coder [range]

Intensive Rust Bootcamp:

Day 1: Rust Introduction + Ownership Deep Dive

Intro Topics:

- Java/C++ to Rust Comparison: syntax, memory, error handling differences
- Why Rust: safety without GC, zero-cost abstractions, cross-platform power

Memory Segments:

- **STACK:** local variables, size known at compile time
- **HEAP:** dynamic allocations, Box, Vec, etc.
- **DATA:** global/static variables
- **TEXT/CODE:** compiled functions

Rust Basics :

Data type’s

Condition’s

Function’s

Rust Memory Basic’s :

Pointers , References

RAII (Resource acquisition is initialization)

- Ownership and moves

- Borrowing

- Lifetimes

ARRAY TYPES - Time : 1 hour

C++ VS RUST

Tuple data type

- Array data type

- Slice data type

Hands-On Programs:

- Rust Hello World (cargo based)
- let, const, shadowing, compound types
- Function with ownership transfer
- Code on memory code

Memory Diagrams:

- Ownership transfer
- Stack/Heap
- Borrow checker flow

Applications:

- Where: Embedded systems, CLI tools, crypto wallets
- Why: Predictable memory behavior, zero GC

Day 2: Control, Error Handling, Structs & Enums

Topics:

- Control Flow: if, match, while let, pattern guards
- Enums, structs: definition method’s, destructuring
- Result, Option, ? operator, custom errors

Rust Memory Hands on code :

RAII (Resource acquisition is initialization)

- Ownership and moves

- Borrowing

- Lifetimes

Hands-On Programs:

- Enum Shape with area()
- Propagate error with Result<T,E>
- Custom trait for logging
- Generic swap function

Error Handling -

panic. Panic values `abort` and `unwind`

- Option and unwrap

- Result. Iterating over Results

- Multiple error types

Memory Diagrams:

- Enum memory layout
- Match pattern breakdown
- Option/Result internal memory

Case Study: Mini UPI (Unified Payments Interface) app

- Handles transactions & logs errors via enums & traits

Applications:

- Where: Backend services, OS tools
- Why: Powerful pattern matching and safe error handling

Day 3: OOPS

OOP Concepts in RUST VS CPP/Java.

Topics :

Struct
Impl
Encapsulation
Abstraction
Polymorphism
Pointers

- Traits and trait bounds
- Generics, default methods

Introduction to traits in Rust - Time : 3 hour

- Derivable traits

- `dyn` keyword

- Operator overloading using traits

- Drop trait

- Iterator trait

- impl trait

- Clone trait

- Supertraits

Hands-On Programs:

1. **OOPS code**
2. **Inheritance code**

Memory Diagrams:

OOPS concepts
Objects
TRait’s

Case Study:

UPIApp simulator that supports:

- **Multiple users (`Bank` / `Wallet`)**
- **Inter-user transfer**
- **Logging transactions**
- **Error handling (insufficient funds, invalid users)**
- **Statement generation**

Applications:

Rust SDK for UPI or related services.

Day 4: Lifetimes, Collections, and Memory Safety

Topics:

Introduction to generics

- Defining generic functions

- Making implementation generic

- Defining Bounds for generic type

- The newtype idiom

- Item association

- Phantom type parameters

RAII, Ownership, and Safety:

- RAII model & Drop
- Move semantics and ownership rules
- Borrowing: & vs &mut
- Lifetimes: 'a, elision rules, lifetime bounds

Box type

- Vectors

- Strings

- Option

- Result

- HashMap

- Rc and Arc

- Lifetimes in functions/structs
- Borrow checker logic

Hands-On Programs:

- Vec of users with filtering
- Function with lifetime annotation
- Aggregate optional fields into HashMap

Memory Diagrams:

- Lifetime scopes
- Heap growth in Vec
- HashMap key-value memory model

Case Study: CLI Task

- Tracks tasks safely using ownership and borrows , DSA

Applications:

- Where: REST APIs, DB-backed apps
- Why: Compile-time safety with memory

Day 5: Traits, Smart Pointers, Macros, Async

Topics:

What are smart pointers

- The Deref and Drop trait

- Rc<T> and RefCell<T>

- Traits, dynamic dispatch
- Smart Pointers: Box, Rc, Arc, RefCell
- Interior mutability, weak refs
- Unsafe: *const, *mut, FFI basics
- Async with Tokio: futures, channels
- Declarative macros (macro_rules!) & #[derive]

