**Rust Programming Workshop**

**Duration**: 2 Days, 7 Hours per Day (Total 14 Hours)  
**Prerequisites**: Familiarity with C/C++ or Python, Access to a Computer with Rust Installed

**Overview**

This 2-day workshop introduces BTech students to Rust, a systems programming language known for safety, speed, and concurrency. The course covers fundamentals to practical applications, with hands-on exercises, live coding, and a mini-project. Each session includes examples, checklists, and references to ensure comprehensive learning.

**Schedule**: 9:00 AM - 5:00 PM daily, with a 1-hour lunch break (12:30 PM - 1:30 PM) and two 15-minute breaks (10:45 AM and 3:15 PM).

**Day 1: Fundamentals of Rust**

**Topic 1: Introduction to Rust (1 Hour)**

**Objectives**: Understand Rust's purpose, history, and advantages.

**Content**:*Cre*

* Rust: A systems language for safety, speed, and concurrency.
* Issues with C programs, all samples
* Features: Memory safety without garbage collection, zero-cost abstractions.
* Comparison: Rust vs. C++ (ownership prevents null pointers, data races).

**Example**:

fn main() {

println!("Hello, Rust! This is a safe and fast language.");

}

**Activity**: Run `cargo new hello\_rust && cd hello\_rust && cargo run`.

**Checklist**:

* ☑ Explain Rust's ownership model.
* ☑ Verify Rust installation (`rustc --version`).
* ☑ Discuss: Why Rust for BTech projects?

**Topic 2: Installation, Setup, and Basic Syntax (1.5 Hours)**

**Objectives**: Set up Rust environment and learn core syntax.

**Content**:

* Install Rust via [rustup.rs](https://rustup.rs/).
* Cargo: Build tool (`cargo new`, `cargo build`, `cargo run`).
* Syntax: Variables (`let`, `mut`), data types (i32, f64, bool), functions.

**Example**:

fn main() {

let x: i32 = 5; // Immutable

let mut y: i32 = 10; // Mutable

y += 1;

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

greet("BTech Students");

}

fn greet(name: &str) {

println!("Welcome to Rust, {}!", name);

}

**Activity**: Create and run a Cargo project.

**Checklist**:

* ☑ Run a basic Cargo project.
* ☑ Understand immutability (try mutating immutable var).
* ☑ Modify example to print student names.

**Topic 3: Control Flow and Loops (1.5 Hours)**

**Objectives**: Master conditionals, loops, and basic pattern matching.

**Content**:

* If-else, match expressions.
* Loops: `loop`, `while`, `for` (iterators).
* Pitfalls: Avoid infinite loops with `break`.

**Example**:

fn main() {

let number = 7;

if number % 4 == 0 {

println!("Divisible by 4");

} else if number % 3 == 0 {

println!("Divisible by 3");

} else {

println!("No divisor");

}

match number {

1 => println!("one"),

2 | 3 => println!("two or three"),

4..=10 => println!("four to ten"),

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

}

for i in 1..=5 {

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

}

}

**Activity**: Implement FizzBuzz using `match`.

**Checklist**:

* ☑ Write a grading program with `match`.
* ☑ Debug unmatched arms in `match`.
* ☑ Complete FizzBuzz exercise.

**Topic 4: Ownership, Borrowing, and Lifetimes (2 Hours)**

**Objectives**: Understand Rust’s memory management.

**Content**:

* Ownership: One owner per value, move vs. copy.
* Borrowing: Immutable (`&`) and mutable (`&mut`) references.
* Lifetimes: Basic `'a` notation.

**Example**:

fn main() {

let s1 = String::from("hello");

let s2 = s1; // Move

// println!("{}", s1); // Error!

let len = calculate\_length(&s2);

println!("Length of '{}' is {}", s2, len);

let mut s3 = String::from("mutable");

change(&mut s3);

println!("Changed: {}", s3);

}

fn calculate\_length(s: &String) -> usize {

s.len()

}

fn change(s: &mut String) {

s.push\_str(", world!");

}

**Activity**: Write a function using borrowing for vector lengths.

**Checklist**:

* ☑ Understand move semantics (test move errors).
* ☑ Practice borrowing in a function.
* ☑ Explain Rust’s safety vs. C++.

