RUSTikales Rust for beginners

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Loops

1. Recap

1. Recap

- Integers in Rust: i8, u8, i16, u16, ... , i128, u128

1. Recap

- Integers in Rust: i8, u8, i16, u16, ... , i128, u128
- let <name> = ...; to declare an immutable variable
- let mut <name> = ...; to declare a mutable variable

1. Recap

- Integers in Rust: i8, u8, i16, u16, ... , i128, u128
- let <name> = ...; to declare an immutable variable
- let mut <name> = ...; to declare a mutable variable
- let var: [type; size] = ...; to declare an array with the given type and size
- let var: Vec<type> = ...; to declare a Vector with the given type

1. Recap

- Integers in Rust: i8, u8, i16, u16, ... , i128, u128
- let <name> = ...; to declare an immutable variable
- let mut <name> = ...; to declare a mutable variable
- let var: [type; size] = ...; to declare an array with the given type and size
- let var: Vec<type> = ...; to declare a Vector with the given type
- Elements in Arrays and Vectors always have the same type

1. Recap

- Integers in Rust: i8, u8, i16, u16, ... , i128, u128
- let <name> = ...; to declare an immutable variable
- let mut <name> = ...; to declare a mutable variable
- let var: [type; size] = ...; to declare an array with the given type and size
- let var: Vec<type> = ...; to declare a Vector with the given type
- Elements in Arrays and Vectors always have the same type
- Arrays have a fixed size, Vectors are resizable

1. Recap

- var[index] to access the element at a given index
  - var[0] = 5; sets the element at index 0 to 5
  - let x = var[1]; gets the element at index 1 and stores it in x

1. Recap

- var[index] to access the element at a given index
- Arrays and Vectors are zero-indexed
  - For size N, indices 0 to N-1 are defined, anything else results in Out-Of-Bounds panics

1. Recap

- var[index] to access the element at a given index
- Arrays and Vectors are zero-indexed
- vec![] is a macro to easily create Vectors

1. Recap

- Important to know:

1. Recap

- Important to know:
  - Many helper methods exist for Arrays and Vectors
    - name.reverse() reverses the order of elements in name, and stores the result in name again
    - vector.pop() removes the last element of a Vector
    - name.len() returns the number of elements in your Array/Vector

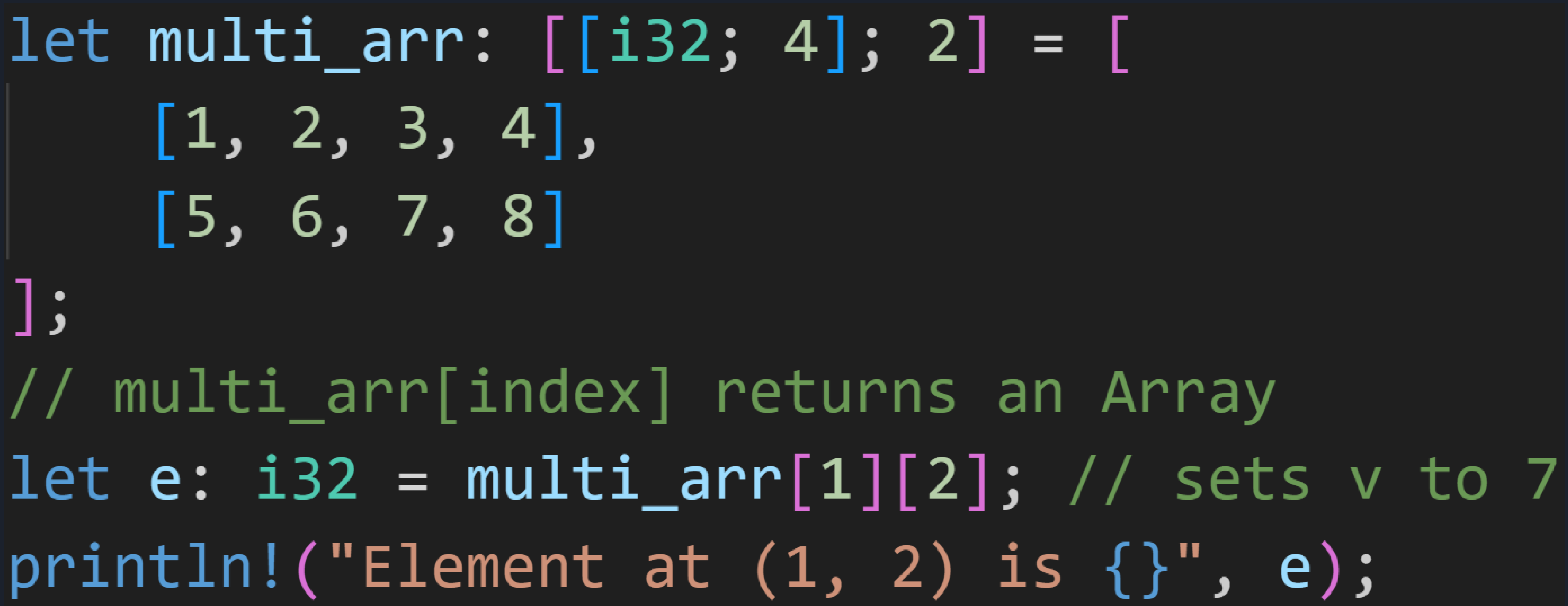
1. Recap

- Important to know:
  - Many helper methods exist for Arrays and Vectors
  - Array Type Declaration itself is a Type

1. Recap

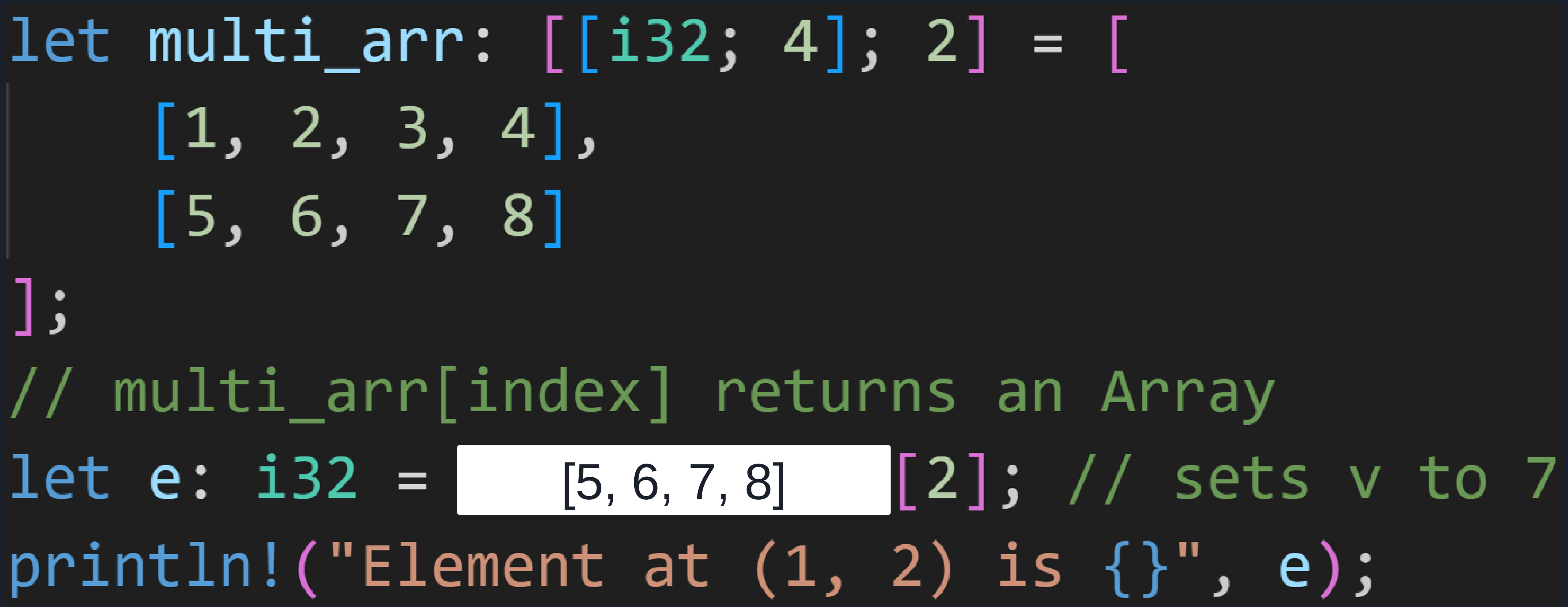
- Important to know:
  - Many helper methods exist for Arrays and Vectors
  - Array Type Declaration itself is a Type
    - [i32; 5] is an Array of i32
    - [[i32; 5]; 5] is an Array of Arrays of i32
    - Vec<[i32; 5]> is a Vector of Arrays of i32
    - [Vec<i32>; 5] is an Array of Vectors of i32

1. Recap



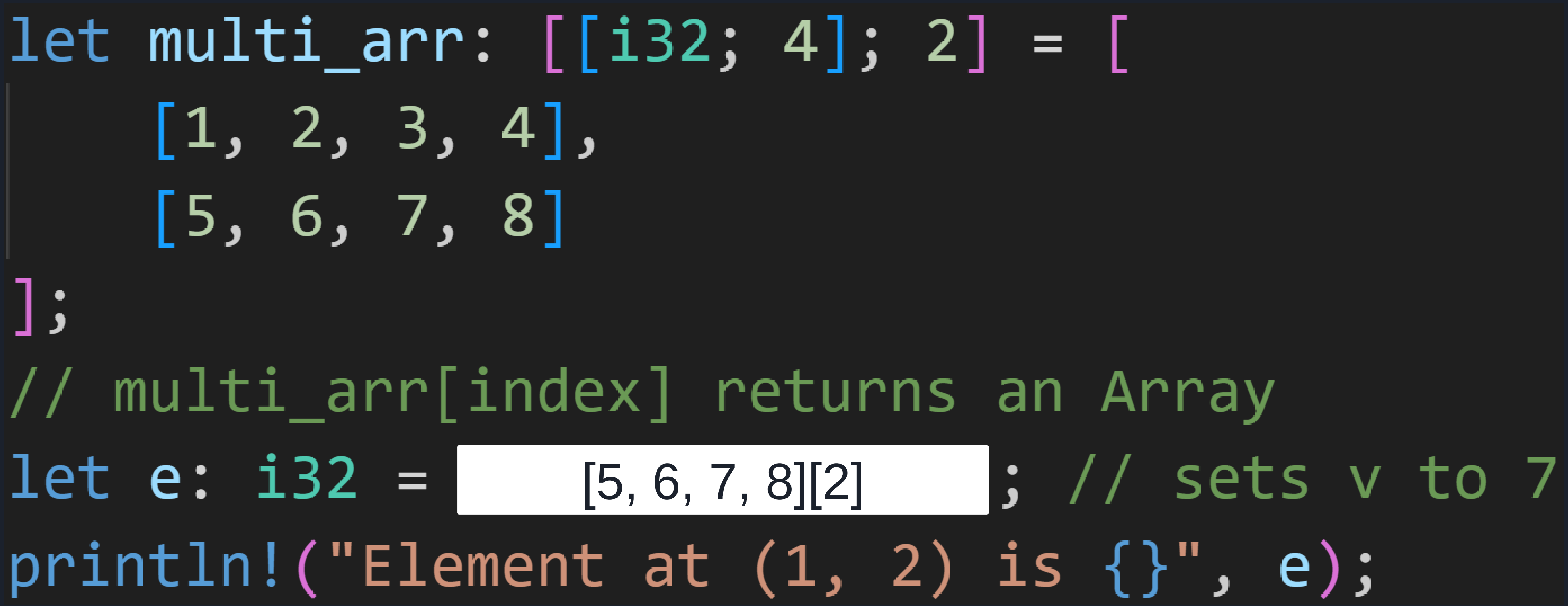
```rust
let multi_arr: [[i32; 4]; 2] = [
    [1, 2, 3, 4],
    [5, 6, 7, 8]
];
// multi_arr[index] returns an Array
let e: i32 = multi_arr[1][2]; // sets v to 7
println!("Element at (1, 2) is {}", e);
```

1. Recap



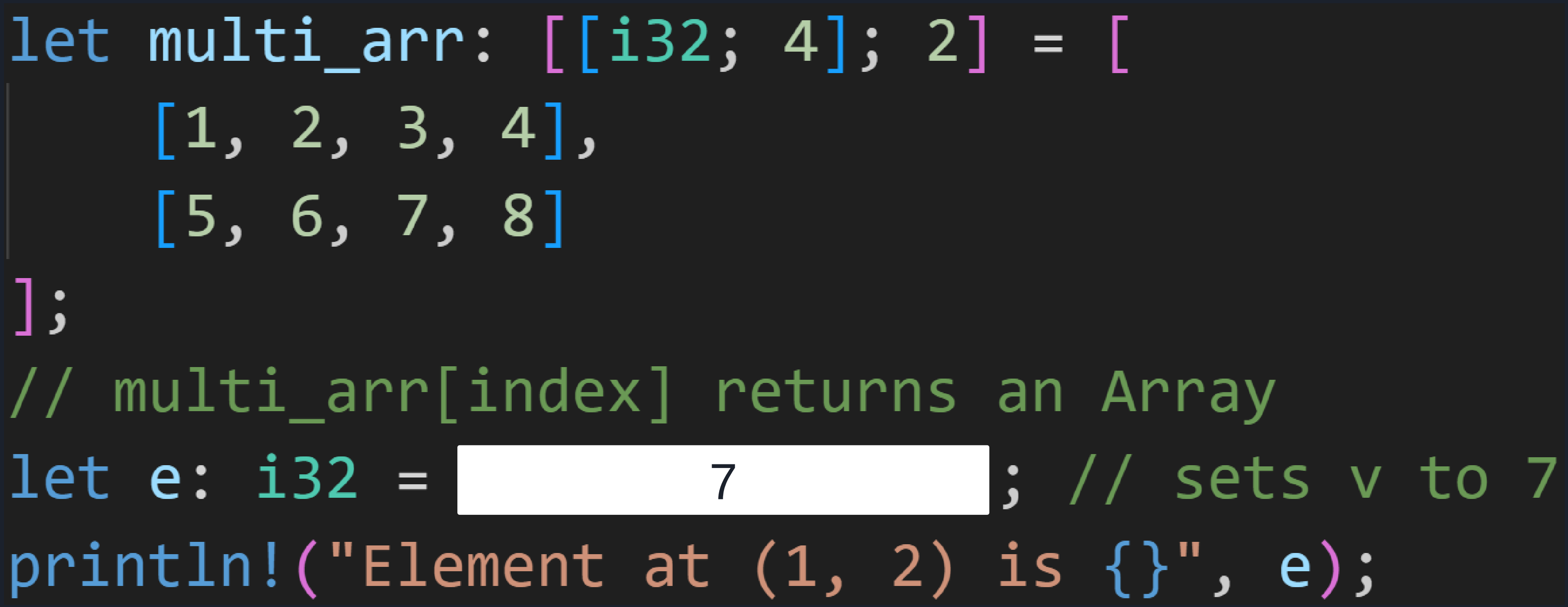
```rust
let multi_arr: [[i32; 4]; 2] = [
    [1, 2, 3, 4],
    [5, 6, 7, 8]
];
// multi_arr[index] returns an Array
let e: i32 = [5, 6, 7, 8][2]; // sets v to 7
println!("Element at (1, 2) is {}", e);
```

1. Recap



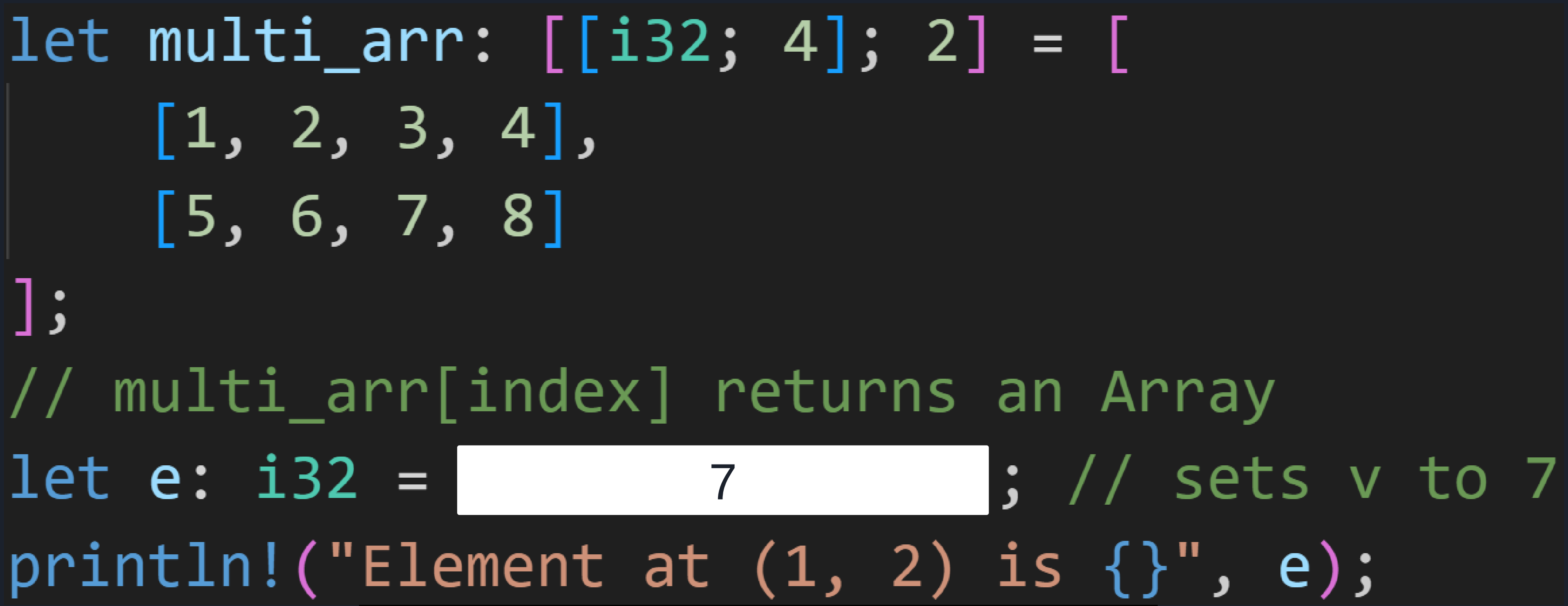
```
let multi_arr: [[i32; 4]; 2] = [
    [1, 2, 3, 4],
    [5, 6, 7, 8]
];
// multi_arr[index] returns an Array
let e: i32 = [5, 6, 7, 8][2]; // sets v to 7
println!("Element at (1, 2) is {}", e);
```

1. Recap

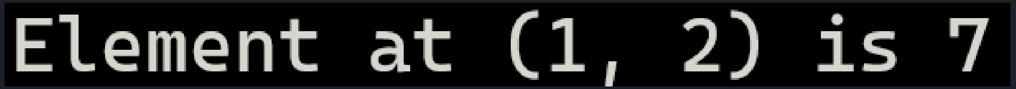


```
let multi_arr: [[i32; 4]; 2] = [
    [1, 2, 3, 4],
    [5, 6, 7, 8]
];
// multi_arr[index] returns an Array
let e: i32 = 7; // sets v to 7
println!("Element at (1, 2) is {}", e);
```

1. Recap



```rust
let multi_arr: [[i32; 4]; 2] = [
    [1, 2, 3, 4],
    [5, 6, 7, 8]
];
// multi_arr[index] returns an Array
let e: i32 = 7; // sets v to 7
println!("Element at (1, 2) is {}", e);
```



```
Element at (1, 2) is 7
```

2. Loops

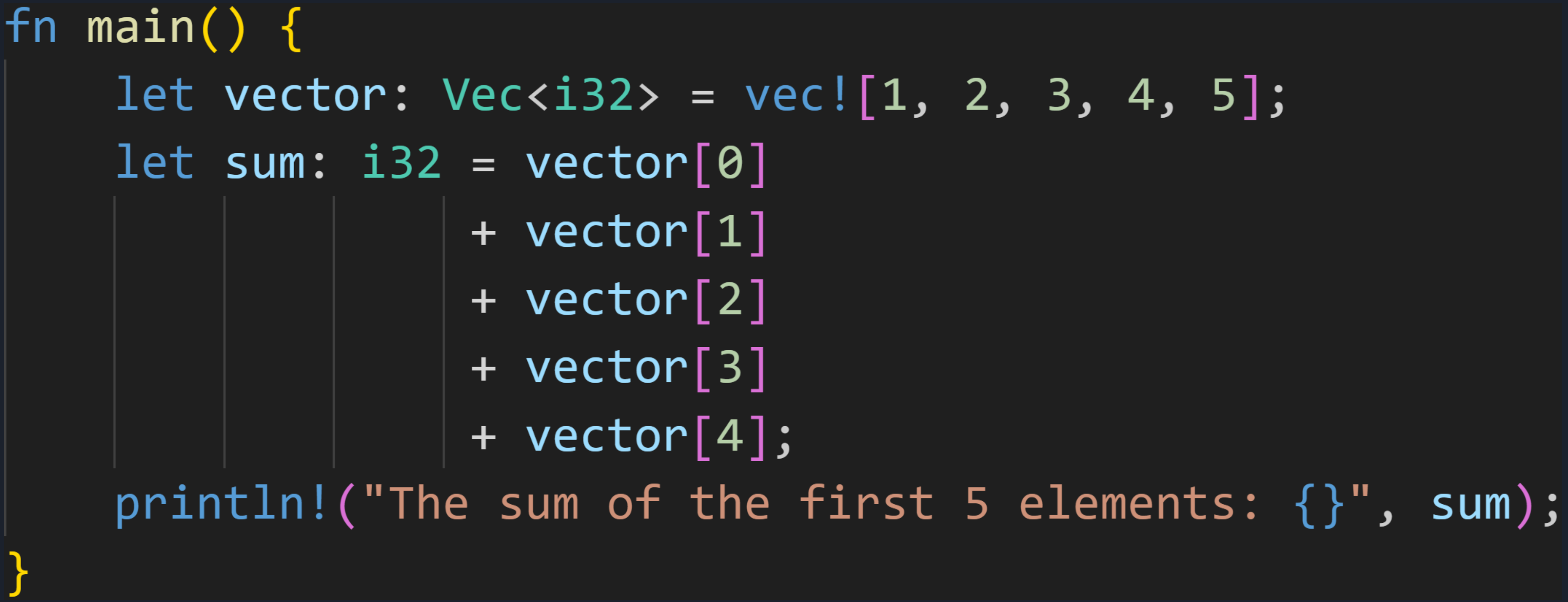
- Note: I will use Arrays and Vectors interchangeably today

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size

2. Loops

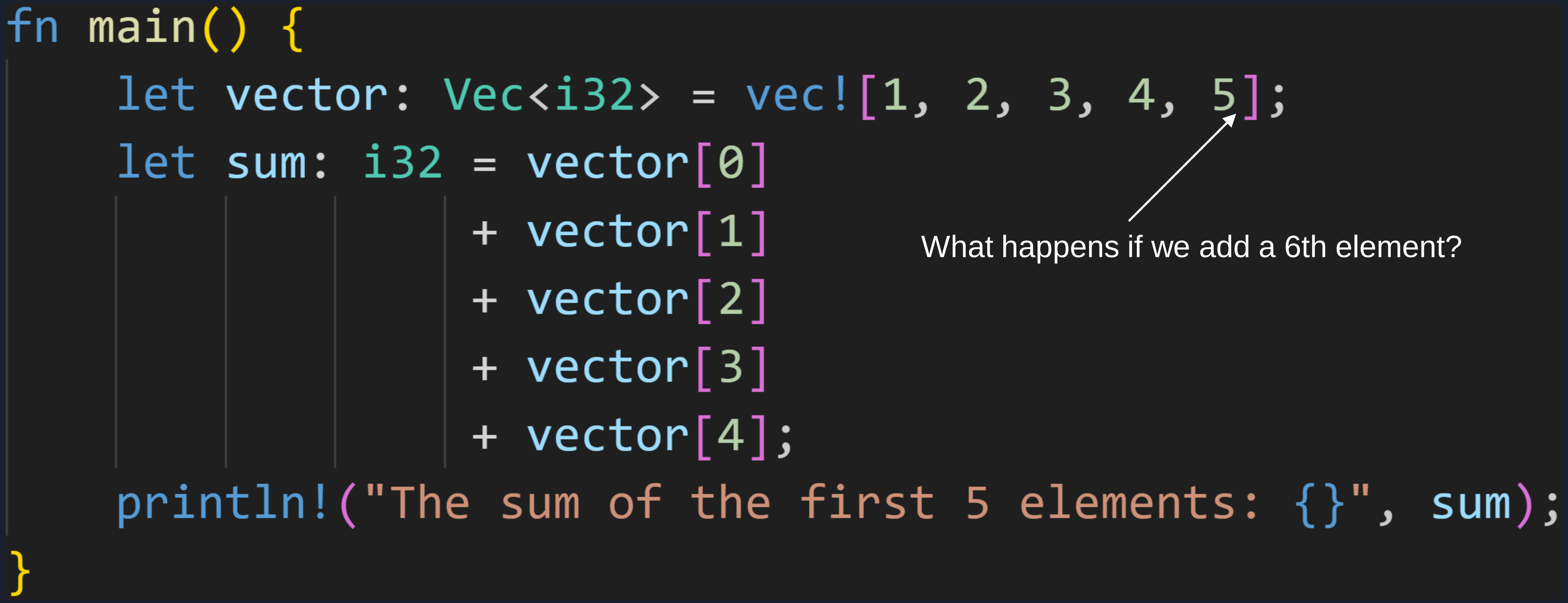
- Writing code that fits a single Vector size is easy, but breaks once you modify its size



```rust
fn main() {
    let vector: Vec<i32> = vec![1, 2, 3, 4, 5];
    let sum: i32 = vector[0]
                 + vector[1]
                 + vector[2]
                 + vector[3]
                 + vector[4];
    println!("The sum of the first 5 elements: {}", sum);
}
```

2. Loops

– Writing code that fits a single Vector size is easy, but breaks once you modify its size

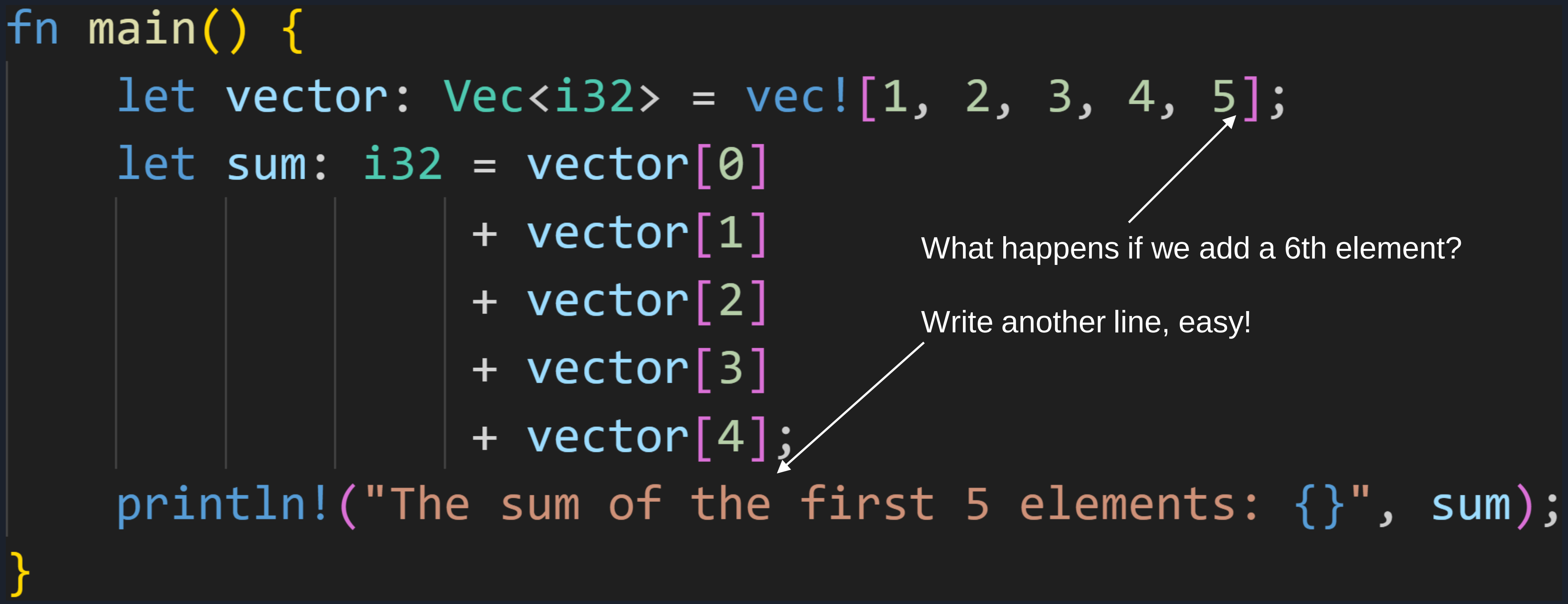


```rust
fn main() {
    let vector: Vec<i32> = vec![1, 2, 3, 4, 5];
    let sum: i32 = vector[0]
                 + vector[1]
                 + vector[2]
                 + vector[3]
                 + vector[4];
    println!("The sum of the first 5 elements: {}", sum);
}
```

What happens if we add a 6th element?

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size



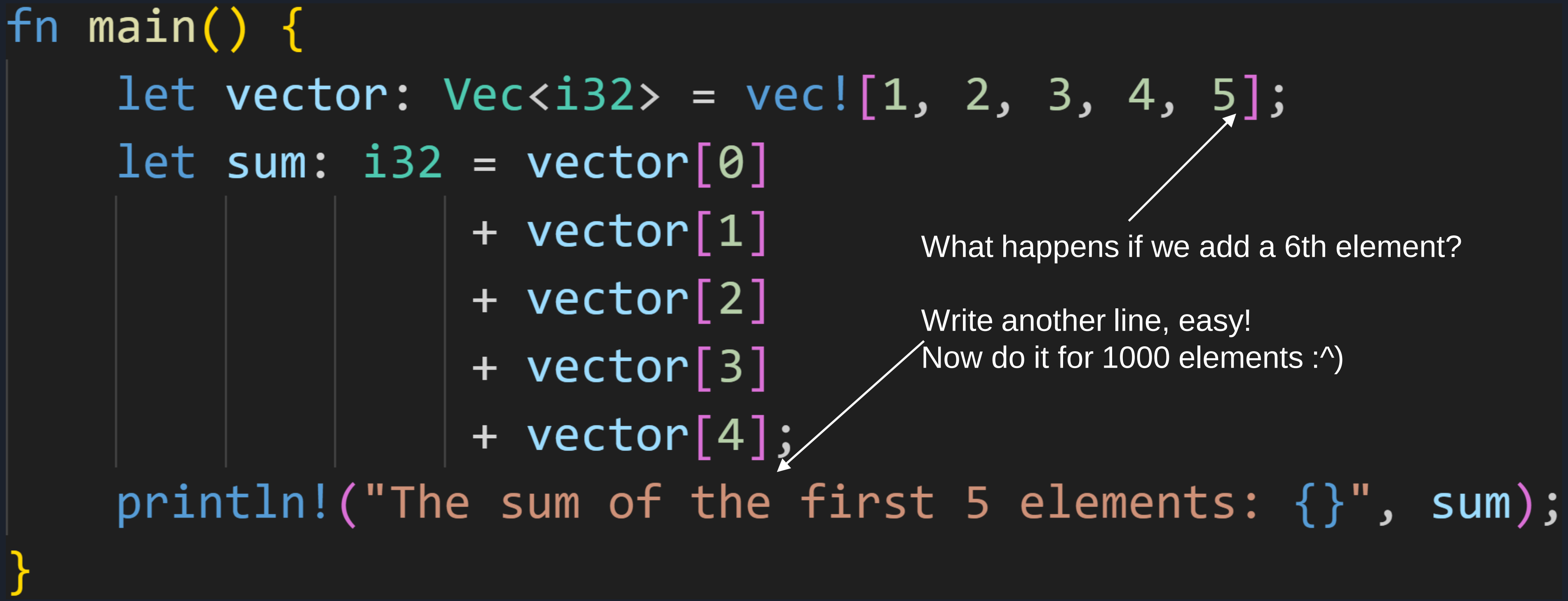
```rust
fn main() {
    let vector: Vec<i32> = vec![1, 2, 3, 4, 5];
    let sum: i32 = vector[0]
                 + vector[1]
                 + vector[2]
                 + vector[3]
                 + vector[4];
    println!("The sum of the first 5 elements: {}", sum);
}
```

What happens if we add a 6th element?

Write another line, easy!

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size



```
fn main() {
    let vector: Vec<i32> = vec![1, 2, 3, 4, 5];
    let sum: i32 = vector[0]
                 + vector[1]
                 + vector[2]
                 + vector[3]
                 + vector[4];
    println!("The sum of the first 5 elements: {}", sum);
}
```

What happens if we add a 6th element?

Write another line, easy!
Now do it for 1000 elements :^)

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size
- Sometimes you don‘t know how big a Vector is, but want to do something for every element in it

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size
- Sometimes you don‘t know how big a Vector is, but want to do something for every element in it
- Sometimes you want to go over a range of numbers, and do something for every number you see

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size
- Sometimes you don‘t know how big a Vector is, but want to do something for every element in it
- Sometimes you want to go over a range of numbers, and do something for every number you see
  - For every number below 1000, print the primes :)

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size
- Sometimes you don‘t know how big a Vector is, but want to do something for every element in it
- Sometimes you want to go over a range of numbers, and do something for every number you see
  - For every number below 1000, print the primes :)
  - Play a game of FizzBuzz

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size
- Sometimes you don‘t know how big a Vector is, but want to do something for every element in it
- Sometimes you want to go over a range of numbers, and do something for every number you see
  - For every number below 1000, print the primes :)
  - Play a game of FizzBuzz
    - For every number below 100:
      - if number is divisible by 3, print „Fizz“
      - if number is divisible by 5, print „Buzz“
      - if number is divisible by 15, print „FizzBuzz“
      - else print the number

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size
- Sometimes you don‘t know how big a Vector is, but want to do something for every element in it
- Sometimes you want to go over a range of numbers, and do something for every number you see
  - For every number below 1000, print the primes :)
  - Play a game of FizzBuzz
    - For every number below 100:
      - if number is divisible by 3, print „Fizz“
      - if number is divisible by 5, print „Buzz“
      - if number is divisible by 15, print „FizzBuzz“
      - else print the number

2. Loops

- Writing code that fits a single Vector size is easy, but breaks once you modify its size
- Sometimes you don‘t know how big a Vector is, but want to do something for every element in it
- Sometimes you want to go over a range of numbers, and do something for every number you see
- Need to do any of that? Loops are your friend!

2. Loops

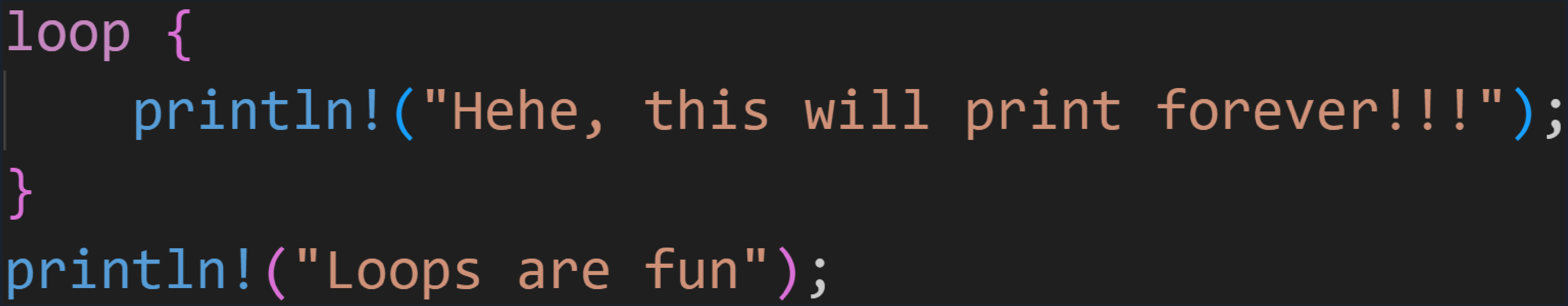
- A Loop does exactly what you think it does → It loops a piece of code

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops
  - loop
  - while
  - for

2. Loops

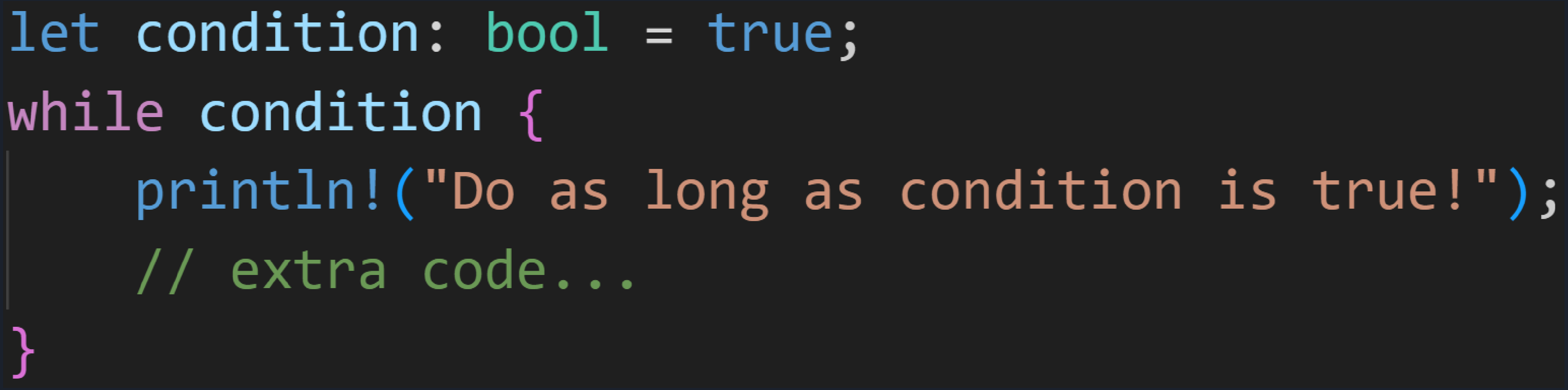
- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops



```rust
loop {
    println!("Hehe, this will print forever!!!");
}
println!("Loops are fun");
```

2. Loops

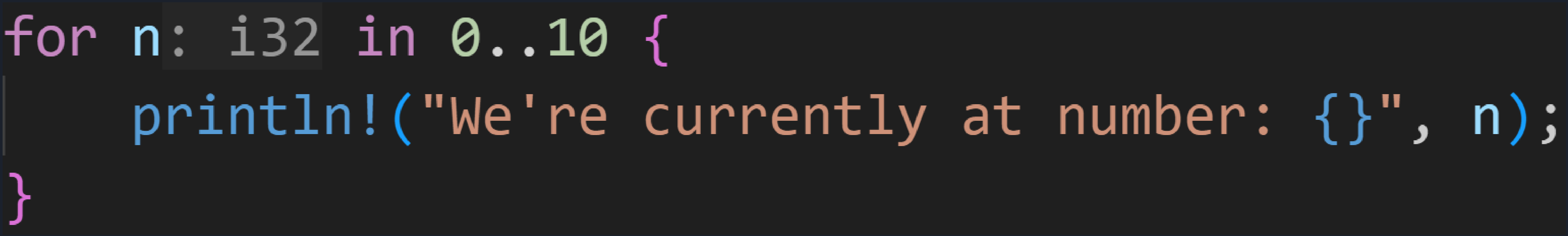
- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops



```rust
let condition: bool = true;
while condition {
    println!("Do as long as condition is true!");
    // extra code...
}
```

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops



```rust
for n: i32 in 0..10 {
    println!("We're currently at number: {}", n);
}
```

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops
- Each type has their own use cases

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops
- Each type has their own use cases
  - Use loop if you know that your loop never stops

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops
- Each type has their own use cases
  - Use loop if you know that your loop never stops
    - A server will never stop listening to requests

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops
- Each type has their own use cases
  - Use loop if you know that your loop never stops
  - Use while if you want to loop based on a condition

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops
- Each type has their own use cases
  - Use loop if you know that your loop never stops
  - Use while if you want to loop based on a condition
    - While there are elements in this set, do X

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops
- Each type has their own use cases
  - Use loop if you know that your loop never stops
  - Use while if you want to loop based on a condition
  - Use for if you want to iterate* over a collection

2. Loops

- A Loop does exactly what you think it does → It loops a piece of code
- There are many types of loops
- Each type has their own use cases
  - Use loop if you know that your loop never stops
  - Use while if you want to loop based on a condition
  - Use for if you want to iterate* over a collection
    - For every number in a range
    - For every element in a Vector
    - For every dog in my local park

2. Loops - loop

- loop is the simplest form of loops in Rust

2. Loops - loop

- loop is the simplest form of loops in Rust
- It‘s also pretty dumb, it can not stop, ever*

2. Loops - loop

- loop is the simplest form of loops in Rust
- It‘s also pretty dumb, it can not stop, ever*
- Use cases include:

2. Loops - loop

- loop is the simplest form of loops in Rust
- It‘s also pretty dumb, it can not stop, ever*
- Use cases include:
  - You don‘t want your code to ever stop
    - Servers listen to requests 24/7

2. Loops - loop

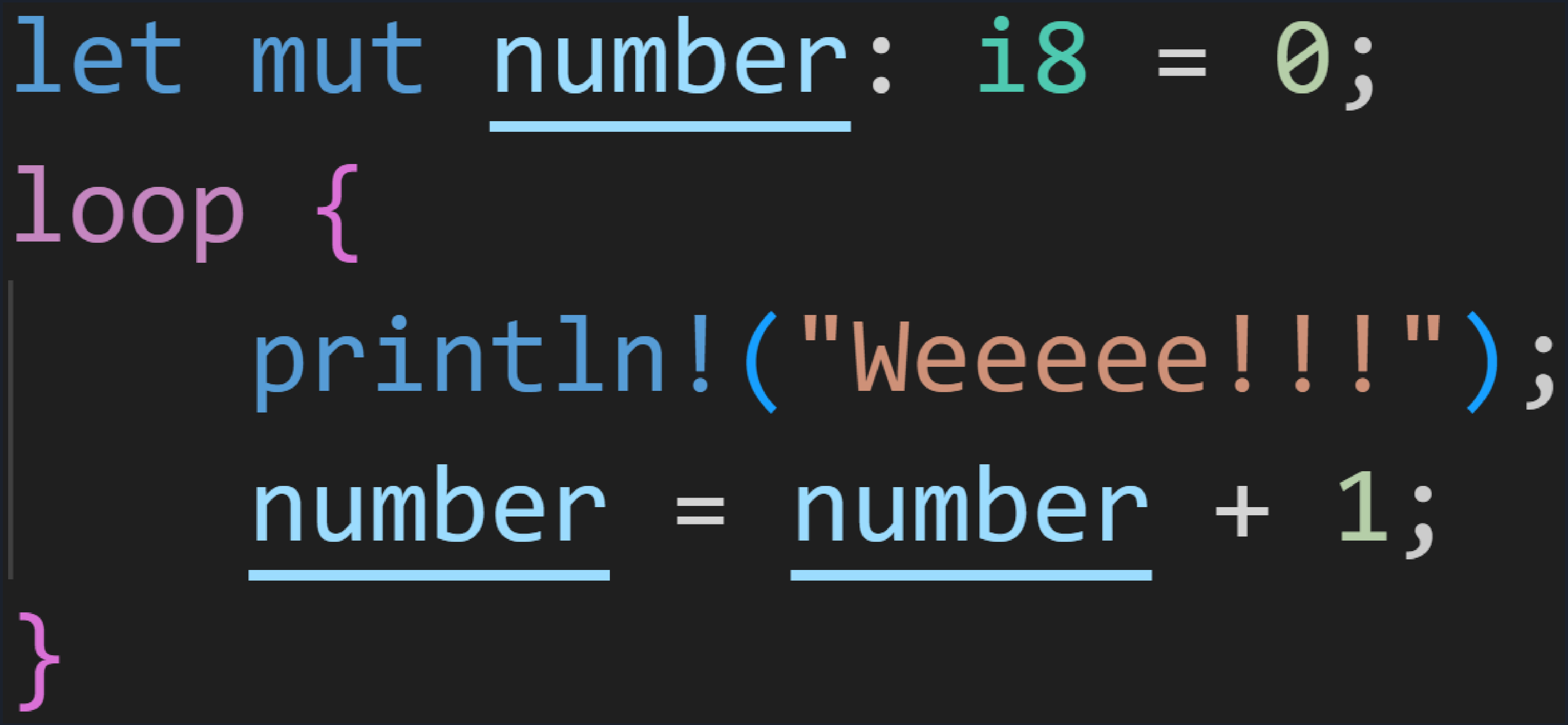
- loop is the simplest form of loops in Rust
- It‘s also pretty dumb, it can not stop, ever*
- Use cases include:
  - You don‘t want your code to ever stop
  - Your while-condition would be too complex to put in one expression
    - Instead, you can split it into sub-expressions and test each of them separately

2. Loops - loop

- loop is the simplest form of loops in Rust
- It‘s also pretty dumb, it can not stop, ever*
- Use cases include:
  - You don‘t want your code to ever stop
  - Your while-condition would be too complex to put in one expression
  - Stress Testing your CPU (simplest way of getting 100% usage)

2. Loops - loop

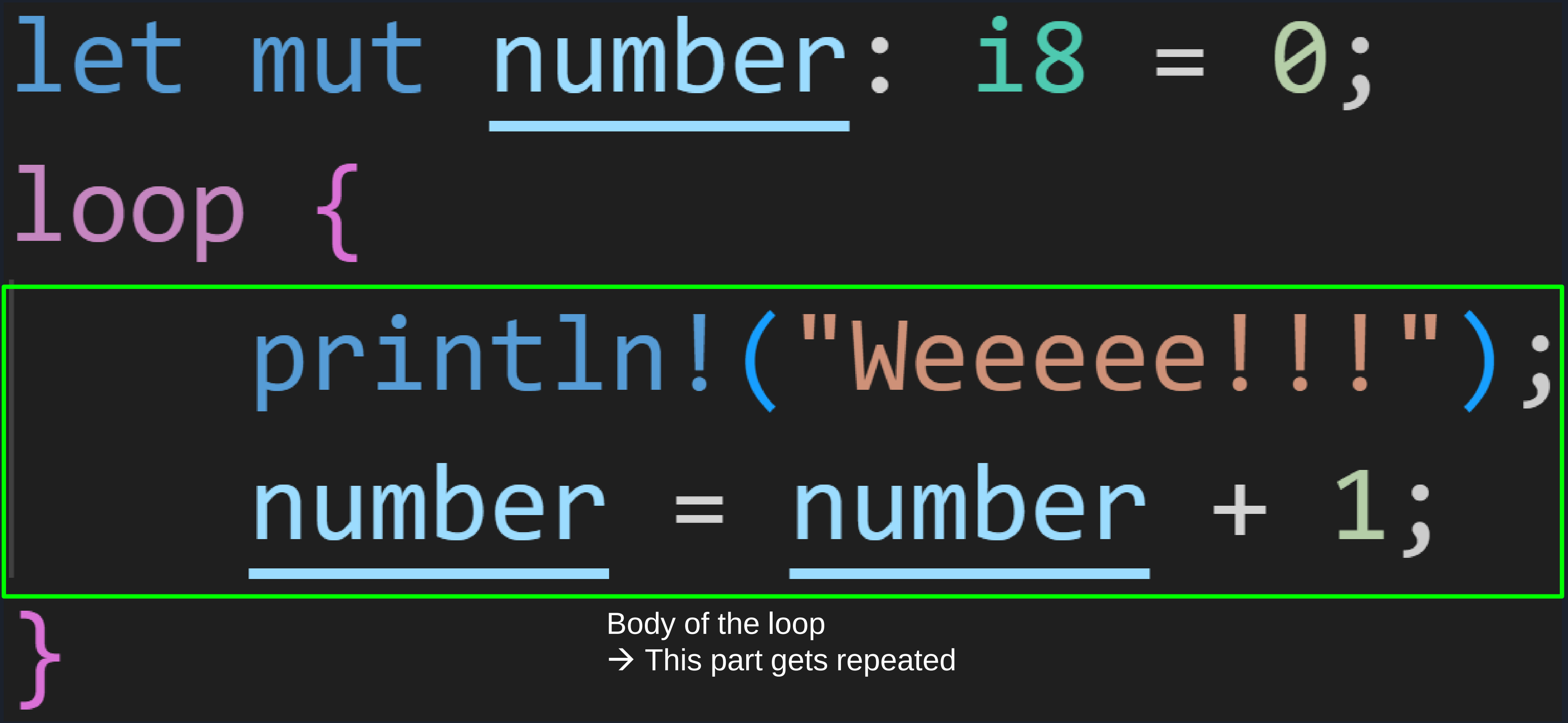
a loop loop may look like this



```rust
let mut number: i8 = 0;
loop {
    println!("Weeeee!!!");
    number = number + 1;
}
```

