

## Comments

```
// line: from double-slash until the end of the line
/* block: from open to close; multi-line & nested. */
/// line documentation for functions, structs, etc.
//! line documentation for crates and modules.
```

## Variables

### Binding names to values

- The "let" keyword binds a variable name to a value. (The variable becomes the owner of the value). The drop function destroys a value.
- A variable name can be bound to a different value – shadows the old value for the rest of the code-block.

```
let x = "outer"; // BINDING
{
    // new code-block
    let x = "inner"; // NEW BINDING, outer x shadowed
    let x = "inner2"; // NEW BINDING, inner x shadowed
} // end block, both inner x dropped
println!("{}", x); // prints "outer"
```

- Variables are immutable by default. The "mut" keyword is required for mutable variables.

```
let mut y = 5; // Mutable binding
y = 10; // Assignment: Same binding, new value
```

- The variable type must be known at compile time, but this can often be inferred by the compiler (as above).

### Static

- static defines a global variable with a fixed memory location. It has a 'static lifetime, which means the data lives for the entire duration of the program.

```
static HELLO: &str = "Hello, world!"; // &'static str
```

- Static variables must be declared at the global (module) level, not inside functions or blocks.

### Const

- const in Rust creates compile-time constants (which are not a variable) – the const value is computed during compilation and embedded directly into your code. A const does not reside as a variable in memory. Constants must be typed.

```
const PI_OVER_2: f32 = 3.14159265 / 2.0f32;
```

## Types

### Scalar types (stored on the stack)

- Scalar types are integers (either signed or unsigned), floating point numbers, characters and Booleans.
- The number after the numeric types indicates how many bits are used to represent the number. Note: isize and usize are pointer-sized integer types whose size depends on the target machine architecture.
- If not explicitly stated, the default integer type is i32. The default floating type is f64.

```
let a: i32 = -42; // i8, i16, i32, i64, i128, isize
let b: u32 = 42; // u8, u16, u32, u64, u128, usize
let c: f64 = 3.14; // f32, f64
let d: bool = true; // Boolean
let e: char = '★'; // Unicode character, single quotes
```

### Built-in (AKA primitive) compound types

- Tuples:** fixed length collection of values (can be of different types) – typically stored on the stack.

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
let (x, y, z) = tup; // destructuring
let one = tup.2; // index from zero accessible
let empty = (); // unit or empty tuple
```

- Arrays:** fixed length collection of values of the same type – typically stored on the stack.

```
let a = [1, 2, 3, 4, 5];
let months = ["January", "February", "March"];
let b: [i32; 5] = [1, 2, 3, 4, 5];
let repeated = [3; 5]; // [3, 3, 3, 3, 3]
let second_value = b[1]; // indexed from 0
```

- A **tuple struct** is a struct with fields that are accessed by position (like a tuple) rather than name. Values can be of a different type.

```
struct UserID(i32);
let uid = UserID(123456);
println!("User ID: {}", uid.0);

struct Point(i32, i32); // 2D point
let point = Point(10, 20);
let (x, y) = (point.0, point.1); // Access fields
let Point(x, y) = point; // Direct destructuring
```

### Nominal (named) types; AKA abstract data types

- Struct:** Like tuples, but with named fields – can encapsulate code as well as data using the impl keyword (discussed later). With encapsulated code, somewhat like classes in C++ (without inheritance).

```
struct Member {
    name: String,
    age: u32,
    is_active: bool,
}

let member_1 = Member {
    name: String::from("Alice Smith"),
    age: 31,
    is_active: true,
};
let member_2 = Member {
    name: String::from("Bob Johnson"),
    is_active: false,
    ..member_1 // moves member_1 for remaining fields
               // unless items implement the copy trait
};

let name = member_1.name;
```

- Enum:** one of multiple alternative data items/values. Unlike enums in other languages, Rust enums can hold data values (they are really tagged unions).

```
// enums without data values
enum Colour {
    Red,
    Green,
    Blue,
}
let favourite_colour = Colour::Red;

// enums with data values
enum Shape {
    Circle(f64), // radius
    Rectangle(f64, f64), // width, height
    Triangle(f64, f64), // base, height
}
let my_shape = Shape::Circle(2.5);
```

## Strings (introduction)

- String slices (type: `&str` – a borrowed string slice) – immutable by default – double quotes – fixed size. Like a pointer to a slice of Unicode characters from either a "literal string" known at compile time, or the contents of a String struct.

```
let greeting = "Hello, world!"; // Type: &str
let multiline = "This is a
multiline string";
```

- String – a built-in struct that manages mutable, ownable, variable length strings (stored on the heap).

```
// Creation
let empty = String::new();
let from_string_literal = String::from("hello");
let pre_alloc_capacity = String::with_capacity(100);
let conversion = "hello".to_string();

// Building
let mut s = String::from("Hello");
s.push('!'); // Add single character
s.push_str(" World!"); // Add string slice
s.insert(5, ','); // Insert char at index
s.insert_str(6, " beautiful"); // Insert string at index
s.remove(0); // Remove char at index

// Inspection
let s = String::from("Hello, 世界!"); // Hello world
let length_in_bytes = s.len();
let character_count = s.chars().count();
let is_empty = s.is_empty();
let current_capacity = s.capacity();

// Conversion
let as_string_slice = s.as_str();
let as_byte_slice = s.as_bytes();

// Case
let a = s.to_lowercase(); // returns new String
let a = s.to_uppercase(); // returns new String

// Trimming white space
let a = s.trim(); // both ends
let a = s.trim_start(); // beginning only
let a = s.trim_end(); // end only
let a = s.trim_matches(|c| c == ' '); // custom
let a = s.trim_start_matches("pre"); // remove prefix
let a = s.trim_end_matches("suf"); // remove suffix

// Searching and replacing
let b = s.contains("sub"); // substring exists
let b = s.starts_with("pre"); // prefix check
let b = s.ends_with("suf"); // suffix check
let position = s.find("pattern"); // first occurrence
let position = s.rfind("pattern"); // last occurrence
let a = s.replace("old", "new"); // replace all
let a = s.replacen("old", "new", 2); // first n only

// split - creates an iterator - by char ('c') or "str"
// Typically needs to be collected into a Vector
let text = "apple,banana,cherry";
let fruits: Vec<&str> = text.split(',').collect();
println!("Fruits: {}", fruits.join(", "));

let text = " Hello world! ";
let w: Vec<&str> = text.split_whitespace().collect();
println!("{}", w); // Output: ["Hello", "world!"]

