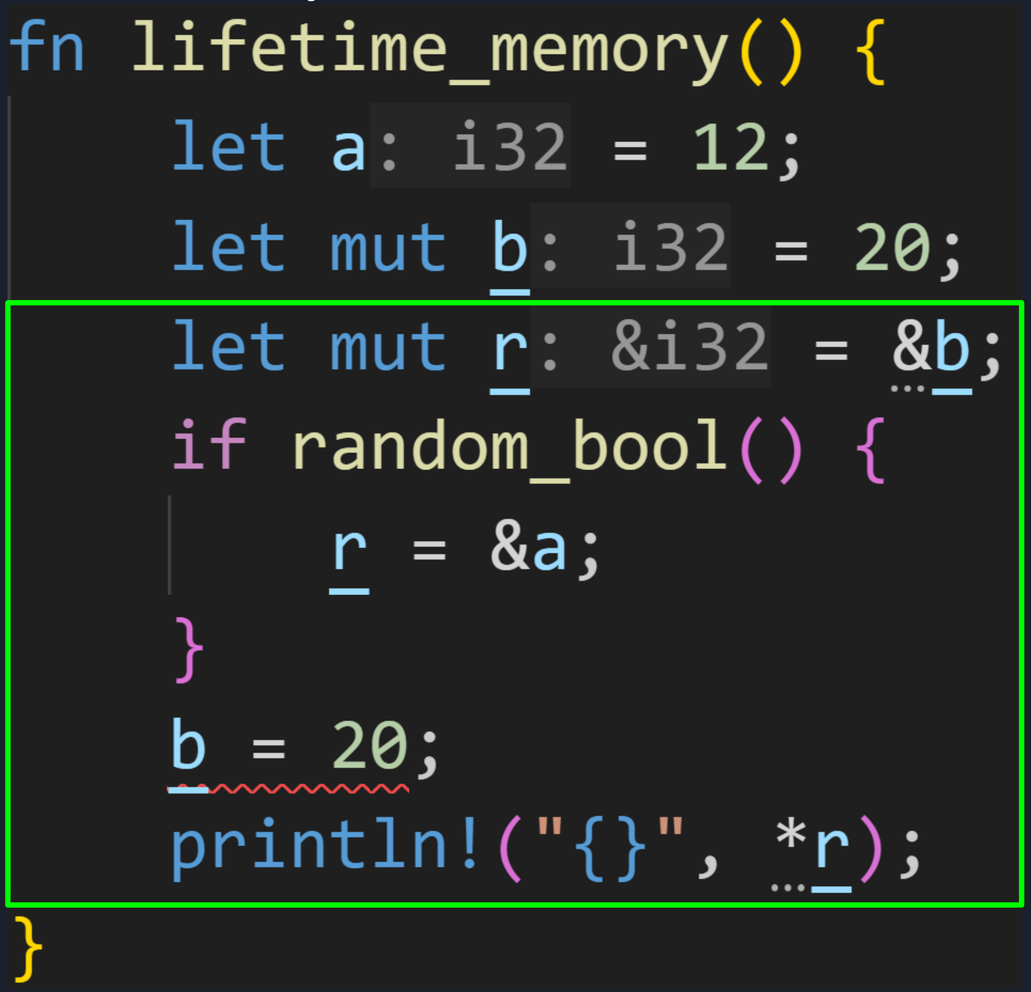
```
error[E0597]: `x` does not live long enough
 --> src\main.rs:6:13
  |
5 |         let x = 12;
  |             - binding `x` declared here
6 |         r = &x;
  |             ^^ borrowed value does not live long enough
7 |     }
  |     - `x` dropped here while still borrowed
8 |     println!("{}", *r);
  |                    -- borrow later used here
```

1. Recap



```
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
    }
    b = 20;
    println!("{}", *r);
}
```

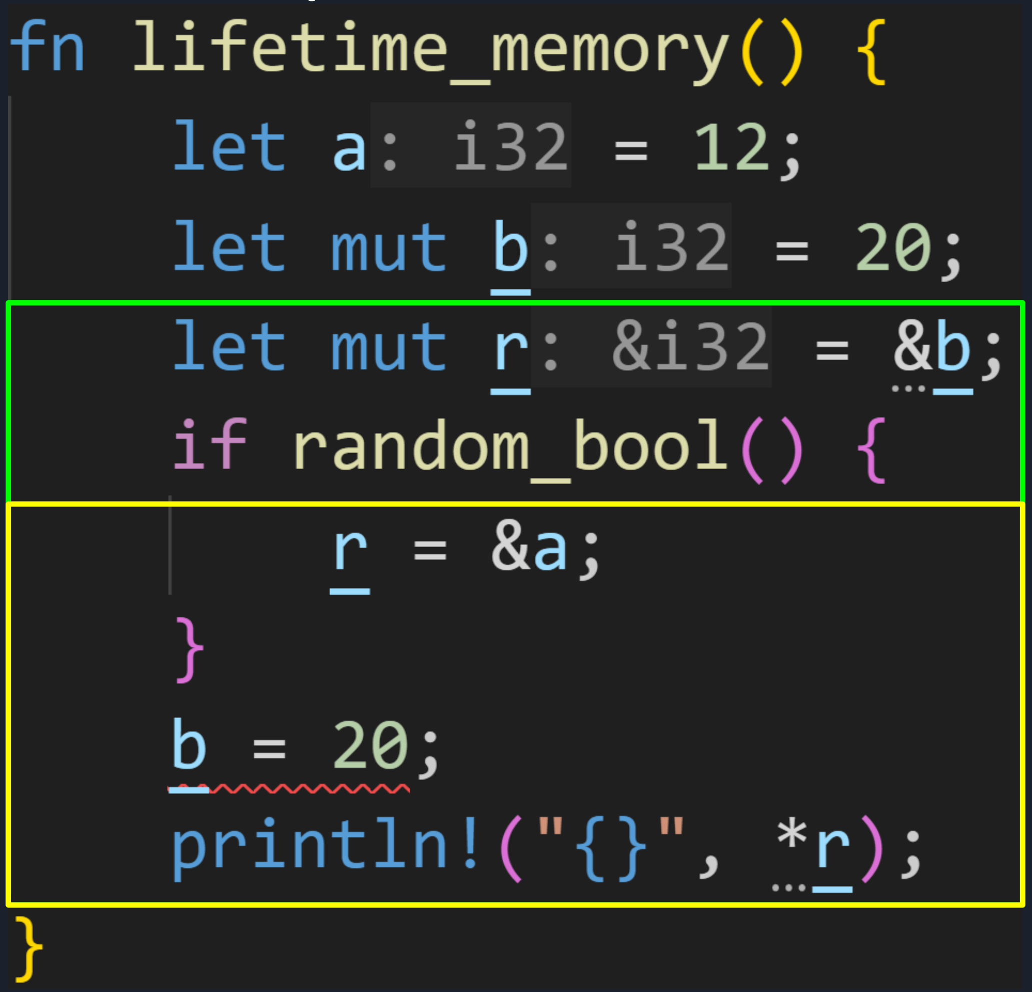
1. Recap



```rust
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
    }
    b = 20;
    println!("{}", *r);
}
```

Lifetime of &b

1. Recap

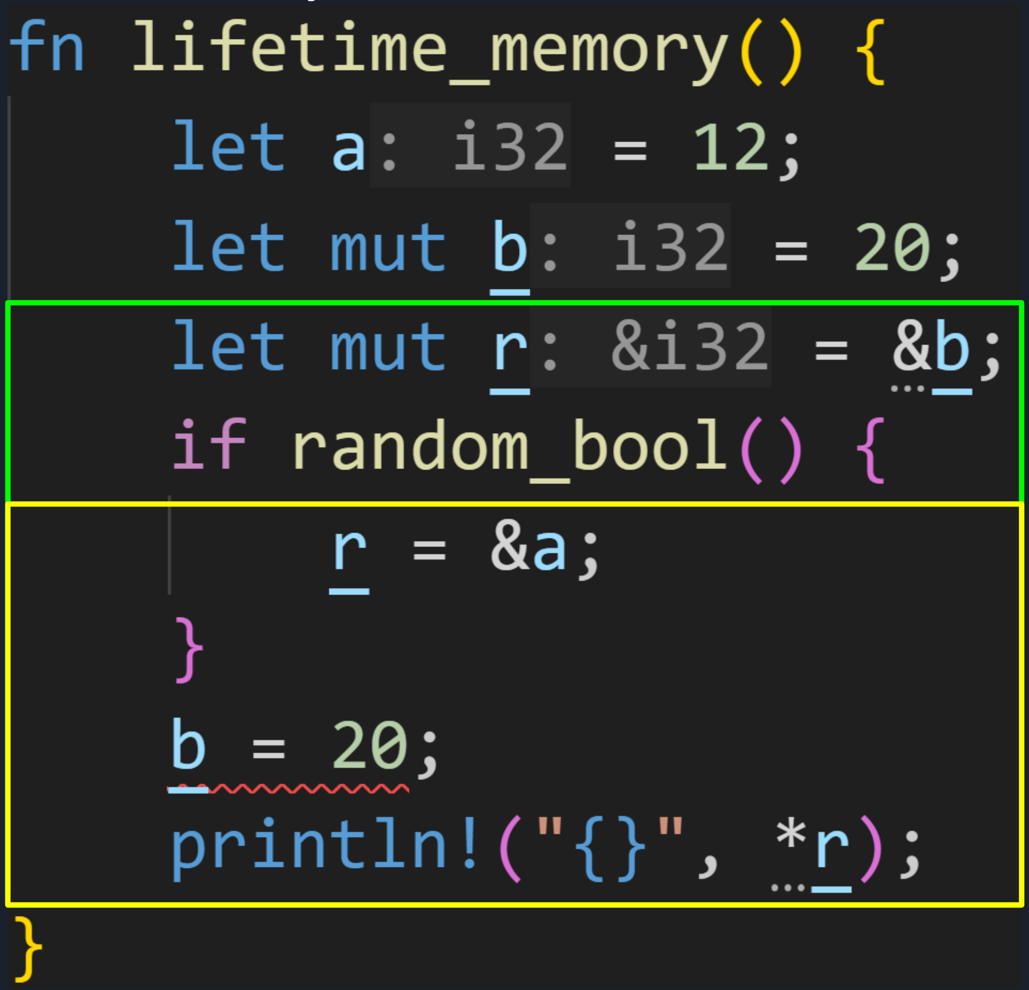


```
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
    }
    b = 20;
    println!("{}", *r);
}
```

Lifetime of &b

Lifetime of &a

1. Recap



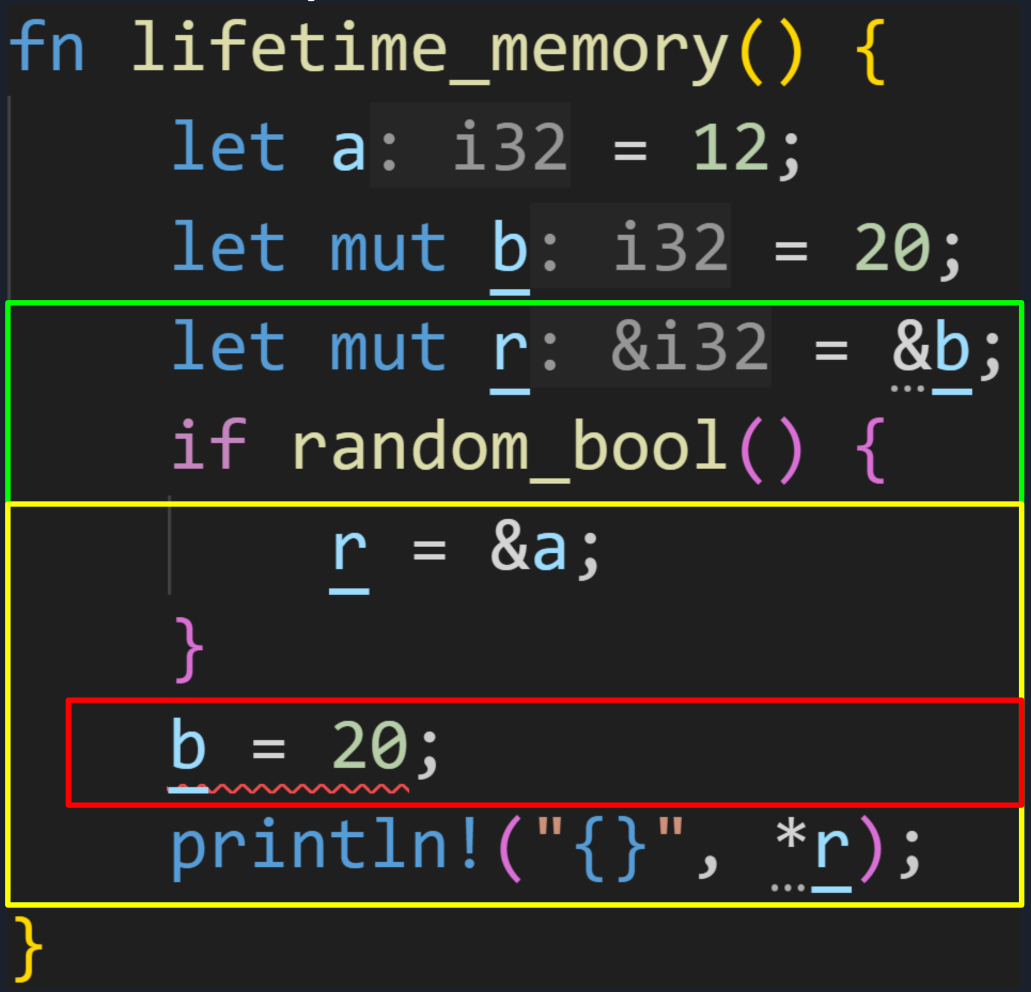
```rust
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
    }
    b = 20;
    println!("{}", *r);
}
```

Lifetime of &b

Lifetime of &a

Everything is fine, r does not outlive a and b, however...

1. Recap



```rust
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
    }
    b = 20;
    println!("{}", *r);
}
```

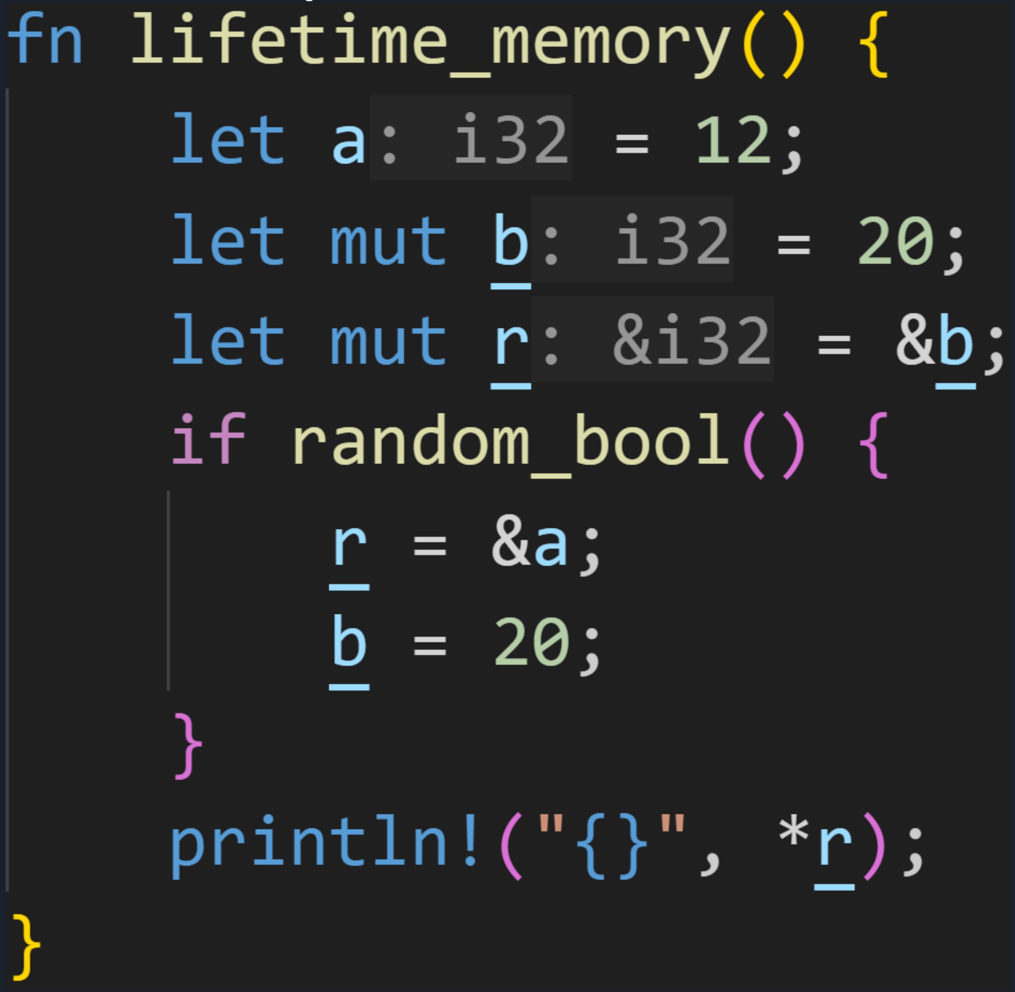
Lifetime of &b

Lifetime of &a

Everything is fine, r does not outlive a and b, however...

We modify b here, while it‘s borrowed! Not allowed.

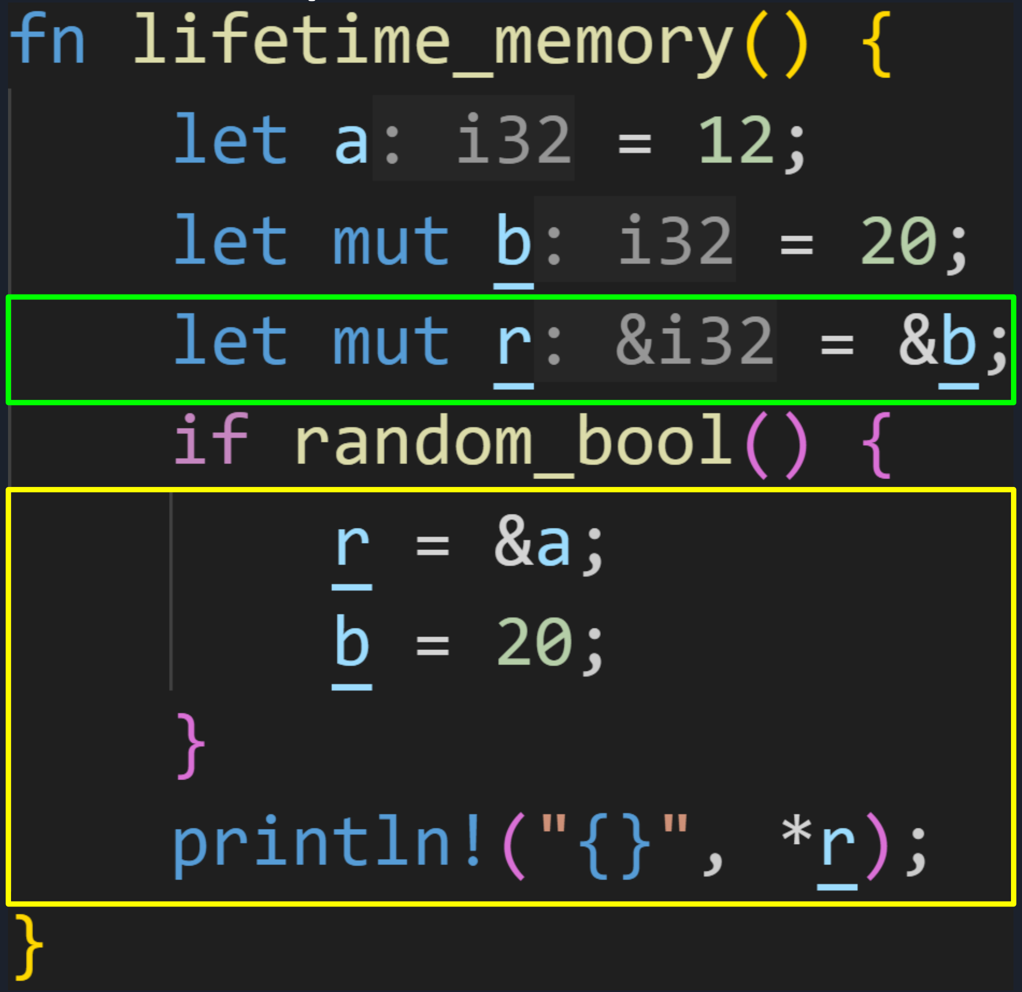
1. Recap



```
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
        b = 20;
    }
    println!("{}", *r);
}
```

Two things can happen here:
random_bool() is either true or false

1. Recap



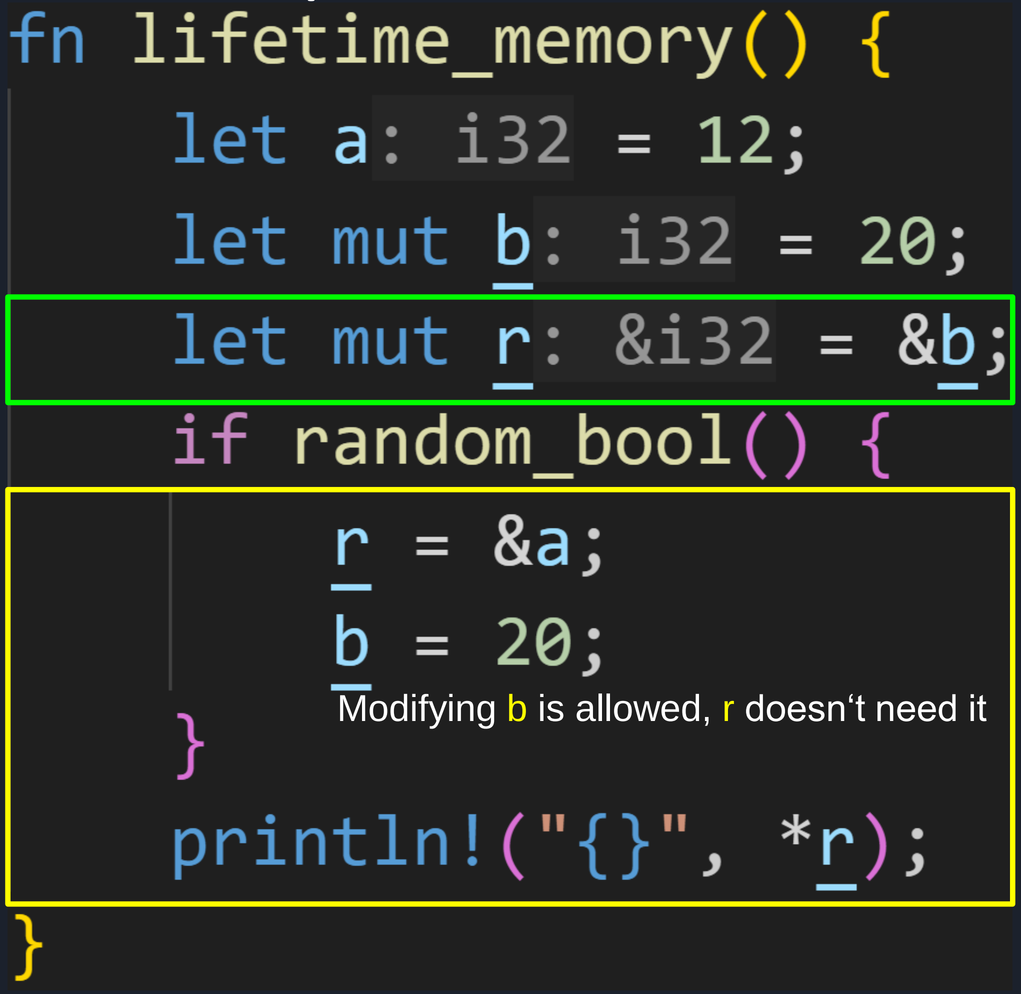
```
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
        b = 20;
    }
    println!("{}", *r);
}
```

Two things can happen here:
random_bool() is either true or false
→ TRUE

Lifetime of &b

Lifetime of &a

1. Recap



```
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
        b = 20;
    }
    println!("{}", *r);
}
```

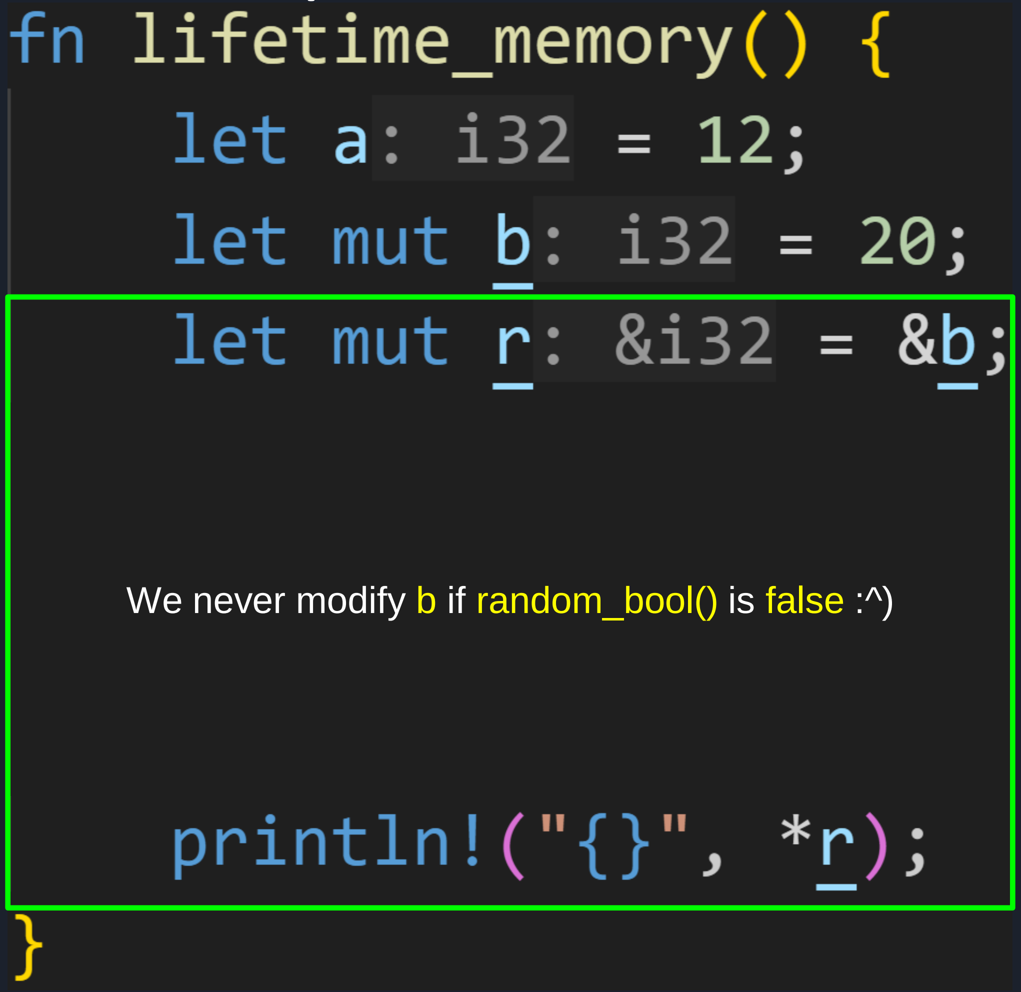
Modifying b is allowed, r doesn‘t need it

Two things can happen here:
random_bool() is either true or false
→ TRUE

Lifetime of &b

Lifetime of &a

1. Recap



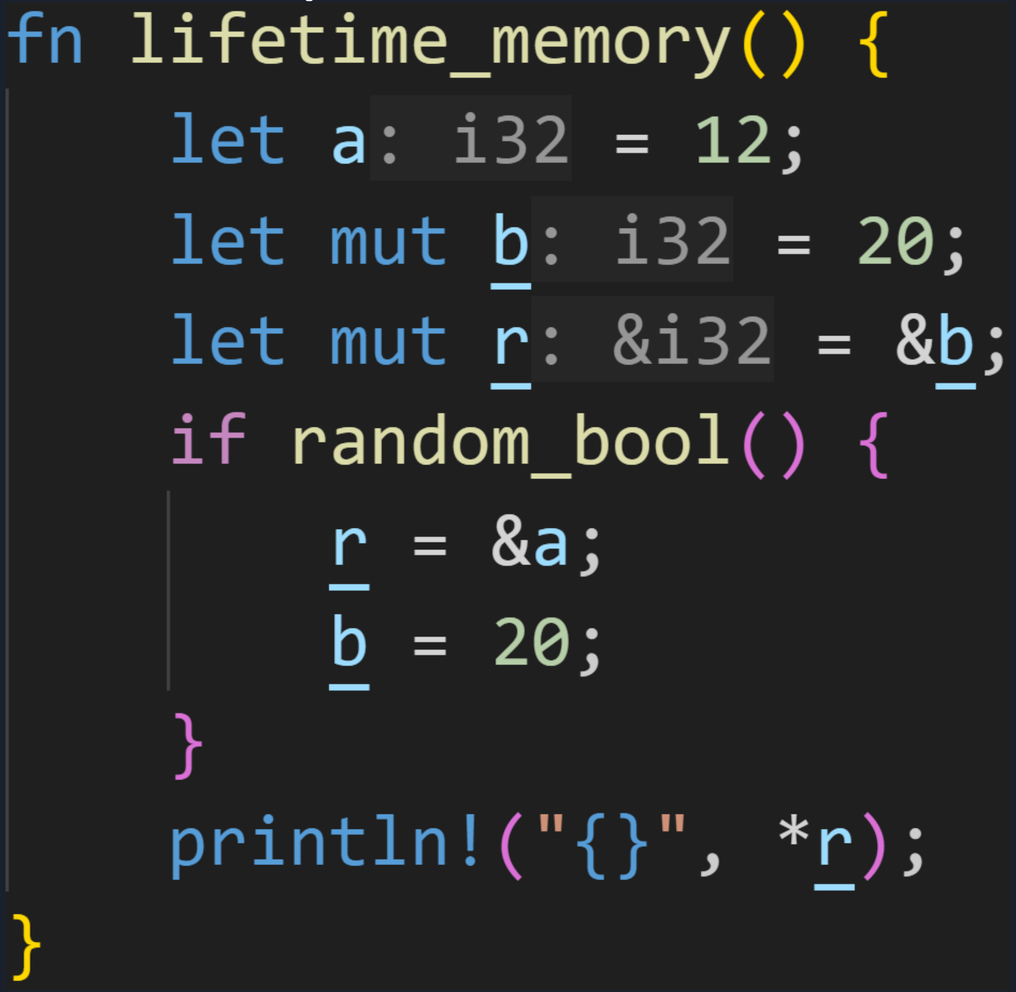
```
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;



    println!("{}", *r);
}
```

We never modify b if random_bool() is false :^)

Lifetime of &b

Two things can happen here:
random_bool() is either true or false
→ FALSE

1. Recap

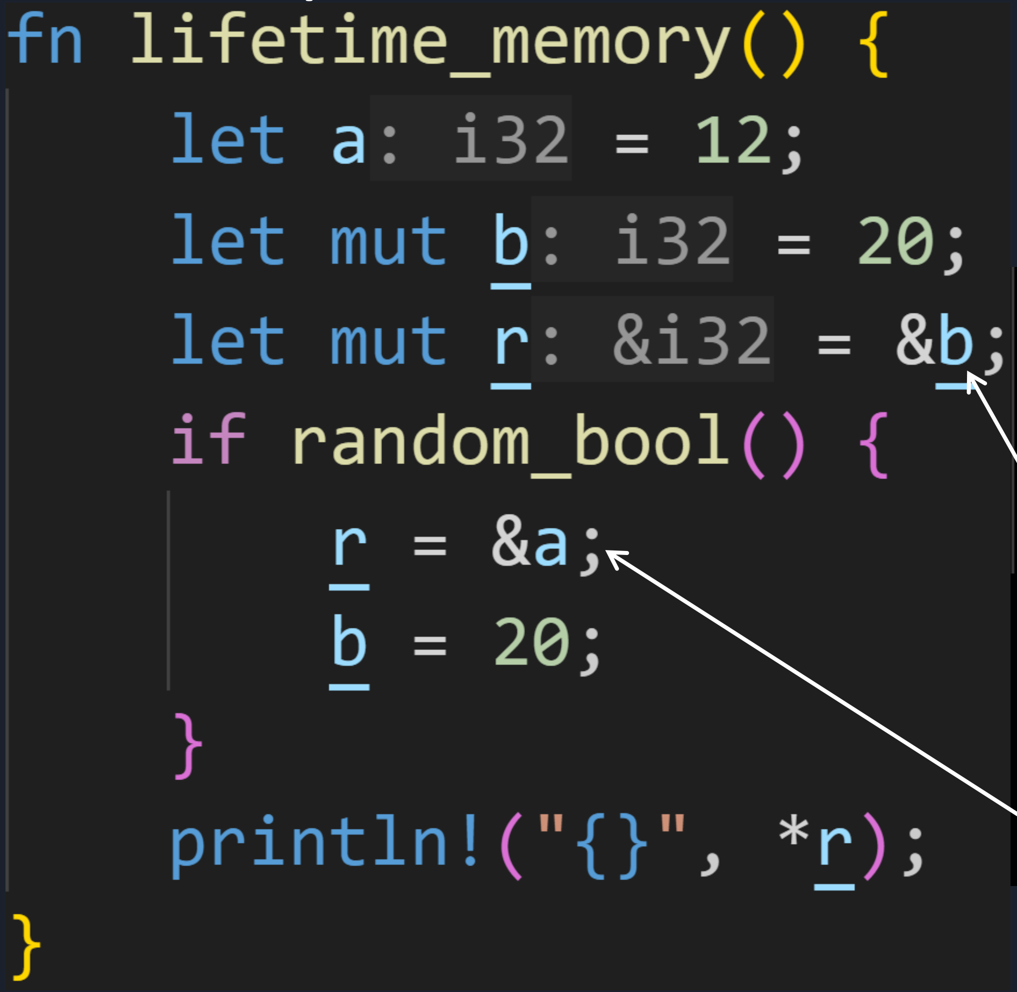


```
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
        b = 20;
    }
    println!("{}", *r);
}
```

Two things can happen here:
random_bool() is either true or false

In both cases, no lifetimes are violated
→ Code allowed

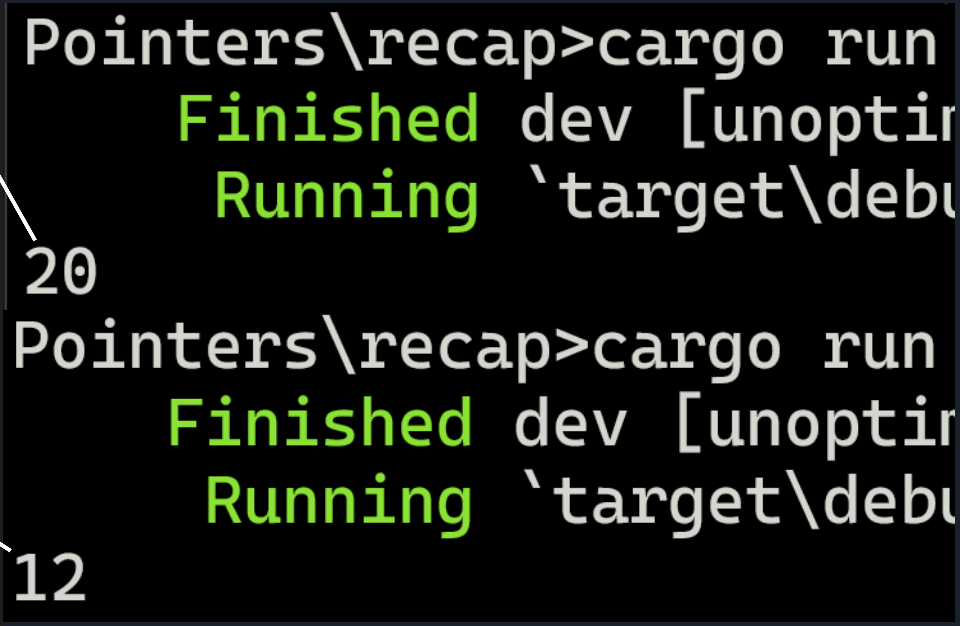
1. Recap



```rust
fn lifetime_memory() {
    let a: i32 = 12;
    let mut b: i32 = 20;
    let mut r: &i32 = &b;
    if random_bool() {
        r = &a;
        b = 20;
    }
    println!("{}", *r);
}
```