2. Loops - loop

a loop loop may look like this

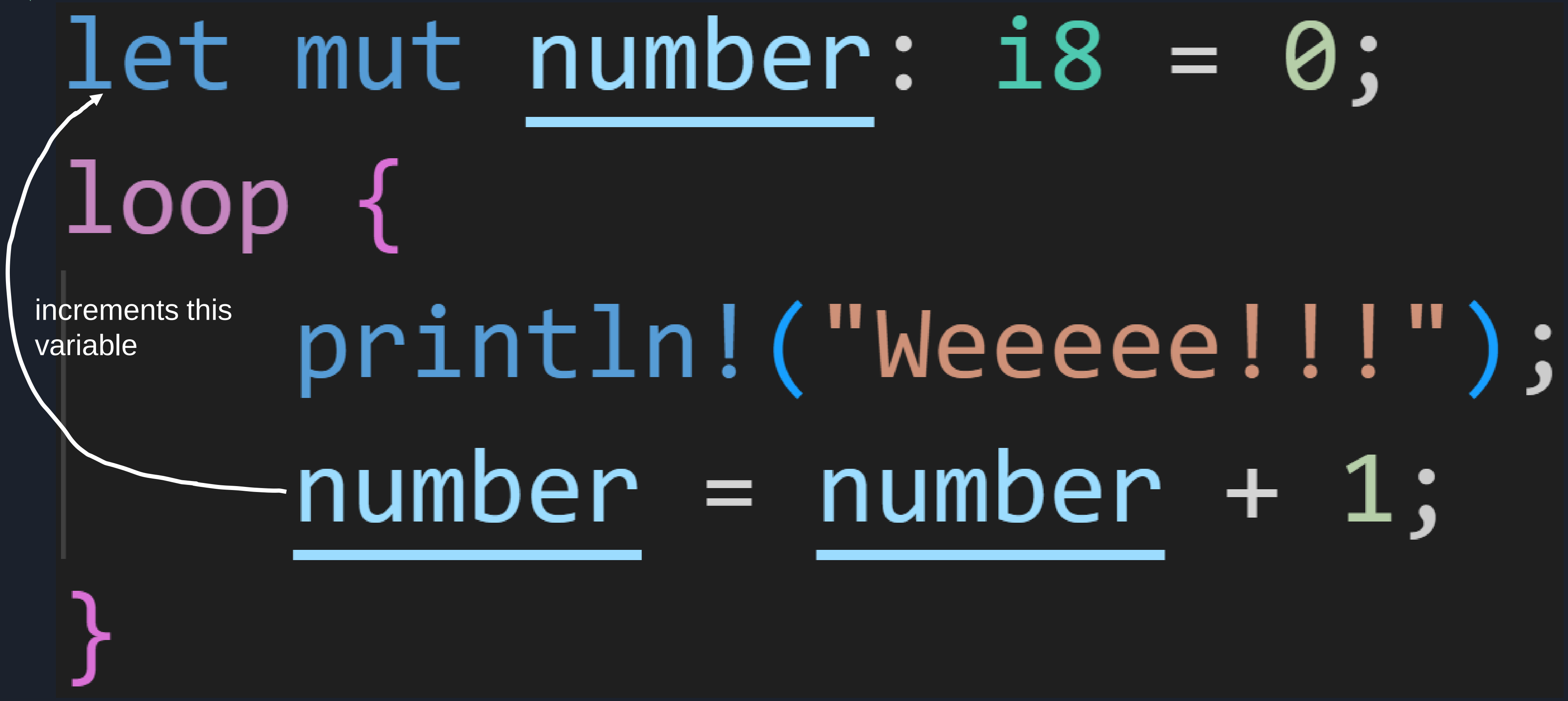


```rust
let mut number: i8 = 0;
loop {
    println!("Weeeee!!!");
    number = number + 1;
}
```

Body of the loop
→ This part gets repeated

2. Loops - loop

a loop loop may look like this

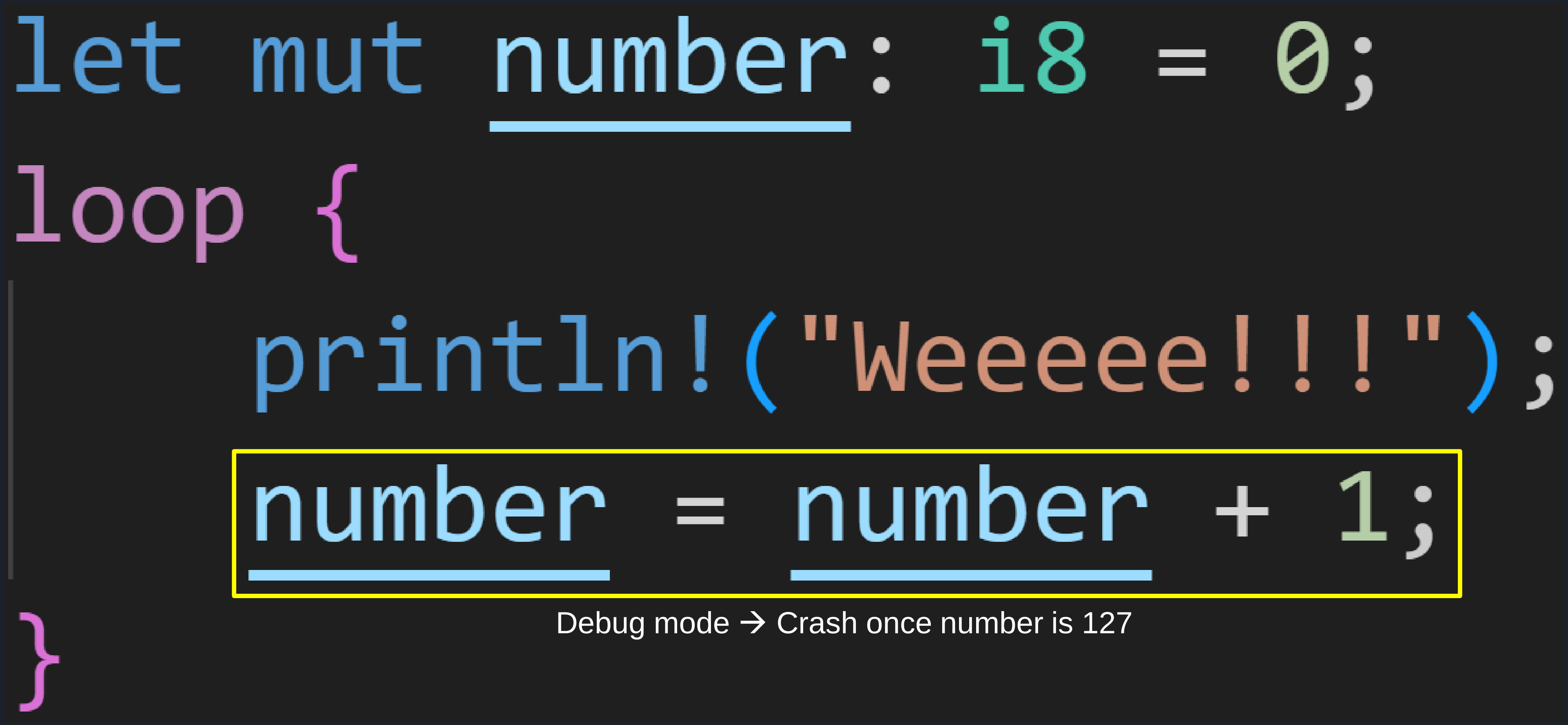


```
let mut number: i8 = 0;
loop {
    println!("Weeeee!!!");
    number = number + 1;
}
```

increments this variable

2. Loops - loop

a loop loop may look like this

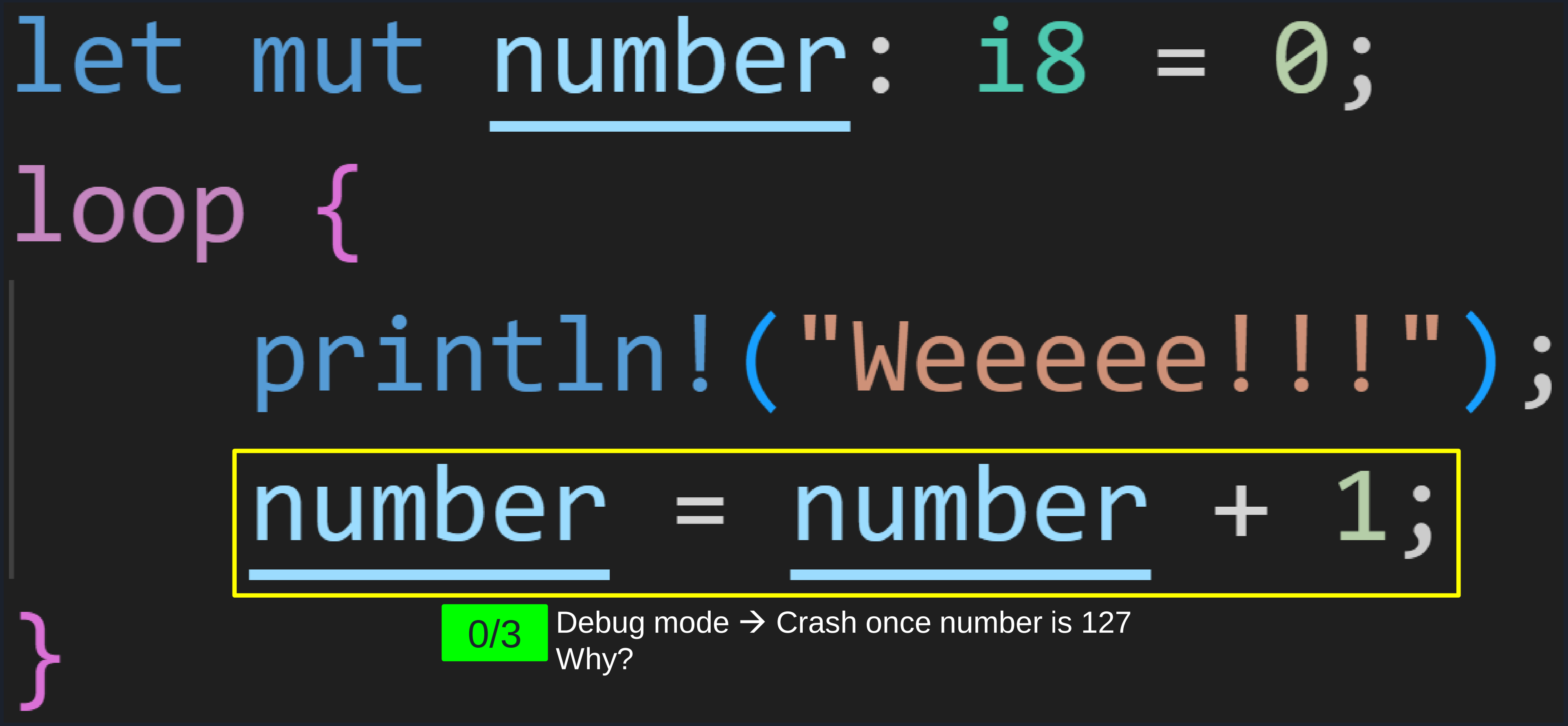


```
let mut number: i8 = 0;
loop {
    println!("Weeeee!!!");
    number = number + 1;
}
```

Debug mode → Crash once number is 127

2. Loops - loop

a loop loop may look like this

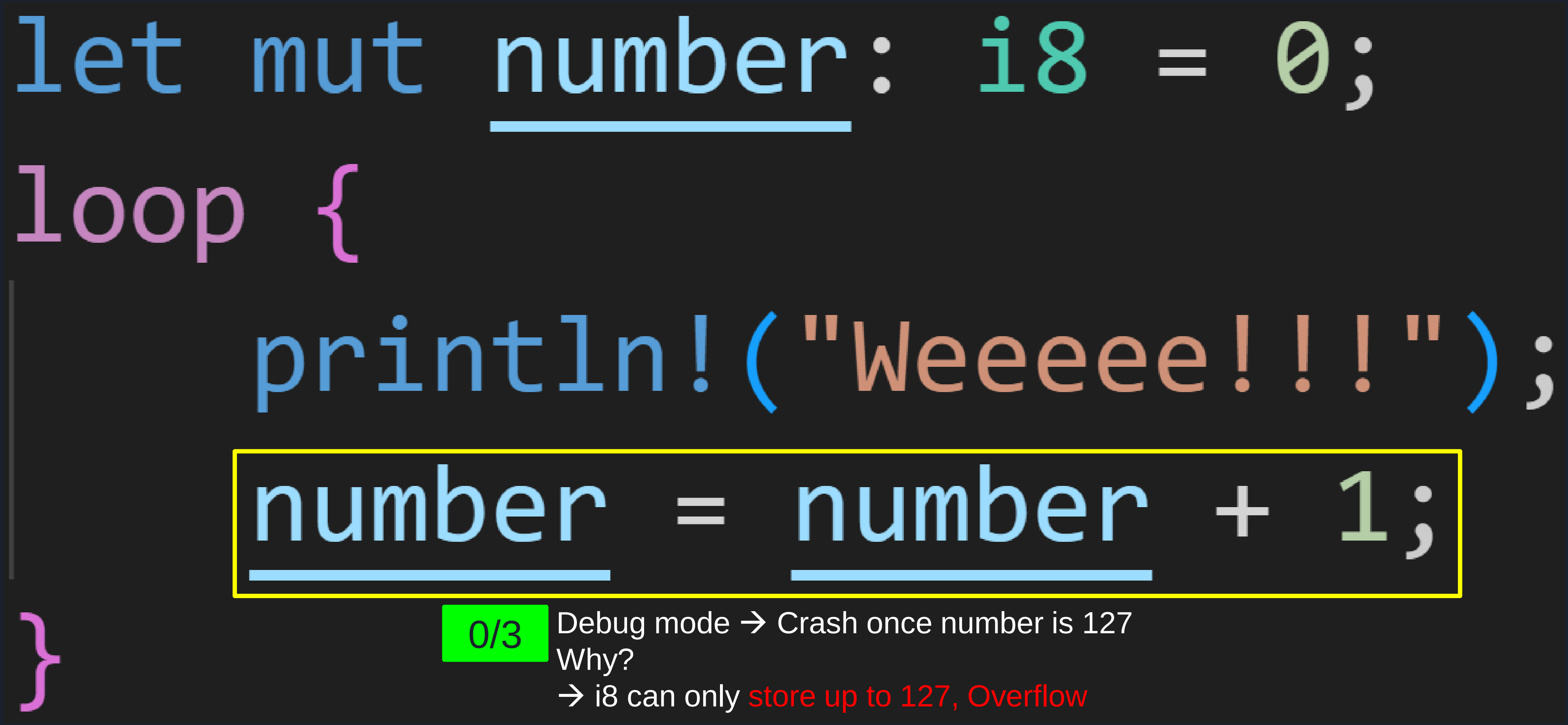


```rust
let mut number: i8 = 0;
loop {
    println!("Weeeee!!!");
    number = number + 1;
}
```

0/3 Debug mode → Crash once number is 127
Why?

2. Loops - loop

a loop loop may look like this



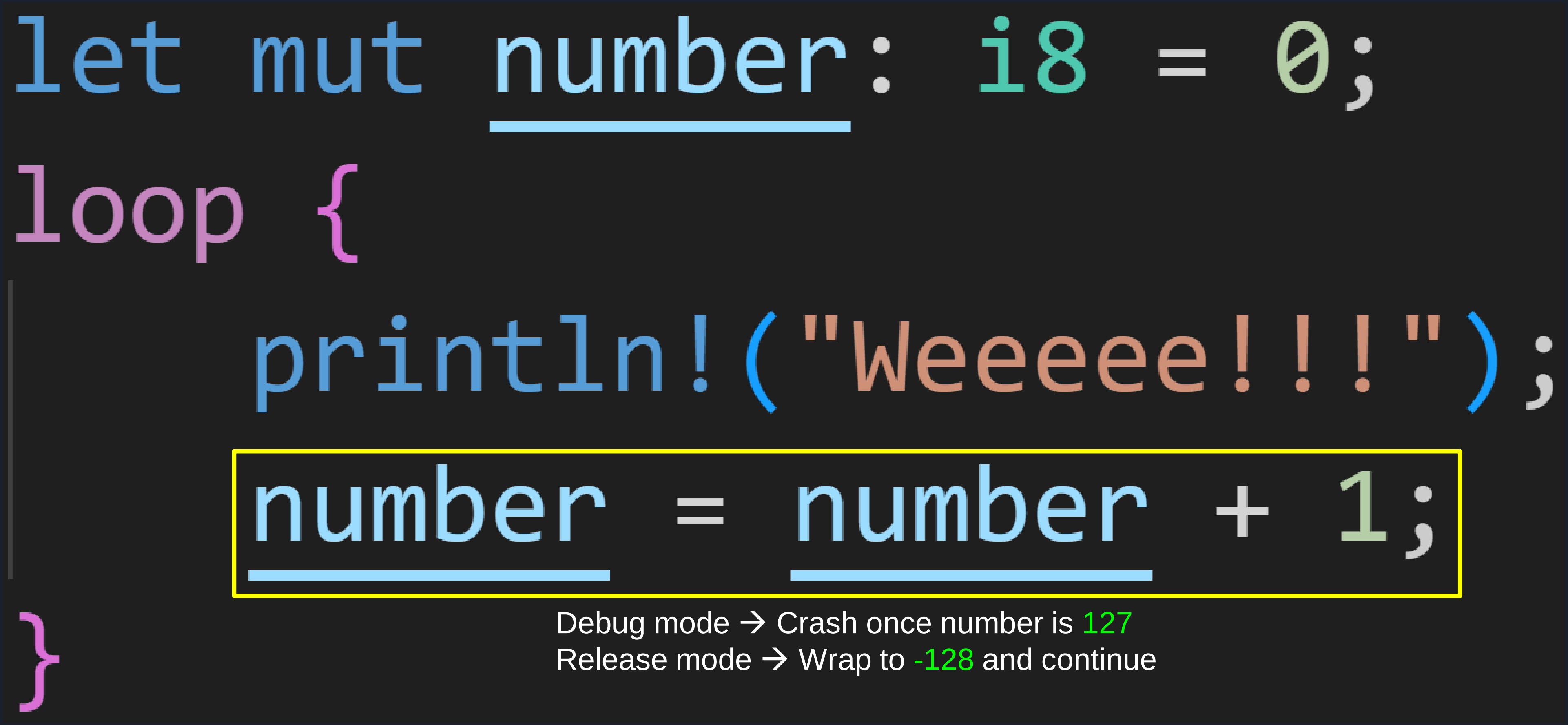
```rust
let mut number: i8 = 0;
loop {
    println!("Weeeee!!!");
    number = number + 1;
}
```

0/3

Debug mode → Crash once number is 127
Why?
→ i8 can only store up to 127, Overflow

2. Loops - loop

a loop loop may look like this



```
let mut number: i8 = 0;
loop {
    println!("Weeeee!!!");
    number = number + 1;
}
```

Debug mode → Crash once number is 127
Release mode → Wrap to -128 and continue

2. Loops - while

- while is the advanced version of loop

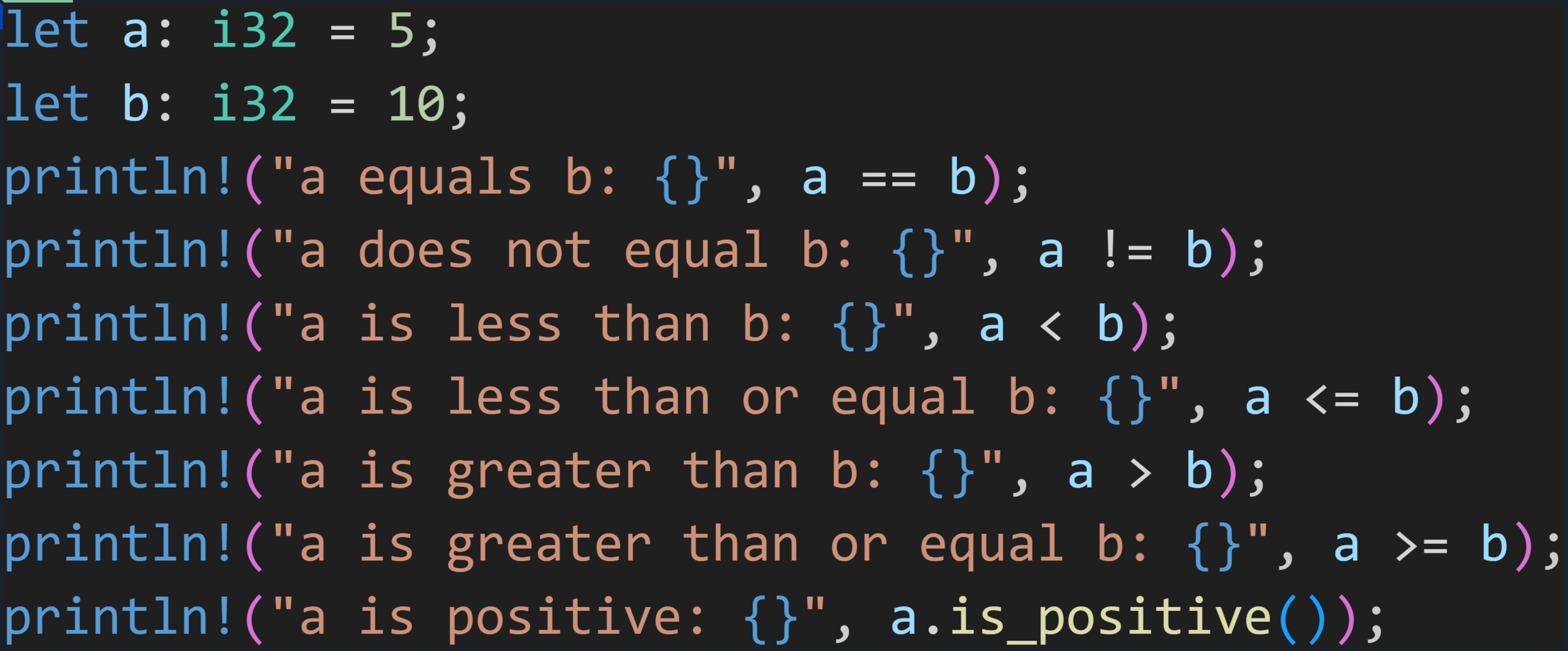
2. Loops - while

- while is the advanced version of loop
- whereas loop loops forever, while stops once a condition is no longer satisfied

Intermission - Conditions

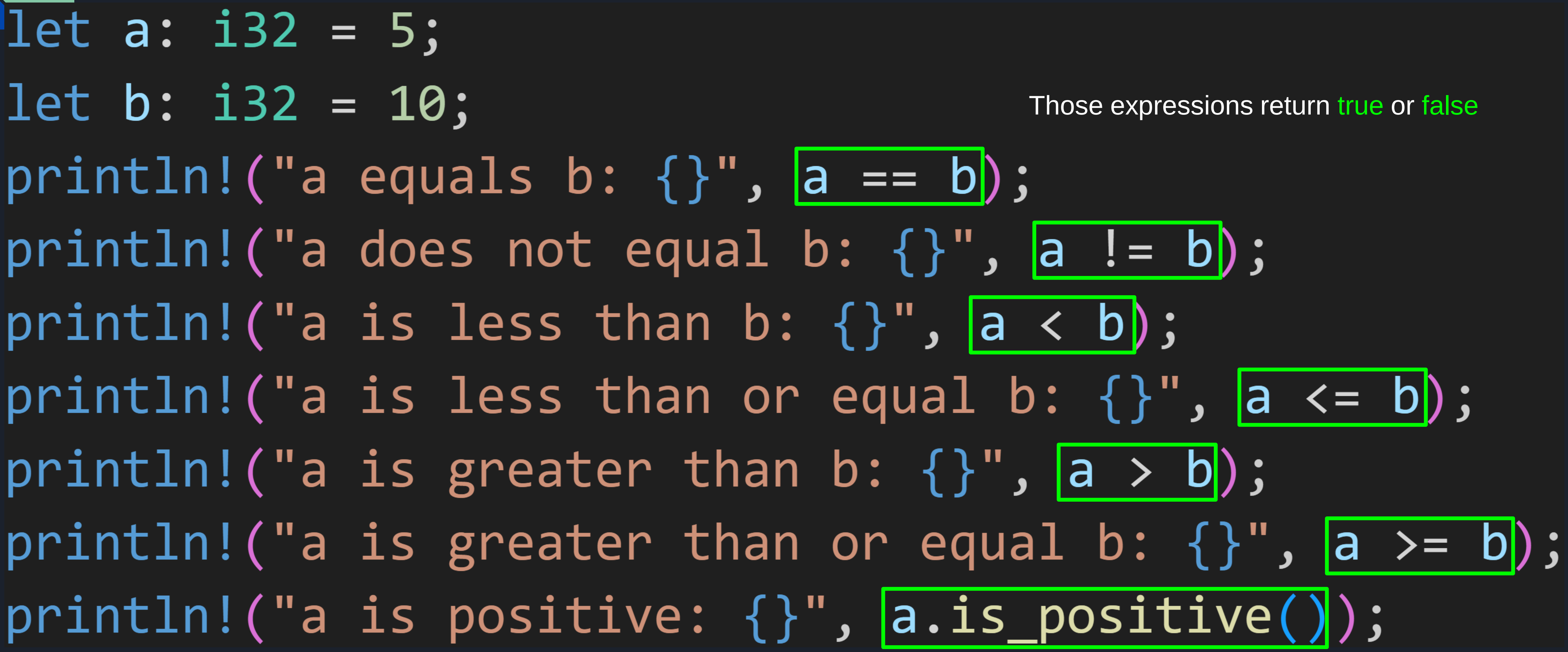
- Conditions are expressions which return a boolean (either true or false)

Intermission - Conditions



```
let a: i32 = 5;
let b: i32 = 10;
println!("a equals b: {}", a == b);
println!("a does not equal b: {}", a != b);
println!("a is less than b: {}", a < b);
println!("a is less than or equal b: {}", a <= b);
println!("a is greater than b: {}", a > b);
println!("a is greater than or equal b: {}", a >= b);
println!("a is positive: {}", a.is_positive());
```

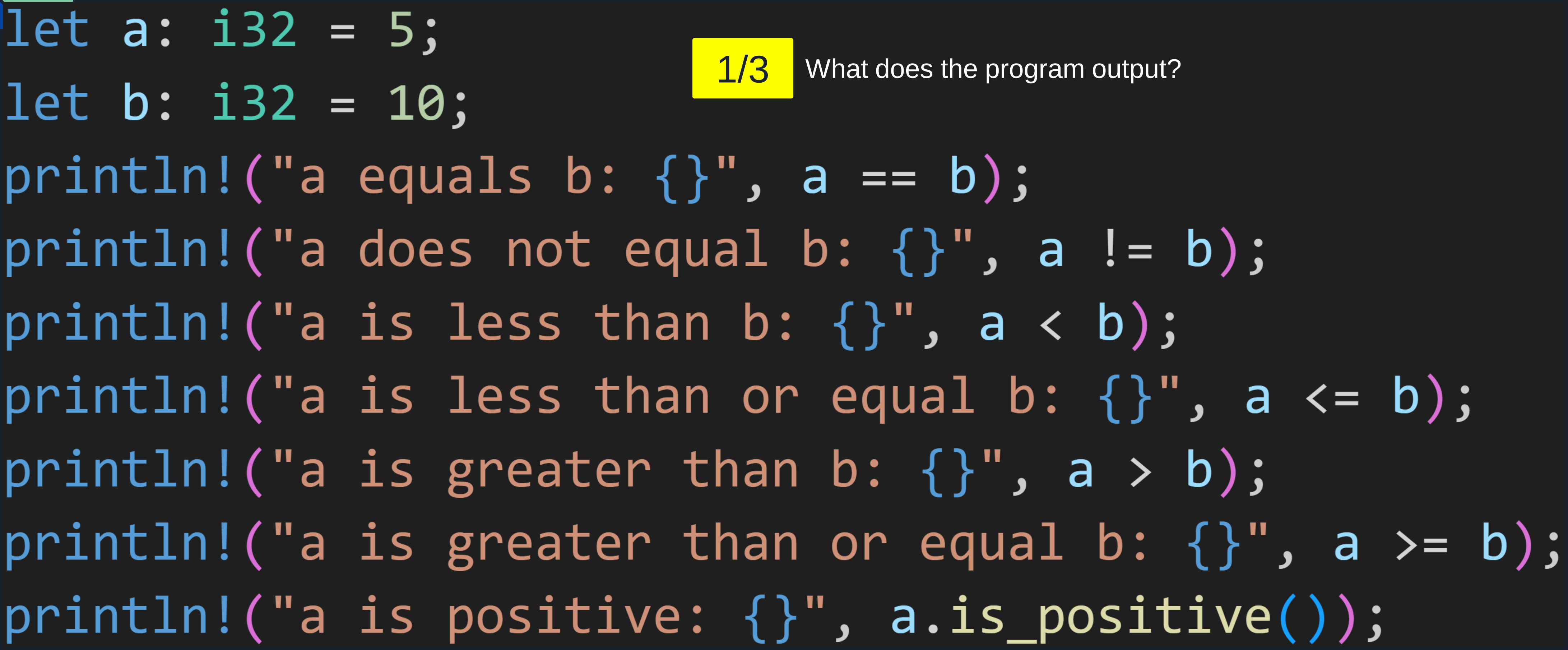
Intermission - Conditions



```
let a: i32 = 5;
let b: i32 = 10;
println!("a equals b: {}", a == b);
println!("a does not equal b: {}", a != b);
println!("a is less than b: {}", a < b);
println!("a is less than or equal b: {}", a <= b);
println!("a is greater than b: {}", a > b);
println!("a is greater than or equal b: {}", a >= b);
println!("a is positive: {}", a.is_positive());
```

Those expressions return true or false

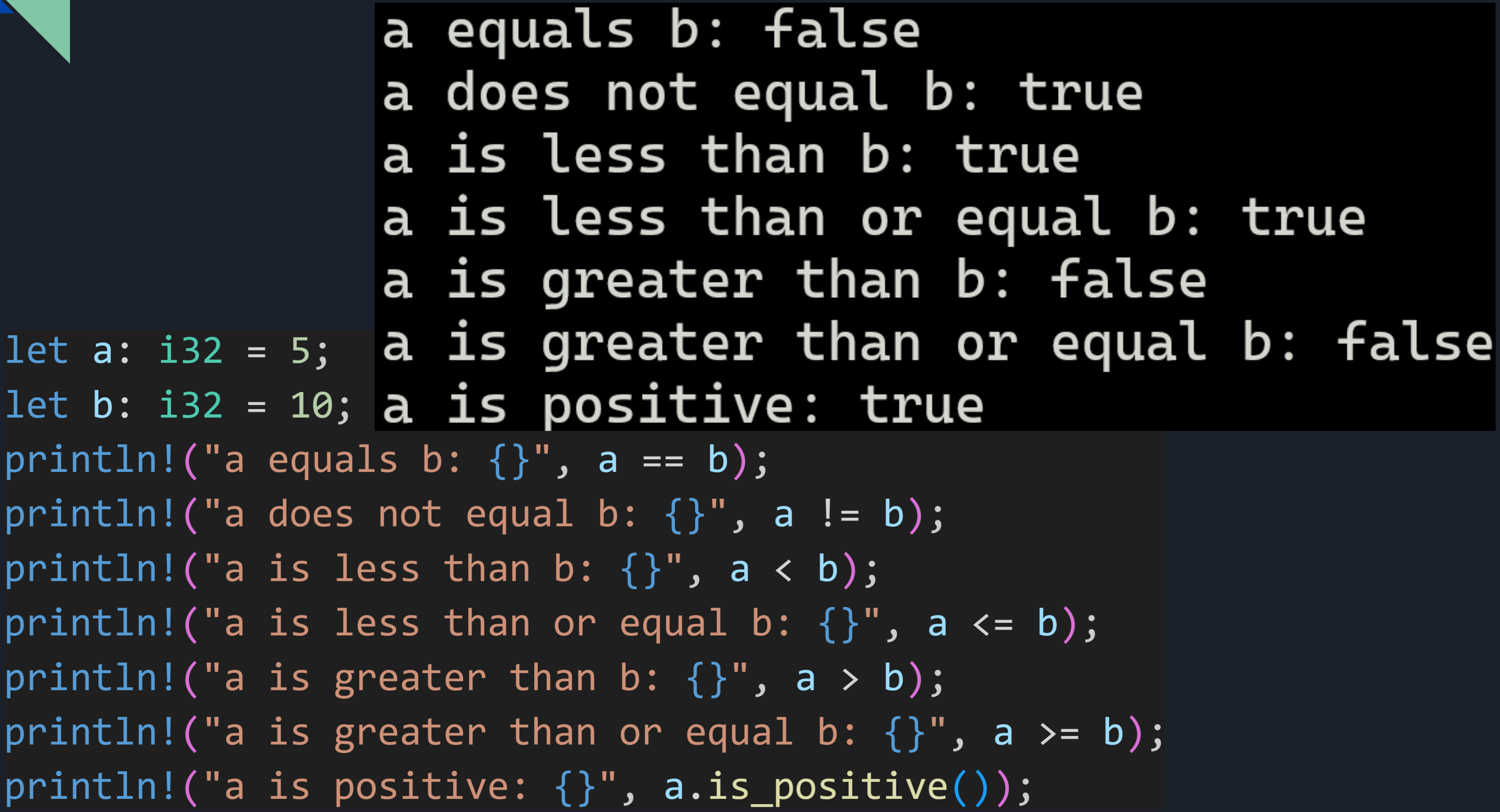
Intermission - Conditions



```
let a: i32 = 5;
let b: i32 = 10;
println!("a equals b: {}", a == b);
println!("a does not equal b: {}", a != b);
println!("a is less than b: {}", a < b);
println!("a is less than or equal b: {}", a <= b);
println!("a is greater than b: {}", a > b);
println!("a is greater than or equal b: {}", a >= b);
println!("a is positive: {}", a.is_positive());
```

1/3 What does the program output?

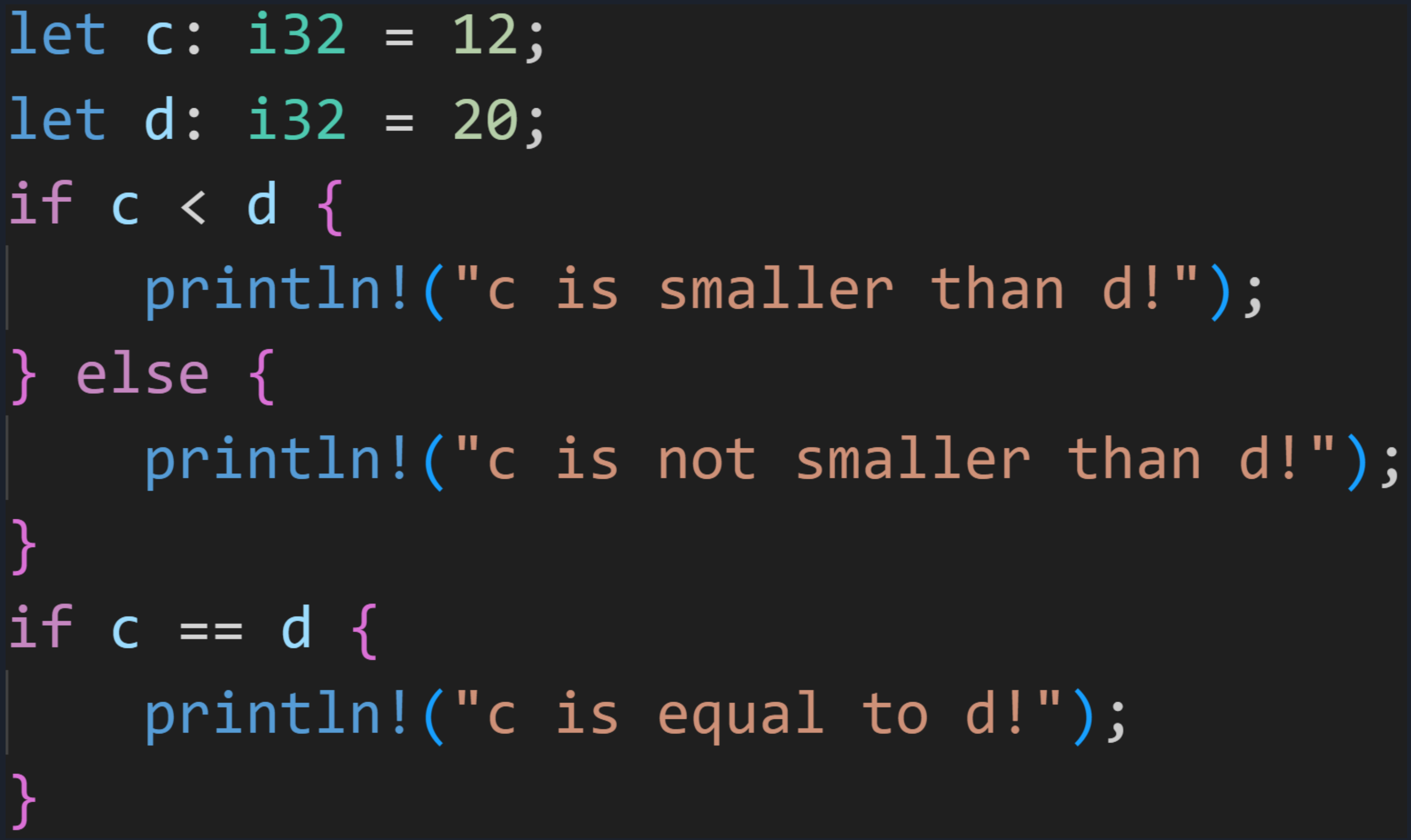
Intermission - Conditions



```
let a: i32 = 5;
let b: i32 = 10;
println!("a equals b: {}", a == b);
println!("a does not equal b: {}", a != b);
println!("a is less than b: {}", a < b);
println!("a is less than or equal b: {}", a <= b);
println!("a is greater than b: {}", a > b);
println!("a is greater than or equal b: {}", a >= b);
println!("a is positive: {}", a.is_positive());
```

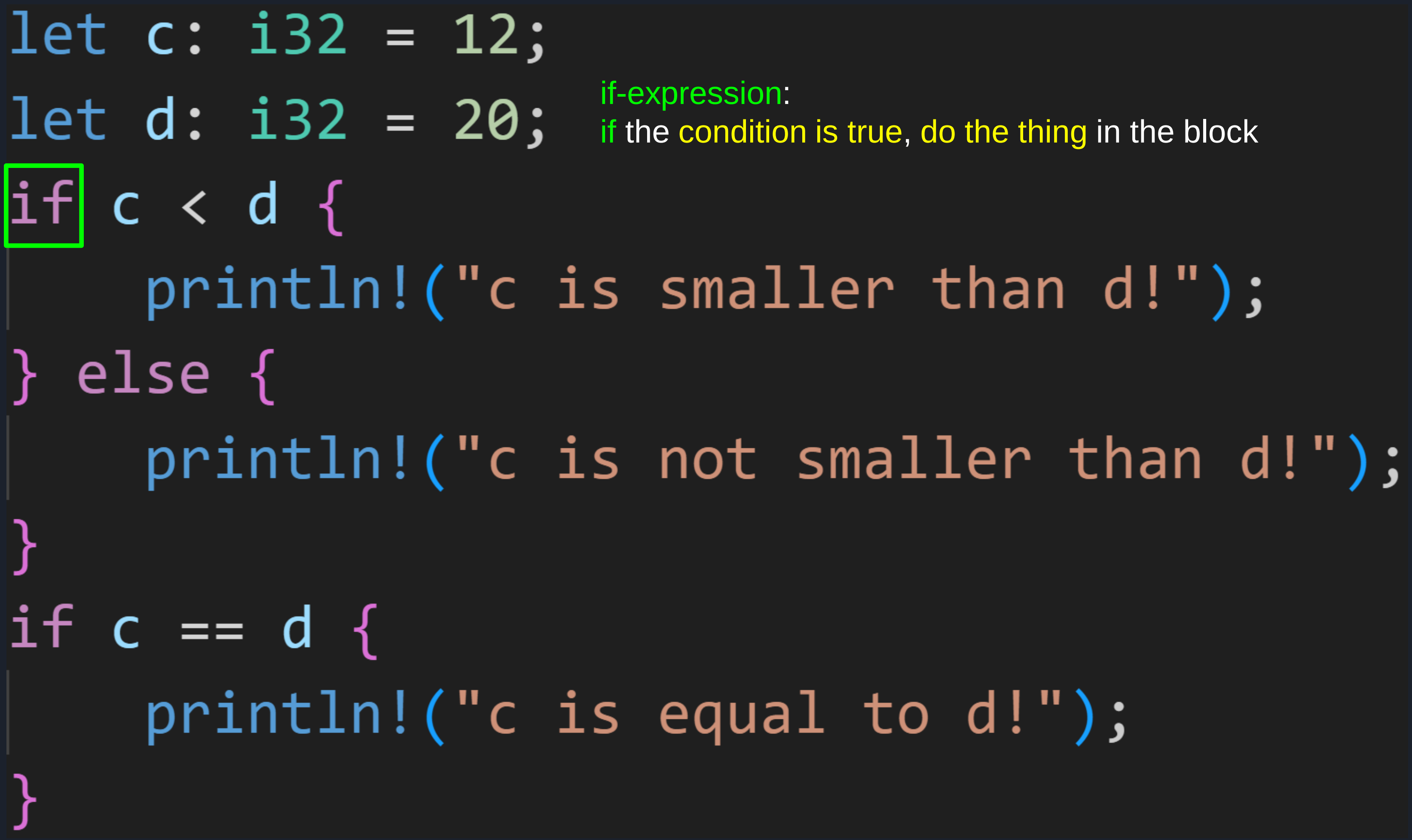
```
a equals b: false
a does not equal b: true
a is less than b: true
a is less than or equal b: true
a is greater than b: false
a is greater than or equal b: false
a is positive: true
```

Intermission - Conditions



```rust
let c: i32 = 12;
let d: i32 = 20;
if c < d {
    println!("c is smaller than d!");
} else {
    println!("c is not smaller than d!");
}
if c == d {
    println!("c is equal to d!");
}
```

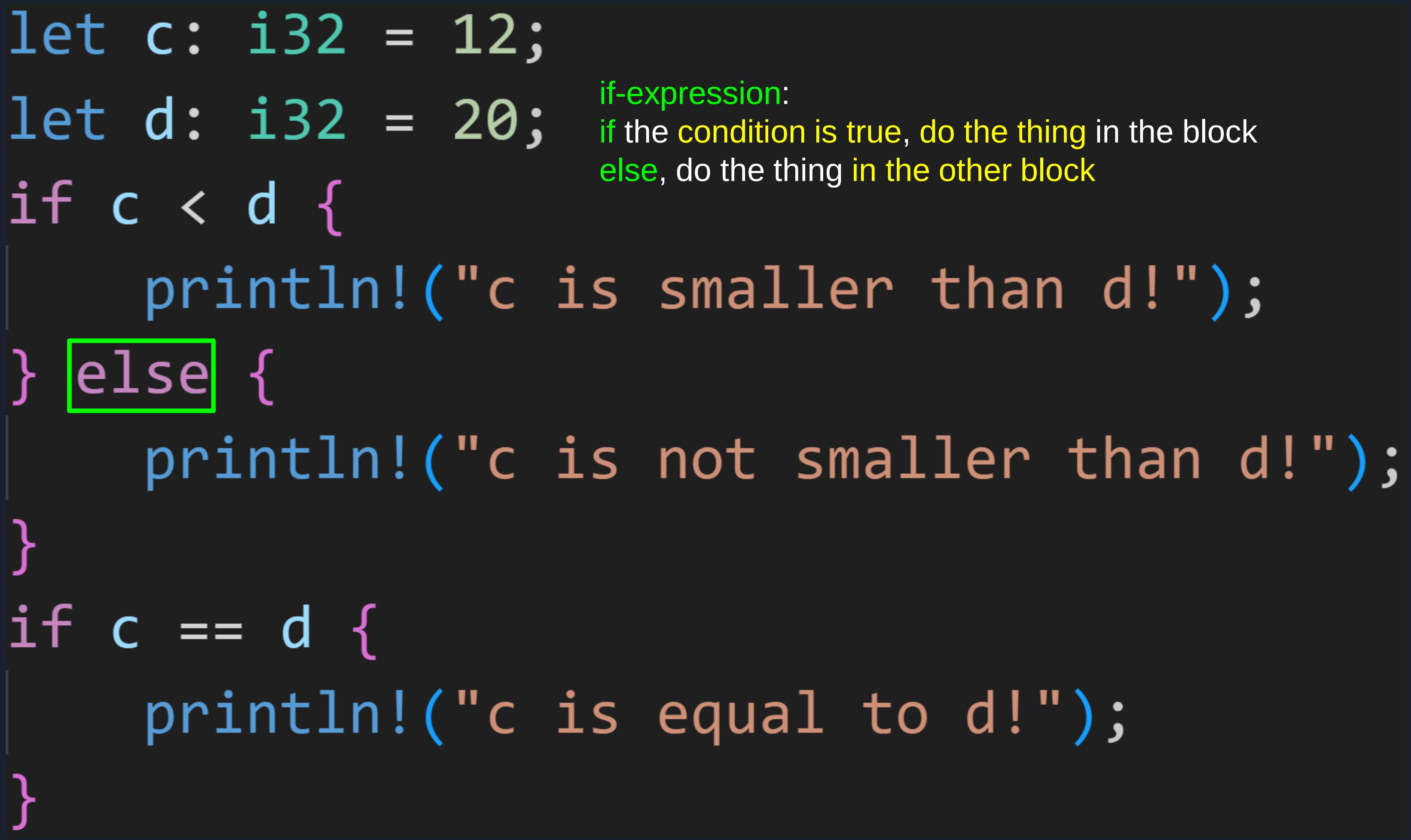
Intermission - Conditions



```rust
let c: i32 = 12;
let d: i32 = 20;
if c < d {
    println!("c is smaller than d!");
} else {
    println!("c is not smaller than d!");
}
if c == d {
    println!("c is equal to d!");
}
```

if-expression:
if the condition is true, do the thing in the block

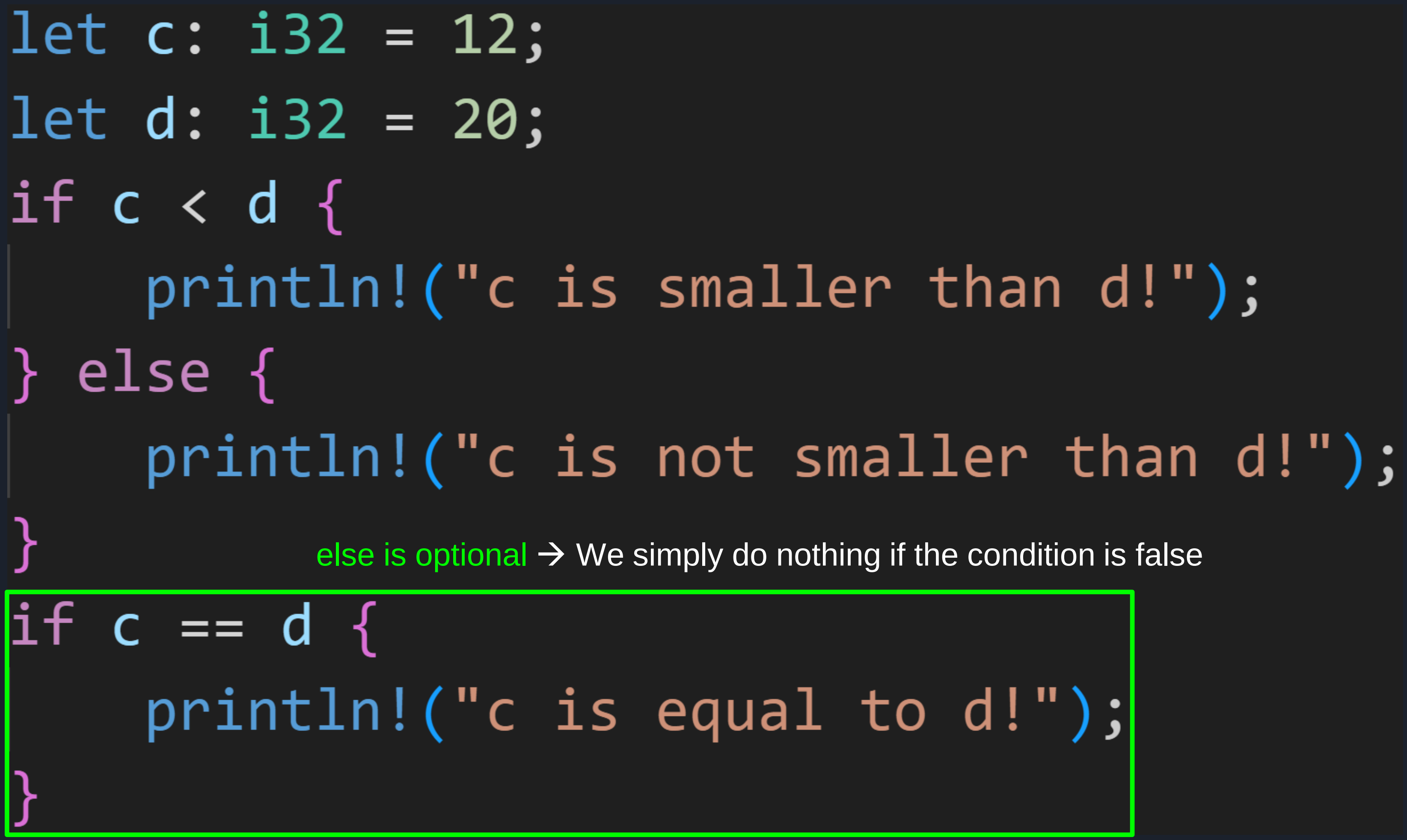
Intermission - Conditions



```
let c: i32 = 12;
let d: i32 = 20;
if c < d {
    println!("c is smaller than d!");
} else {
    println!("c is not smaller than d!");
}
if c == d {
    println!("c is equal to d!");
}
```

if-expression:
if the condition is true, do the thing in the block
else, do the thing in the other block