// Slicing strings
let (s, start, end) = ("01234567890", 3, 7);
let o = s.get(start..end); //safe slice, returns Option

// Iteration over characters
let text = "Hello 世界";
for ch in text.chars() {
    println!("{}", ch);
}
```

## Data containers

- Rust has many useful data containers (that are built-in structs with methods), including dynamic arrays, hash-maps and hash-sets (similar to lists, dictionaries and sets in Python).

- Vectors – dynamic arrays

```
let mut vec: Vec<i32> = Vec::new();
let mut vec: Vec<String> = Vec::with_capacity(10);
let mut vec = vec![1, 2, 3]; // a useful macro

vec.push(4); // add to end
let item = vec.pop(); // take from end, returns Option
vec.insert(0, 0); // insert - (index, item)
vec.remove(0); // remove at index
let len = vec.len();
let item = vec[0]; // direct access (can panic)
let item = vec.get(0); // safe access, returns Option
vec.clear(); // remove all items

for item in &vec { println!("{}", item); } // borrow
for item in vec.iter_mut() { *item += 1; } // mut borrow
for (i, e) in vec.iter().enumerate() {
    println!("{}", i, e);
}
for item in vec { println!("{}", item); } // consume
```

- Hash Maps

```
use std::collections::HashMap; // import

// Think about borrowed vs owned keys and values
let mut map: HashMap<&str, &i32> = HashMap::new();
let mut map: HashMap<String, i32> = HashMap::new();
let mut map = HashMap::from([("key", "value")]);

map.insert("key", "value");
let value = map.get("key"); // returns Option<&V>
let value = map["key"]; // direct access (can panic)
map.remove("key");
let contains = map.contains_key("key");

map.entry("key").or_insert("val"); // insert if missing

// a counter
let mut cnt: HashMap<&str, i32> = HashMap::new();
cnt.entry("key").and_modify(|v| *v += 1).or_insert(1);

for (key, value) in &map {
    println!("{}", key, value);
}
for key in map.keys() { println!("{}", key); }
for value in map.values() { println!("{}", value); }
```

- Hash Sets

```
use std::collections::HashSet;

let mut set: HashSet<i32> = HashSet::new();
let mut set = HashSet::from([1, 2, 3]);

set.insert(4);
set.remove(&2);
let contains = set.contains(&3);
let len = set.len();
for item in &set { println!("{}", item); }
for (i, e) in set.iter().enumerate() {
    println!("{}", i, e);
}
for item in set { println!("{}", item); }

let s1 = HashSet::from([1, 2, 3]);
let s2 = HashSet::from([3, 4, 5]);

let union: HashSet<_> = s1.union(&s2).collect();
let intersection: HashSet<_> =
    s1.intersection(&s2).collect();
let diff: HashSet<_> = s1.difference(&s2).collect();
```

## Ownership and Borrowing (Pointers in Rust)

### Core Ownership Rules

- Values only have one variable owner at any given time
- If the owner goes out of scope, the value is dropped. Values can be dropped at any time manually.

```
let x = String::new(); // x owns the empty String value
drop(x);               // x is inaccessible after this line
```

- Values also go out of scope when the code execution leaves the code block {} in which they were declared.

### Ownership Transfer (Moving)

- With non-Copy types, ownership moves when you assign it or pass it to a function by value.
- Non-Copy types (eg. String, Vec, HashMap, custom structs as well as mutable references) are always moved, invalidating the original variable.

```
let v1 = vec![1, 2, 3];
let v2 = v1; // v1 is moved, only v2 is valid

let mut s = String::from("Hello sweetie");
let r1 = &mut s; // take a mutable reference to s
let r2 = r1;     // r1 is moved to r2, only r2 is valid

fn takes_ownership(s: String) {} //s dropped at close
fn main() {
    let mine = String::new();
    takes_ownership(mine); // ownership of mine moved
    // "mine" is no longer valid
}
```

- Note: rust has a "deep copy" .clone() method that many structs implement. It produces a new independent value, but it must be called explicitly.

```
let v = vec![1, 2, 3];
let w = v.clone(); // w is copied, both v and w valid
```

- Caution: cloning large data structures or repeated cloning in a loop can be time and space inefficient.
- Mutable references cannot be copied or cloned.

- Copy types: Values that implement the .copy() trait are copied and not moved. Note: primitive scalar types and immutable references are always copied.

```
let x = 5; // int, float, bool and char are primitive
let y = x; // x is copied, both x and y are valid

let s = String::from("Hello sweetie");
let r1 = &s; // take an immutable reference to s
let r2 = r1; // r1 is copied, now two references to s
```

### Borrowing Rules (apply to reference types &T, &mut T)

- Borrowing in Rust is like taking a reference in C/C++. A reference is a smart pointer to a value.
- **Immutable Borrowing (&T)**
  - Multiple immutable references simultaneously
  - Can read but cannot modify
  - Original owner retains ownership

```
let s = String::from("hello");
let r1 = &s; // immutable borrow
let r2 = &s; // multiple immutable borrows OK
println!("{}", r1, r2); // both valid
```

- While an immutable borrow is active, the owner cannot: mutate the value, move the value, nor create mutable borrows. The owner can read the value.

```
let mut x = vec![1, 2, 3];
let last = x.last().unwrap(); // immutable borrow &x[2]
x.push(4); // FORBIDDEN - owner cannot mutate x
println!("{}", last); // Rust cannot certify last ref
```

### Mutable Borrowing (&mut T)

- Only ONE mutable reference allowed at a time
- Cannot have immutable references while a mutable reference exists
- Can read and modify

```
let mut s = String::from("hello");
let r1 = &mut s; // mutable borrow
r1.push_str(" world");
```

- While a mutable borrow is active, the owner cannot: read the value, mutate the value, move the value, nor create any other borrows (mutable or not).

### Automatic dereferencing of borrowed items

- Like C/C++, Rust uses "\*" as the dereference operator.
- Unlike C/C++ (which uses "->"), Rust uses "." as the dereference for borrowed struct member values.
- Rust doesn't need the explicit "\*" dereference in many situations.