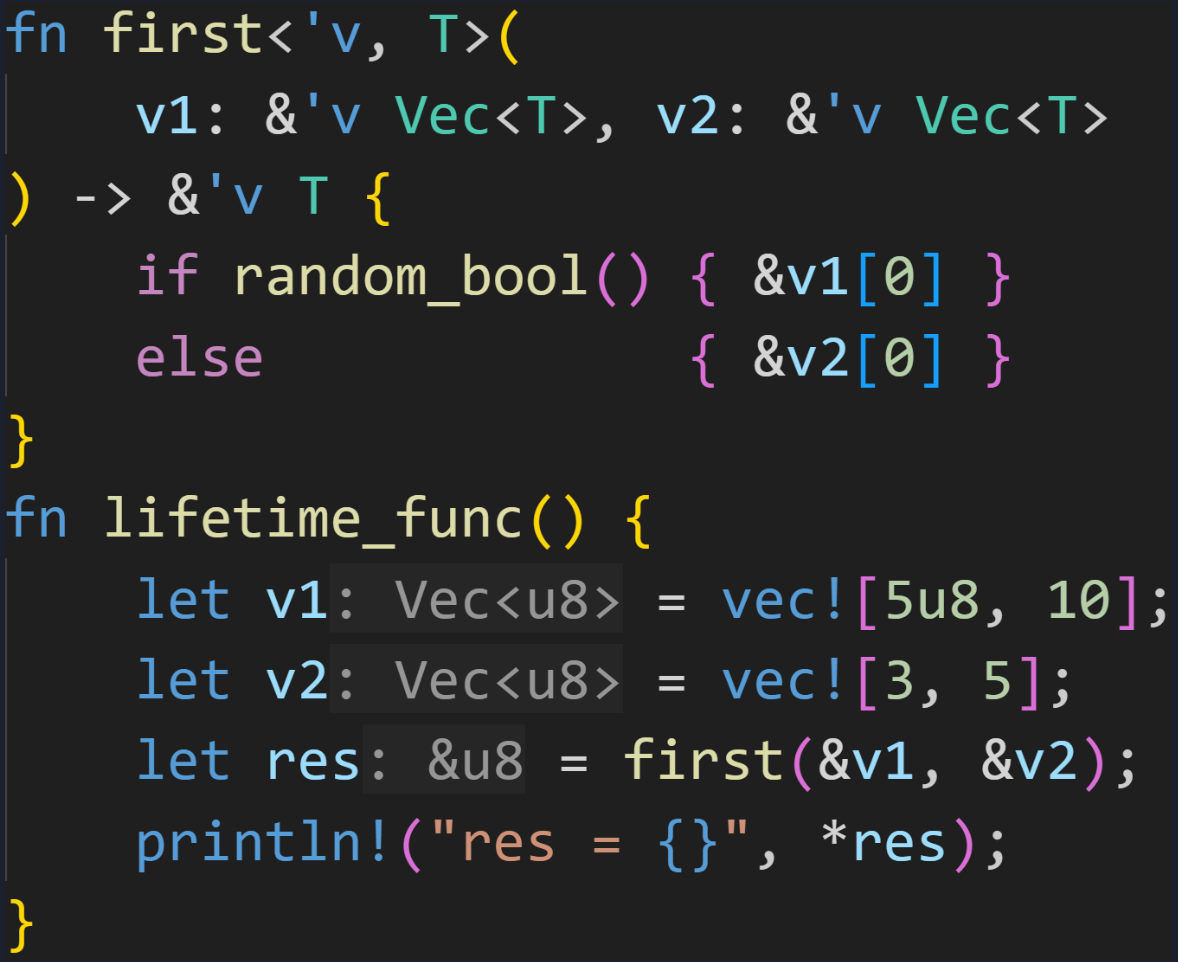
Two things can happen here:
random_bool() is either true or false

In both cases, no lifetimes are violated
→ Code allowed



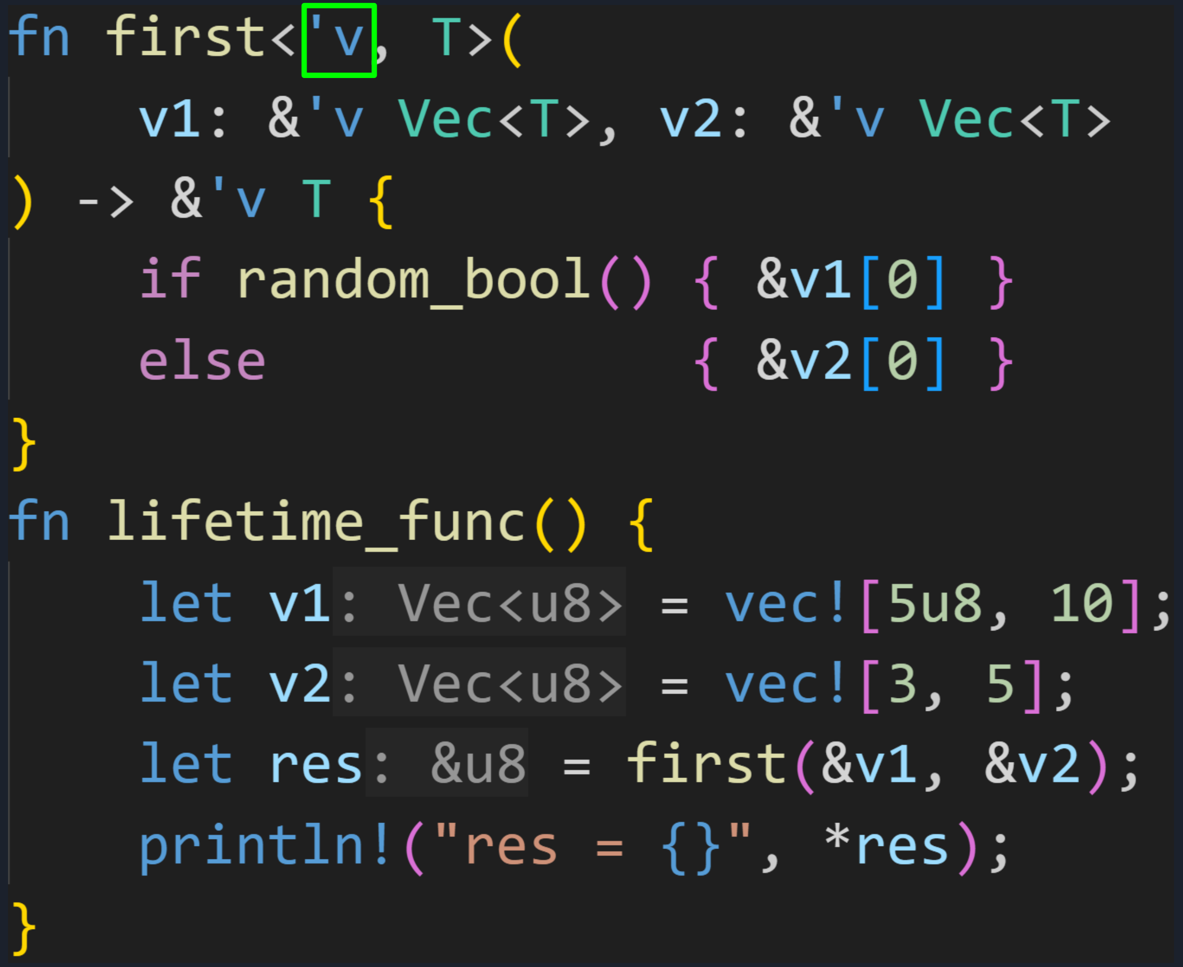
```
Pointers\recap>cargo run
    Finished dev [unoptim
     Running `target\debu
20
Pointers\recap>cargo run
    Finished dev [unoptim
     Running `target\debu
12
```

1. Recap



```rust
fn first<'v, T>(
    v1: &'v Vec<T>, v2: &'v Vec<T>
) -> &'v T {
    if random_bool() { &v1[0] }
    else             { &v2[0] }
}
fn lifetime_func() {
    let v1: Vec<u8> = vec![5u8, 10];
    let v2: Vec<u8> = vec![3, 5];
    let res: &u8 = first(&v1, &v2);
    println!("res = {}", *res);
}
```

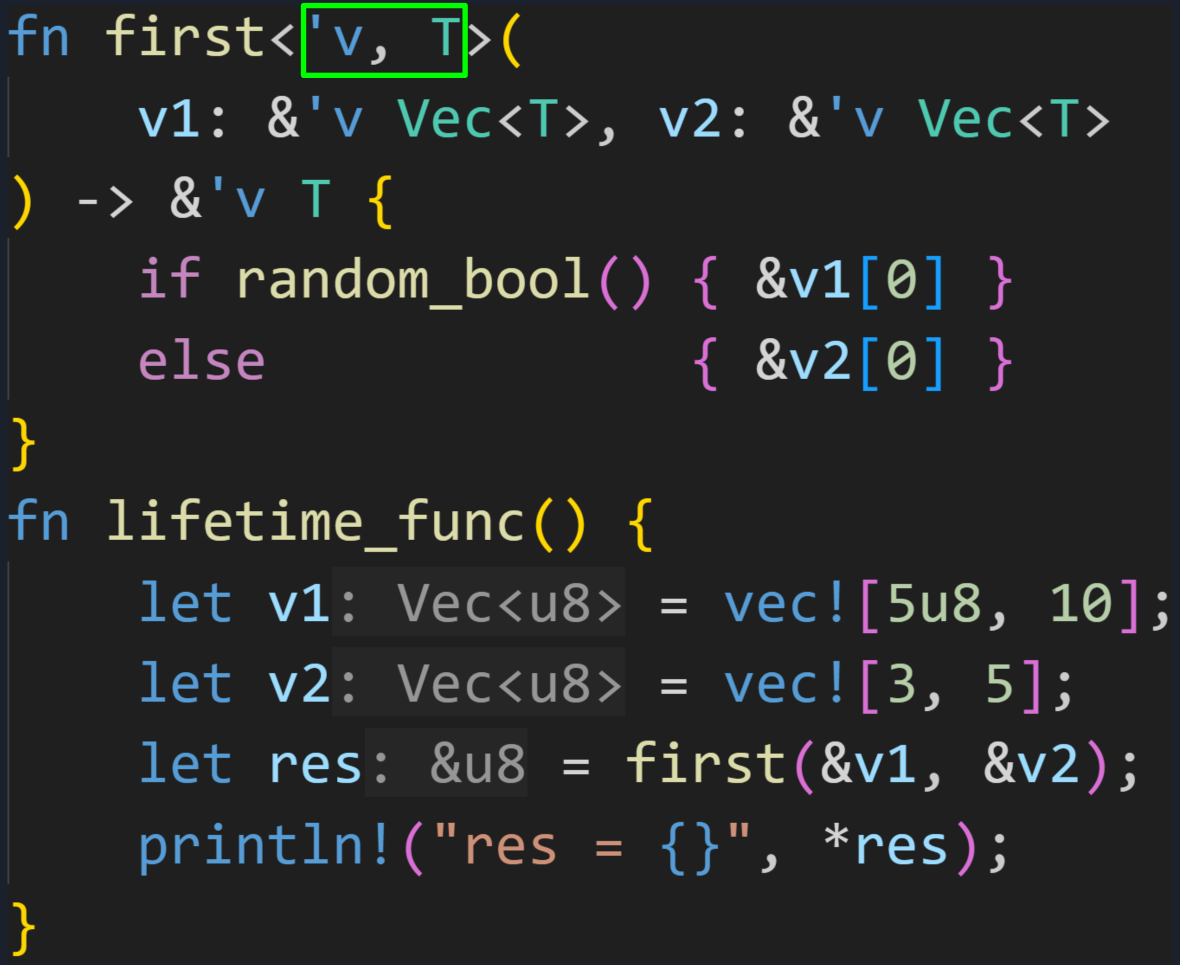
1. Recap



```
fn first<'v, T>(
    v1: &'v Vec<T>, v2: &'v Vec<T>
) -> &'v T {
    if random_bool() { &v1[0] }
    else             { &v2[0] }
}
fn lifetime_func() {
    let v1: Vec<u8> = vec![5u8, 10];
    let v2: Vec<u8> = vec![3, 5];
    let res: &u8 = first(&v1, &v2);
    println!("res = {}", *res);
}
```

Named lifetime parameter

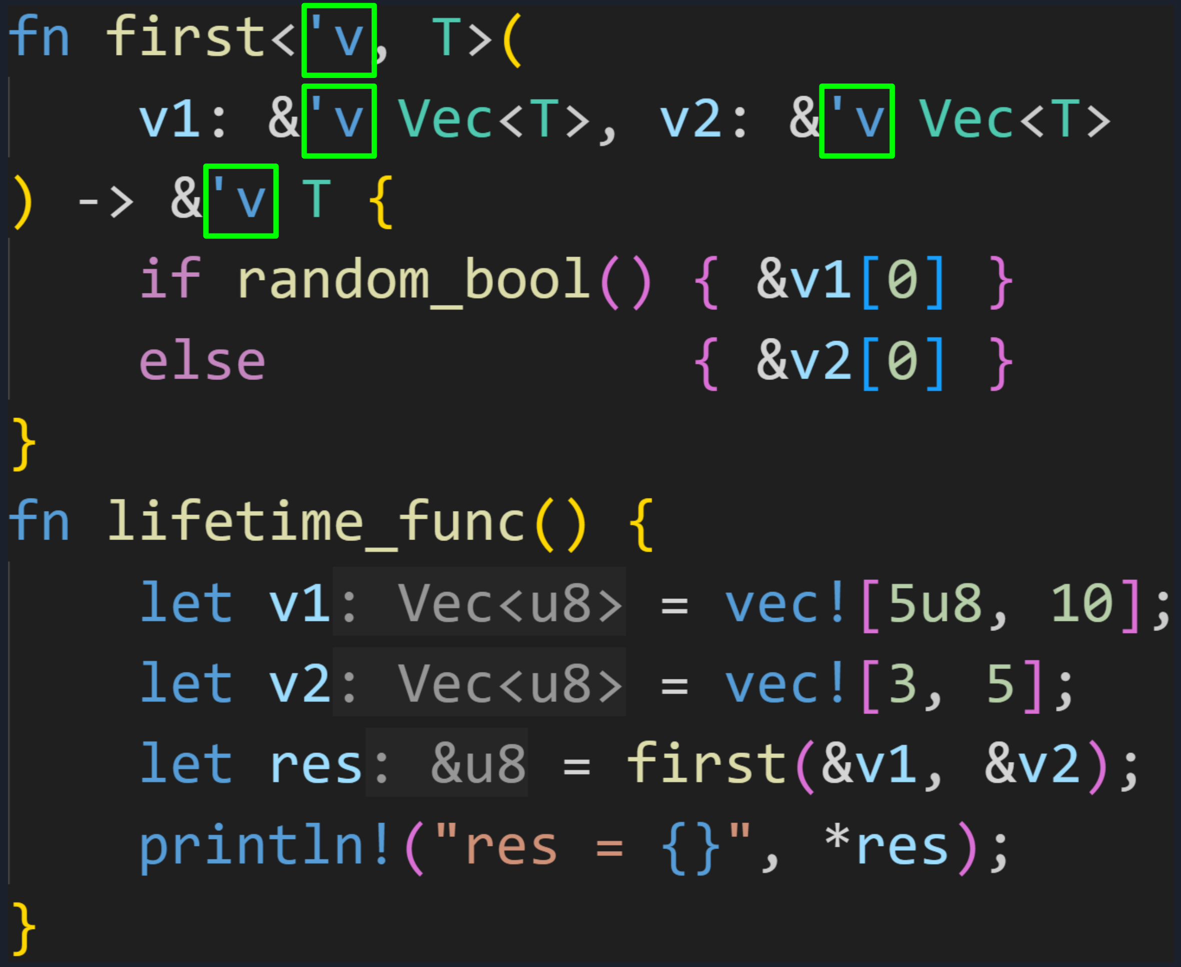
1. Recap



```
fn first<'v, T>(
    v1: &'v Vec<T>, v2: &'v Vec<T>
) -> &'v T {
    if random_bool() { &v1[0] }
    else             { &v2[0] }
}
fn lifetime_func() {
    let v1: Vec<u8> = vec![5u8, 10];
    let v2: Vec<u8> = vec![3, 5];
    let res: &u8 = first(&v1, &v2);
    println!("res = {}", *res);
}
```

Lifetime parameters and generic parameters can be used together, but lifetimes have to come first!

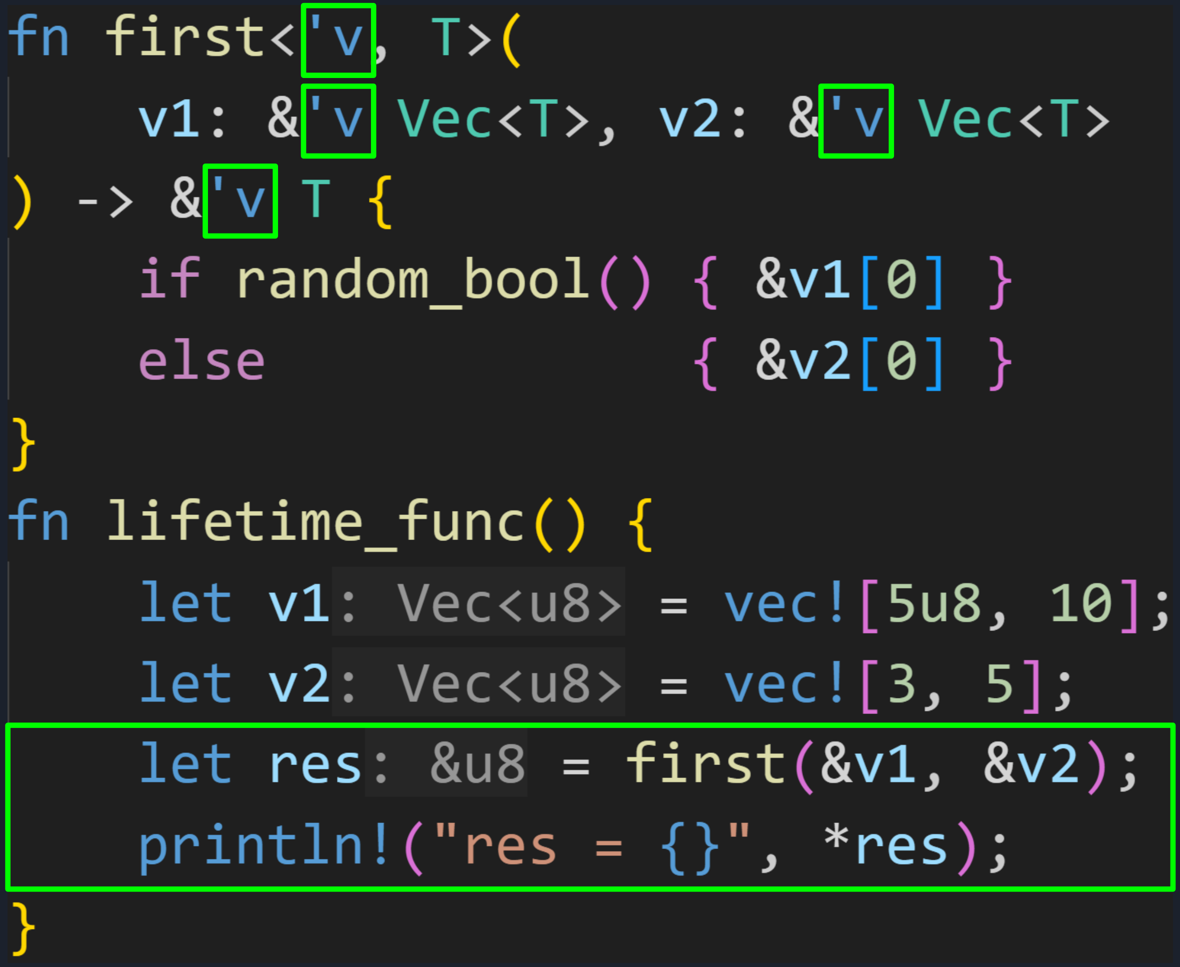
1. Recap



```
fn first<'v, T>(
    v1: &'v Vec<T>, v2: &'v Vec<T>
) -> &'v T {
    if random_bool() { &v1[0] }
    else             { &v2[0] }
}
fn lifetime_func() {
    let v1: Vec<u8> = vec![5u8, 10];
    let v2: Vec<u8> = vec![3, 5];
    let res: &u8 = first(&v1, &v2);
    println!("res = {}", *res);
}
```

For a small moment in time, there exists a lifetime which borrows from both v1 and v2 at the same time

1. Recap



```
fn first<'v, T>(
    v1: &'v Vec<T>, v2: &'v Vec<T>
) -> &'v T {
    if random_bool() { &v1[0] }
    else             { &v2[0] }
}
fn lifetime_func() {
    let v1: Vec<u8> = vec![5u8, 10];
    let v2: Vec<u8> = vec![3, 5];
    let res: &u8 = first(&v1, &v2);
    println!("res = {}", *res);
}
```

For a small moment in time, there exists a lifetime which borrows from both v1 and v2 at the same time

In this region, we also borrowed v1 and v2

2. Slices

2. Slices

- Imagine you want to write an efficient tokenizer
  - Tokenizer: Turns some source text into tokens for further processing, e.g. parsing

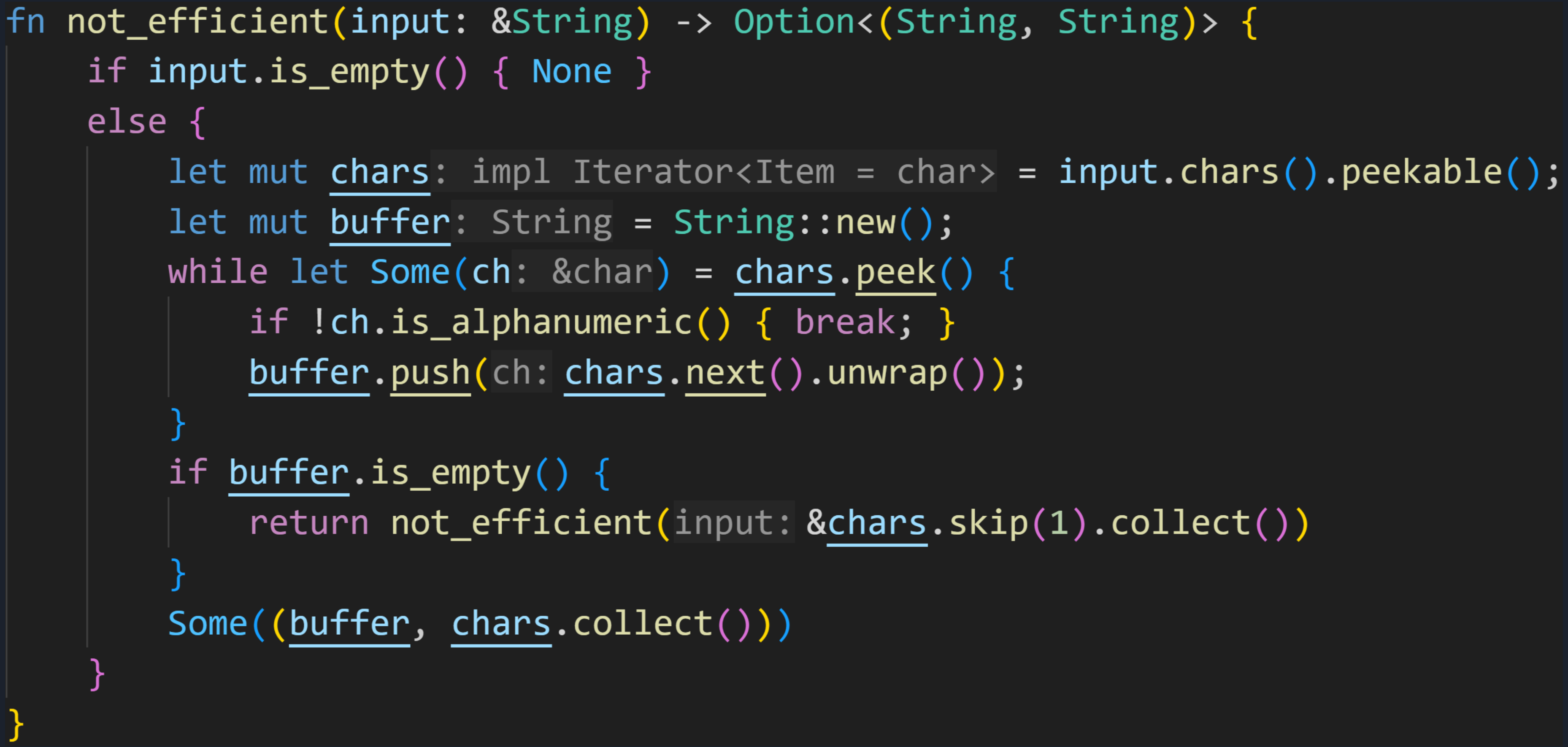
2. Slices

- Imagine you want to write an efficient tokenizer
- Given a source text (a String), we want to get a sequence of words
  - Word: Any sequence of alphanumeric (a, 1, U) characters
  - We want to ignore all other characters, and skip them

2. Slices

- Imagine you want to write an efficient tokenizer
- Given a source text (a String), we want to get a sequence of words
- Example: „Hello, how are you?“ → [„Hello“, „how“, „are“, „you“]

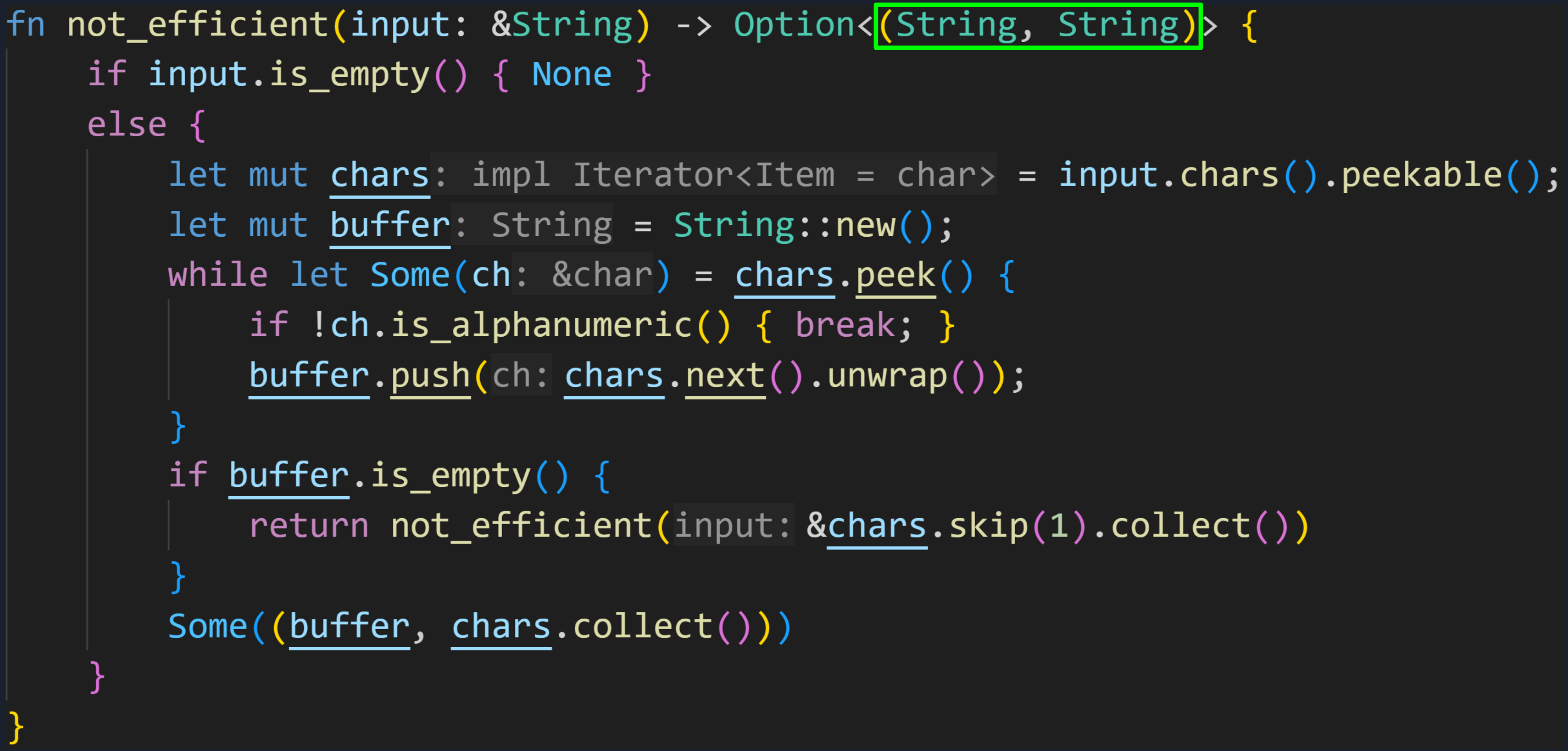
2. Slices



```rust
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

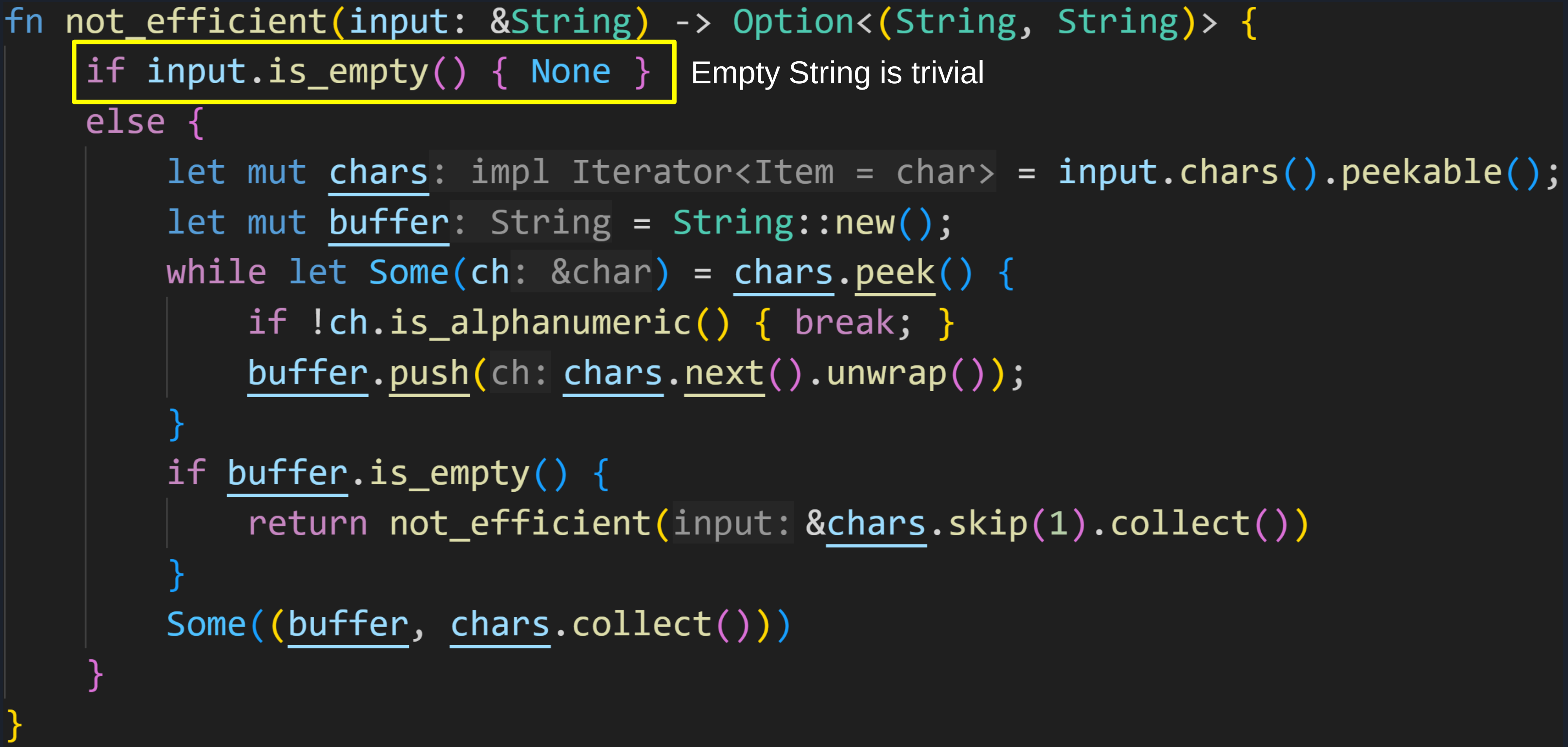
2. Slices

Return a pair of (word, rest of input)