**Topic 5: Structs, Enums, and Methods (1 Hour)**

**Objectives**: Define custom types and behaviors.

**Content**:

* Structs: Custom data types.
* Enums: Variants (e.g., `Option`, `Result`).
* Methods: `impl` blocks.

**Example**:

#[derive(Debug)]

struct Student {

name: String,

id: u32,

grade: f32,

}

impl Student {

fn new(name: String, id: u32, grade: f32) -> Student {

Student { name, id, grade }

}

fn display(&self) {

println!("{} (ID: {}) - Grade: {}", self.name, self.id, self.grade);

}

}

enum Grade {

A, B, C, Fail,

}

fn main() {

let student = Student::new("Alice".to\_string(), 123, 8.5);

student.display();

let g = Grade::A;

match g {

Grade::A => println!("Excellent!"),

\_ => println!("Keep trying."),

}

}

**Activity**: Define a `Course` struct for a BTech project.

**Checklist**:

* ☑ Define a struct and methods.
* ☑ Use enum for error types.
* ☑ Test `impl` methods.

**Day 2: Advanced Basics and Practical Application**

**Topic 6: Error Handling (1 Hour)**

**Objectives**: Handle errors without panics.

**Content**:

* Panic vs. recoverable errors.
* `Result` and `Option`.
* `?` operator for error propagation.

**Example**:

use std::fs::File;

fn main() -> Result<(), Box> {

let f = File::open("hello.txt")?;

Ok(())

}

fn divide(a: f64, b: f64) -> Result {

if b == 0.0 {

Err("Division by zero!".to\_string())

} else {

Ok(a / b)

}

}

**Activity**: Write a function with `Result` for file I/O.

**Checklist**:

* ☑ Write a `Result`-returning function.
* ☑ Handle errors with `match` or `?`.
* ☑ Discuss panic vs. `Result`.

**Topic 7: Modules, Crates, and Collections (1.5 Hours)**

**Objectives**: Organize code and use data structures.

**Content**:

* Modules: `mod`, `pub` for visibility.
* Crates: Add dependencies in `Cargo.toml`.
* Collections: `Vec`, `HashMap`, strings.

**Example**:

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

}

use std::collections::HashMap;

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice".to\_string(), 95);

scores.insert("Bob".to\_string(), 88);

for (name, score) in &scores {

println!("{}: {}", name, score);

}

}

**Activity**: Create a multi-module project with `rand` crate.

**Checklist**:

* ☑ Create a module structure.
* ☑ Add `rand` crate and use `Vec`.
* ☑ Sort a `Vec` of structs.

**Topic 8: Concurrency Basics (1.5 Hours)**

**Objectives**: Introduce safe multithreading.

**Content**:

* Threads: `std::thread::spawn`.
* Channels: `mpsc` for communication.
* `Mutex` and `Arc` for shared data.

**Example**:

use std::thread;

use std::sync::mpsc;

use std::time::Duration;

fn main() {

let (tx, rx) = mpsc::channel();

let handle = thread::spawn(move || {

let val = String::from("Hi from thread!");

tx.send(val).unwrap();

});

let received = rx.recv().unwrap();

println!("Got: {}", received);

handle.join().unwrap();

}

**Activity**: Run a multi-threaded sum computation.

**Checklist**:

* ☑ Run a multi-threaded program.
* ☑ Explain data race prevention.
* ☑ Debug unsafe sharing errors.

**Topic 9: Mini-Project and Best Practices (2 Hours)**

**Objectives**: Build a CLI todo list app and learn best practices.

**Content**:

* Project: CLI todo app with structs, vectors, error handling.
* Best Practices: Use `rustfmt`, `cargo test`, documentation.
* Pitfalls: Lifetimes, borrowing rules.

**Example**:

#[derive(Debug)]

struct Todo {

task: String,

done: bool,

}

impl Todo {

fn new(task: String) -> Todo {

Todo { task, done: false }

}

fn complete(&mut self) {

self.done = true;

}

}

fn main() {

let mut todos: Vec = Vec::new();

todos.push(Todo::new("Learn Rust".to\_string()));

todos[0].complete();

println!("{:?}", todos);

}

**Activity**: Extend todo app with add/delete functions.

**Checklist**:

* ☑ Complete and run the project.
* ☑ Write a unit test.
* ☑ Conduct code review.

**Topic 10: Review, Q&A, and Assessment (1 Hour)**

**Objectives**: Reinforce learning and evaluate.

**Content**:

* Recap all topics.
* Open Q&A session.
* Assign future reading.