Intermission - Conditions



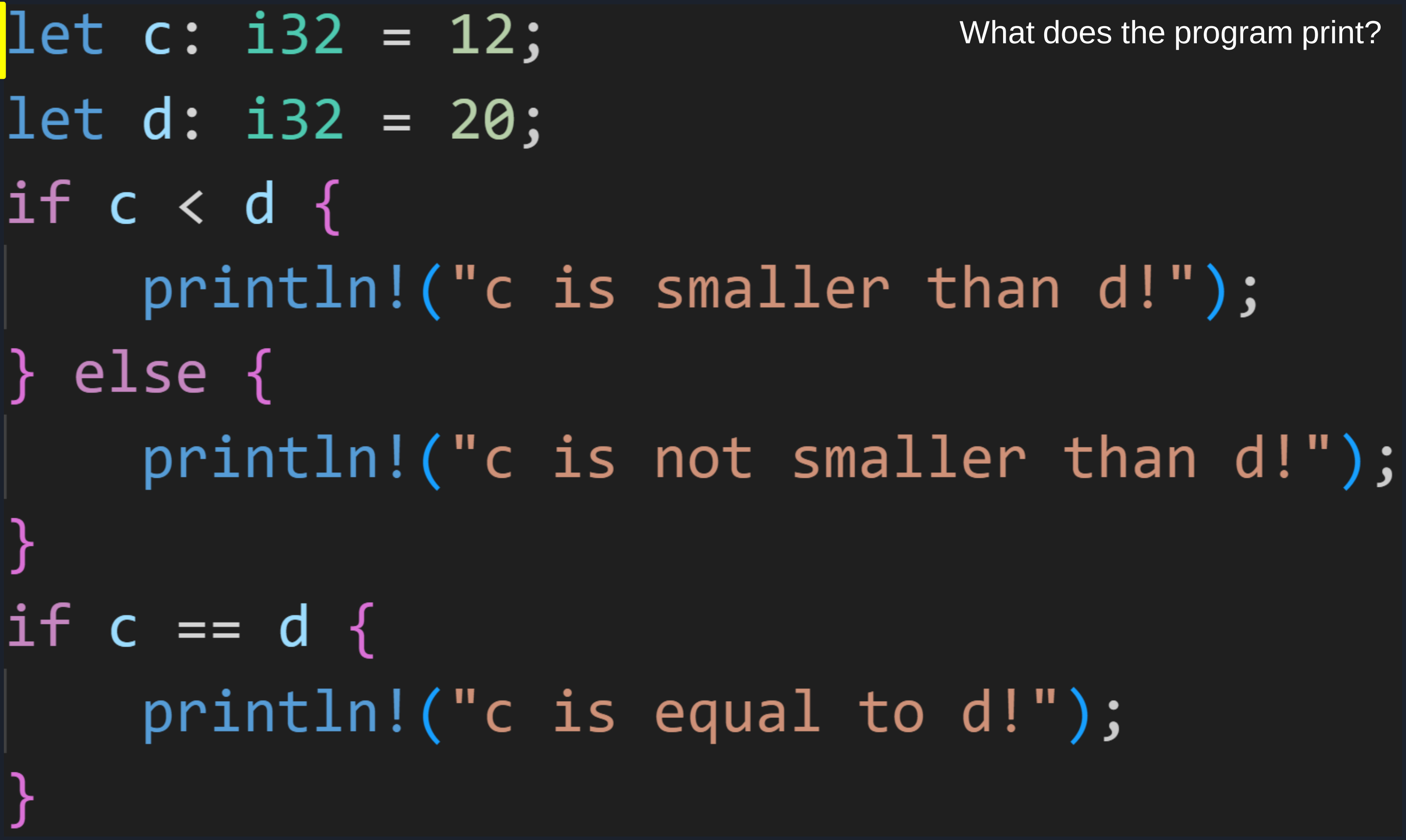
```
let c: i32 = 12;
let d: i32 = 20;
if c < d {
    println!("c is smaller than d!");
} else {
    println!("c is not smaller than d!");
}
```

else is optional → We simply do nothing if the condition is false

```
if c == d {
    println!("c is equal to d!");
}
```

Intermission - Conditions

1/3

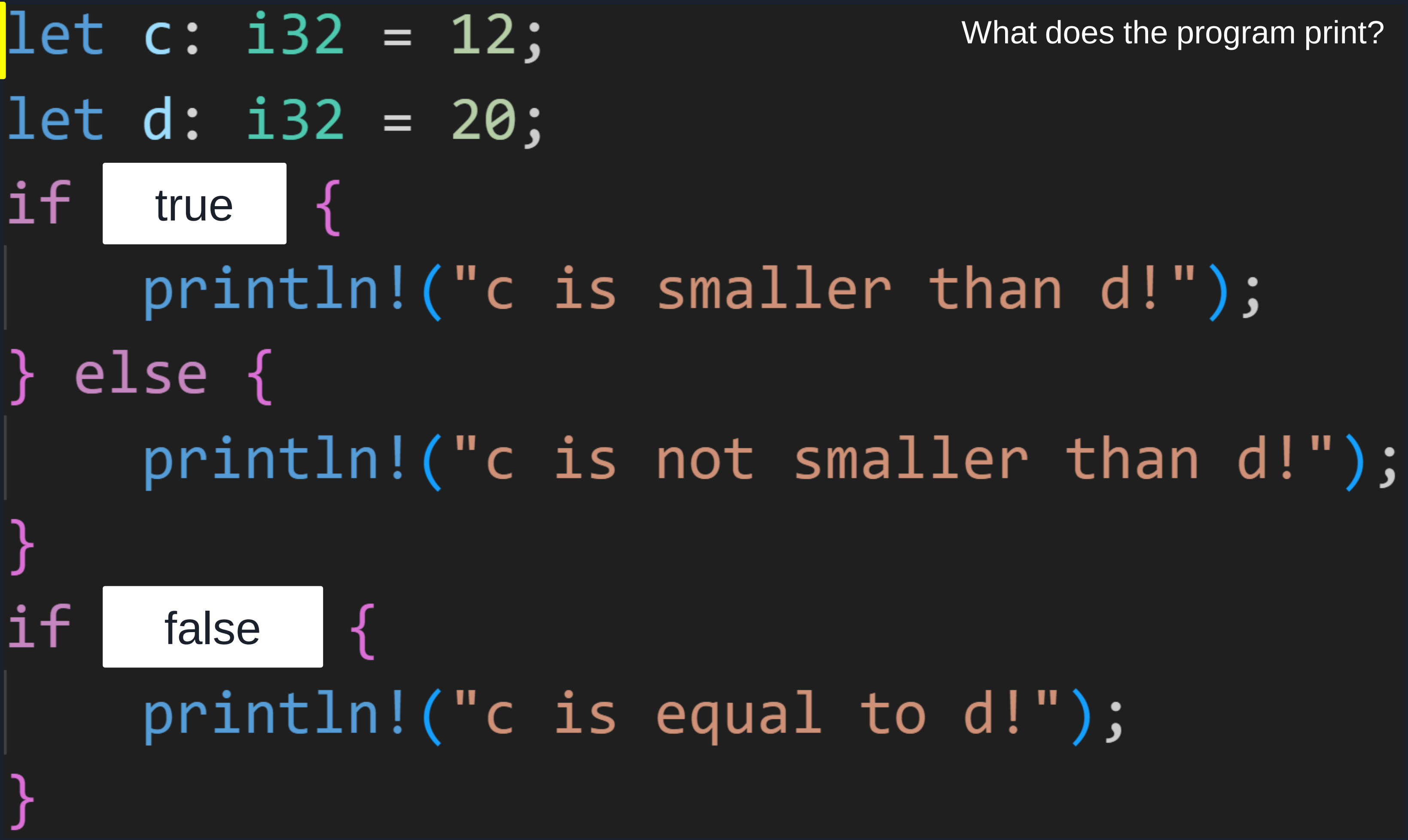


What does the program print?

```
let c: i32 = 12;
let d: i32 = 20;
if c < d {
    println!("c is smaller than d!");
} else {
    println!("c is not smaller than d!");
}
if c == d {
    println!("c is equal to d!");
}
```

Intermission - Conditions

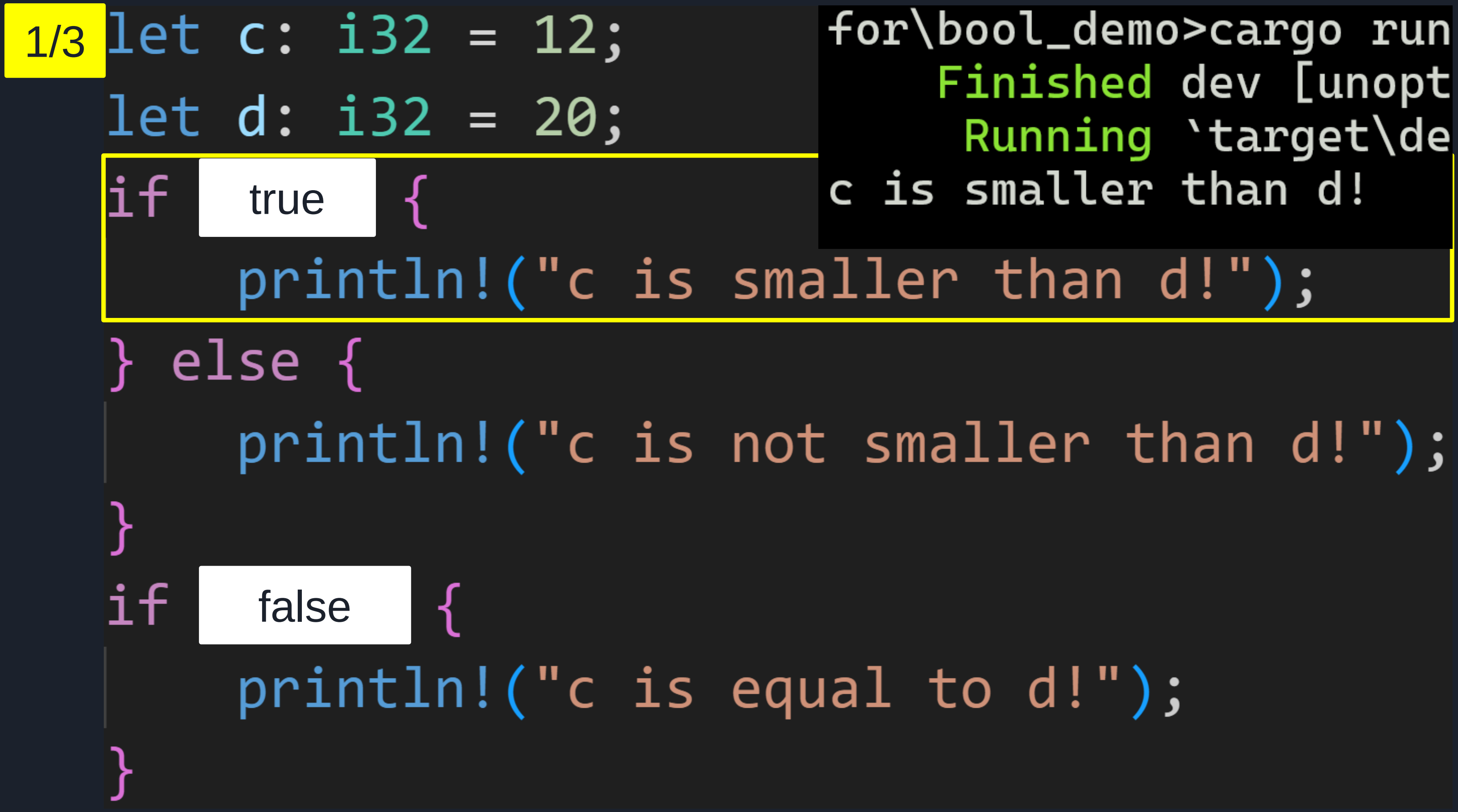
1/3



What does the program print?

```
let c: i32 = 12;
let d: i32 = 20;
if true {
    println!("c is smaller than d!");
} else {
    println!("c is not smaller than d!");
}
if false {
    println!("c is equal to d!");
}
```

Intermission - Conditions



We only get one line of output

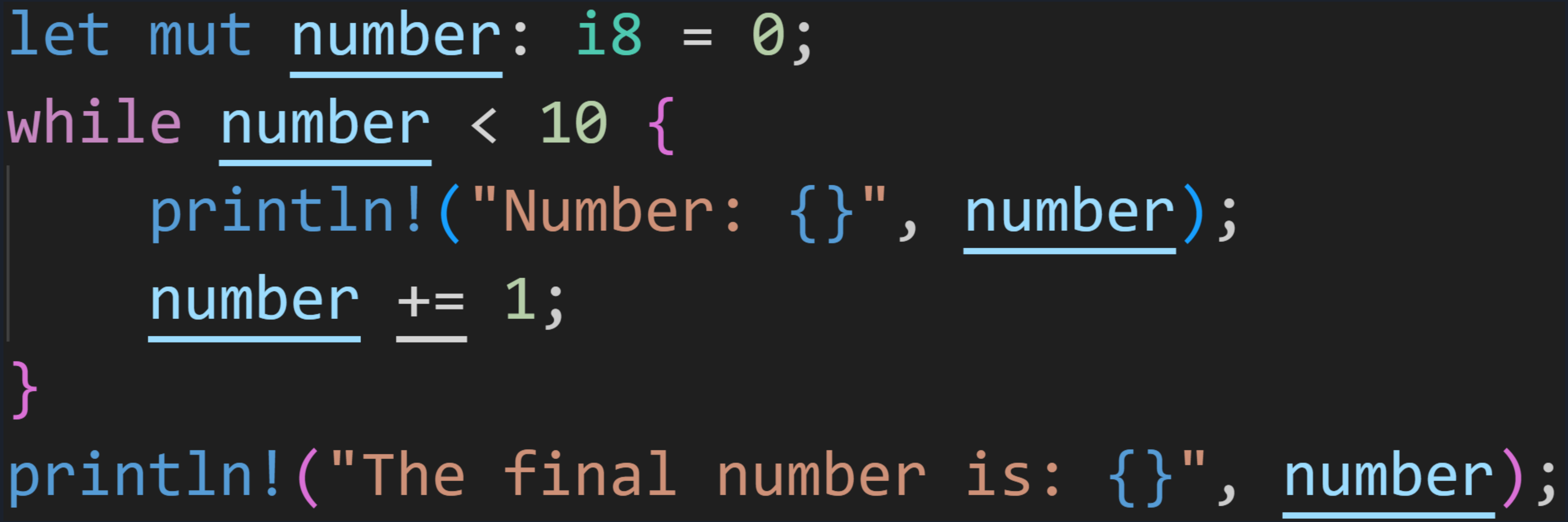
```
for\bool_demo>cargo run
    Finished dev [unopt
     Running `target\de
c is smaller than d!
```

1/3

```
let c: i32 = 12;
let d: i32 = 20;
if true {
    println!("c is smaller than d!");
} else {
    println!("c is not smaller than d!");
}
if false {
    println!("c is equal to d!");
}
```

2. Loops - while

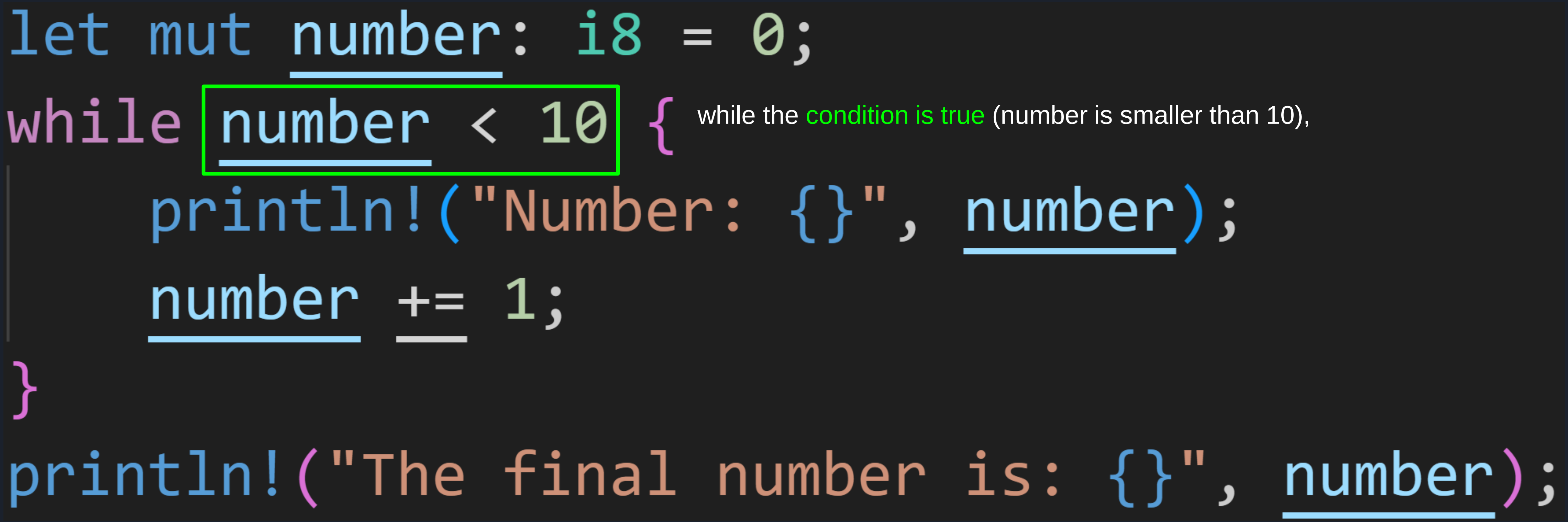
a while loop may look like this



```rust
let mut number: i8 = 0;
while number < 10 {
    println!("Number: {}", number);
    number += 1;
}
println!("The final number is: {}", number);
```

2. Loops - while

a while loop may look like this

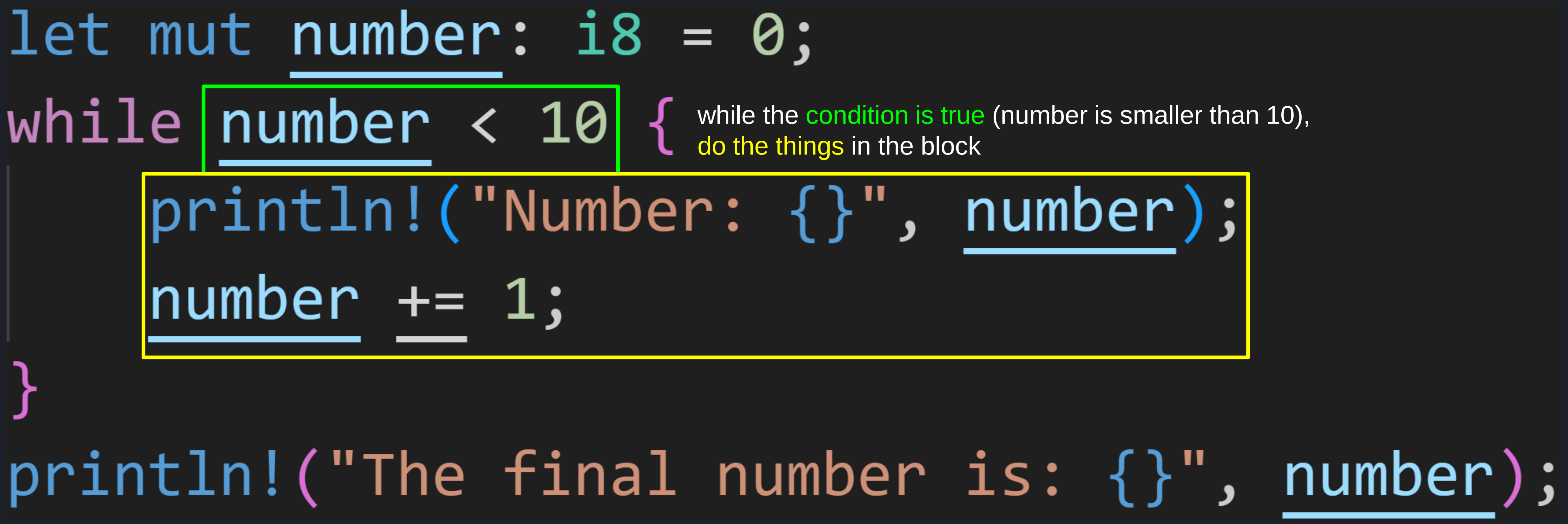


```
let mut number: i8 = 0;
while number < 10 {
    println!("Number: {}", number);
    number += 1;
}
println!("The final number is: {}", number);
```

while the condition is true (number is smaller than 10),

2. Loops - while

a while loop may look like this

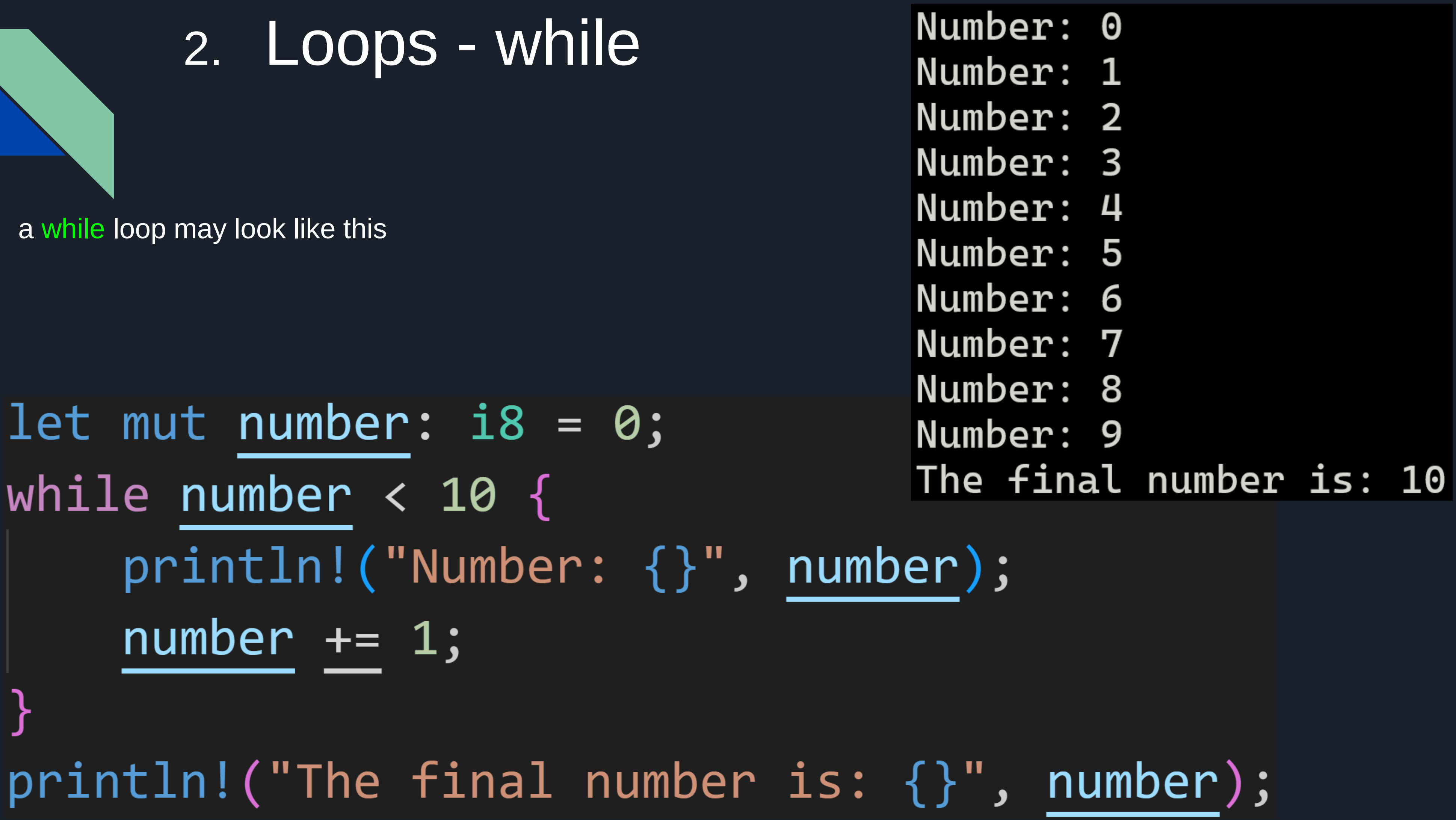


```
let mut number: i8 = 0;
while number < 10 {
    println!("Number: {}", number);
    number += 1;
}
println!("The final number is: {}", number);
```

while the condition is true (number is smaller than 10),
do the things in the block

2. Loops - while

a while loop may look like this

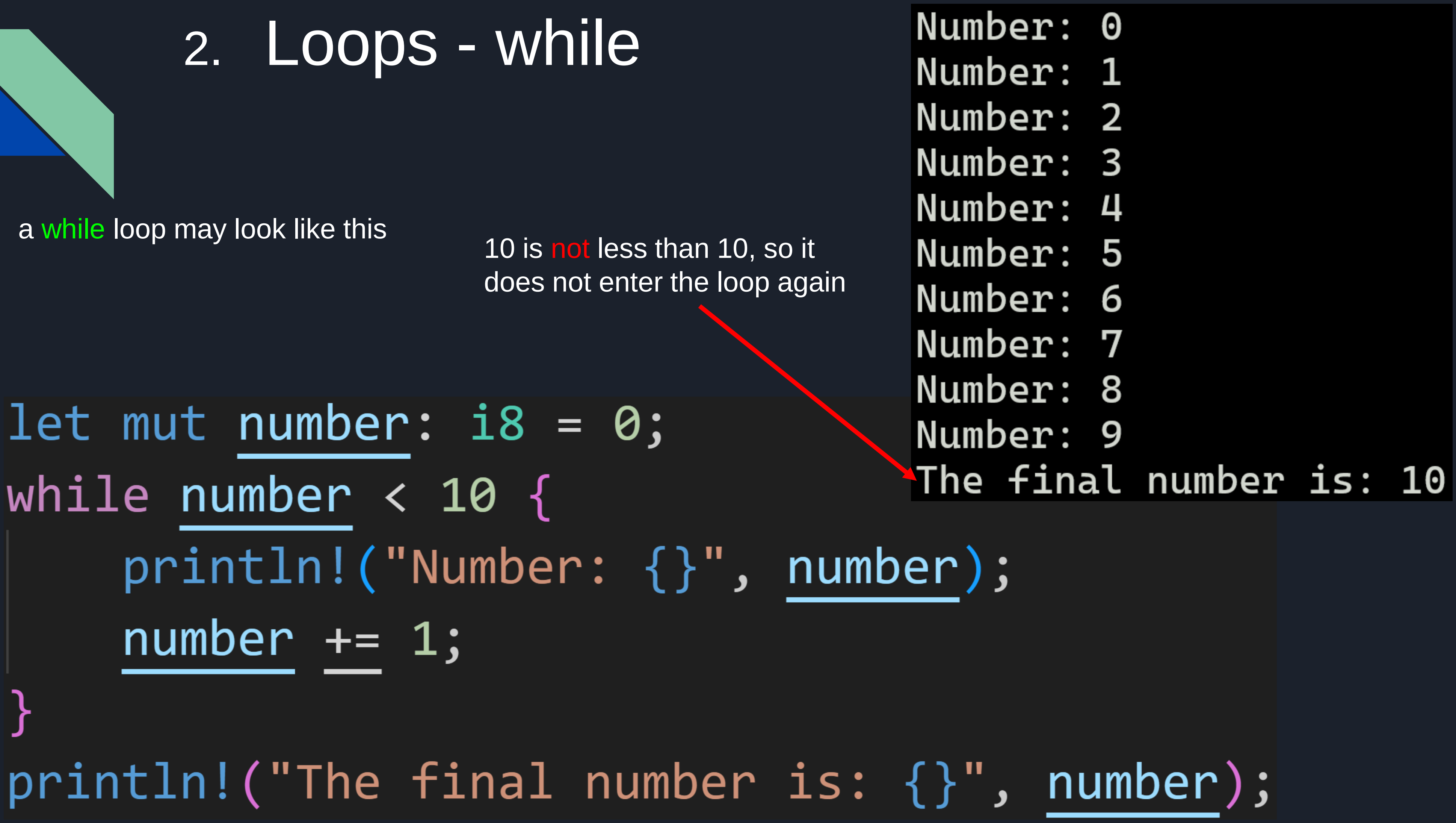


```rust
let mut number: i8 = 0;
while number < 10 {
    println!("Number: {}", number);
    number += 1;
}
println!("The final number is: {}", number);
```

```
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
The final number is: 10
```

2. Loops - while

a while loop may look like this



10 is not less than 10, so it does not enter the loop again

```rust
let mut number: i8 = 0;
while number < 10 {
    println!("Number: {}", number);
    number += 1;
}
println!("The final number is: {}", number);
```

```
Number:  0
Number:  1
Number:  2
Number:  3
Number:  4
Number:  5
Number:  6
Number:  7
Number:  8
Number:  9
The final number is:  10
```

Intermission - Scopes

Intermission - Scopes

- Statements wrapped in curly brackets are called code blocks

Intermission - Scopes

- Statements wrapped in curly brackets are called code blocks
- Each Block has a scope in which it operates

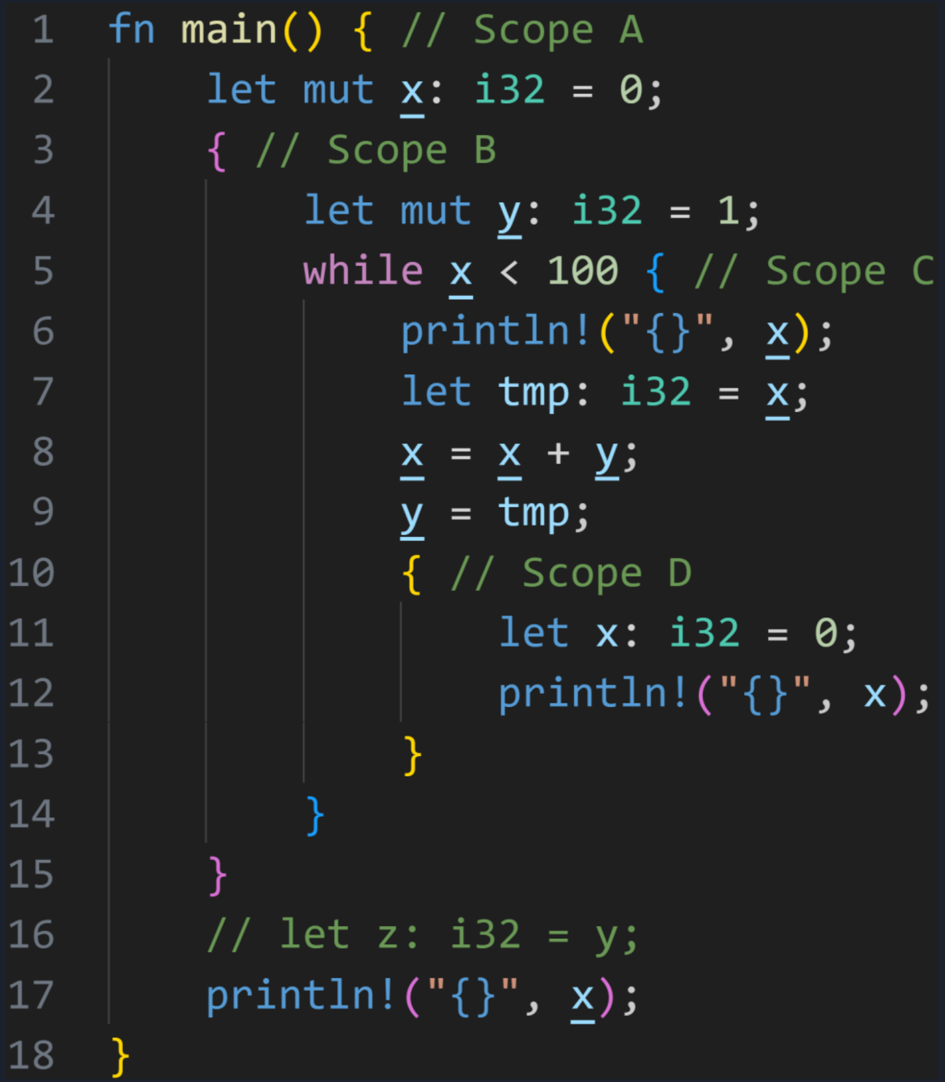
Intermission - Scopes

- Statements wrapped in curly brackets are called code blocks
- Each Block has a scope in which it operates
- Blocks can access variables from outer scopes

Intermission - Scopes

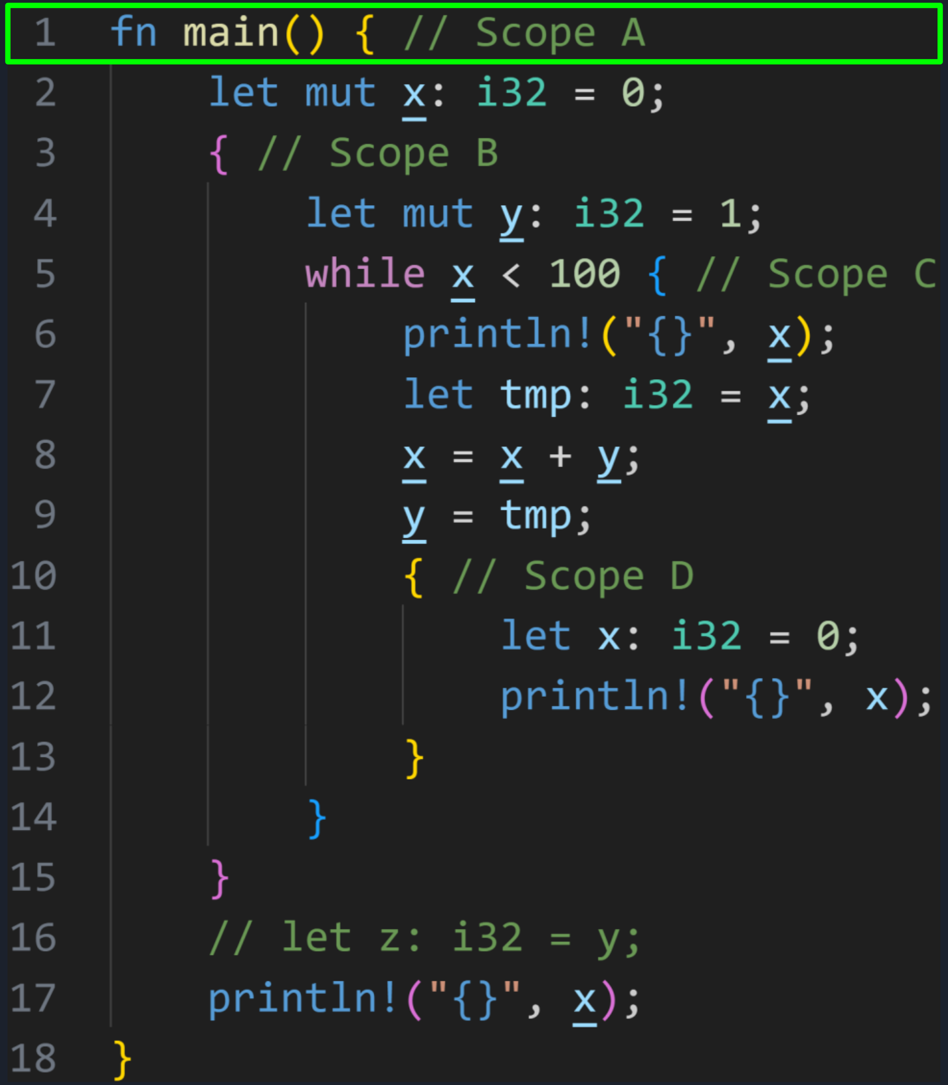
- Statements wrapped in curly brackets are called code blocks
- Each Block has a scope in which it operates
- Blocks can access variables from outer scopes
- Variables can only be used in scopes they‘re defined in
  - They‘re dropped once we leave the scope

Intermission - Scopes



```
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

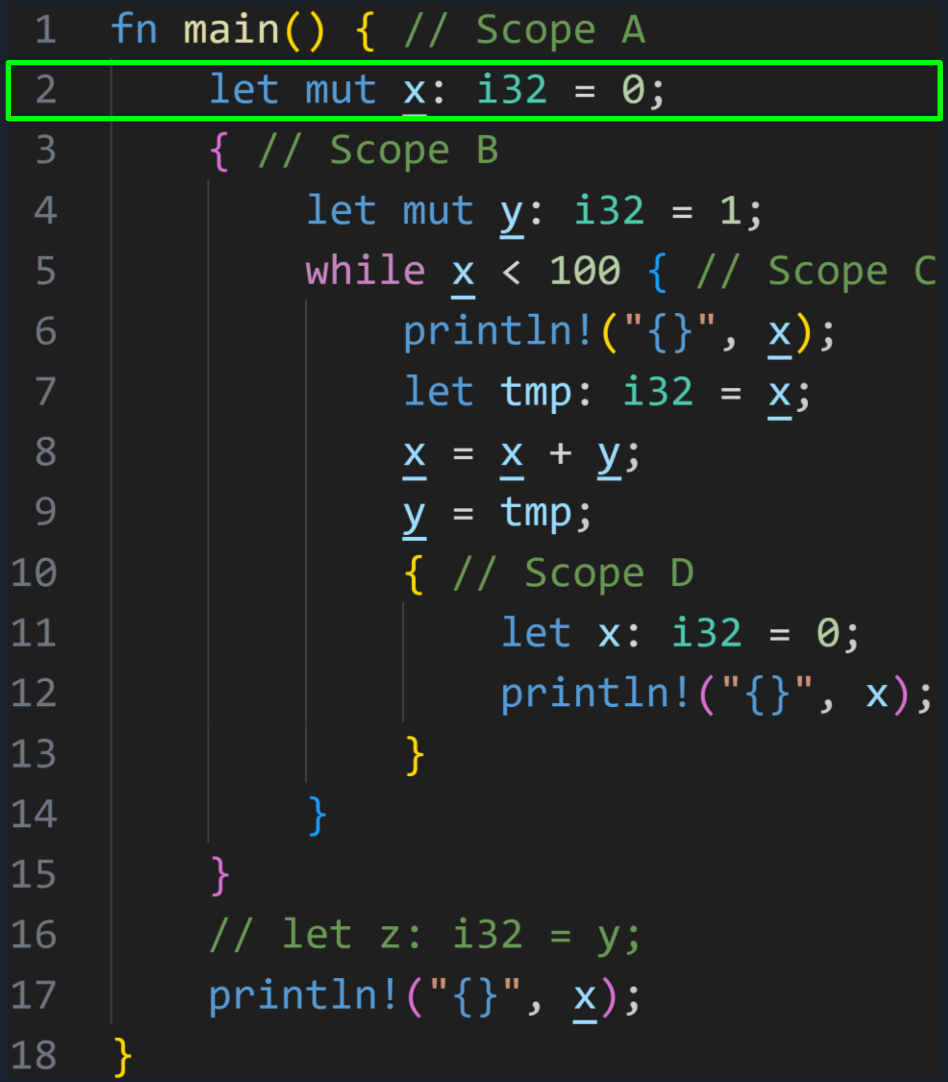
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	

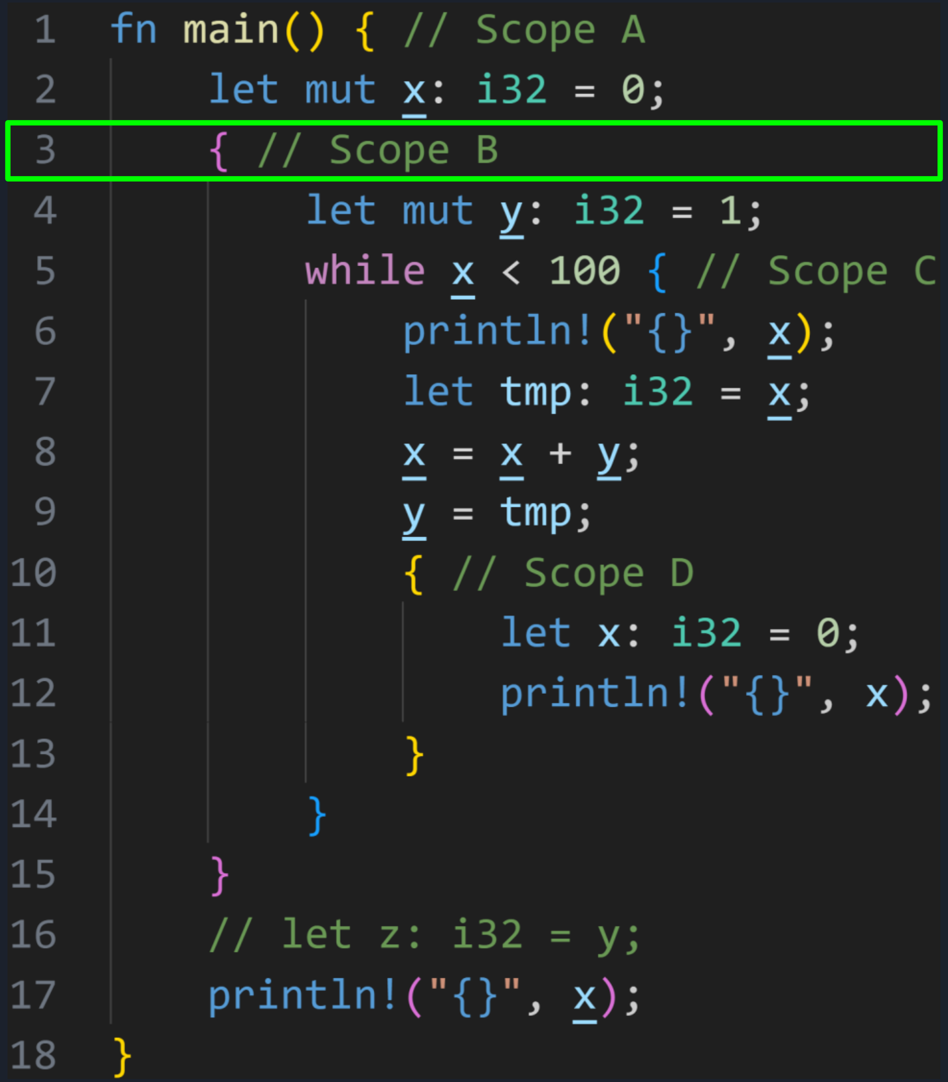
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 0

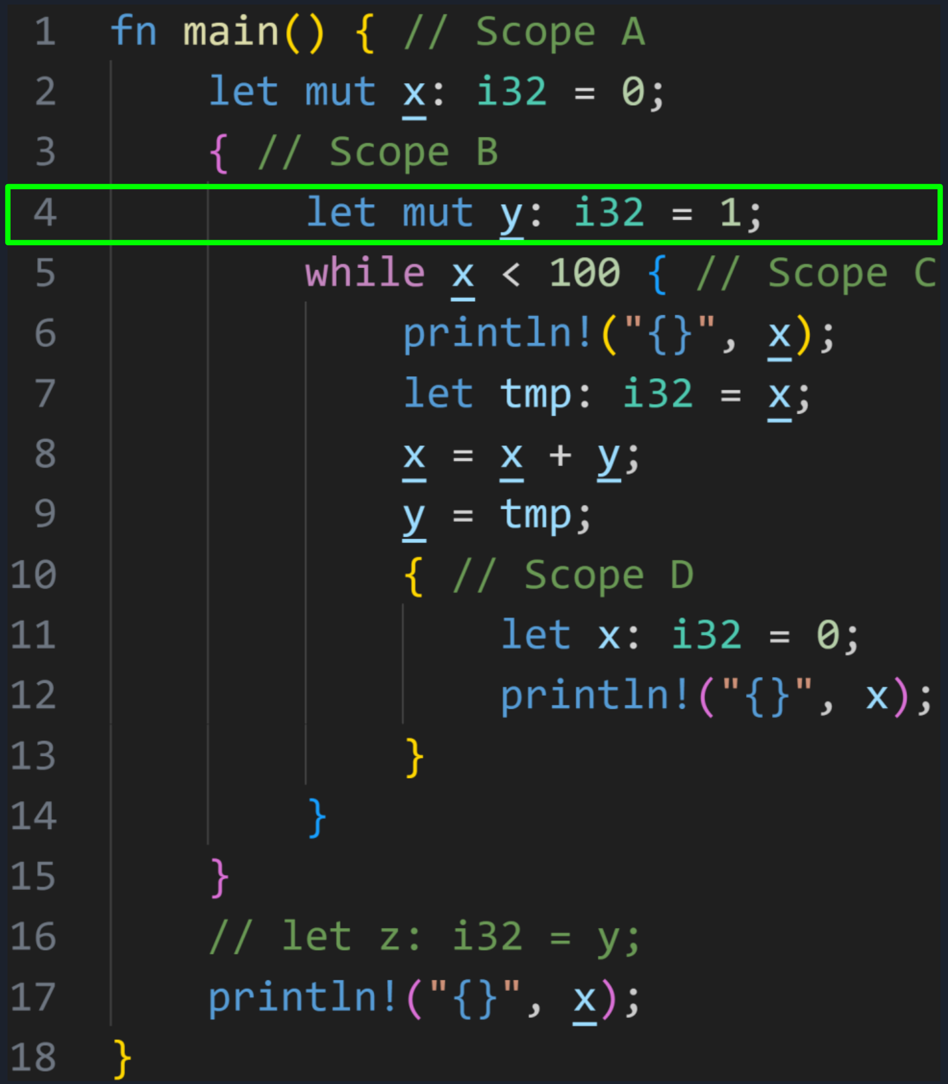
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 0
B	

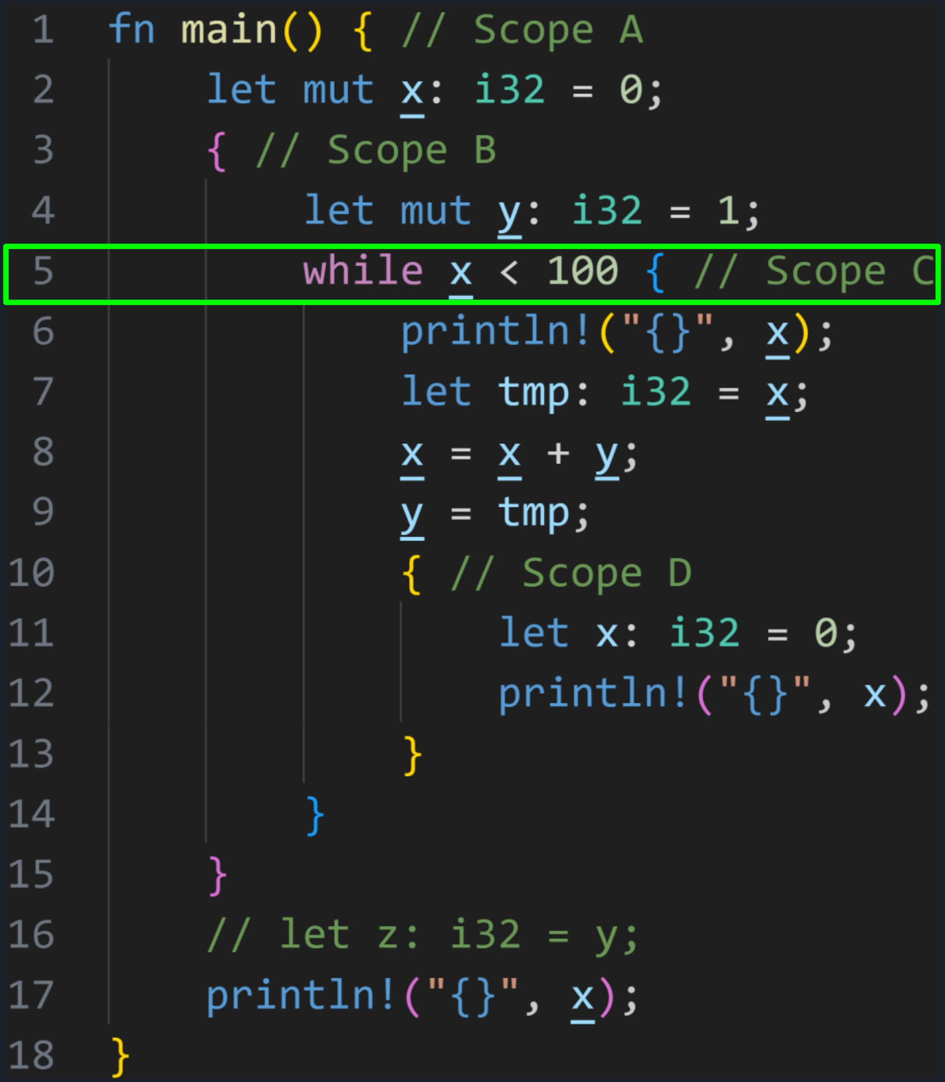
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 0
B	y = 1

Intermission - Scopes

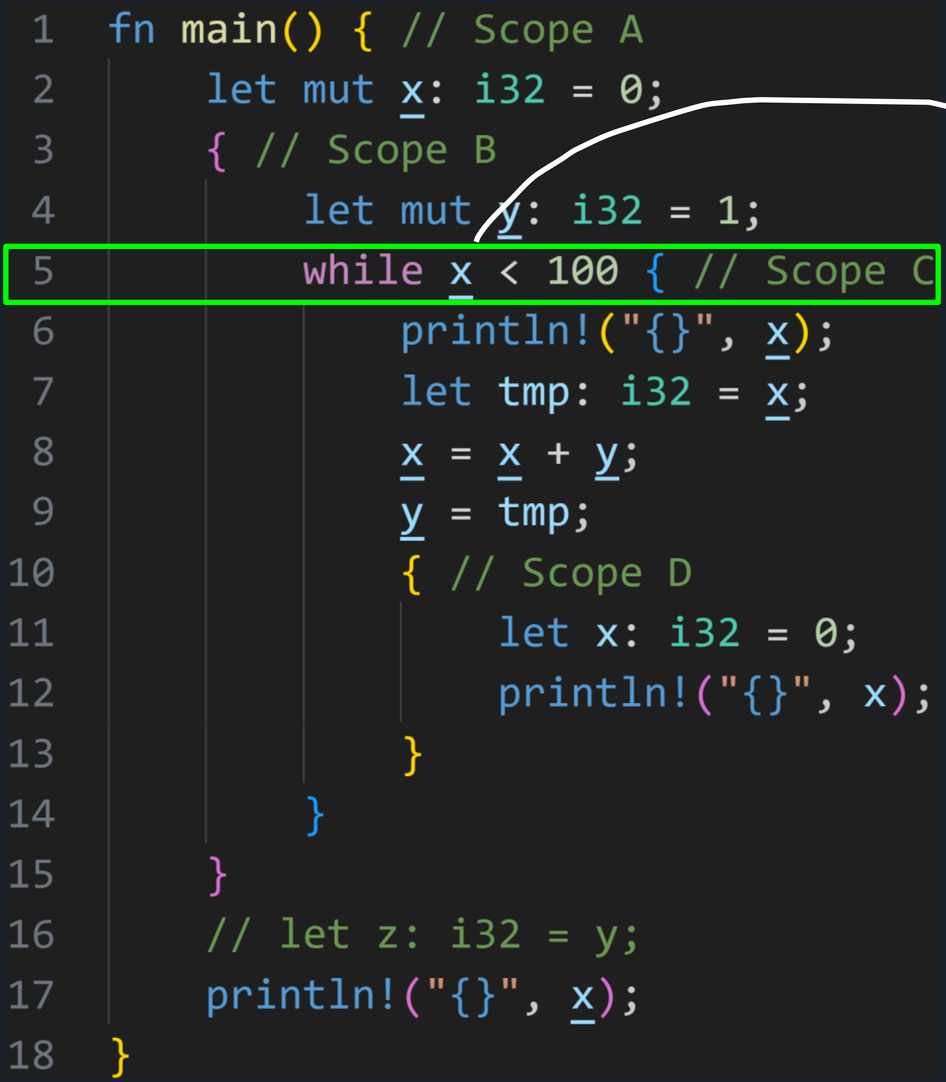


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 0
B	y = 1

Check current scope
→ No x found in B
→ Check previous scope

Intermission - Scopes

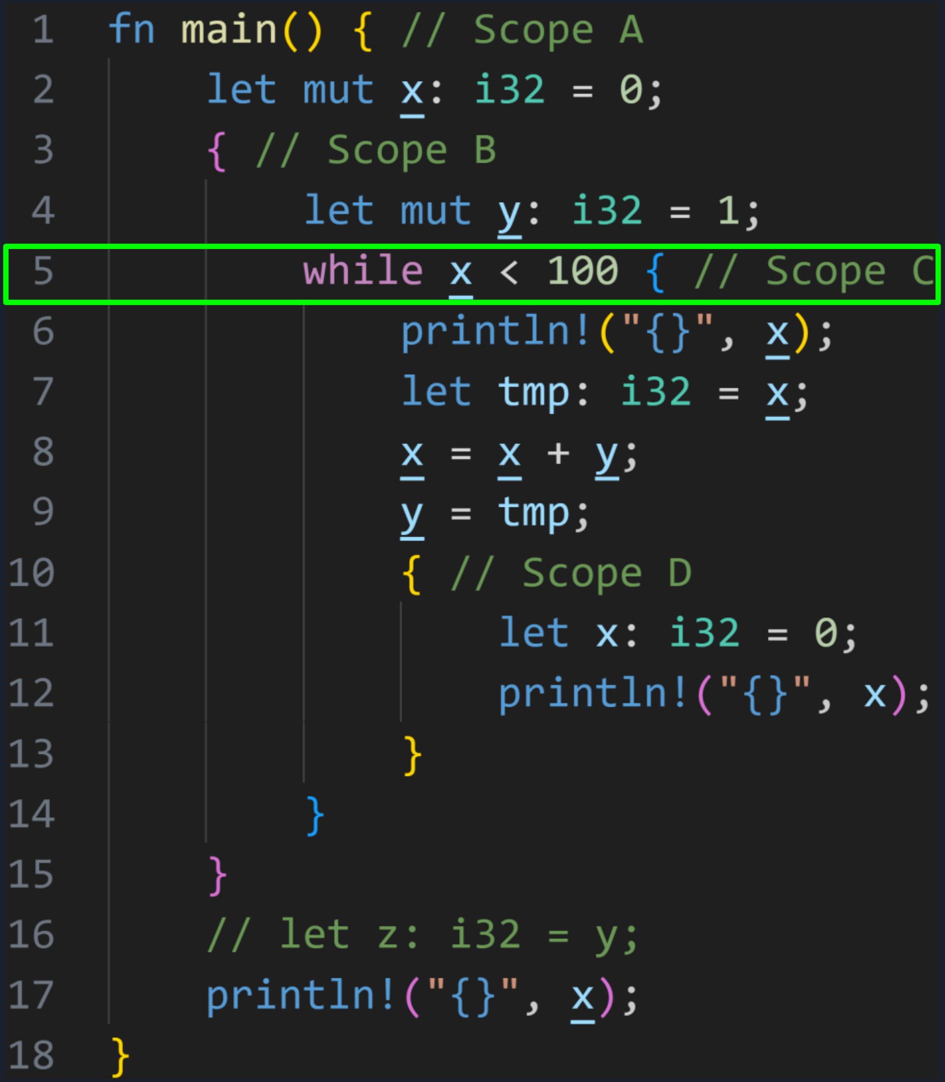


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 0
B	y = 1

Uses x from Scope A

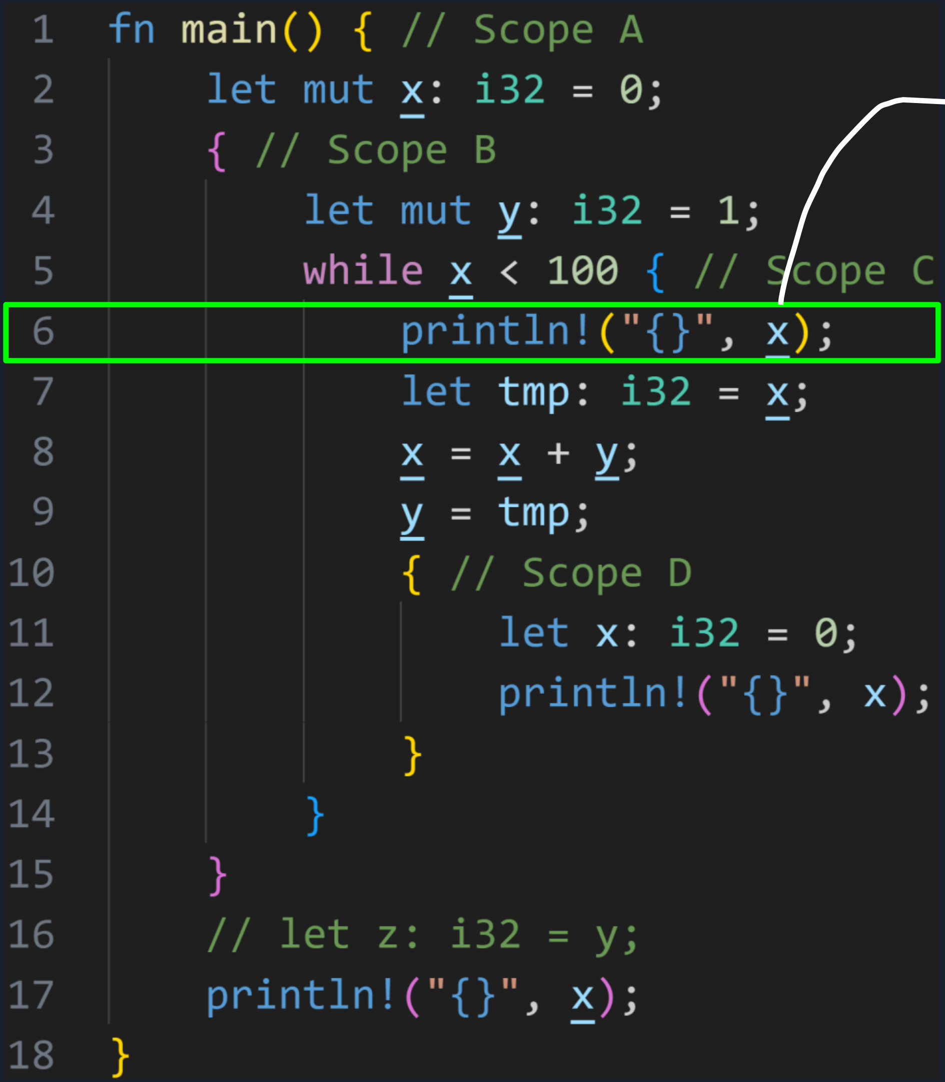
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 0
B	y = 1
C	

Intermission - Scopes

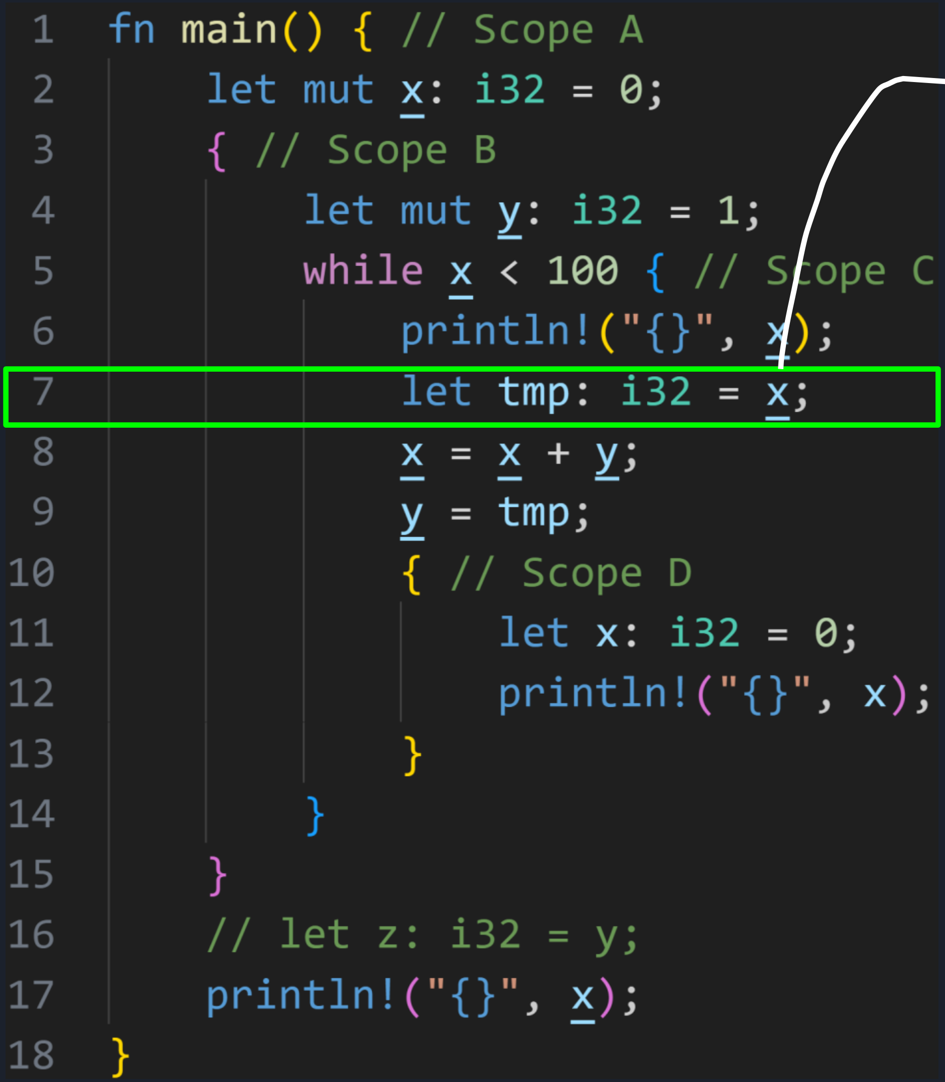


