**Rust Cheat Sheet**

**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.

// The primitive scalar types

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 five\_hundred = tup.0; // index 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 Empty()

struct UserID(i32)

struct Point(i32, i32); // 2D point

let point = Point(10, 20);

let x = point.0; // Access first field

let y = point.1; // Access second field

**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 // copy member\_1 for remaining fields

};

let name = member\_1.name;

* **Enum**: one of multiple alternative data items/values. Unlike enums in other languages, Rust enums can hold data values. Enums can have code impl blocks just like structs (discussed later).

// 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.
* 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; } // m 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();

let set1 = HashSet::from([1, 2, 3]);

let set2 = HashSet::from([3, 4, 5]);

let union: HashSet<\_> =

set1.union(&set2).collect();

let intersection: HashSet<\_> =

set1.intersection(&set2).collect();

let difference: HashSet<\_> =

set1.difference(&set2).collect();

for item in &set { println!("{}", item); }

for (i, e) in set.iter().enumerate()

{ println!("{}: {}", i, e); }

for item in set { println!("{}", item); }

**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.
* Move types (typically items stored in heap memory): String, Vec, HashMap, custom structs, etc, as well as mutable references are always moved. The original variable becomes invalid to prevent double-free errors and ensure memory safety.

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 (implement .copy()): 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!("{} and {}", 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.
* **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).

**Lifetimes**

* Lifetimes ensure references don't outlive the data they point to.

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {

if x.len() > y.len() { x } else { y }

}

* The 'a lifetime parameter tells Rust that the returned reference will live as long as both input references.
* Lifetime parameters are rarely needed in a function definition, as the compiler can usually work it out.
* Note: the 'static lifetime is the longest lasting.

**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);

// first references &mut [1], rest refs &mut [2, 3]

**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 = 10.0;

let y = 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 = 5;

let b = 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

\*mr = 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 ().

let y = {

let a = 2;

let b = 3;

a + b // no semicolon, value of block is returned

};

* Code blocks are important in Rust because:
  + They control scope – contain temporary variables.
  + They return values – like inline mini-functions.
  + Resource management – when a block ends, local variables are dropped automatically.

**Functions**

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

fn greet(name: &str) {

println!("Hello, {}!", name);

}

fn main() {

greet("Alice"); // calling the function

}

* In idiomatic Rust, functions that return a value, are written as a code block that ends with an expression.

fn add(x: i32, y: i32) -> i32 {

x + y // No semicolon! This expression is returned

}

* 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"

}

**Macros**

* Rust has built-in macros (that look like function calls, but with a name ending with a bang! – see println! above). Macros are expanded into Rust code at compile time. Note: you can also write your own macros.
* Commonly used built-in macros include:

// String and Vector creation

let numbers = vec![1, 2, 3]; // create a Vector

let s = format!("{} + {} = {}", 2, 3, 5); // Strings

// Output

println!("Hello, {}!", "world");

print!("Hello, {}!", "world"); // without a newline

dbg!(2 + 2); // provide a debug print with line number

// Stop running if something is unexpected

assert\_eq!(2 + 2, 4);

assert!(true);

panic!("Something went wrong!");

**Error handling**

* There are no exceptions in Rust. If a function can fail, it typically returns either the Option or the Result enum. This requires you to handle the error.
  + The Result enum represents success or failure. It has two variants: Ok and Err.
  + The Option enum either holds a value or none. It has two variants: Some and None.
* **.unwrap() or .expect()** methods: These cause your program to panic if an error is detected. While unwrap and expect are useful tools when prototyping, they are best not used in production.
* The expect method provides a message to the user.

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 value in the case of an error.