**Activity**: Collect feedback and rate understanding (1-10).

**Checklist**:

* ☑ Confirm topic coverage.
* ☑ Distribute resources.
* ☑ Collect feedback.

**Important Questions**

1. What are Rust’s three pillars? Explain ownership in 2-3 sentences.
2. How do you declare a mutable variable? Compare `String` vs. `&str`.
3. Write a `match` expression for HTTP status codes (200, 404, 500).
4. What happens when a value is moved? Show with an example.
5. Define an enum for grades (A, B, C) and use in a function.
6. When should you use the `?` operator? Example with `File::open`.
7. How do you import a module? Compare `Vec` vs. array.
8. What is a channel in Rust? Why use `Arc<mutex>`?</mutex
9. How does Cargo manage dependencies? Show `Cargo.toml` example.
10. Write a unit test for a function that checks string length.

**Note**: Expand to 20 questions for homework assignments.

**MCQ Assessment**

Test your Rust knowledge with this online quiz:

[Rust MCQ Quiz](https://www.javaguides.net/2024/05/rust-quiz-mcq-questions-and-answers.html)

Scan to access the quiz.

**Instructions**: Generate a QR code for the quiz link using [qr-code-generator.com](https://www.qr-code-generator.com/) and replace the placeholder image.

**References and Resources**

* [The Rust Programming Language (Official Book)](https://www.rust-lang.org/learn) - Free, comprehensive guide.
* [Rustlings](https://github.com/rust-lang/rustlings) - Interactive exercises.
* [Rust by Example](https://doc.rust-lang.org/rust-by-example/) - Code-focused examples.
* [Comprehensive Rust (Google)](https://google.github.io/comprehensive-rust/) - Beginner tutorial.
* [Cargo Documentation](https://doc.rust-lang.org/cargo/) - Package management.
* [Rust Quiz](https://github.com/dtolnay/rust-quiz) - Advanced questions.
* [CodeChef Rust MCQs](https://www.codechef.com/learn/course/rust/LBRS11/problems/RSB057) - Practice problems.

**Community**: Join r/rust on Reddit or Rust Discord for support.

**Additional Notes**

* Ensure all students install [VS Code](https://code.visualstudio.com/) with the rust-analyzer extension for a better coding experience.
* Encourage exploring crates like serde for JSON handling in future projects.
* Provide certificates upon completion to motivate students.

**📝 Rust Iterator Closure Cheat Sheet**

**1. Basic Iteration**

let v = vec![1, 2, 3];

for x in v.iter() {

println!("{}", x);

}

* v.iter() → iterator of &i32
* x: &i32

**2. Map Example**

let v = vec![1, 2, 3];

let doubled: Vec<i32> = v.iter().map(|x| x \* 2).collect();

* Before map: x: &i32
* After map: produces **i32** values

**3. Filter Example**

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

let result: Vec<i32> = nums

.iter() // Iterator of &i32

.map(|x| x \* 2) // Produces i32

.filter(|x| x > &5) // x: &i32 here

.cloned() // Convert &i32 back to i32

.collect();

**Why &5?**

* After map, values are i32
* But filter hands them in as &i32
* So comparison must be with &5

**4. Alternative Filter Patterns**

.filter(|x| \*x > 5) // Explicit dereference

.filter(|&x| x > 5) // Pattern match to take the value

.filter(|x| x > &5) // Compare with &5

✅ All are equivalent.

**5. Fold Example**

let sum: i32 = nums.iter().fold(0, |acc, x| acc + x);

* x: &i32
* acc: i32 (starts from 0)
* Returns i32

**Quick Type Flow Recap**

* .iter() → &T
* .map() → returns **T**, but next closure sees references
* .filter() → closure arg is always &T
* .fold(init, closure) → closure takes (acc, &T)

# 🌐 Rust Web Programming Cheat Sheet

## 1. Setup

cargo new web\_demo

cd web\_demo

cargo add actix-web tokio

## 2. Hello Web Server (Actix-Web)

use actix\_web::{get, App, HttpResponse, HttpServer, Responder};

#[get("/")]

async fn index() -> impl Responder {

HttpResponse::Ok().body("Hello, Rust Web!")

}

#[tokio::main]

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

HttpServer::new(|| App::new().service(index))

.bind(("127.0.0.1", 8080))?