```
let owned = String::from("Hello friend");
let b = &owned; // immutable borrow

// These all work without explicit dereferencing:
println!("{}", b.len()); // Method calls auto-derefs
println!("{}", b);       // Display auto-derefs
let first_char = b[0..1]; // Indexing auto-dereferences
```

- But there are some situations where the explicit dereference is required:
  - Assignment through a reference

```
let mut owned = 5;
let mut borrowed = &mut owned;
*mut_borrowed = 10; // Explicit dereference to assign
```

- Comparison of referenced values

```
let owned = 5;
let borrow = &owned;

// Compare the value, not the reference
if *borrow == 5 { println!("Equal!"); }
```

### Common Patterns with ownership/borrowing

- Function parameters – prefer borrow to move:

```
fn process(s: &String) { } // borrowing (preferred)
fn take_ownership(s: String) { } // moving
```

- Returning references:

```
fn get_first(v: &Vec<i32>) -> &i32 {
    &v[0] // lifetime tied to input parameter
}
```

- Split borrows:

```
let mut v = vec![1, 2, 3];
let (first, rest) = v.split_at_mut(1); // note: coercion
// first references &mut [1], rest refs &mut [2, 3]
```

## (Reference) Lifetimes

### What are lifetimes?

Lifetimes are Rust's way of tracking how long references are valid. Every reference has a lifetime, even if not explicitly written. The compiler ensures references don't outlive the data they point to.

```
&'a T // Reference with lifetime 'a
&'a mut T // Mutable reference with lifetime 'a
```

## Function Lifetimes

When a function returns a reference, you must specify how input and output lifetimes relate:

```
// The returned reference lives as long as both inputs
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

// Different lifetimes - tied only to first param
fn first_word<'a, 'b>(s: &'a str, _other: &'b str)
-> &'a str {
    s.split_whitespace().next().unwrap_or("")
}
```

## Lifetime Elision Rules (when you can omit them)

The compiler automatically infers lifetimes that are obvious from the context, where:

1. Each input reference gets its own lifetime
2. If there's one input lifetime, output gets the same lifetime
3. If there's &self or &mut self, output gets self's lifetime

```
fn print(s: &str)           // Elided
fn print<'a>(s: &'a str)    // Explicit equivalent

fn get_first(s: &str) -> &str           // Elided
fn get_first<'a>(s: &'a str) -> &'a str // Explicit

impl Thing {
    fn get_name(&self) -> &str           // Elided (rule 3)
    fn get_name<'a>(&'a self) -> &'a str // Explicit
}
```

## Struct Lifetimes

Structs storing references need lifetime parameters:

```
struct Book<'a> {
    // These references must live as long as Book
    title: &'a str,
    auth: &'a str,
}

impl<'a> Book<'a> {
    fn new(title: &'a str, auth: &'a str) -> Book<'a> {
        Book { title, auth }
    }
}

let title = String::from("Rust");
let auth = String::from("Klabnik");
let book = Book::new(&title, &auth);
// book cannot outlive title or auth
```

## Static Lifetime

'static means the reference lives for the entire program:

```
let s: &'static str = "literals have static lifetime";

// T: 'static - T contains no non-static references
fn needs_static<T: 'static>(t: T) {}
```

## Common Patterns

```
// Multiple lifetimes with constraints
fn h<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    // 'b: 'a means 'b lives at least as long as 'a
    if x.len() > 5 { x } else { y }
}

// Lifetime in where clause
fn where_example<'a, T>(item: &'a T) -> &'a T
where
    T: std::fmt::Display + 'a
{
    println!("{}", item);
    item
}
```

## Operators

### Arithmetic operators

```
let a = 10;
let b = 3;

// Basic arithmetic
let sum = a + b;           // 13
let difference = a - b;    // 7
let product = a * b;       // 30
let quotient = a / b;      // 3 (integer division)
let remainder = a % b;     // 1

// Floating point division
let (x, y) = (10.0, 3.0);
let float_quotient = x / y; // 3.333...

// Unary operators
let negative = -5;
```

- Note: there are no implicit type conversions for numeric values – you must explicitly cast types to the same type for arithmetic operations.

```
// Numeric conversions
let x = 42i32;           // i32
let y = x as f64;        // Cast to f64
let z = y as u32;        // Cast to u32
let byte = z as u8;      // Cast to u8
```

- But Rust can use context to infer the types of literals.

```
let x: f32 = 3.14; // 3.14 is inferred to be f32
let y = x + 2.0;   // 2.0 is inferred as f32 to match x
```

- Note: the std crate includes many maths functions:
  - Integer methods – abs(), pow(), etc.
  - Floating methods – sqrt(), sin(), exp(), ln(), etc.
  - Min/Max/Rounding – min(), max(), round(), etc.

### Assignment operators

```
let mut x = 5;

// Basic assignment
x = 10;

// Compound assignment operators
x += 3; // x = x + 3
x -= 2; // x = x - 2
x *= 4; // x = x * 4
x /= 2; // x = x / 2
x %= 3; // x = x % 3

// Bitwise compound assignment
x &= 0b1010; // x = x & 0b1010
x |= 0b0101; // x = x | 0b0101
x ^= 0b1111; // x = x ^ 0b1111
x <<= 2;     // x = x << 2
x >>= 1;     // x = x >> 1
```

### Comparison operators

```
let (a, b) = (5, 10);

// Equality and inequality
let equal = a == b; // false
let not_equal = a != b; // true

// Ordering
let less = a < b; // true
let greater = a > b; // false
let less_equal = a <= b; // true
let greater_equal = a >= b; // false

// String comparison
let s1 = "apple";
let s2 = "banana";
let str_less = s1 < s2; // true (lexicographical)
```

## Logical Operators

```
let a = true;
let b = false;

// Logical AND (short-circuiting)
let and_result = a && b; // false

// Logical OR (short-circuiting)
let or_result = a || b; // true

// Logical NOT
let not_a = !a; // false
let not_b = !b; // true
```

- In Rust, the logical operators && (and) and || (or) are lazy (short-circuited). That means they only evaluate the right-hand side if they really need to (same as C/C++).

## Bitwise operators

```
let a = 0b1010; // 10 in binary
let b = 0b1100; // 12 in binary

// Bitwise AND
let and = a & b; // 0b1000 (8)

// Bitwise OR
let or = a | b; // 0b1110 (14)

// Bitwise XOR
let xor = a ^ b; // 0b0110 (6)

// Bitwise NOT
let not_a = !a; // Flips all bits

// Bit shifting
let left_shift = a << 2; // 0b101000 (40)
let right_shift = a >> 1; // 0b0101 (5)
```

## Range operators