```
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 0
B	y = 1
C	

Uses x from Scope A

Intermission - Scopes

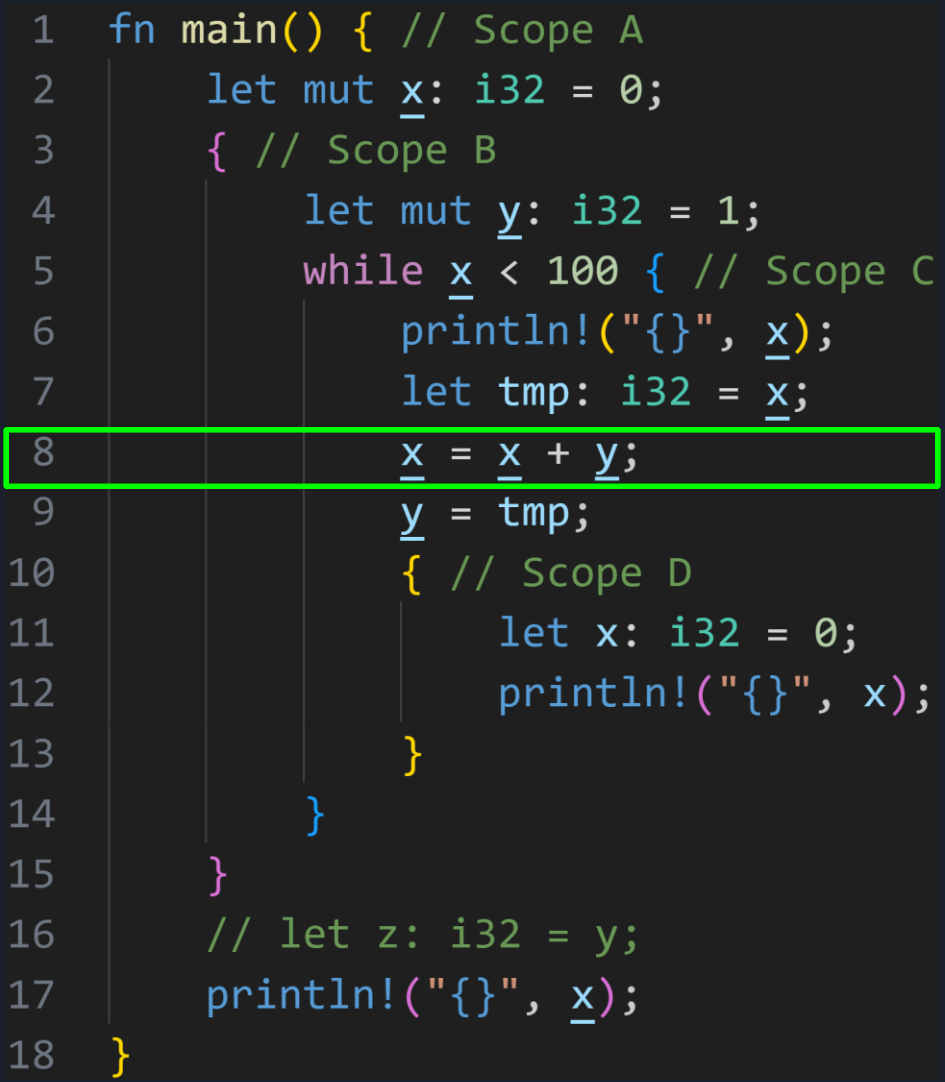


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 0
B	y = 1
C	tmp = 0

Uses x from Scope A

Intermission - Scopes

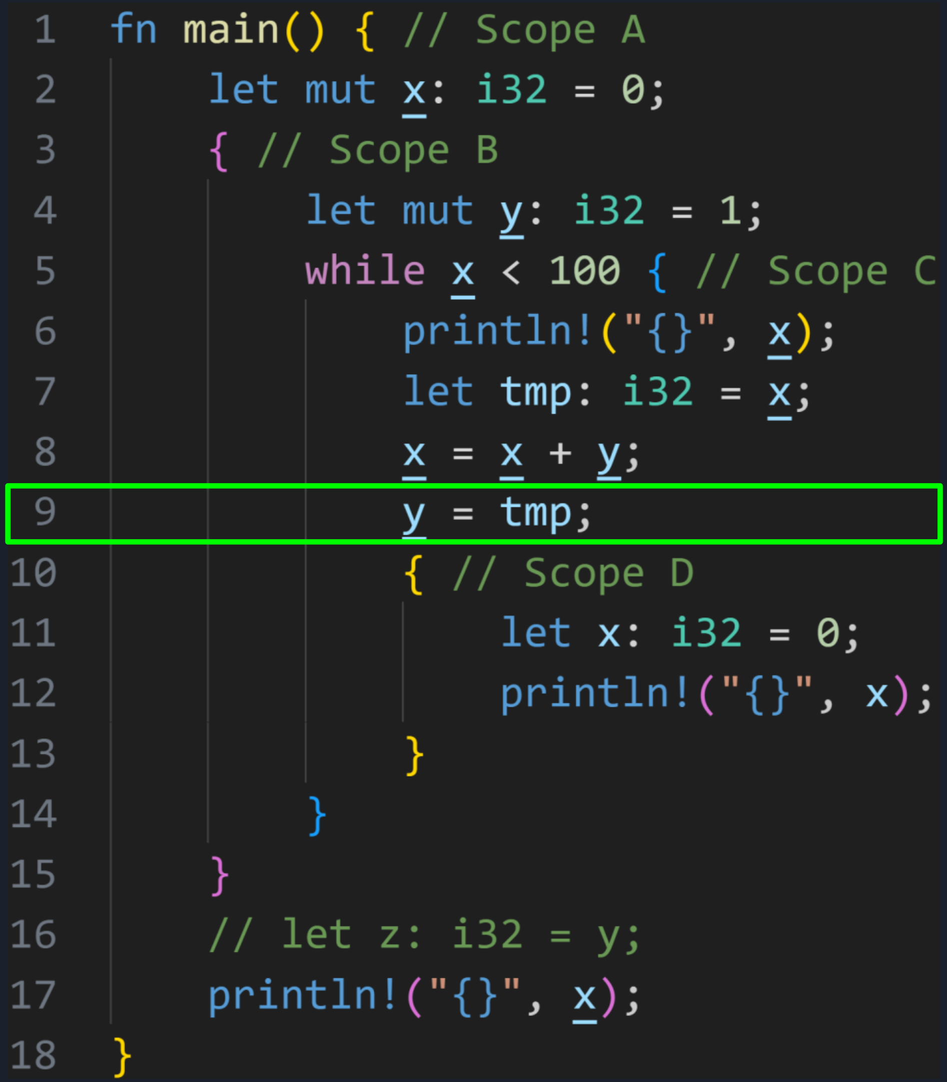


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 1
C	tmp = 0

Uses x from Scope A
Uses y from Scope B

Intermission - Scopes

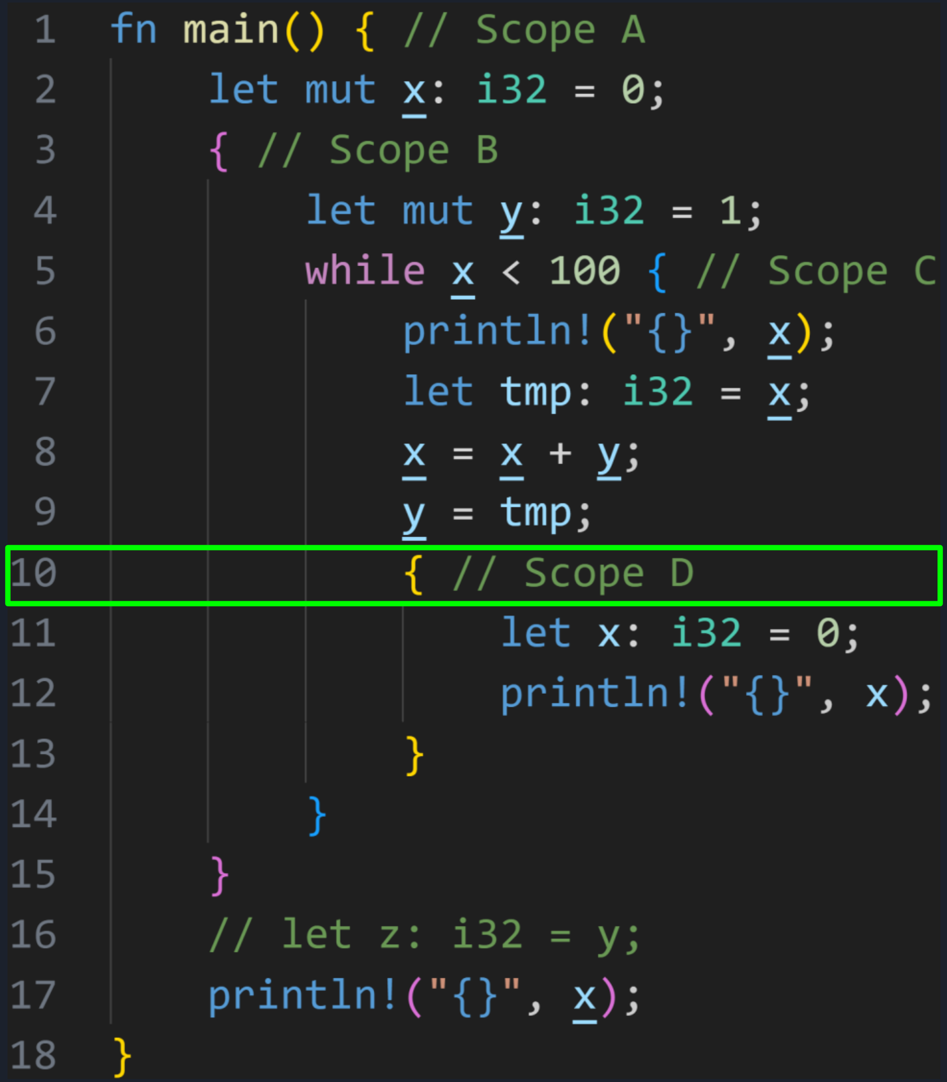


```
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 0
C	tmp = 0

Uses x from Scope A
Uses y from Scope B

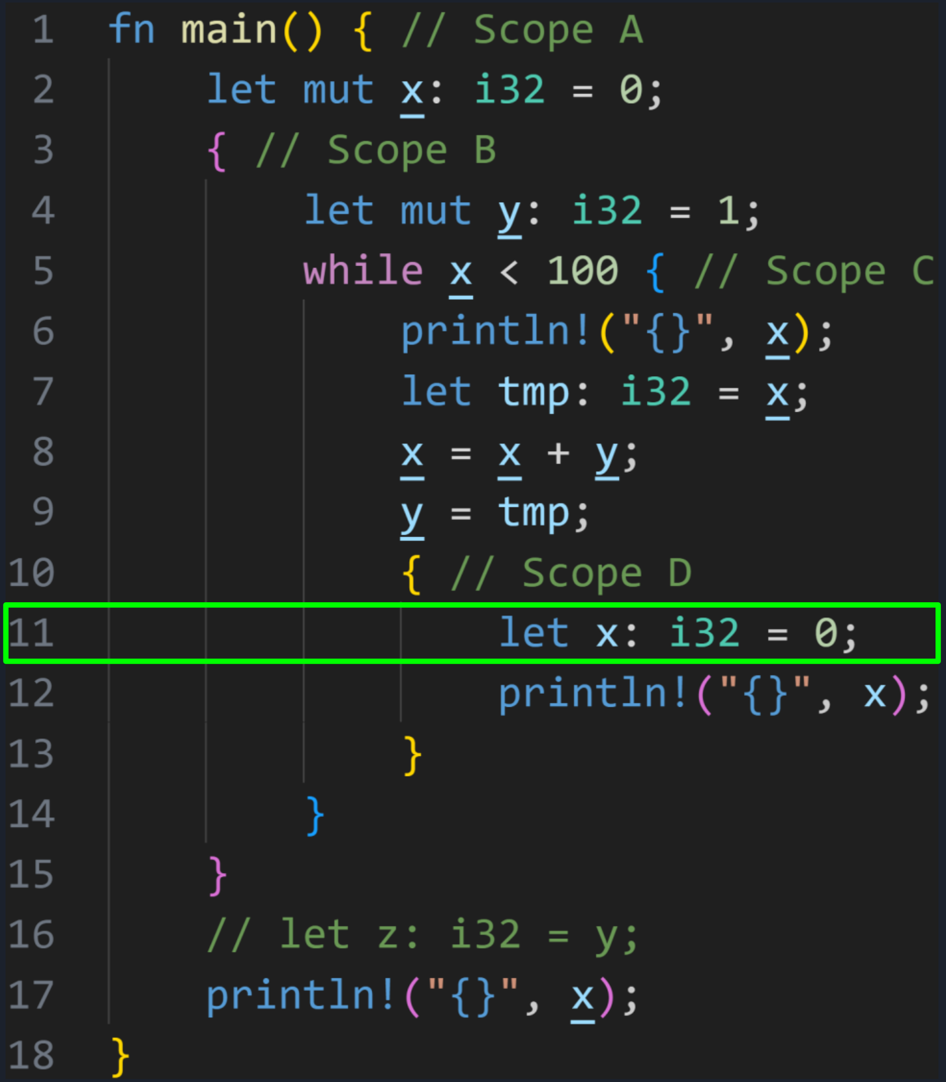
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 0
C	tmp = 0
D	

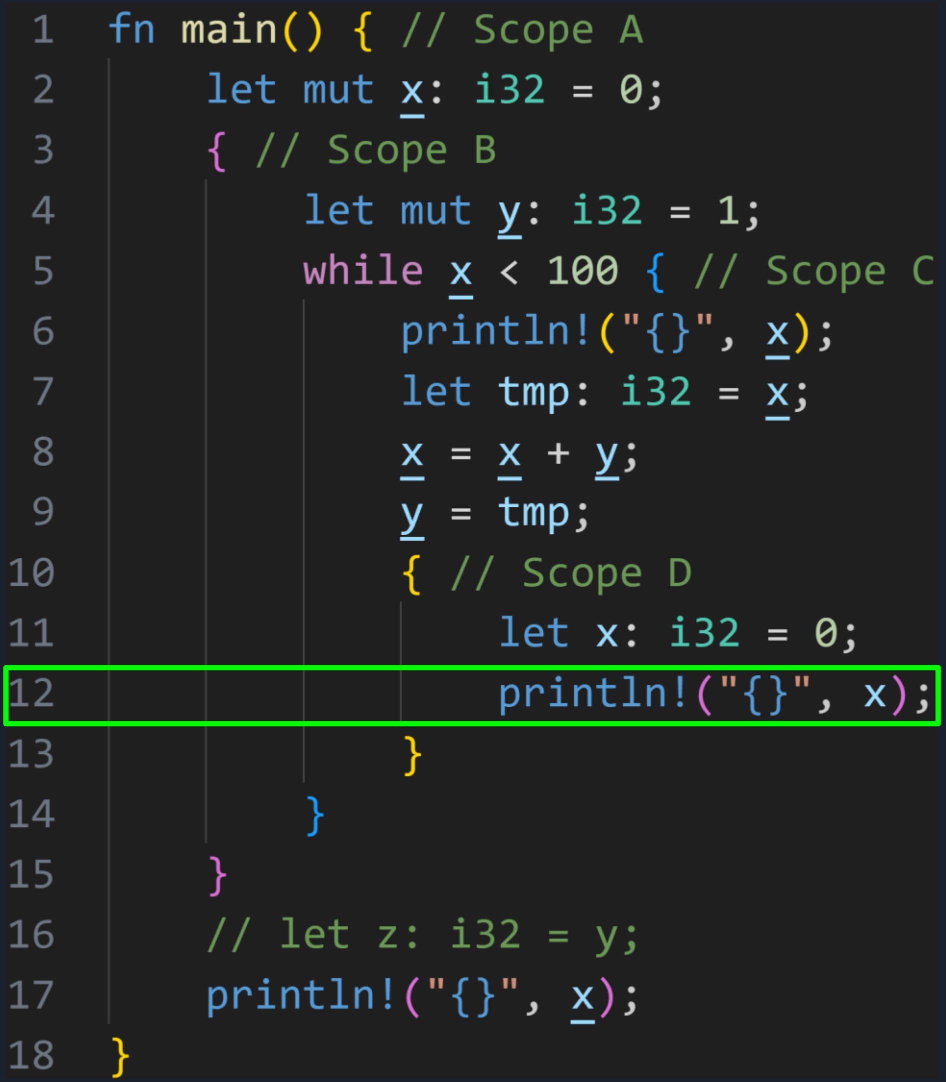
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 0
C	tmp = 0
D	x = 0

Intermission - Scopes

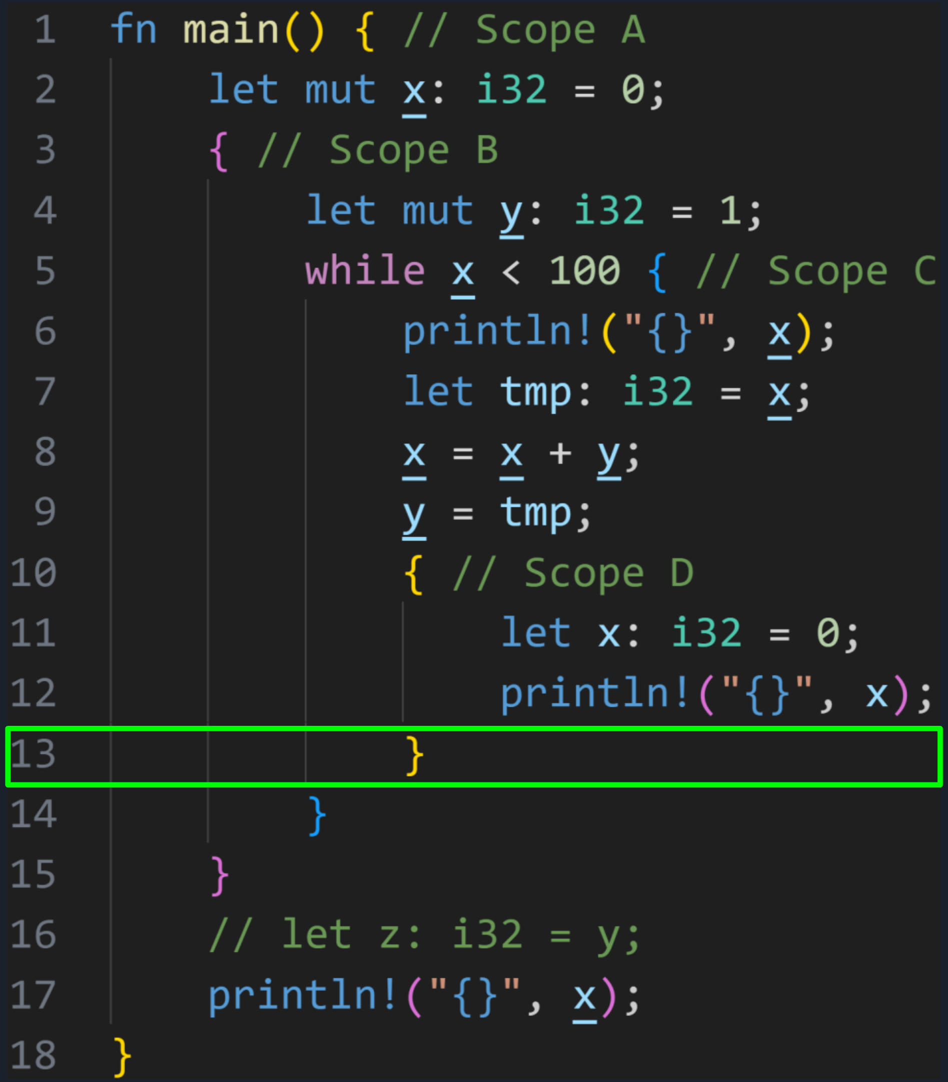


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 0
C	tmp = 0
D	x = 0

x is defined in the current scope, use this one!
→ Variable shadowing

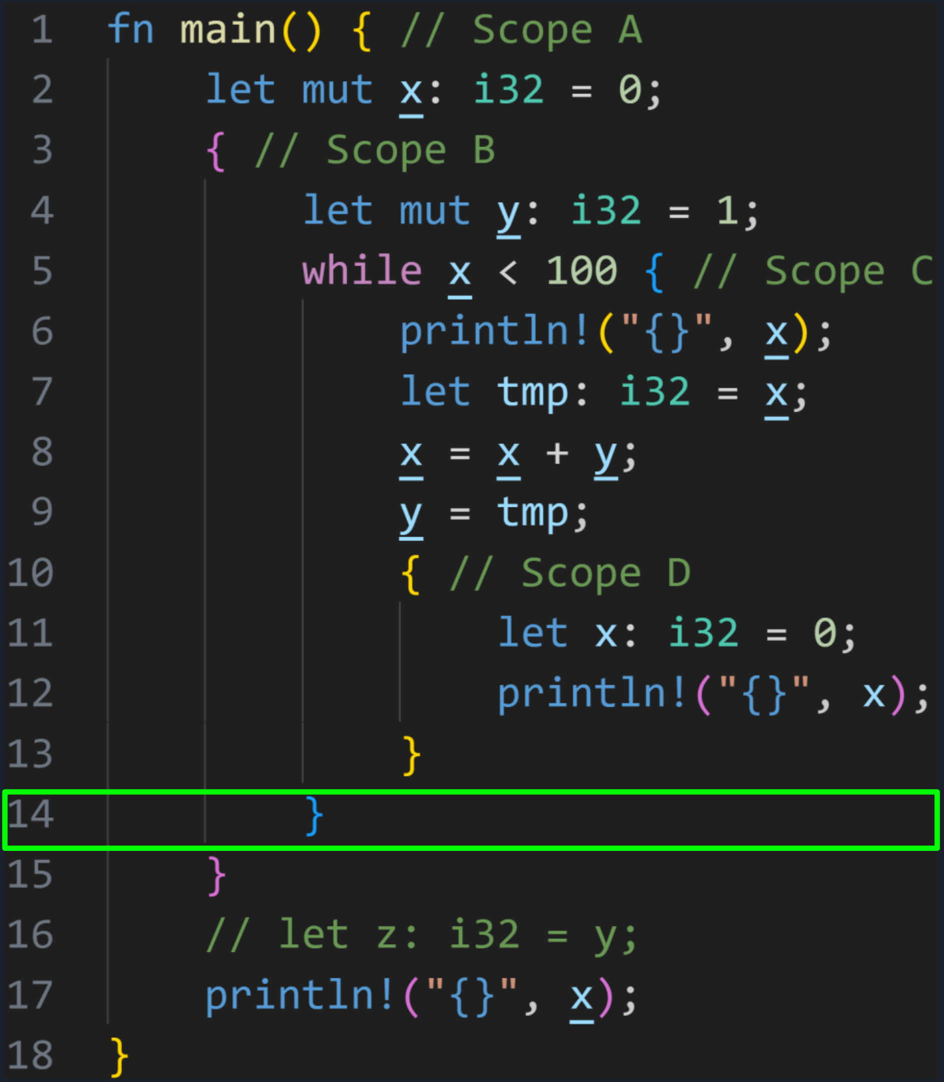
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 0
C	tmp = 0

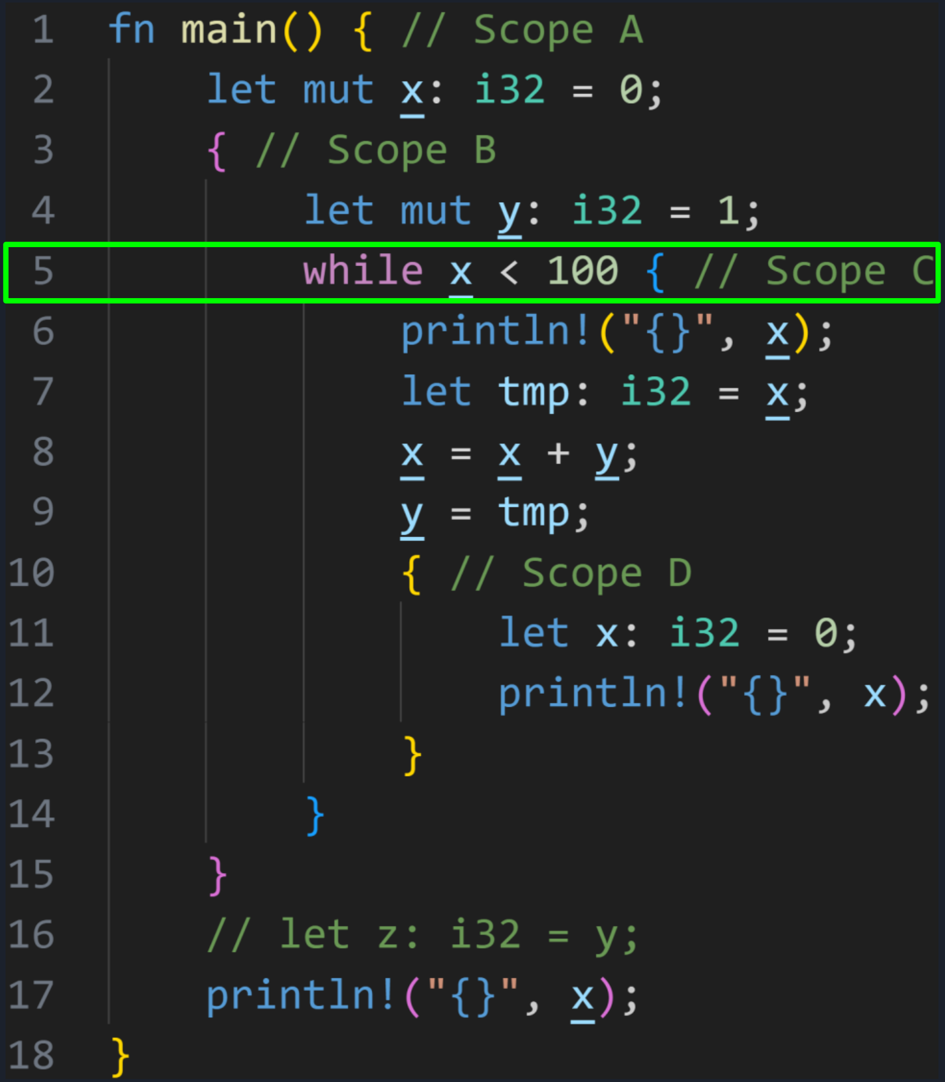
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 0

Intermission - Scopes

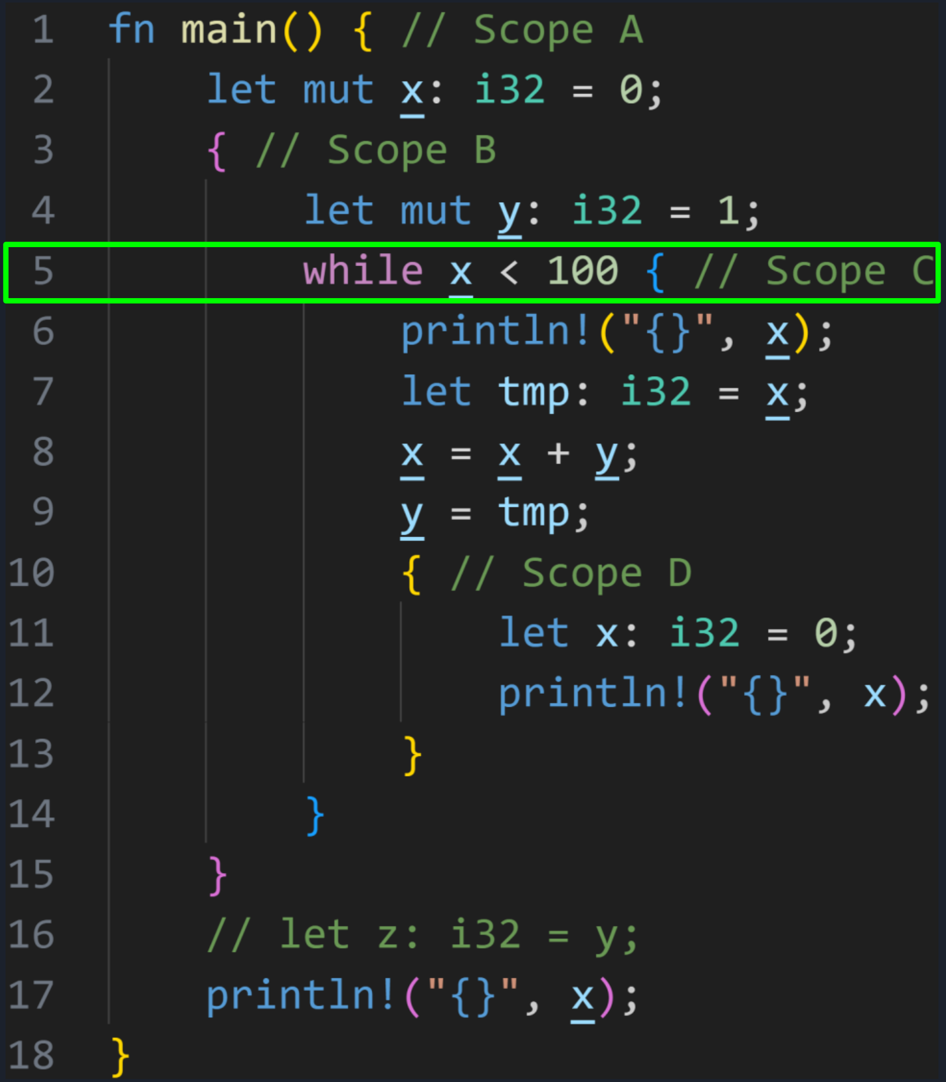


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 0

Check condition again, and repeat until it is no longer true

Intermission - Scopes

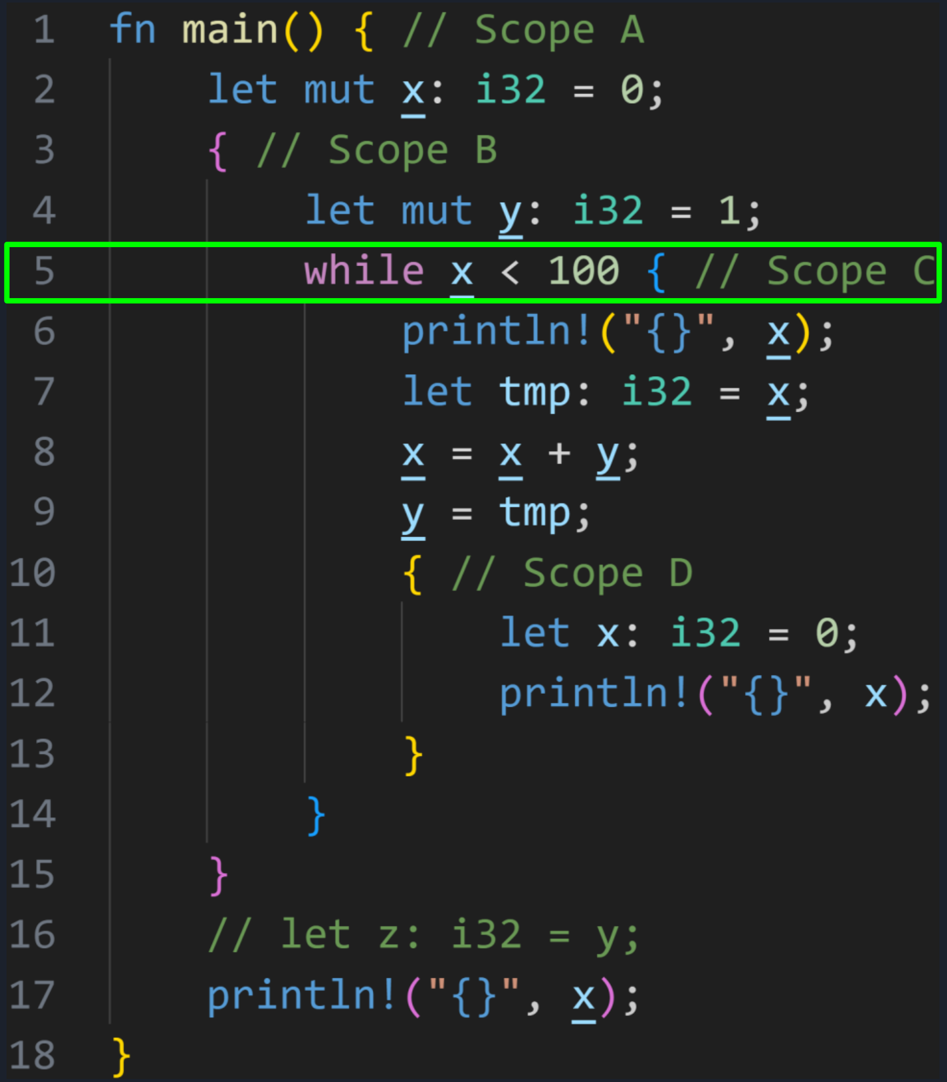