.run()

.await

}

👉 Visit: http://127.0.0.1:8080

## 3. Basic Routing

use actix\_web::{get, web, App, HttpServer, Responder};

#[get("/user/{name}")]

async fn greet(path: web::Path<String>) -> impl Responder {

format!("Hello, {}!", path.into\_inner())

}

* /{name} → Path parameter
* web::Json<T> → For JSON input/output

## 4. JSON API Example

use actix\_web::{post, web, App, HttpServer, Responder};

use serde::{Deserialize, Serialize};

#[derive(Deserialize)]

struct Input { name: String }

#[derive(Serialize)]

struct Output { message: String }

#[post("/hello")]

async fn hello(data: web::Json<Input>) -> impl Responder {

web::Json(Output {

message: format!("Hello, {}!", data.name),

})

}

* serde required:
* cargo add serde serde\_json

## 5. Static Files

use actix\_files as fs;

App::new()

.service(fs::Files::new("/static", "./static").show\_files\_listing())

👉 Serve HTML/CSS/JS files from ./static

## 6. Middleware Example

use actix\_web::{middleware, App};

App::new()

.wrap(middleware::Logger::default()) // Logs requests

## 7. Database (Diesel + SQLx)

* **Diesel ORM**
* cargo add diesel
* #[derive(Queryable)]
* struct User { id: i32, name: String }
* **SQLx async**
* cargo add sqlx

## 8. Async with Tokio

use tokio::time::{sleep, Duration};

#[tokio::main]

async fn main() {

println!("Start");

sleep(Duration::from\_secs(2)).await;

println!("End after 2s");

}

# 🦀 Rust Error Handling

Rust does **not** use exceptions. Instead, it has **types** that enforce you to handle errors at compile-time.

## 1. Result<T, E>

Used when an operation can succeed (Ok) or fail (Err).

fn divide(a: i32, b: i32) -> Result<i32, String> {

if b == 0 {

Err("Division by zero".to\_string())

} else {

Ok(a / b)

}

}

fn main() {

match divide(10, 2) {

Ok(val) => println!("Result: {}", val),

Err(e) => println!("Error: {}", e),

}

}

✔️ Output: Result: 5

## 2. Option<T>

Used when a value might be **present (Some)** or **absent (None)**.

fn find\_item(items: Vec<&str>, name: &str) -> Option<usize> {

for (i, item) in items.iter().enumerate() {

if \*item == name {

return Some(i);

}

}

None

}

fn main() {

let fruits = vec!["apple", "banana", "mango"];

match find\_item(fruits, "banana") {

Some(i) => println!("Found at index {}", i),

None => println!("Not found"),

}

}

## 3. unwrap()

A shortcut → **crash the program if error happens**.

let text = "42";

let num: i32 = text.parse().unwrap();

println!("{}", num); // works

But if text = "hello", it panics. 🚨

## 4. expect(msg)

Like unwrap(), but lets you print a **custom error message**.

let text = "hello";

let num: i32 = text.parse().expect("Not a valid number");

✔️ Panics with: thread 'main' panicked at 'Not a valid number'

## 5. ? Operator

Propagates error upward instead of handling it immediately.

use std::fs::File;

use std::io::{self, Read};

fn read\_file(path: &str) -> Result<String, io::Error> {

let mut f = File::open(path)?; // if error, return Err

let mut contents = String::new();

f.read\_to\_string(&mut contents)?; // propagate if error

Ok(contents)

}

fn main() -> Result<(), io::Error> {

let data = read\_file("hello.txt")?;

println!("File content: {}", data);

Ok(())

}

## ✅ Summary

* Result<T, E> → success/failure.
* Option<T> → value/no value.
* unwrap() → quick + dirty, panics on error.
* expect("msg") → unwrap + custom message.
* ? → bubble error up cleanly.