MODULE

Purpose of modules in rust code

- Defining own custom module

- private and public visibility of members in a module

- use keyword for deep modules

- super and self keyword

Introduction to concept of crates in rust

- Custom libray creation and linking to another crate

- Cargo as Rust package management tool

- Managing dependencies using the cargo tool

- Using cargo to run unit and integration tests

- Cargo build scripts

Hands-On Programs:

- Trait with dynamic dispatch using Box<dyn>
- Shared counter via Arc<Mutex>
- Tokio async downloader
- Unsafe block with FFI

Memory Diagrams:

- Rc/Arc ref count model
- Async task lifecycle
- Macro expansion flow

Case Study: Async File Server with shared state

Applications:

- Where: Servers, device drivers, microservices
- Why: Predictable and concurrent execution

Day 6: Fearless concurrency

- Threads and thread safety
- Shared state concurrency:
  - Mutexes
  - Atomic types
- Message passing with channels
- Async/Await and the Tokio runtime
- Lock-free programming techniques

Difference between threading and async programming

- async/.await syntax
- Future trait
- Pinning
- The Stream trait
- join!, select!
- Shortcomings of the async programming model in Rust

Unsafe Rust and FFI - Time

- Understanding unsafe Rust
- Raw pointers and mutable statics
- Calling unsafe functions
- Foreign Function Interface (FFI) with C
- Best practices for minimizing and encapsulating unsafe code

Hands-On Programs:

An async log framework in Rust using custom macros and background writer task for high-performance, non-blocking logging.

Day 7: Framework

concurrency pattern Time

Worker poll implementation

Topics:

- HTTP client/server: Actix-web, Reqwest
- WebSockets, SSE
- SeaORM + PostgreSQL

RUST -Function

- Closures
- Capturing of data in closures
- Closures as input to functions and output parameters
- Higher order functions

Working with Databases

- Working with a SQL Database
- Serving a JSON API
- Testing and Building

Hands-On Programs:

- REST API: CRUD with Actix + SeaORM
- WebSocket notification system
- gRPC streaming service
- Wasm counter app integrated with JS

Memory Diagrams:

- HTTP route-to-response model
- DB schema to ORM memory map
- Wasm linear memory with JS bridge

Applications:

- Where: Real-time apps, SaaS, analytics
- Why: Type-safe full stack with top-tier speed

Day 8: Project-Based Learning

> 💡 Projects reinforce all concepts via real-world builds

Topics:

- Testing: unit, integration, proptest
- Benchmarking: Criterion
- File I/O, TCP/UDP with Tokio
- WebAssembly: wasm-pack, JS interop
- FFI: C, Java (JNI), Python (PyO3)

gRPC Vs REST

• gRPC API Types
gRPC with Tonic
• REST Paradigms

gRPC & protobuf

• Implement Unary gRPC API
• Implement Client RPC with Server-Side Streaming

Project 1: CLI Tool (0.5 day)

- Uses clap for CLI arg parsing
- Apply ownership, Result, and modules

Day 9: Project-Based Learning

Project 2: REST API with SeaORM (1 day)

- Actix CRUD API
- DB interactions, error flows, lifetimes

Creating and using Wasm modules in web projects
Invoking C functions from Rust, writing Rust bindings for C libraries.

Project : Concurrent Server (0.5 day)

- Tokio-based file or TCP server
- Use Arc, Mutex, channels, concurrency

Best Practices & Design Patterns (Threaded Throughout)

Topics:

- Rust idioms: Result<T, E>, ?, .iter(), unwrap_or, pattern matching
- Code structure: lib.rs vs main.rs, module tree
- Design Patterns:
  - Builder (config setup)
  - State Machine (e.g., server states)
  - Observer (via channels)
- Optimization:
  - For embedded: #![no_std], compile size, traits

- For backend: memory pools, async ops, Arc reus

Tools and Ecosystem
- Advanced Cargo usage
- Debugging Rust programs
- Profiling and benchmarking
- Useful crates for development

Assignments ::

🛠️ Assignment 1: ATM Machine with Limited Cash (RAII + Memory Control)

- **Topics covered**: Ownership, Borrowing, RAII, Stack/Heap usage.
- **Task**: Simulate an ATM machine:
  - Cash is stored as a struct with total amount.
  - Each withdrawal attempts to move/borrow from the cash store.
  - Ensure memory safety and no double free using ownership tracking.
  - Cash is automatically released (RAII) when ATM shuts down.
- **Challenge**: Print memory address & segment info (stack/heap) during operations for awareness.