```
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 1
B	y = 1

Check condition again, and repeat until it is no longer true

Intermission - Scopes

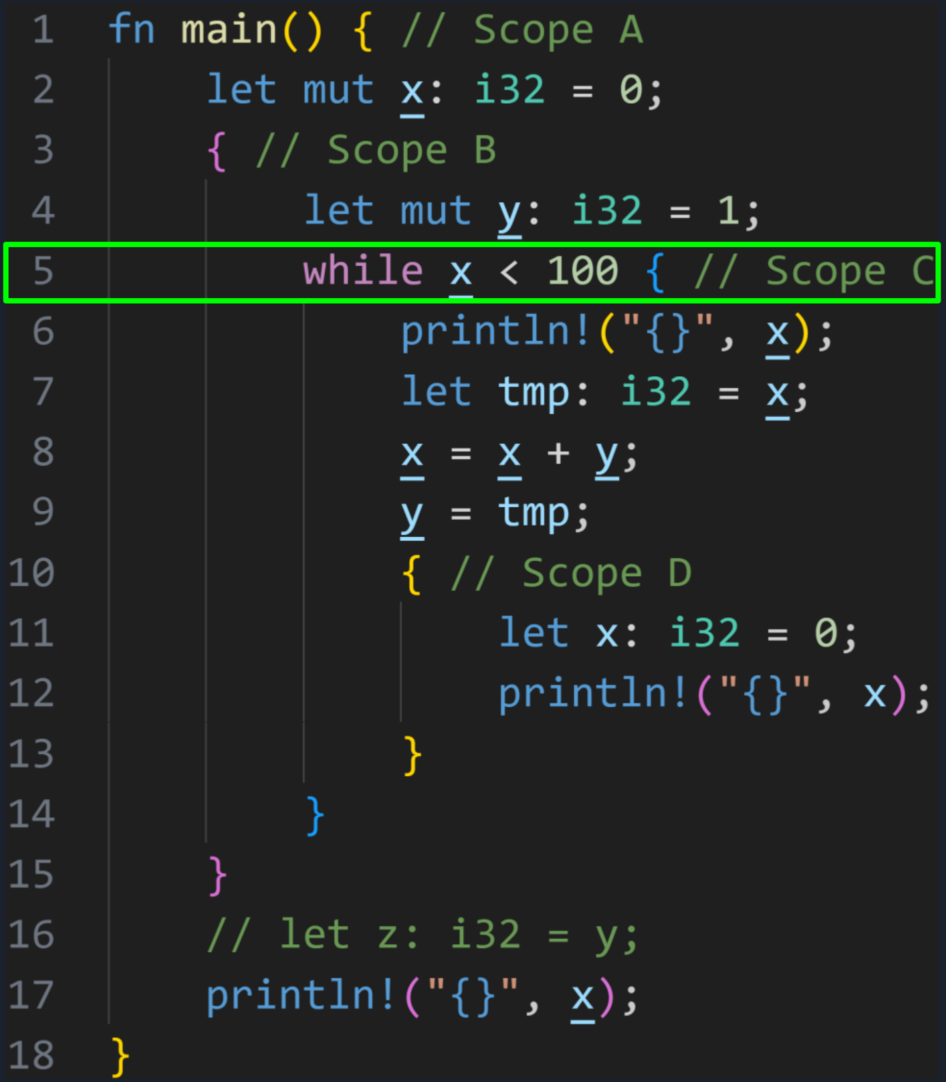


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 2
B	y = 1

Check condition again, and repeat until it is no longer true

Intermission - Scopes

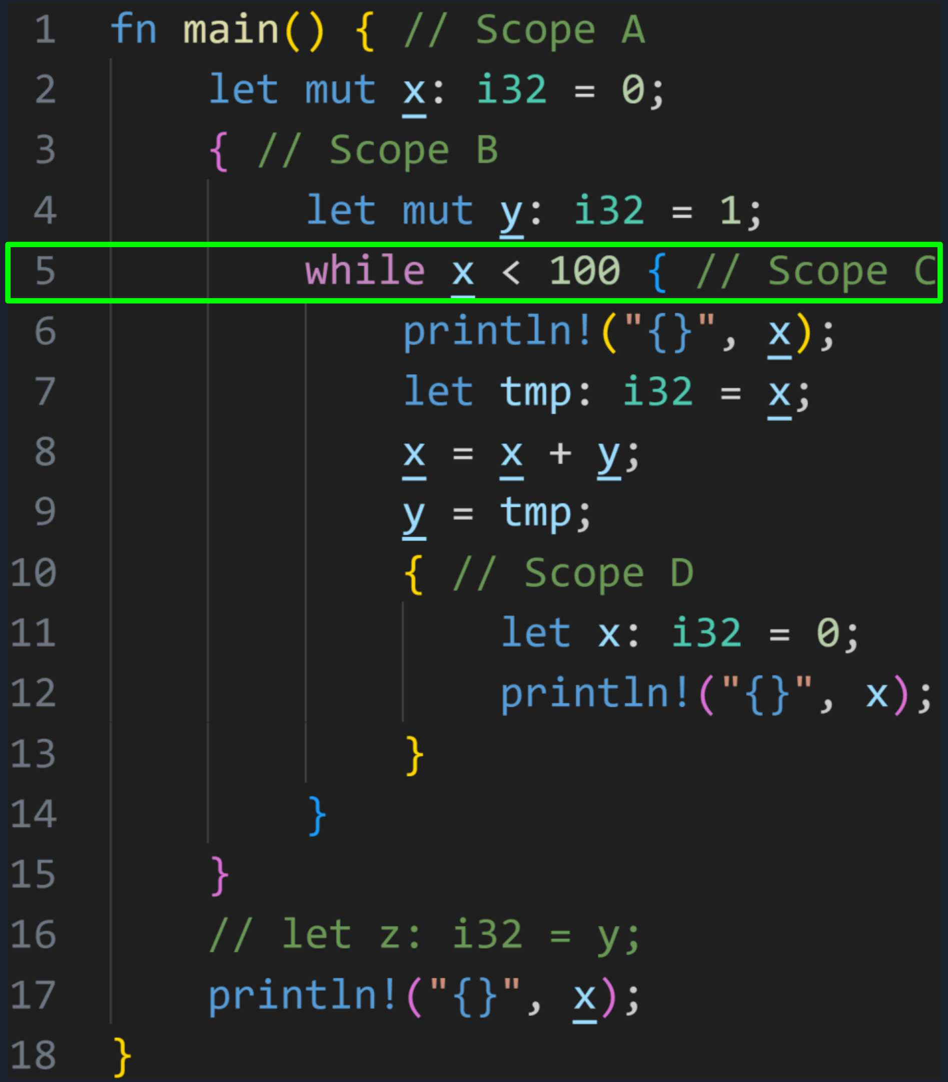


```
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 3
B	y = 2

Check condition again, and repeat until it is no longer true

Intermission - Scopes

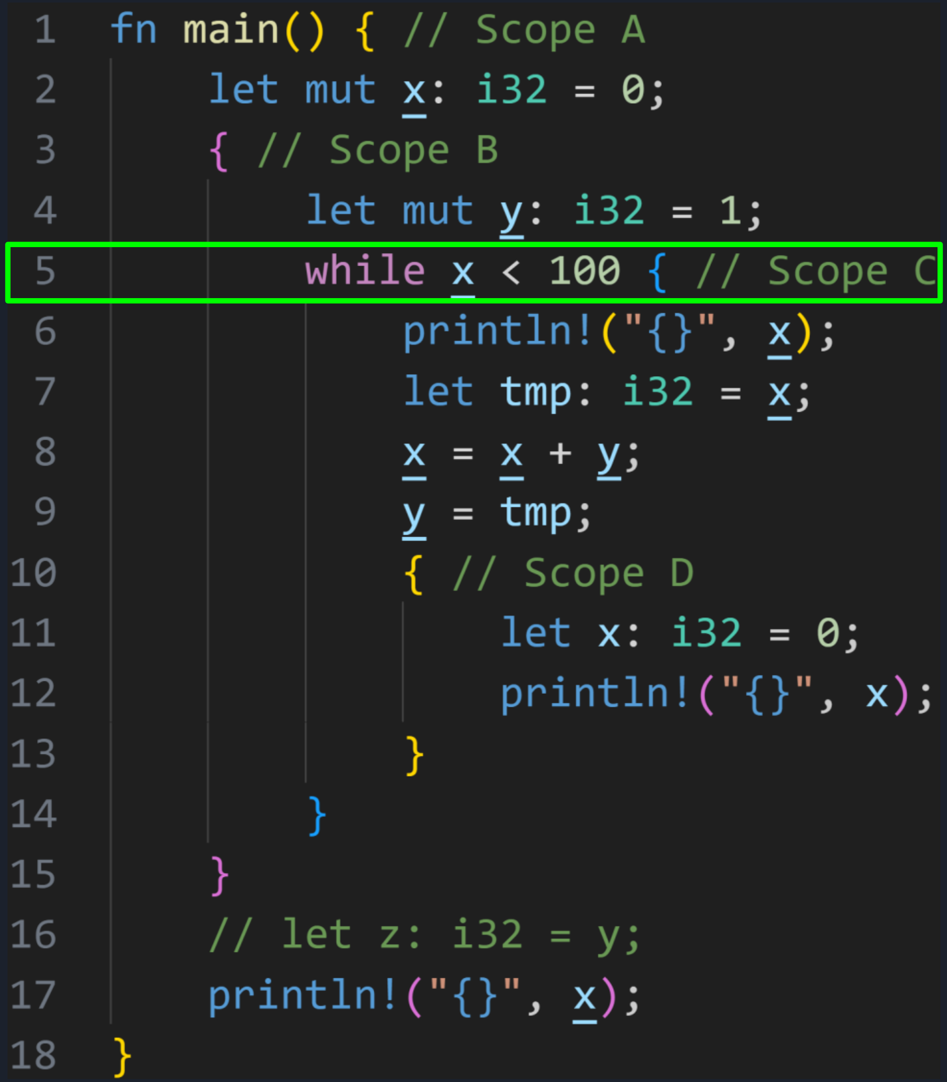


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 5
B	y = 3

Check condition again, and repeat until it is no longer true

Intermission - Scopes

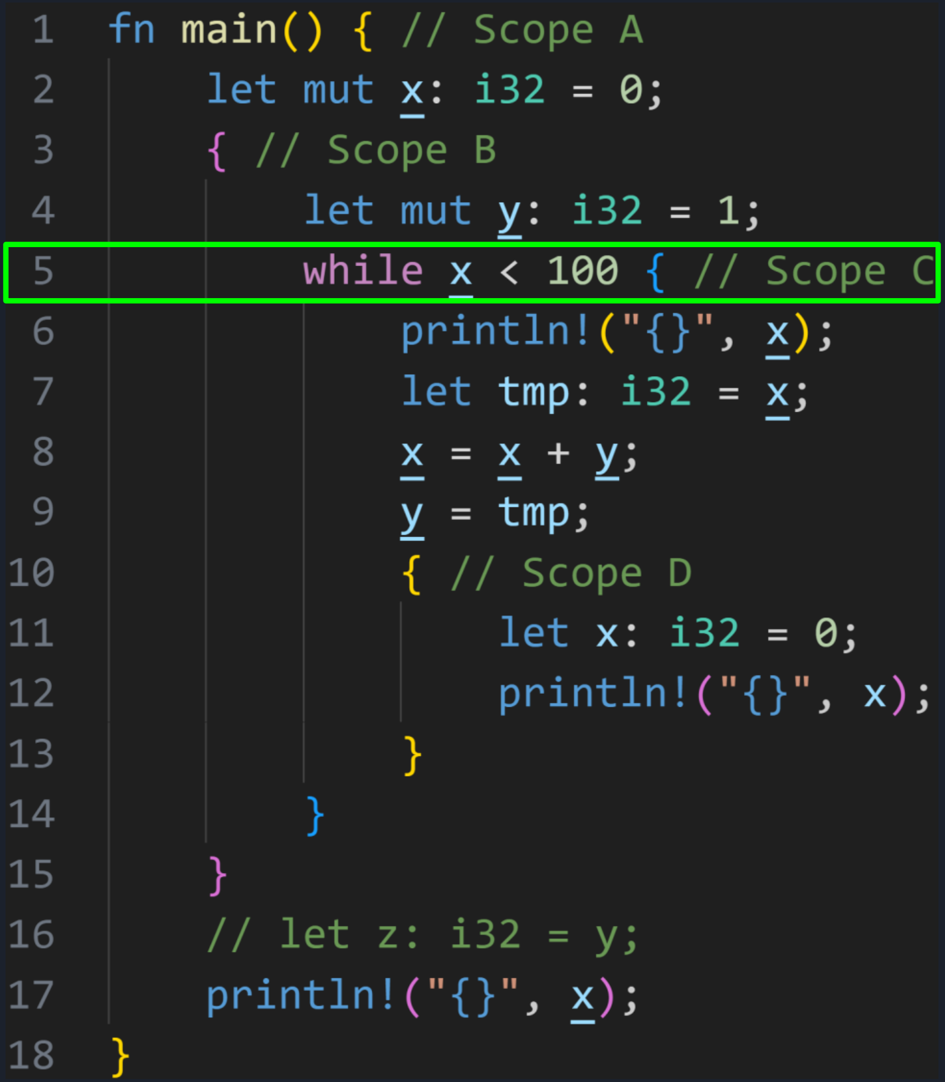


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 8
B	y = 5

Check condition again, and repeat until it is no longer true

Intermission - Scopes

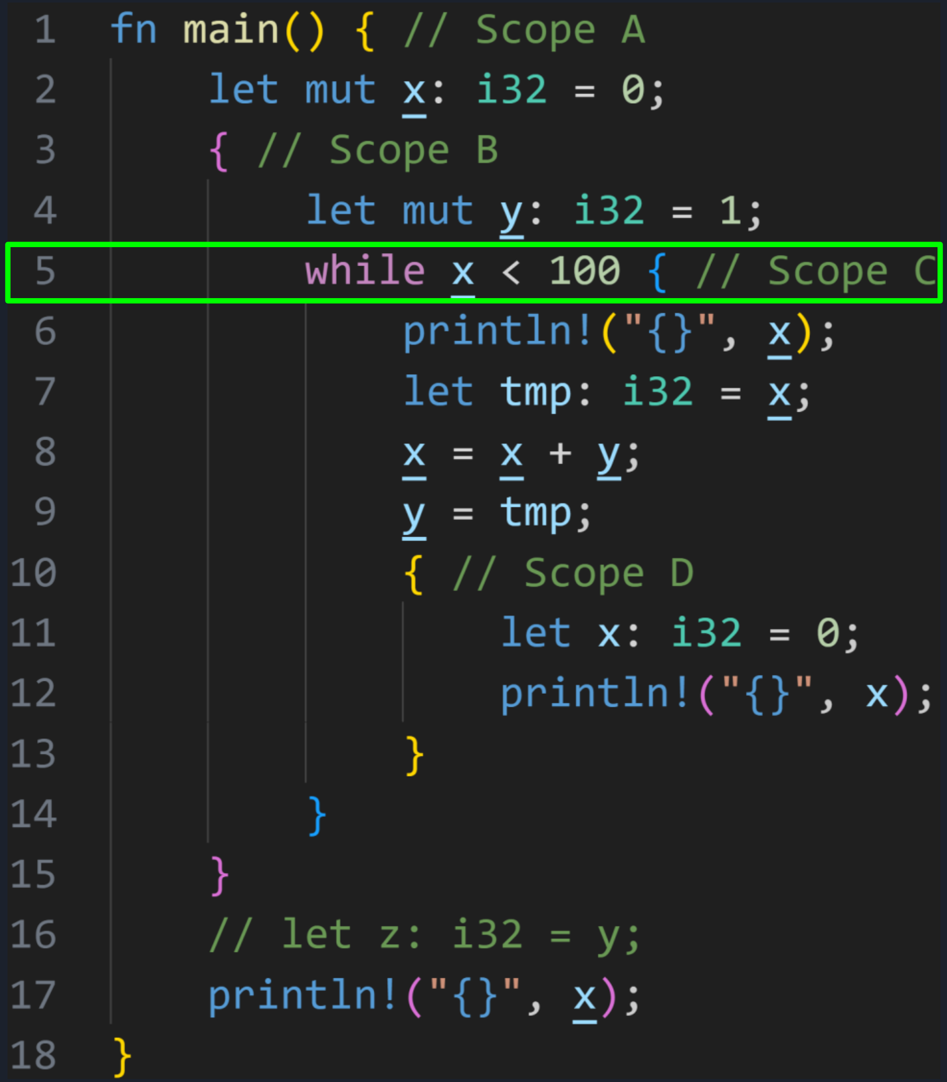


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 13
B	y = 8

Check condition again, and repeat until it is no longer true

Intermission - Scopes

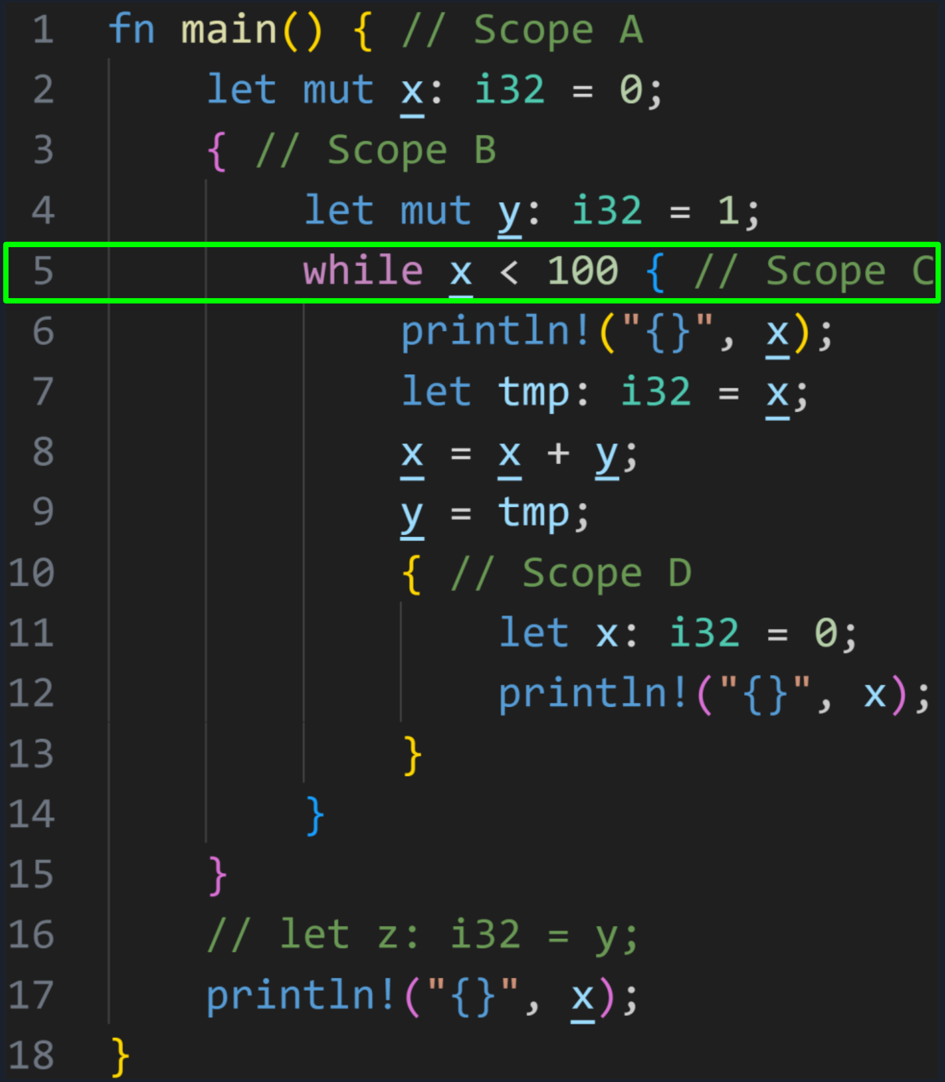


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 89
B	y = 55

Check condition again, and repeat until it is no longer true
Eventually...

Intermission - Scopes

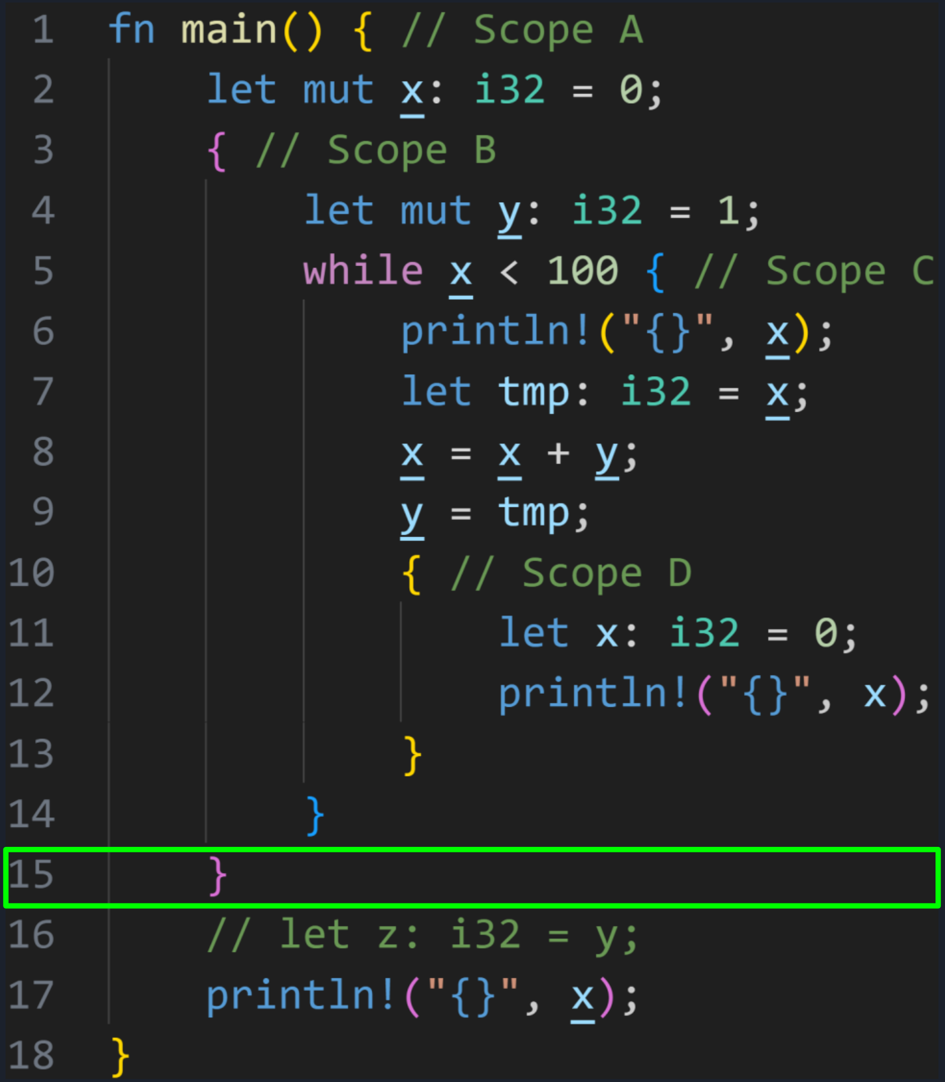


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 144
B	y = 89

Check condition again, and repeat until it is no longer true
Eventually...
144 is not smaller than 100!

Intermission - Scopes

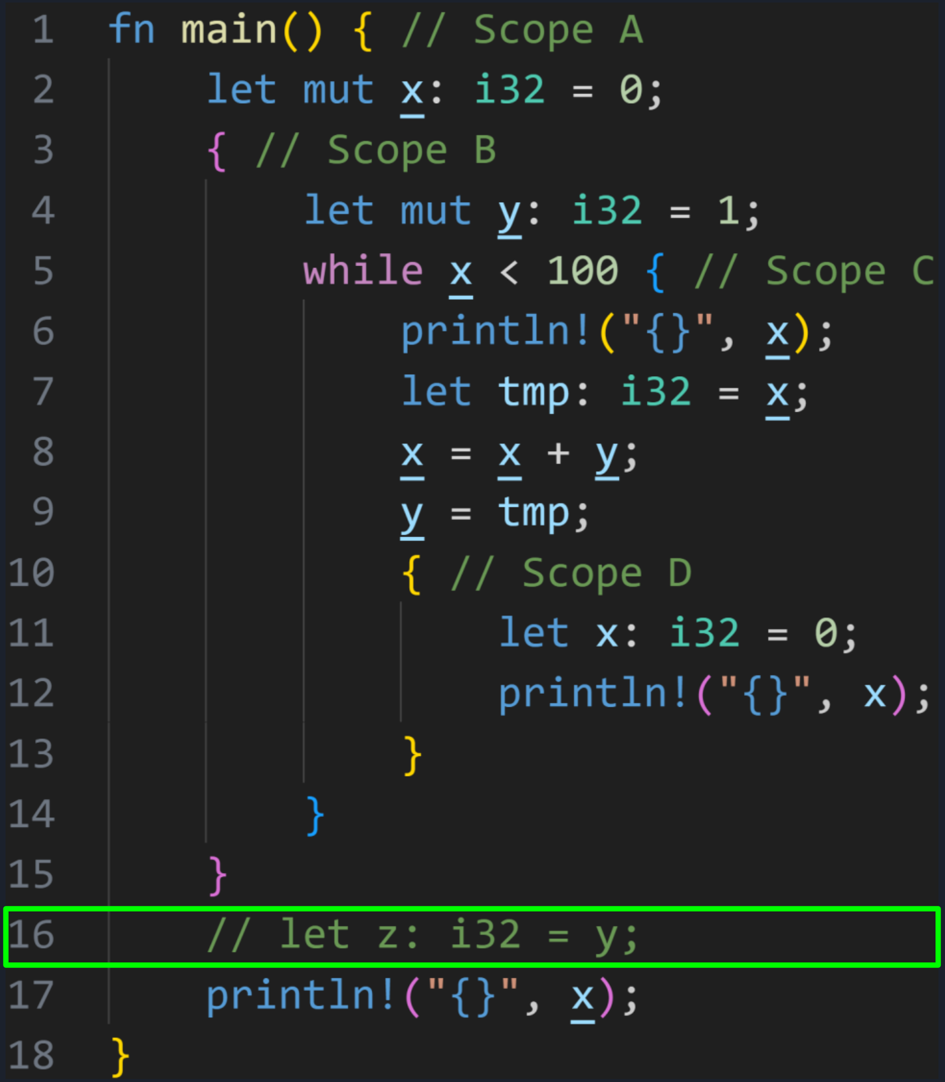


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Scope	Variables
A	x = 144

Check condition again, and repeat until it is no longer true
Eventually...
144 is not smaller than 100!
And we continue.

Intermission - Scopes

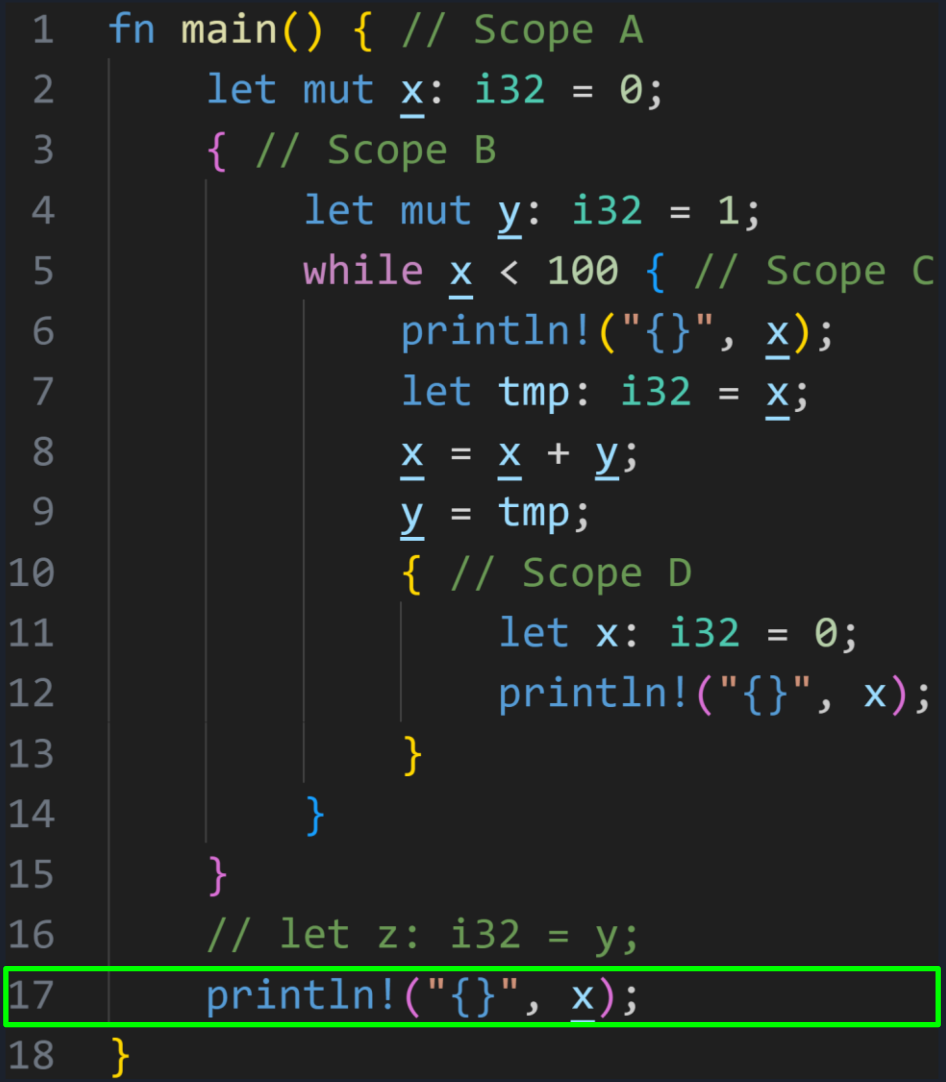


```
1  fn main() { // Scope A
2      let mut x: i32 = 0;
3      { // Scope B
4          let mut y: i32 = 1;
5          while x < 100 { // Scope C
6              println!("{}", x);
7              let tmp: i32 = x;
8              x = x + y;
9              y = tmp;
10             { // Scope D
11                 let x: i32 = 0;
12                 println!("{}", x);
13             }
14         }
15     }
16     // let z: i32 = y;
17     println!("{}", x);
18 }
```

Scope	Variables
A	x = 144

The only valid Scope is A, which does not have y
→ Scope B did have y, but it ended in line 15
→ Scope B is not a previous scope of A :^)
→ This would be an error

Intermission - Scopes

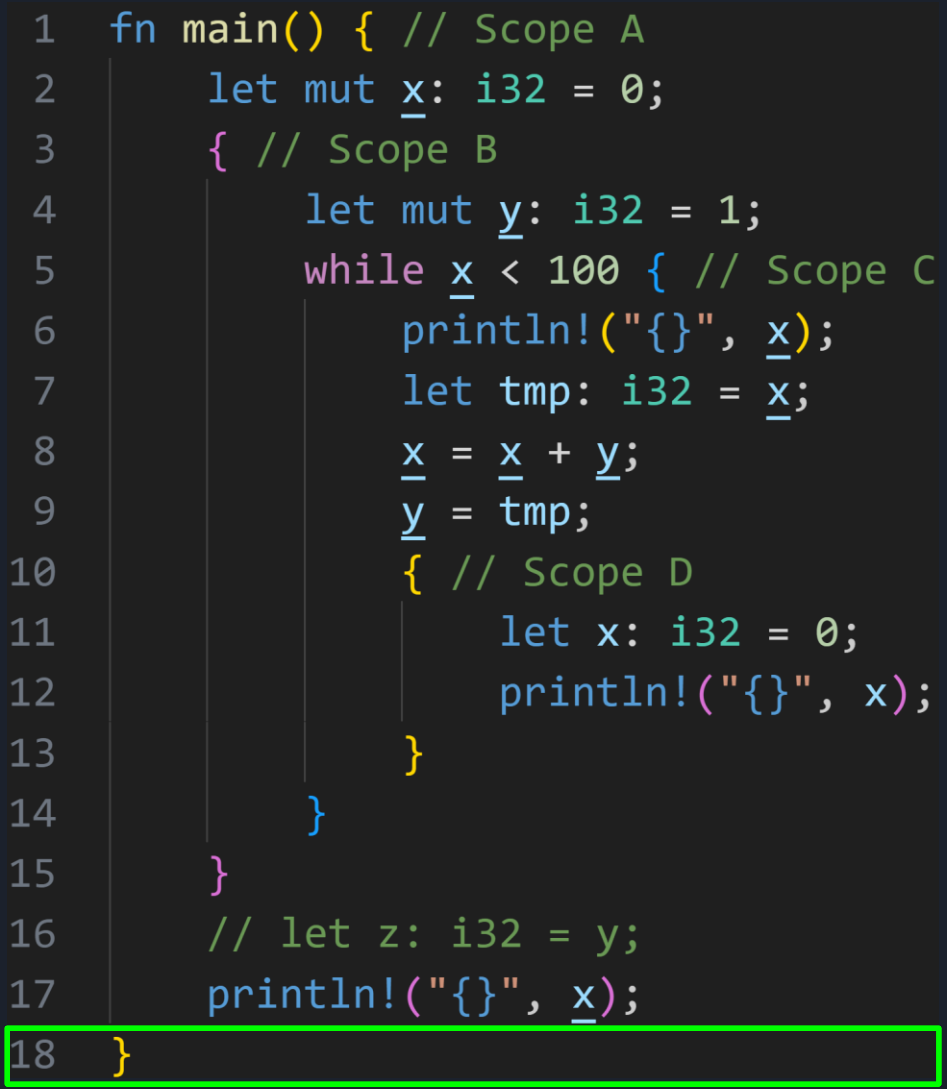


```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Prints 144

Scope	Variables
A	x = 144

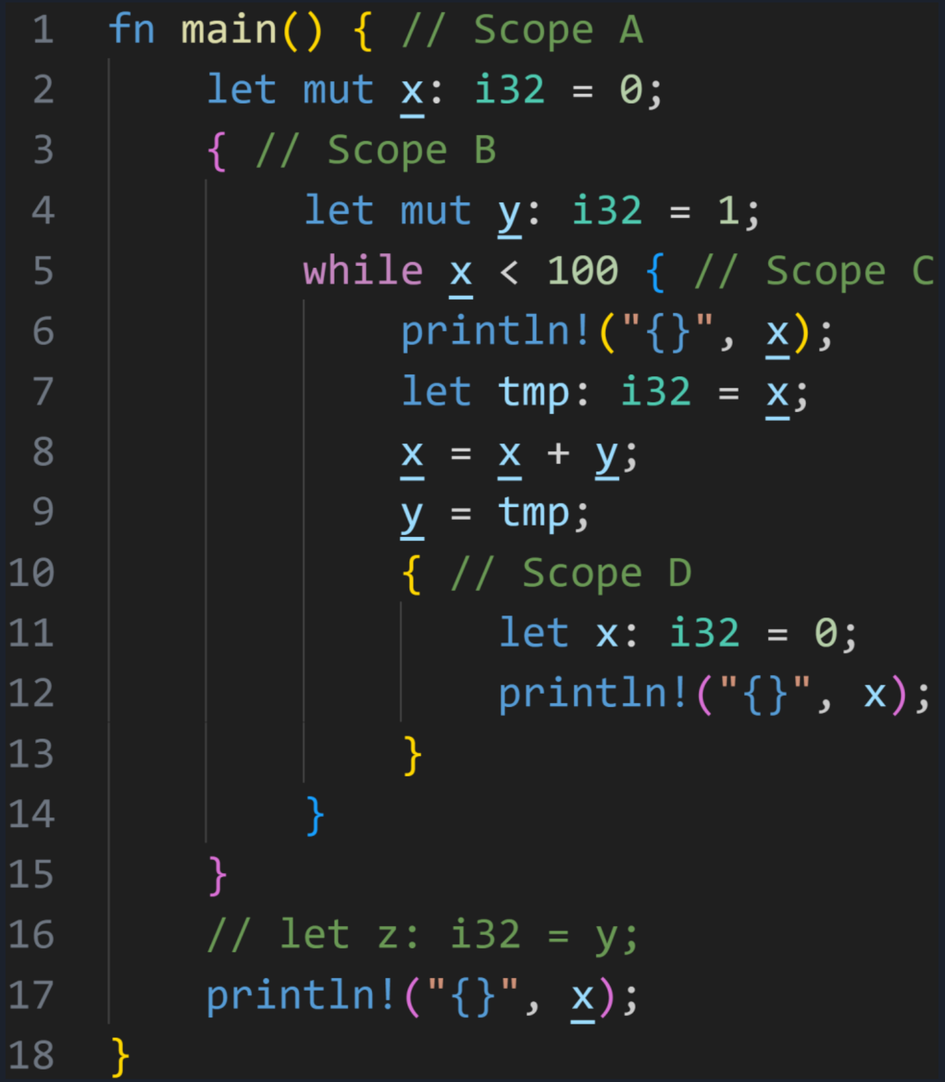
Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

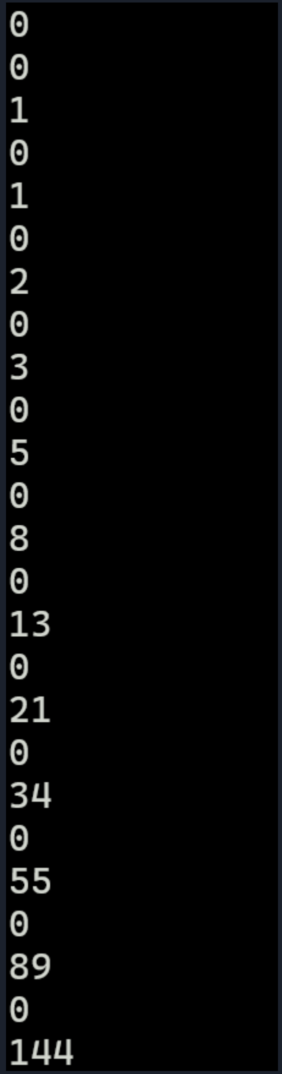
Scope	Variables

Intermission - Scopes



```rust
fn main() { // Scope A
    let mut x: i32 = 0;
    { // Scope B
        let mut y: i32 = 1;
        while x < 100 { // Scope C
            println!("{}", x);
            let tmp: i32 = x;
            x = x + y;
            y = tmp;
            { // Scope D
                let x: i32 = 0;
                println!("{}", x);
            }
        }
    }
    // let z: i32 = y;
    println!("{}", x);
}
```

Output:



```
0
0
1
0
1
0
2
0
3
0
5
0
8
0
13
0
21
0
34
0
55
0
89
0
144
```

2. Loops - for

- for is a very powerful loop

2. Loops - for

- for is a very powerful loop
- It allows you to comfortably loop over data collections

2. Loops - for

- for is a very powerful loop
- It allows you to comfortably loop over data collections
- However, where power is involved, rules are necessary

2. Loops - for

- for is a very powerful loop
- It allows you to comfortably loop over data collections
- However, where power is involved, rules are necessary
  - To be able to use for on a collection, it needs to implement an Iterator-Trait

2. Loops - for

- for is a very powerful loop
- It allows you to comfortably loop over data collections
- However, where power is involved, rules are necessary
  - To be able to use for on a collection, it needs to implement an Iterator-Trait
  - for implicitly turns your collections into iterators

2. Loops - for

- for is a very powerful loop
- It allows you to comfortably loop over data collections
- However, where power is involved, rules are necessary
  - To be able to use for on a collection, it needs to implement an Iterator-Trait
  - for implicitly turns your collections into iterators
  - If you‘re using the original collection, it gets moved and you can‘t use it anymore
    - You have to borrow the collection to prevent that

2. Loops - for

- for is a very powerful loop
- It allows you to comfortably loop over data collections
- However, where power is involved, rules are necessary
  - To be able to use for on a collection, it needs to implement an Iterator-Trait
  - for implicitly turns your collections into iterators
  - If you‘re using the original collection, it gets moved and you can‘t use it anymore
    - You have to borrow the collection to prevent that
- What this means will be covered next week when we introduce the Ownership Model, for now most examples only work as isolated blocks :^)

2. Loops - for

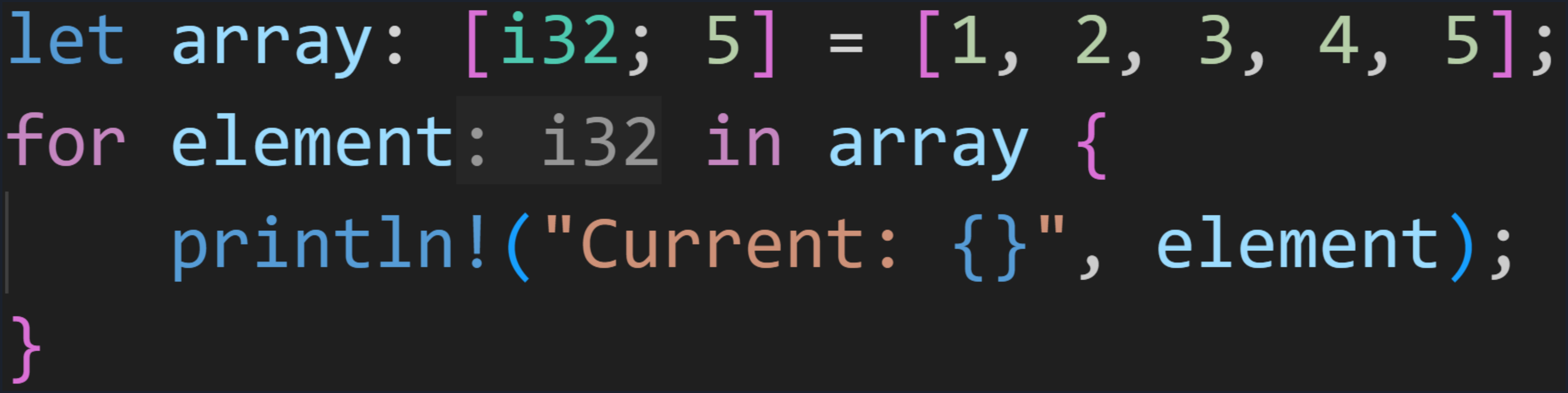
- for is a very powerful loop
- It allows you to comfortably loop over data collections
- However, where power is involved, rules are necessary
  - To be able to use for on a collection, it needs to implement an Iterator-Trait
  - for implicitly turns your collections into iterators
  - If you‘re using the original collection, it gets moved and you can‘t use it anymore
    - You have to borrow the collection to prevent that
- What this means will be covered next week when we introduce the Ownership Model, for now most examples only work as isolated blocks :^)
  - If you feel very brave, you can try and slap & in front of vectors when you use them in a for loop, and see what happens :^)

2. Loops - for

- for is a very powerful loop
- It allows you to comfortably loop over data collections
- However, where power is involved, rules are necessary
  - To be able to use for on a collection, it needs to implement an Iterator-Trait
  - for implicitly turns your collections into iterators
  - If you‘re using the original collection, it gets moved and you can‘t use it anymore
    - You have to borrow the collection to prevent that
- What this means will be covered next week when we introduce the Ownership Model, for now most examples only work as isolated blocks :^)
- Iterators are scary and complex, we‘ll ignore most of it as long as possible and still have fun with our non-idiomatic code :)

2. Loops - for

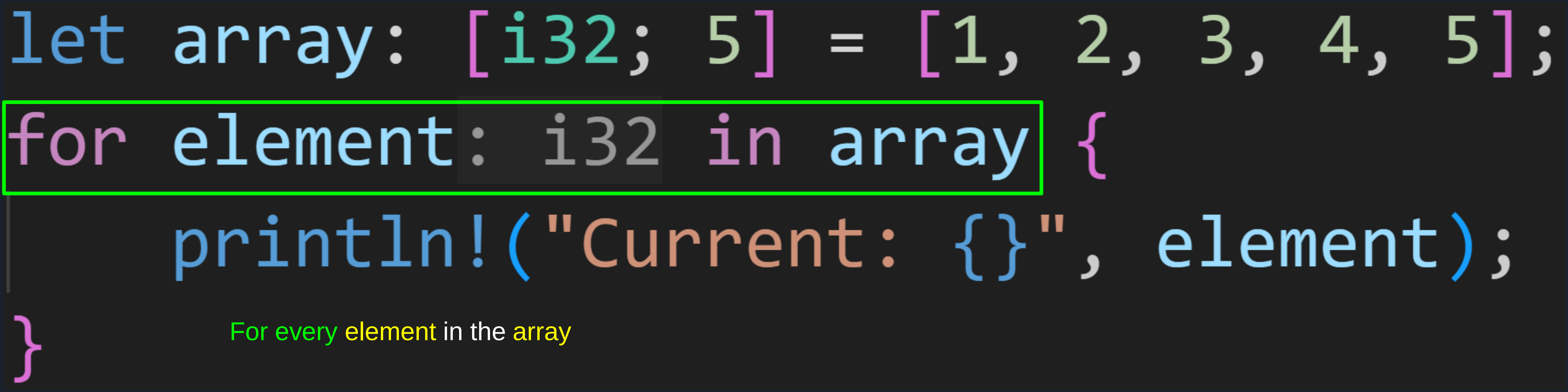
a for loop may look like this



```
let array: [i32; 5] = [1, 2, 3, 4, 5];
for element: i32 in array {
    println!("Current: {}", element);
}
```

2. Loops - for

a for loop may look like this

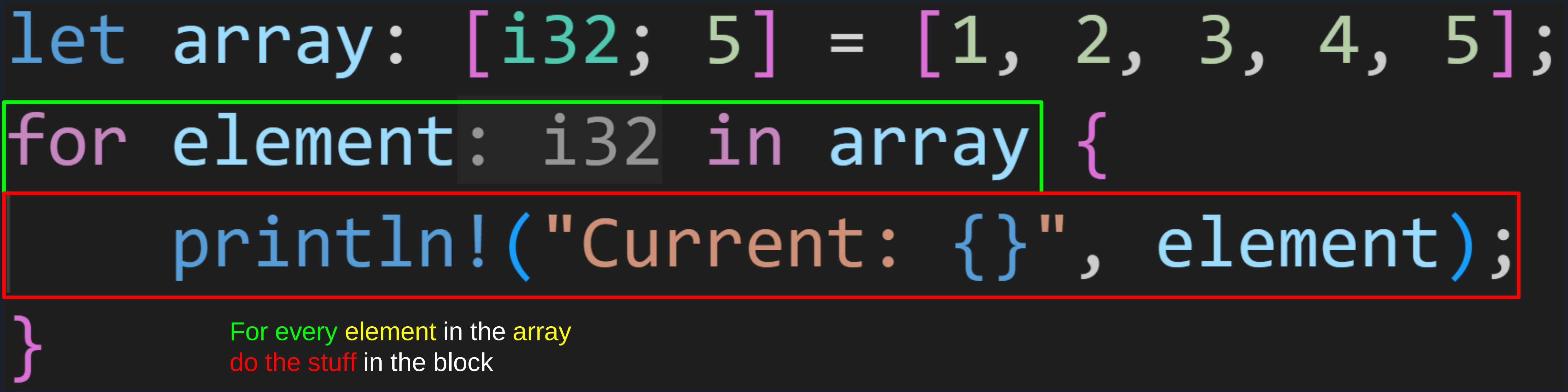


```
let array: [i32; 5] = [1, 2, 3, 4, 5];
for element: i32 in array {
    println!("Current: {}", element);
}
```

For every element in the array

2. Loops - for

a for loop may look like this

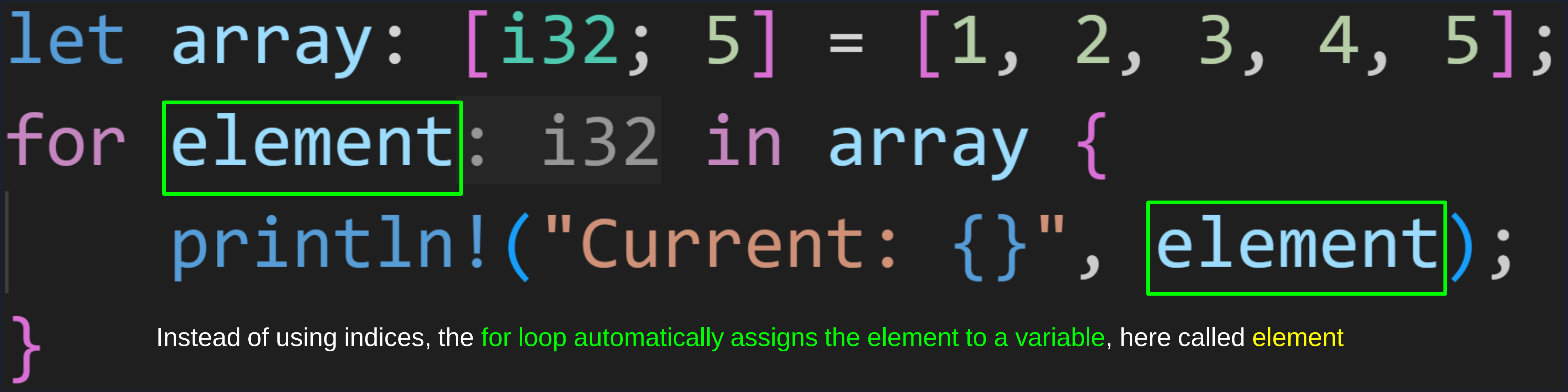


```rust
let array: [i32; 5] = [1, 2, 3, 4, 5];
for element: i32 in array {
    println!("Current: {}", element);
}
```

For every element in the array
do the stuff in the block

2. Loops - for

a for loop may look like this



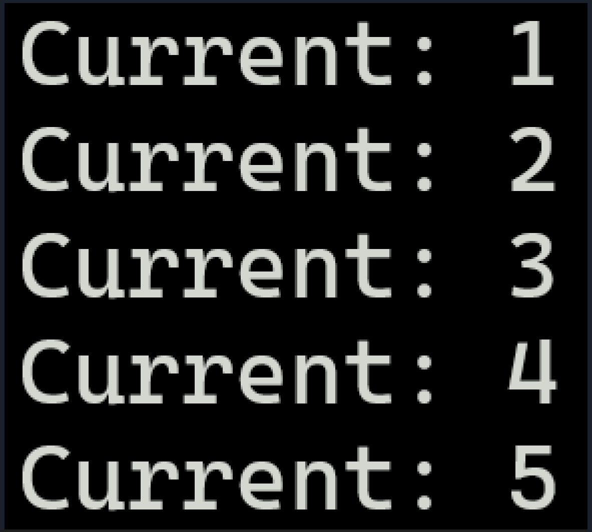
```rust
let array: [i32; 5] = [1, 2, 3, 4, 5];
for element: i32 in array {
    println!("Current: {}", element);
}
```

Instead of using indices, the for loop automatically assigns the element to a variable, here called element

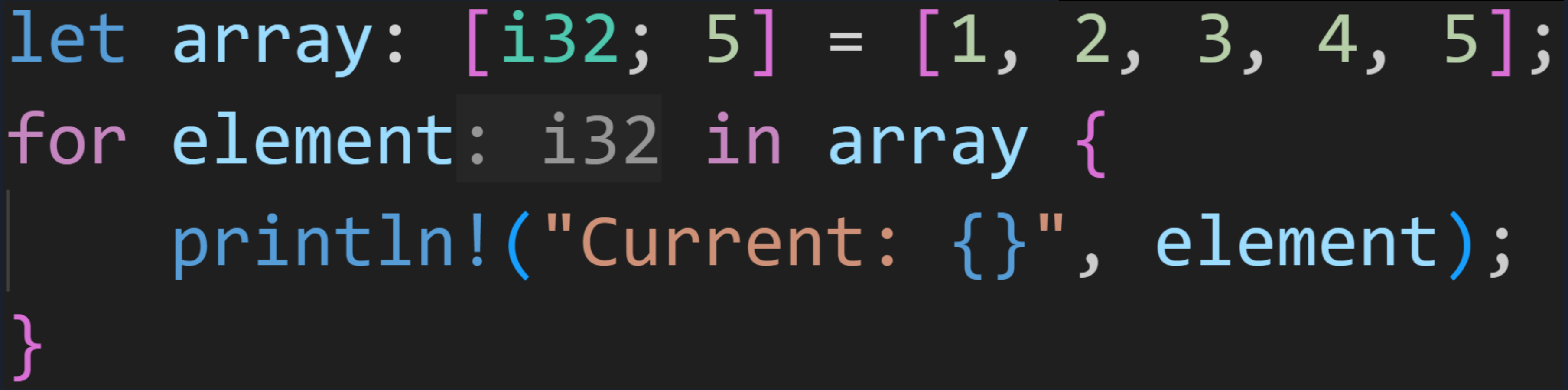
2. Loops - for

a for loop may look like this

Output:



```
Current: 1
Current: 2
Current: 3
Current: 4
Current: 5
```

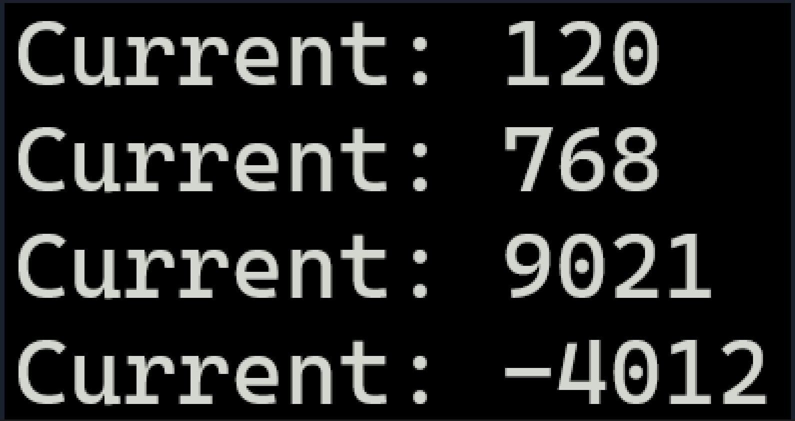


```rust
let array: [i32; 5] = [1, 2, 3, 4, 5];
for element: i32 in array {
    println!("Current: {}", element);
}
```

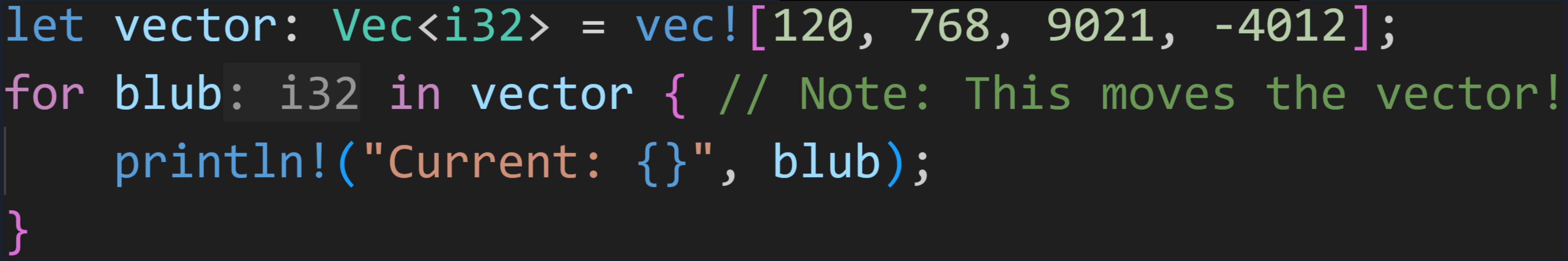
2. Loops - for

a for loop may look like this

Output:



```
Current: 120
Current: 768
Current: 9021
Current: -4012
```

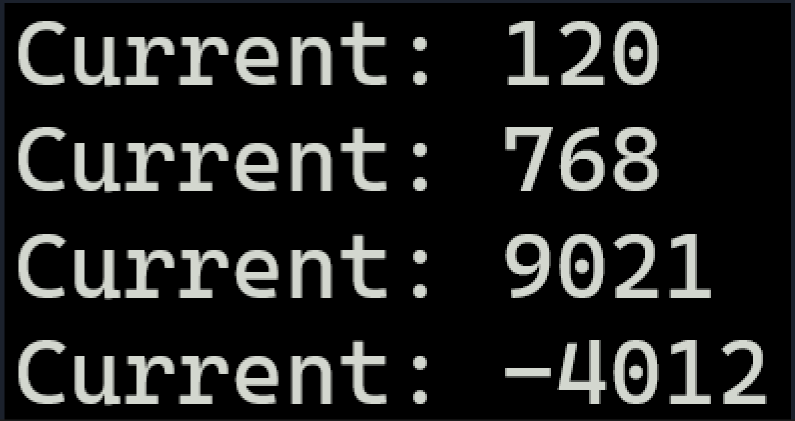


```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for blub: i32 in vector { // Note: This moves the vector!
    println!("Current: {}", blub);
}
```