# 🦀 Rust Collections Overview

Collections store and organize data in memory. The most common are:

## 1. **Vector (**Vec<T>**)**

* A **growable array**, stored on the heap.
* Fast indexing, but inserting/removing in the middle is slower.

fn main() {

let mut v = Vec::new();

v.push(10);

v.push(20);

println!("{:?}", v); // [10, 20]

// Access

println!("{}", v[0]); // panic if out of bounds

println!("{:?}", v.get(2)); // Option type

}

## 2. **String**

* A **growable UTF-8 text** collection.
* Built on top of Vec<u8>.

fn main() {

let mut s = String::from("Rust");

s.push('!');

s.push\_str(" is awesome");

println!("{}", s);

}

## 3. **HashMap<K, V>**

* Key–value pairs, fast lookup.
* Keys must implement Eq + Hash.

use std::collections::HashMap;

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice", 10);

scores.insert("Bob", 20);

println!("{:?}", scores.get("Alice")); // Some(10)

for (k, v) in &scores {

println!("{} -> {}", k, v);

}

}

## 4. **HashSet<T>**

* Stores unique values (like mathematical set).
* Based on HashMap.

use std::collections::HashSet;

fn main() {

let mut set = HashSet::new();

set.insert(1);

set.insert(2);

set.insert(2); // ignored

println!("{:?}", set); // {1, 2}

}

## 5. **VecDeque<T>**

* A **double-ended queue**.
* Efficient push\_front, pop\_front, push\_back, pop\_back.

use std::collections::VecDeque;

fn main() {

let mut dq = VecDeque::new();

dq.push\_back(1);

dq.push\_front(2);

println!("{:?}", dq); // [2, 1]

}

## 6. **LinkedList<T>**

* Doubly linked list.
* Rarely needed (slower cache performance vs Vec).

use std::collections::LinkedList;

fn main() {

let mut list = LinkedList::new();

list.push\_back(1);

list.push\_front(2);

println!("{:?}", list); // [2, 1]

}

## 7. **BinaryHeap<T>**

* Priority queue (max-heap by default).
* Always pops the largest element.

use std::collections::BinaryHeap;

fn main() {

let mut heap = BinaryHeap::new();

heap.push(4);

heap.push(10);

heap.push(2);

println!("{:?}", heap.pop()); // Some(10)

}

## 8. **BTreeMap<K,V>** / **BTreeSet<T>**

* Sorted key/value map (or set).
* Useful when you need ordered traversal.

use std::collections::BTreeMap;

fn main() {

let mut map = BTreeMap::new();

map.insert(2, "b");

map.insert(1, "a");

map.insert(3, "c");

for (k, v) in &map {

println!("{}: {}", k, v); // keys sorted

}

}

# ⚡ Key Notes

* Vec<T> → default, general-purpose.
* HashMap / HashSet → fast lookup, unordered.
* BTreeMap / BTreeSet → sorted keys.
* VecDeque → efficient queue.
* BinaryHeap → priority queue.
* LinkedList → rare, but useful if you often remove/insert in the middle.

✅ Collections + Iterators = super powerful:

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

let squares: Vec<\_> = nums.iter().map(|x| x \* x).collect();

println!("{:?}", squares); // [1, 4, 9, 16, 25]

# 🦀 Rust Collections – More Examples

## 1. Update Values in a HashMap

use std::collections::HashMap;

fn main() {

let mut scores = HashMap::new();

scores.insert("Alice", 10);

// Increase Alice’s score by 5

\*scores.entry("Alice").or\_insert(0) += 5;

// Insert Bob if not exists

scores.entry("Bob").or\_insert(20);

println!("{:?}", scores); // {"Alice": 15, "Bob": 20}

}

## 2. Grouping with HashMap

use std::collections::HashMap;

fn main() {

let words = vec!["apple", "banana", "apple", "orange", "banana"];

let mut freq = HashMap::new();

for w in words {

\*freq.entry(w).or\_insert(0) += 1;

}

println!("{:?}", freq); // {"apple": 2, "banana": 2, "orange": 1}

}

## 3. Using HashSet for Intersection & Union

use std::collections::HashSet;

fn main() {

let a: HashSet<\_> = [1, 2, 3].into\_iter().collect();

let b: HashSet<\_> = [3, 4, 5].into\_iter().collect();

println!("Union: {:?}", a.union(&b)); // {1, 2, 3, 4, 5}

println!("Intersection: {:?}", a.intersection(&b)); // {3}

println!("Difference: {:?}", a.difference(&b)); // {1, 2}

}

## 4. Queue with VecDeque

use std::collections::VecDeque;

fn main() {

let mut queue = VecDeque::new();

queue.push\_back("Task1");

queue.push\_back("Task2");

queue.push\_front("Urgent");

println!("{:?}", queue); // ["Urgent", "Task1", "Task2"]

let first = queue.pop\_front();

println!("Processing: {:?}", first); // Urgent

}

## 5. Priority Queue with BinaryHeap

use std::collections::BinaryHeap;

fn main() {

let mut heap = BinaryHeap::new();

heap.push(5);

heap.push(1);

heap.push(10);

while let Some(top) = heap.pop() {

println!("{}", top);

