

# MyRs Cheat Sheet 😊

Quick Reference — Draft v1.0 (November 9, 2025)

This template sets up a compact, two-column A4 layout. Listings are configured for small print and good readability.

## I RUST BASICS

### 1 Cargo Quick Commands

- cargo new myproj – Create new binary project
- cargo new -lib mylib – Create new library crate
- cargo new -bin myproj – Create new binary project
- cargo add <crate> – Add dependency
- cargo add -dev <crate> – Add dev-dependency
- cargo check – Check code without building
- cargo clippy – Lint code
- cargo doc -open – Generate and open docs
- cargo run – Build and run
- cargo run -quiet – Quiet run
- cargo build -release – Optimized build
- cargo build -quiet – Quiet build
- cargo clean – Remove build artifacts
- cargo bench – Run benchmarks
- cargo test – Run tests
- cargo fmt – Format code

### 2 Hello, world! (Rust)

Create and compile with bash:

```
# Create file
touch main.rs
```

Rust code:

```
fn main() {
    println!("Hello, world!");
}
```

```
# Compile with rustc
rustc main.rs
```

```
# Run the binary
./main
```

#### 2.1 File Naming Convention

- ✓ hello\_world.rs – Correct: snake\_case
- ✗ helloworld.rs – Wrong: no separator

### 3 Variables and Types

```
fn main() {
    // Scalar types:

    let decimal: i32 = 98_222;      // Integer
    let hex: i32 = 0xff;            // Hexadecimal
    let octal: i32 = 0o77;          // Octal
    let binary: i32 = 0b111_0000;   // Binary
    let byte: u8 = b'A';           // Byte (u8)
    let character: char = 'R';     // Character
    let hugging_face: char = ' ';  // Unicode character

    // Compound types:

    let tuple: (i32, f64, u8) = (500, 6.4, 1);
    let (a, b, c) = tuple;
    let array: [i32; 4] = [1, 2, 3, 4];
    let first = array[0];

    // Constants:
    const MAX_POINTS: u32 = 100_000;
}
```

#### 3.1 Integer Types Overview

- Fixed-size types: u8/i8, u16/i16, u32/i32, u64/i64, u128/i128
- Pointer-sized types:
  - **usize**: Unsigned, size depends on architecture (4 bytes on 32-bit, 8 bytes on 64-bit)
  - **isize**: Signed, same size rules as **usize**
- Use cases:
  - Use **usize/isize** for indexing collections and representing memory sizes
  - Use fixed-size types (**u32/i32, u64/i64**) for precise control over value ranges
  - **u64** is always 8 bytes, similar to **usize** on 64-bit systems

### 3.2 Mutability

```
fn main() {
    let x = 5;      // immutable variable by default
    println!("Initial x: {}", x); // 5
    // x = x + 1; // error: cannot assign twice to immutable variable

    let mut x = 5; // mutable variable
    println!("Initial x: {}", x); // 5
    x = x + 1;    // modify x
    println!("Modified x: {}", x); // 6
}
```

### 3.3 Shadowing

```
fn main() {
    let x = 5;
    let x = x + 1; // shadows previous x
    {
        let x = x * 2; // shadows again in inner scope
        println!("Inner x: {}", x); // 12
    }
    println!("Outer x: {}", x); // 6
}
```

#### 3.3.1 Shadowing vs Mutability

```
fn main() {
    // using mutability
    let mut x = 5;
    x = x + 1; // modify x
    println!("Mutable x: {}", x); // 6
    // using shadowing
    let x = 5;
    let x = x + 1; // shadow previous x
    println!("Shadowed x: {}", x); // 6
}
```

## 4 Macros

### 4.1 Declarative Macros

```
macro_rules! greet {
    ($name:expr) => {
        println!("Hello, {}!", $name);
    };
}

fn main() {
    greet!("Alice"); // Hello, Alice!
    greet!("Bob"); // Hello, Bob!
}
```

### 4.2 Procedural Macros

Procedural macros work on the abstract syntax tree (AST) of Rust code.