```
// Inclusive range (includes end)
let inclusive = 1..=5; // 1, 2, 3, 4, 5

// Exclusive range (excludes end)
let exclusive = 1..5; // 1, 2, 3, 4

// Usage in loops
for i in 1..=3 {
    println!("{}", i); // Prints 1, 2, 3
}

// Slicing
let arr = [1, 2, 3, 4, 5];
let slice = &arr[1..4]; // [2, 3, 4]
let slice2 = &arr[..3]; // [1, 2, 3] (from start)
let slice3 = &arr[2..]; // [3, 4, 5] (to end)
let slice4 = &arr[..]; // [1, 2, 3, 4, 5] (all)
```

## Reference and dereference operators (&, \*)

```
let x = 5;
let r = &x; // Create a reference to x
let val = *r; // Dereference: get the value r points to

let mut y = 10;
let mr = &mut y; // Mutable reference
*m_r = 20; // Dereference and assign
```

## Index operator – []

```
let mut arr = [1, 2, 3, 4, 5];
let first = arr[0]; // 1 – copied assignment
arr[0] = 100; // Replace first value
```

## Field access operator - .

```
struct Point { x: f64, y: f64 }
let point = Point { x: 5, y: 10 };
let x_coord = point.x; // Field access - dot operator
```

## Flow Control

### Statements, expressions and code blocks

- Statements** in Rust are semi-colon terminated.
- Expressions**, evaluate to a value, and do not have semi-colon terminators.
- A **code block** is just a chunk of code wrapped in curly braces {}. Code blocks can return a value, if the last element in the code block is an expression. If the last element is a statement, the unit tuple is returned ().  
Note: multi-line blocks are indented four spaces.

```
let y = {
    let a = 2;
    let b = 3;
    a + b // no semicolon, value of block is returned
};
```

### Functions (and function pointers)

- Functions are a key unit of flow control. Every program starts with a "main" function.

```
fn greet(name: &str) { // declaring the greet function
    println!("Hello, {}!", name);
}

fn main() { // declaring the main function
    greet("Alice"); // calling the greet function
}
```

- In idiomatic Rust, functions that return a value, are written as a code block that ends with an expression.
- A return statement exists and is used for the early exit from a function.

```
fn check(n: i32) -> &'static str {
    if n < 0 {
        return "negative";
    }
    "non-negative" // No semicolon! Returned expression
}
```

- Arguments are passed by value or by reference.

```
fn pass_by_value(n: i32) {} // takes a value
fn pass_by_reference(n: &i32) {} // pass by reference
fn pass_mut_ref(n: &mut i32) {} // pass by mut ref
```

- Tricky: every function is its own unique type. But Rust allows for more generalised function pointers.

```
fn hello() { println!("Hello!"); } // a function item
let f: fn() = hello; // function coerces to fn pointer
f(); // call through the fn pointer

fn run(f: fn()) { // function pointer parameter
    f();
}

// other example function pointer types
fn() -> i32 // no params, returns i32
fn(&i32) // takes i32 reference, returns ()
fn(i32, f64) -> i32 // takes i32 and f64, returns i32
```

### Macros

- Rust has built-in macros (they look like function calls, with a name ending with a bang!). Macros are expanded into Rust code at compile time.
- Commonly used built-in macros include:

```
let numbers = vec![1, 2, 3]; // create a Vector
let s = format!("{}", + {} = {}", 2, 3, 5); // Strings

println!("Hello, {}!", "world"); // also print!()
dbg!(2 + 2); // provide a debug print with line number
// Also assert!(true); assert_eq!(2+2, 4); panic!("{}", s);
```



## Raising an error

- There are no exceptions in Rust. If a function can fail, typically it returns an `Option` or `Result` enum.

```
fn divide(x: i32, y: i32) -> Result<f64, String> {
    if y == 0 { //returns Err if asked to divide by zero
        Err("Cannot divide by zero".to_string())
    } else {
        Ok(x as f64 / y as f64)
    }
}

fn divide2(x: i32, y: i32) -> Option<f64> {
    match y { //returns None if asked to divide by zero
        0 => None,
        _ => Some(x as f64 / y as f64),
    }
}
```

- Note: Options are also useful as struct members.

## Handling errors

- .unwrap() or .expect()** methods: These cause your program to panic if an error is detected. Note: Useful tools when prototyping, best not used in production.

```
let number: i32 = "42".parse()
    .expect("Failed to parse the number");
println!("The number is {}", number); // will be 42
```

- The `unwrap` method does not provide a message.

```
let number: i32 = "42".parse().unwrap();
println!("The number is {}", number); // will be 42
```

- .unwrap\_or()** can be used to provide a default.

```
let number: i32 = "abc".parse().unwrap_or(0);
println!("Parsed number = {}", number); // will be 0
```

- .unwrap\_or\_else()** allows you to compute the default dynamically. This often done in a closure (see below).

```
let number: i32 = "abc".parse().unwrap_or_else(|| {
    println!("Value was missing, using default");
    42
});
println!("Parsed number = {}", number); // will be 42
```

- ? (error propagation) operator**: the `?` operator unwraps the value. If `None/Err` it returns early.

```
fn get_first_char(s: &str) -> Option<char> {
    s.chars().next()
}

fn get_first_two(s: &str) -> Option<(char, char)> {
    let first = get_first_char(s)?;
    let second = get_first_char(&s[1])?;
    Some((first, second))
}
```

- ? (error propagation) operator with From**: The `From` implementations tell Rust how to convert other error types into your error type, making `?` work seamlessly.

```
#[derive(Debug)]
enum MyError {
    Io(std::io::Error),
    Parse(std::num::ParseIntError),
}

impl From<std::io::Error> for MyError {
    fn from(err: std::io::Error) -> Self {
        MyError::Io(err)
    }
}

impl From<std::num::ParseIntError> for MyError {
    fn from(err: std::num::ParseIntError) -> Self {
        MyError::Parse(err)
    }
}

// Now ? automatically converts errors to MyError
fn read_number() -> Result<i32, MyError> {
    let text = std::fs::read_to_string("file.txt")?;
    let num = text.trim().parse()?;
    Ok(num)
}
```

- Repackaging**: Sometimes you will need to repackage an error to the correct type (with the `.map_err()` method and an appropriate closure) for the current function before propagation.