🏦 Assignment 2: Loan Approval System (Option + Result Based Workflow)

- **Topics covered**: Option, Result, Custom Error Types, Pattern Guards.
- **Task**:
  - Take inputs: income, age, loan amount.

- Use:
  - `Option<T>` when checking optional co-applicant.
  - `Result<T, E>` for loan eligibility errors (`AgeError`, `IncomeError`).
- Implement `while let` and `match` for control flow.

- **Challenge**: Code must handle nested error types cleanly with `?` operator and propagate errors elegantly.

💾 Assignment 3: File Logger with Panic Handling (Real-World Safe Rust)

- **Topics covered**: File handling, Custom Traits, Panic handling (`abort` vs `unwind`), Memory Safety.
- **Task**:
  - Build a logger that writes to file.
  - On critical failure (disk full, permissions issue), `panic!`.
  - Show difference between `abort` and `unwind` behaviors.
  - Use trait for generalized logging (console/file).
- **Challenge**: Draw borrow checker flow chart for logger resource access.

5 Real-World Style Rust Assignments (Better & Deeper)

1️⃣Secure Digital Wallet CLI (Ownership + Borrowing + Traits + Errors)

Goal: Build a command-line **digital wallet** that supports:

- User accounts (struct).
- Balance check & fund transfers.
- Password-based authentication using borrowing (`&str`).
- Central logging using trait objects (`dyn Logger`).
- Use `Result<T, E>` for errors like `IncorrectPassword`, `InsufficientFunds`.
- Handle transaction history using vector slices.

Why better:

- Mimics real crypto-wallet models.
- Enforces borrow checking in authentication.
- Combines traits, ownership, error handling.

2️⃣ File-Based Database (Vec, Slice, Box, RAII, Lifetimes)

Goal: Create a **lightweight key-value store** using:

- File-backed persistence (append-only).
- Records stored as heap-allocated `Box` structs.
- Access records using slices (`&[T]`).
- Use lifetime annotations to avoid data leaks.
- Auto-writeback on shutdown using RAII.

Advanced Twist: Visualize Box memory allocation with stack/heap diagrams after each operation.

Why better:

- Mimics real backend storage techniques.
- Combines heap allocations, slices, and lifetime control.
- Reinforces resource management without GC.

3️⃣ Real-Time Airline Booking Engine (Match, Pattern Guards, Option, Result)

Goal: Simulate an **airline seat reservation system**:

- Planes have a vector of seats (`Vec<Option<Seat>>`).
- Allocate/free seats using Option matching.
- Return custom errors using Result types.
- Use pattern guards (`if let Some(seat) = ...`).
- Handle overbooking via `panic!` and recover.

Advanced: Draw Option memory layout diagram vs C++ nullable pointer.

Why better:

- Models concurrency-safe resource allocation.
- Reinforces Rust's Option/Result power.

4️⃣ Banking Transaction Processor (Enum Dispatch + Trait Objects + Threads)

Goal: Multi-threaded transaction processor with:

- Transaction types as Enums (`Deposit`, `Withdraw`, `Transfer`).
- Trait object for processing (`dyn Processor`).
- Background thread that:
  - Reads transaction queue.
  - Executes and logs each using dynamic dispatch.
- Communicate using channels (`std::sync::mpsc`).

Why better:

- Practical for backend/microservices.
- Exposes ownership/borrowing in thread-safe contexts.
- Shows trait object usage vs static dispatch.

5️⃣ Memory-Safe In-Memory Cache with Expiry (RAII + HashMap + Drop Trait)

Goal: Build an **in-memory caching layer**:

- Key-value store using `HashMap`.
- Supports automatic expiry via timers.
- Implement custom cleanup logic using `Drop` trait (on cache shutdown).

- Demonstrate RAII in resource cleanup.

- Slice-based retrieval for efficient memory reads.