}

}

// Prints in descending order: 10, 5, 1

## 6. Sorted BTreeSet and BTreeMap

use std::collections::{BTreeSet, BTreeMap};

fn main() {

let mut set = BTreeSet::new();

set.insert(3);

set.insert(1);

set.insert(2);

println!("{:?}", set); // {1, 2, 3} (sorted)

let mut map = BTreeMap::new();

map.insert(2, "b");

map.insert(1, "a");

map.insert(3, "c");

println!("{:?}", map); // keys sorted: {1: "a", 2: "b", 3: "c"}

}

## 7. Convert Between Collections

use std::collections::HashSet;

fn main() {

let nums = vec![1, 2, 2, 3];

let set: HashSet<\_> = nums.iter().cloned().collect();

let back\_to\_vec: Vec<\_> = set.into\_iter().collect();

println!("{:?}", back\_to\_vec); // order not guaranteed

}

## 8. Nested Collections

use std::collections::HashMap;

fn main() {

let mut students: HashMap<&str, Vec<&str>> = HashMap::new();

students.entry("Alice").or\_insert(vec![]).push("Math");

students.entry("Alice").or\_insert(vec![]).push("Physics");

students.entry("Bob").or\_insert(vec![]).push("Chemistry");

println!("{:?}", students);

// {"Alice": ["Math", "Physics"], "Bob": ["Chemistry"]}

}

## ✅ Takeaways

* Vec → use when order matters and you access by index.
* HashSet → remove duplicates, set operations.
* HashMap → group and count things.
* VecDeque → queue (FIFO) or double-ended operations.
* BinaryHeap → priority queue.
* BTreeMap/BTreeSet → keep elements sorted.

# 🦀 Rust Advanced Examples

## 1. Struct Example – **Bank Account**

struct BankAccount {

owner: String,

balance: f64,

}

impl BankAccount {

fn new(owner: &str, balance: f64) -> Self {

BankAccount {

owner: owner.to\_string(),

balance,

}

}

fn deposit(&mut self, amount: f64) {

self.balance += amount;

}

fn withdraw(&mut self, amount: f64) {

if self.balance >= amount {

self.balance -= amount;

} else {

println!("Insufficient funds!");

}

}

}

fn main() {

let mut acc = BankAccount::new("Alice", 1000.0);

acc.deposit(500.0);

acc.withdraw(200.0);

println!("{} has balance: {}", acc.owner, acc.balance);

}

✅ Shows **state changes** using methods.

## 2. Enum Example – **Traffic Lights**

enum TrafficLight {

Red,

Yellow,

Green,

}

impl TrafficLight {

fn duration(&self) -> u8 {

match self {

TrafficLight::Red => 30,

TrafficLight::Yellow => 5,

TrafficLight::Green => 25,

}

}

}

fn main() {

let light = TrafficLight::Green;

println!("Green light stays for {} seconds", light.duration());

}

✅ Encodes **different states with behavior**.

## 3. Trait Example – **Messaging System**

trait Messenger {

fn send(&self, msg: &str);

}

struct Email;

struct Sms;

impl Messenger for Email {

fn send(&self, msg: &str) {

println!("📧 Email sent: {}", msg);

}

}

impl Messenger for Sms {

fn send(&self, msg: &str) {

println!("📱 SMS sent: {}", msg);

}

}

fn notify<T: Messenger>(service: T, msg: &str) {

service.send(msg);

}

fn main() {

let email = Email;

let sms = Sms;

notify(email, "Hello via Email!");

notify(sms, "Hello via SMS!");

}

✅ Shows **polymorphism** in Rust.

## 4. Enum + Struct + Trait – **Game Characters**

trait Attack {

fn attack(&self) -> u32;

}

enum Weapon {

Sword,

Bow,

Magic,

}

struct Character {

name: String,

weapon: Weapon,

}

impl Attack for Character {

fn attack(&self) -> u32 {

match self.weapon {

Weapon::Sword => 50,

Weapon::Bow => 30,

Weapon::Magic => 70,

}

}

}

fn main() {

let hero = Character { name: "Archer".into(), weapon: Weapon::Bow };

println!("{} attacks with damage: {}", hero.name, hero.attack());

}

✅ Combines **enum choice + struct data + trait behavior**.