```
use std::fs;
use std::io;

fn read_number_from_file(path: &str) -> io::Result<i32> {
    let number = fs::read_to_string(path)?
        .trim()
        .parse()
        .map_err(|e|
            io::Error::new(io::ErrorKind::InvalidData, e)
        );
    Ok(number)
}

fn main() -> io::Result<()> {
    let n = read_number_from_file("number.txt");
    println!("Number: {}", n);
    Ok(())
}
```

- Match (see below)** can be used to handle errors:

```
use std::fs::File;
use std::io::{self, Read};

fn main() {
    let result = File::open("hello.txt");

    match result {
        Ok(mut file) => {
            let mut contents = String::new();
            file.read_to_string(&mut contents)
                .expect("Failed to read file.");
            println!("File contents:\n{}", contents);
        }
        Err(e) => {
            println!("Failed to open file: {}", e);
        }
    }
}
```

## If expressions (note: no semicolons inside the curly {})

```
let num = 5;
let r = if num > 0 {"positive"} else {"not positive"};
// Note all expressions must return the same type
```

## If statements (the following example should be a match)

```
if number < 0 {
    println!("Negative");
} else if number == 0 {
    println!("Zero");
} else if number < 10 {
    println!("Small positive");
} else {
    println!("Large positive");
}
```

## While loops

```
let mut n = 0;
while n < 10 {
    if n == 5 {
        break; // exit the loop completely
    }
    if n % 2 == 0 {
        n += 1;
        continue; // skip printing even numbers
    }
    println!("{}", n);
    n += 1;
}
println!("Loop stopped at n = {}", n);
```

## Loop until you break

```
let mut count = 0;
loop {
  println!("count = {}", count);
  count += 1;
  if count == 3 {
    break; // exit loop
  }
}
```

- You can break with a value

```
let result = loop {
  let x = 2 + 2;
  break x; // returns 4
};
```

## For loops over things that are iterable

```
let numbers = [10, 20, 30];
for n in numbers {
  println!("n = {}", n);
}
```

- Also, you can loop over ranges

```
for i in 0..5 { // 0,1,2,3,4
  println!("{}", i);
}
```

## Match – must cover all possible cases

```
let n = 2;
match n {
  1 => println!("one"),
  2 | 3 => println!("two or three"), // or
  4..6 => println!("between 4 and 6"), // ranges
  n @ 7..9 => println!("{}", n), // @ binding
  _ => println!("something else"), // default
}
```

- Match guards and conditional matching

```
// match
let result = match n {
  x if x % 2 == 0 => "even", // conditional match
  _ => "odd",
};
println!("{}", n, result);
```

- Tuple destructuring in a match

```
let pair = (0, -2);
match pair {
  (0, y) => println!("First is 0, second = {}", y),
  (x, 0) => println!("Second is 0, first = {}", x),
  _ => println!("No zeros"),
}
```

- Matching enums

```
enum Shape {
  Circle(f64),
  Rectangle(f64, f64),
}
let shape = Shape::Circle(2.0);

let (name, area) = match shape {
  Shape::Circle(r) => ("Circle", 3.14159265 * r * r),
  Shape::Rectangle(w, h) => ("Rectangle", w * h),
};
println!("{}", shape, "has area of {} sq units.", name, area);
```

## if let – match on a single value

```
let some_value = Some(42);
if let Some(x) = some_value { // Will not match None
  println!("Got value: {}", x);
}
```

## While let – loop while matching

```
let mut stack = vec![1, 2, 3, 4, 5];

// Keep popping until the vector is empty
while let Some(value) = stack.pop() {
  println!("Popped: {}", value);
}
```

## Closures – anonymous functions

- Closures are anonymous functions that can both take parameters (within vertical bars) and capture variables from their surrounding environment.

```
let x = 5;
let by_ref = || println!("{}", x); // borrows x
let by_move = move || println!("{}", x); // moves x

let (x, y) = (5, 8);
println!("{}", (|a: &i32| a + y)(x)); // borrows x & y
println!("{}", (|a| a + y)(x)); // moves x / br y

let my_addition = |a, b| a + b;
println!("{}", my_addition(2, 3)); // infers i32
println!("{}", my_addition(5u64, 10)); // infers u64

// Many methods take closures as an argument
let even = vec![1, 2, 3, 4, 5]
  .into_iter() // consuming iterator
  .filter(|x| x % 2 == 0) // keep evens
  .collect(); // collect in a vector
```

## Iterators (often better than for loops)

- Creating iterators

```
let v = vec![1, 2, 3];
v.iter() // Iterator over &T (borrows)
v.iter_mut() // Iterator over &mut T (mut borrows)
v.into_iter() // Iterator over T (takes ownership)

// From ranges
(0..10) // 0 to 9
(0..=10) // 0 to 10 inclusive

// From arrays/slices
[1, 2, 3].iter()
"hello".chars() // Iterator over characters
"hello".bytes() // Iterator over bytes
```

- Consuming iterators – Note: you cannot modify a collection while iterating over it.

```
// Collectors
iter.collect::<Vec<T>>() // Collect into Vec
iter.collect::<HashSet<T>>() // Collect into HashSet

// Single value consumers
iter.sum::<i32>() // Sum all elements
iter.count() // Count elements
iter.nth(5) // Get element at index 5
iter.last() // Get last element
iter.find(|x| x > &5) // Find first matching
iter.max() / iter.min() // Maximum/minimum

// Boolean consumers
iter.all(|x| x > 0) // True if all match
iter.any(|x| x > 0) // True if any match

Iterator adapters (chainable)
// Transforming
iter.map(|x| x * 2) // Transform each element
iter.filter(|x| x % 2 == 0) // Keep matching elements

// Taking/skipping
iter.take(5) // Take first 5 elements
iter.skip(5) // Skip first 5 elements
iter.take_while(|x| x < &10) // Take while condition
iter.skip_while(|x| x < &10) // Skip while condition
```

```
// Combining
iter.chain(other_iter)    // Concatenate iterators
iter.zip(other_iter)      // Pair up elements: (a, b)
iter.enumerate()          // Add index: (index, value)

// Folding - reduce to a single value
iter.fold(0, |acc, x| acc + x) // An accumulator

// Scanning - result for each step in the process
iter.scan(0, |state, x| {    // Cumulative sum
    *state += x;
    Some(*state)
})

// Reversing
iter.rev() // rev() requires a double-ended iterator
```

### Common patterns

```
// Process and collect
let results: Result<Vec<_>, _> =
    items.iter()
        .map(|x| process(x)) // .map(process)
        .collect();          // Stops on first Err

// Window/chunk operations
v.windows(2) // Sliding window: [[1,2], [2,3], ...]
v.chunks(2)  // Non-overlapping: [[1,2], [3,4], ...]

