



To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

Rust Essentials

Class Duration

21 hours of live training delivered over 3-5 days to accommodate your scheduling needs.

Student Prerequisites

- Professional software development experience in any modern language
- Comfort with the command line
- No prior Rust experience required

Target Audience

Software engineers adding Rust to their toolkit for systems programming, performance-critical services, embedded work, or memory-safe rewrites of existing C/C++ code. Equally suitable for developers using AI coding assistants who want to understand the Rust the compiler is asking them to write — not just accept what an LLM suggests.

Description

The Rust Essentials course is a modern introduction to Rust on the 2024 edition (Rust 1.85 introduced the edition; 1.96+ recommended for the current toolchain). It covers the core mental model — ownership, borrowing, lifetimes, expressive types, pattern matching, and idiomatic error handling — and pairs it with a realistic working setup: rust-analyzer, modern editor integration, and AI assistants (GitHub Copilot, Cursor, Claude Code) used as a productivity multiplier rather than a black box. Participants leave able to read and write idiomatic Rust, navigate the borrow checker confidently, and use AI tools effectively while letting the Rust compiler catch what AI gets wrong.

Learning Outcomes

- Read and write idiomatic Rust on the 2024 edition with confidence in ownership, borrowing, and lifetimes.
- Build small CLI applications and library crates with Cargo, including dependencies, tests, and documentation.



To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

- Model data with structs, enums (including `Option` and `Result`), tuples, and vectors, choosing the right shape for the problem.
- Apply pattern matching, `if let`, `while let`, and 2024-edition `let-chains` to write expressive control flow.
- Use AI coding assistants productively for Rust — generating boilerplate, navigating crates, drafting tests — while using the compiler and `clippy` as the source of truth.
- Diagnose common borrow-checker errors and fix them without resorting to `Rc/RefCell` or `unsafe` reflexively.

Training Materials

All students receive comprehensive courseware covering all topics in the course. Courseware is distributed via GitHub in the form of documentation and extensive code samples.

Software Requirements

A free GitHub account, the latest stable Rust toolchain installed via `rustup` (Rust 1.96+ on the 2024 edition), Visual Studio Code or another supported editor with the `rust-analyzer` extension, and an AI coding assistant of choice (GitHub Copilot, Cursor, or Claude Code). A cloud-based environment can be provided if local installation is restricted.

Training Topics

Introduction to Modern Rust

- What Rust is and where it fits today
- Rust 2024 edition: what changed and why it matters
- The borrow checker as a productivity tool, not an obstacle
- The Rust ecosystem: `crates.io`, `lib.rs`, and the Rust community
- The Rust Playground for quick experimentation

Toolchain and Editor Setup

- Installing Rust with `rustup` and managing toolchains
- `cargo`, `rustc`, `rustfmt`, and `clippy`
- VS Code, JetBrains RustRover, and Zed with `rust-analyzer`
- Debugging Rust with CodeLLDB
- Working with AI assistants in Rust: Copilot, Cursor, and Claude Code



To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

Hello, Cargo

- Creating a new project with `cargo new`
- The main function and `println!`
- `cargo run`, `cargo build`, `cargo build --release`
- Adding dependencies and managing `Cargo.toml`
- Workspaces at a glance and `cargo publish --workspace`

Scalar Types and Variables

- Integers, floats, booleans, characters
- Constants and statics
- Immutability by default and `mut`
- Variable shadowing and scoping
- Numeric overflow behavior in `debug` vs. `release`

Control Flow

- `if`, `else`, and expressions vs. statements
- `loop` with `break` and labeled loops
- `while` and `for` loops
- 2024-edition `let`-chains in `if` and `while`
- Block expressions returning values

Functions and Closures

- Defining and calling functions
- Parameter and return types
- Closures and their capture modes
- Function pointers vs. closures
- Where AI assistants help and where they hallucinate signatures

Modules and Crates

- Module hierarchy with `mod` and `use`
- Using the standard library
- Adding third-party crates from `crates.io`
- Re-exports and visibility

Built-In Macros

- `println!`, `println!`, `eprintln!`, and `format!`
- `vec!`, `dbg!`, and `assert!`
- `include_str!` and `include_bytes!`



To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

- `cfg!`, `env!`, and `panic!`

Memory Management and Ownership

- Why neither manual memory management nor garbage collection is ideal
- Ownership, moves, and copies
- Borrowing: `&T` and `&mut T`
- Lifetimes and the borrow checker
- Common ownership patterns and how to talk to AI tools about them

Strings and String Slices

- `String` vs. `&str`
- UTF-8 handling and Unicode boundaries
- Parsing, trimming, splitting, and joining
- String formatting and interpolation

Tuples, Enums, and Structs

- Tuples and tuple structs for lightweight data
- Enums with associated data and variant methods
- `Option<T>` and `Result<T, E>` as the cornerstone of Rust error handling
- Structs, methods, associated functions, and constructors
- The newtype pattern

Vectors and Collections

- `Vec<T>`: creation, indexing, slicing, iteration
- Iterators: `iter`, `iter_mut`, `into_iter`
- `HashMap` and `HashSet` at a glance
- Ownership and borrowing rules with collections

Pattern Matching

- match exhaustiveness and the borrow checker
- `if let` and `while let`
- Destructuring structs, tuples, and enums
- Pattern guards and bindings
- Refutable vs. irrefutable patterns

Idiomatic Error Handling

- `Result<T, E>` and the `?` operator
- Defining custom error enums



To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

- Choosing between `anyhow` and `thiserror`
- Avoiding `unwrap` and `expect` in library code

AI-Assisted Rust Development

- Using `rust-analyzer` + AI assistants as a tight feedback loop
- Letting the compiler and `clippy` be the source of truth
- Prompts that work well for Rust ownership and lifetime questions
- Reviewing AI-generated `unsafe` and `unwrap` calls before accepting them
- A realistic CLI feature built end-to-end with AI assistance and the compiler