```
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

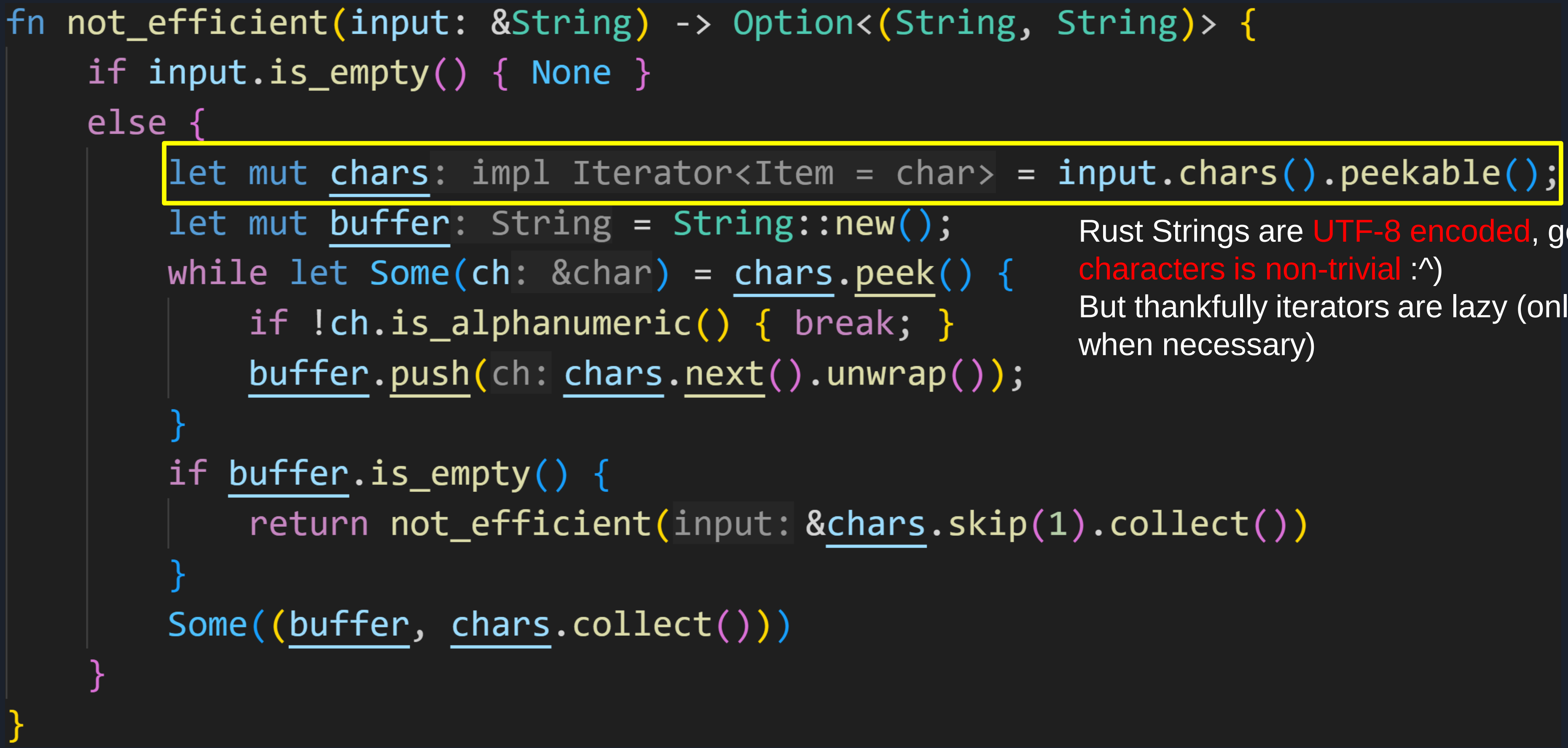
2. Slices



```rust
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

Empty String is trivial

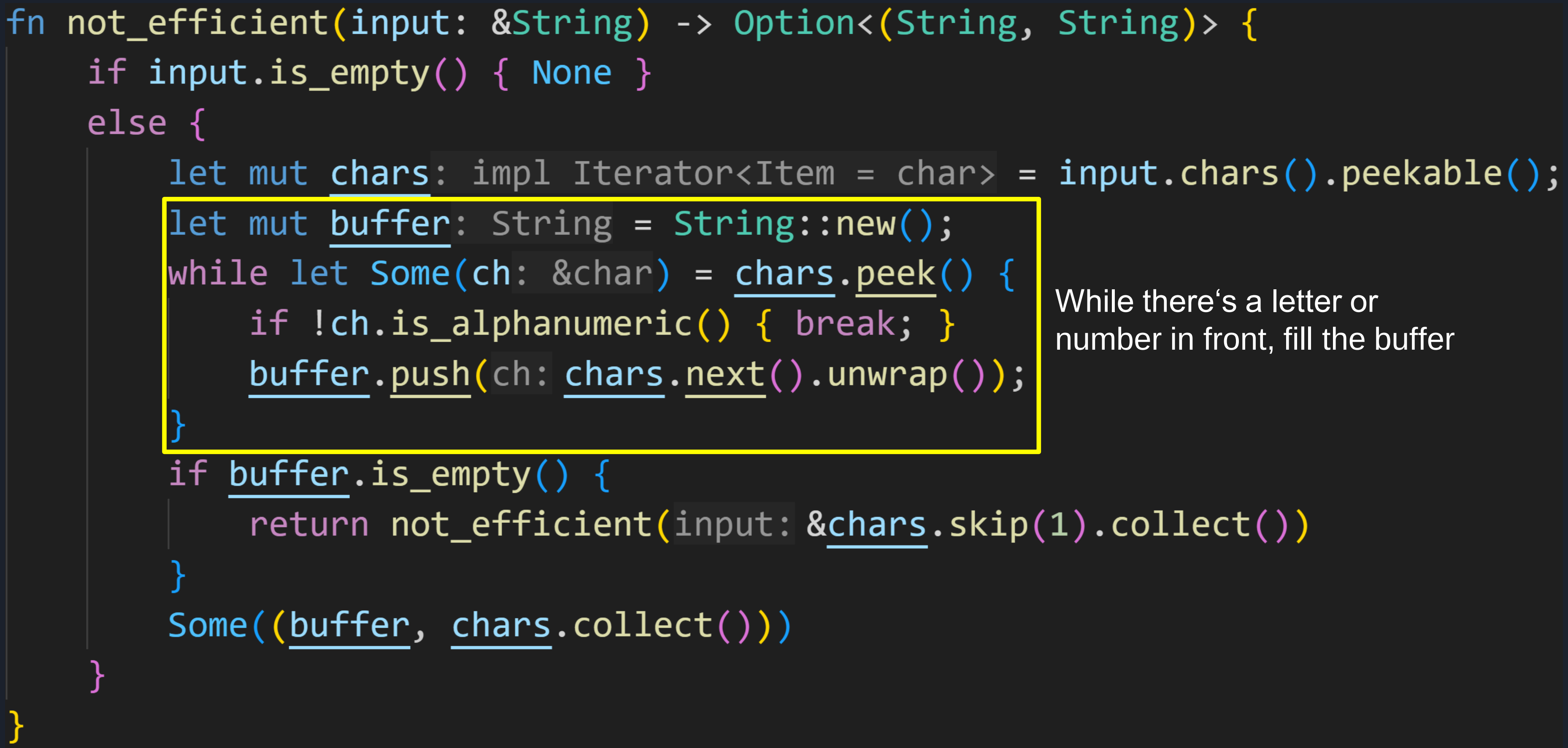
2. Slices



```
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

Rust Strings are UTF-8 encoded, getting the characters is non-trivial :^)
But thankfully iterators are lazy (only compute when necessary)

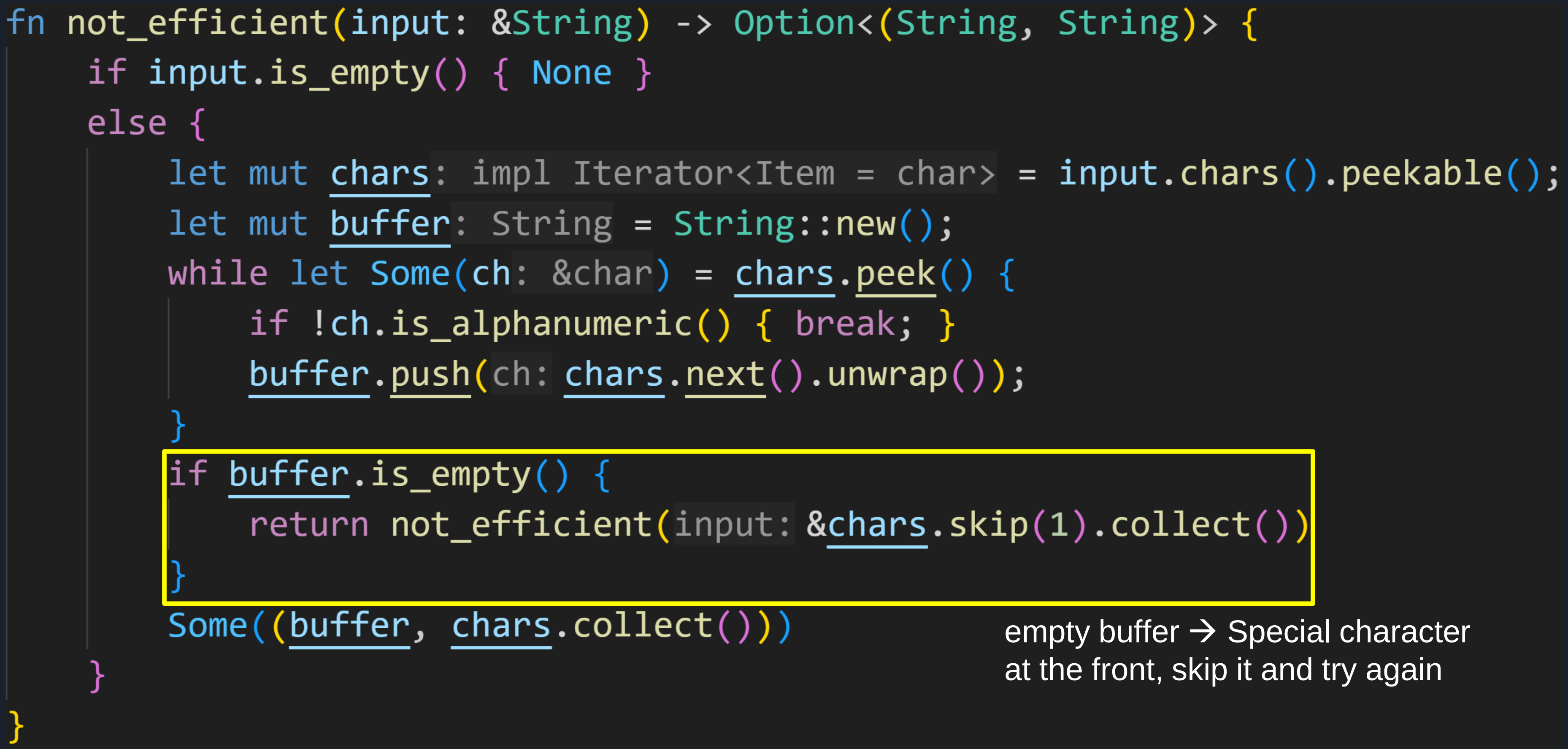
2. Slices



```
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

While there‘s a letter or number in front, fill the buffer

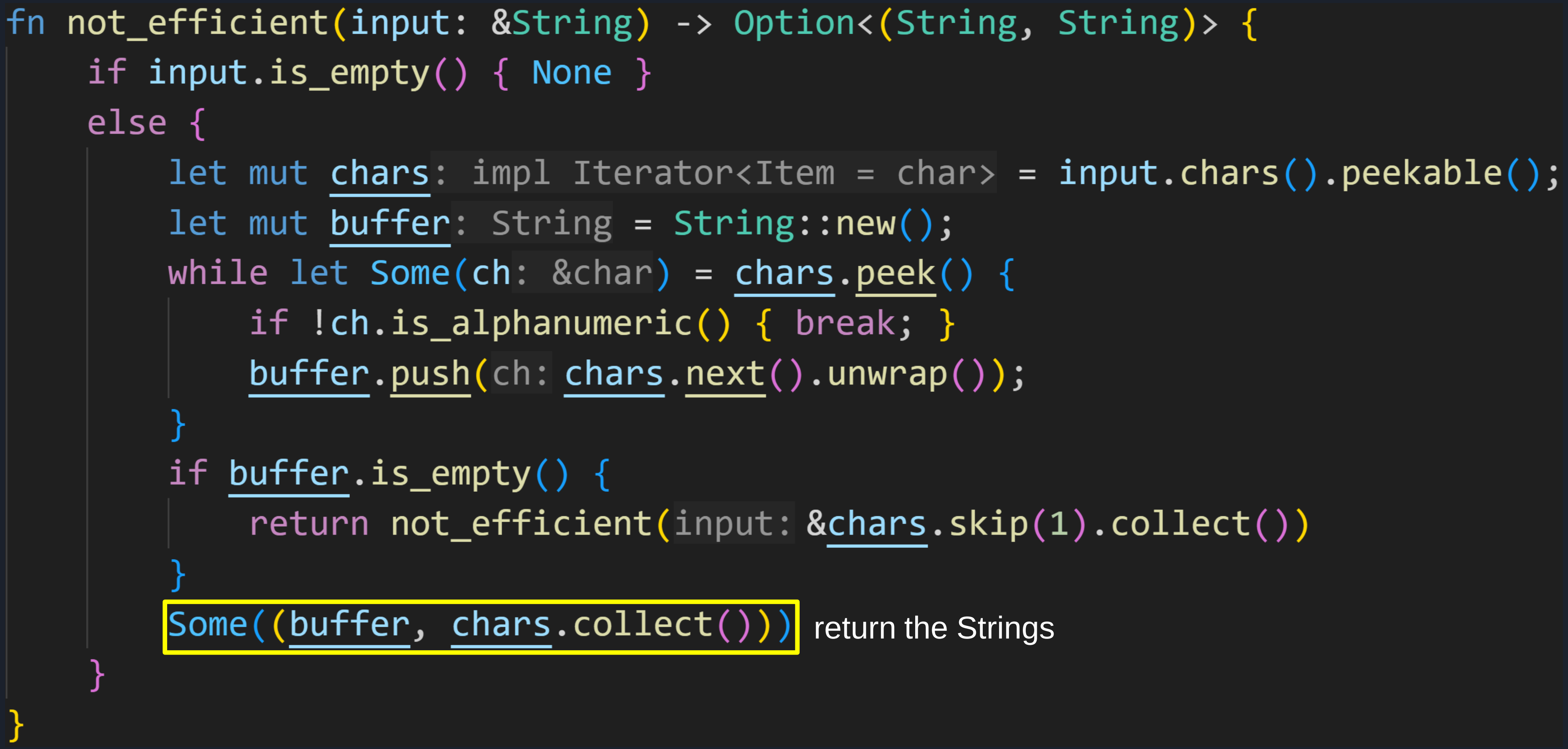
2. Slices



```
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

empty buffer → Special character at the front, skip it and try again

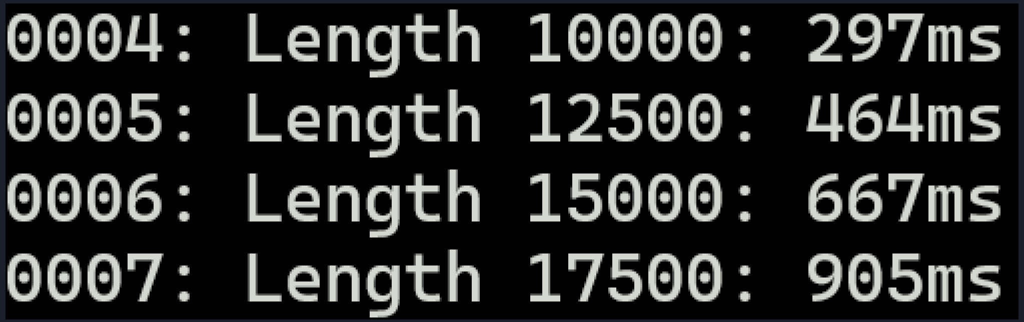
2. Slices



```rust
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

return the Strings

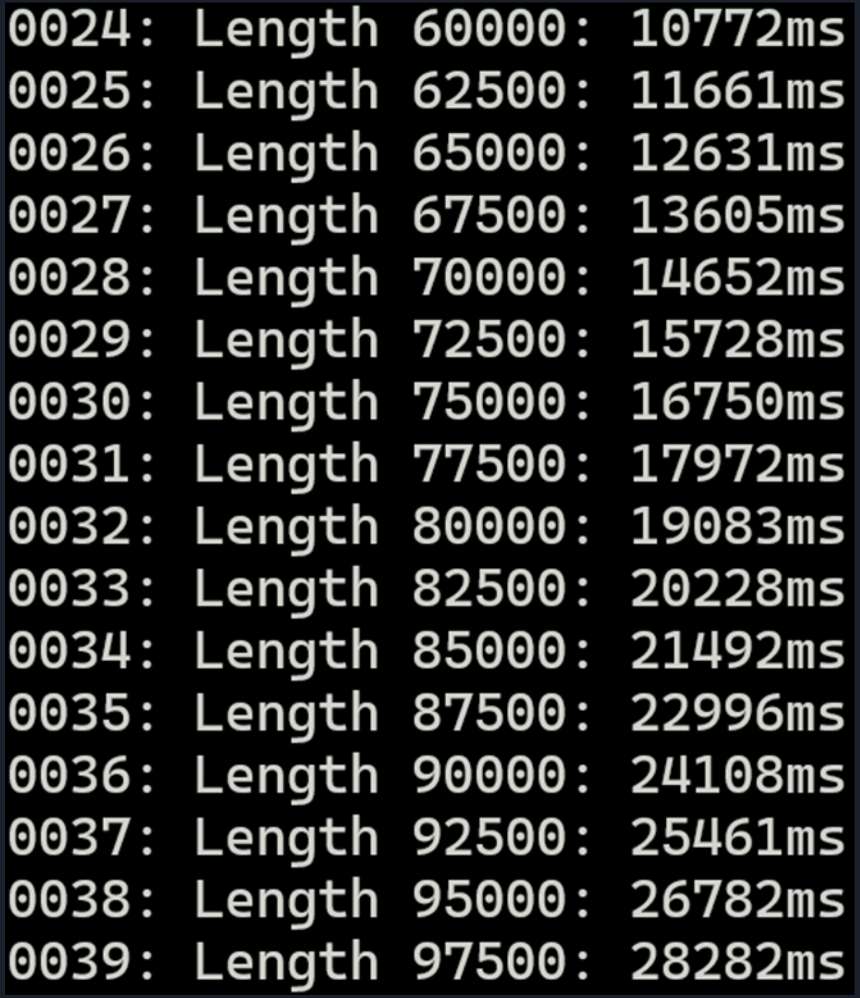
2. Slices



```
0004: Length 10000: 297ms
0005: Length 12500: 464ms
0006: Length 15000: 667ms
0007: Length 17500: 905ms
```

A second to tokenize a String of 20.000 characters!!

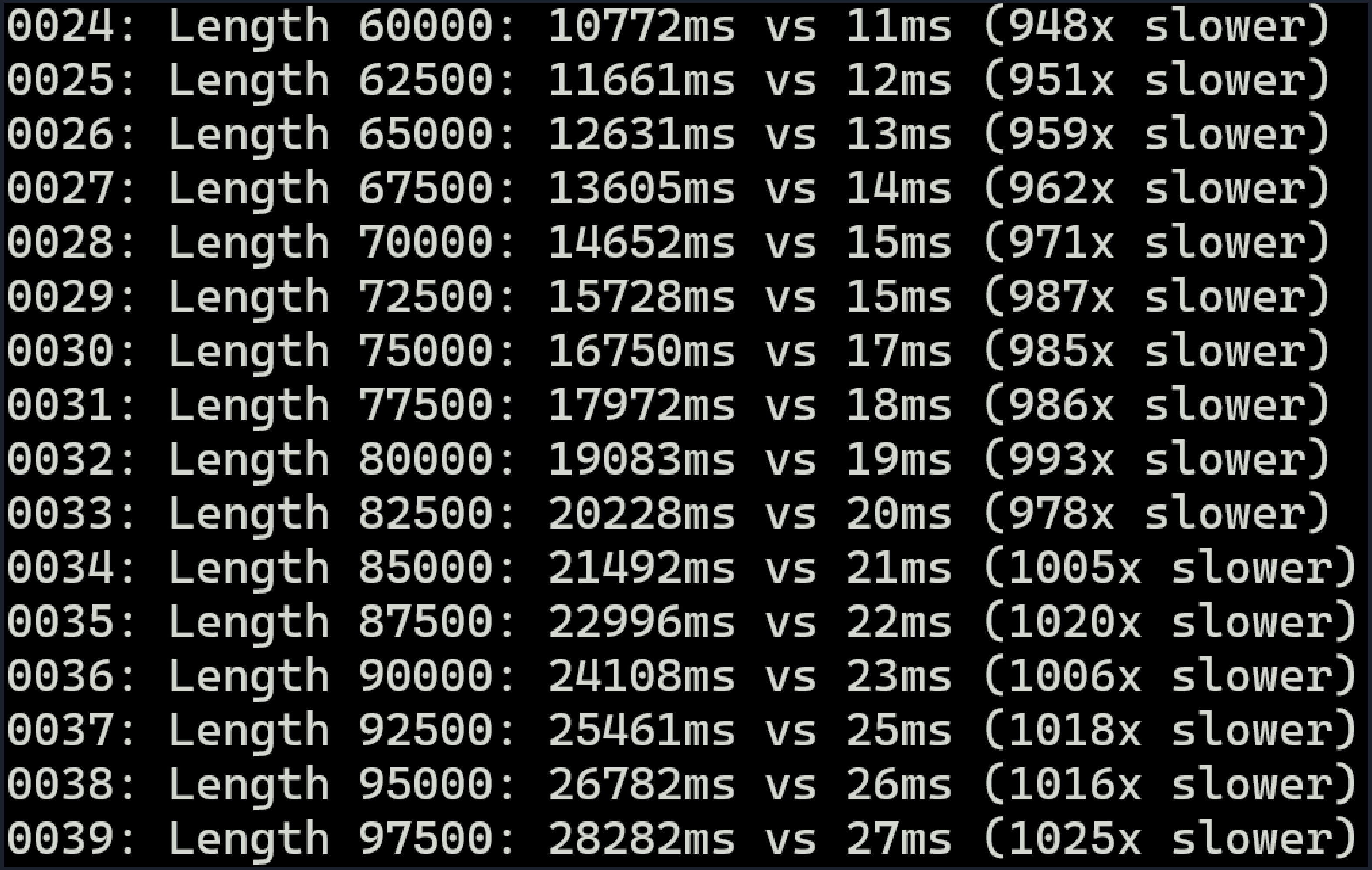
2. Slices



```
0024: Length 60000: 10772ms
0025: Length 62500: 11661ms
0026: Length 65000: 12631ms
0027: Length 67500: 13605ms
0028: Length 70000: 14652ms
0029: Length 72500: 15728ms
0030: Length 75000: 16750ms
0031: Length 77500: 17972ms
0032: Length 80000: 19083ms
0033: Length 82500: 20228ms
0034: Length 85000: 21492ms
0035: Length 87500: 22996ms
0036: Length 90000: 24108ms
0037: Length 92500: 25461ms
0038: Length 95000: 26782ms
0039: Length 97500: 28282ms
```

This is fine...

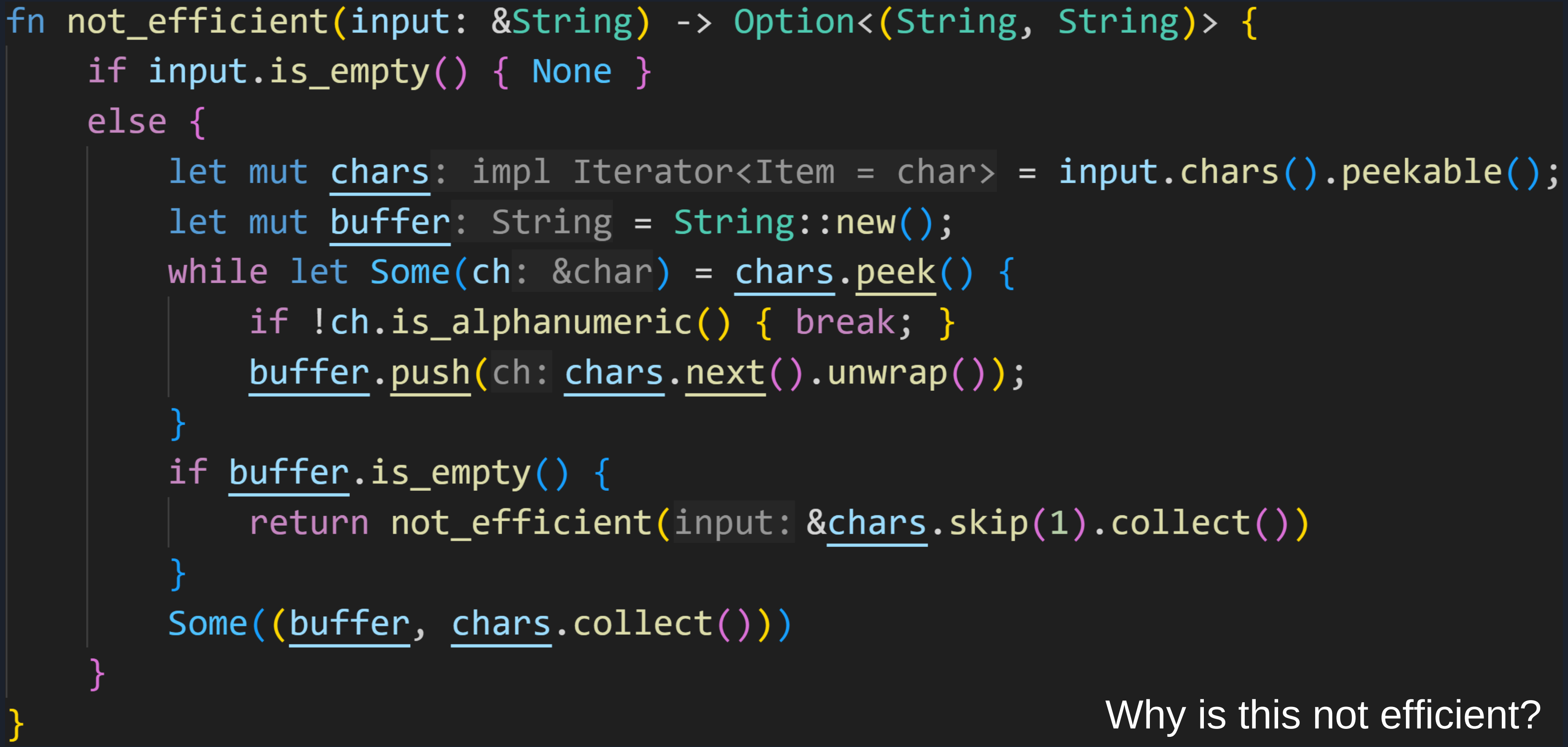
2. Slices



```
0024: Length 60000: 10772ms vs 11ms (948x slower)
0025: Length 62500: 11661ms vs 12ms (951x slower)
0026: Length 65000: 12631ms vs 13ms (959x slower)
0027: Length 67500: 13605ms vs 14ms (962x slower)
0028: Length 70000: 14652ms vs 15ms (971x slower)
0029: Length 72500: 15728ms vs 15ms (987x slower)
0030: Length 75000: 16750ms vs 17ms (985x slower)
0031: Length 77500: 17972ms vs 18ms (986x slower)
0032: Length 80000: 19083ms vs 19ms (993x slower)
0033: Length 82500: 20228ms vs 20ms (978x slower)
0034: Length 85000: 21492ms vs 21ms (1005x slower)
0035: Length 87500: 22996ms vs 22ms (1020x slower)
0036: Length 90000: 24108ms vs 23ms (1006x slower)
0037: Length 92500: 25461ms vs 25ms (1018x slower)
0038: Length 95000: 26782ms vs 26ms (1016x slower)
0039: Length 97500: 28282ms vs 27ms (1025x slower)
```

But we can do better :^)

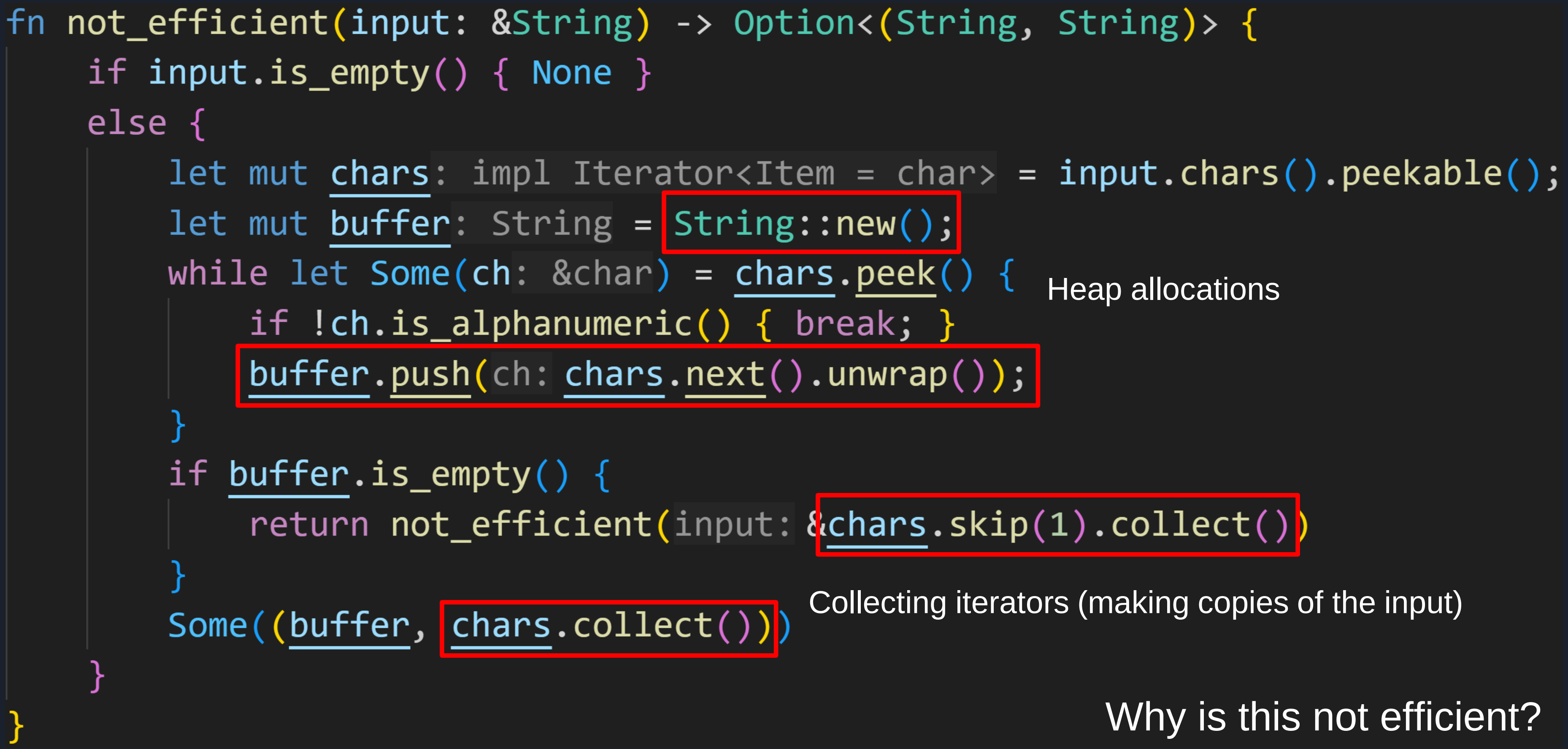
2. Slices



```rust
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

Why is this not efficient?

2. Slices



```rust
fn not_efficient(input: &String) -> Option<(String, String)> {
    if input.is_empty() { None }
    else {
        let mut chars: impl Iterator<Item = char> = input.chars().peekable();
        let mut buffer: String = String::new();
        while let Some(ch: &char) = chars.peek() {
            if !ch.is_alphanumeric() { break; }
            buffer.push(ch: chars.next().unwrap());
        }
        if buffer.is_empty() {
            return not_efficient(input: &chars.skip(1).collect())
        }
        Some((buffer, chars.collect()))
    }
}
```

Heap allocations

Collecting iterators (making copies of the input)

Why is this not efficient?

2. Slices

- What‘s the problem?

2. Slices

- What‘s the problem?
  - There are many problems, but the biggest problem is creating new Strings

2. Slices

- What‘s the problem?
  - There are many problems, but the biggest problem is creating new Strings
  - Wouldn‘t it be nice if we could reuse the original string? Maybe point into it? Take substrings?

2. Slices

- What‘s the problem?
  - There are many problems, but the biggest problem is creating new Strings
  - Wouldn‘t it be nice if we could reuse the original string? Maybe point into it? Take substrings?
    - Yes, of course we can, and we should

2. Slices

- Rustdocs: „Slices let you reference contiguous sequences in collections, instead of the whole collection“

2. Slices

- Rustdocs: „Slices let you reference contiguous sequences in collections, instead of the whole collection"
- The most commonly seen form of Slices is the String Slice &str

2. Slices

- Rustdocs: „Slices let you reference contiguous sequences in collections, instead of the whole collection“
- The most commonly seen form of Slices is the String Slice &str
- Slices are special references which are made out of two fields
  - A pointer into the collection
  - A length → How big is the slice

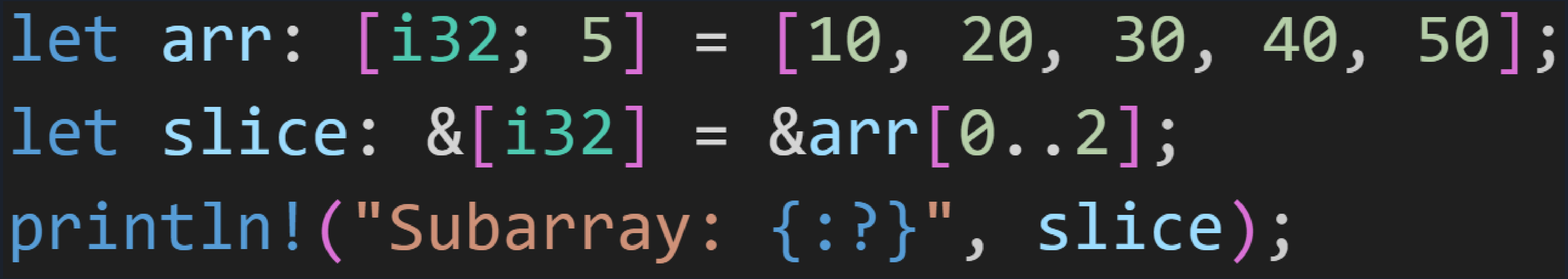
2. Slices

- Rustdocs: „Slices let you reference contiguous sequences in collections, instead of the whole collection“
- The most commonly seen form of Slices is the String Slice &str
- Slices are special references which are made out of two fields
- The type signature for Slices is [T]
  - Slices do NOT implement the Sized trait
    → can’t use [T] itself, you always need a reference

2. Slices

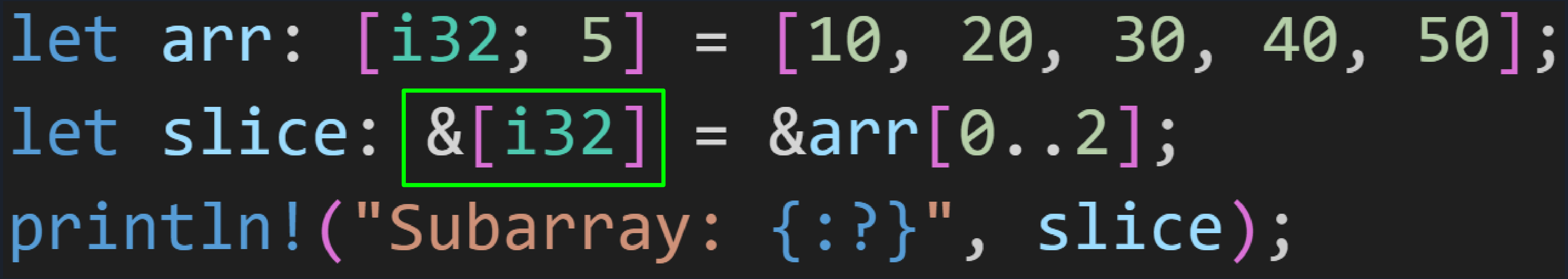
- Rustdocs: „Slices let you reference contiguous sequences in collections, instead of the whole collection“
- The most commonly seen form of Slices is the String Slice &str
- Slices are special references which are made out of two fields
- The type signature for Slices is [T]
- You can get a Slice of a collection by using ranges as indices

2. Slices



```rust
let arr: [i32; 5] = [10, 20, 30, 40, 50];
let slice: &[i32] = &arr[0..2];
println!("Subarray: {:?}", slice);
```

2. Slices

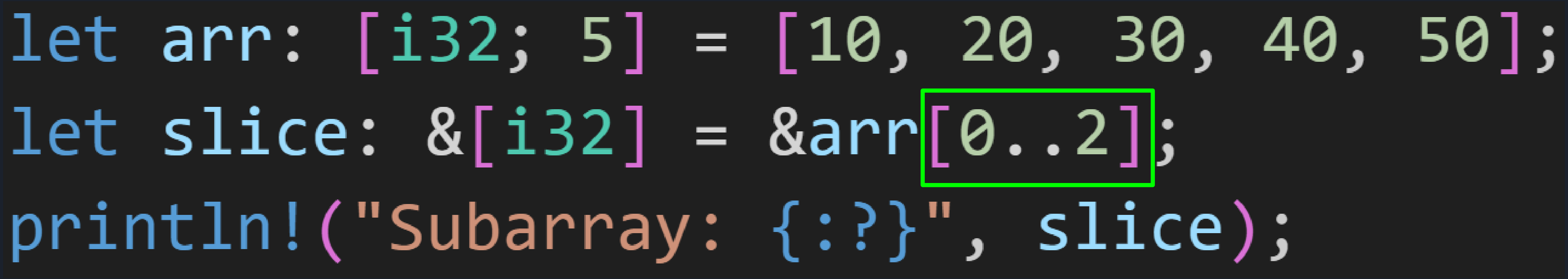


Slice of an i32-array

```
let arr: [i32; 5] = [10, 20, 30, 40, 50];
let slice: &[i32] = &arr[0..2];
println!("Subarray: {:?}", slice);
```

2. Slices

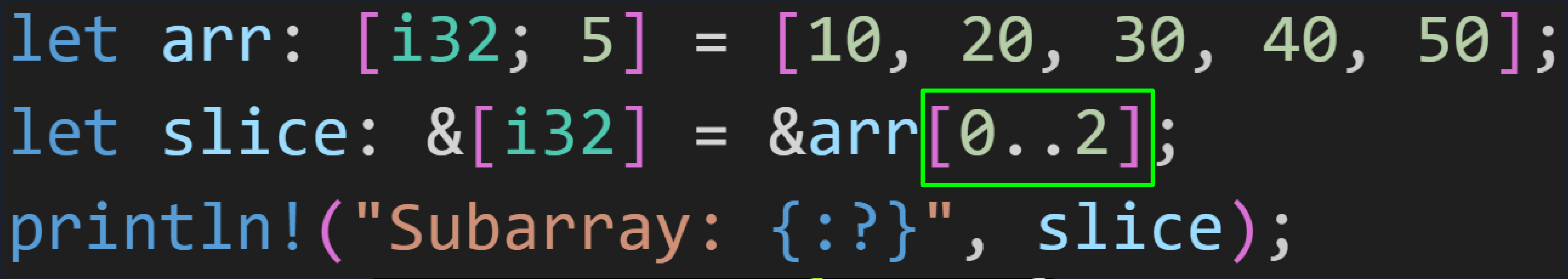
Range start is inclusive
Range end is exclusive
→ The slice refers to indices 0 and 1