Types:

- Derive macros: #[derive(MyTrait)]
- Attribute macros: #[my\_attribute]
- Function-like macros: my\_macro!(input)

Example: Custom Derive Macro

```
// In Cargo.toml:
// [lib]
// proc-macro = true

use proc_macro::TokenStream;
use quote::quote;
use syn::parse_macro_input, DeriveInput;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    let ast = parse_macro_input!(input as DeriveInput);
    let name = &ast.ident;

    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello from {}!", stringify!(#name));
            }
        };
    }
    gen.into()
}
```

Usage:

```
trait HelloMacro {
    fn hello_macro();
}

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro(); // Hello from Pancakes!
}
```

## 5 Control Flow

```
fn main() {
    let number = 6;
    if number % 2 == 0 {
        println!("{} is even", number);
    } else {
        println!("{} is odd", number);
    }
}
```

### 5.1 Combining Conditions

```
fn main() {
    let number = 6;
    if number > 0 && number % 2 == 0 {
        println!("{} is a positive even number", number);
    } else if number > 0 && number % 2 != 0 {
        println!("{} is a positive odd number", number);
    } else {
        println!("{} is not positive", number);
    }
}
```

OR || operator

```
fn main() {
    let number = 6;
    if number > 0 || number % 2 == 0 {
        println!("{} is a positive even number", number);
    } else if number > 0 || number % 2 != 0 {
        println!("{} is a positive odd number", number);
    }
}
```

```

    println!("{} is a positive odd number", number);
} else {
    println!("{} is not positive", number);
}

```

## 5.2 Basic Loop

```

fn main() {
    let mut count = 0;
    loop {
        count += 1;
        if count == 5 {
            break;
        }
        println!("Count: {}", count);
    }
}

```

### 5.2.1 Loop Labels

```

fn main() {
    let mut count = 0;
    'outer: loop {
        count += 1;
        let mut inner_count = 0;
        loop {
            inner_count += 1;
            if inner_count == 3 {
                break 'outer; // breaks the outer loop
            }
            println!("Inner Count: {}", inner_count);
        }
        println!("Count: {}", count);
    }
}

```

## 6 For and While

```

fn main() {
    let numbers = [10, 20, 30, 40, 50];
    for n in numbers.iter() {
        println!("Number: {}", n);
    }
}

fn main() {
    let mut count = 0;
    while count < 5 {
        println!("Count: {}", count);
        count += 1;
    }
}

```

### 6.1 Range in For Loop

```

fn main() {
    for i in 1..=5 { // inclusive range
        println!("i: {}", i);
    }
}

```

### 6.2 Break and Continue

```

fn main() {
    for i in 1..10 {
        if i % 2 == 0 {
            continue; // skip even numbers
        }
        if i > 7 {
            break; // exit loop if i > 7
        }
        println!("Odd i: {}", i);
    }
}

```

## 7 Pattern Matching

```

fn main() {
    let number = 3;
    match number {
        1 => println!("One"),
        2 => println!("Two"),
        3 | 4 | 5 => println!("Three, Four, or Five"),
        _ => println!("Something else"),
    }
}

```

### Or | operator

```

fn main() {
    let x = 2;
    match x {
        1 | 3 | 5 => println!("Odd"),
        2 | 4 => println!("Even"),
        _ => println!("Something else"),
    }
}

```

### Matching Ranges

```

fn main() {
    let x = 5;
    match x {
        1..=5 => println!("In range 1 to 5"),
        _ => println!("Out of range"),
    }
}

```

### Ignoring Values with \_

```

fn main() {
    let point = (3, 5);
    match point {
        (x, _) => println!("x is {}, y is ignored", x),
    }
}