## 5. Multiple impl Blocks – **Temperature Converter**

struct Temperature(f64); // Celsius by default

impl Temperature {

fn new(c: f64) -> Self {

Temperature(c)

}

fn celsius(&self) -> f64 { self.0 }

}

impl Temperature {

fn to\_fahrenheit(&self) -> f64 {

(self.0 \* 9.0/5.0) + 32.0

}

}

fn main() {

let t = Temperature::new(25.0);

println!("{}°C = {}°F", t.celsius(), t.to\_fahrenheit());

}

✅ Demonstrates **associated functions + multiple impls**.

## 🔑 Key Takeaways

* **Structs** = hold data + methods for behavior.
* **Enums** = define states/variants.
* **Traits** = define common behavior across types.
* **Impl blocks** = add methods and constructors.
* Together, they let you **model real-world systems** (banking, traffic lights, games, converters).

# 🦀 Modules & Crates in Rust

## 1. What is a **Crate**?

* A **crate** is the smallest unit of compilation in Rust.
* Every Rust program is at least one crate.
* Crates can be:
  + **Binary crate** → produces an executable (main.rs).
  + **Library crate** → produces a .rlib that can be used by other crates (lib.rs).

👉 A **package** (Cargo project) can contain **one library** + multiple binaries.

**Example project structure:**

my\_pkg/

├─ Cargo.toml

├─ src/lib.rs ← library crate root

└─ src/bin/

├─ tool.rs ← binary crate 1 (cargo run --bin tool)

└─ helper.rs ← binary crate 2

## 2. What is a **Module**?

* A **module** is a way to **organize code** inside a crate.
* Defined with mod keyword.
* Controls **visibility** with pub.

**Example inside main.rs:**

mod math {

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

fn hidden() {} // private by default

}

fn main() {

println!("{}", math::add(2, 3)); // works

// math::hidden(); ❌ cannot call, it's private

}

## 3. File & Folder Structure

Modules can be split across files:

// main.rs

mod utils; // looks for src/utils.rs

mod network; // looks for src/network.rs OR src/network/mod.rs

src/

├─ main.rs

├─ utils.rs

└─ network/

├─ mod.rs

├─ client.rs

└─ server.rs

Inside network/mod.rs:

pub mod client;

pub mod server;

## 4. Paths in Modules

* crate:: → root of crate
* self:: → current module
* super:: → parent module

pub mod a {

pub mod b {

pub fn f() {}

}

}

// absolute path

use crate::a::b::f;

// relative path with alias

use self::a::b::f as g;

## 5. Re-exports (pub use)

Sometimes you don’t want to expose deep module paths to users.

// lib.rs

mod internal;

pub use internal::Config; // re-export

// external user writes:

use my\_crate::Config;

## 6. Cargo.toml & Dependencies

[dependencies]

serde = { version = "1", features = ["derive"] }

reqwest = { version = "0.12", features = ["json"] }

[features]

default = ["json"]

cli = []

👉 Enable/disable features:

cargo build --no-default-features --features cli

## 7. Workspaces (Multiple Crates Together)

Workspaces help manage multiple crates in one repo.

Cargo.toml (workspace root)

core/Cargo.toml

cli/Cargo.toml

web/Cargo.toml

Workspace Cargo.toml:

[workspace]

members = ["core", "cli", "web"]

Now crates can depend on each other:

# cli/Cargo.toml

[dependencies]

core = { path = "../core" }

## 8. Mini Example

Library (src/lib.rs):

pub fn greet(name: &str) -> String {

format!("Hello, {}", name)

}

Binary (src/bin/app.rs):

use my\_pkg::greet;

fn main() {

println!("{}", greet("Kamal"));

}

Run with:

cargo run --bin app

## ✅ Summary

* **Crate** → compilation unit (binary or library).
* **Module** → code organization inside a crate.
* **Paths** (crate::, self::, super::) → navigate modules.
* **pub** → makes items visible.
* **pub use** → re-export for cleaner API.
* **Cargo.toml** → manage deps, features, dev-deps.
* **Workspaces** → organize multi-crate projects.