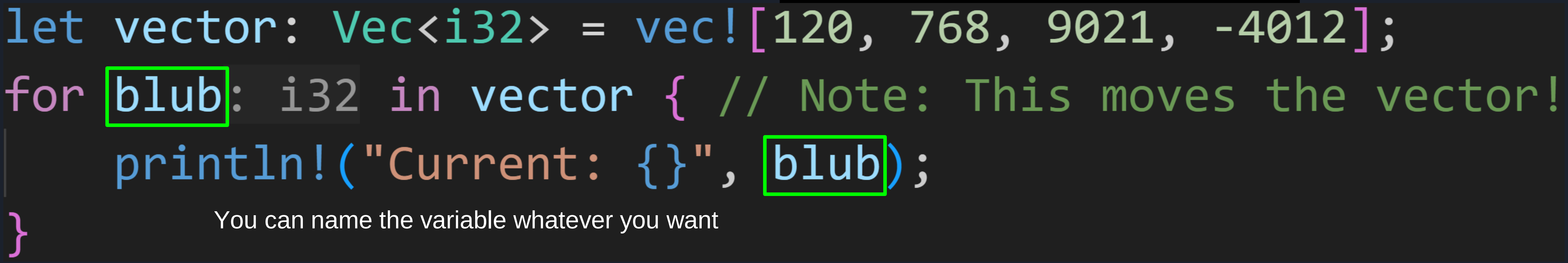
2. Loops - for

a for loop may look like this

Output:



```
Current: 120
Current: 768
Current: 9021
Current: -4012
```

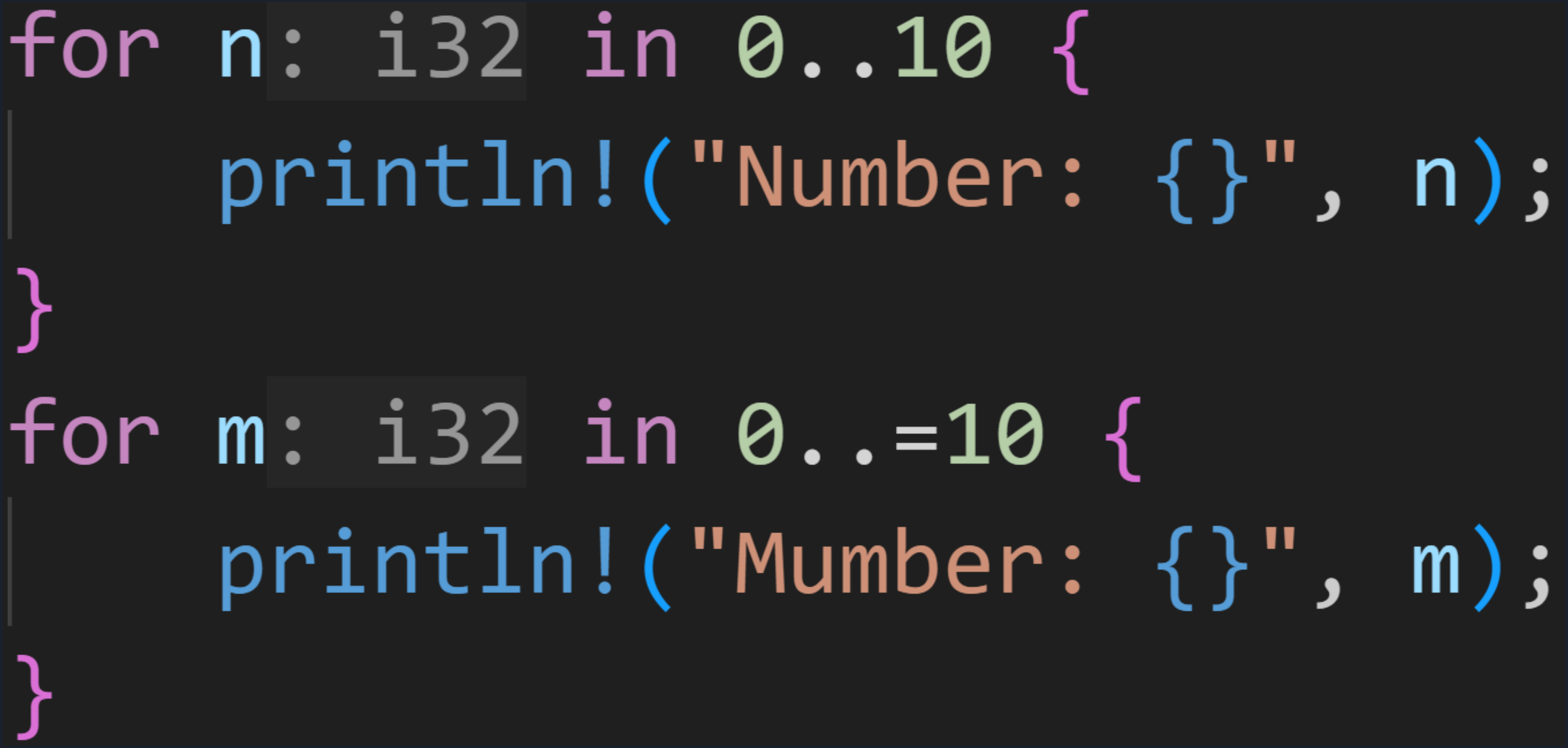


```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for blub: i32 in vector { // Note: This moves the vector!
    println!("Current: {}", blub);
}
```

You can name the variable whatever you want

2. Loops - for

a for loop may look like this

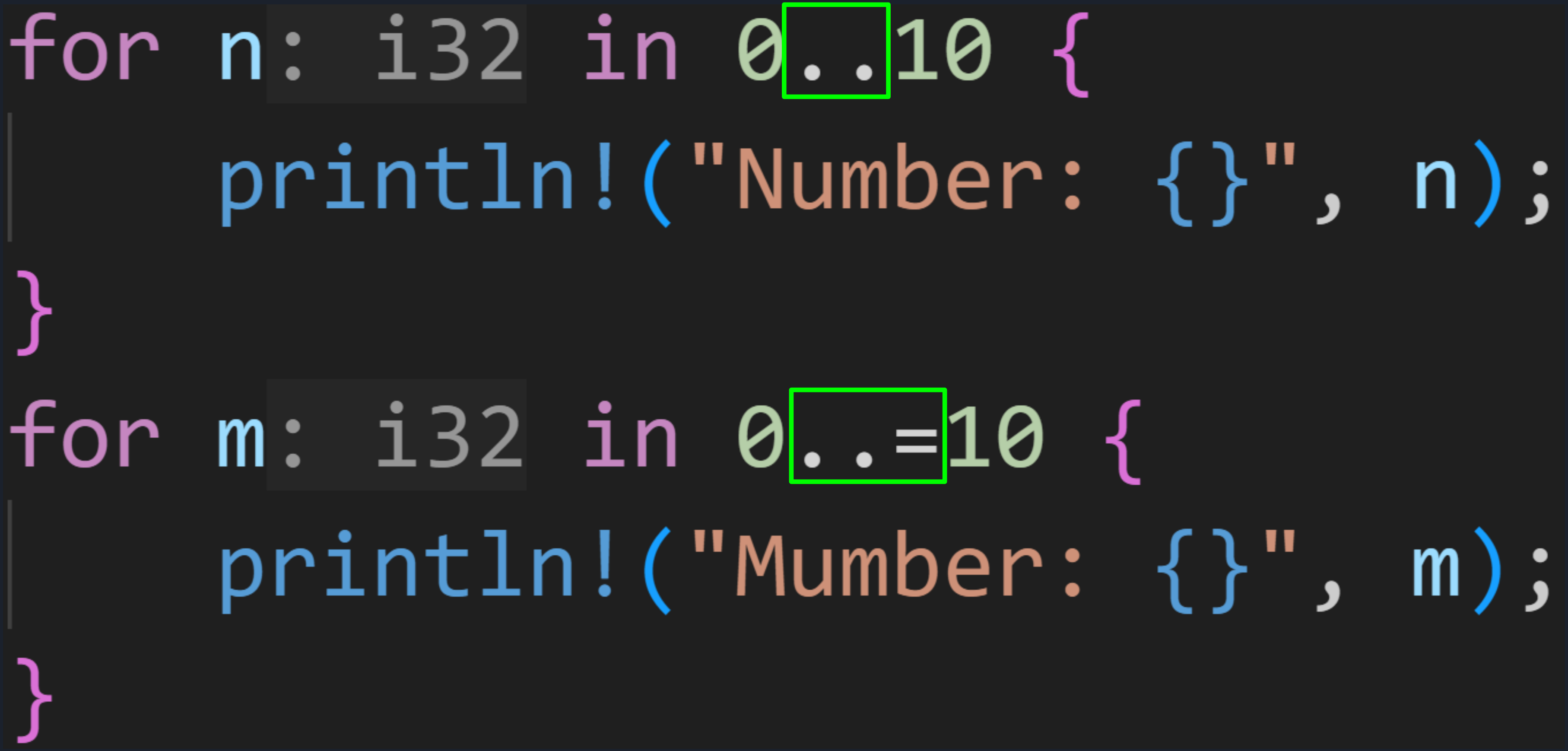


```
for n: i32 in 0..10 {
    println!("Number: {}", n);
}
for m: i32 in 0..=10 {
    println!("Mumber: {}", m);
}
```

2. Loops - for

Range operator
→ Creates a collection of numbers between start and end

a for loop may look like this



```rust
for n: i32 in 0..10 {
    println!("Number: {}", n);
}
for m: i32 in 0..=10 {
    println!("Mumber: {}", m);
}
```

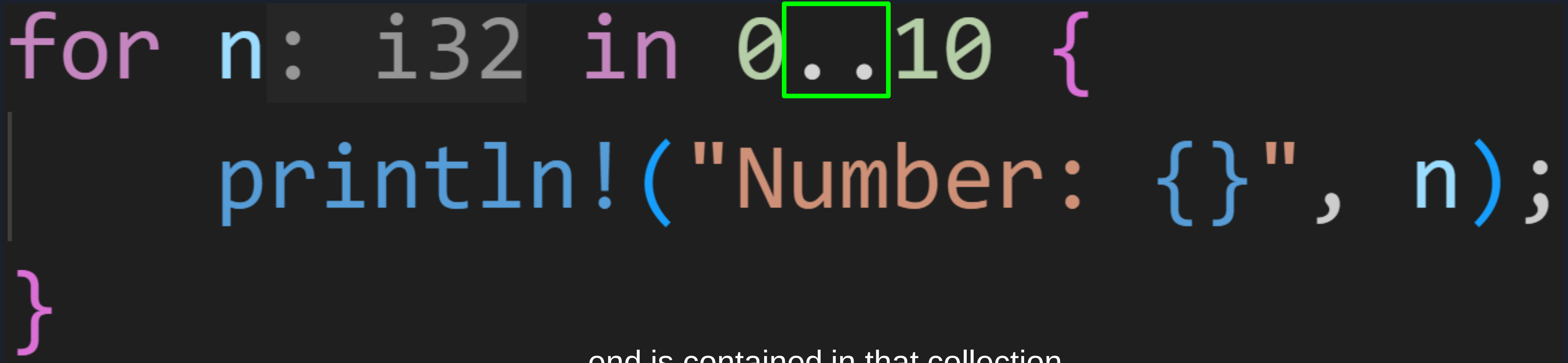
2. Loops - for

Range operator

→ Creates a collection of numbers between start and end

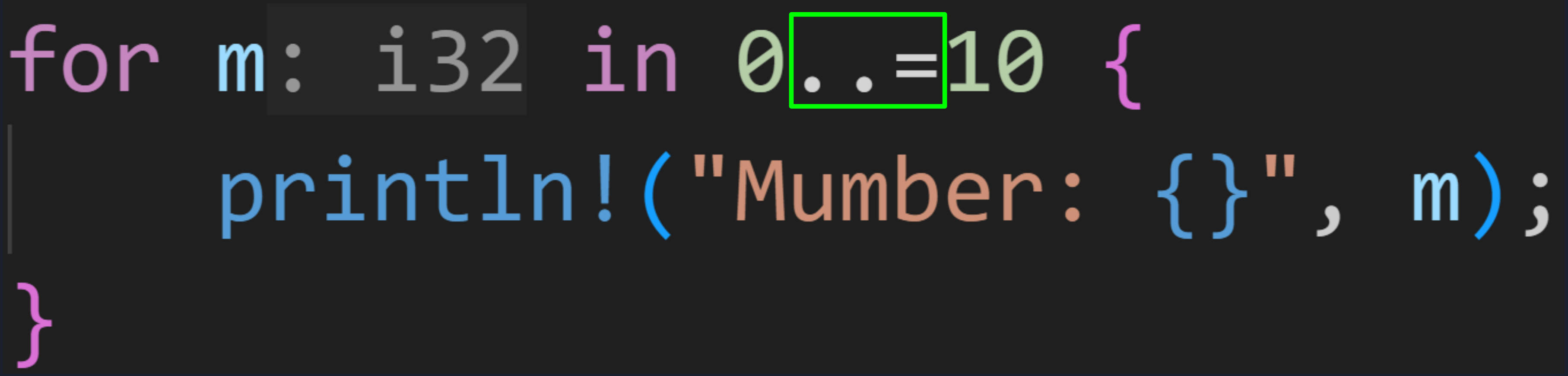
a for loop may look like this

end is NOT contained in that collection



```
for n: i32 in 0..10 {
    println!("Number: {}", n);
}
```

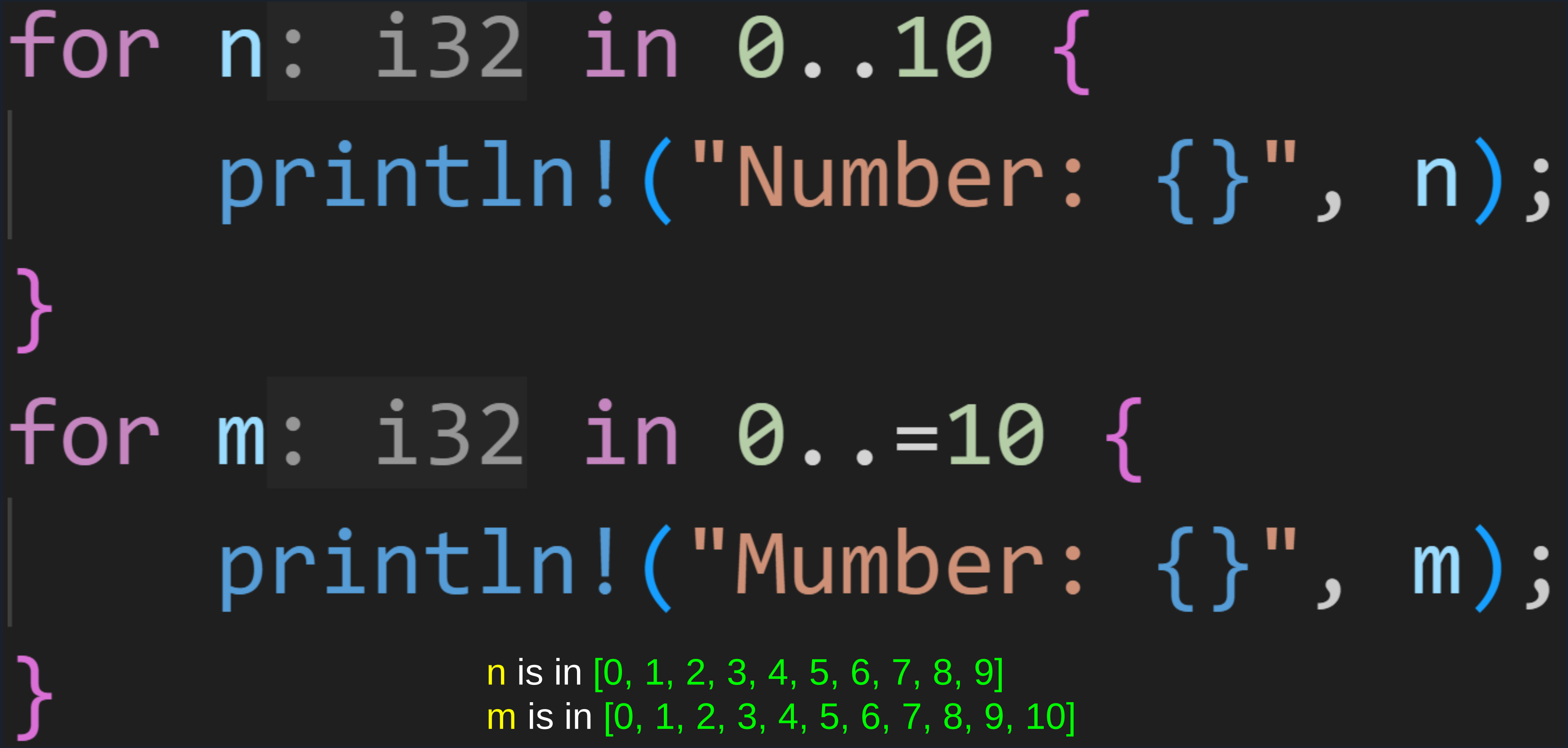
end is contained in that collection



```
for m: i32 in 0..=10 {
    println!("Mumber: {}", m);
}
```

2. Loops - for

a for loop may look like this

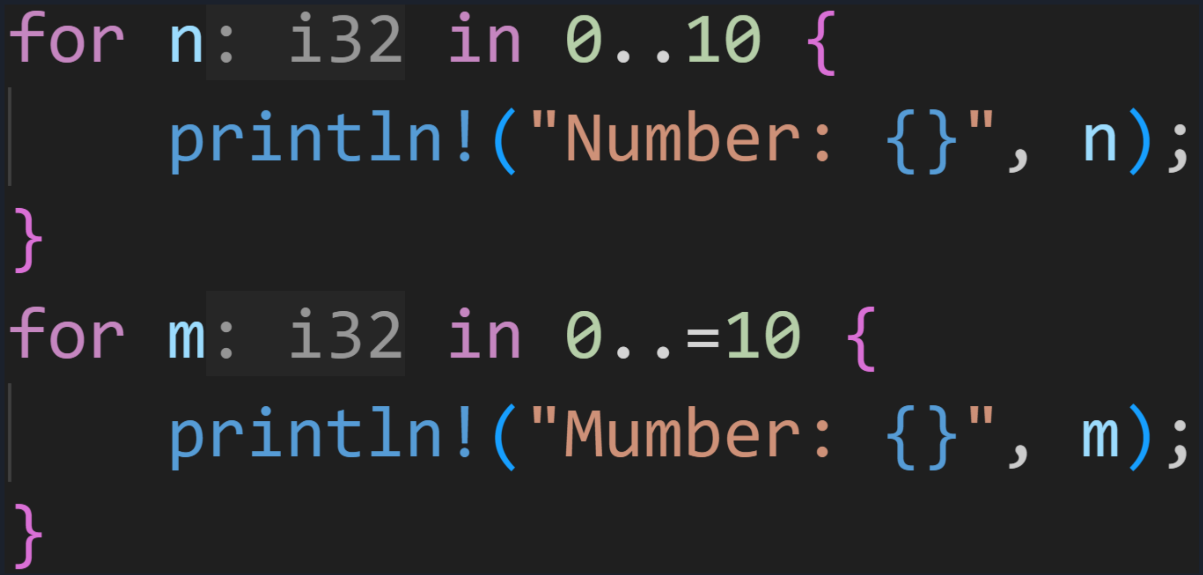


```
for n: i32 in 0..10 {
    println!("Number: {}", n);
}
for m: i32 in 0..=10 {
    println!("Mumber: {}", m);
}
```

n is in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
m is in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

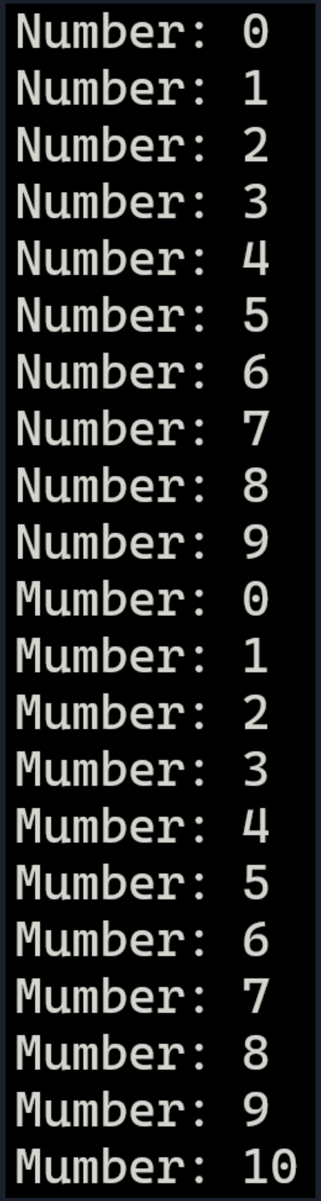
2. Loops - for

a for loop may look like this



```
for n: i32 in 0..10 {
    println!("Number: {}", n);
}
for m: i32 in 0..=10 {
    println!("Mumber: {}", m);
}
```

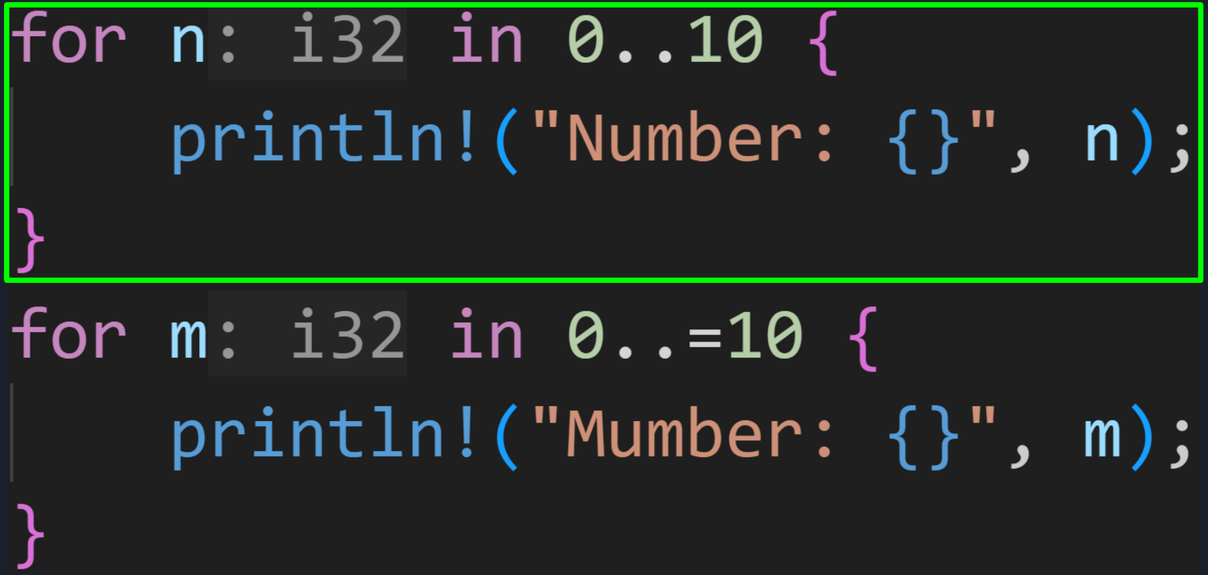
Output:



```
Number:  0
Number:  1
Number:  2
Number:  3
Number:  4
Number:  5
Number:  6
Number:  7
Number:  8
Number:  9
Mumber:  0
Mumber:  1
Mumber:  2
Mumber:  3
Mumber:  4
Mumber:  5
Mumber:  6
Mumber:  7
Mumber:  8
Mumber:  9
Mumber:  10
```

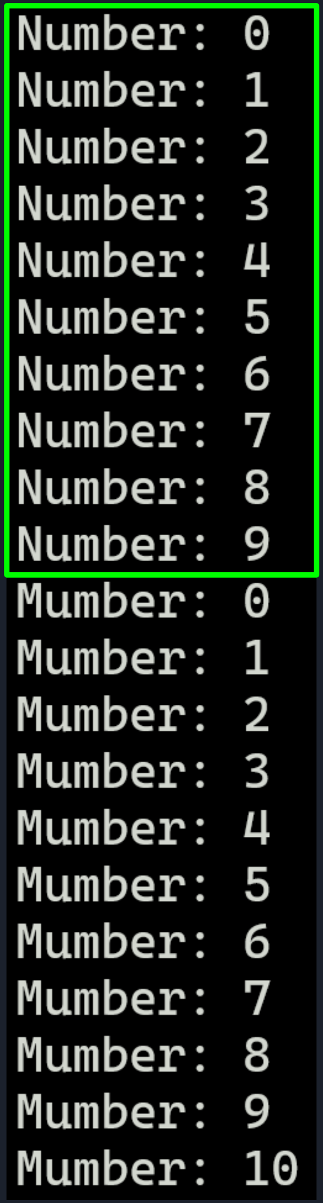
2. Loops - for

a for loop may look like this



```rust
for n: i32 in 0..10 {
    println!("Number: {}", n);
}
for m: i32 in 0..=10 {
    println!("Mumber: {}", m);
}
```

Output:

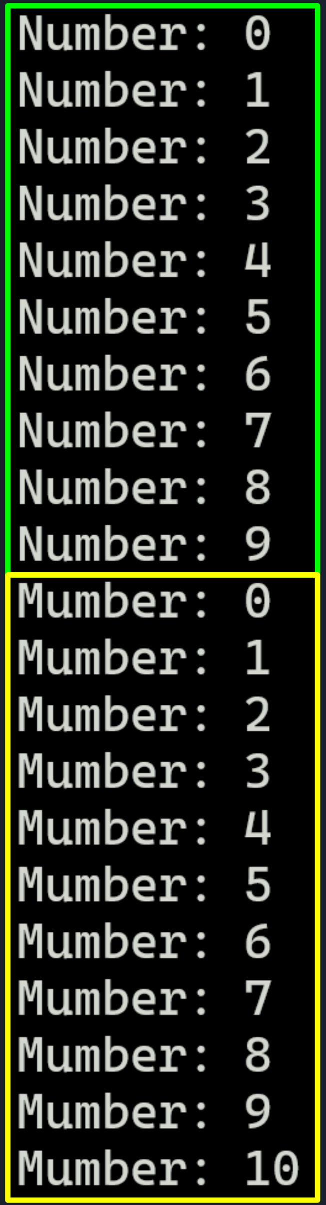


```
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Mumber: 0
Mumber: 1
Mumber: 2
Mumber: 3
Mumber: 4
Mumber: 5
Mumber: 6
Mumber: 7
Mumber: 8
Mumber: 9
Mumber: 10
```

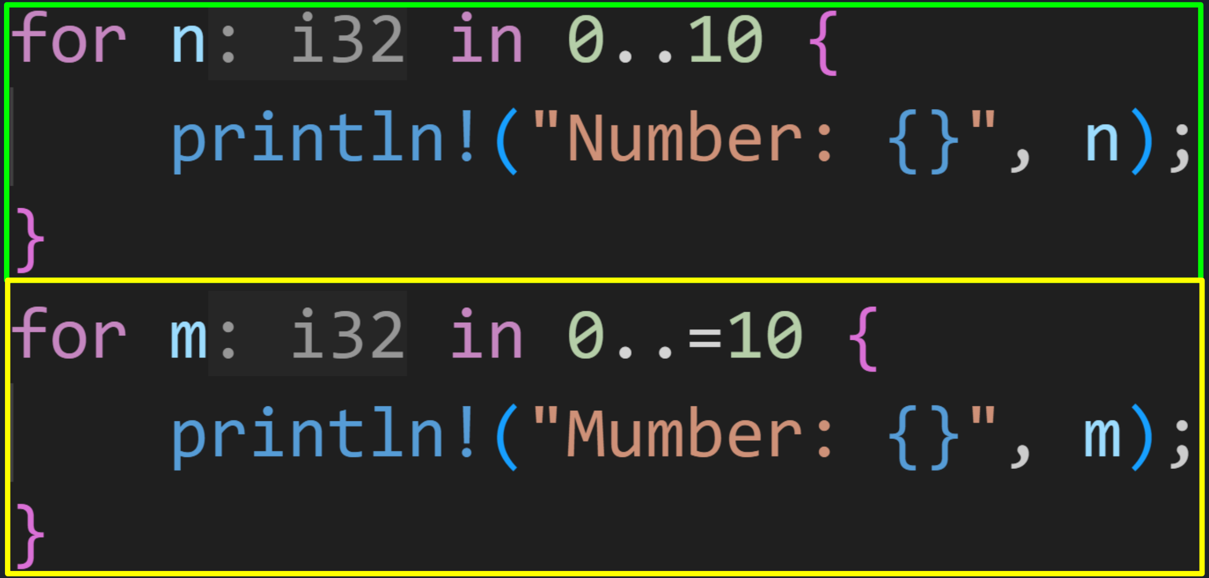
2. Loops - for

a for loop may look like this

Output:



```
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Mumber: 0
Mumber: 1
Mumber: 2
Mumber: 3
Mumber: 4
Mumber: 5
Mumber: 6
Mumber: 7
Mumber: 8
Mumber: 9
Mumber: 10
```



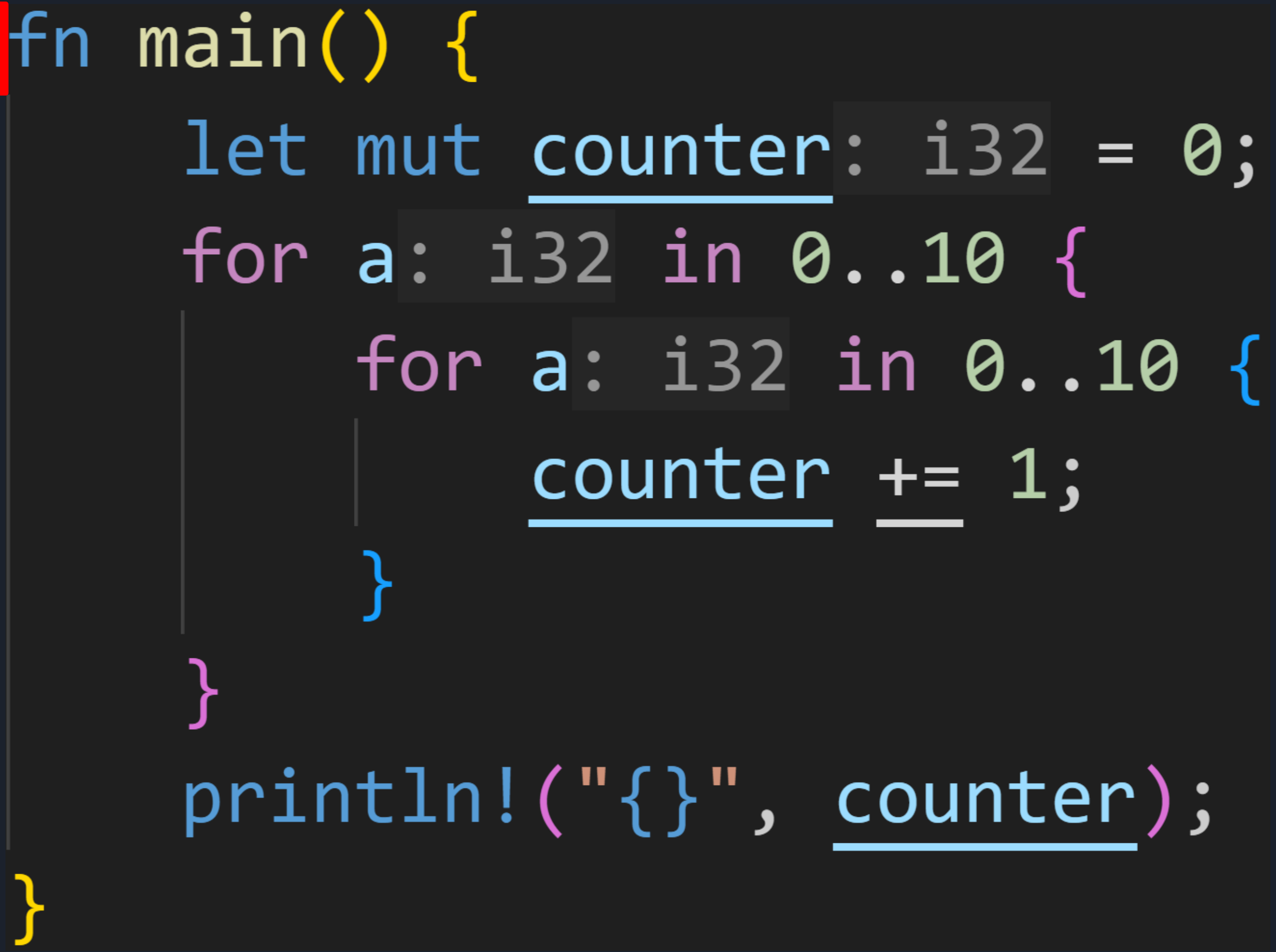
```rust
for n: i32 in 0..10 {
    println!("Number: {}", n);
}
for m: i32 in 0..=10 {
    println!("Mumber: {}", m);
}
```

Intermission - Exercises

- Time for exercises!

Intermission - Exercises

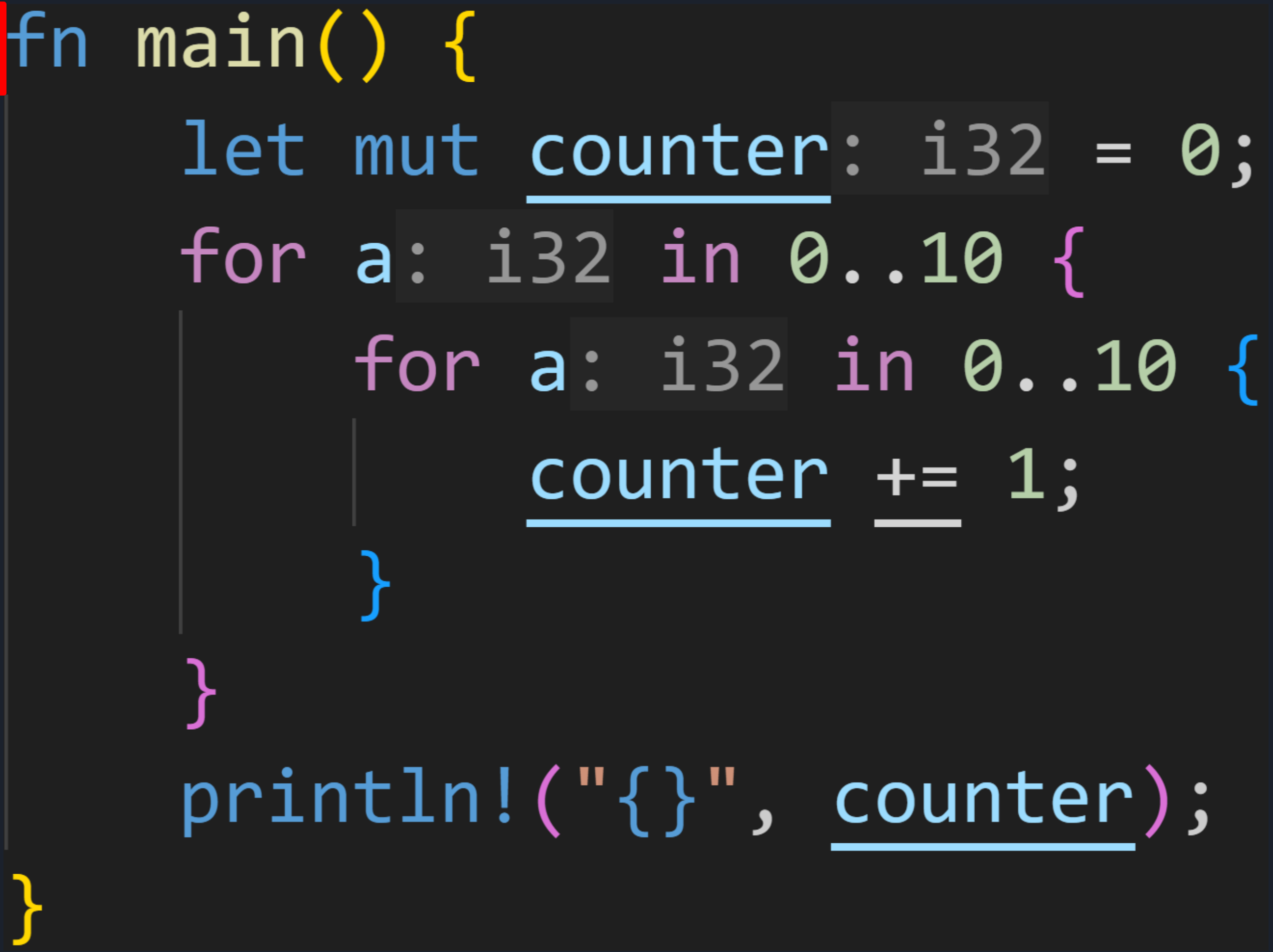
2/3



```
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

Intermission - Exercises

2/3

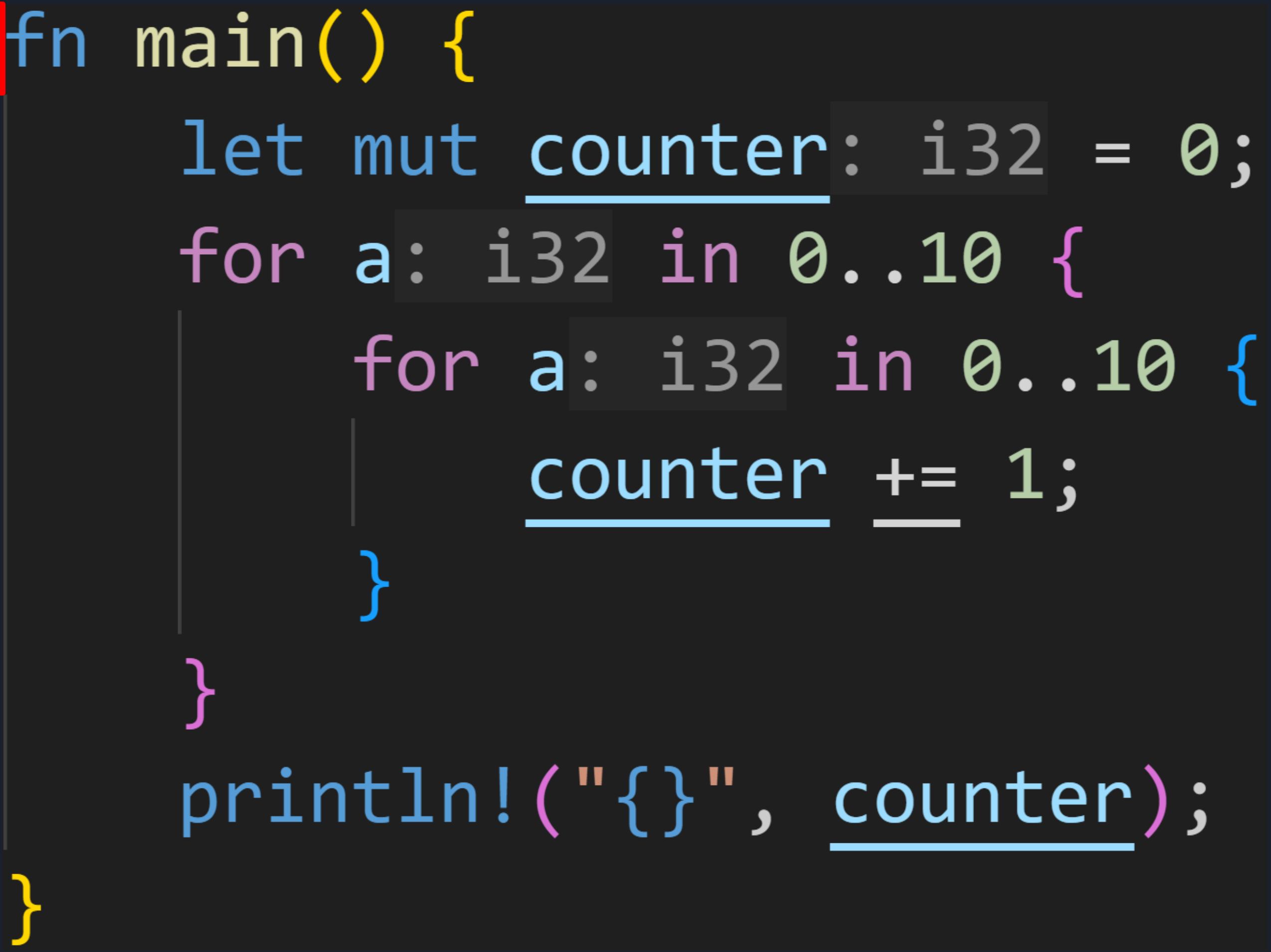


```
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

Does this code compile?
If yes, what does it print?

Intermission - Exercises

2/3



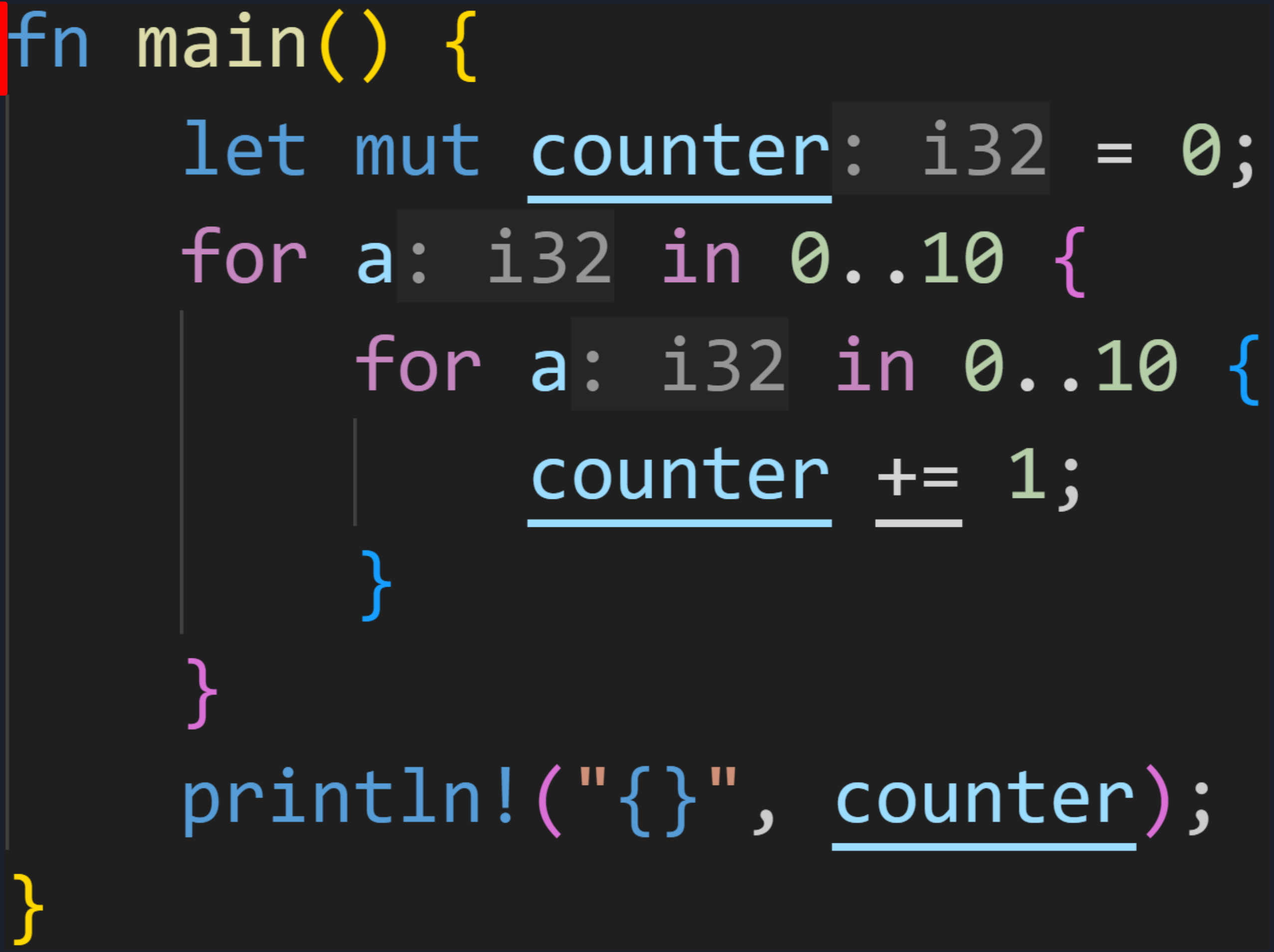
```rust
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

Does this code compile?
If yes, what does it print?

It does compile!

Intermission - Exercises

2/3



```rust
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

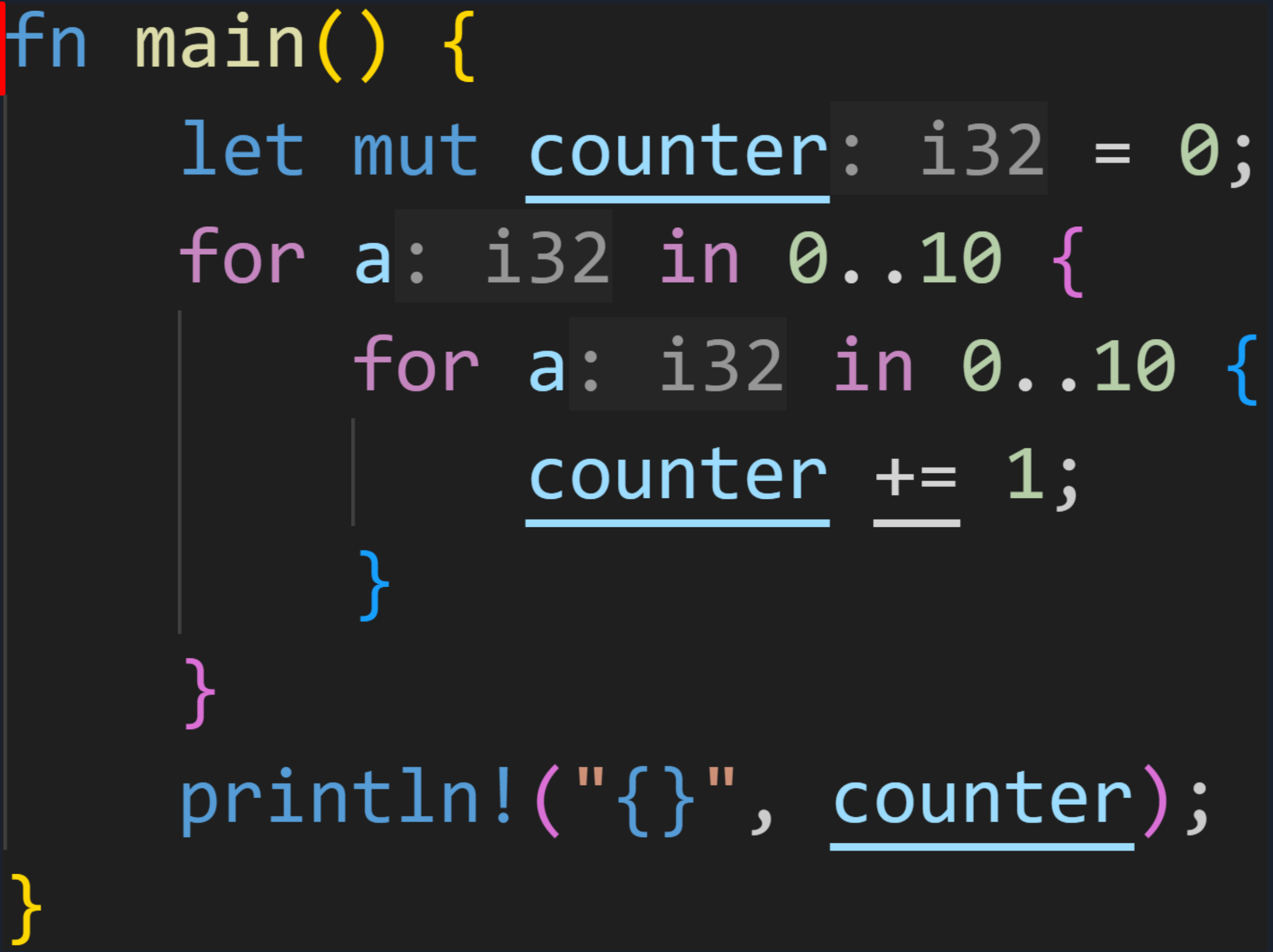
Does this code compile?
If yes, what does it print?

It does compile!

The answer is 100.

Intermission - Exercises

2/3



```
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

Does this code compile?
If yes, what does it print?

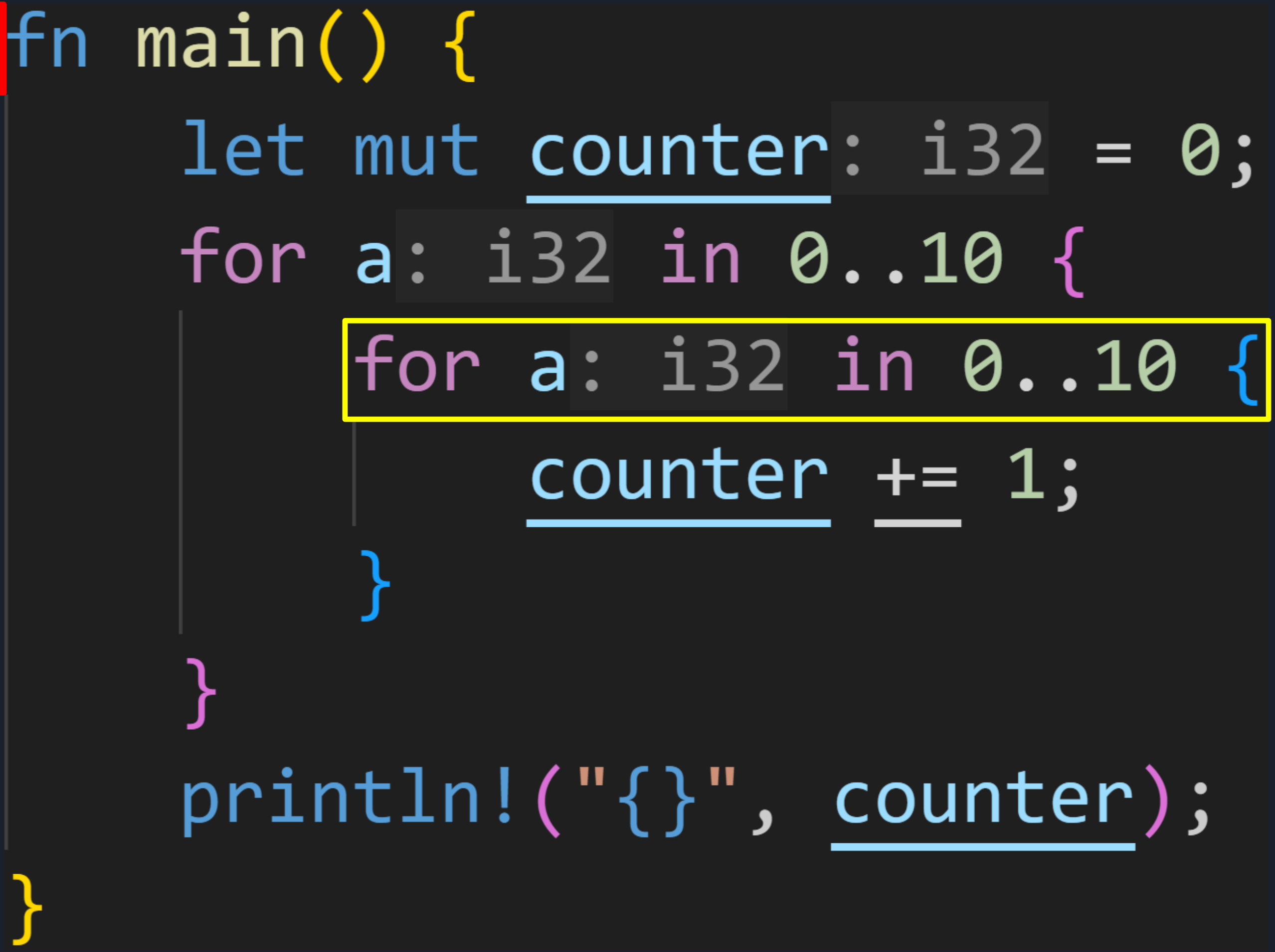
It does compile!

The answer is 100.

When nesting loops, runtime easily blows up

Intermission - Exercises

2/3



```
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

Does this code compile?
If yes, what does it print?