```

## 8 Functions

```

fn greet(name: &str) {
    println!("Hello, {}!", name);
}

fn main() {
    greet("Alice");
    greet("Bob");
}

```

### 8.1 Functions with Return Values

```

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

fn main() {
    let sum = add(5, 10);
    println!("Sum: {}", sum);
}

```

## 9 Statements and Expressions

```

fn main() {
    let x = 5; // statement
    let y = {
        let z = 10; // statement
        z + 5 // expression
    }; // expression

    println!("x: {}, y: {}", x, y);
}

```

### If as an Expression

```

fn main() {
    let condition = true;
    let number = if condition { 5 } else { 10 }; // if expression
    println!("The number is: {}", number);
}

```

### match as an Expression

```

fn main() {
    let number = 3;
    let result = match number {
        1 => "One",
        2 => "Two",
        3 => "Three",
        _ => "Something else",
    }; // match expression
    println!("The result is: {}", result);
}

```

## 10 Ownership and Borrowing

```

fn main() {
    let s1 = String::from("hello"); // s1 owns the string
    let s2 = s1; // ownership moved to s2
    // println!("{}", s1); // error: s1 is no longer valid

    let s3 = String::from("world");
    let s4 = &s3; // borrow s3
    println!("{}: {}, s4: {}", s3, s4); // both valid
} // s3 and s4 go out of scope here

```

### 10.1 Stack and Heap

- Stack:** Fast, fixed-size data. FiFo structure.
- Heap:** Dynamic-size data. Slower access due to indirection.

```

fn main() {
    let x = 5; // stored on stack
    let s1 = String::from("hello"); // stored on heap
    let s2 = s1; // ownership moved to s2
    // println!("{}: {}, s1"); // error: s1 is no longer valid
    println!("{}: {}, s2: {}", x, s2);
} // x and s2 go out of scope here and memory is freed

```

### 10.2 String Type

**String Literals:** immutable, fixed-size, stored on stack, example:

```
let s: &str = "Hello, world!"; // string slice (immutable)
```

**String Type:** mutable, growable, stored on heap, example:

```
let mut s = String::from("Hello"); // String type (mutable)
```

### 10.3 Ownership and Scope

```

fn main() {
    {
        let s = String::from("hello"); // s is valid here
        println!("{}: {}", s);
    } // s goes out of scope and memory is freed here
}

```

### 10.4 Move

#### moving with integers:

```

fn main() {
    let x = 5;
    let y = x; // x is moved to y - Copy trait for integers
    // println!("{}: {}, x"); // error: x is no longer valid
    println!("{}: {}, y");
}

```

#### moving with String:

```

fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // s1 is moved to s2 - ownership transferred
    // println!("{}: {}, s1"); // error: s1 is no longer valid
    println!("{}: {}, s2");
}

```

**Notes:** When you create a String, the actual string struct itself is stored on the stack, but the contents (the characters) are stored on the heap. When you move a String, you are transferring ownership of the heap data to the new variable. **Shallow Copy** means copying only the pointer to the data, not the data itself. In Rust, moving a String is like a shallow copy because the ownership of the heap data is transferred to the new variable, and the original variable can no longer access it.

## 10.5 Copy vs Clone

Copy	Clone
The bits are copied exactly ( <b>shallow copy</b> )	The actual data on the heap is duplicated ( <b>deep copy</b> )
Only types that do not manage heap-allocated memory can implement Copy	Necessary for types that manage resources that need to be separately owned, like String

Example:

```
fn main() {
    // Copy trait (implicit)
    let x = 5;
    let y = x; // x is copied to y
    println!("x: {}, y: {}", x, y); // both valid

    // Clone trait (explicit)
    let s1 = String::from("hello");
    let s2 = s1.clone(); // explicit clone
    println!("s1: {}, s2: {}", s1, s2); // both valid
}
```