// Partition into two collections
let (evens, odds): (Vec<_>, Vec<_>) =
    nums.iter().partition(|x| x % 2 == 0);
```

### Implementation Blocks (impl)

- impl blocks let you define methods and associated functions for types (often structs and enums, but they are not limited to structs and enums).

```
struct Rectangle {
    width: f64,
    height: f64,
}

impl Rectangle {
    // Associated function (called with ::)
    fn new(width: f64, height: f64) -> Rectangle {
        Rectangle { width, height }
    }

    fn area(&self) -> f64 { // Method (called with .)
        self.width * self.height
    }

    fn destroy(self) { // Method that consumes self
        println!("Rectangle destroyed!");
        // self is moved and dropped
    }
}
```

- You can have multiple implementation blocks for the same struct or enum.

```
impl Rectangle {
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

### Traits

- Traits are guarantees for shared behaviour across types. They are similar to interfaces in Java and Go.

```
trait Speak { // provide signature OR default method
    fn speak(&self) -> String; // Signature

    // fn speak(&self) -> String { // Default method
    //     "(silence)...".to_string()
    // }
}
```

```
// Implement the trait for Dog
struct Dog;
impl Speak for Dog {
    fn speak(&self) -> String {
        "Woof!".to_string()
    }
}

// Implement the trait for Cat
struct Cat;
impl Speak for Cat {
    fn speak(&self) -> String {
        "Meow!".to_string()
    }
}

// Generic function for ANY type implementing Speak
fn animal_talk<T: Speak>(animal: T) {
    println!("{}", animal.speak());
}

fn main() {
    let d = Dog;
    let c = Cat;

    animal_talk(d); // Woof!
    animal_talk(c); // Meow!
}
```

### Standard traits and the derive macro

- Rust has many standard traits that you can implement. For example, you can implement the Add trait, to get operator overloading. The big four ...

```
// Display - User-facing string representation
impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

// Debug - Developer-facing (and usually derived)
#[derive(Debug)] // Gives you {:?} formatting

// Default - Provides default values
impl Default for Config {
    fn default() -> Self {
        Config { timeout: 30, retries: 3 }
    }
}
let cfg = Config::default();

// From/Into - Type conversions
impl From<i32> for Number {
    fn from(item: i32) -> Self {
        Number { value: item }
    }
}
let num = Number::from(42);
let num: Number = 42.into(); // Into comes with From
```

- Derive macros automatically implement standard traits for your types, reducing boilerplate code.

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
struct Point {
    x: i64,
    y: i64,
}
// Clone - enables explicit duplication with .clone()
// Copy - enables implicit copying (stack-only types)
// Debug - enables formatting with {:?} and {:#?}
// Default - provide fn default() for a default value
// Hash - enables hashing
// PartialEq - enables == and != operators if possible
// Eq - requires PartialEq - rigorous/total equality
// PartialOrd - enables comparison where possible
// Ord - rigorous comparison and ordering
```



## Generics

- Generics let you write a function, struct, enum or trait once, and use it with different types.

```
// struct example
struct Point<T> {
    x: T,
    y: T,
}
let int_point = Point { x: 5, y: -5 };
let float_point = Point{ x: 1.2, y: -2.1 };

// function example - with trait binding
fn biggest<T: PartialOrd + Copy>(list: &[T])
-> Option<T> {

    if list.is_empty() {
        return None;
    }
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    Some(largest)
}
```

## Extending existing types in Rust

- The Problem:** You can't add methods directly to types you don't own (like Vec, String, etc.) due to Rust's orphan rule (which says, you can only implement a trait for a type if you own either the trait OR the type (or both)).
- Thin wrapper types** - give you full control to write your own implementation blocks, but you lose automatic type coercion. BUT a bit clunky.

```
struct MyVec<T>(Vec<T>); // generic tuple struct

impl<T> MyVec<T> {
    fn double_len(&self) -> usize { self.0.len() * 2 }
    fn push(&mut self, item: T) { self.0.push(item); }
}
```

- Extension traits** allow you to write a new implementation block for an existing type. But to use the trait you need an explicit "use" statement.

```
trait StringExt {
    fn is_palindrome(&self) -> bool;
}

impl StringExt for String {
    fn is_palindrome(&self) -> bool {
        self == &self.chars().rev().collect::<String>()
    }
}

// Usage: must import trait / can be in same file
use StringExt;
let s = String::from("racecar");
println!("{}", s.is_palindrome()); // true
```

## Smart pointers

- Smart pointers in Rust are data structures that act like pointers but add extra features for memory management, ownership, mutability, or thread safety.
- Box<T> is the simplest smart pointer. It puts a value on the heap instead of the stack. It is automatically freed when it goes out of scope.

```
let b = Box::new(5);
println!("{}", *b); // dereference to get 5
```

- Note: in the above example, although we can dereference b (because it implements the Deref trait), it is not really a reference – it is Box<i32> object that is owned by b.

- Rc<T> is reference counted, allowing multiple ownership. By keeping count of how many Rcs point to the same value. The value is freed when the last Rc goes out of scope.

```
use std::rc::Rc;

let a = Rc::new(String::from("hello"));
let b = Rc::clone(&a); // increase reference count
```

- Arc<T> is like Rc<T>, but thread safe because it uses atomic operations (that happen all at once).
- Related objects: Cell<T> and RefCell<T> which allows for the run-time mutability of immutable objects. Mutex<T> ensures only one thread can access the data at a time. RwLock<T> allows multiple readers or one writer.

## Modules, Libraries and Crates

### Modules

- A module is Rust's way to organize code into namespaces and control visibility (privacy). It allows one to group related functionality together.
- Declaration:** modules can be declared in three ways
  - As an inline module code-block
  - Every file.rs is a module
  - Every directory with mod.rs is a module

```
mod my_module {...} // Inline module named my_module
mod utils; // File-based (utils.rs)
mod network; // Directory-based (network/mod.rs)
```

### Visibility

```
pub fn public() {} // Public to parent
fn private() {} // Private (default)
pub(crate) fn crate_only() {} // Visible within crate
pub(super) fn parent_only() {} // Visible to parent only
```

### Use