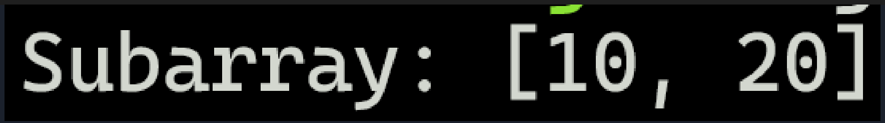
```rust
let arr: [i32; 5] = [10, 20, 30, 40, 50];
let slice: &[i32] = &arr[0..2];
println!("Subarray: {:?}", slice);
```

2. Slices

Range start is inclusive
Range end is exclusive
→ The slice refers to indices 0 and 1

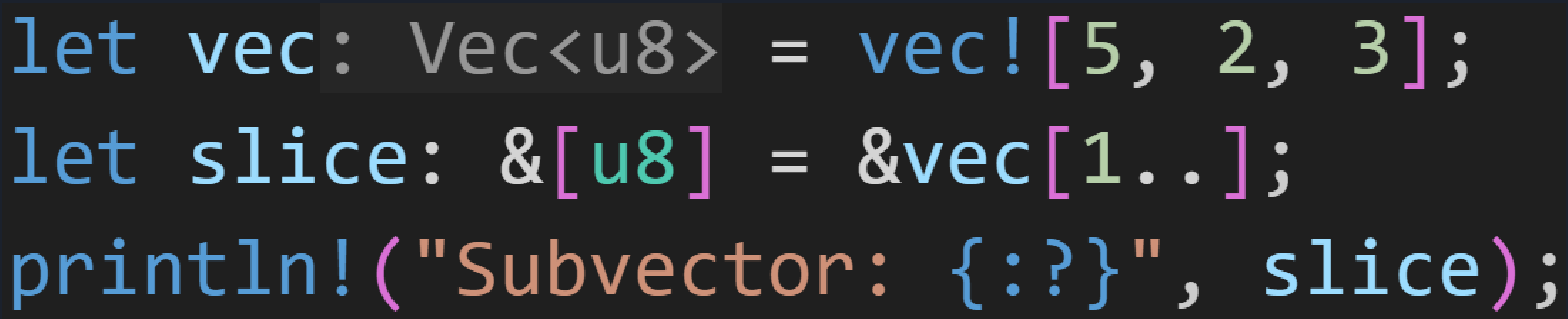


```rust
let arr: [i32; 5] = [10, 20, 30, 40, 50];
let slice: &[i32] = &arr[0..2];
println!("Subarray: {:?}", slice);
```



```
Subarray: [10, 20]
```

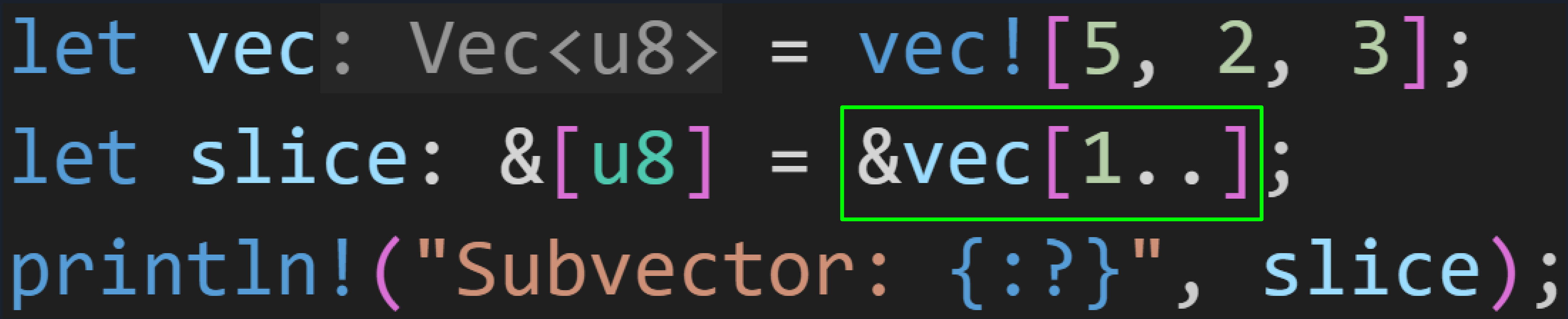
2. Slices



```rust
let vec: Vec<u8> = vec![5, 2, 3];
let slice: &[u8] = &vec[1..];
println!("Subvector: {:?}", slice);
```

2. Slices

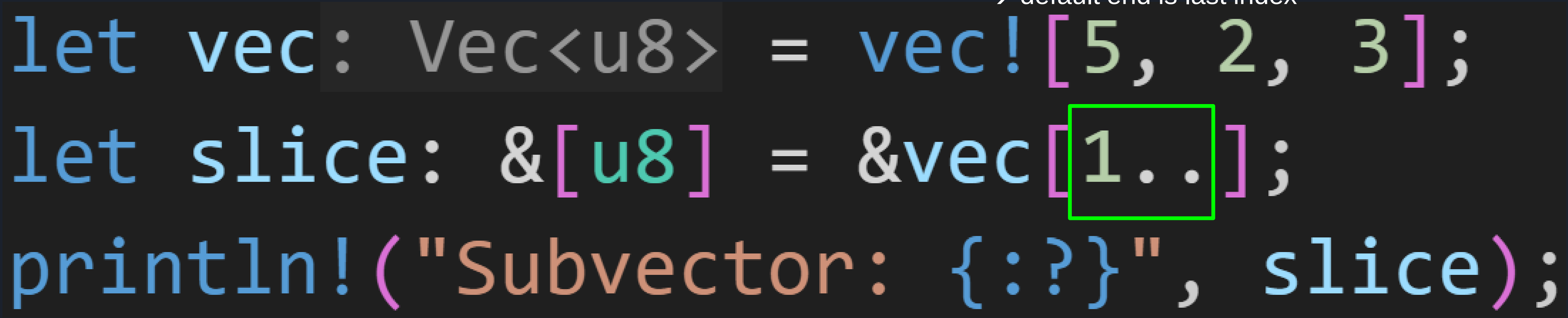
You can also slice into Vectors



```rust
let vec: Vec<u8> = vec![5, 2, 3];
let slice: &[u8] = &vec[1..];
println!("Subvector: {:?}", slice);
```

2. Slices

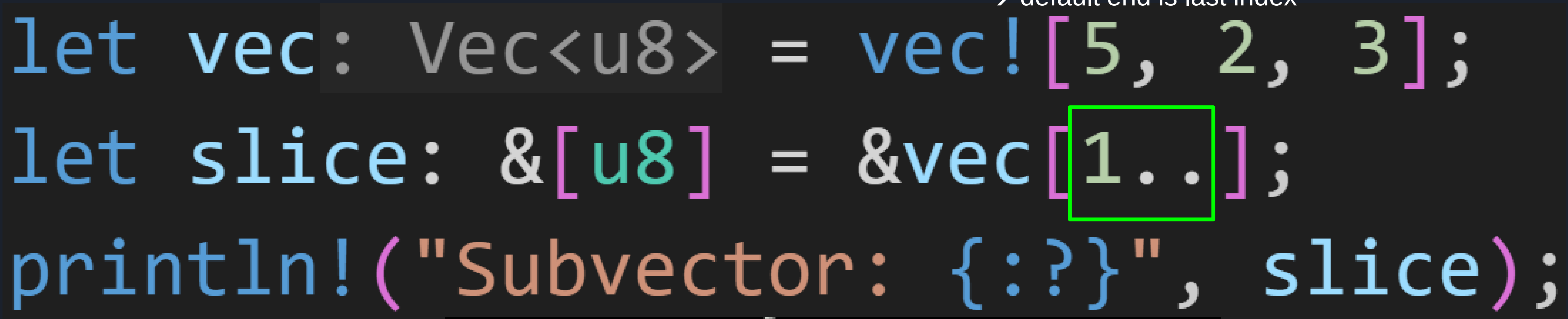
You can omit start and end
→ default start is index 0
→ default end is last index



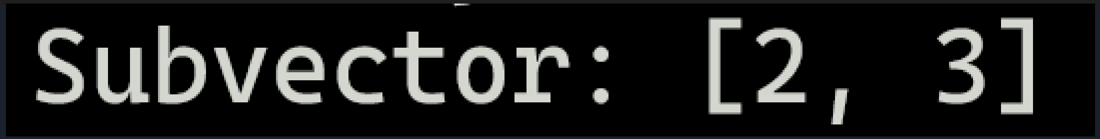
```rust
let vec: Vec<u8> = vec![5, 2, 3];
let slice: &[u8] = &vec[1..];
println!("Subvector: {:?}", slice);
```

2. Slices

You can omit start and end
→ default start is index 0
→ default end is last index



```rust
let vec: Vec<u8> = vec![5, 2, 3];
let slice: &[u8] = &vec[1..];
println!("Subvector: {:?}", slice);
```

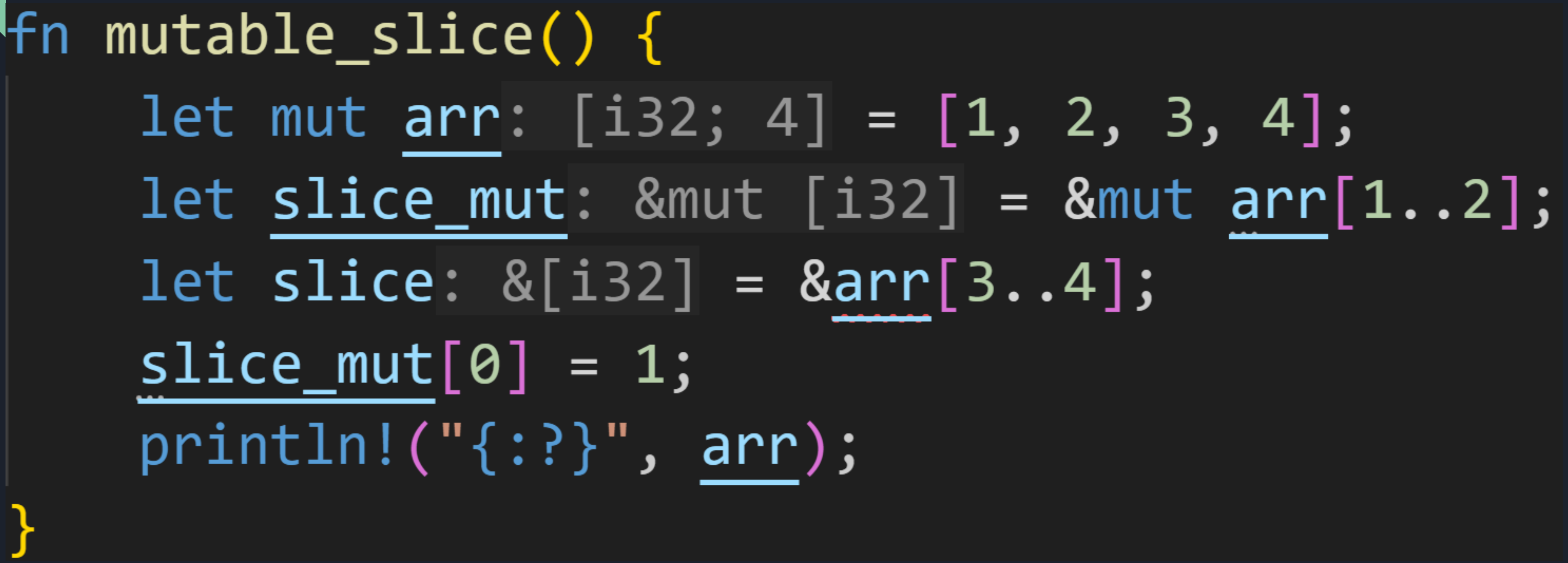


```
Subvector: [2, 3]
```

2. Slices

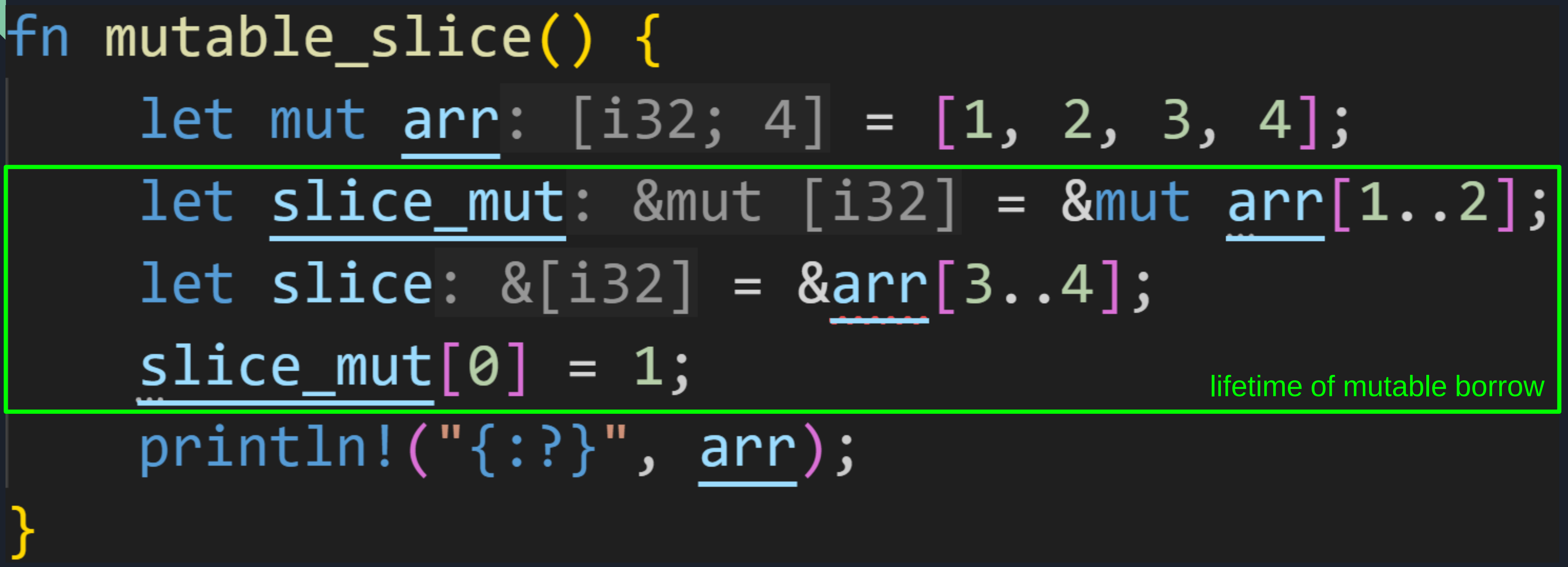
- Rustdocs: „Slices let you reference contiguous sequences in collections, instead of the whole collection“
- The most commonly seen form of Slices is the String Slice &str
- Slices are special references which are made out of two fields
- The type signature for Slices is [T]
- You can get a Slice of a collection by using ranges as indices
- As Slices are references, normal Ownership and Borrow Checker rules apply
  - While you borrow a Slice of a collection, you can‘t modify it
  - Slices don‘t own any elements
  - No Moves or Copies happen

2. Slices



```
fn mutable_slice() {
    let mut arr: [i32; 4] = [1, 2, 3, 4];
    let slice_mut: &mut [i32] = &mut arr[1..2];
    let slice: &[i32] = &arr[3..4];
    slice_mut[0] = 1;
    println!("{:?}", arr);
}
```

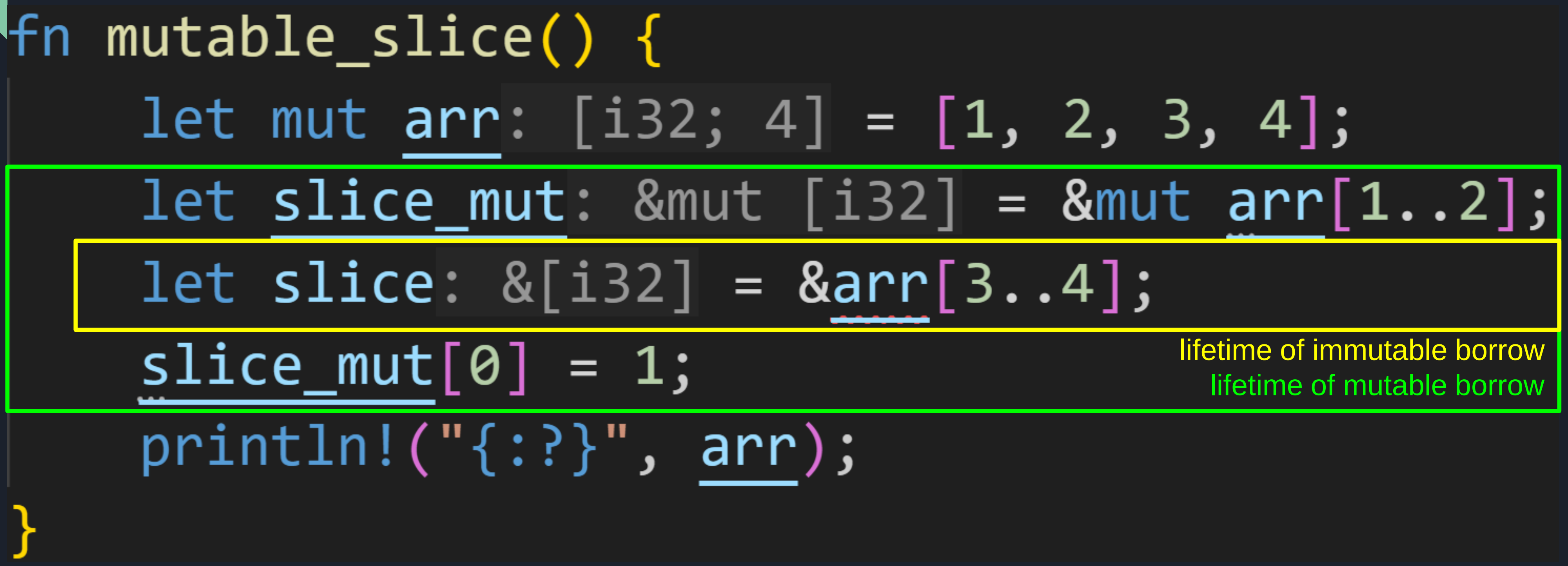
2. Slices



```rust
fn mutable_slice() {
    let mut arr: [i32; 4] = [1, 2, 3, 4];
    let slice_mut: &mut [i32] = &mut arr[1..2];
    let slice: &[i32] = &arr[3..4];
    slice_mut[0] = 1;
    println!("{:?}", arr);
}
```

lifetime of mutable borrow

2. Slices

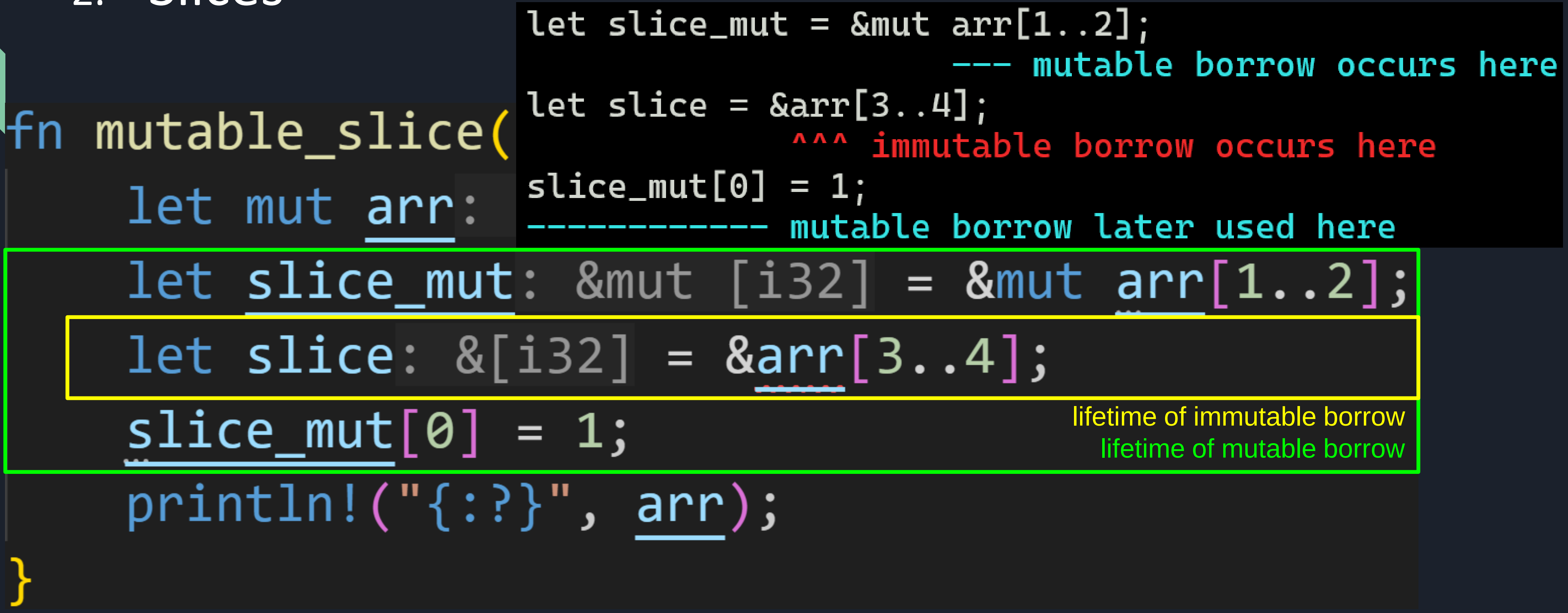


```rust
fn mutable_slice() {
    let mut arr: [i32; 4] = [1, 2, 3, 4];
    let slice_mut: &mut [i32] = &mut arr[1..2];
    let slice: &[i32] = &arr[3..4];
    slice_mut[0] = 1;
    println!("{:?}", arr);
}
```

lifetime of immutable borrow

lifetime of mutable borrow

2. Slices



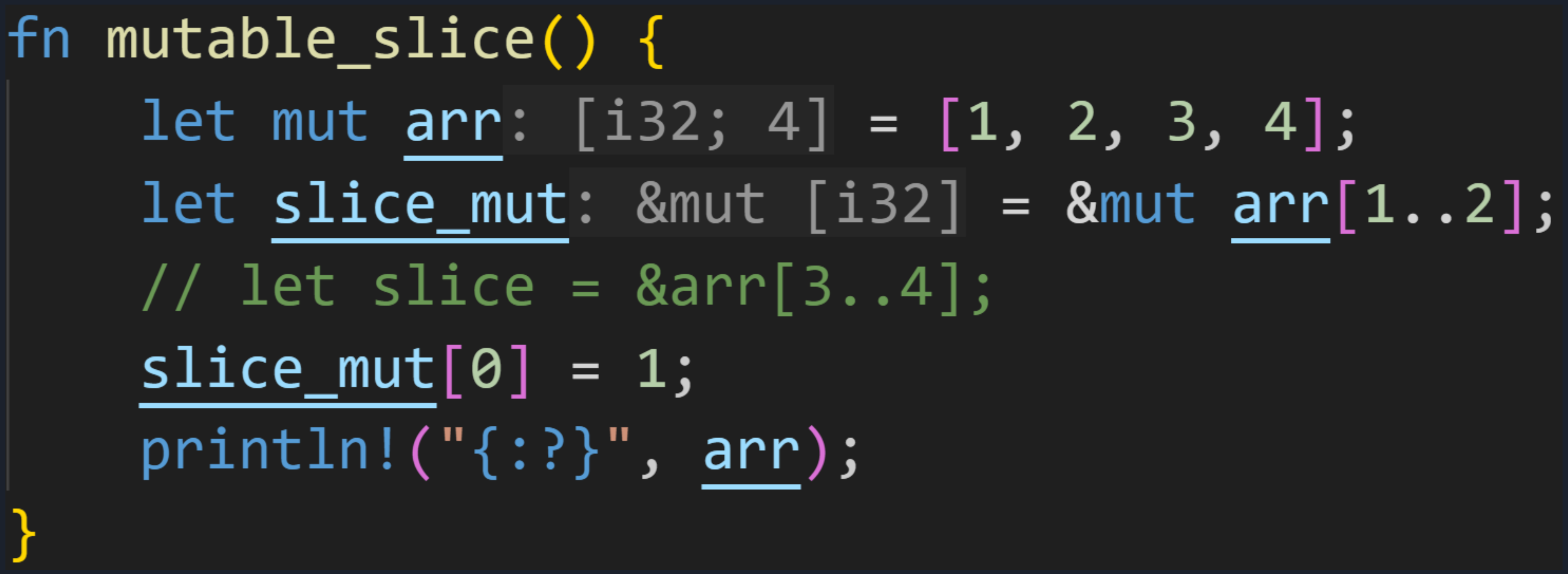
```
let slice_mut = &mut arr[1..2];
                ---- mutable borrow occurs here
let slice = &arr[3..4];
             ^^^ immutable borrow occurs here
slice_mut[0] = 1;
------------ mutable borrow later used here
```

```
fn mutable_slice(
    let mut arr:
    let slice_mut: &mut [i32] = &mut arr[1..2];
    let slice: &[i32] = &arr[3..4];
    slice_mut[0] = 1;
    println!("{:?}", arr);
}
```

lifetime of immutable borrow

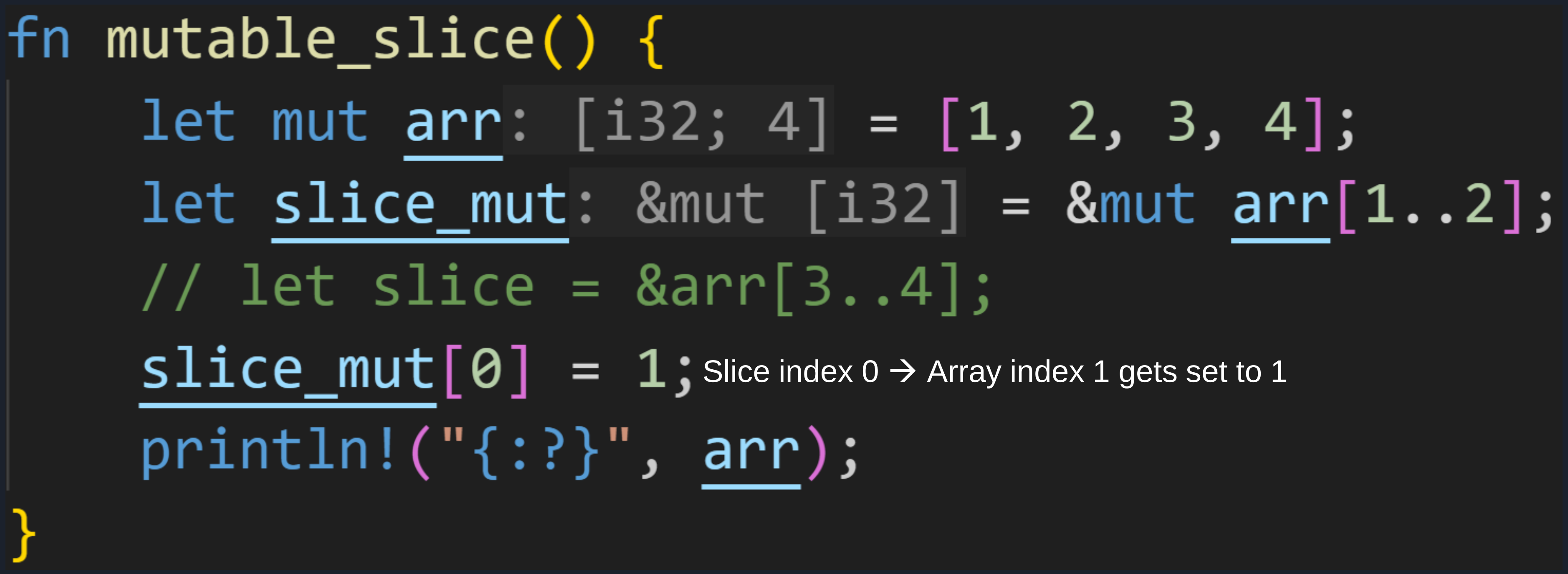
lifetime of mutable borrow

2. Slices



```rust
fn mutable_slice() {
    let mut arr: [i32; 4] = [1, 2, 3, 4];
    let slice_mut: &mut [i32] = &mut arr[1..2];
    // let slice = &arr[3..4];
    slice_mut[0] = 1;
    println!("{:?}", arr);
}
```

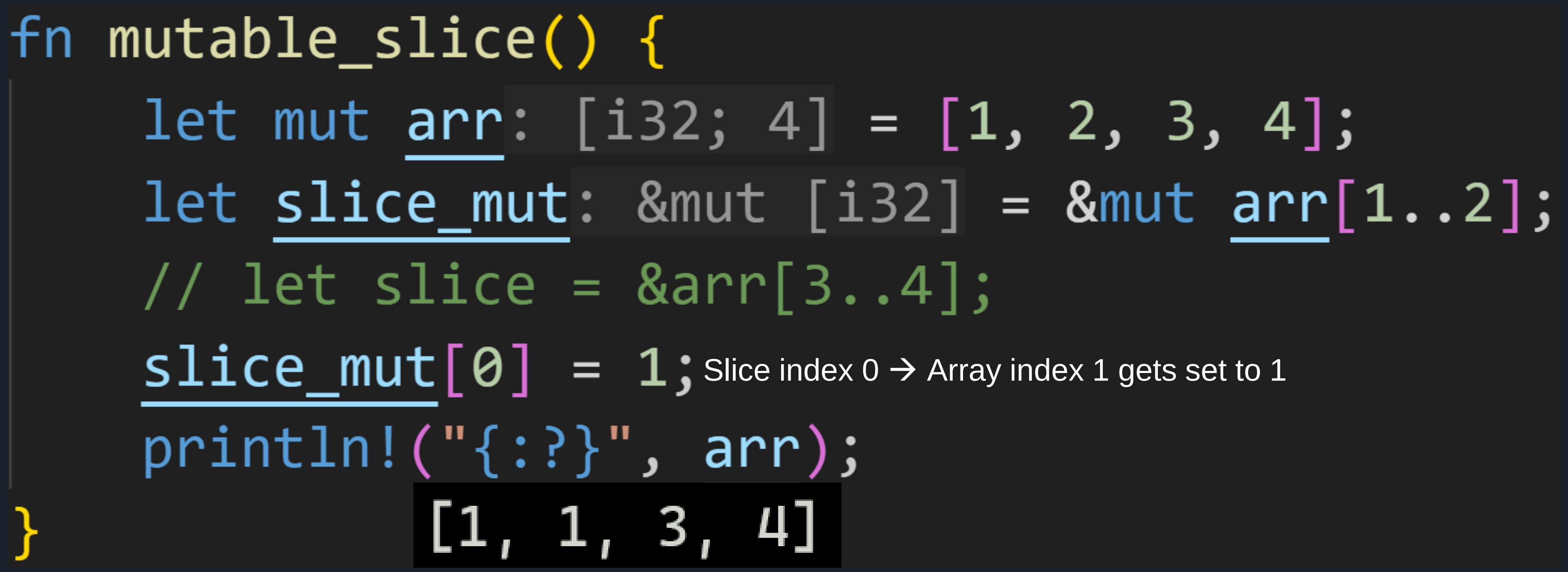
2. Slices



```
fn mutable_slice() {
    let mut arr: [i32; 4] = [1, 2, 3, 4];
    let slice_mut: &mut [i32] = &mut arr[1..2];
    // let slice = &arr[3..4];
    slice_mut[0] = 1;
    println!("{:?}", arr);
}
```

Slice index 0 → Array index 1 gets set to 1

2. Slices

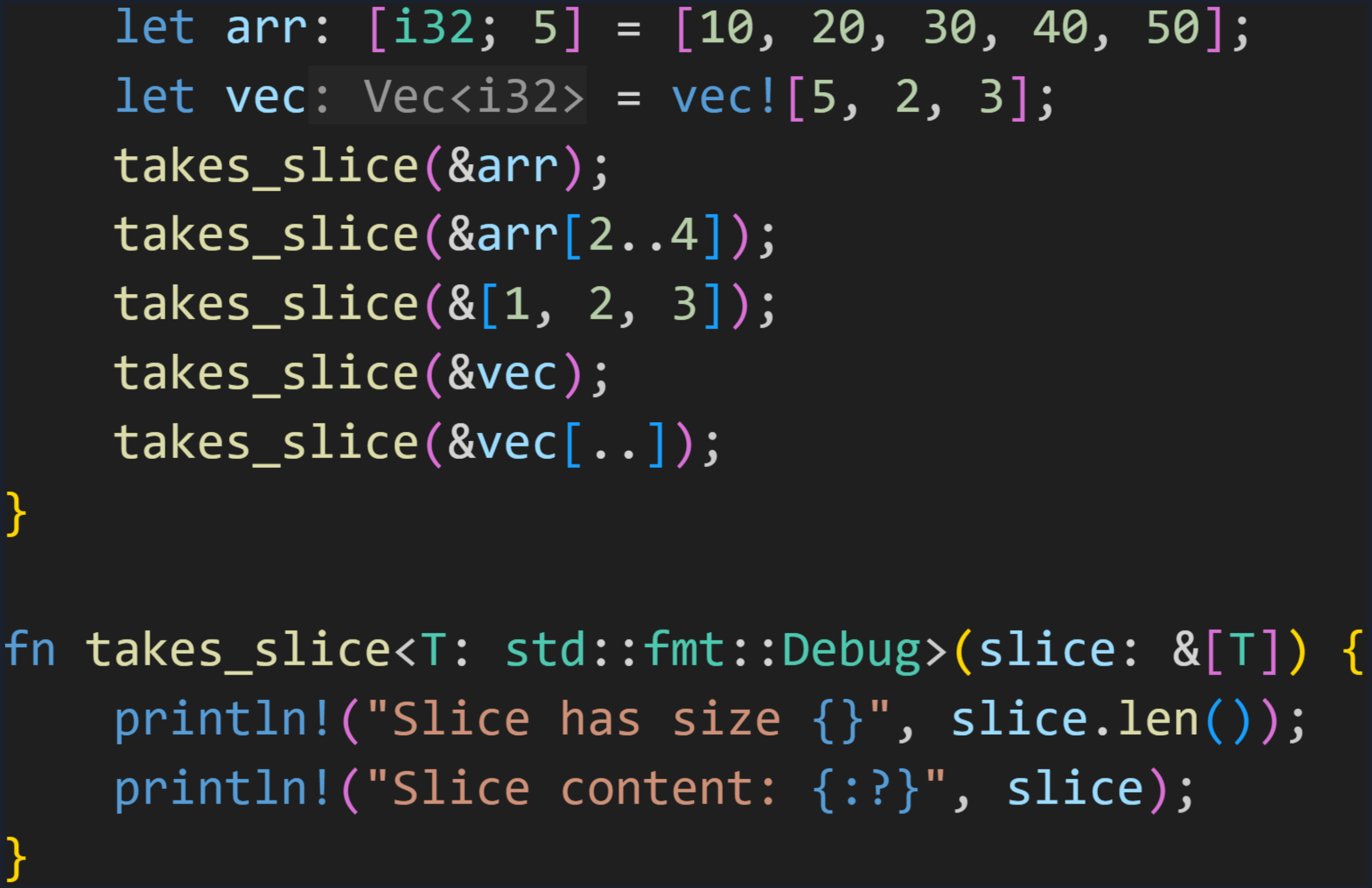


```
fn mutable_slice() {
    let mut arr: [i32; 4] = [1, 2, 3, 4];
    let slice_mut: &mut [i32] = &mut arr[1..2];
    // let slice = &arr[3..4];
    slice_mut[0] = 1;
    println!("{:?}", arr);
}
```