## 10.6 Passing Ownership to Functions

```
fn takes_ownership(s: String) {
    println!("{}!", s);
} // s goes out of scope and memory is freed here

fn main() {
    let s = String::from("hello");
    takes_ownership(s); // ownership moved to function
    // println!("{}!", s); // error: s is no longer valid
}
```

## 10.7 Returning Ownership from Functions

```
fn gives_ownership() -> String {
    let s = String::from("hello");
    s // ownership moved to caller
}

fn takes_and_gives_back(s: String) -> String {
    s // ownership moved back to caller
}

fn main() {
    let s = gives_ownership(); // ownership moved to s
    println!("{}!", s);
}
```

## 10.8 References and Borrowing

- Pointer:** A variable that stores the memory address of another variable.
- Reference:** Acts like a pointer to the value it references, but it is guaranteed to always be valid and cannot be null. References are created using the & symbol.
  - Immutable references:** &T allows you to borrow a value without taking ownership and modifying it.
  - Mutable references:** &mut T allows you to borrow a value and modify it. We can have only one mutable reference to a particular piece of data in a particular scope.

## 10.9 Rules of References

At any given time,

- you can have either one mutable reference
- or any number of immutable references.

Single Mutable Reference Allowed:

```
fn main() {
    let mut s = String::from("hello");
    let r1 = &mut s; // first mutable reference
    // let r2 = &mut s; // error: second mutable reference not allowed
    r1.push_str(", world");
    println!("{}!", r1);

    // let r2 = &mut s; // error: second mutable reference not allowed
}
```

## 10.10 The Slice Type

- Allows to reference parts of a collection without copying data.
- Slices are references with bound checking, ensuring memory outside the slice is not accessed.

```
fn main() {
    let s = String::from("hello world");
    let hello = &s[0..5]; // slice for "hello"
    let world = &s[6..11]; // slice for "world"
    println!("{} {}", hello, world);
}
```

### Rules for Slices

- Slice must always point to a valid data.
- One mutable slice or any number of immutable slices allowed at a time.

- Combining mutable and immutable slices is not allowed until all references go out of scope.

## 11 Collections

### 11.1 Structs

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        email: String::from("user1@example.com"),
        username: String::from("user1"),
        sign_in_count: 1,
        active: true,
    };
    println!("Username: {}", user1.username);
}
```

#### 11.1.1 Debug Trait

```
#[derive(Debug)]
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
fn main() {
    let user1 = User {
        email: String::from("user1@example.com"),
        username: String::from("user1"),
        sign_in_count: 1,
        active: true,
    };
    println!("User1: {:?}", user1);
}
```

#### 11.1.2 Methods

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect = Rectangle { width: 10, height: 5 };
    println!("Area: {}", rect.area());
}
```

#### 11.1.3 Mutable Methods

Mutable methods are similar to regular methods, but they take a mutable reference to self (&mut self) instead of an immutable reference (&self). This allows the method to modify the instance it is called on.

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Self {
        Counter { count: 0 }
    }

    fn increment(&mut self) {
        self.count += 1;
    }

    fn get_count(&self) -> u32 {
        self.count
    }
}

fn main() {
    let mut counter = Counter::new();
    counter.increment();
    println!("Count: {}", counter.get_count());
}
```

## 11.2 Enums

An enum is like a list of possible values. Each value in the enum is called a variant.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::Write(String::from("Hello"));
    match msg {
        Message::Quit => println!("Quit message"),
        Message::Move { x, y } => println!("Move to ({}, {})", x, y),
        Message::Write(text) => println!("Write message: {}", text),
        Message::ChangeColor(r, g, b) => println!("Change color to RGB({}, {}, {})", r, g, b),
    }
}
```

#### 11.2.1 Option<T>

- Represents an optional value: every Option is either Some(value) and contains a value, or None, and does not.
- Used to handle cases where a value may be absent without using null.
- Helps avoid runtime errors related to null references.

```
fn main() {
    let some_number: Option<i32> = Some(5);
    let no_number: Option<i32> = None;
}
```