```
my_module::function(); // Full path
use my_module::function; // Import single item
use my_module::{a, b, c} // Import multiple items
use my_module::*; // Import all public (avoid)
use super::parent_item; // From parent module
use crate::root_item; // From crate root
```

### File structure

```
src/
├─ main.rs (and/or lib.rs) // Crate/Project root
├─ utils.rs // Simple module
└─ network/ // Directory module
    ├─ mod.rs // Module entry point
    └─ client.rs // Submodule
```

- Note: main.rs is used for executables that contain a main() function, and the lib.rs filename is used as the root file for libraries that do not contain a main() function. Both might be present when you have an executable that uses the library code.

## Libraries and external crates

- Rust has a standard library that provides extra functionality, and there are many publicly available crates (or libraries) of code on the crates.io website.
- You import the standard library and external crates with the "use" keyword. For example:

```
// for collections
use std::collections::{HashMap, HashSet, BTreeMap};

// for IO and the file system
use std::fs;
use std::io::{self, Read, Write};
use std::path::Path;

// for networking
use std::net::{TcpListener, TcpStream, UdpSocket};
```

- To use library packages from the crates.io website, you will need to first identify the package names in a cargo.toml file in the root directory of your project, under a [dependencies] heading.
- For a project that uses the random number package, you might have a cargo.toml file like this.

```
[package]
name = "rand-example"
version = "0.1.0"
edition = "2024" # latest rust - released Feb 2025

[dependencies]
rand = "0.8"
```

- Inside your code it might look like this.

```
use rand::{thread_rng, Rng};
use rand::seq::SliceRandom; // for .choose()

fn main() {
    let mut rng = thread_rng();
    println!("Dice: {}", rng.gen_range(1..=6));
    println!("Float: {:.2}", rng.gen::
```

- Running cargo build or cargo run will automatically download, compile, and link the dependency.

## Testing

### Basics

- Rust has built-in support for testing with the #[test] attribute. You write test functions inside a #[cfg(test)] module, and run them with cargo test.

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn adds_two_numbers() {
        assert_eq!(add(2, 3), 5);
    }

    #[test]
    #[should_panic(expected = "specific message")]
    fn test_specific_panic() {
        panic!("specific message");
    }
}
```

- The assert! family of macros are the backbone of testing in Rust. Their role is simple: they check that some condition holds true during a test, and if it doesn't, the test fails.
- Note: you can have more than one mod tests (or any number of #[cfg(test)] modules) in a single Rust file.
- Unit tests should appear in the same file as the functions/methods being tested.
- Integration tests should live in a tests/ subdirectory in the src directory.

- Tests are run from the command line using cargo.

```
cargo test
```

## Compiler attributes

### Attributes

- Attributes are code annotations that instruct the compiler, tools, or runtime about how to handle your code. They're written with a #[...] or #![...] syntax, and are essential for many Rust features.

- Examples follow:

```
#[allow(dead_code)] // suppress unused function warning
fn unused() {}

#[cfg(target_os = "linux")] // Conditional compilation
fn run_on_linux() {}

#[derive(Debug, Clone, PartialEq)] // Derive traits
struct Point { x: i32, y: i32 }

// This next one is a crate-level annotation
#![deny(warnings)] // fail build on any warning
```

### Built in attributes you will see often

```
#[derive(...)] // Auto-implemented traits
#[cfg(...)] // Conditional compilation
#[test] // Mark test functions
#[allow(warning)] // Suppress specific warnings
#[repr(...)] // Control memory layout
#[inline] // Inlining hints
#[must_use] // Warn about unused results
#[deprecated] // Mark as deprecated
```

## Concurrency / Parallelism

### Some definitions

- **Parallelism** means your code is running on multiple CPUs at the same time (it's about execution). The standard threads library and the third-party Rayon crate offer strong parallelism support. Best for mathematical computations, data processing, CPU-heavy algorithms where you can divide work across multiple cores.
- **Concurrency** means dealing with multiple things at the same time whether that is happening on one CPU with context switching, or multiple CPUs (or perhaps many tasks running on just a few CPUs with some context switching). It's about program design (task separation and coordination), without necessarily a commitment to parallelism. Tokio and smol are two crates that support concurrency. Best for managing tasks that can spend a lot of time waiting for I/O.

## Parallelism – Threads and Communication

- Basic thread spawning

```
use std::{thread, num::NonZeroUsize};
use std::time::Duration;

let cpu_count = thread::available_parallelism();
println!("CPU count: {}", cpu_count.unwrap_or(
    NonZeroUsize::new(1).unwrap()));

let handle = thread::spawn(|| {
    println!("Hello from thread!");
    // simulate work ...
    thread::sleep(Duration::from_millis(100));
    42 // return value
});
let result = handle.join().unwrap(); // wait/get result
```

- The `.join()` method in threading is used to block the calling thread until the thread on which `.join()` is called completes execution.

- Thread communication via channels

```
use std::sync::mpsc; // multi-producer, single-consumer

let (tx, rx) = mpsc::channel();
let tx2 = tx.clone(); // multiple senders allowed

thread::spawn(move || {
    tx.send("Hello").unwrap(); // move tx into thread
});
thread::spawn(move || {
    tx2.send("World").unwrap();
});

// Receive messages (blocks until available)
println!("{}", rx.recv().unwrap());
println!("{}", rx.recv().unwrap());
```

- Shared state with `Arc<Mutex<T>>`

```
use std::sync::{Arc, Mutex};

let counter = Arc::new(Mutex::new(0));
let counter2 = Arc::clone(&counter);
thread::spawn(move || {
    let mut num = counter2.lock().unwrap();
    *num += 1; // safely modify shared data
});
```

## Concurrency using Tokio, async and await

```
// Async/await example (requires tokio or async-std)
use tokio::time::{sleep, Duration};

async fn fetch_data() -> String {
    sleep(Duration::from_millis(100)).await;
    "Data ready".to_string()
}

#[tokio::main] // macro creates runtime
async fn main() {
    let result = fetch_data().await; // wait until ready
    println!("{}", result);
    // Run multiple tasks concurrently
    let (r1, r2) = tokio::join!(
        fetch_data(),
        fetch_data()
    ); // both run in parallel
    println!("{}", r1 and r2);
}
```

## Example for static and dynamic dispatch

```
trait Animal {
    fn speak(&self);
}

struct Dog;
struct Cat;

impl Animal for Dog {
    fn speak(&self) { println!("Woof!"); }
}

impl Animal for Cat {
    fn speak(&self) { println!("Meow!"); }
}

// Decide at compile-time which method to use
fn static_sound<T: Animal>(animal: &T) {
    animal.speak();
}

// Decide at run-time which method to use
fn dynamic_sound(animal: &dyn Animal) {
    animal.speak()
}

fn main() {
    let dog = Dog;
    let cat = Cat;

    static_sound(&dog); // Woof!
    dynamic_sound(&cat); // Meow!
}
```

## Object-Safe Traits necessary for dynamic dispatch

- Traits must be object-safe for `dyn Trait`. Cannot have:
  - Generic methods
  - Methods returning `Self`
  - Static methods without `self`

## Which one to use:

- Static: faster, but duplicate copies of the function, or
- Dynamic: slightly slower, but more flexible.