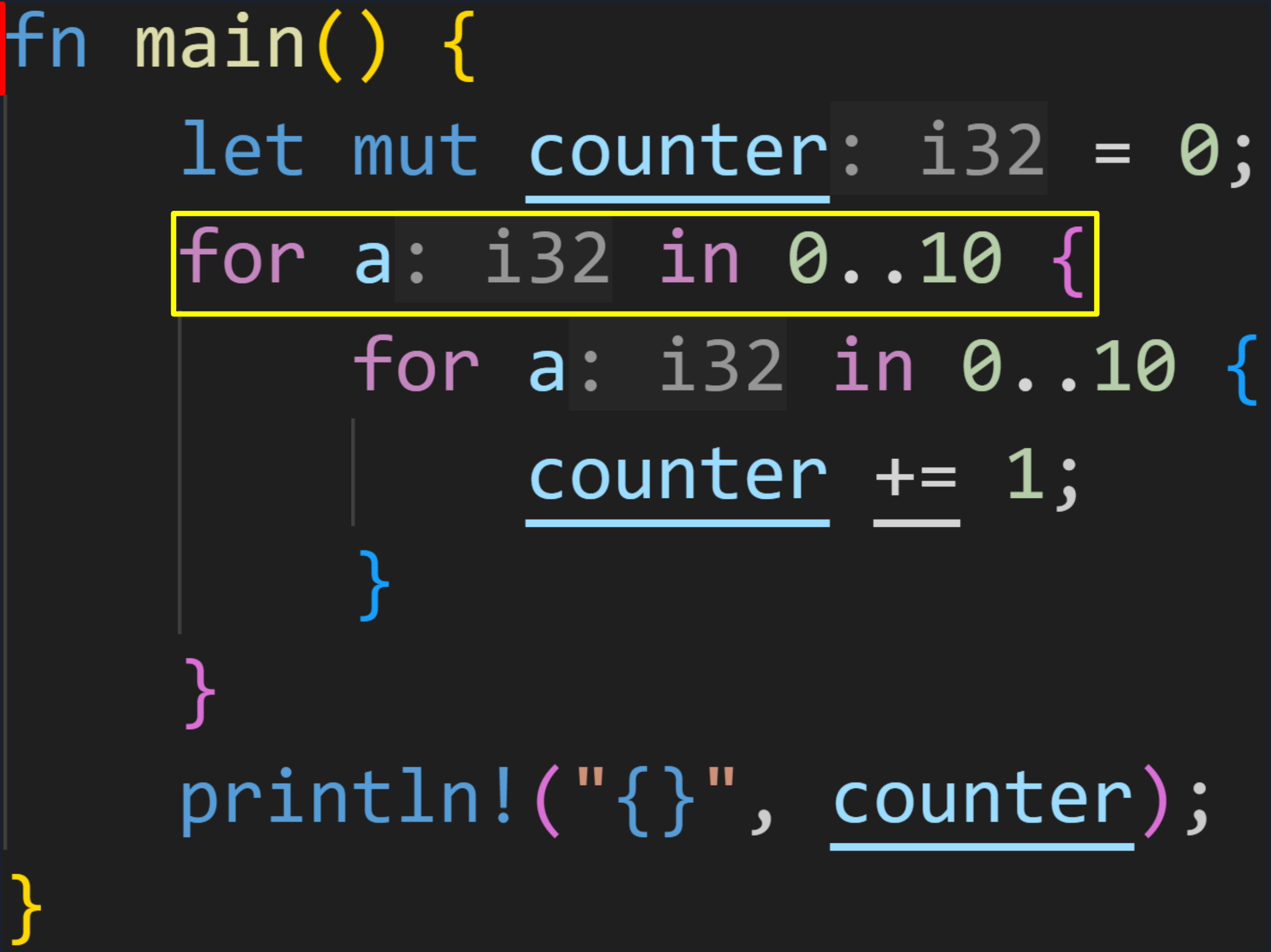
It does compile!

The answer is 100.

When nesting loops, runtime easily blows up
→ For every outer loop you need to run the inner loop 10 times

Intermission - Exercises

2/3



```
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

Does this code compile?
If yes, what does it print?

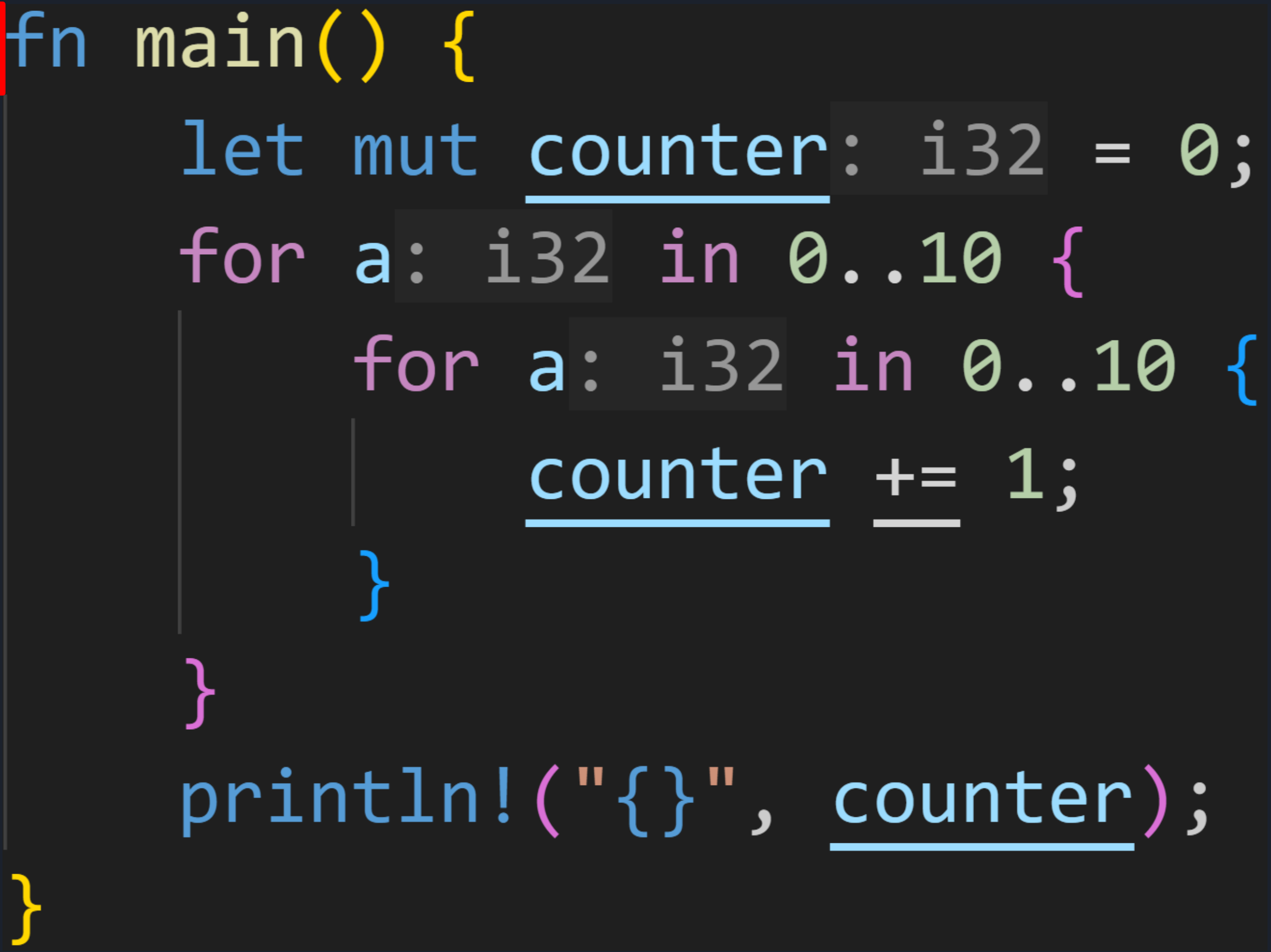
It does compile!

The answer is 100.

When nesting loops, runtime easily blows up
→ For every outer loop you need to run the inner loop 10 times
→ We run the outer loop 10 times

Intermission - Exercises

2/3



```
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

Does this code compile?
If yes, what does it print?

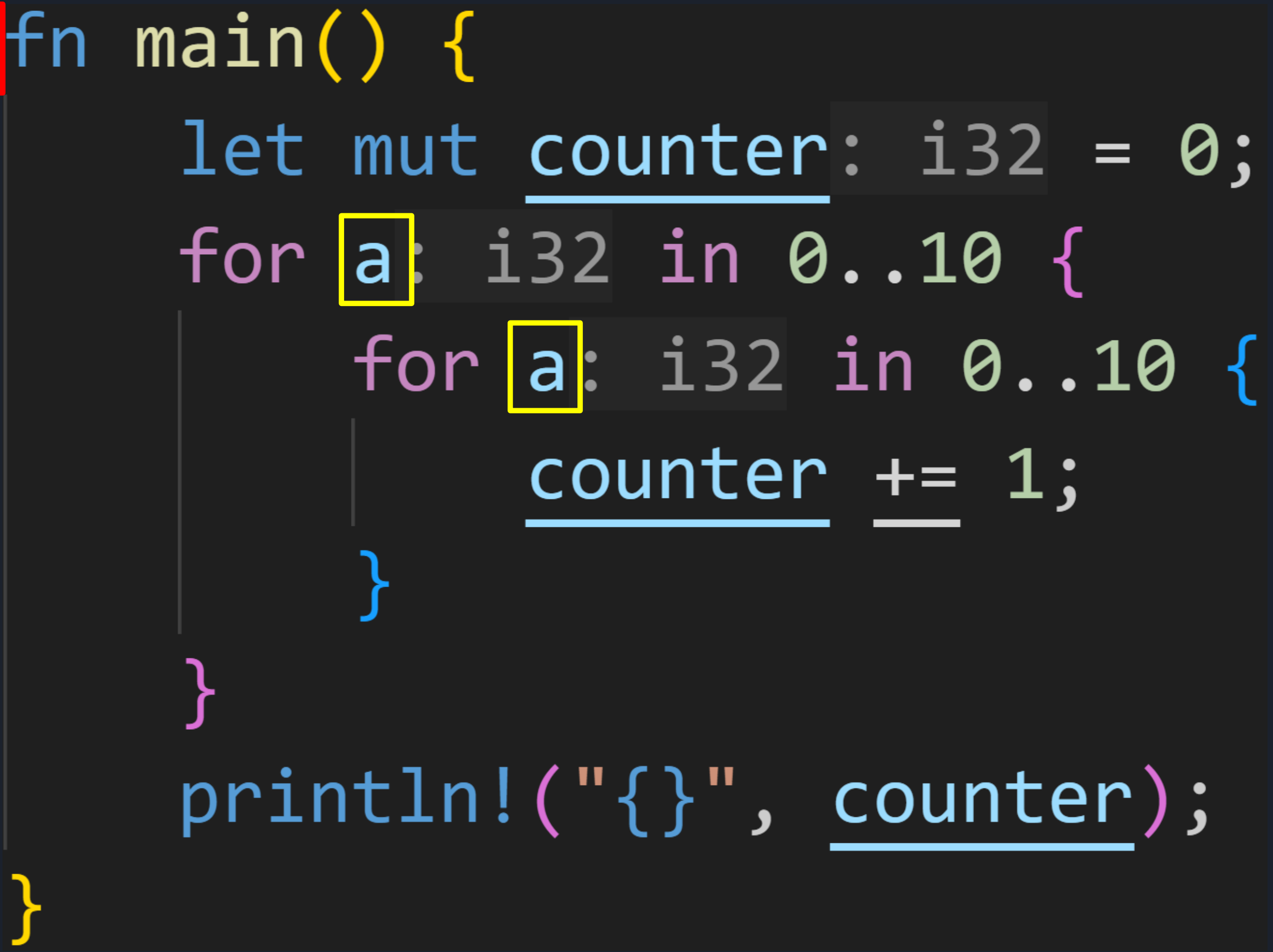
It does compile!

The answer is 100.

When nesting loops, runtime easily blows up
→ For every outer loop you need to run the inner loop 10 times
→ We run the outer loop 10 times
→ 10x10 == 100 iterations

Intermission - Exercises

2/3



```
fn main() {
    let mut counter: i32 = 0;
    for a: i32 in 0..10 {
        for a: i32 in 0..10 {
            counter += 1;
        }
    }
    println!("{}", counter);
}
```

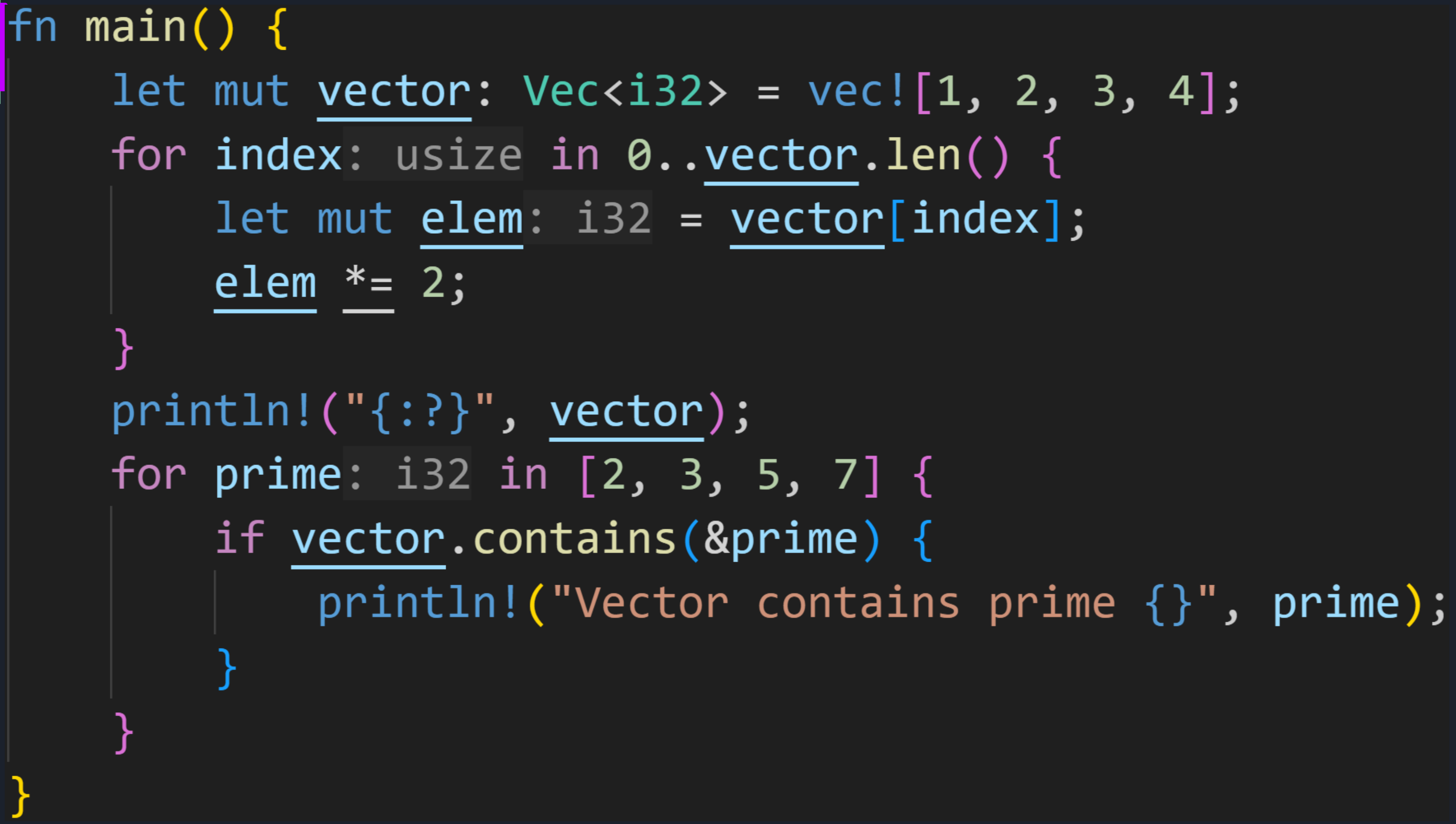
Does this code compile?
If yes, what does it print?

How often we run through the loop is not linked to the variable name

Variable shadowing did not matter

Intermission - Exercises

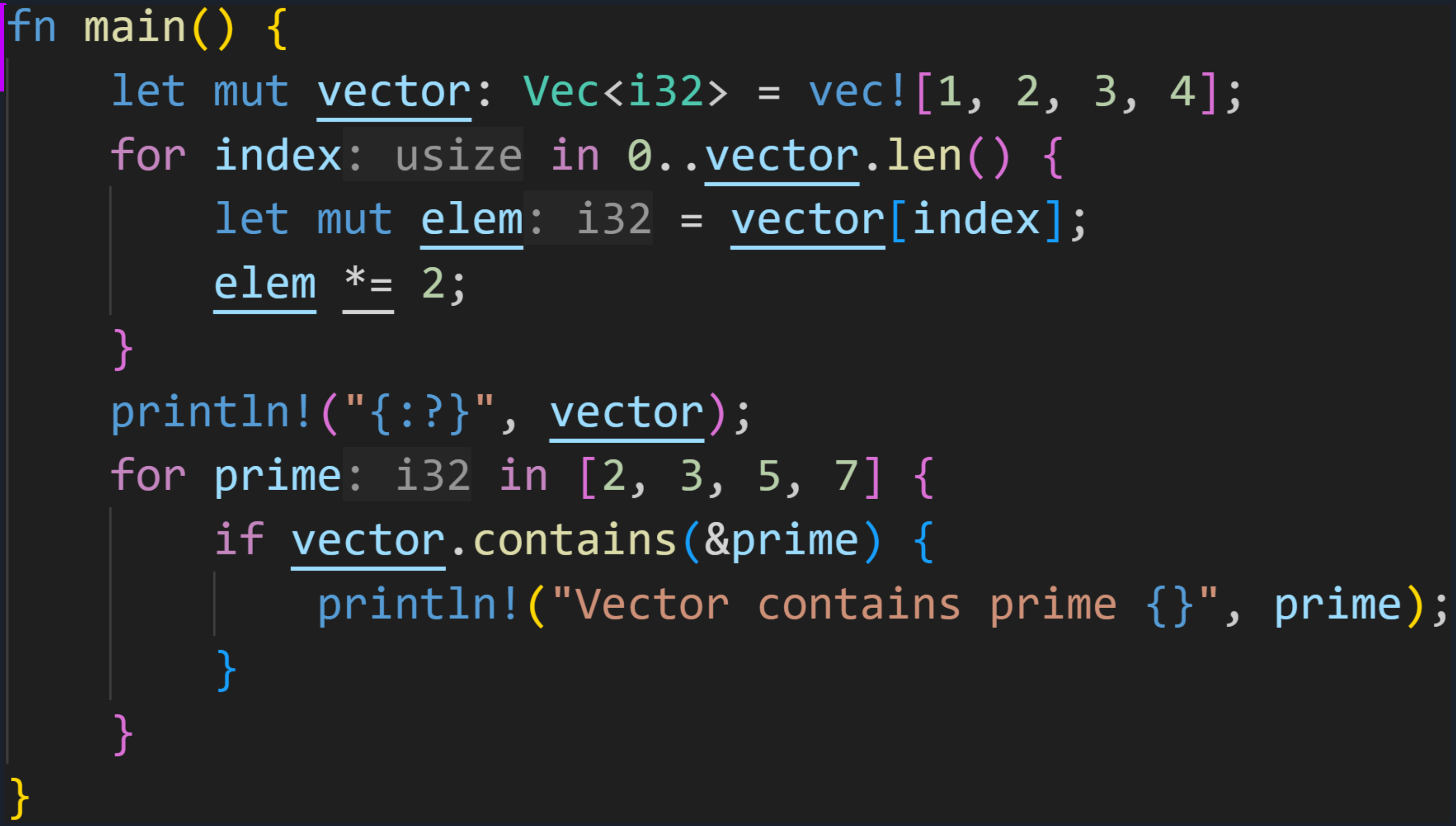
3/3



```
fn main() {
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    for index: usize in 0..vector.len() {
        let mut elem: i32 = vector[index];
        elem *= 2;
    }
    println!("{:?}", vector);
    for prime: i32 in [2, 3, 5, 7] {
        if vector.contains(&prime) {
            println!("Vector contains prime {}", prime);
        }
    }
}
```

Intermission - Exercises

3/3

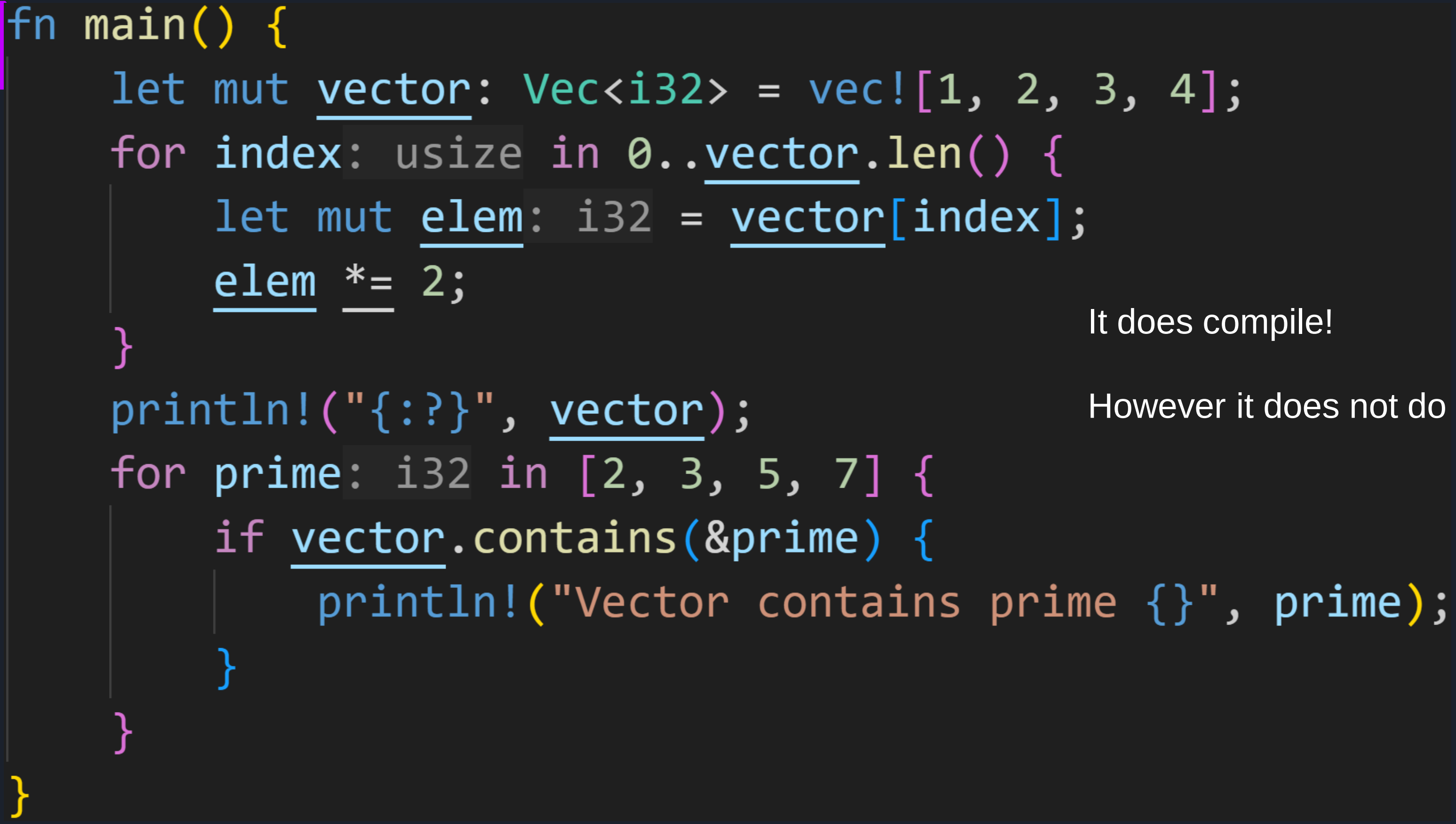


```
fn main() {
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    for index: usize in 0..vector.len() {
        let mut elem: i32 = vector[index];
        elem *= 2;
    }
    println!("{:?}", vector);
    for prime: i32 in [2, 3, 5, 7] {
        if vector.contains(&prime) {
            println!("Vector contains prime {}", prime);
        }
    }
}
```

Does this code compile?
If yes, what does it print?

Intermission - Exercises

3/3



```
fn main() {
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    for index: usize in 0..vector.len() {
        let mut elem: i32 = vector[index];
        elem *= 2;
    }
    println!("{:?}", vector);
    for prime: i32 in [2, 3, 5, 7] {
        if vector.contains(&prime) {
            println!("Vector contains prime {}", prime);
        }
    }
}
```

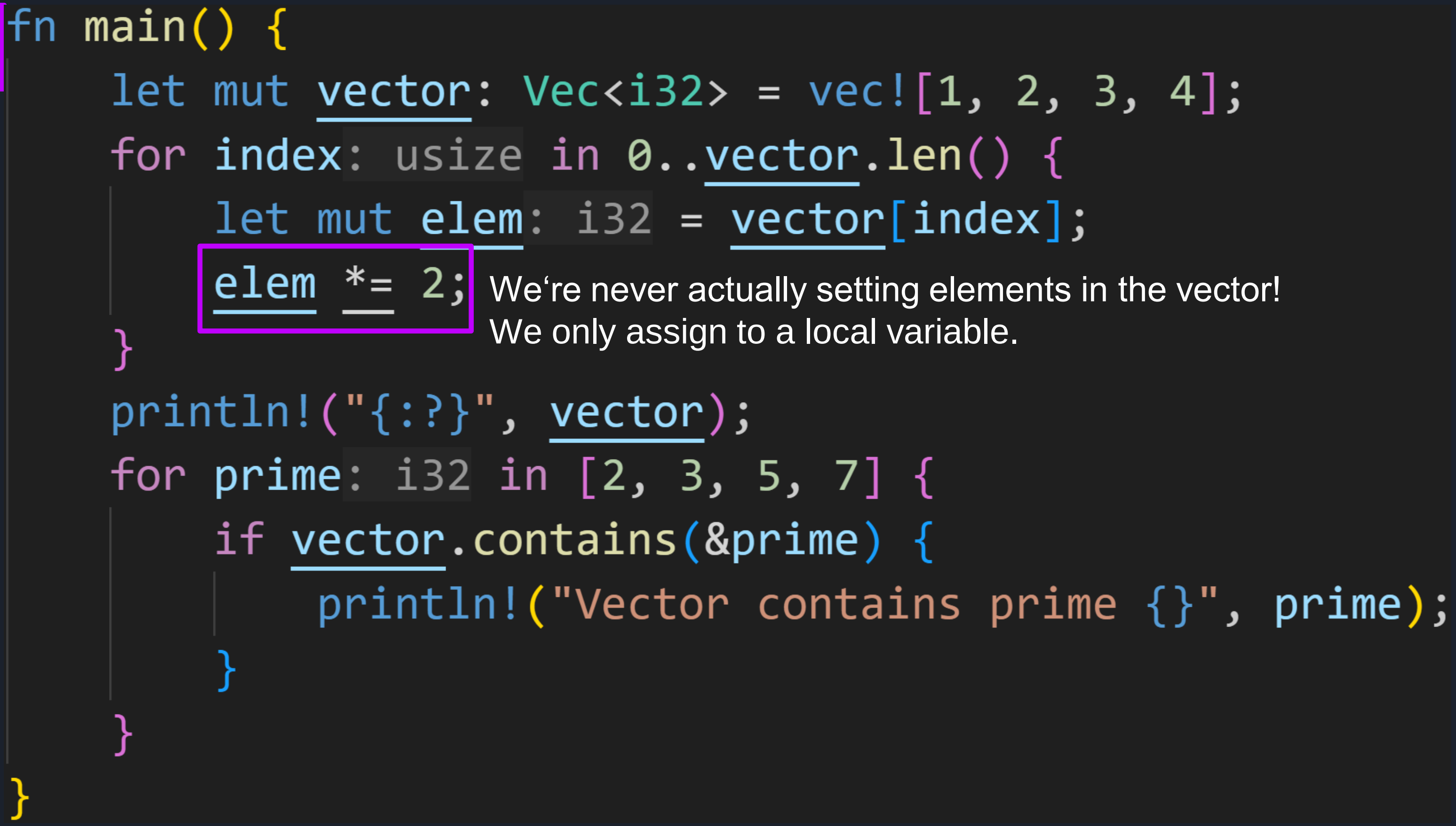
Does this code compile?
If yes, what does it print?

It does compile!

However it does not do what you might think it does...

Intermission - Exercises

3/3



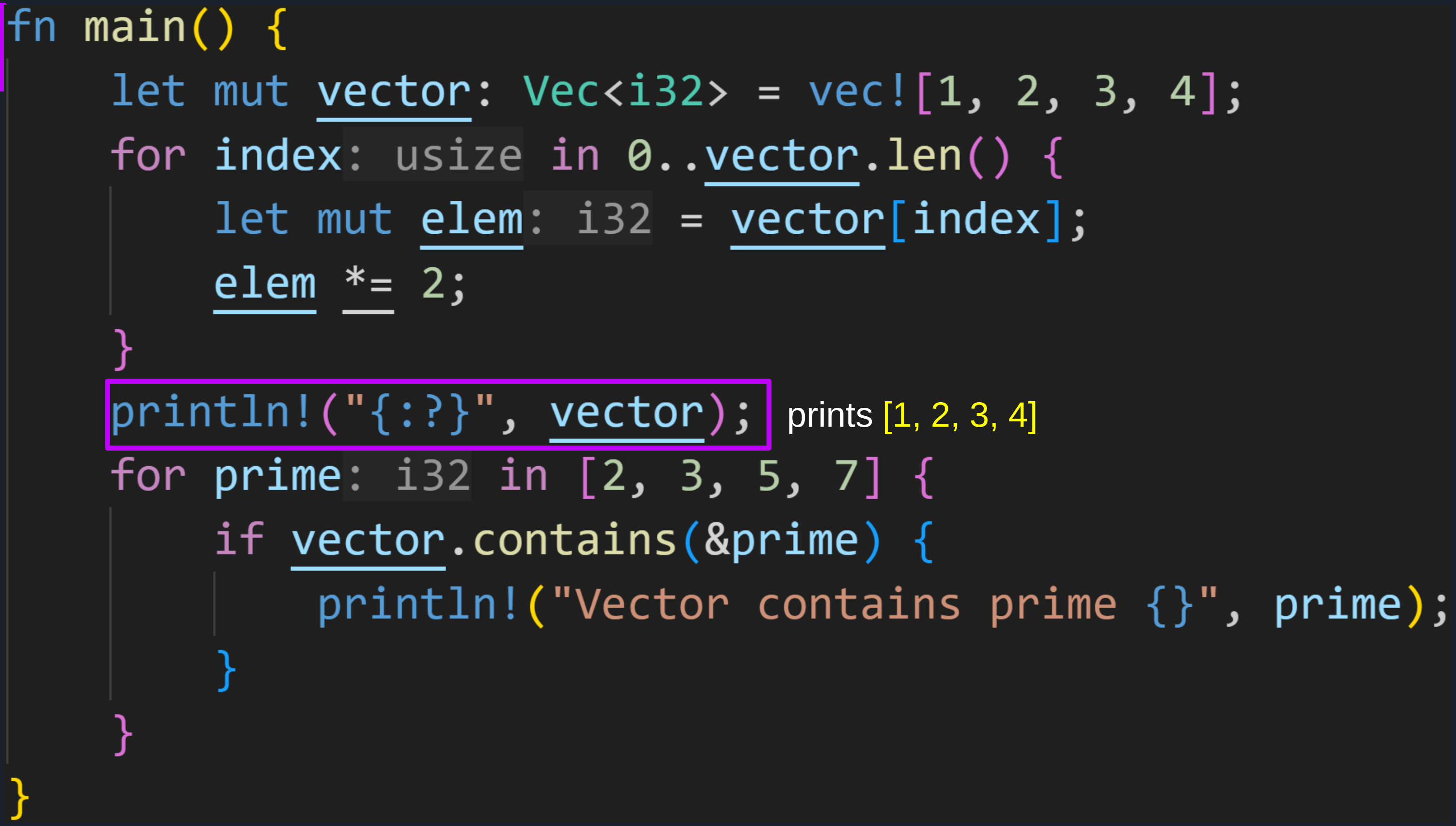
```rust
fn main() {
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    for index: usize in 0..vector.len() {
        let mut elem: i32 = vector[index];
        elem *= 2;
    }
    println!("{:?}", vector);
    for prime: i32 in [2, 3, 5, 7] {
        if vector.contains(&prime) {
            println!("Vector contains prime {}", prime);
        }
    }
}
```

We‘re never actually setting elements in the vector!
We only assign to a local variable.

Does this code compile?
If yes, what does it print?

Intermission - Exercises

3/3



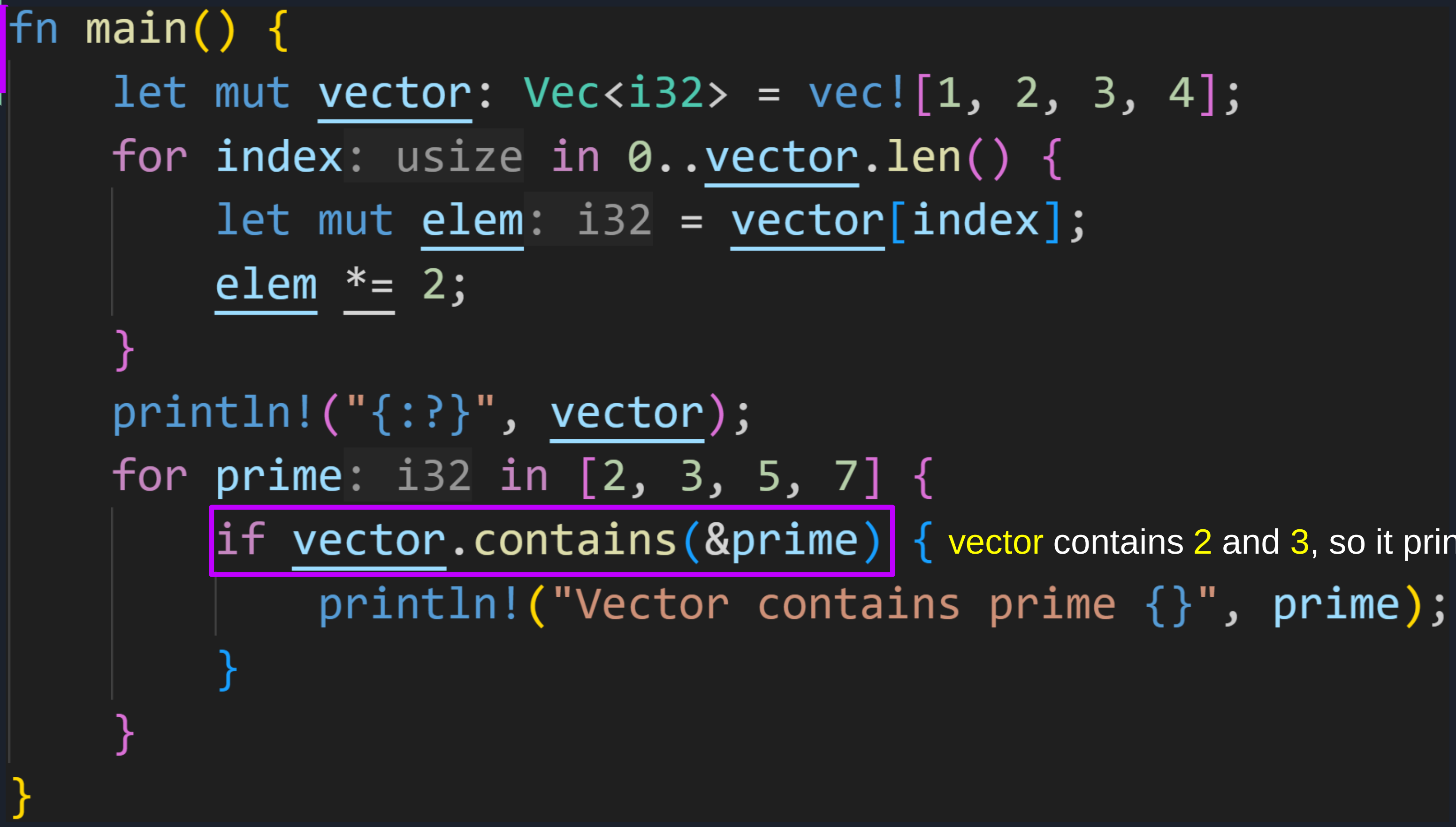
```
fn main() {
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    for index: usize in 0..vector.len() {
        let mut elem: i32 = vector[index];
        elem *= 2;
    }
    println!("{:?}", vector);
    for prime: i32 in [2, 3, 5, 7] {
        if vector.contains(&prime) {
            println!("Vector contains prime {}", prime);
        }
    }
}
```

prints [1, 2, 3, 4]

Does this code compile?
If yes, what does it print?

Intermission - Exercises

3/3



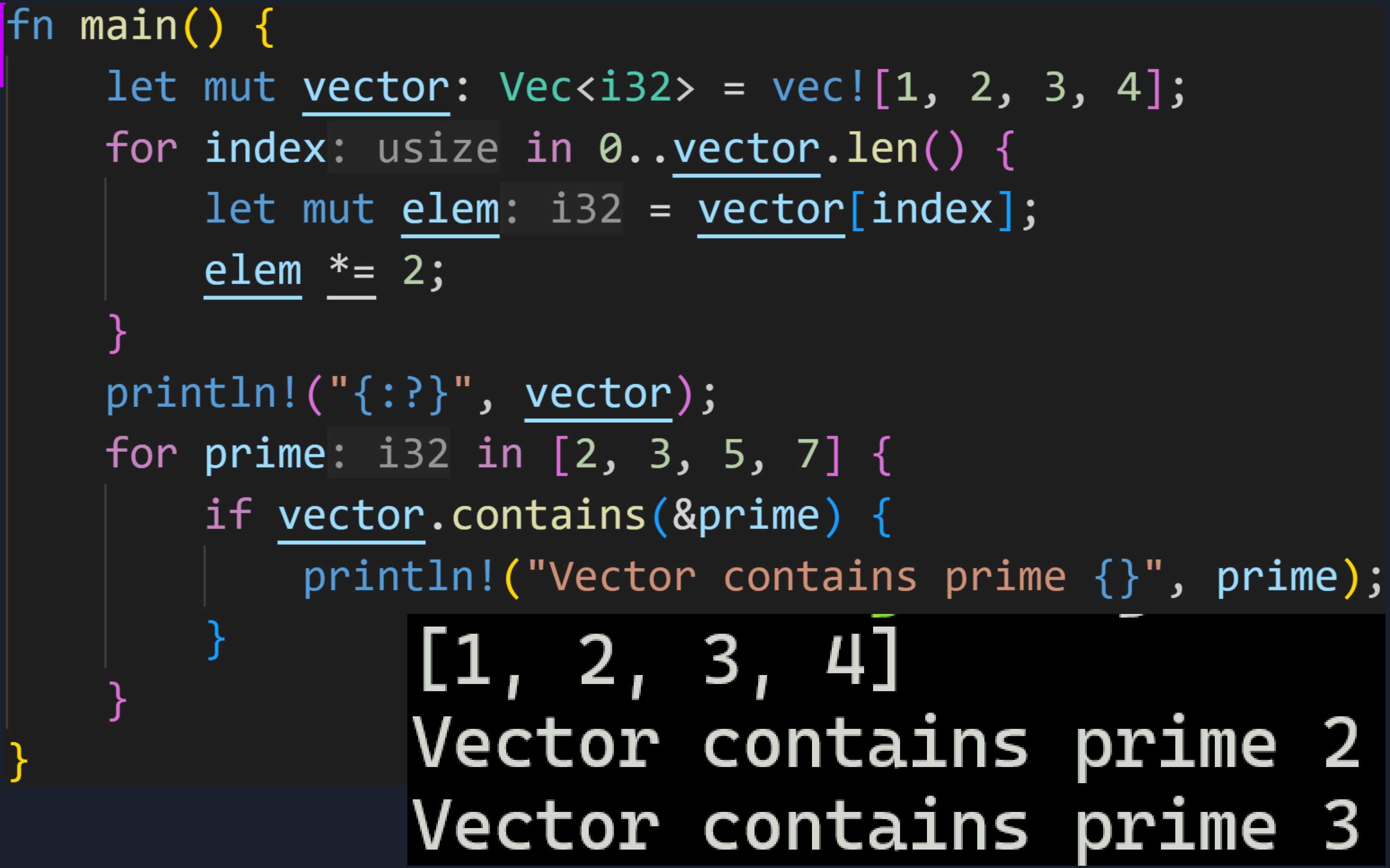
```
fn main() {
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    for index: usize in 0..vector.len() {
        let mut elem: i32 = vector[index];
        elem *= 2;
    }
    println!("{:?}", vector);
    for prime: i32 in [2, 3, 5, 7] {
        if vector.contains(&prime) {
            println!("Vector contains prime {}", prime);
        }
    }
}
```

vector contains 2 and 3, so it prints those two

Does this code compile?
If yes, what does it print?

Intermission - Exercises

3/3



```rust
fn main() {
    let mut vector: Vec<i32> = vec![1, 2, 3, 4];
    for index: usize in 0..vector.len() {
        let mut elem: i32 = vector[index];
        elem *= 2;
    }
    println!("{:?}", vector);
    for prime: i32 in [2, 3, 5, 7] {
        if vector.contains(&prime) {
            println!("Vector contains prime {}", prime);
        }
    }
}
```

```
[1, 2, 3, 4]
Vector contains prime 2
Vector contains prime 3
```

Does this code compile?
If yes, what does it print?

3. Next time

- – Ownership
- – Borrow Checker
- – References

4. Loops - Extra

4. Loops - Extra

- You might think that there exists the „most powerful“ loop

4. Loops - Extra

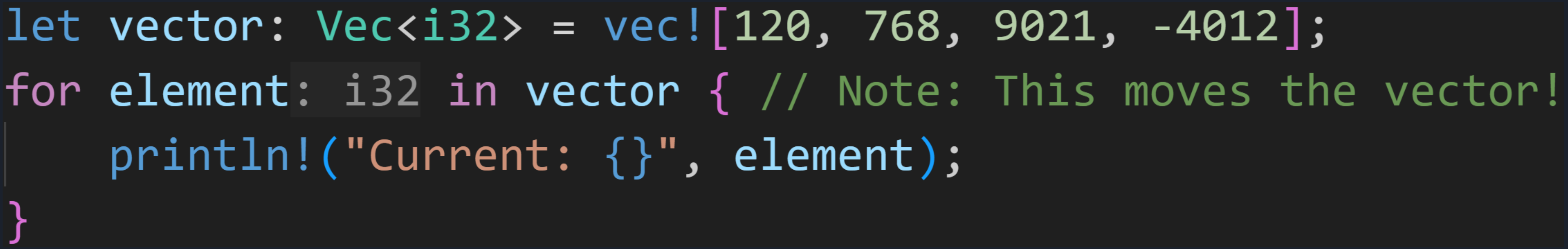
- You might think that there exists the „most powerful“ loop
- Indeed, some loops are more convenient than others, you would not use loop to iterate over a collection!

4. Loops - Extra

- You might think that there exists the „most powerful“ loop
- Indeed, some loops are more convenient than others, you would not use loop to iterate over a collection!
- However, all loops can be converted to each other → All loops are equally powerful!

4. Loops - Extra

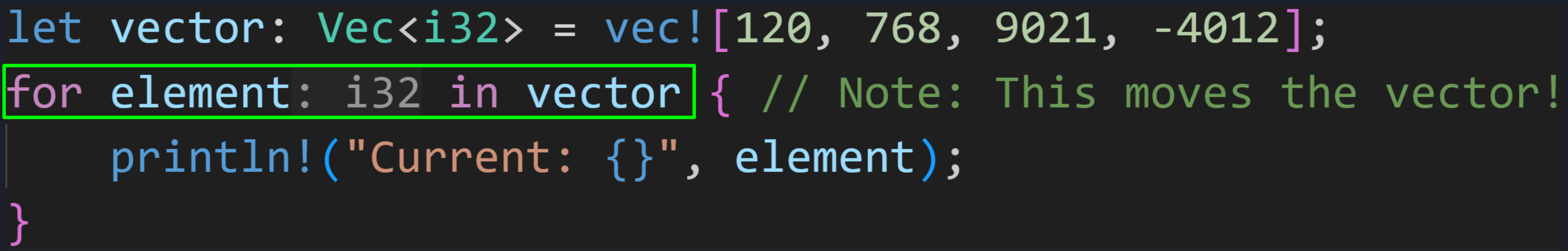
Goal: Convert this for loop into a normal loop loop



```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for element: i32 in vector { // Note: This moves the vector!
    println!("Current: {}", element);
}
```

4. Loops - Extra

Goal: Convert this for loop into a normal loop loop

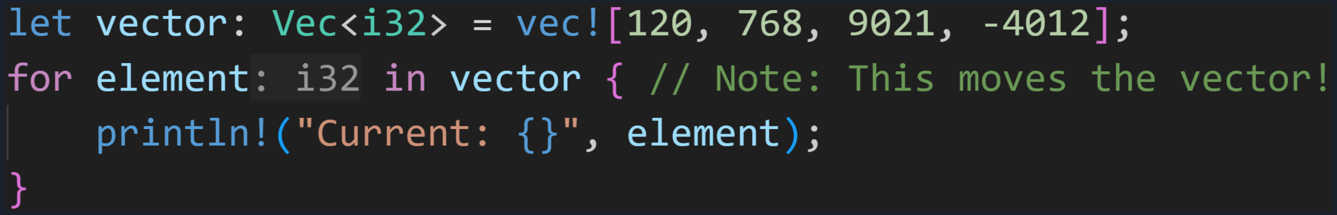


```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for element: i32 in vector { // Note: This moves the vector!
    println!("Current: {}", element);
}
```

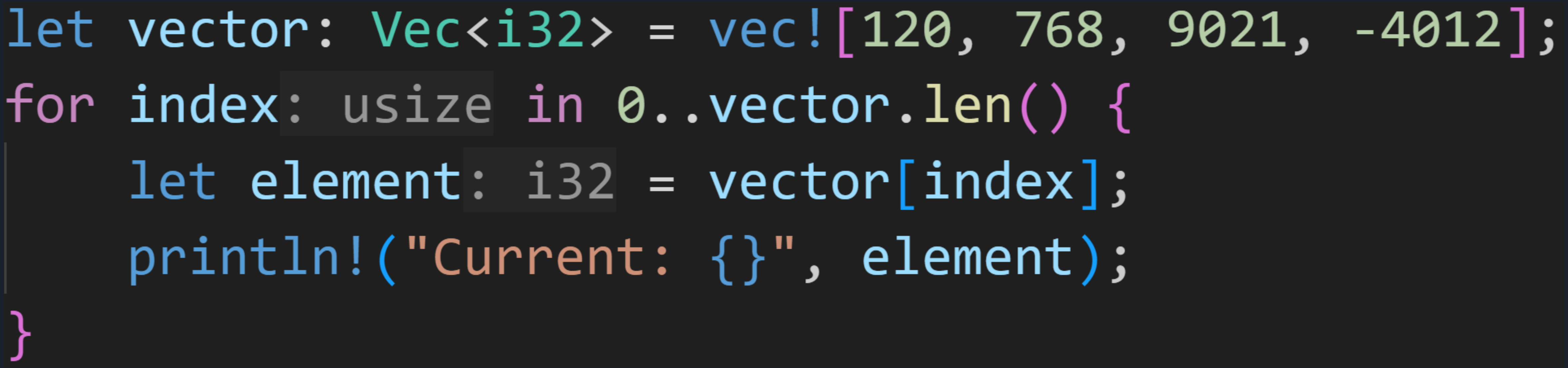
Normal loop can‘t easily do the assign thing, we need to first desugar this line

4. Loops - Extra

Goal: Convert this for loop into a normal loop loop



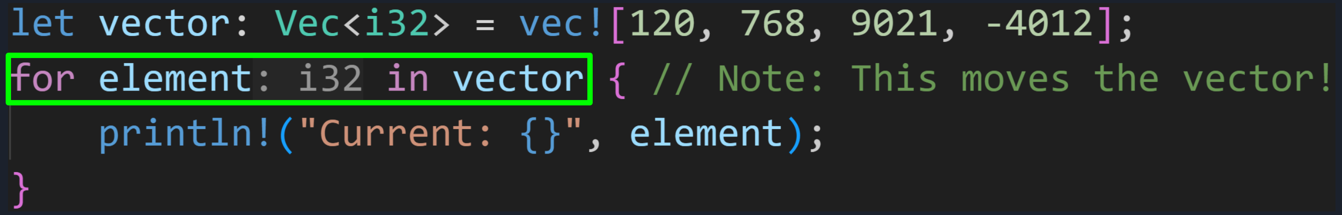
```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for element: i32 in vector { // Note: This moves the vector!
    println!("Current: {}", element);
}
```



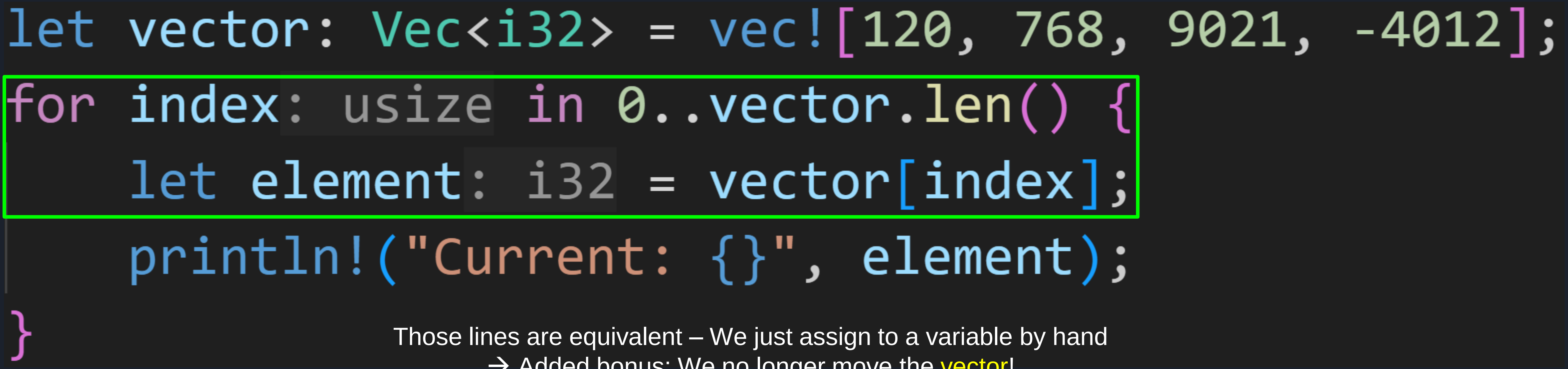
```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```

4. Loops - Extra

Goal: Convert this for loop into a normal loop loop



```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for element: i32 in vector { // Note: This moves the vector!
    println!("Current: {}", element);
}
```



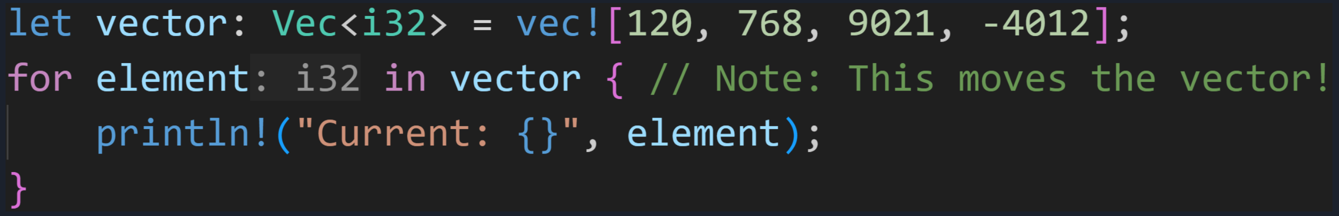
```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```

Those lines are equivalent – We just assign to a variable by hand

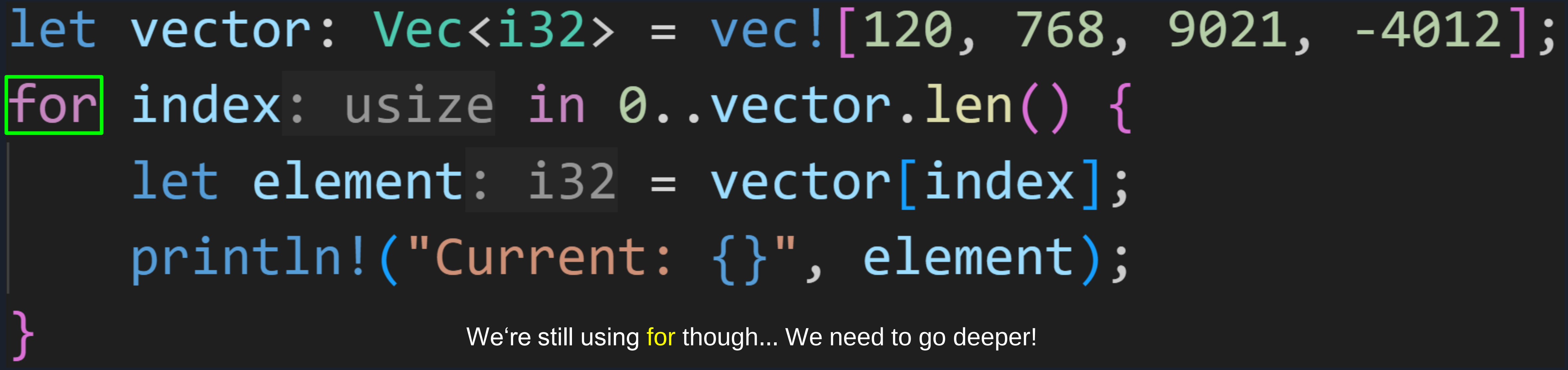
→ Added bonus: We no longer move the vector!