## 11.2.2 Result<T, E>

- Represents either success (`Ok(value)`) or failure (`Err(error)`).
- Used for error handling in Rust.

```
fn divide(dividend: f64, divisor: f64) -> Result<f64, String> {
    if divisor == 0.0 {
        Err(String::from("Division by zero"))
    } else {
        Ok(dividend / divisor)
    }
}

fn main() {
    match divide(10.0, 2.0) {
        Ok(result) => println!("Result: {}", result),
        Err(e) => println!("Error: {}", e),
    }
}
```

## 12 Vectors

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // new method to create an empty vector
    //
    //let mut numbers: Vec<i32> = Vec::new();

    for n in &numbers { println!("{}n"); }

    match numbers.get(10) {
        Some(x) => println!("found: {}x"),
        None => println!("none"),
    }

    // Accessing elements directly
    let first = numbers[0];
    println!("First element: {}, first");
    let second = &numbers[1];
    println!("Second element: {}, second");

    // Accessing elements safely
    if let Some(value) = numbers.get(2) {
        println!("Element at index 2: {}", value);
    } else {
        println!("No element at index 2");
    }

    // Adding elements
    numbers.push(6);

    // Removing elements
    let last = numbers.pop();

    // Inserting elements
    numbers.insert(2, 1); // insert 1 at index 2

    // Removing elements by index
    numbers.remove(3); // remove element at index 3

    // iterating with indices
    for (index, value) in numbers.iter().enumerate() {
        println!("Index: {}, Value: {}", index, value);
    }
}
```

### Notes:

- Vectors are heap-allocated, growable arrays.
- Use `vec![]` macro to create vectors.
- Use `push` to add elements, `pop` to remove the last element.
- Access elements using indexing or `get` method for safe access.
- `iter` method returns an iterator over references to the elements of the vector.
- `iter_mut` method returns an iterator that allows modifying each element.
- `into_iter` method consumes the vector and returns an iterator that yields owned elements
- `enumerate` method returns an iterator that yields pairs of index and reference to the element.

## 12.1 Common Vector Methods

- `len()` - Returns the number of elements in the vector.
- `is_empty()` - Checks if the vector is empty.
- `clear()` - Removes all elements from the vector.
- `remove(index)` - Removes the element at the specified index.
- `contains(&value)` - Checks if the vector contains a specific value.
- `sort()` - Sorts the elements of the vector in ascending order.
- `reverse()` - Reverses the order of elements in the vector.
- `extend(iterable)` - Extends the vector by appending elements from an iterable.

## 13 Hashmap

- HashMap is a collection that stores key-value pairs.
- Keys must be unique and implement the `Eq` and `Hash` traits.
- Values can be of any type.

```
use std::collections::HashMap;
fn main() {
    // Creating a new HashMap
    let mut scores = HashMap::new();

    // Inserting key-value pairs
    scores.insert(String::from("Alice"), 10);
    scores.insert(String::from("Bob"), 20);
}
```

```
// Accessing values
if let Some(score) = scores.get("Alice") {
    println!("Alice's score: {}", score);
}

// Iterating over key-value pairs
for (name, score) in &scores {
    println!("{}: {}", name, score);
}

// Updating values
scores.entry(String::from("Alice")).and_modify(|e| *e += 5).or_insert(10);

// Removing a key-value pair
scores.remove("Bob");