## Unsafe Rust

### Rationale

- Unsafe Rust allows you to do things that the compiler cannot verify as safe, but this means you take on the responsibility for ensuring your code upholds Rust's memory safety guarantees.
- Here we are manually managing memory that Rust can't track (uninitialized vector elements).

```
use std::ptr;
let mut vec = Vec::with_capacity(3);
unsafe {
    // Write directly to uninitialized memory
    ptr::write(vec.as_mut_ptr(), 42);
    ptr::write(vec.as_mut_ptr().add(1), 84);
    vec.set_len(2); // Fix length after manual init
}
println!("{}", vec); // [42, 84]
```

## Trait objects and dynamic dispatch

### What are Trait Objects?

- Trait objects (`dyn Trait`) enable runtime polymorphism, allowing different types to be used interchangeably through dynamic dispatch.

## The Rust Ecosystem

### Compiler (install from <https://rustup.rs/>)

```
rustc file.rs # compile a standalone file
rustup update # update the compiler/tools
rustup self update # update the rustup tool
```

## Cargo build and package management system

- Project Management

```
cargo new my_project # Create new binary project
cargo new --lib my_lib # Create new library project
cargo init # Initialize cargo project
```

- Building and running

```
cargo build # Build project (debug mode)
cargo build --release # Build release version
cargo run # Build and run binary
cargo run --bin specific_bin # Run specific binary
```

- Dependencies

```
cargo add serde # Add dependency to Cargo.toml
cargo remove serde # Remove dependency
cargo update # Update dependencies to latest
cargo tree # Show dependency tree
```

- Testing and checking

```
cargo test # Run all tests
cargo test test_name # Run specific test
cargo check # Check code without building
cargo clippy # Run linter
cargo fmt # Format code
```

- Documentation and publishing

```
cargo doc # Generate documentation
cargo doc --open # Generate and open docs in browser
cargo publish # Publish crate to crates.io
cargo login # Login to crates.io
```

- Cleanup and information

```
cargo clean # Remove build artifacts
cargo --version # Show Cargo version
cargo search keyword # Search crates.io
```

- Help: most cargo commands accept the `--help` flag.

## Development Tools

- rustup – toolchain installer and version manager.
- rust-analyzer – Powerful LSP server for IDEs
- Clippy – Advanced linter with 500+ lint rules
- rustfmt – Automatic code formatting
- Miri – Interpreter for detecting undefined behaviour

## Getting started

### Working with the borrow checker

- **Don't fight with the borrow checker** – learn from it. Compiler errors are your friend. Read them carefully; they're often very specific and even suggest fixes.
- **Think in terms of ownership.** Ask yourself: Who owns this value? How long should it live? Who is allowed to mutate it?
- **Prefer immutable borrows first.** Use `&T` instead of `&mut T` if possible. Rust allows many simultaneous immutable borrows, but only one mutable borrow.
- **Break problems into smaller scopes.** Long functions with many borrows are harder for the compiler to reason about. Introduce helper functions or inner blocks `{ ... }` to shorten lifetimes.
- **It's okay to clone() early on** (when prototyping) to get code working but treat it as scaffolding. Later, refactor to eliminate any unnecessary copies.

- **Use Rust's patterns.** Iterators, pattern matching, and ownership-based APIs often make borrow checker issues disappear.
- **Understand references vs. values.** Much frustration comes from mixing `T` and `&T`. Practice with `String` vs `&str` until it clicks.
- **Take lifetimes step by step.** Don't fear them; start with compiler-inferred lifetimes, then add annotations only when necessary.
- **Read error messages top-down.** The first line explains what failed, the rest explains why and often suggests how to fix it.

### Some common rust gotchas

- Rust has multiple string types

```
fn greet(s: String) { println!("Hello {}", s); }
let name = "Alice";
greet(name); // ERROR: expected String, found &str
// THE FIX: greet(String::from(name))
// OR: fn greet(s: &str) { println!("Hello {}", s); }
```

- Rules of thumb:

- Fn params: use `&str` unless you need ownership
- Return values: use `String` for new strings
- Struct fields: use `String` for owned data

- Functions cannot return temporary (scope) variables

```
fn get_string() -> &str {
    let s = String::from("hello");
    &s // ERROR: s is dropped at end of function
    // so the reference to s cannot be returned
}

// THE FIX: fn gs() -> String { String::from("hello") }
// OR: fn gs() -> &'static str { "hello" }
```

- For loops can consume your data

```
let numbers = vec![1, 2, 3];
for n in numbers { // 'numbers' is moved here
    println!("{}", n);
}
println!("{}", numbers); // ERROR: numbers moved

// THE FIX: for n in &numbers {} OR: for n in numbers.iter()
```

- Closures reference scope-level data by default

```
let mut count = 0;
let mut increment = || count += 1; // mut borrow
// count += 1; // ERROR: count is borrowed by closure
// so we cannot directly modify it
increment(); // we use the borrowed count here

// THE FIX: use a code block
let mut count = 0;
{
    let mut increment = || count += 1;
    increment();
} // borrow ends here
count += 1; // Now this works
```

- A closure move, moves the data at declaration

```
let s = String::from("hello");
let closure = move || println!("{}", s); // s is moved
// println!("{}", s); // ERROR: s was moved
closure(); // Works
closure(); // Still works! (s was moved, not consumed)

// THE FIX: Clone if you need both
let s = String::from("hello");
let s_clone = s.clone();
let closure = move || println!("{}", s_clone);
println!("{}", s); // Original still available
```