Slice index 0 → Array index 1 gets set to 1

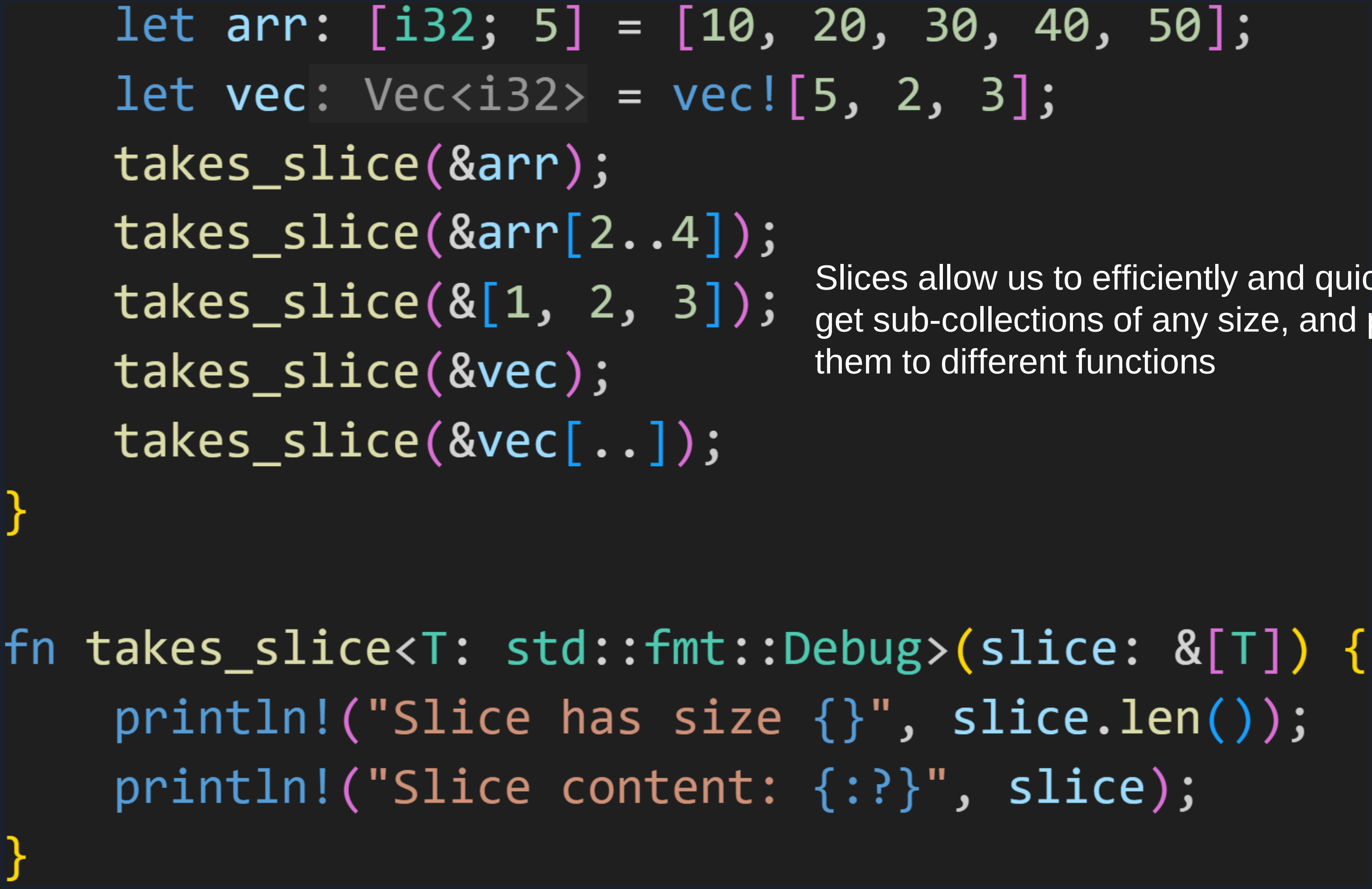
```
[1, 1, 3, 4]
```

2. Slices



```rust
    let arr: [i32; 5] = [10, 20, 30, 40, 50];
    let vec: Vec<i32> = vec![5, 2, 3];
    takes_slice(&arr);
    takes_slice(&arr[2..4]);
    takes_slice(&[1, 2, 3]);
    takes_slice(&vec);
    takes_slice(&vec[..]);
}

fn takes_slice<T: std::fmt::Debug>(slice: &[T]) {
    println!("Slice has size {}", slice.len());
    println!("Slice content: {:?}", slice);
}
```

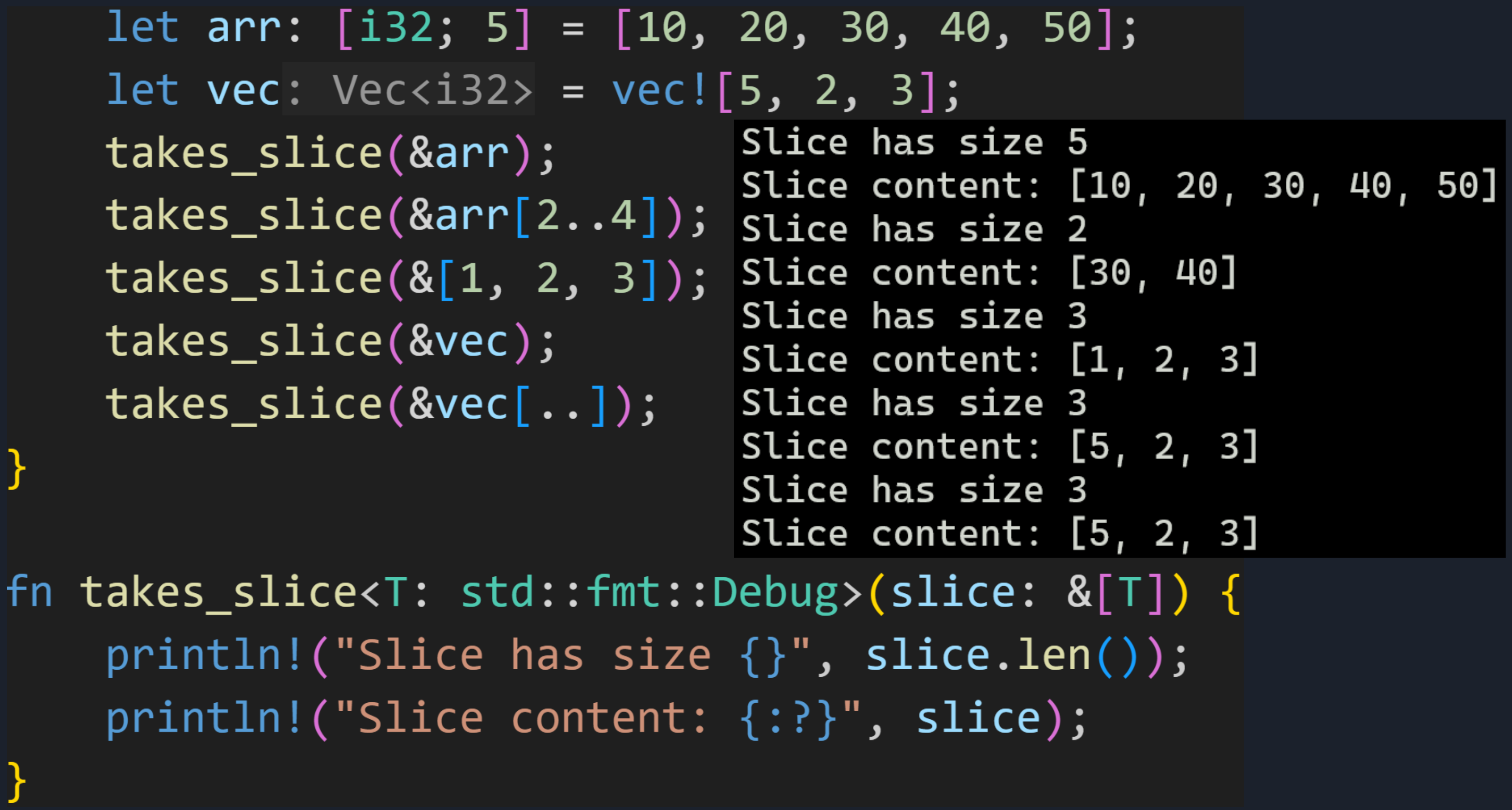
2. Slices



```
    let arr: [i32; 5] = [10, 20, 30, 40, 50];
    let vec: Vec<i32> = vec![5, 2, 3];
    takes_slice(&arr);
    takes_slice(&arr[2..4]);
    takes_slice(&[1, 2, 3]);
    takes_slice(&vec);
    takes_slice(&vec[..]);
}

fn takes_slice<T: std::fmt::Debug>(slice: &[T]) {
    println!("Slice has size {}", slice.len());
    println!("Slice content: {:?}", slice);
}
```

Slices allow us to efficiently and quickly get sub-collections of any size, and pass them to different functions

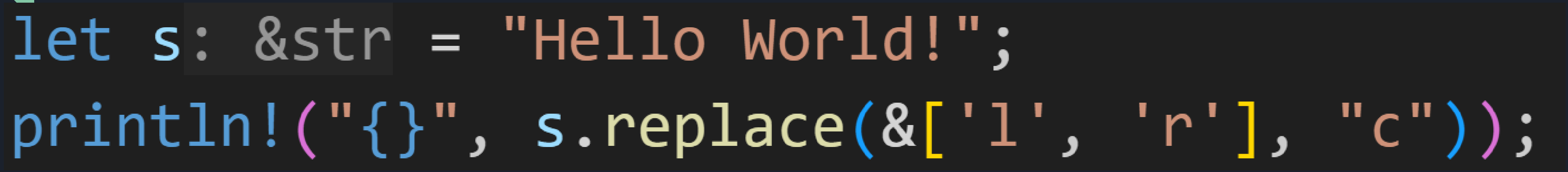
2. Slices



```rust
    let arr: [i32; 5] = [10, 20, 30, 40, 50];
    let vec: Vec<i32> = vec![5, 2, 3];
    takes_slice(&arr);
    takes_slice(&arr[2..4]);
    takes_slice(&[1, 2, 3]);
    takes_slice(&vec);
    takes_slice(&vec[..]);
}

fn takes_slice<T: std::fmt::Debug>(slice: &[T]) {
    println!("Slice has size {}", slice.len());
    println!("Slice content: {:?}", slice);
}
```

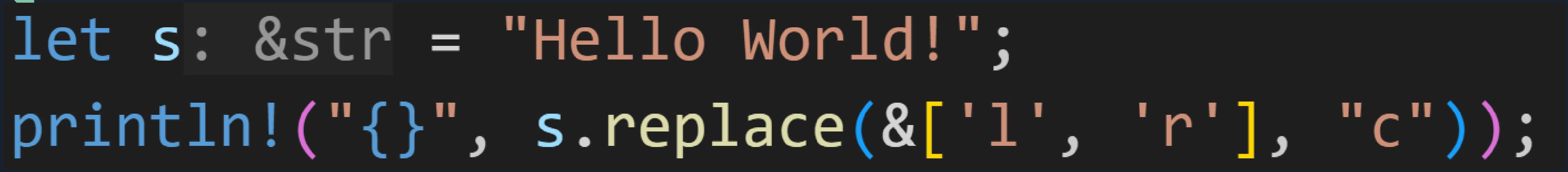
```
Slice has size 5
Slice content: [10, 20, 30, 40, 50]
Slice has size 2
Slice content: [30, 40]
Slice has size 3
Slice content: [1, 2, 3]
Slice has size 3
Slice content: [5, 2, 3]
Slice has size 3
Slice content: [5, 2, 3]
```

2. Slices



```rust
let s: &str = "Hello World!";
println!("{}", s.replace(&['l', 'r'], "c"));
```

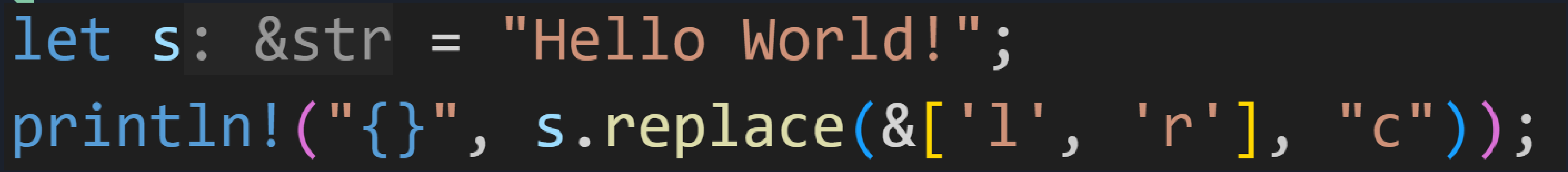
2. Slices



```rust
let s: &str = "Hello World!";
println!("{}", s.replace(&['l', 'r'], "c"));
```

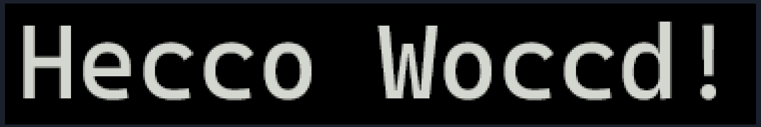
Many functions in the standard library accept slices
→ Here: Replace every „l“ and „r“ in the String s with a „c“

2. Slices



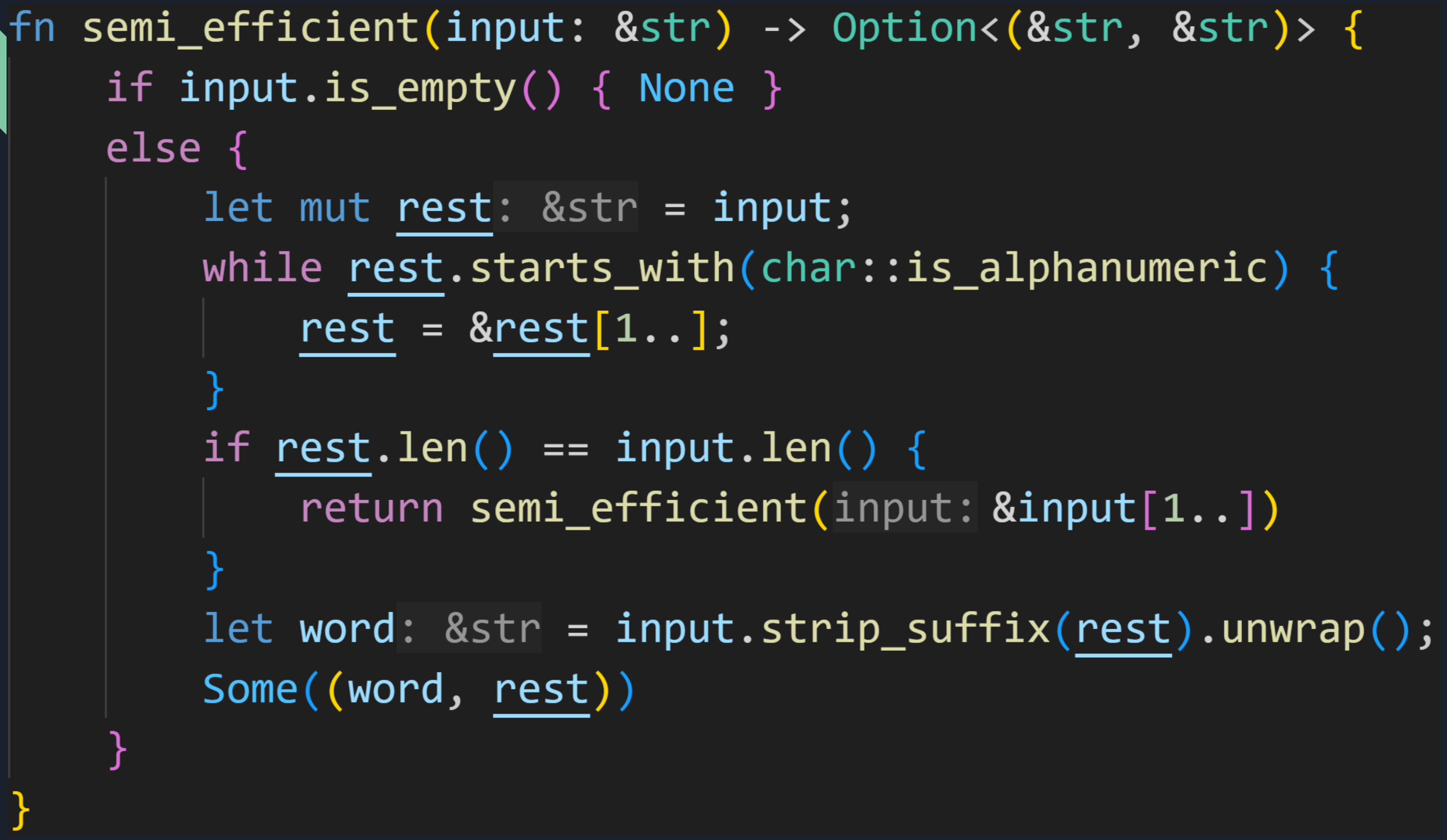
```rust
let s: &str = "Hello World!";
println!("{}", s.replace(&['l', 'r'], "c"));
```

Many functions in the standard library accept slices
→ Here: Replace every „l“ and „r“ in the String s with a „c“



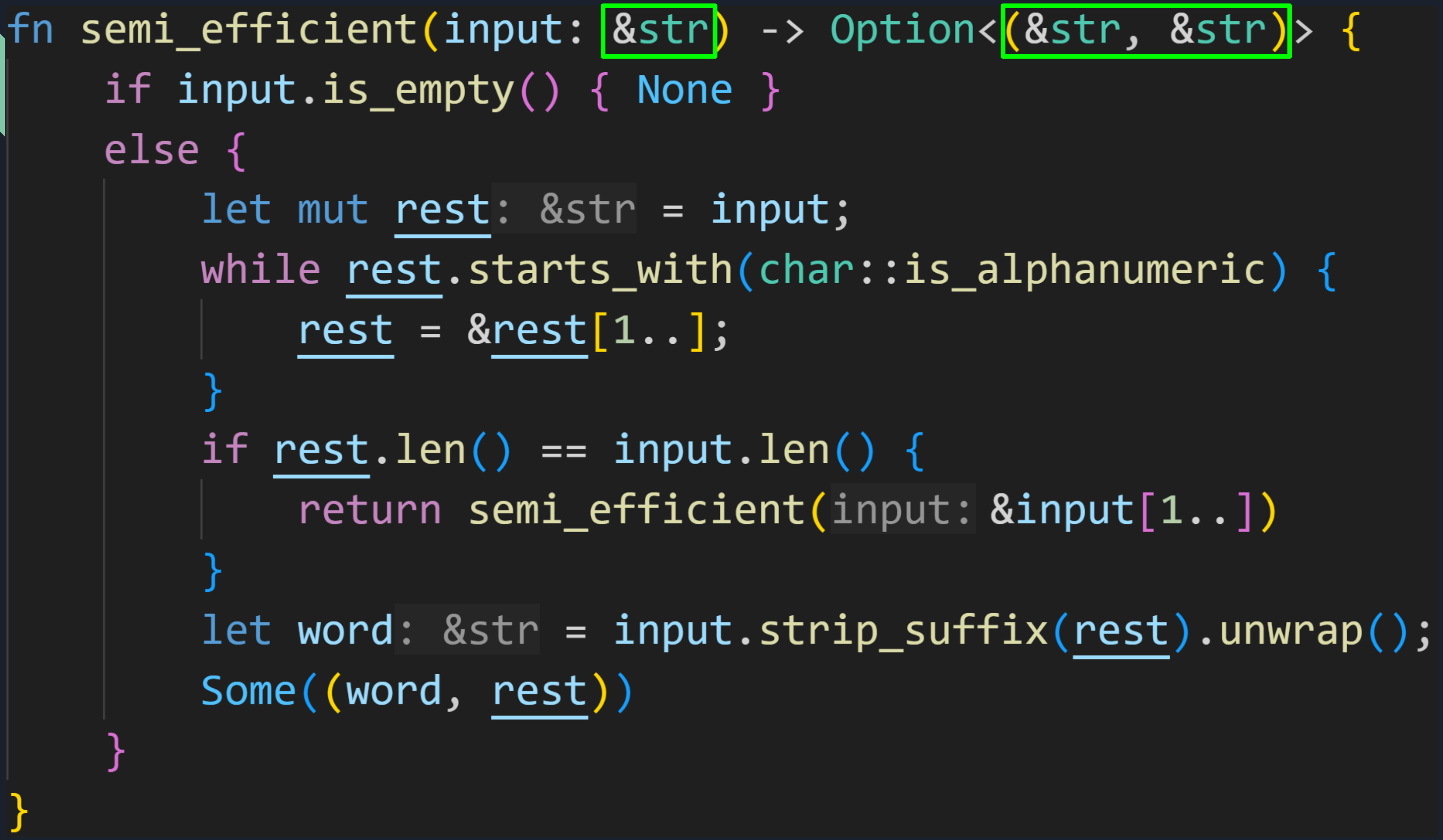
```
Hecco Woccd!
```

2. Slices



```
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

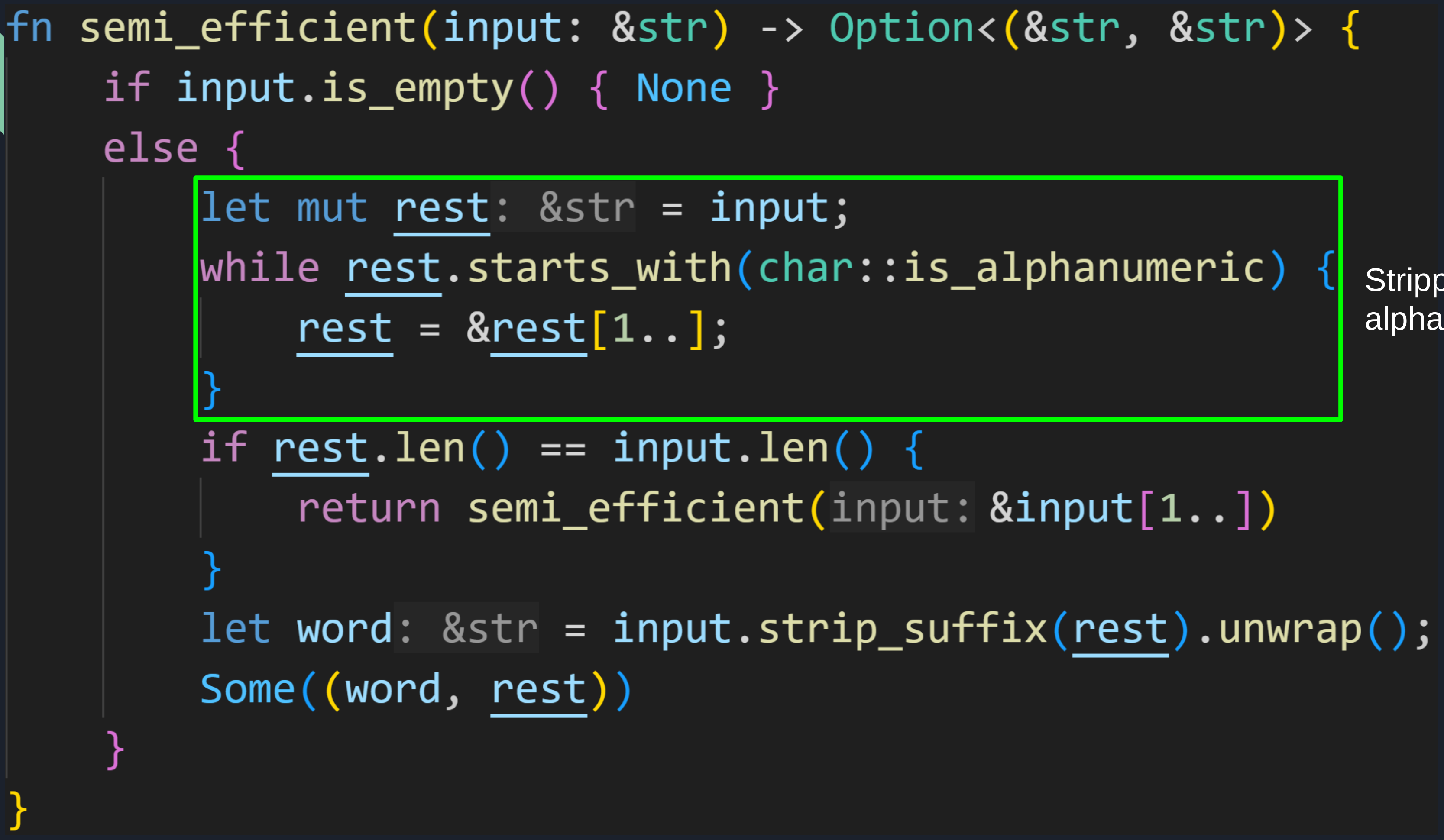
2. Slices



Slices into the original string

```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

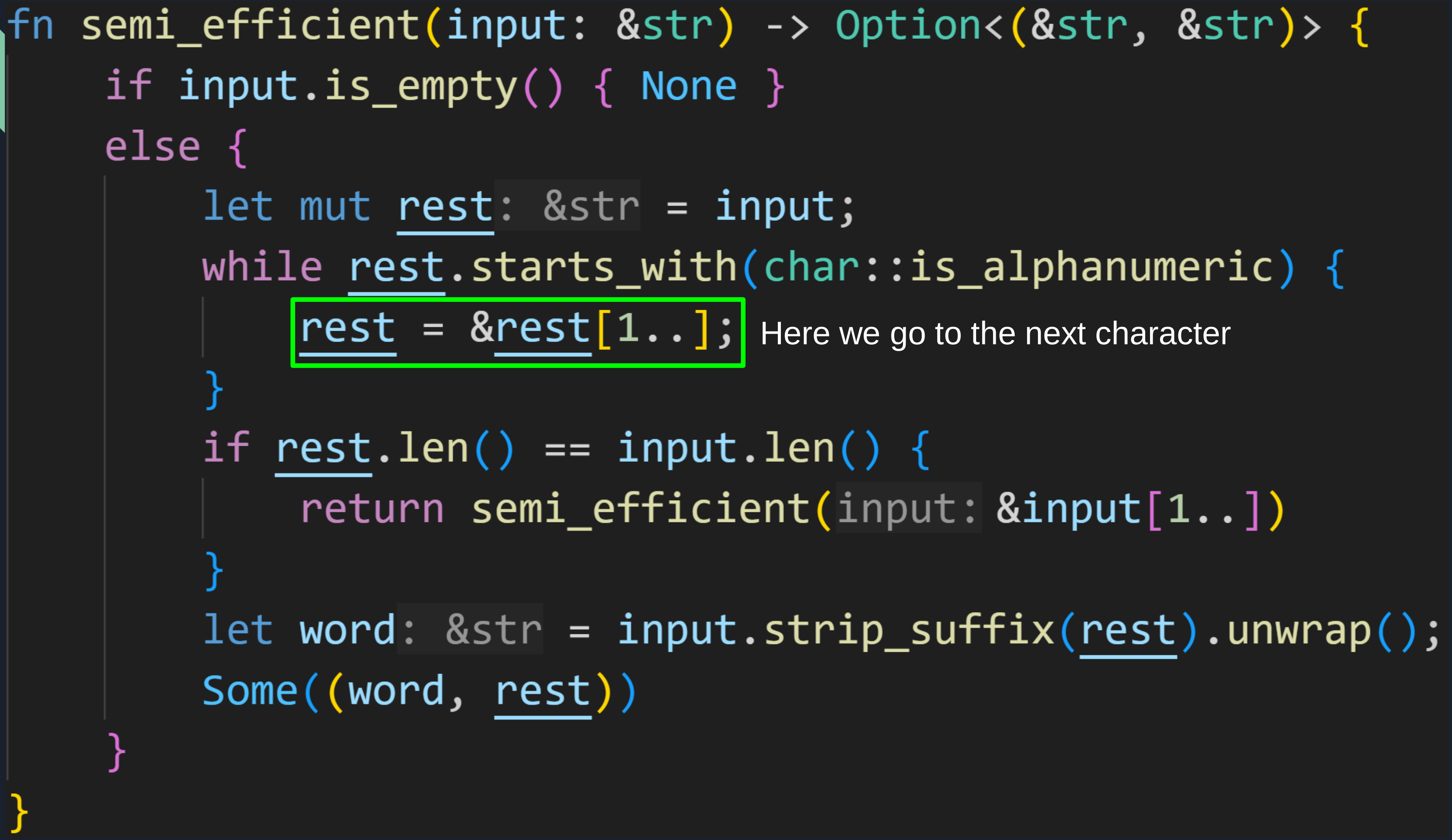
2. Slices



```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

Stripping off all alphanumeric characters

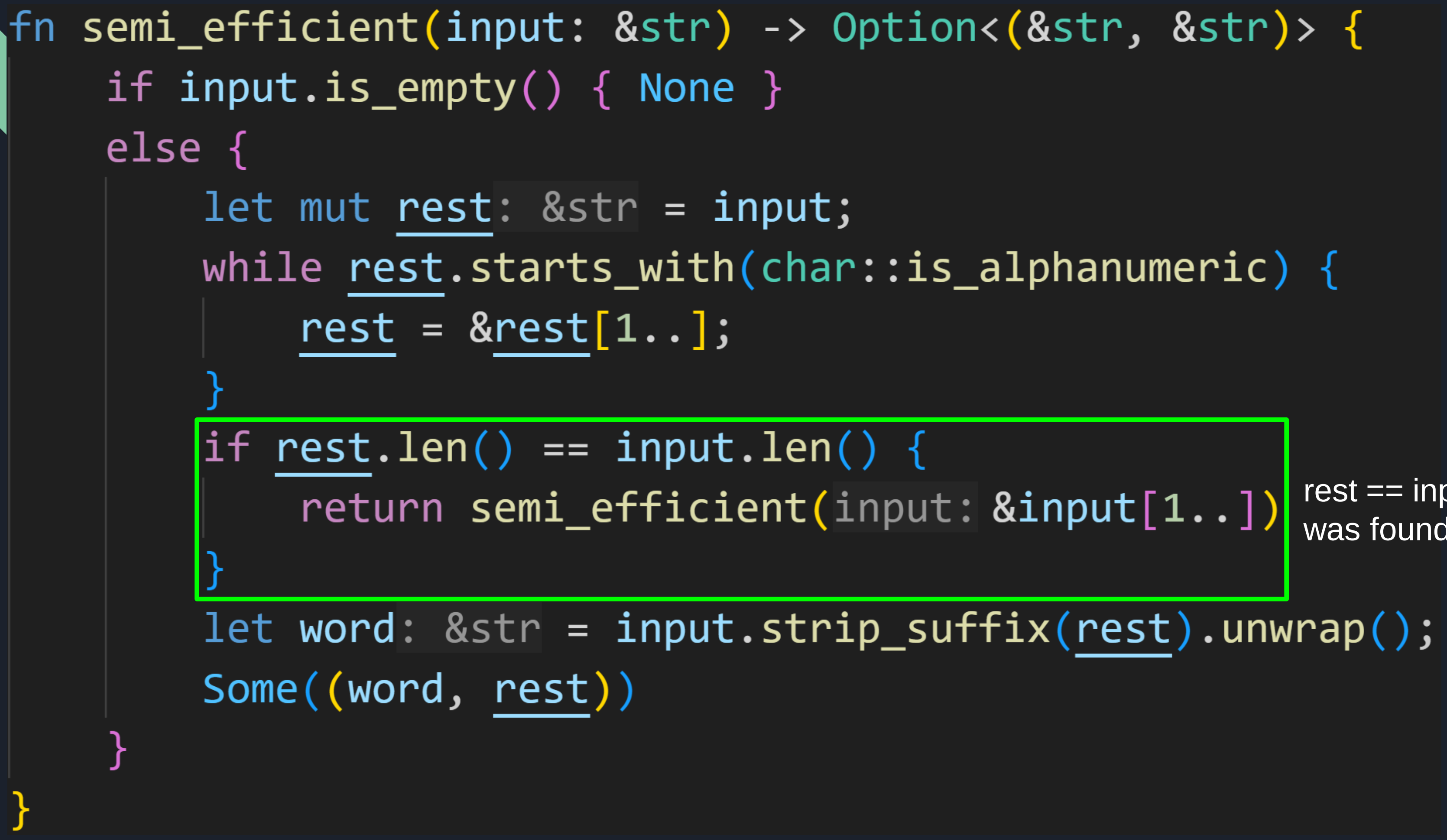
2. Slices