4. Loops - Extra

Goal: Convert this for loop into a normal loop loop



```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for element: i32 in vector { // Note: This moves the vector!
    println!("Current: {}", element);
}
```

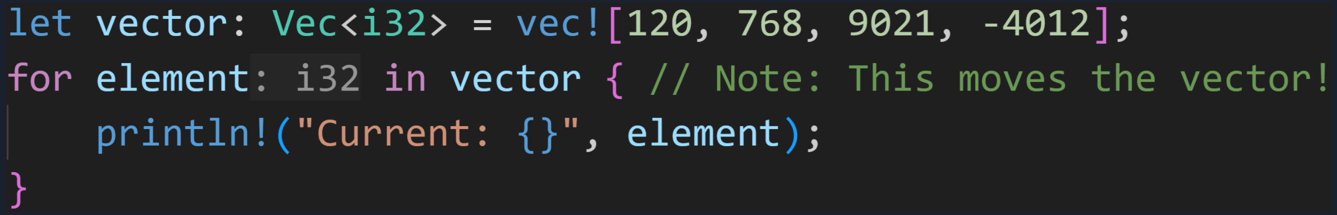


```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```

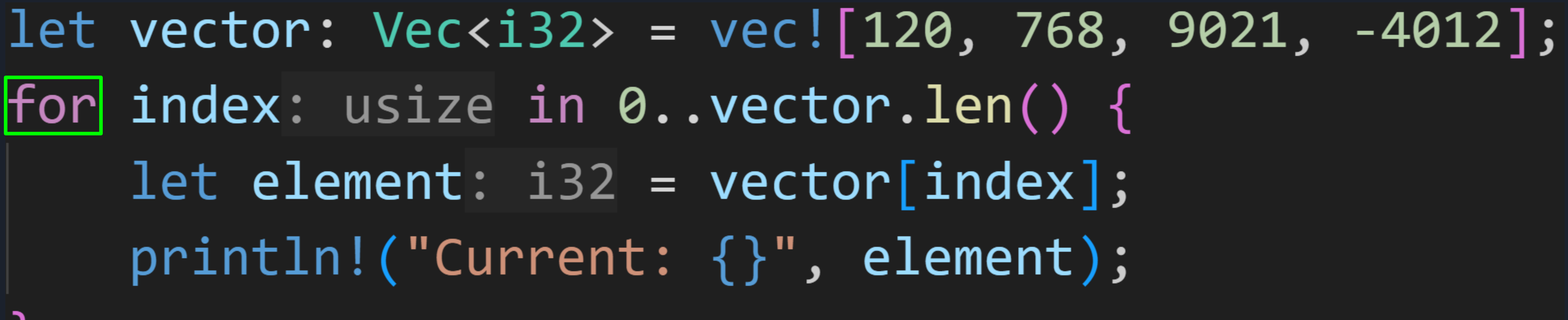
We‘re still using for though... We need to go deeper!

4. Loops - Extra

Goal: Convert this for loop into a normal loop loop



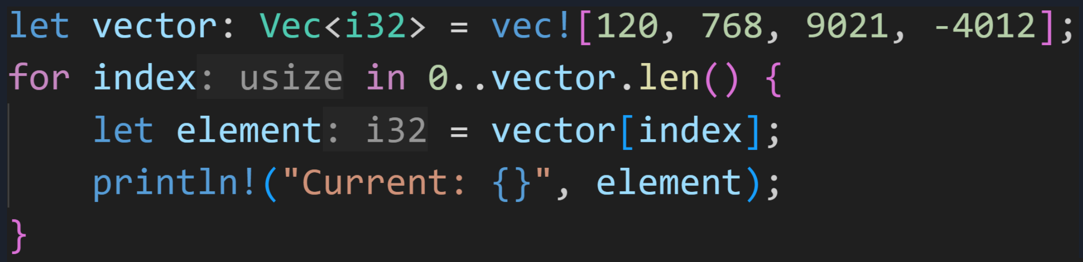
```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for element: i32 in vector { // Note: This moves the vector!
    println!("Current: {}", element);
}
```



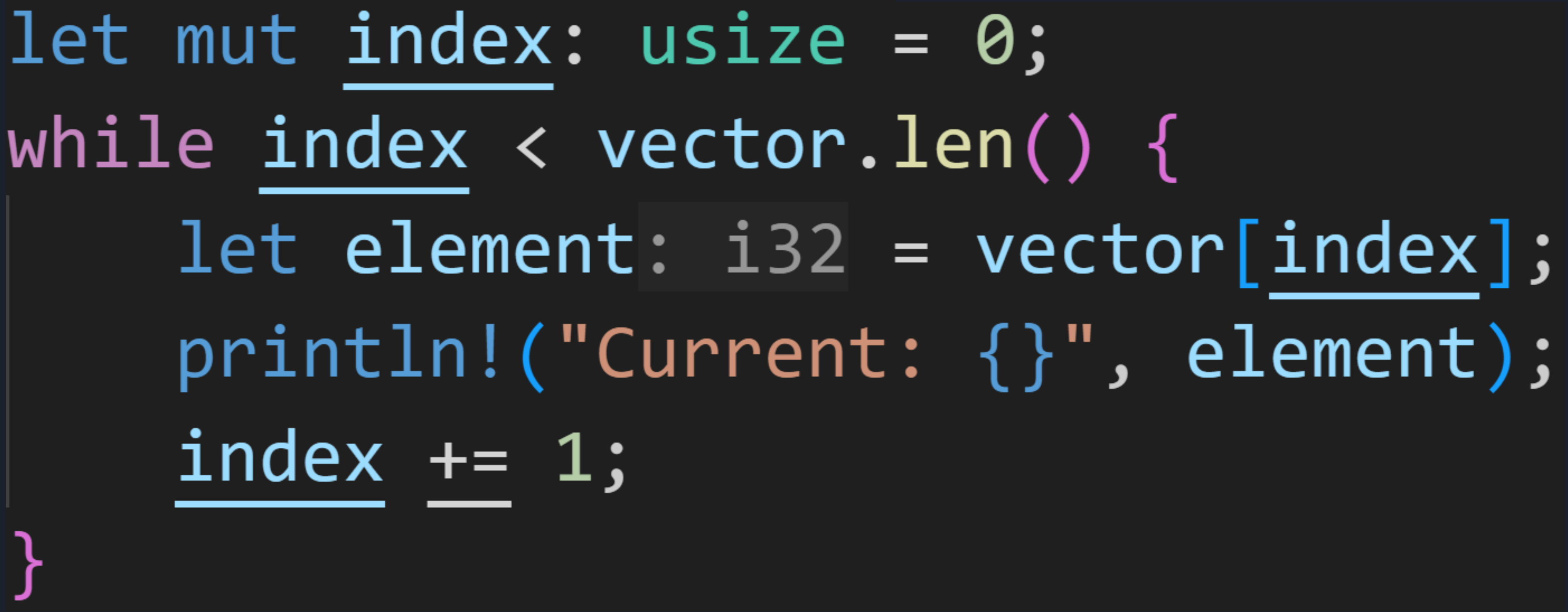
```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```

We‘re still using for though... We need to go deeper!
We need to understand what the Range does.

4. Loops - Extra

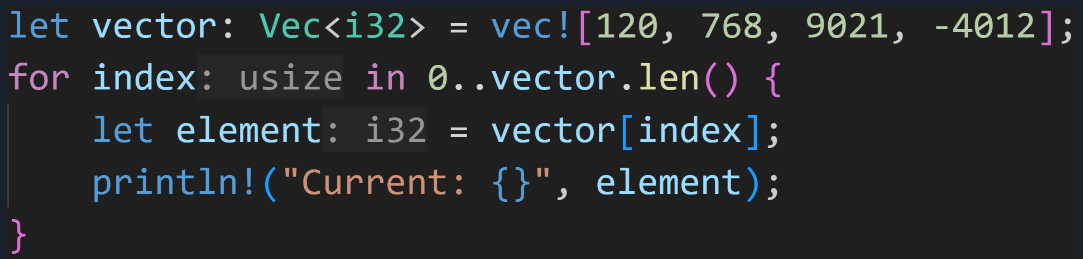


```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```



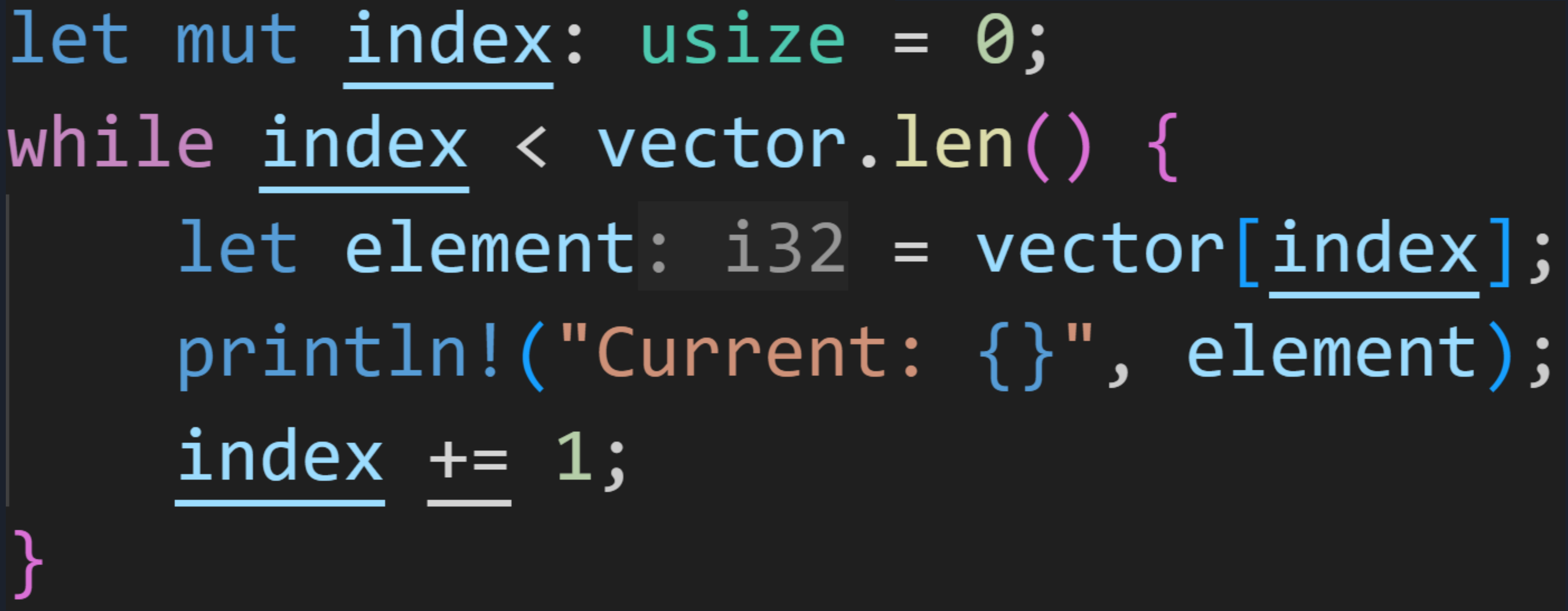
```
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

4. Loops - Extra



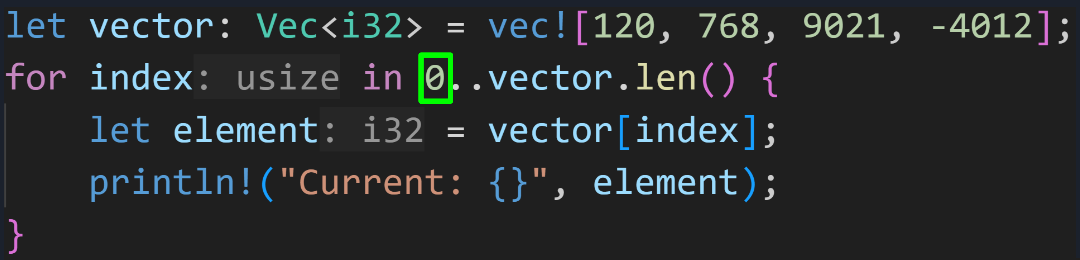
```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```

Where did our range go?

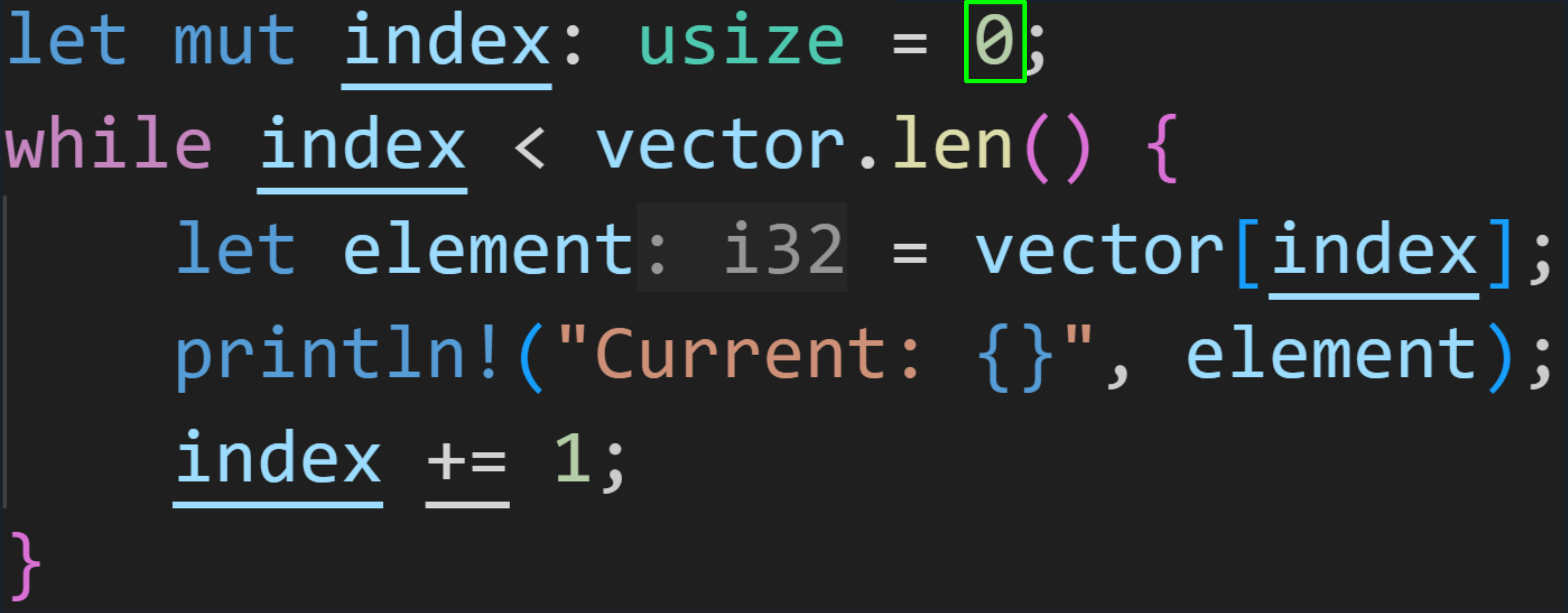


```
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

4. Loops - Extra



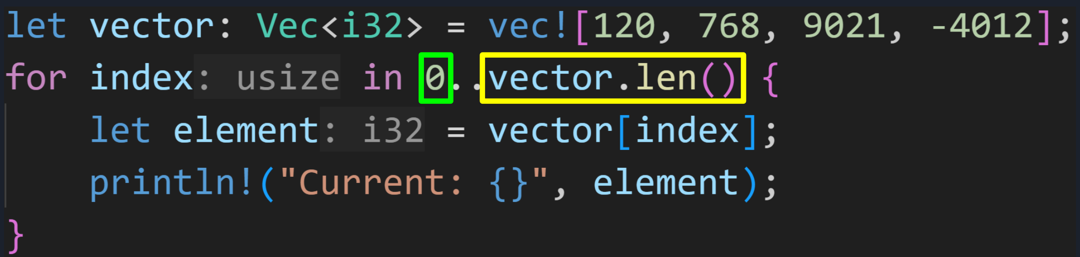
```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```



```rust
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

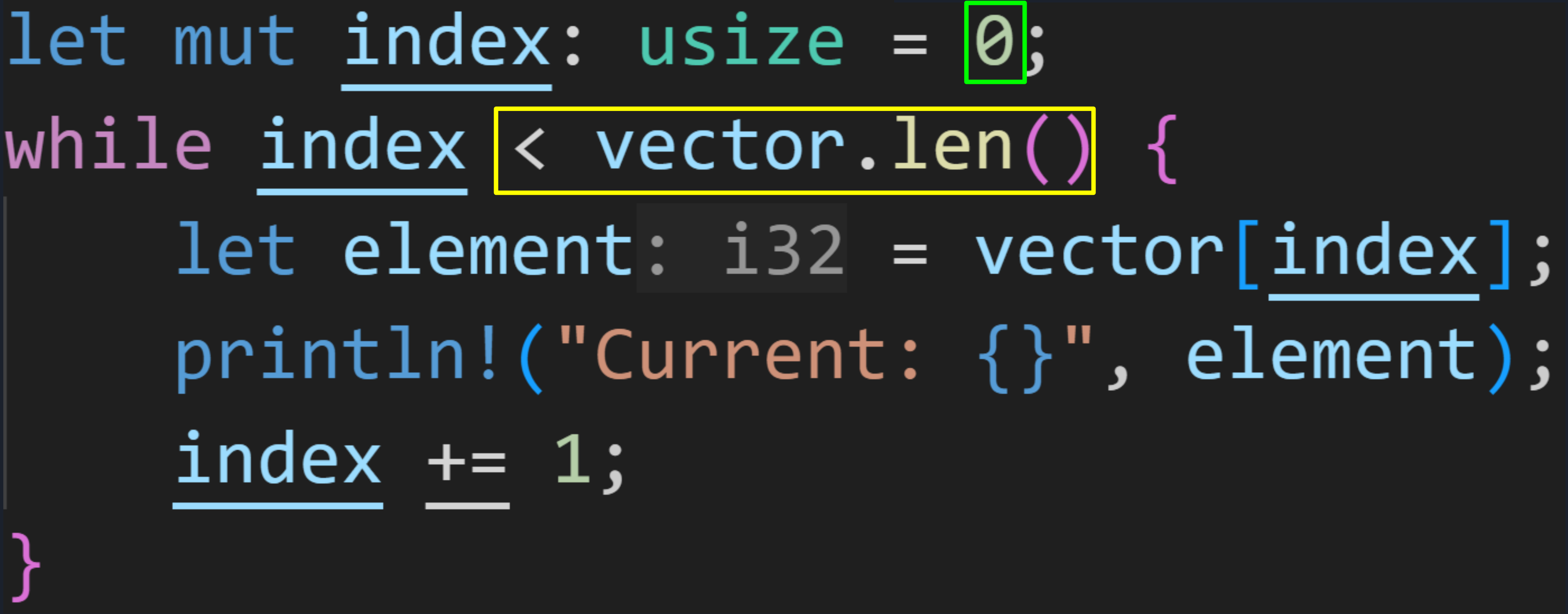
Where did our range go?

4. Loops - Extra



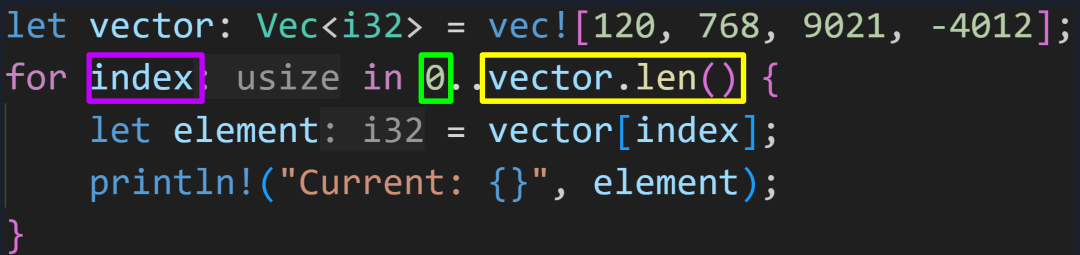
```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```

Where did our range go?



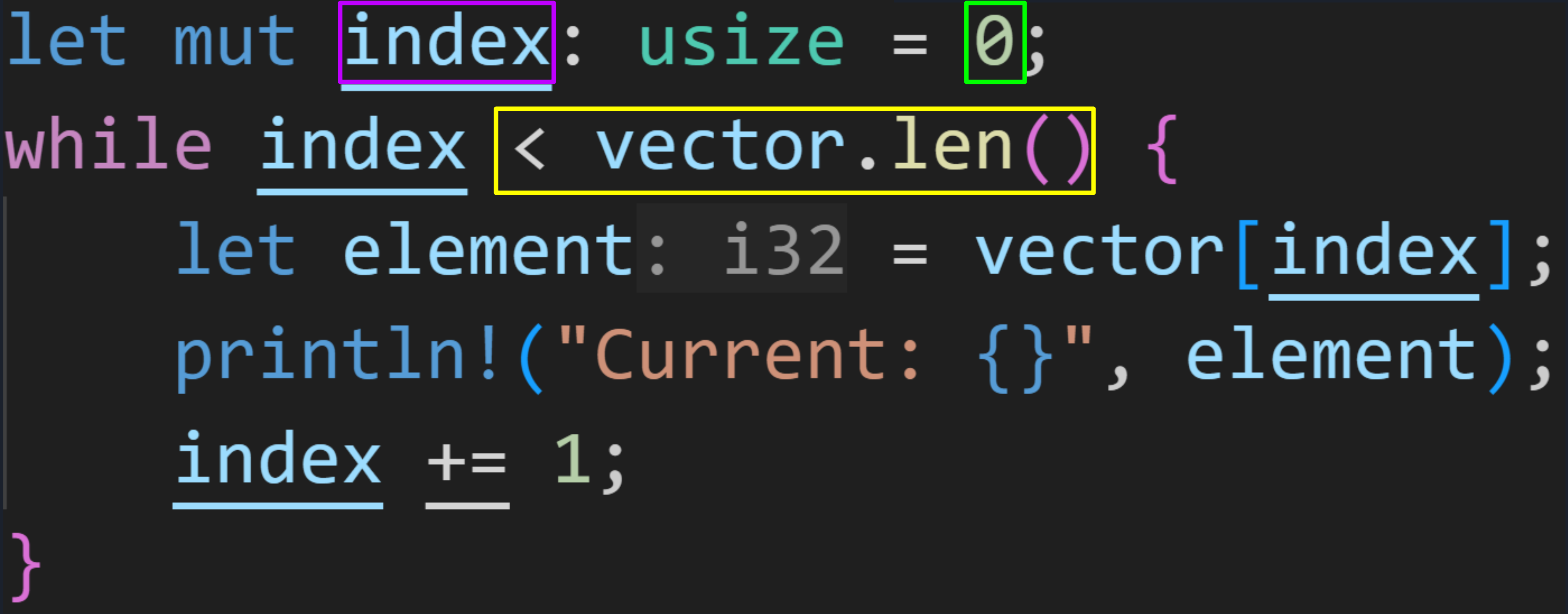
```rust
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

4. Loops - Extra



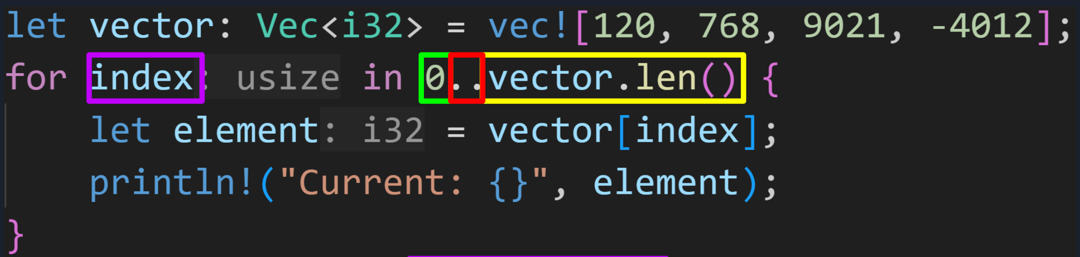
```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```

Where did our range go?

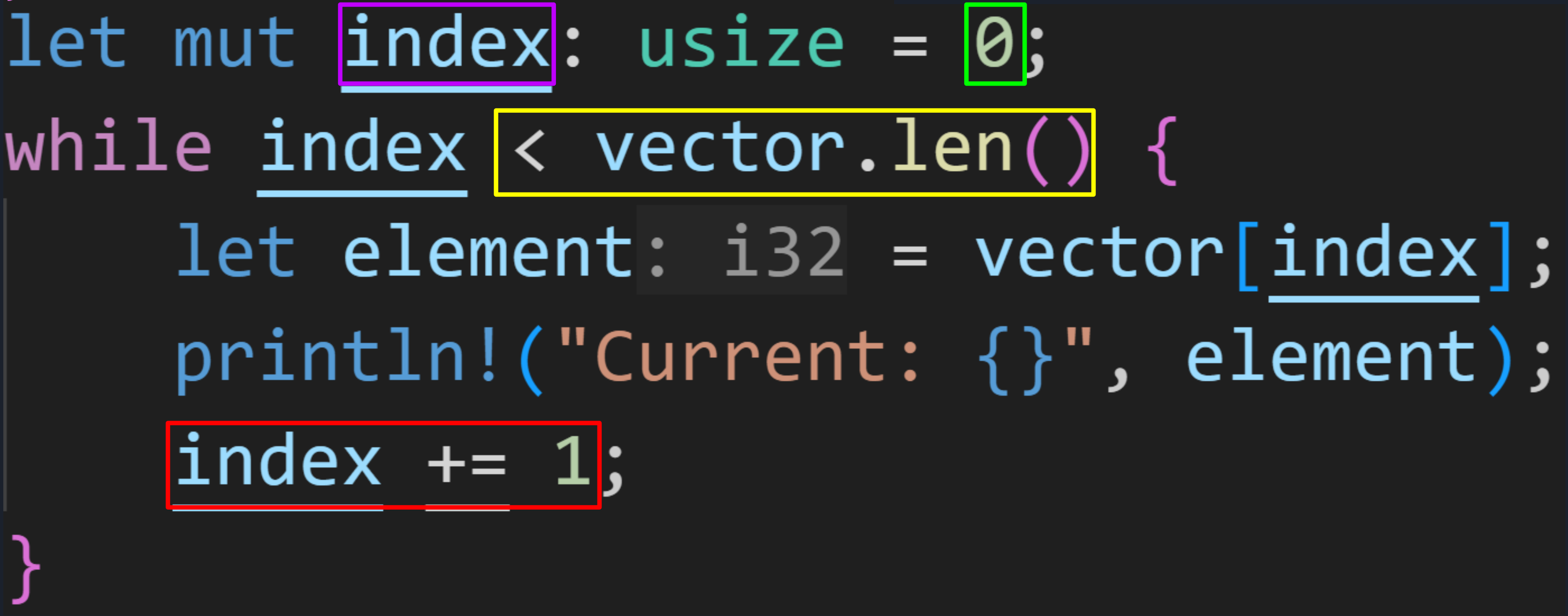


```
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

4. Loops - Extra



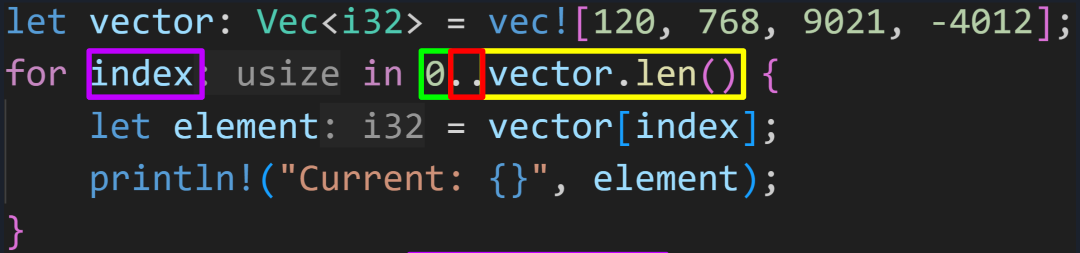
```rust
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```



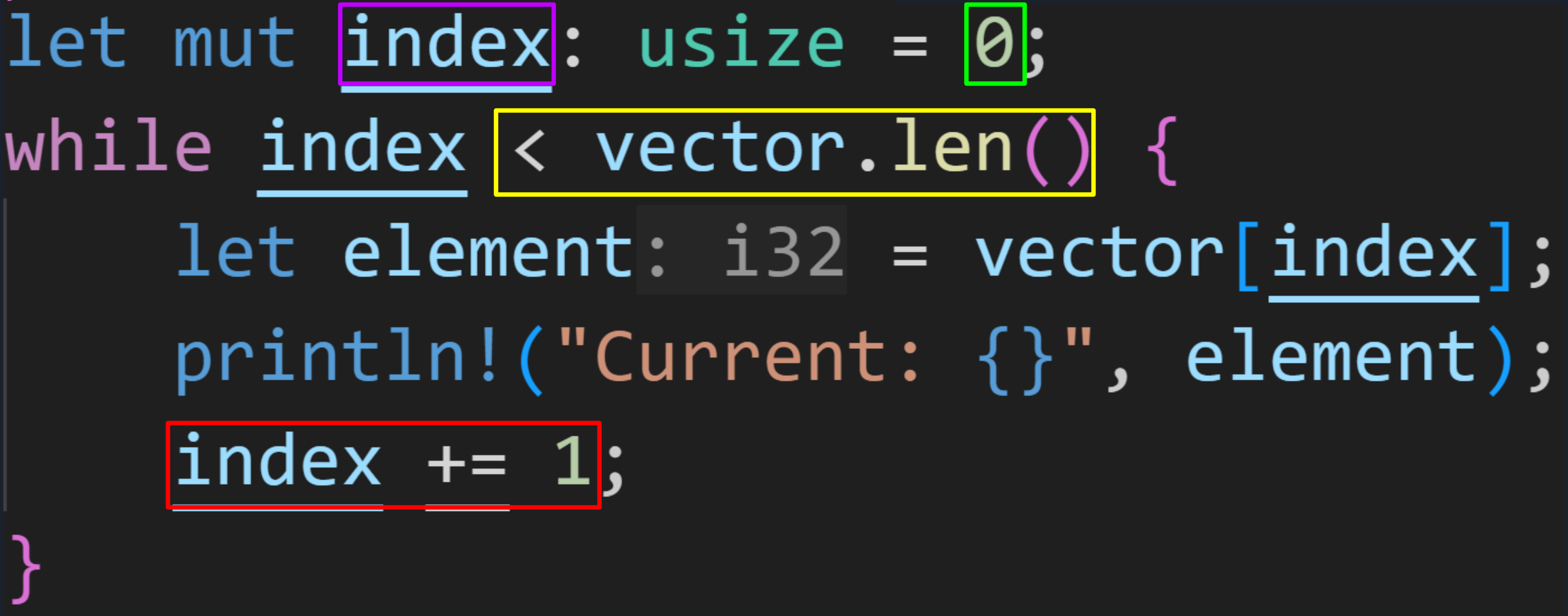
```rust
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

Where did our range go?

4. Loops - Extra



```
let vector: Vec<i32> = vec![120, 768, 9021, -4012];
for index: usize in 0..vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
}
```

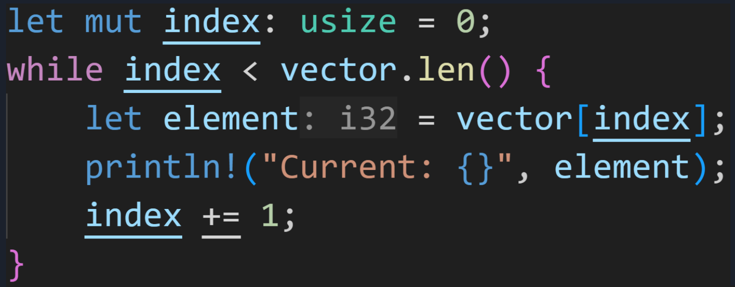


```
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

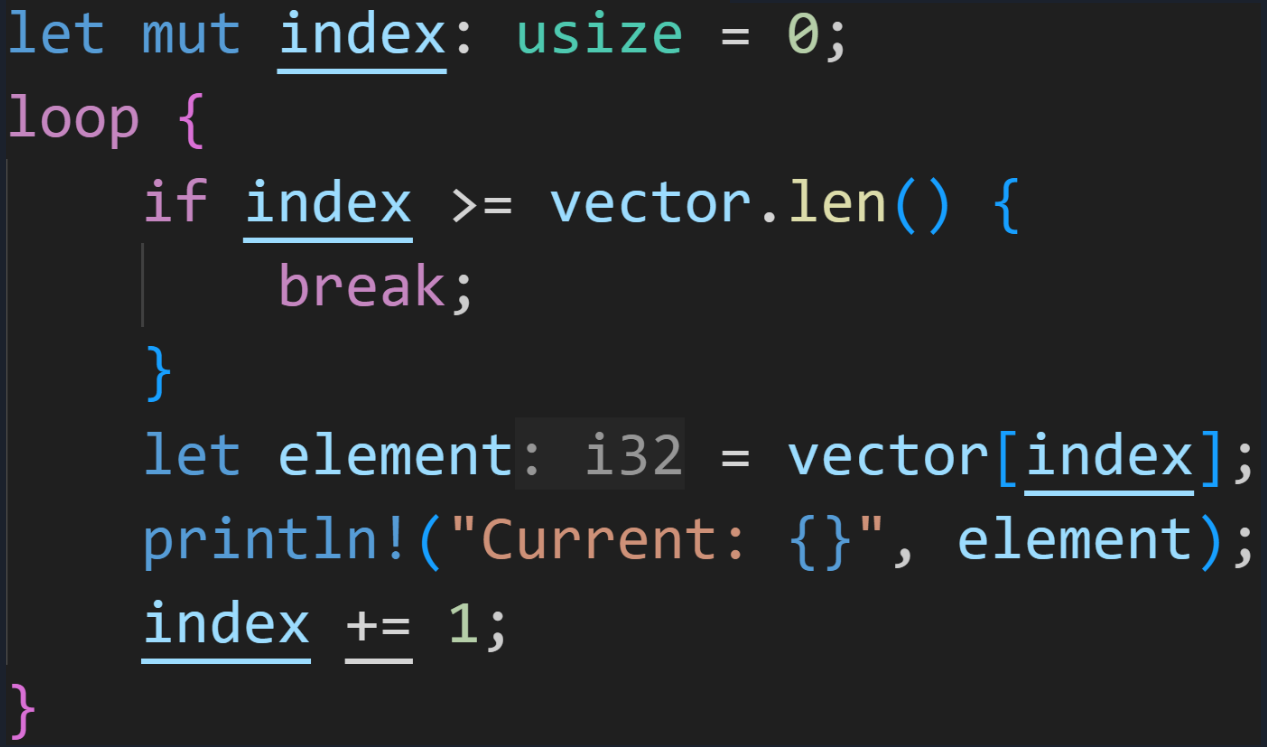
Where did our range go?

Damn, impressive! But we still didn‘t find loop :(

4. Loops - Extra

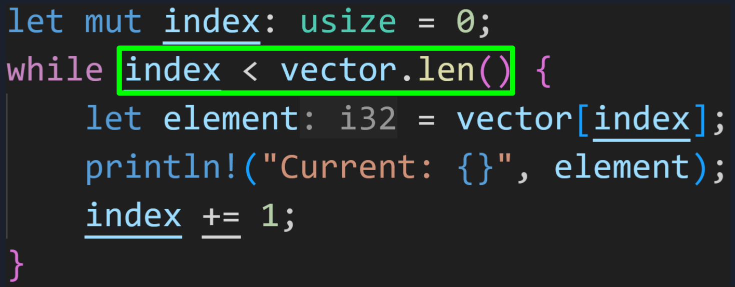


```rust
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

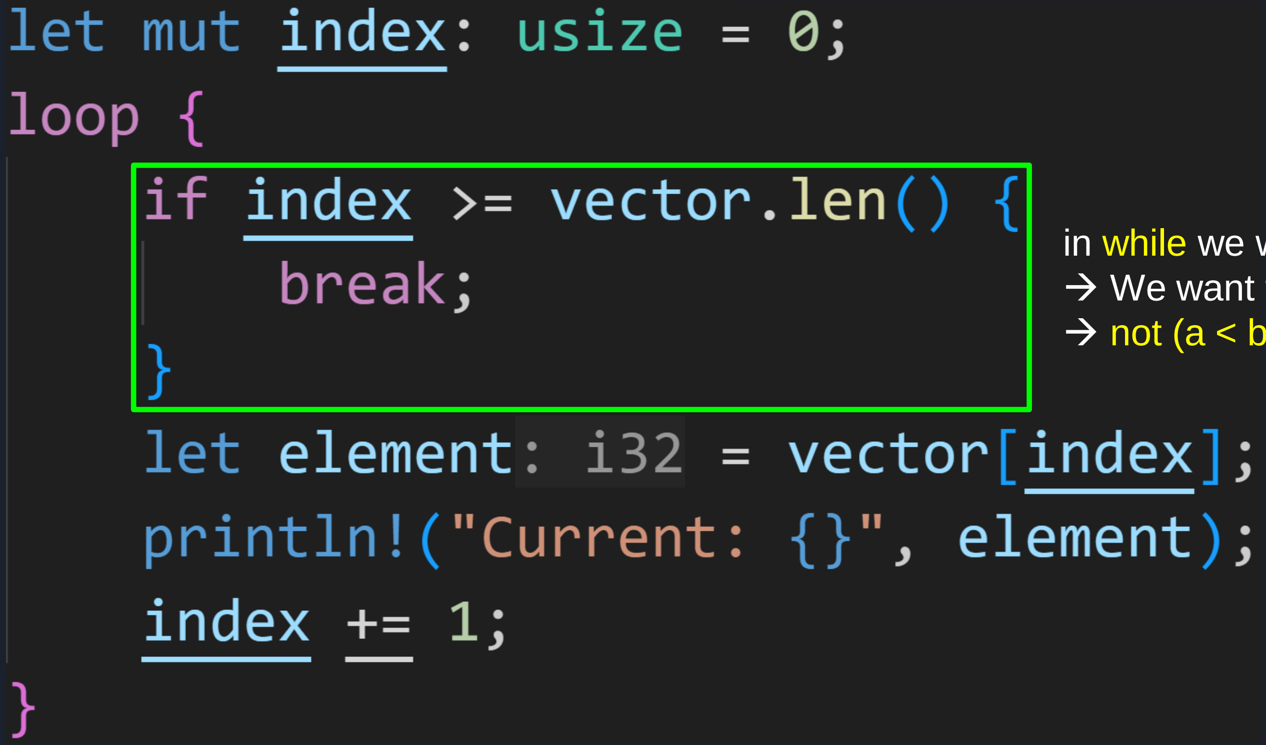


```rust
let mut index: usize = 0;
loop {
    if index >= vector.len() {
        break;
    }
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

4. Loops - Extra



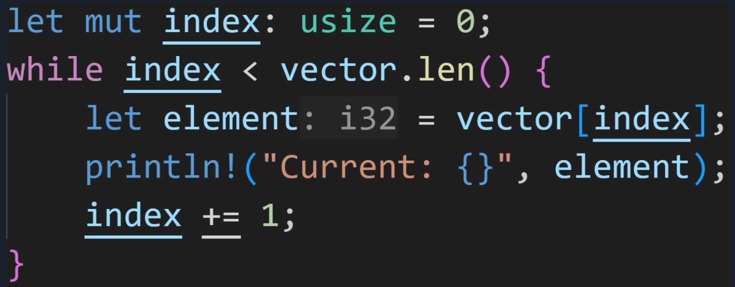
```
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```



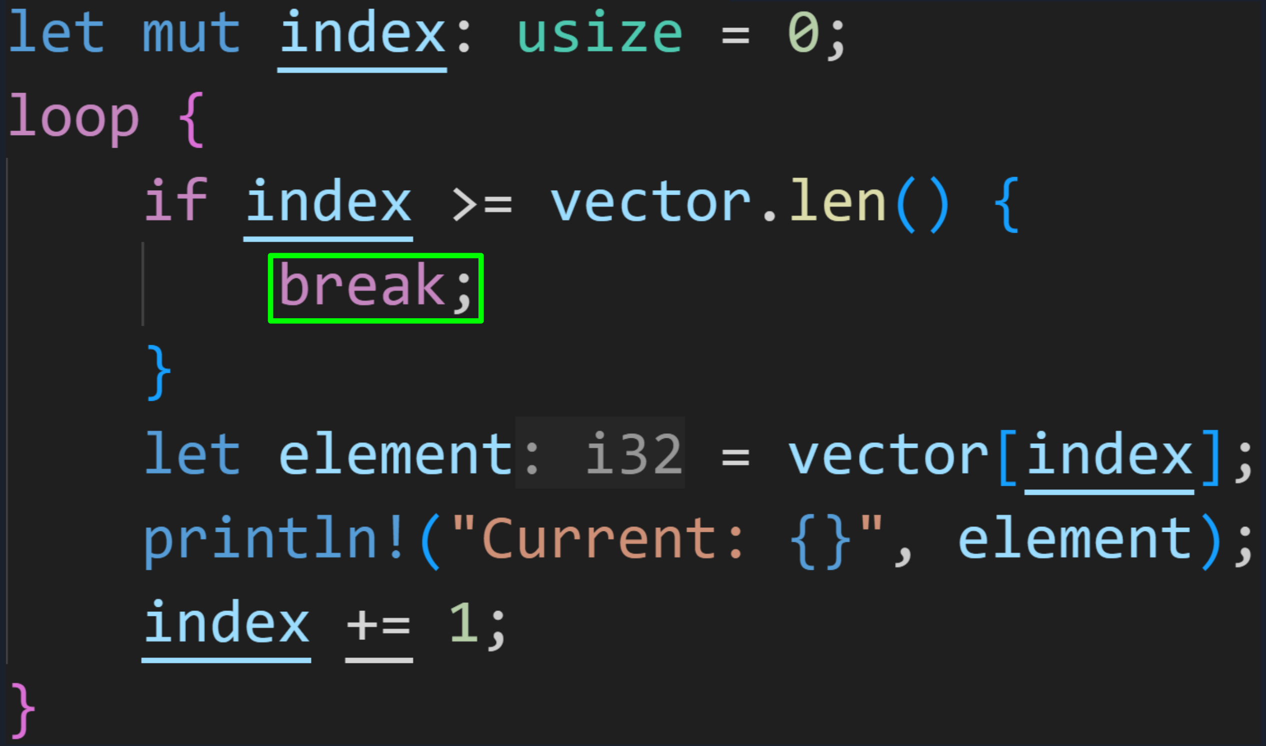
```
let mut index: usize = 0;
loop {
    if index >= vector.len() {
        break;
    }
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

in while we want to loop as long as the condition is true
→ We want to break the loop once the condition is false
→ not (a < b) → a >= b

4. Loops - Extra



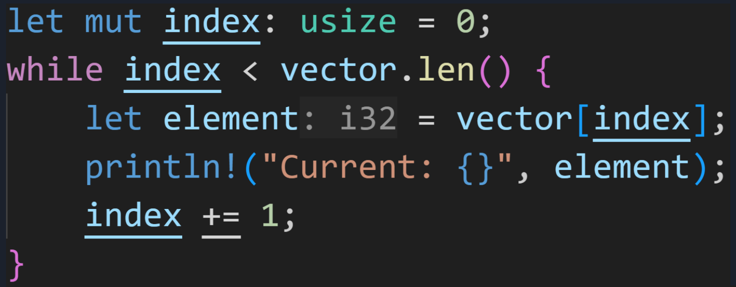
```
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```



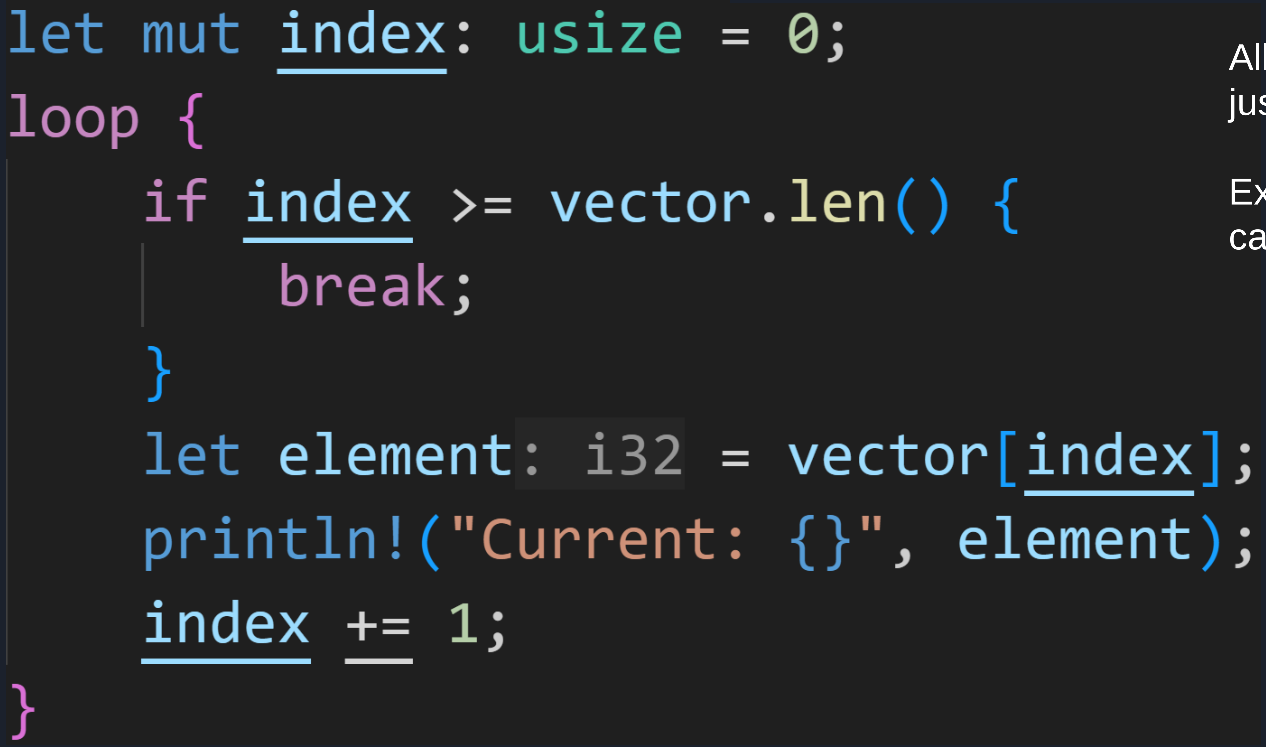
```
let mut index: usize = 0;
loop {
    if index >= vector.len() {
        break;
    }
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

break is a special keyword to break out of any loop, while or for
→ Program continues execution after loop-body
→ Here: After the pink brackets

4. Loops - Extra



```
let mut index: usize = 0;
while index < vector.len() {
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```



```
let mut index: usize = 0;
loop {
    if index >= vector.len() {
        break;
    }
    let element: i32 = vector[index];
    println!("Current: {}", element);
    index += 1;
}
```

Here we finally got loop!

All the previous loops are identical in what they do, just using different language constructs.

Exception: for moved the vector, which was not the case in while or loop

4. Loops - Extra

- You might think that there exists the „most powerful“ loop
- Indeed, some loops are more convenient than others, you would not use loop to iterate over a collection!
- However, all loops can be converted to each other → All loops are equally powerful!
- However, as we‘ve seen... It gets quite convoluted. Thank you for and while for existing :^)

3. Next time

- Ownership
- Borrow Checker
- References