// Create a HashMap from an iterator
let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];
let scores: HashMap<_, _> = teams.into_iter().zip(initial_scores.into_iter()).collect();

println!("{}: {}", ":", scores);
}
```

## 13.1 Common HashMap Methods

- `len()` - Returns the number of elements in the HashMap.
  - `is_empty()` - Checks if the HashMap is empty.
  - `remove(&key)` - Removes the key-value pair for the specified key.
  - `contains_key(&key)` - Checks if the HashMap contains the specified key.
- ```
let mut map = HashMap::new();
map.insert("key1", 10);
map.insert("key2", 20);
println!("Length: {}", map.len()); // Length: 2

let map: HashMap<&str, i32> = HashMap::new();
println!("Is empty: {}", map.is_empty()); // Is empty: true

let mut map = HashMap::new();
map.insert("key1", 10);
map.remove("key1");
println!("Length after removal: {}", map.len()); // Length after removal: 0

let mut map = HashMap::new();
map.insert("key1", 10);
println!("Contains key1: {}", map.contains_key("key1")); // Contains key1
```

## 14 Errors

`panic!` macro is used to indicate unrecoverable errors in Rust. When called, it stops the normal execution of the program and starts unwinding the stack, cleaning up resources along the way.

```
fn divide(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("Division by zero is not allowed");
    }
    a / b
}
```

## 14.1 Creating Custom Error Types

```
use std::fmt;

#[derive(Debug)]
enum MyError {
    NotFound,
    InvalidInput,
    ConnectionError,
}

impl fmt::Display for MyError {
    fn fmt(&self, f: &mut fmt::Fmatter) -> fmt::Result {
        match self {
            MyError::NotFound => write!(f, "Resource not found"),
            MyError::InvalidInput => write!(f, "Invalid input provided"),
            MyError::ConnectionError => write!(f, "Connection error occurred"),
        }
    }
}
```

## 14.2 Logging

The ‘log‘ crate provides a lightweight logging facade. To use it:

```
use log::info, warn, error, debug;

fn main() {
    env_logger::init();
    info!("Starting application");
    warn!("Low disk space");
    error!("Failed to connect to database");
    debug!("Debugging information");
}
```

The `slog` crate is another popular logging library that provides more features and flexibility.

```
use slog::{Drain, Logger, o, info};
use slog_async;
use slog_term;

fn main() {
    let decorator = slog_term::TermDecorator::new().build();
    let drain = slog_term::CompactFormat::new(decorator).build().fuse();
    let drain = slog_async::Async::new(drain).build().fuse();
    let log = Logger::root(drain, o!());

    info!(log, "Application started");
}
```

## 15 Crates, Packages and Modules

- **Crate:** A binary or library. The smallest unit of code distribution in Rust.
  - **Binary Crate:** Produces an executable. Contains a `main` function. `cargo new -bin my_app`
  - **Library Crate:** Provides functionality to be used by other crates. Does not have a `main` function. `cargo new -lib my_lib`
- **Package:** A bundle of one or more crates. Managed by Cargo.
- **Module:** A way to organize code within a crate. Can contain functions, structs, enums, etc.

## II ADVANCED CONCEPTS

### 16 Generics

#### 16.1 Generic functions

```
fn first_element<T>(list: &[T]) -> Option<&T> {
    if list.is_empty() {
        None
    } else {
        Some(&list[0])
    }
}

fn main() {
    let numbers = vec![1, 2, 3];
    let words = vec!["hello", "world"];

    if let Some(first_num) = first_element(&numbers) {
        println!("First number: {}", first_num);
    }

    if let Some(first_word) = first_element(&words) {
        println!("First word: {}", first_word);
    }
}
```

#### 16.2 Generic structs

```
struct Point<T> {
    x: T,
    y: T,
    z: i32,
}

impl<T> Point<T> {
    fn new(x: T, y: T, z: i32) -> Self {
        Point { x, y, z }
    }
}

fn main() {
    let int_point = Point::new(1, 2, 3);
    let float_point = Point::new(1.0, 2.0, 3.0);

    println!("Integer Point: ({}, {}, {})", int_point.x, int_point.y, int_point.z);
    println!("Float Point: ({}, {}, {})", float_point.x, float_point.y, float_point.z);
}
```

#### 16.3 Generic Enums