```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

Here we go to the next character

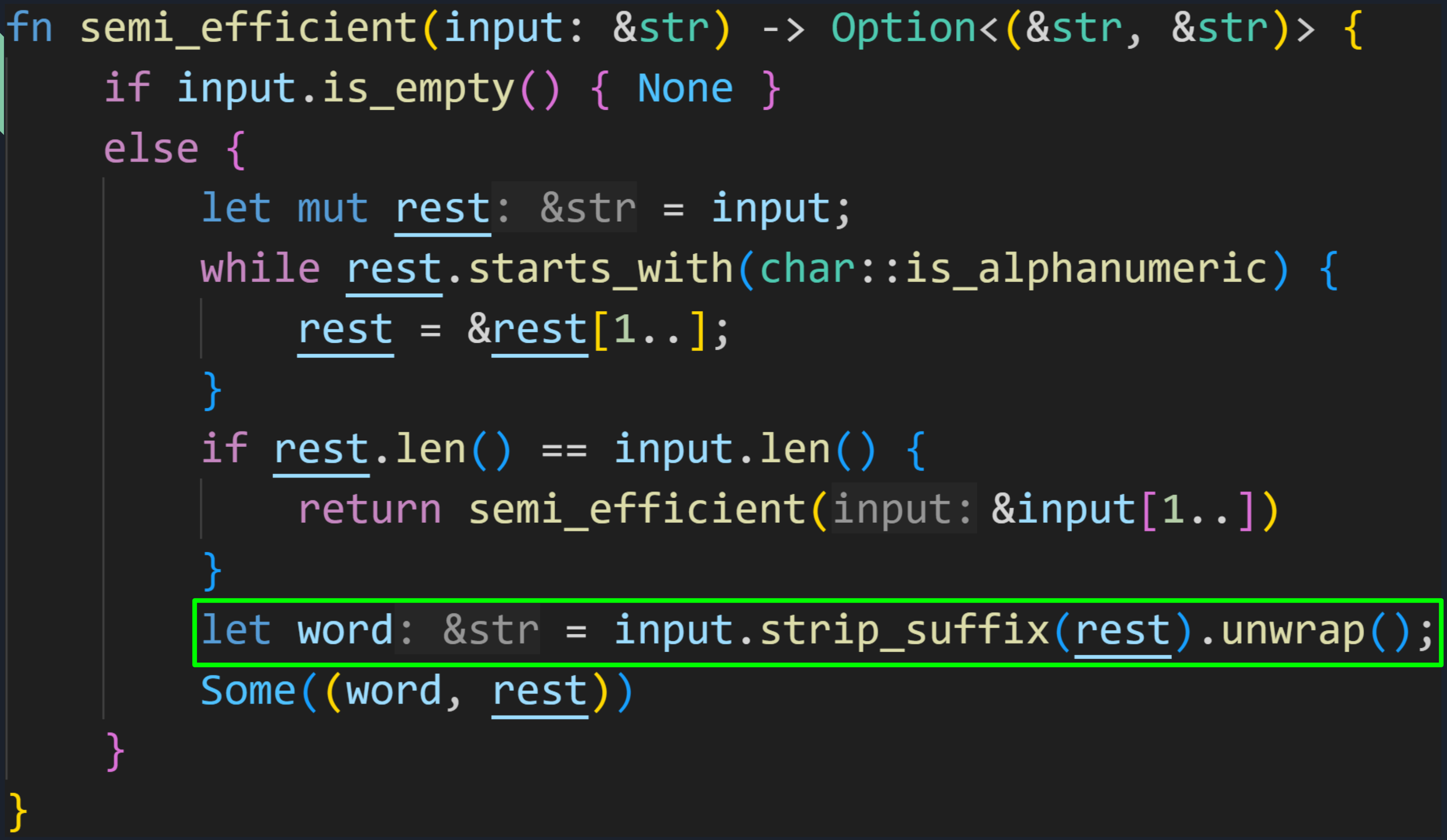
2. Slices



```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

rest == input iff no alphanumeric character was found at the start → special character

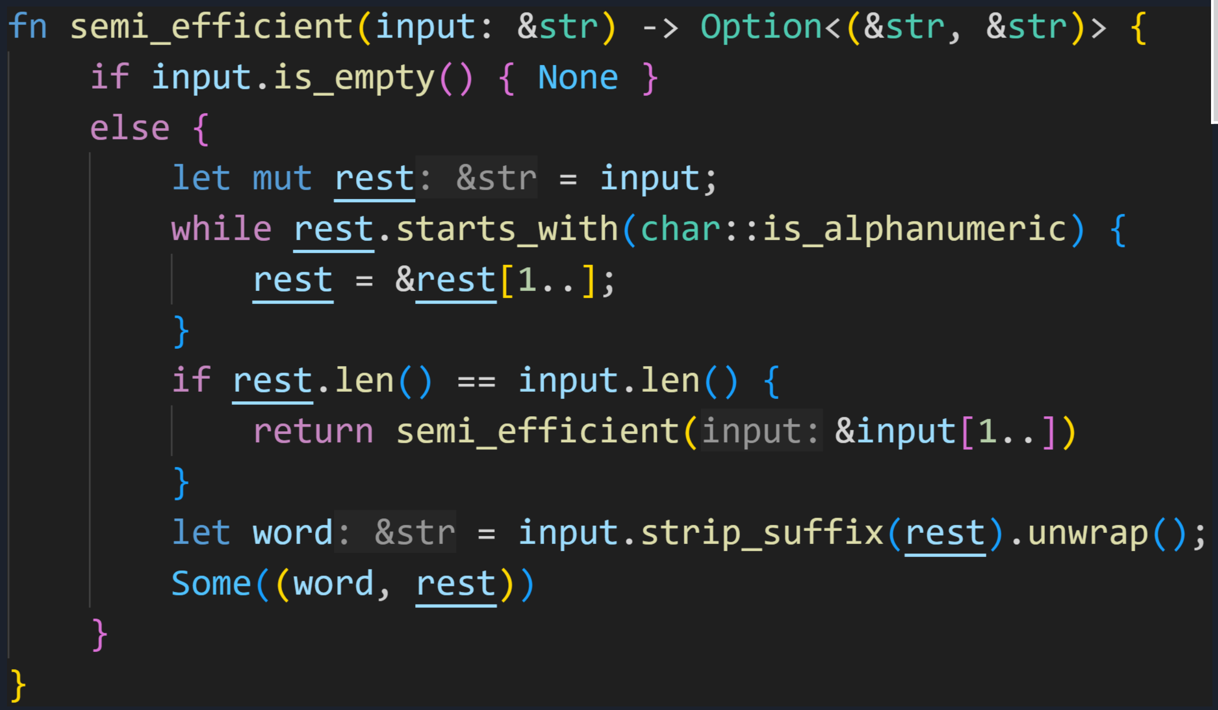
2. Slices



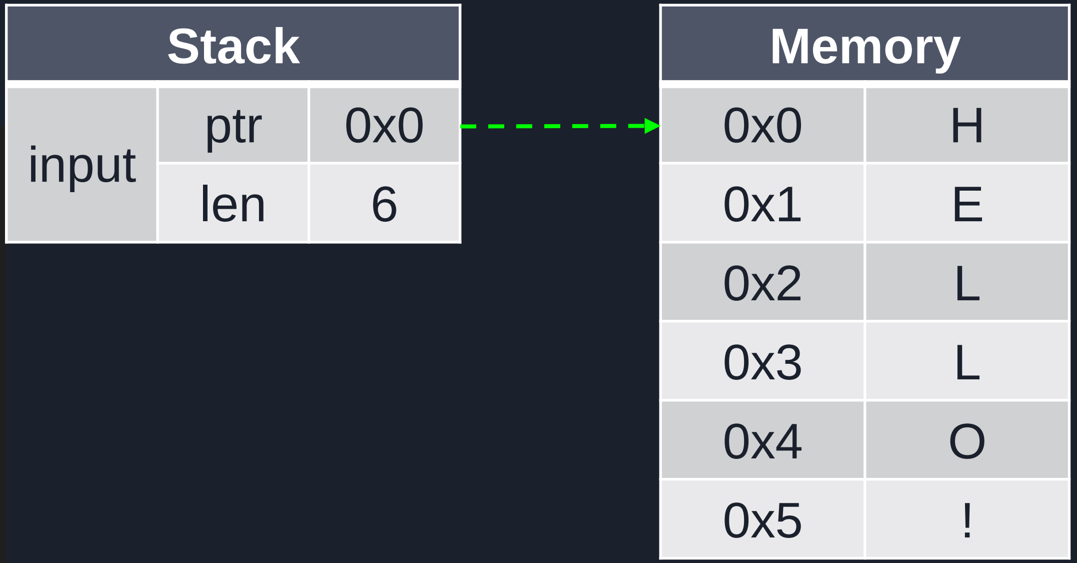
```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

Actual word is input - rest

2. Slices



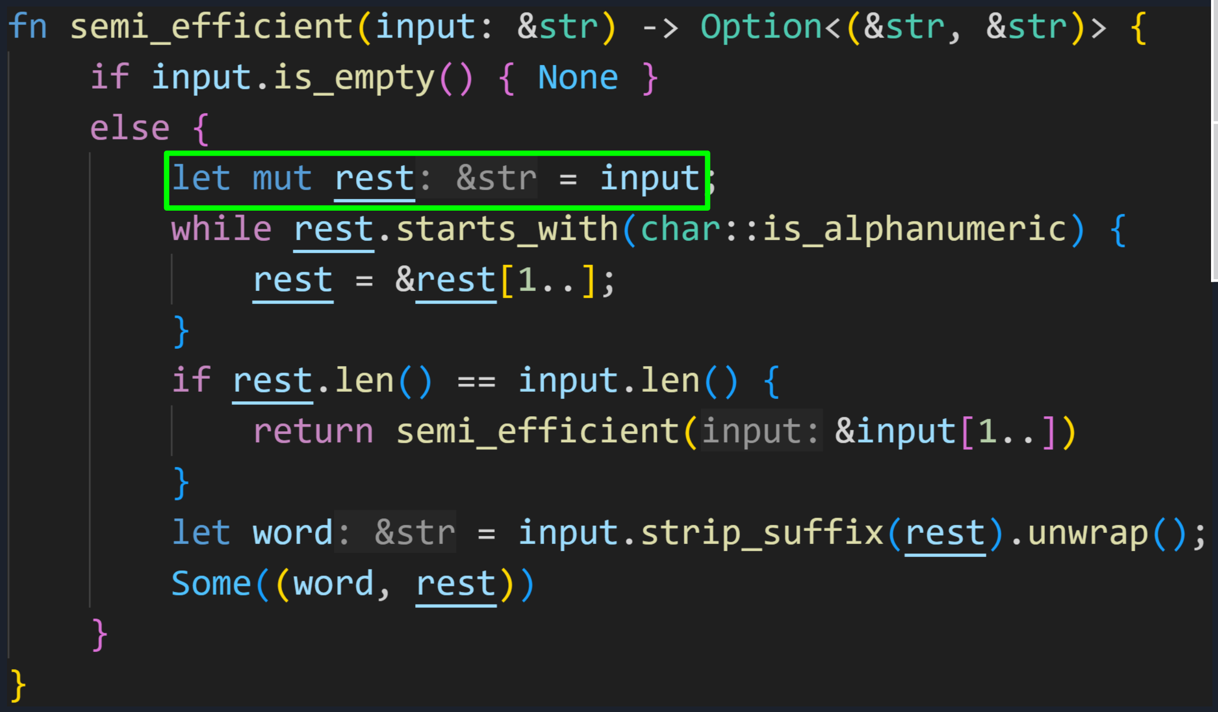
```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```



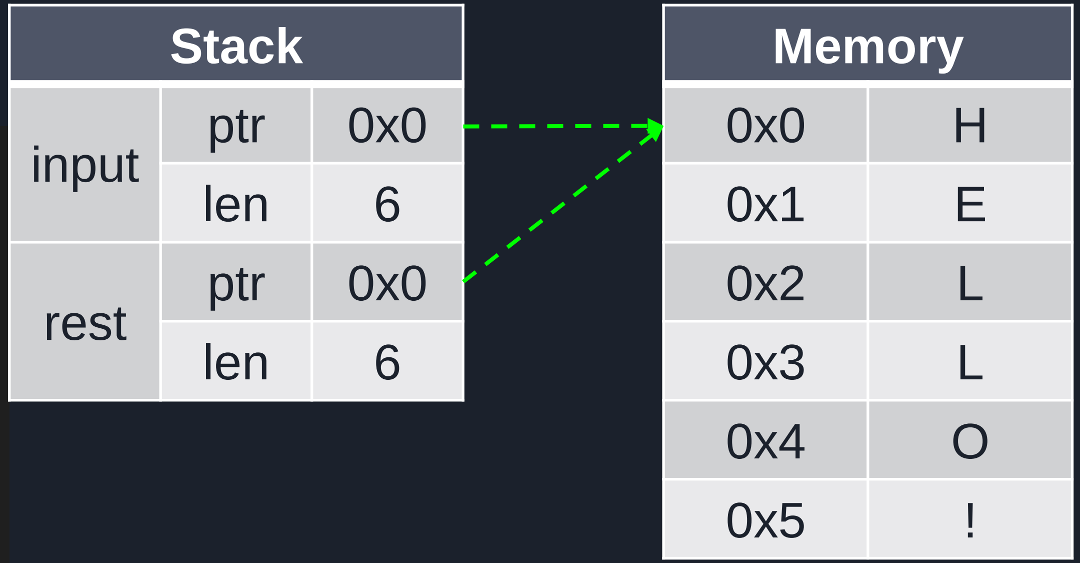
Stack		
input	ptr	0x0
	len	6

Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

2. Slices



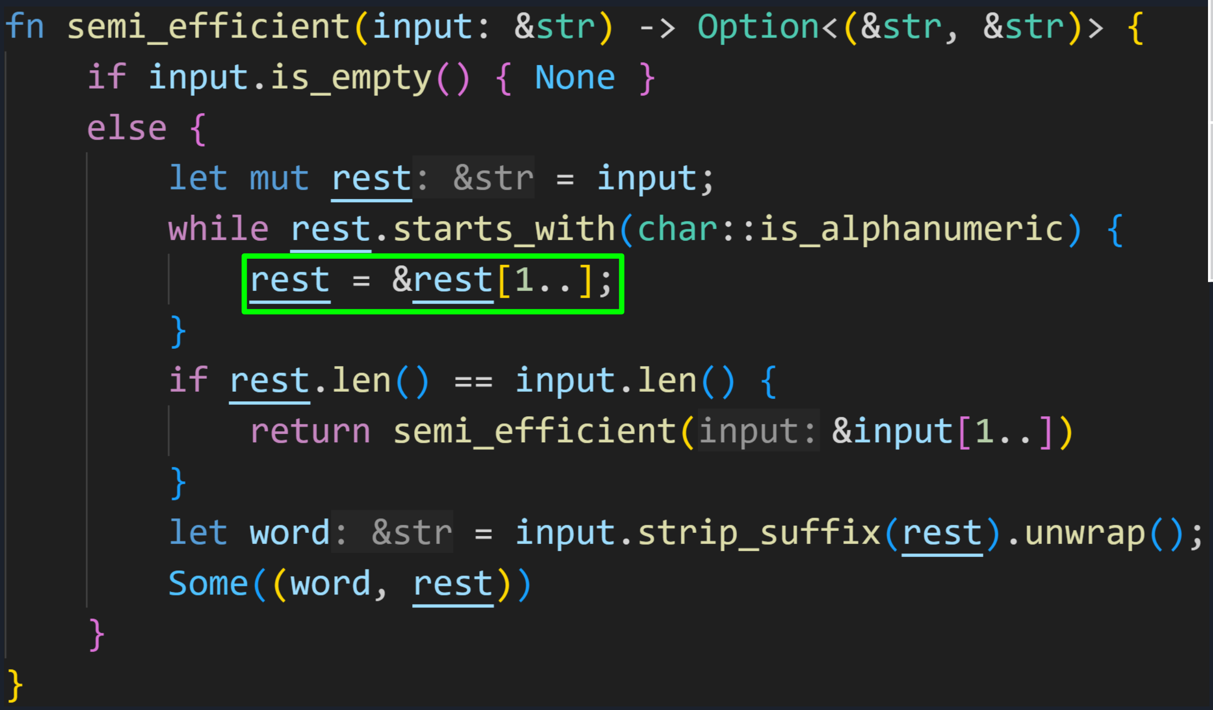
```
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```



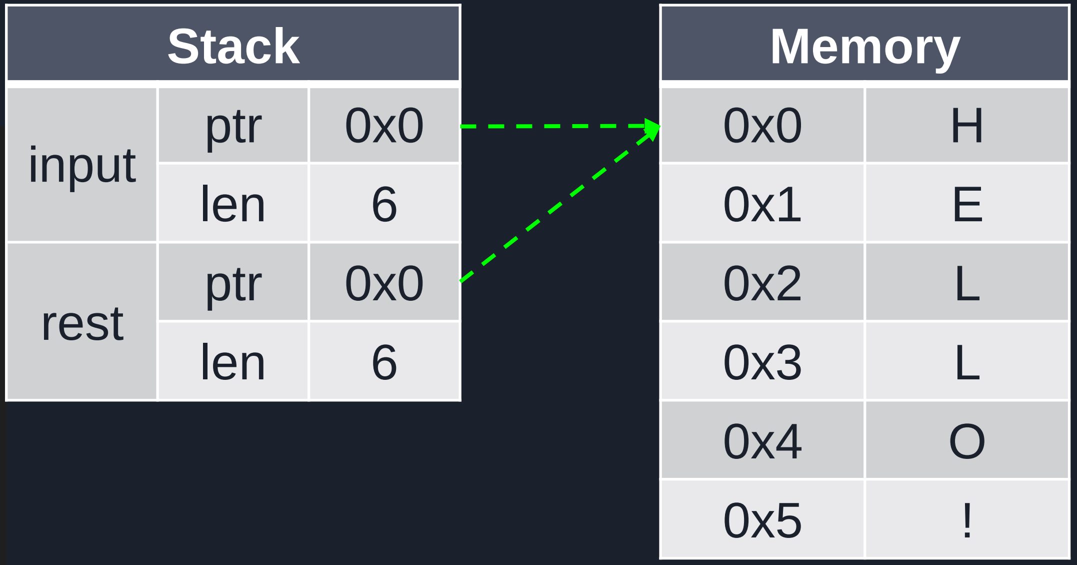
Stack		
input	ptr	0x0
	len	6
rest	ptr	0x0
	len	6

Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

2. Slices



```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

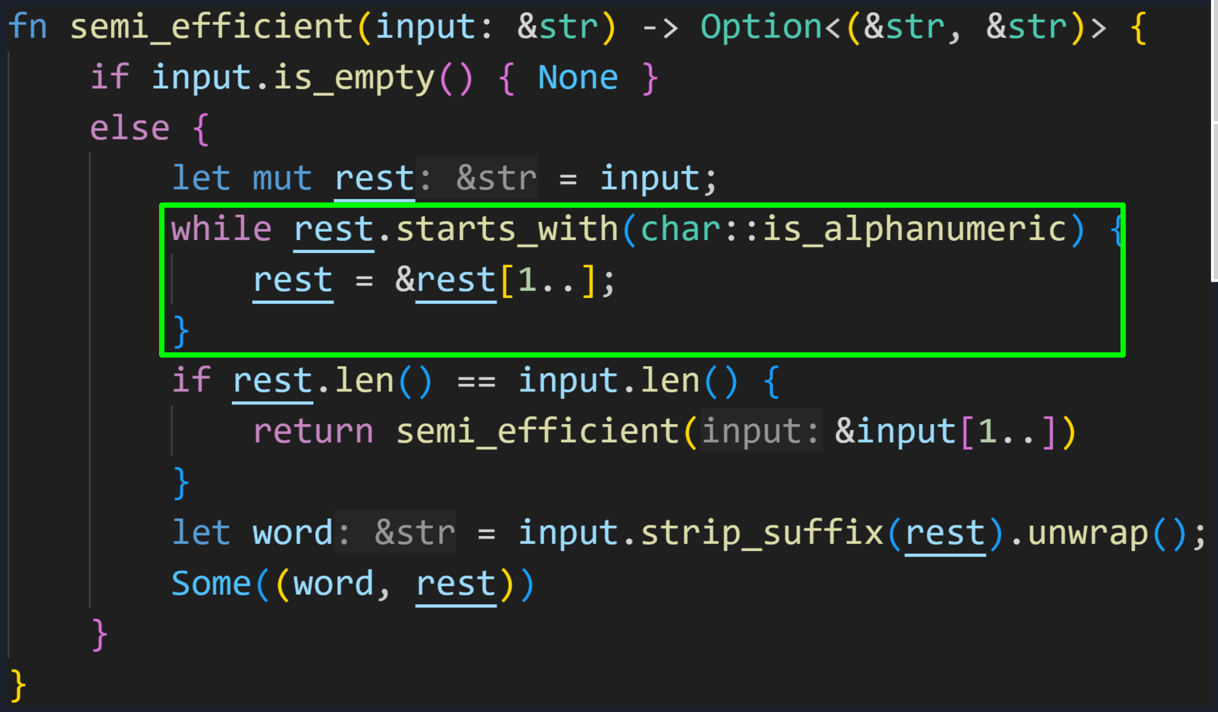


Stack		
input	ptr	0x0
	len	6
rest	ptr	0x0
	len	6

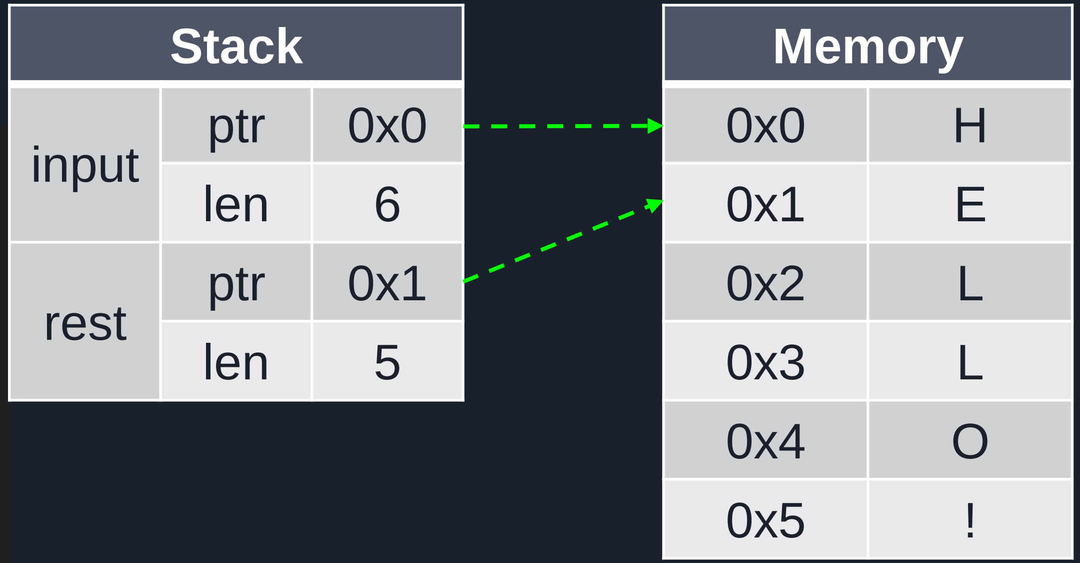
Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

Next slice is at the first element of rest, until the end of rest

2. Slices



```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

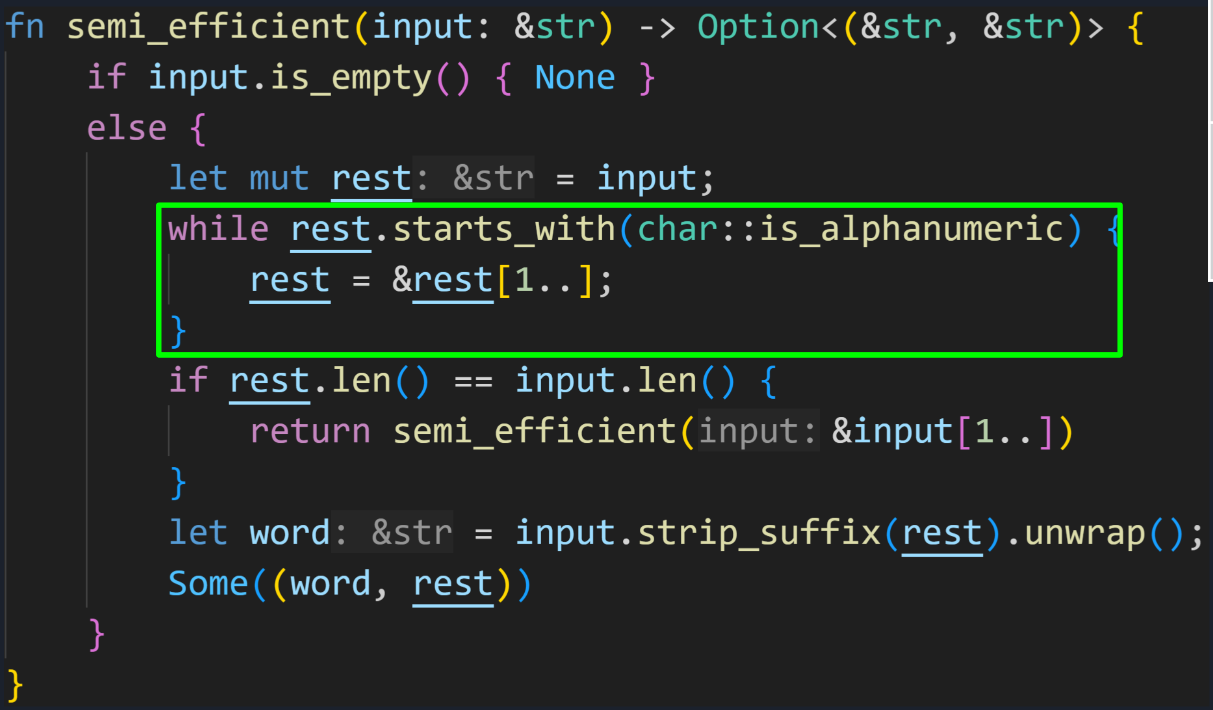


Stack		
input	ptr	0x0
	len	6
rest	ptr	0x1
	len	5

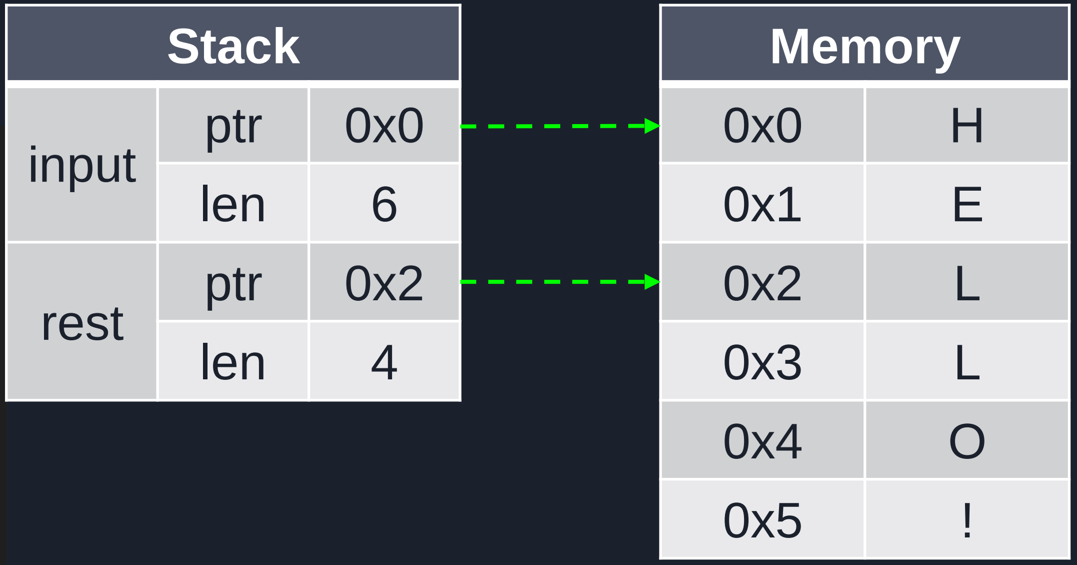
Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

Next slice is at the first element of rest, until the end of rest...
until it‘s no longer alphanumeric

2. Slices



```
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

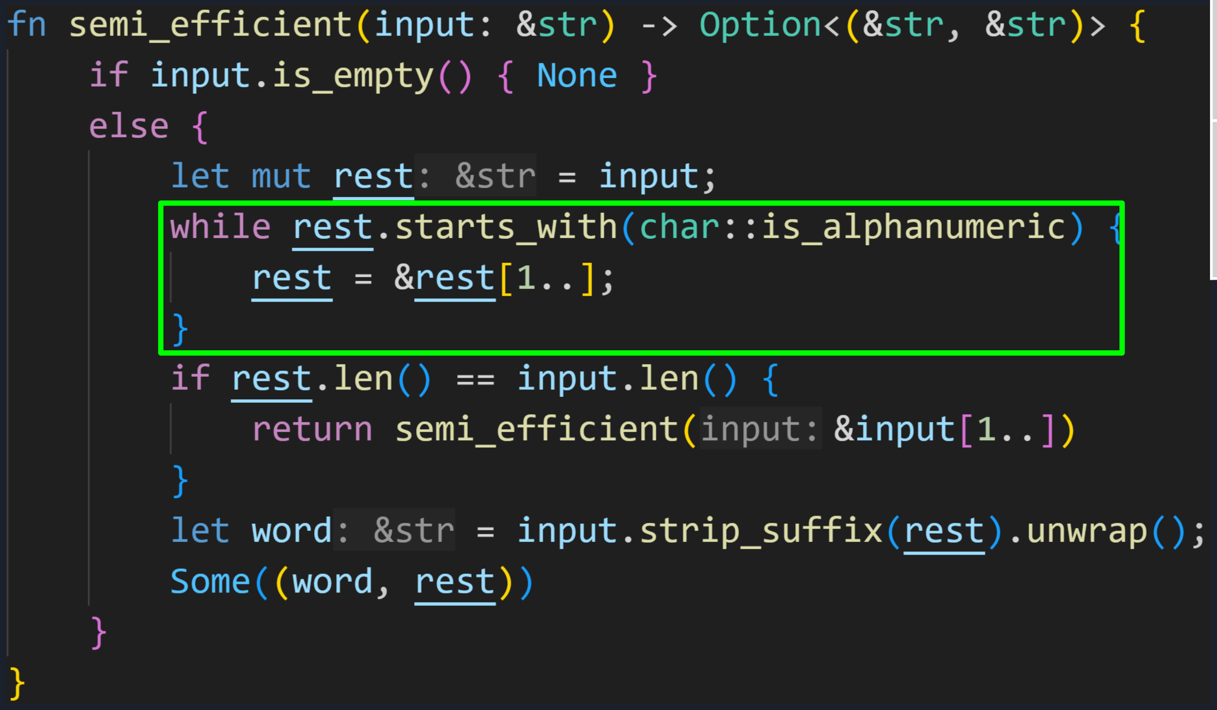


Stack		
input	ptr	0x0
	len	6
rest	ptr	0x2
	len	4

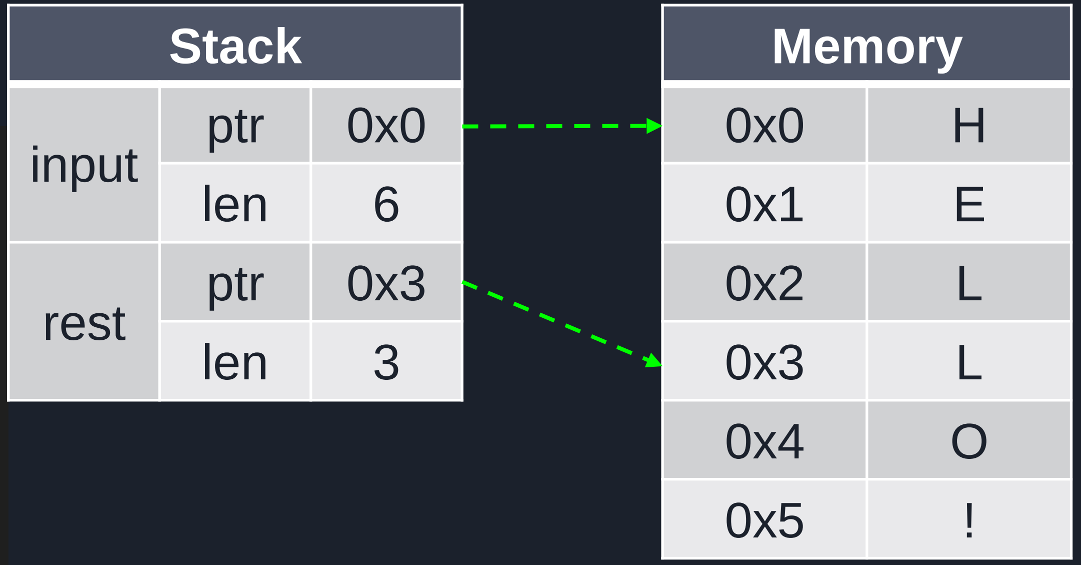
Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

Next slice is at the first element of rest, until the end of rest...
until it‘s no longer alphanumeric

2. Slices



```
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

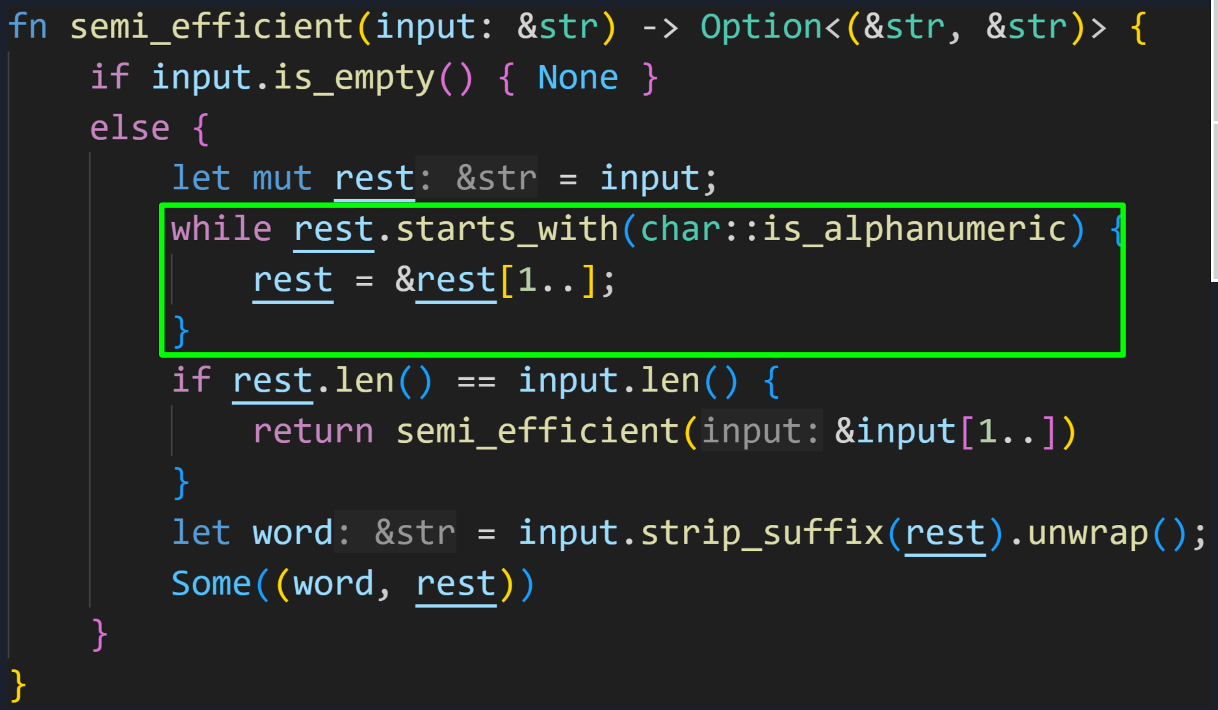


Stack		
input	ptr	0x0
	len	6
rest	ptr	0x3
	len	3

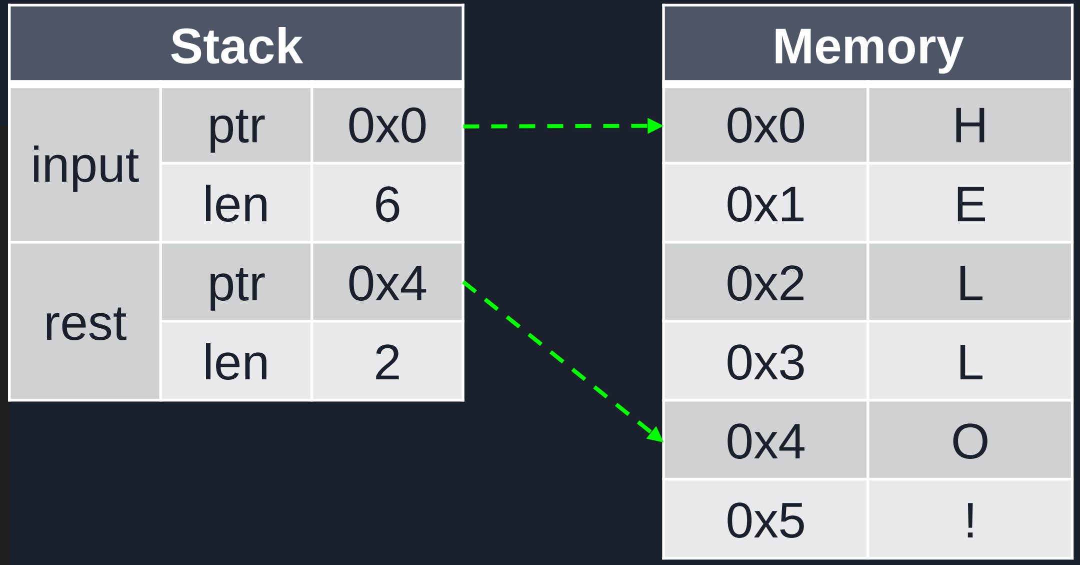
Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

Next slice is at the first element of rest, until the end of rest...
until it's no longer alphanumeric