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**: if the Result is Ok, or the Option is Some, the ? operator unwraps it. If it is not good, it returns early from the function with that error. Note: io::Result is a specialised version of the generic Result type.

use std::fs;

fn main() -> std::io::Result<()> {

let contents = fs::read\_to\_string("hello.txt")?;

println!("{}", contents);

Ok(())

}

* **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**

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

\_ => println!("something else"), // default

}

* Match is an expression

let result = match n {

x if x % 2 == 0 => "even", // conditional match

\_ => "odd",

};

println!("{} is {}.", 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!("{} 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 capture variables from their surrounding environment. They are often used in chained method calls.

let add = |x, y| x + y;

println!("{}", add(5, 3)); // 8

* Or …

let (x, y) = (5, 8);

println!("{}", (|a, b| a + b)(&x, &y)); // borrows

println!("{}", (|a, b| a + b)(x, y)); // consumes

* Many methods take closures as an argument

let numbers = vec![1, 2, 3, 4, 5, 6];

let even: Vec<i32> = numbers

.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

// Collectors

iter.collect::<Vec<\_>>() // Collect into Vec

iter.collect::<HashSet<\_>>() // Collect into HashSet

iter.collect::<Result<Vec<\_>, \_>>() // Collect Results

// Single value consumers

iter.sum::<i32>() // Sum all elements

iter.product::<i32>() // Multiply all elements

iter.count() // Count elements

iter.last() // Get last element

iter.nth(5) // Get element at index 5

iter.find(|x| x > &5) // Find first matching

iter.position(|x| x > 5) // Index of first match

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

iter.filter\_map(|x| x.checked\_div(2)) // Filter + map

iter.flat\_map(|x| x.children) // Flatten nested iters

iter.flatten() // Flatten one level

// 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)

})

// Inspection (doesn't consume)

iter.inspect(|x| println!("{}", x)) // Debug print

// Reversing (if double-ended)

iter.rev()

// Uniqueness

iter.unique() // Requires itertools crate

* 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);

// Custom iterator loop with mutable state

let mut iter = vec![1, 2, 3].into\_iter();

while let Some(x) = iter.next() {

println!("{}", x);

}

**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 }

}

// Method (called with .)

fn area(&self) -> f64 {

self.width \* self.height

}

// Method that consumes self

fn destroy(self) {

println!("Rectangle destroyed!");

// self is moved and dropped

}

}

* You can have multiple implementation blocks for the same stuct or enum.

impl Rectangle {

fn perimeter(&self) -> f64 {

2.0 \* (self.width + self.height)

}

}

**Traits**

* Traits are guarantees for shared behaviour across types (in particular but not limited to structs and enums). They are similar to interfaces in Java, Go and C# or protocols in Swift.

trait Speak {

// required methods to be implemented

fn speak(&self) -> String;

// or we could provide a default method:

// fn speak(&self) -> String {

// "(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!

}

**Derive macros**

* Derive macros automatically implement common traits for your types, reducing boilerplate code.

#[derive(Debug, Clone, Copy, PartialEq, Eq)]

struct Point {

x: i64,

y: i64,

}

// Debug - enables formatting with {:?} and {:#?}

// Clone - enables explicit duplication with .clone()

// Copy - enables implicit copying (stack-only types)

// PartialEq - enables == and != operators

// Eq – requires PartialEq – rigorous/total equality

* Copy allows implicit copying. Clone allows explicit duplication via the .clone() method.

**Generics**

* Generics let you write a function, struct, enum or trait once, and use it with different types, while the compiler ensures everything is type safe.

// 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

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)

}

* Generics can have more than one type in the signature.

**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)).
* **The work arounds**:
  + Thin wrapper types (you own the type)
  + More common: Extension traits (you own the trait)
* Thin wrapper types - give you full control to write your own implementation blocks, but you lose automatic type coercion. Also a bit klunky

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

**Modules, Libraries and Crates**

**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 = "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::<f64>());

println!("Color: {}",

["R", "B", "G"].choose(&mut rng).unwrap());

}

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

Still to do …

1. Testing basics - #[test], assert! macros
2. Concurrency - threads, Arc, Mutex
3. Async/Await -
4. Smart pointers – Box, Rc, Arc, RefCell, Mutex, etc.