```
enum Option<T> {
    Some(T),
    None,
}

fn main() {
    let some_number = Option::Some(5);
    let no_number: Option<i32> = Option::None;

    match some_number {
        Option::Some(value) => println!("Got a number: {}", value),
        Option::None => println!("No number"),
    }

    match no_number {
        Option::Some(value) => println!("Got a number: {}", value),
        Option::None => println!("No number"),
    }
}
```

#### 16.4 Notes:

- Using generics doesn't slow down your code. The compiler generates optimized versions for each type used.
- Monomorphization is the process of generating specific implementations for each type used with generics.
- Monomorphization happens at compile time, so there is no runtime overhead.

## 17 Traits

- Traits define shared behavior across types.
- A trait is like a contract that types can implement.
- Traits enable polymorphism and code reuse.

#### 17.1 Defining and Implementing Traits

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}

struct NewsArticle {
    headline: String,
    location: String,
    content: String,
}

impl Summary for NewsArticle {
```

```
fn summarize(&self) -> String {
    format!("{} - {}", self.headline, self.location)
}
}

struct Tweet {
    username: String,
    content: String,
    reply: bool,
    retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}

fn main() {
    let article = NewsArticle {
        headline: String::from("Rust is awesome!"),
        location: String::from("Internet"),
        content: String::from("Rust is a systems programming language..."),
    };
    let tweet = Tweet {
        username: String::from("user123"),
        content: String::from("Hello, world!"),
        reply: false,
        retweet: false,
    };
    println!("Article Summary: {}", article.summarize());
    println!("Tweet Summary: {}", tweet.summarize());
}
```

#### 17.1.1 Polymorphism

A function that accepts any type implementing a trait. Using traits as function parameters:

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

fn main() {
    let article = NewsArticle {
        headline: String::from("Rust is awesome!"),
        location: String::from("Internet"),
        content: String::from("Rust is a systems programming language..."),
    };
    let tweet = Tweet {
        username: String::from("user123"),
        content: String::from("Hello, world!"),
        reply: false,
        retweet: false,
    };
    notify(&article);
    notify(&tweet);
}
```

#### 17.1.2 Trait Bounds and Generics

Using trait bounds in generic functions:

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

## 18 Lifetimes

Lifetimes manage how long references are valid to prevent dangling references.

- Ensure references do not outlive the data they point to.
- Specified using apostrophes (e.g., `'a`).
- Prevents references from pointing to invalid data.

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

fn main() {
    let string1 = String::from("long string");
    let string2 = "short";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

#### 18.1 Lifetime Annotations in Structs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let excerpt = ImportantExcerpt { part: first_sentence };
    println!("Excerpt: {}", excerpt.part);
}
```

#### 18.1.1 Lifetime Elision

Rust applies three rules to infer lifetimes when they are not explicitly annotated:

- Each parameter that is a reference gets its own lifetime parameter.
- If there is exactly one input lifetime parameter, that lifetime is assigned to all output reference parameters.
- If there are multiple input lifetime parameters, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all output reference parameters.

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
```

```

        return &s[0..i];
    }
}

&s[..]
}

```

### 18.1.2 Static Lifetimes

Static lifetimes are the longest possible lifetimes in Rust. They last for the entire duration of the program. They are stored in the binary's read-only memory.

```

fn main() {
    // A string literal has a 'static lifetime
    let s: &static str = "I have a static lifetime.";

    // Global variable with 'static lifetime
    static GLOBAL_VAR: &str = "I am a global variable with a static lifetime.";
}

}

```

## 19 Smart Pointers

Smart pointers are data structures that not only act like a pointer but also have additional metadata and capabilities.

Single ownership with Box<T>:

```

fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}

```

Shared ownership with Rc<T>:

```

use std::rc::Rc;

fn main() {
    let a = Rc::new(5);
    let b = Rc::clone(&a);
    println!("a = {}, b = {}", a, b);
}

```