2. Slices



```
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

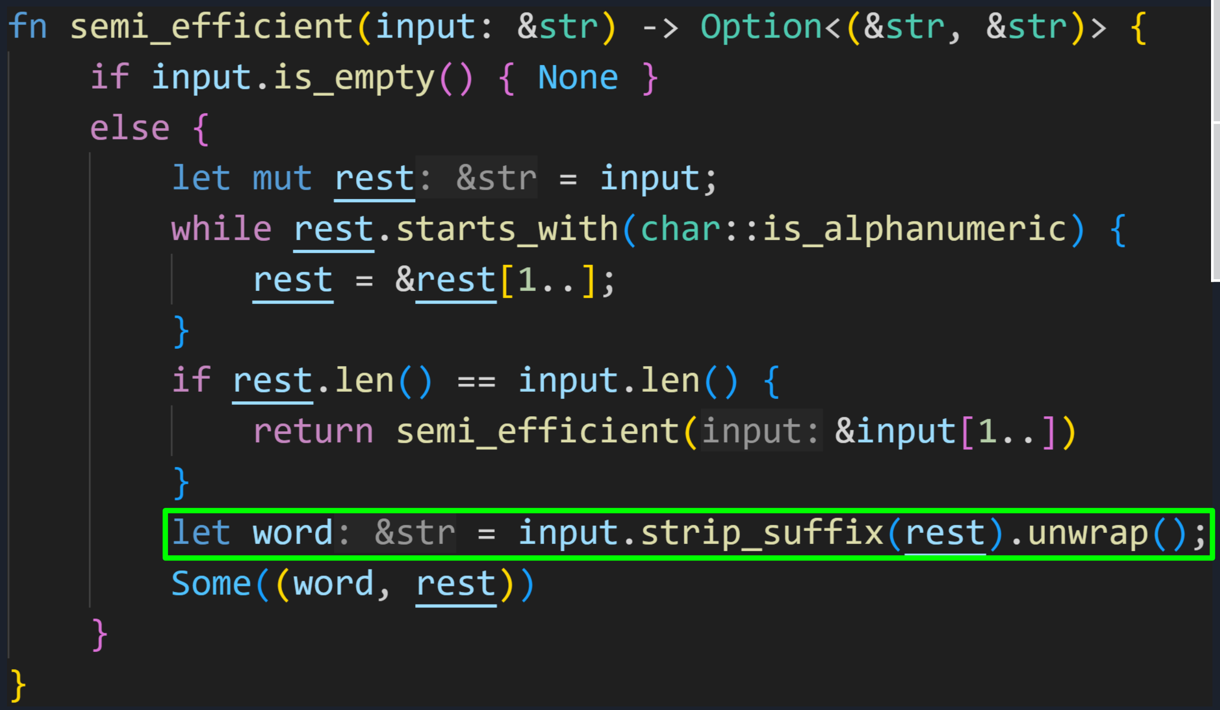


Stack		
input	ptr	0x0
	len	6
rest	ptr	0x4
	len	2

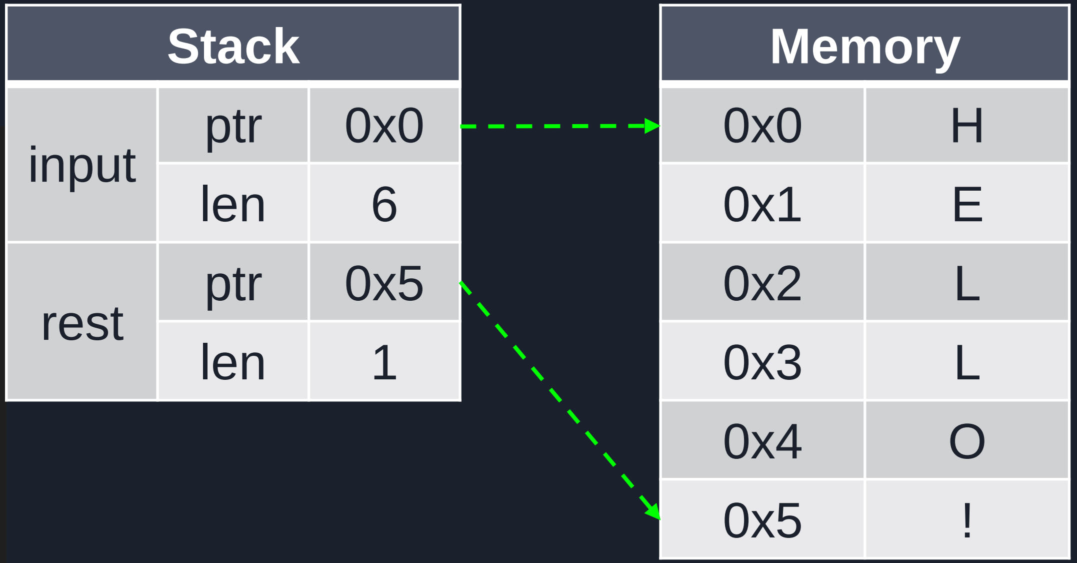
Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

Next slice is at the first element of rest, until the end of rest...
until it's no longer alphanumeric

2. Slices



```
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

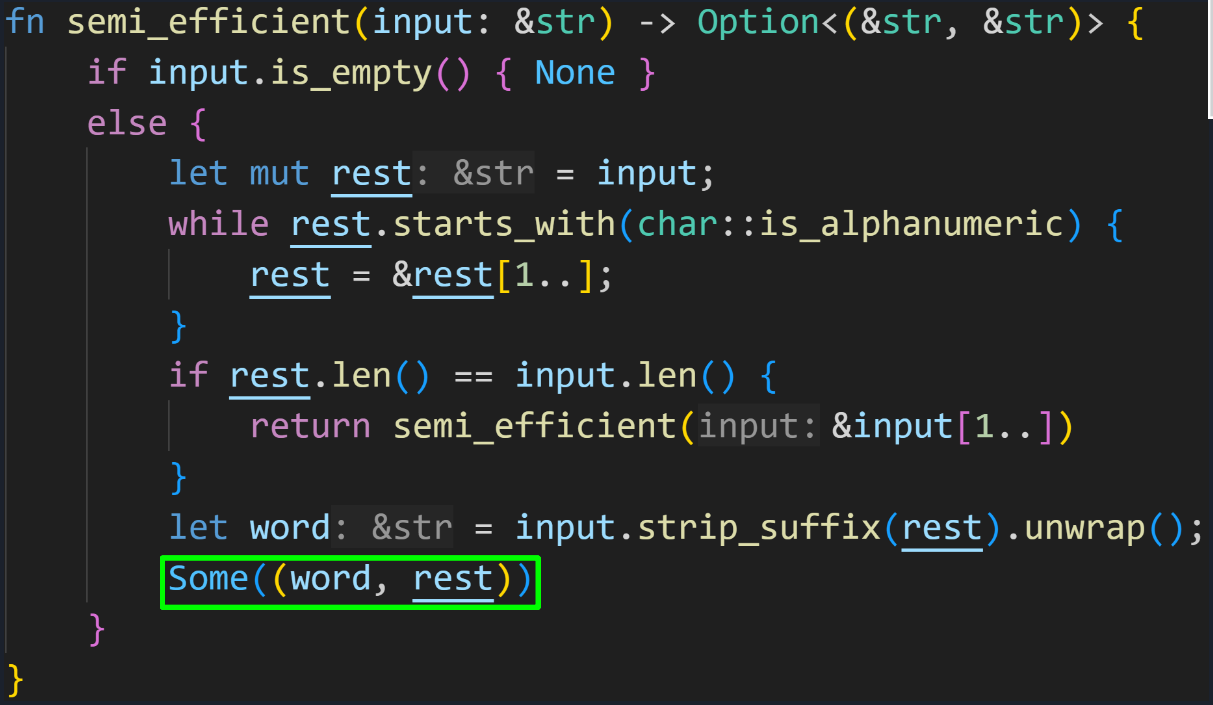


Stack		
input	ptr	0x0
	len	6
rest	ptr	0x5
	len	1

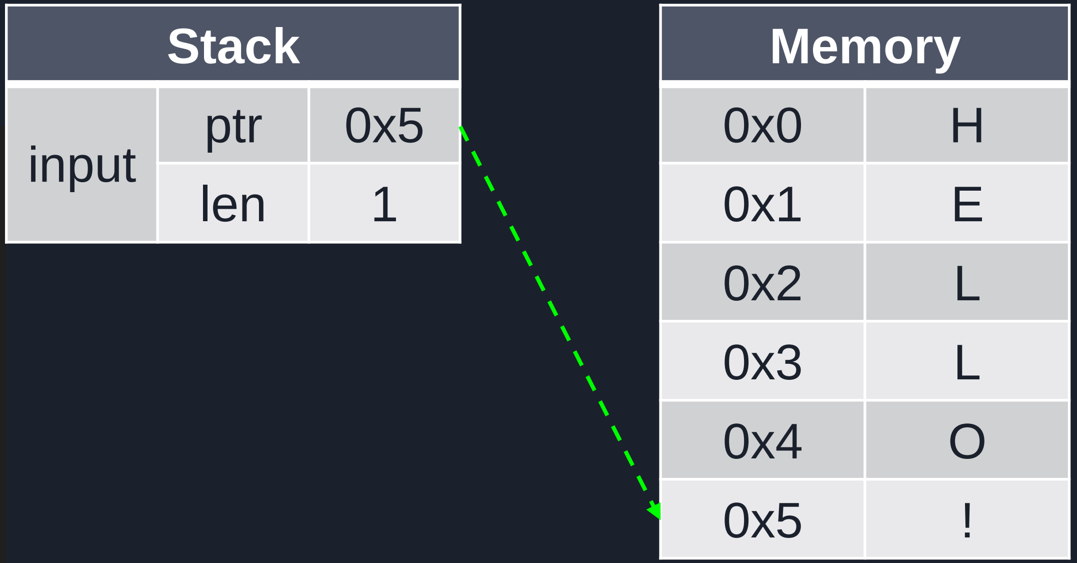
Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

At this point, we perform the strip
→ word = HELLO

2. Slices



```rust
fn semi_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return semi_efficient(input: &input[1..])
        }
        let word: &str = input.strip_suffix(rest).unwrap();
        Some((word, rest))
    }
}
```

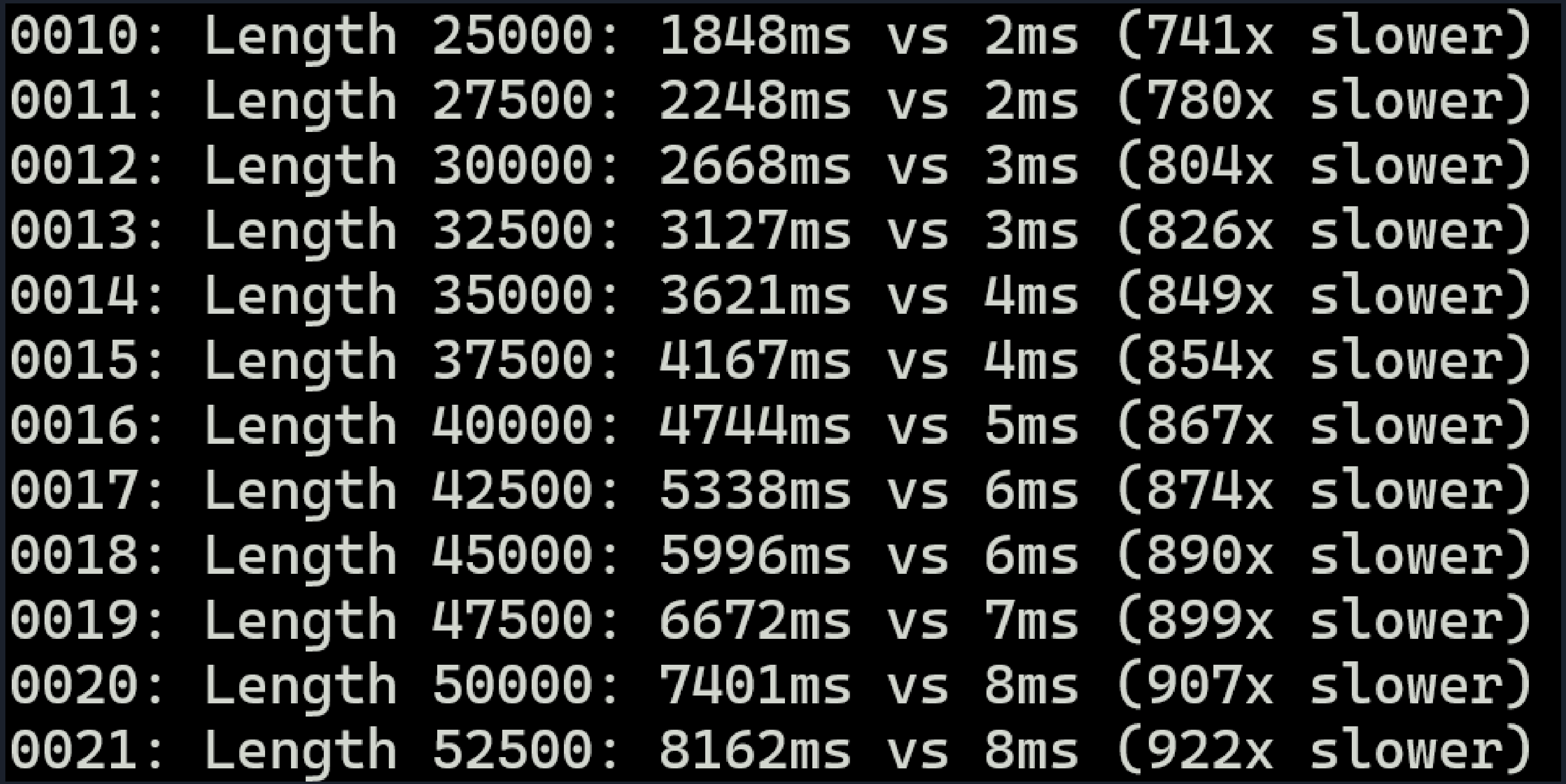


Stack		
input	ptr	0x5
	len	1

Memory	
0x0	H
0x1	E
0x2	L
0x3	L
0x4	O
0x5	!

The caller (hopefully) sets input to rest, and the cycle begins again

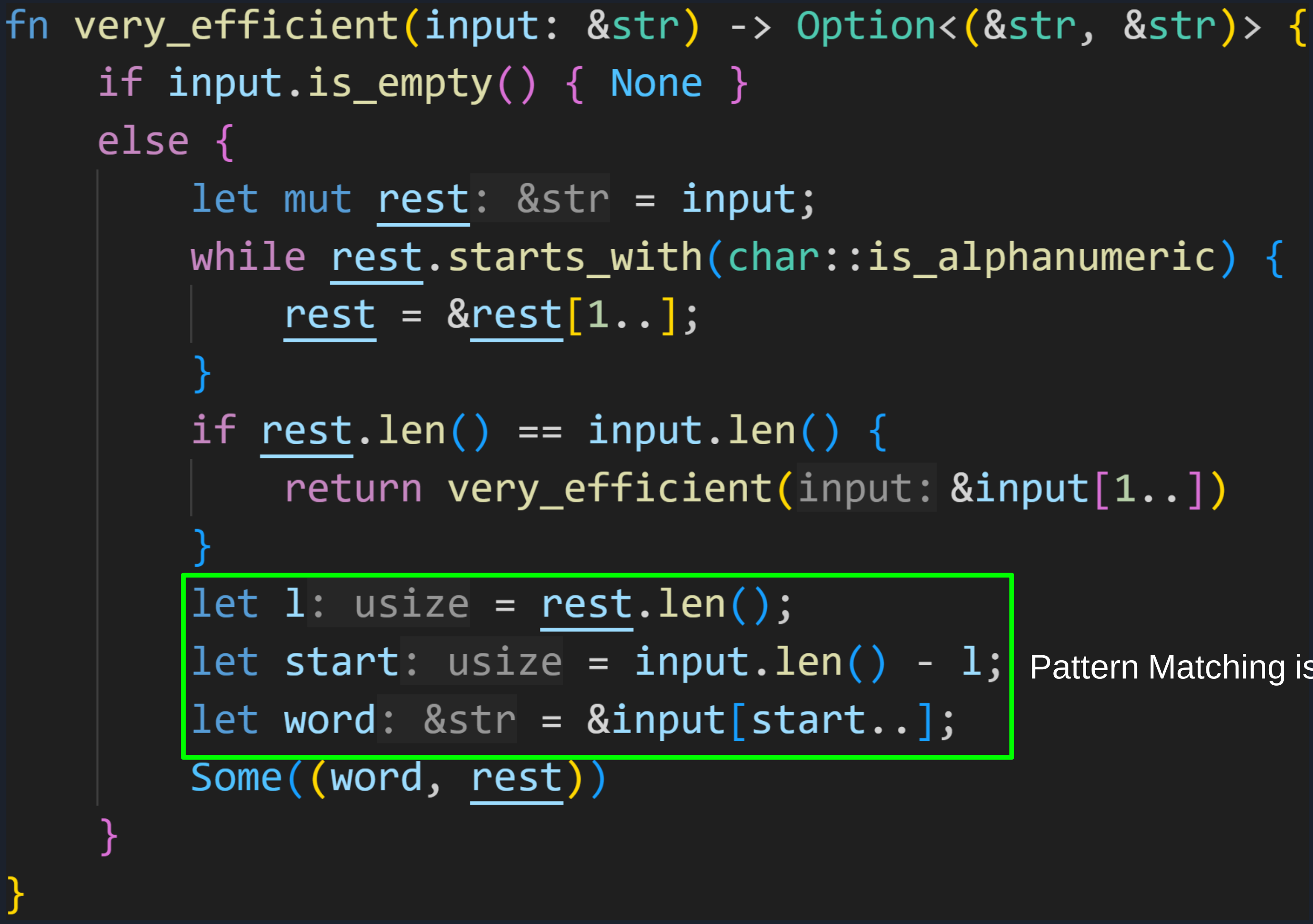
2. Slices



```
0010: Length 25000: 1848ms vs 2ms (741x slower)
0011: Length 27500: 2248ms vs 2ms (780x slower)
0012: Length 30000: 2668ms vs 3ms (804x slower)
0013: Length 32500: 3127ms vs 3ms (826x slower)
0014: Length 35000: 3621ms vs 4ms (849x slower)
0015: Length 37500: 4167ms vs 4ms (854x slower)
0016: Length 40000: 4744ms vs 5ms (867x slower)
0017: Length 42500: 5338ms vs 6ms (874x slower)
0018: Length 45000: 5996ms vs 6ms (890x slower)
0019: Length 47500: 6672ms vs 7ms (899x slower)
0020: Length 50000: 7401ms vs 8ms (907x slower)
0021: Length 52500: 8162ms vs 8ms (922x slower)
```

Nice improvement!

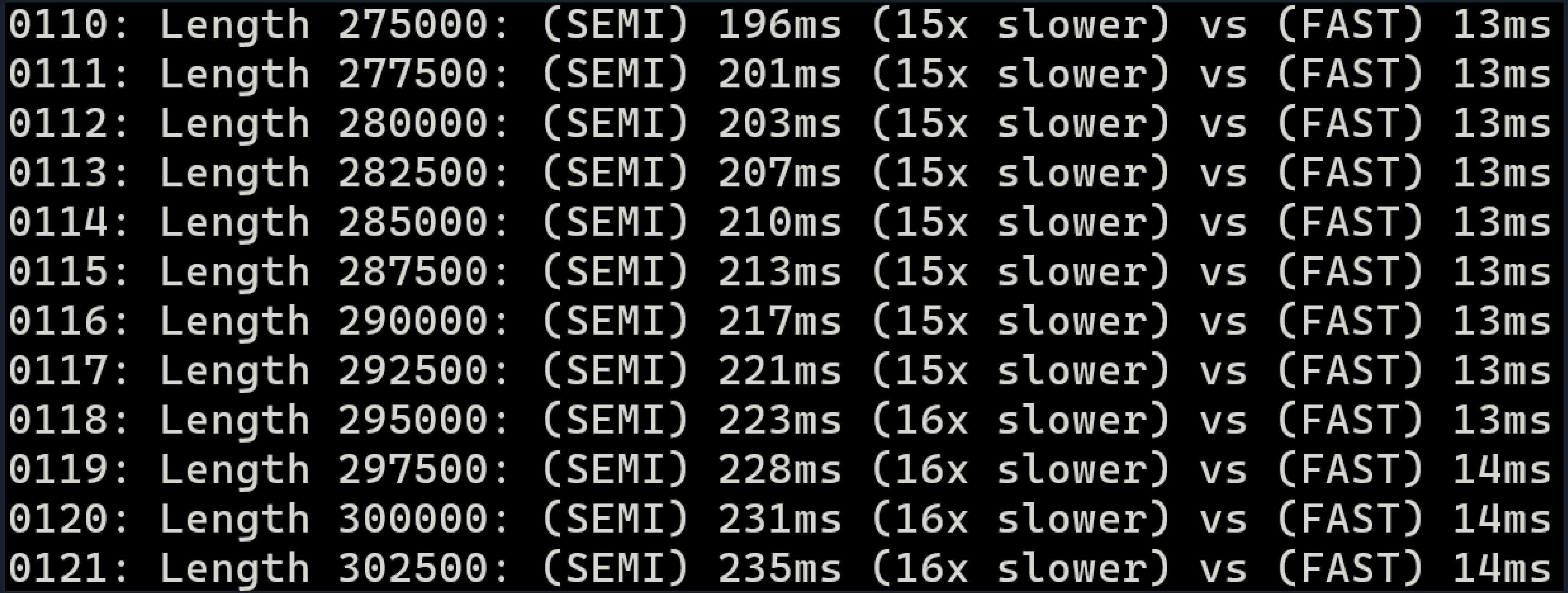
2. Slices



```
fn very_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return very_efficient(input: &input[1..])
        }
        let l: usize = rest.len();
        let start: usize = input.len() - l;
        let word: &str = &input[start..];
        Some((word, rest))
    }
}
```

Pattern Matching is slow, we can do better!

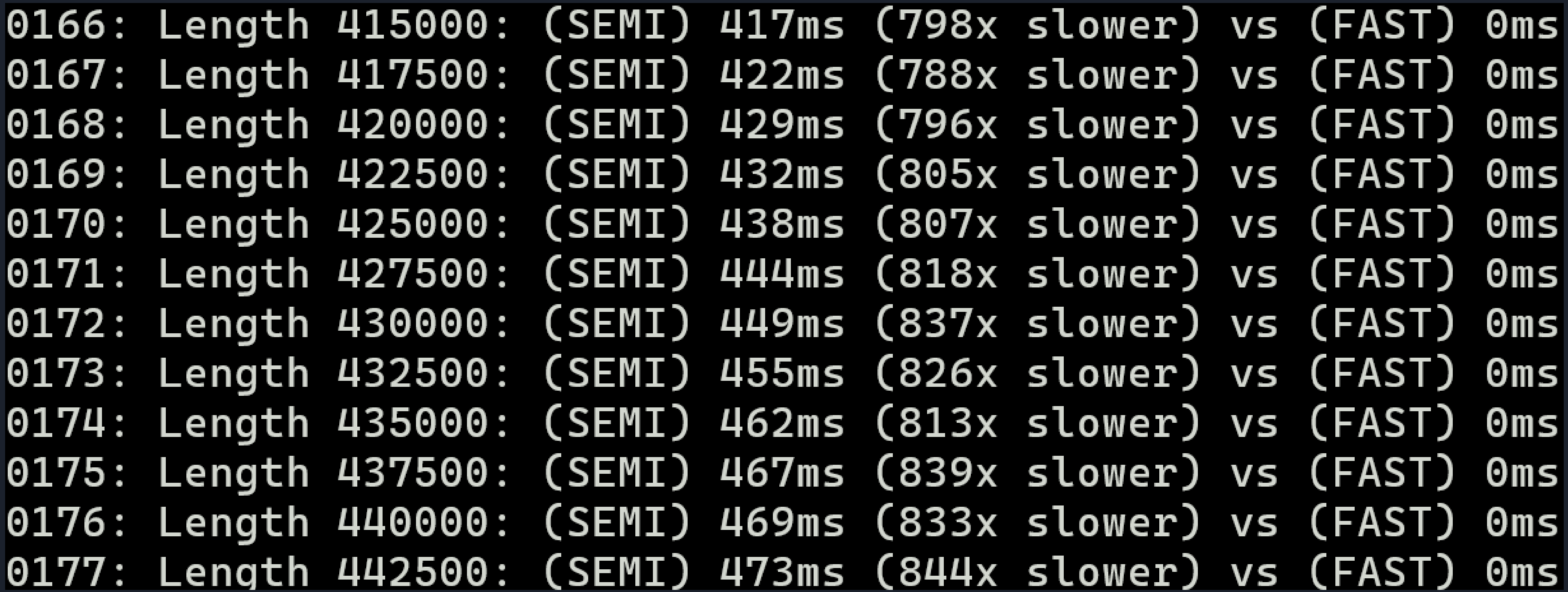
2. Slices



```
0110: Length 275000: (SEMI) 196ms (15x slower) vs (FAST) 13ms
0111: Length 277500: (SEMI) 201ms (15x slower) vs (FAST) 13ms
0112: Length 280000: (SEMI) 203ms (15x slower) vs (FAST) 13ms
0113: Length 282500: (SEMI) 207ms (15x slower) vs (FAST) 13ms
0114: Length 285000: (SEMI) 210ms (15x slower) vs (FAST) 13ms
0115: Length 287500: (SEMI) 213ms (15x slower) vs (FAST) 13ms
0116: Length 290000: (SEMI) 217ms (15x slower) vs (FAST) 13ms
0117: Length 292500: (SEMI) 221ms (15x slower) vs (FAST) 13ms
0118: Length 295000: (SEMI) 223ms (16x slower) vs (FAST) 13ms
0119: Length 297500: (SEMI) 228ms (16x slower) vs (FAST) 14ms
0120: Length 300000: (SEMI) 231ms (16x slower) vs (FAST) 14ms
0121: Length 302500: (SEMI) 235ms (16x slower) vs (FAST) 14ms
```

In debug mode

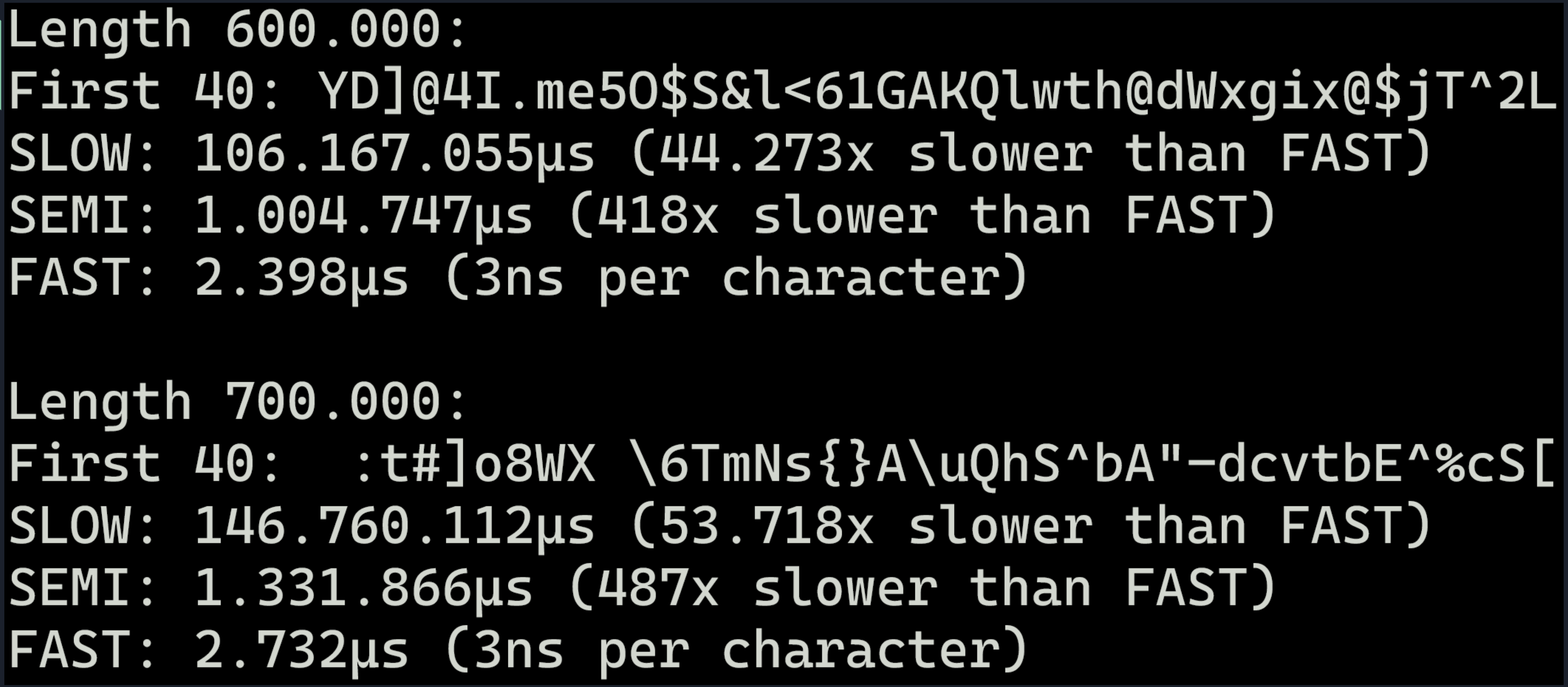
2. Slices



```
0166: Length 415000: (SEMI) 417ms (798x slower) vs (FAST) 0ms
0167: Length 417500: (SEMI) 422ms (788x slower) vs (FAST) 0ms
0168: Length 420000: (SEMI) 429ms (796x slower) vs (FAST) 0ms
0169: Length 422500: (SEMI) 432ms (805x slower) vs (FAST) 0ms
0170: Length 425000: (SEMI) 438ms (807x slower) vs (FAST) 0ms
0171: Length 427500: (SEMI) 444ms (818x slower) vs (FAST) 0ms
0172: Length 430000: (SEMI) 449ms (837x slower) vs (FAST) 0ms
0173: Length 432500: (SEMI) 455ms (826x slower) vs (FAST) 0ms
0174: Length 435000: (SEMI) 462ms (813x slower) vs (FAST) 0ms
0175: Length 437500: (SEMI) 467ms (839x slower) vs (FAST) 0ms
0176: Length 440000: (SEMI) 469ms (833x slower) vs (FAST) 0ms
0177: Length 442500: (SEMI) 473ms (844x slower) vs (FAST) 0ms
```

In release mode

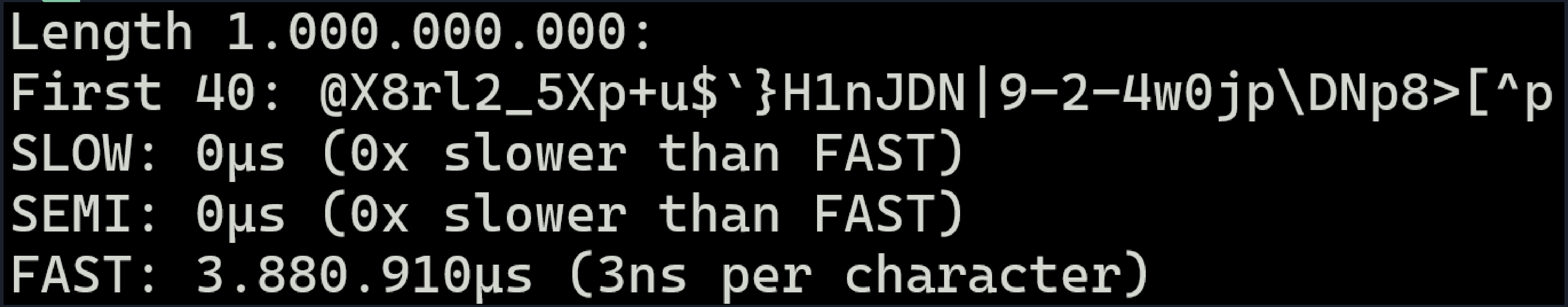
2. Slices



```
Length 600.000:
First 40: YD]@4I.me5O$S&l<61GAKQlwth@dWxgix@$jT^2L
SLOW: 106.167.055µs (44.273x slower than FAST)
SEMI: 1.004.747µs (418x slower than FAST)
FAST: 2.398µs (3ns per character)

Length 700.000:
First 40:  :t#]o8WX \6TmNs{}A\uQhS^bA"-dcvtbE^%cS[
SLOW: 146.760.112µs (53.718x slower than FAST)
SEMI: 1.331.866µs (487x slower than FAST)
FAST: 2.732µs (3ns per character)
```

Rust is a *very* efficient language, if you know how to utilize it properly

2. Slices



```
Length 1.000.000.000:
First 40: @X8rl2_5Xp+u$`}H1nJDN|9-2-4w0jp\DNp8>[^p
SLOW: 0µs (0x slower than FAST)
SEMI: 0µs (0x slower than FAST)
FAST: 3.880.910µs (3ns per character)
```

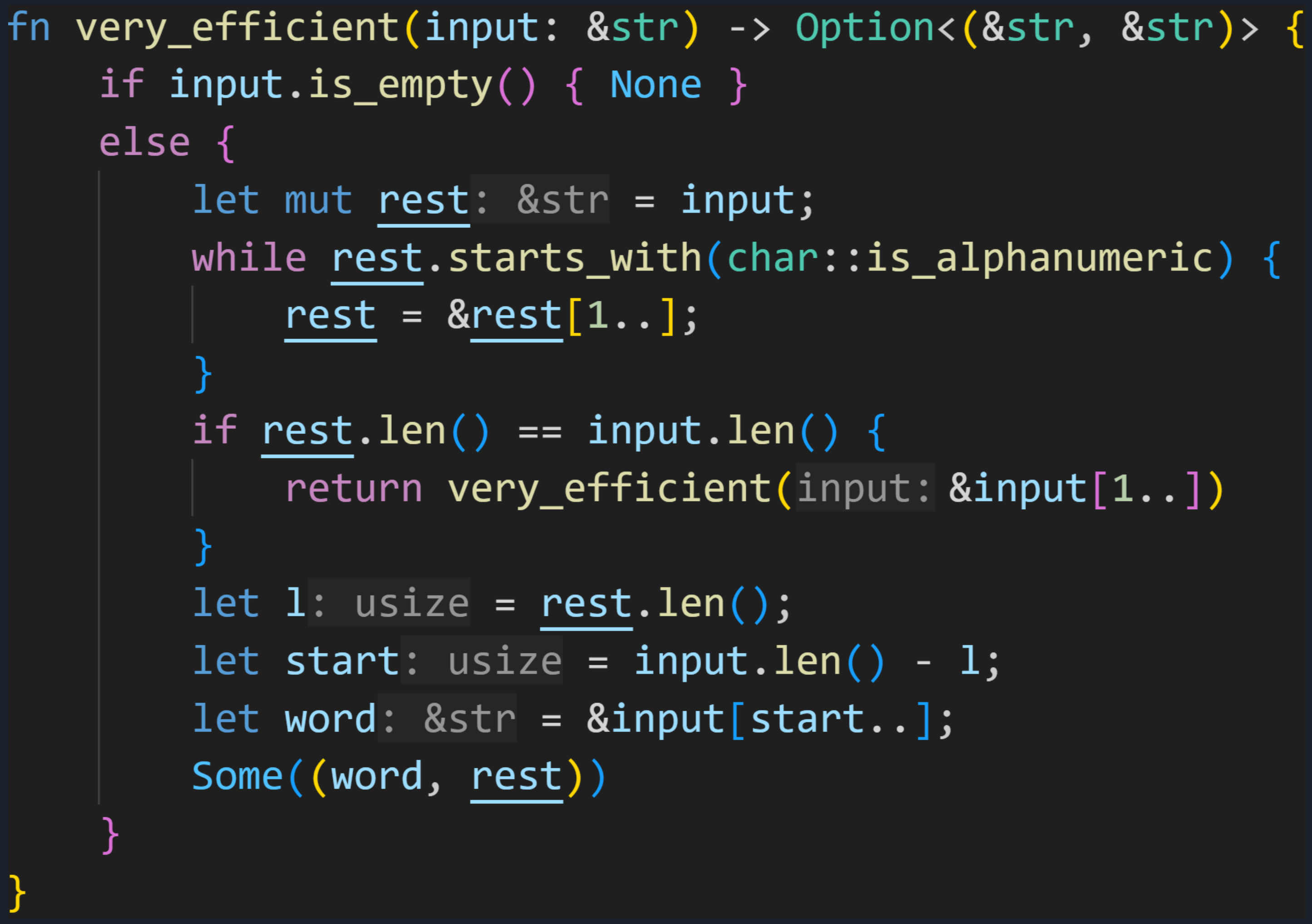
Rust is a *very* efficient language, if you know how to utilize it properly

→ 4 seconds to get all words in a string of length 1 billion

→ Allocating the String almost took longer than that :^)

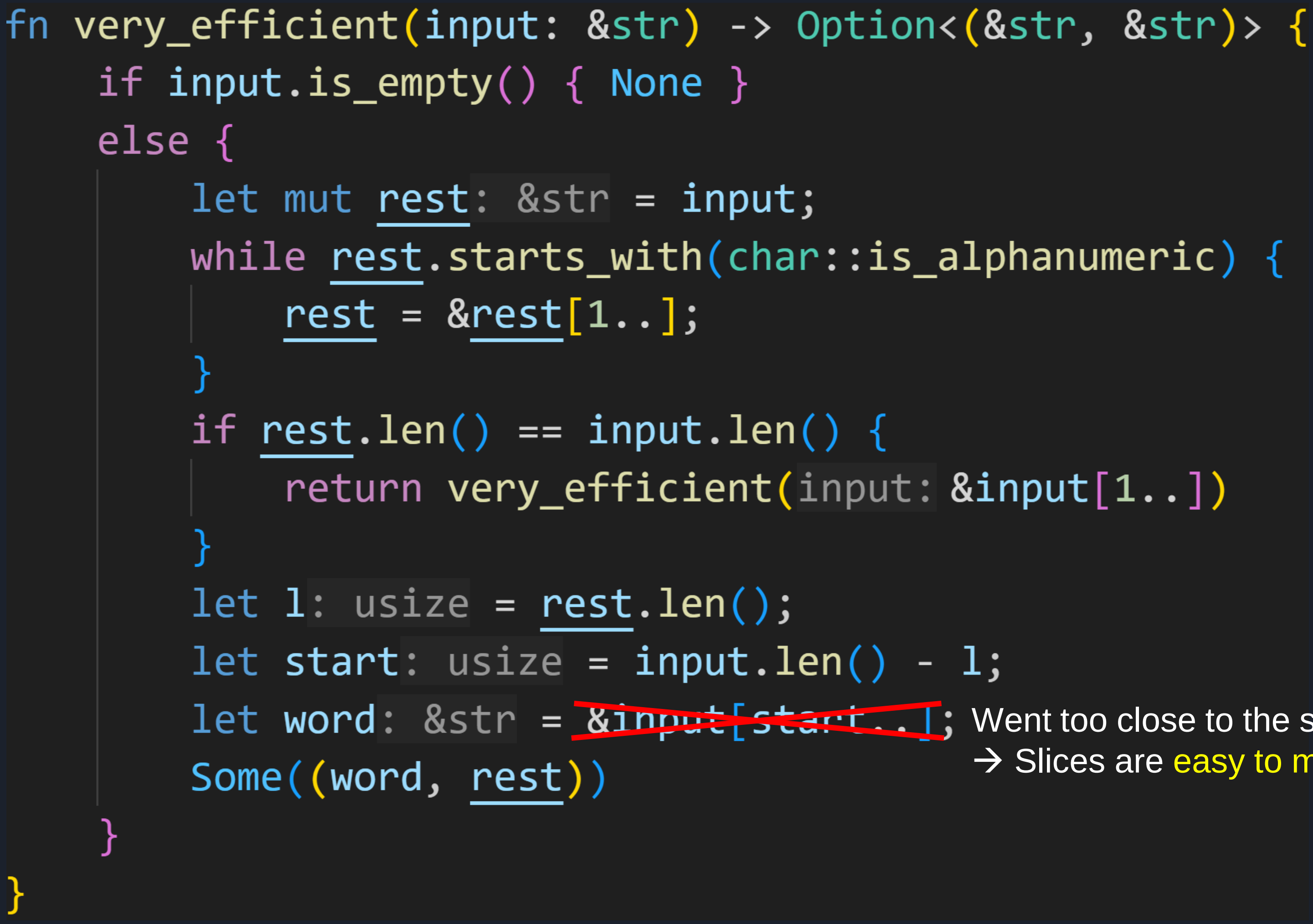
→ SLOW would've taken years... Literally

2. Slices



```
fn very_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return very_efficient(input: &input[1..])
        }
        let l: usize = rest.len();
        let start: usize = input.len() - l;
        let word: &str = &input[start..];
        Some((word, rest))
    }
}
```

2. Slices

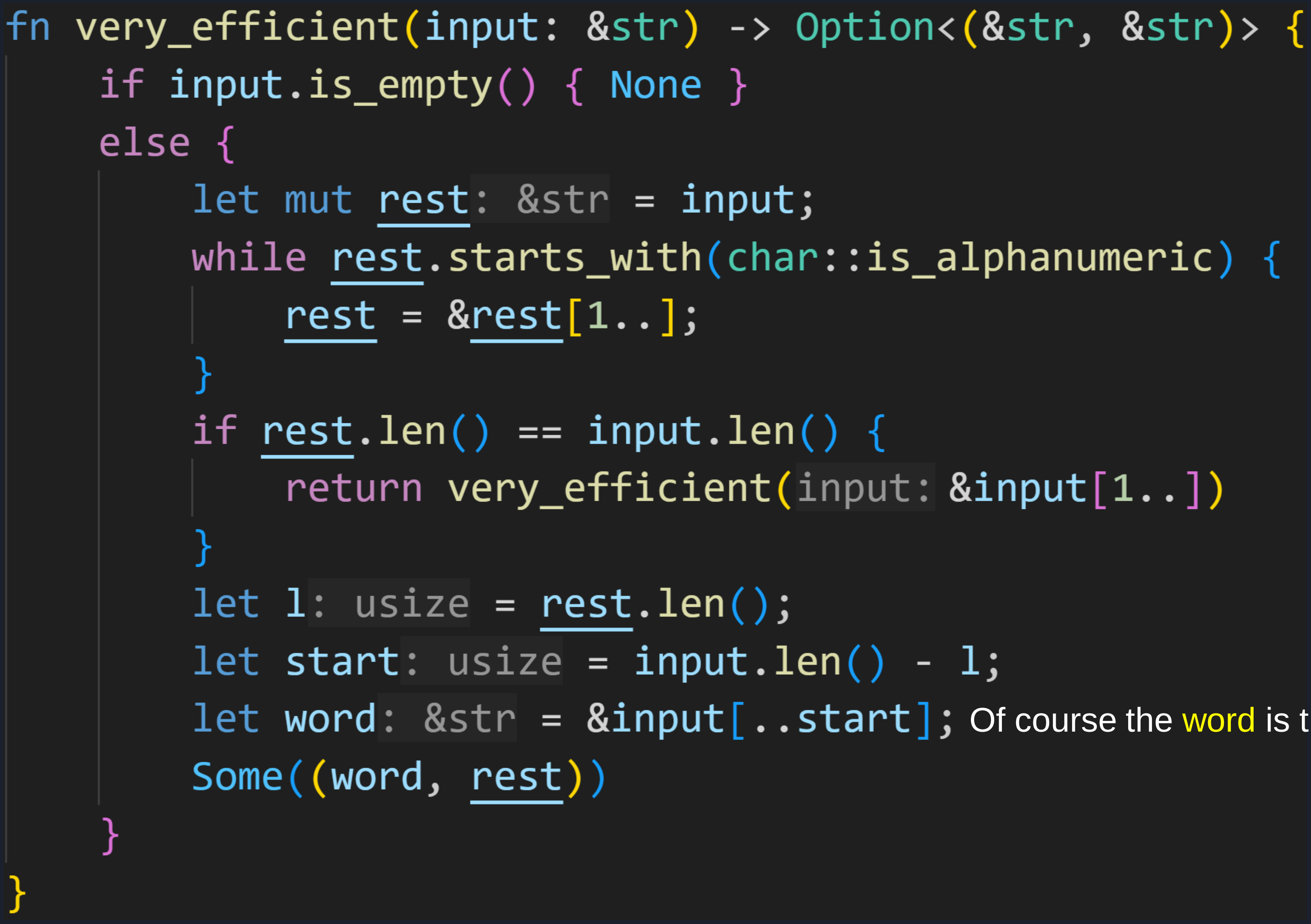


```
fn very_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return very_efficient(input: &input[1..])
        }
        let l: usize = rest.len();
        let start: usize = input.len() - l;
        let word: &str = ~~&input[start..]~~;
        Some((word, rest))
    }
}
```

Went too close to the sun, and introduced a bug

→ Slices are easy to mishandle

2. Slices

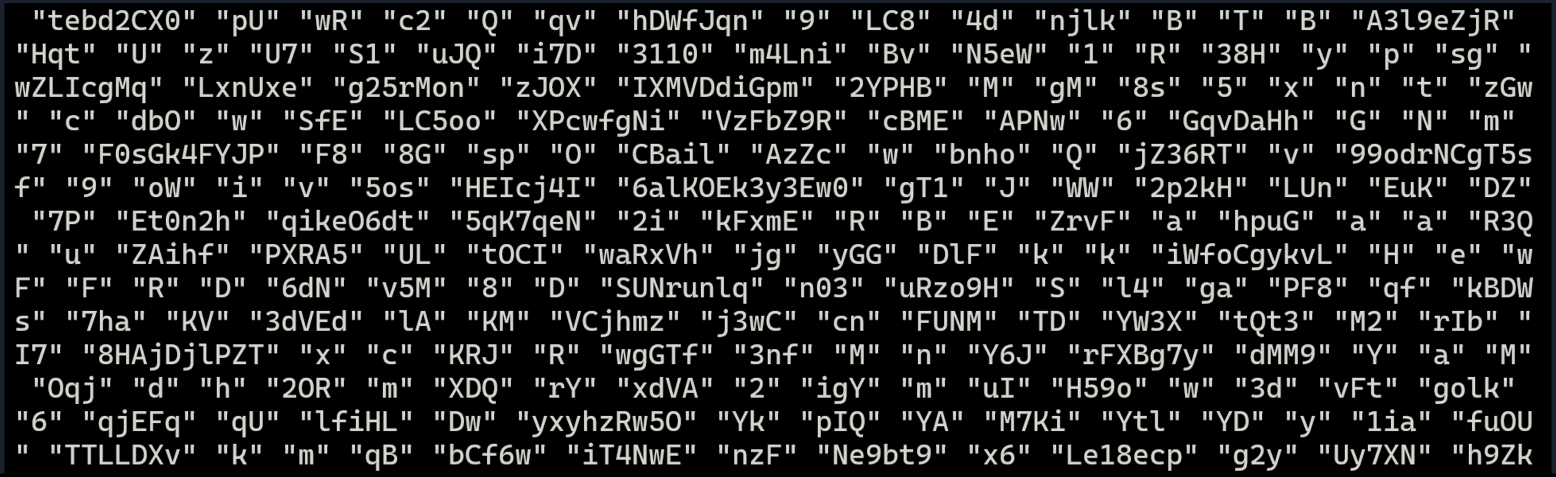


```rust
fn very_efficient(input: &str) -> Option<(&str, &str)> {
    if input.is_empty() { None }
    else {
        let mut rest: &str = input;
        while rest.starts_with(char::is_alphanumeric) {
            rest = &rest[1..];
        }
        if rest.len() == input.len() {
            return very_efficient(input: &input[1..])
        }
        let l: usize = rest.len();
        let start: usize = input.len() - l;
        let word: &str = &input[..start];
        Some((word, rest))
    }
}
```

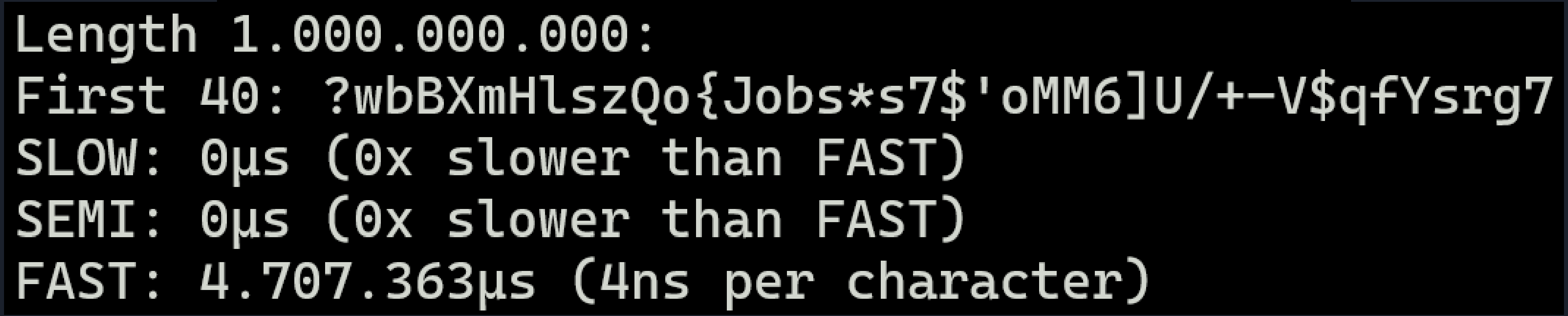
Of course the word is the start :^)

2. Slices

Subwords:



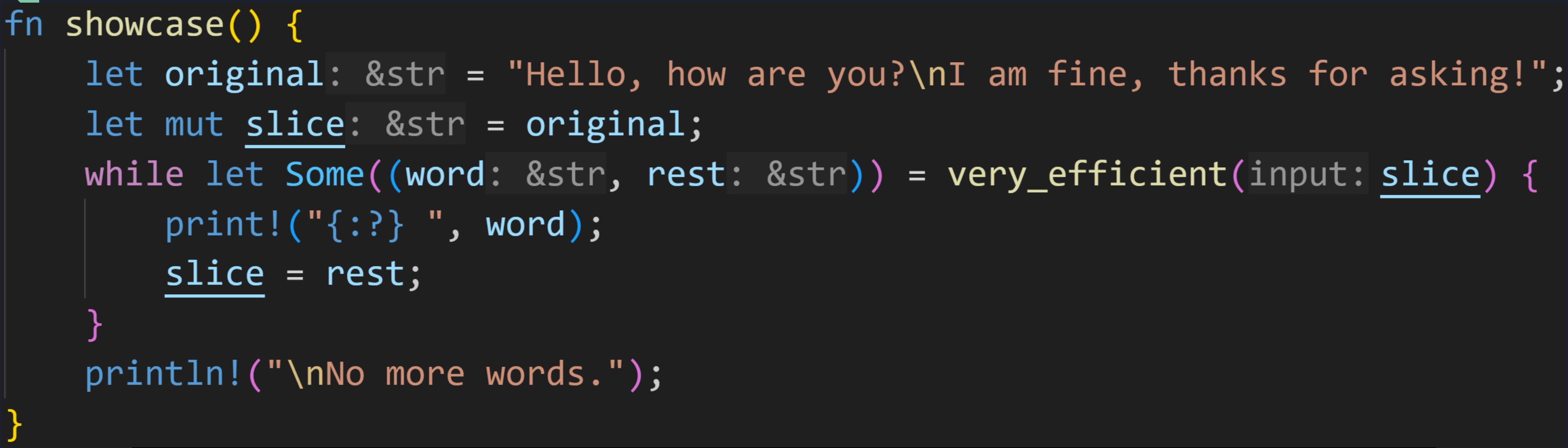
```
 "tebd2CX0" "pU" "wR" "c2" "Q" "qv" "hDWfJqn" "9" "LC8" "4d" "njlk" "B" "T" "B" "A3l9eZjR"
"Hqt" "U" "z" "U7" "S1" "uJQ" "i7D" "3110" "m4Lni" "Bv" "N5eW" "1" "R" "38H" "y" "p" "sg" "
wZLIcgMq" "LxnUxe" "g25rMon" "zJOX" "IXMVDdiGpm" "2YPHB" "M" "gM" "8s" "5" "x" "n" "t" "zGw
" "c" "dbO" "w" "SfE" "LC5oo" "XPcwfgNi" "VzFbZ9R" "cBME" "APNw" "6" "GqvDaHh" "G" "N" "m"
"7" "F0sGk4FYJP" "F8" "8G" "sp" "O" "CBail" "AzZc" "w" "bnho" "Q" "jZ36RT" "v" "99odrNCgT5s
f" "9" "oW" "i" "v" "5os" "HEIcj4I" "6alKOEk3y3Ew0" "gT1" "J" "WW" "2p2kH" "LUn" "EuK" "DZ"
 "7P" "Et0n2h" "qikeO6dt" "5qK7qeN" "2i" "kFxmE" "R" "B" "E" "ZrvF" "a" "hpuG" "a" "a" "R3Q
" "u" "ZAihf" "PXRA5" "UL" "tOCI" "waRxVh" "jg" "yGG" "DlF" "k" "k" "iWfoCgykvL" "H" "e" "w
F" "F" "R" "D" "6dN" "v5M" "8" "D" "SUNrunlq" "n03" "uRzo9H" "S" "l4" "ga" "PF8" "qf" "kBDW
s" "7ha" "KV" "3dVEd" "lA" "KM" "VCjhmz" "j3wC" "cn" "FUNM" "TD" "YW3X" "tQt3" "M2" "rIb" "
I7" "8HAjDjlPZT" "x" "c" "KRJ" "R" "wgGTf" "3nf" "M" "n" "Y6J" "rFXBg7y" "dMM9" "Y" "a" "M"
 "Oqj" "d" "h" "2OR" "m" "XDQ" "rY" "xdVA" "2" "igY" "m" "uI" "H59o" "w" "3d" "vFt" "golk"
"6" "qjEFq" "qU" "lfiHL" "Dw" "yxyhzRw5O" "Yk" "pIQ" "YA" "M7Ki" "Ytl" "YD" "y" "1ia" "fuOU
" "TTLLDXv" "k" "m" "qB" "bCf6w" "iT4NwE" "nzF" "Ne9bt9" "x6" "Le18ecp" "g2y" "Uy7XN" "h9Zk
```



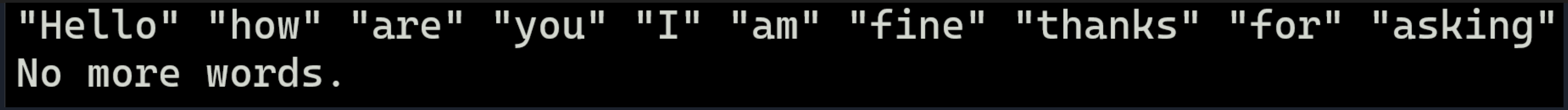
```
Length 1.000.000.000:
First 40: ?wbBXmHlszQo{Jobs*s7$'oMM6]U/+-V$qfYsrg7
SLOW: 0µs (0x slower than FAST)
SEMI: 0µs (0x slower than FAST)
FAST: 4.707.363µs (4ns per character)
```

Still very fast :^)

2. Slices

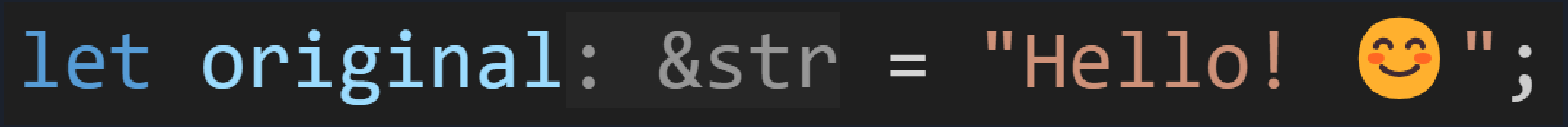


```rust
fn showcase() {
    let original: &str = "Hello, how are you?\nI am fine, thanks for asking!";
    let mut slice: &str = original;
    while let Some((word: &str, rest: &str)) = very_efficient(input: slice) {
        print!("{:?} ", word);
        slice = rest;
    }
    println!("\nNo more words.");
}
```



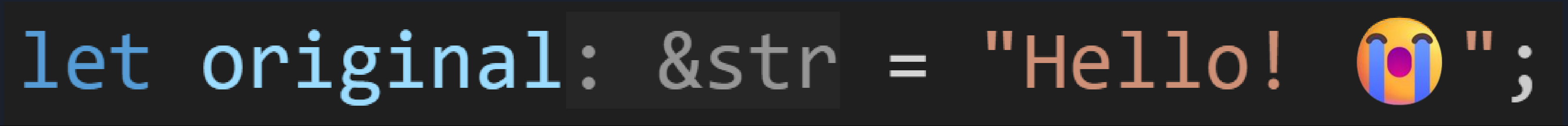
```
"Hello" "how" "are" "you" "I" "am" "fine" "thanks" "for" "asking"
No more words.
```

2. Slices

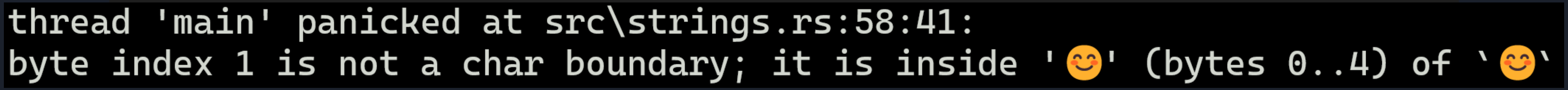


```rust
let original: &str = "Hello! 😊";
```

2. Slices

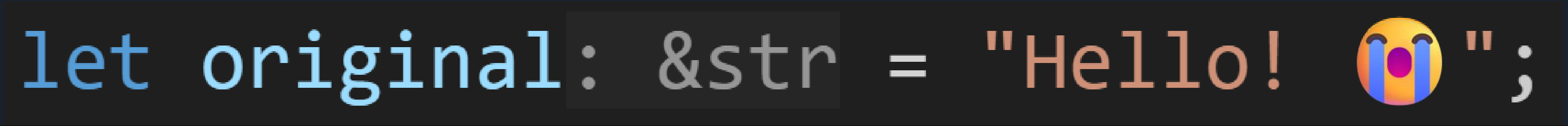


```
let original: &str = "Hello! 😭";
```

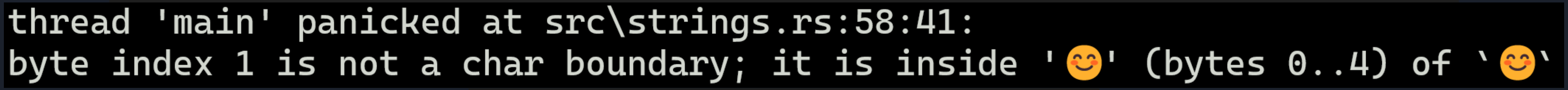


```
thread 'main' panicked at src\strings.rs:58:41:
byte index 1 is not a char boundary; it is inside '😊' (bytes 0..4) of `😊`
```

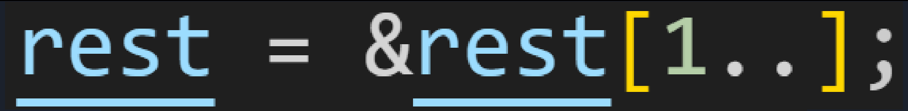
2. Slices



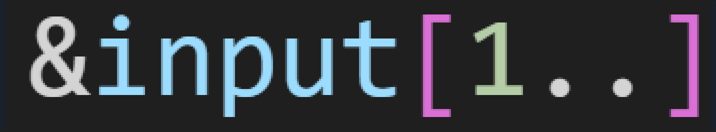
```rust
let original: &str = "Hello! 😭";
```



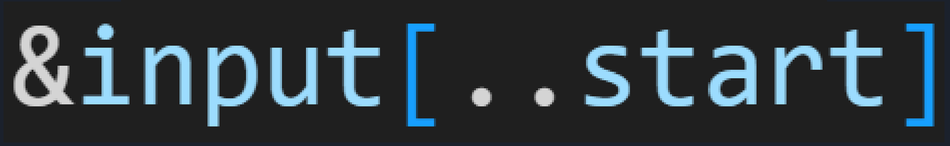
```
thread 'main' panicked at src\strings.rs:58:41:
byte index 1 is not a char boundary; it is inside '😊' (bytes 0..4) of `😊`
```



```rust
rest = &rest[1..];
```

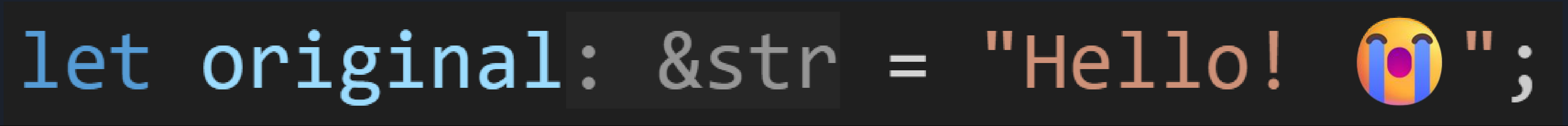


```rust
&input[1..]
```

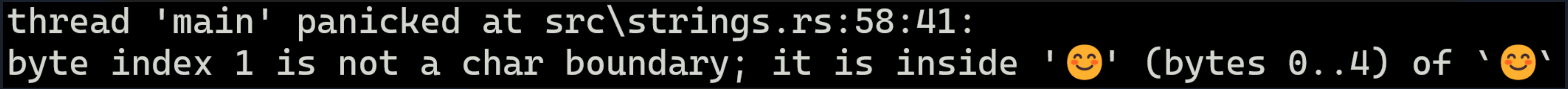


```rust
&input[..start]
```

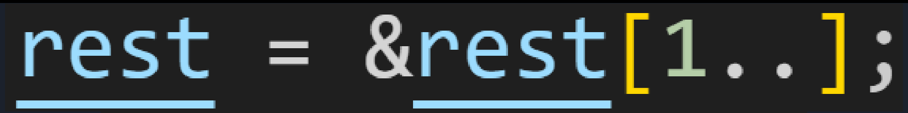
2. Slices



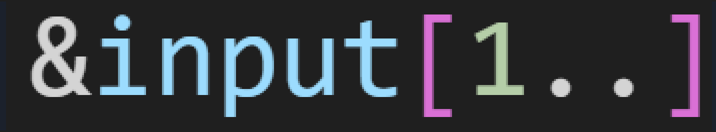
```rust
let original: &str = "Hello! 😭";
```



```
thread 'main' panicked at src\strings.rs:58:41:
byte index 1 is not a char boundary; it is inside '😊' (bytes 0..4) of `😊`
```

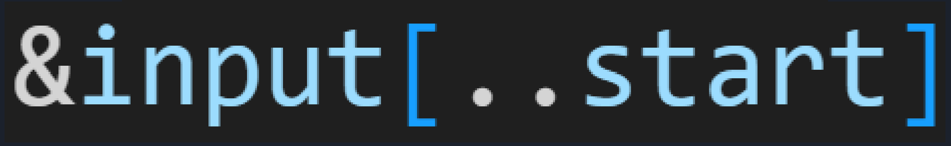


```rust
rest = &rest[1..];
```



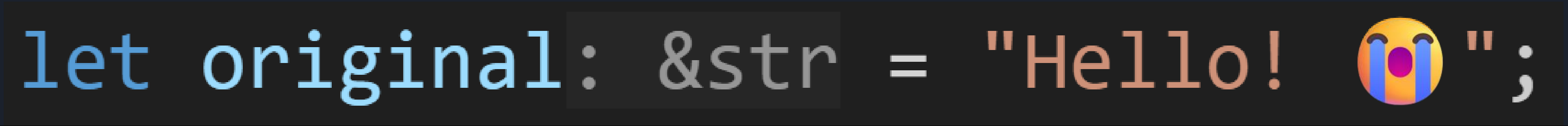
```rust
&input[1..]
```

Byte indices, not grapheme indices!

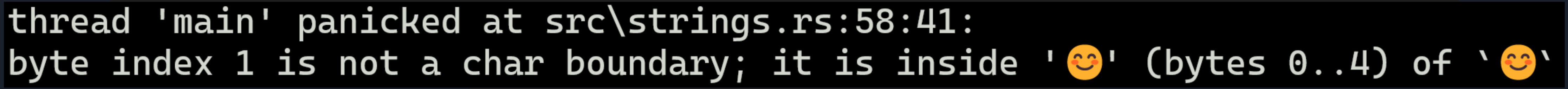


```rust
&input[..start]
```

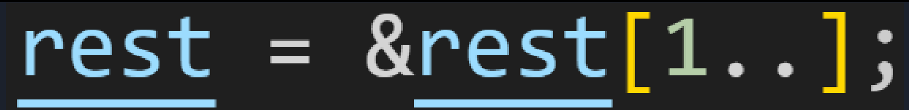
2. Slices



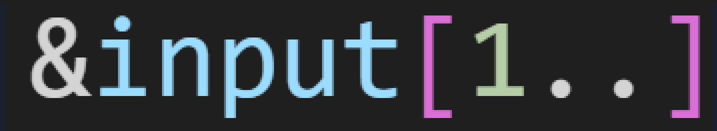
```
let original: &str = "Hello! 😭";
```



```
thread 'main' panicked at src\strings.rs:58:41:
byte index 1 is not a char boundary; it is inside '😊' (bytes 0..4) of `😊`
```

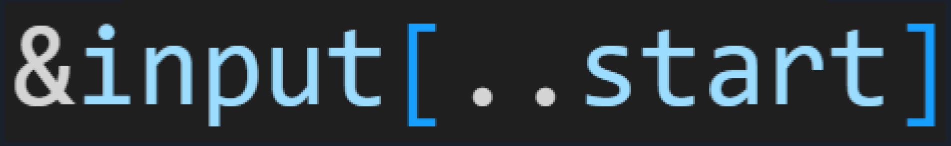


```
rest = &rest[1..];
```



```
&input[1..]
```

Byte indices, not grapheme indices!

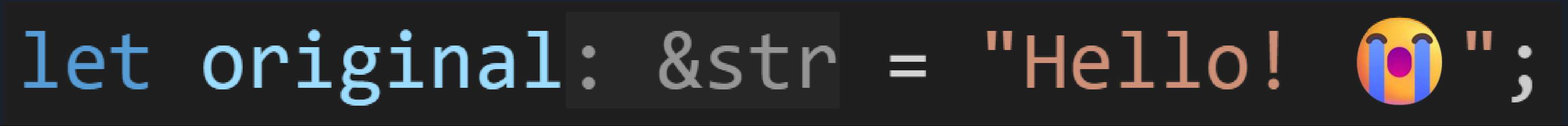


```
&input[..start]
```

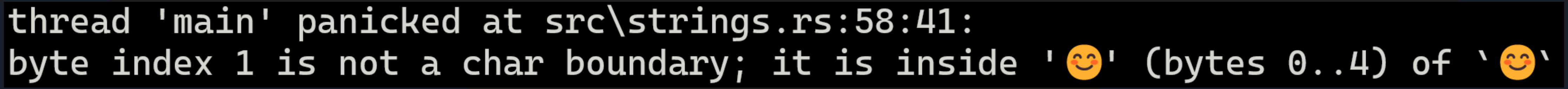
4/3

Exercise for you: Make this code UTF-8 compliant :^)

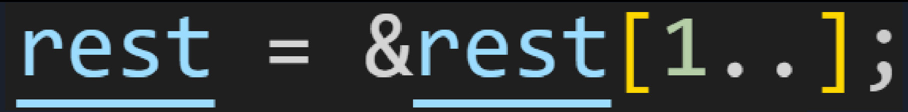
2. Slices



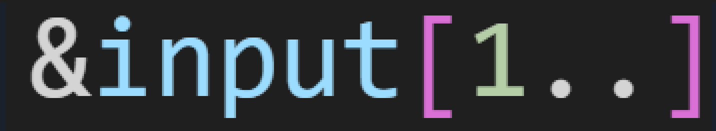
```rust
let original: &str = "Hello! 😭";
```



```
thread 'main' panicked at src\strings.rs:58:41:
byte index 1 is not a char boundary; it is inside '😊' (bytes 0..4) of `😊`
```

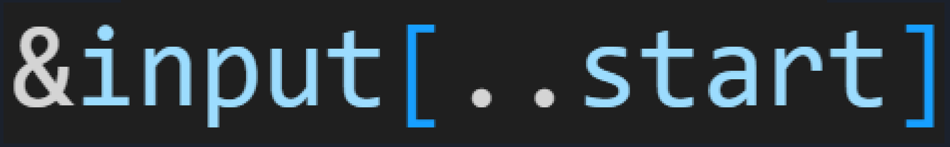


```rust
rest = &rest[1..];
```



```rust
&input[1..]
```

Byte indices, not grapheme indices!



```rust
&input[..start]
```

4/3

Exercise for you: Make this code UTF-8 compliant :^)

The slow version accepts this string, but is that a good tradeoff?

2. Slices

- Slices are a very powerful tool

2. Slices

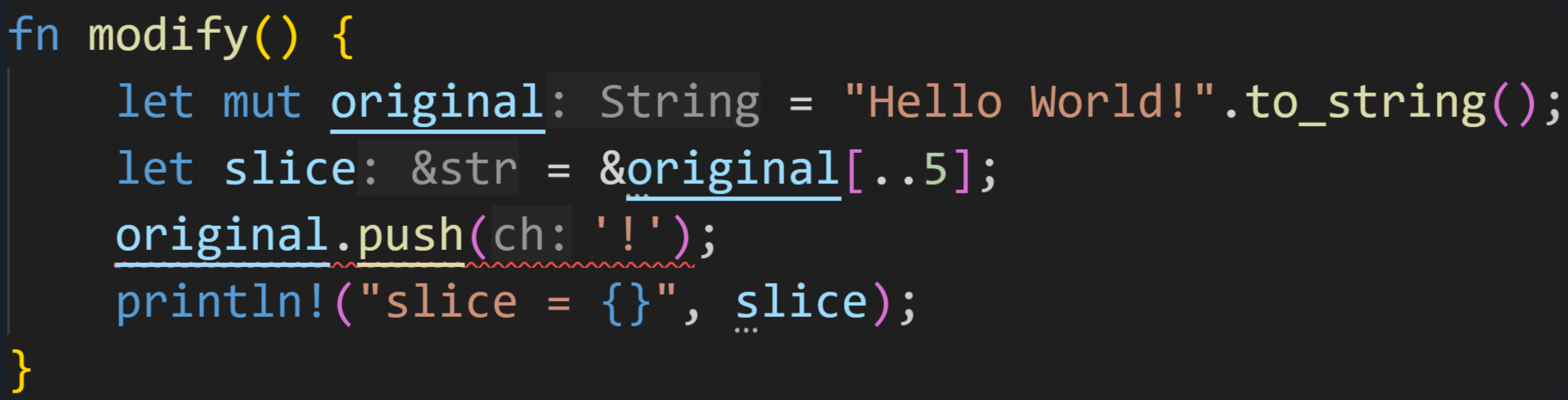
- Slices are a very powerful tool
- Allow us to efficiently work on sub-collections without copying any data

2. Slices

- Slices are a very powerful tool
- Allow us to efficiently work on sub-collections without copying any data
- Slices are super fast
  - Only needs a pointer and a length → CPUs are *very* good at numbercrunching

2. Slices

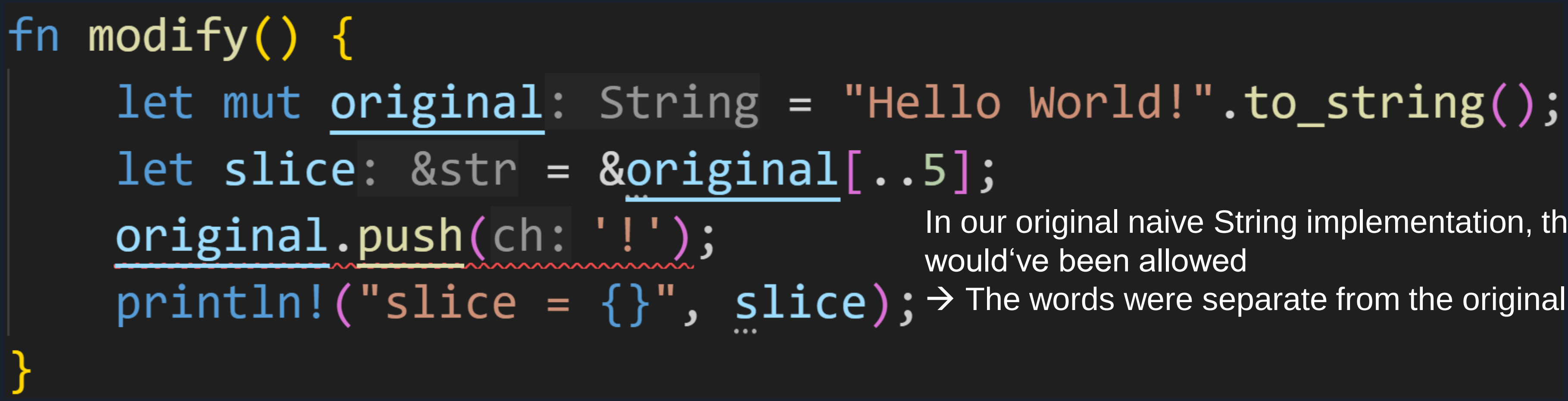
- Slices are a very powerful tool
- Allow us to efficiently work on sub-collections without copying any data
- Slices are super fast
- Because Slices point into the original collection, normal Borrow Checker rules apply



```
fn modify() {
    let mut original: String = "Hello World!".to_string();
    let slice: &str = &original[..5];
    original.push(ch: '!');
    println!("slice = {}", slice);
}
```

2. Slices

- Slices are a very powerful tool
- Allow us to efficiently work on sub-collections without copying any data
- Slices are super fast
- Because Slices point into the original collection, normal Borrow Checker rules apply



```
fn modify() {
    let mut original: String = "Hello World!".to_string();
    let slice: &str = &original[..5];
    original.push(ch: '!');
    println!("slice = {}", slice);
}
```

In our original naive String implementation, this would‘ve been allowed
→ The words were separate from the original text

3. Next time

- Smart Pointers
  - Rc<T>
  - RefCell<T>
- Declarative Macros