17-350 / 17-650

Lecture 01: Safe Systems Programming in Rust

Jonathan Aldrich
Aug 25, 2026

Carnegie Mellon University

S3D Software and Societal Systems Department

Today at a Glance

- Course introduction
- Rust overview
- Rust interpreter demo

Why Rust for Systems Code?

Rust is the future of system programming!

- As efficient as C or C++, better than Go
- Can work at systems level of abstraction
- Stronger safety guarantees even than Java
  - Memory safety
  - Concurrency safety
- Great abstraction mechanisms

Learning Goals

- Write medium-sized systems programs in Rust.
- Use mechanisms such as ownership, safe manual memory management, safe concurrency, and async effectively.
- Encapsulate unsafe code behind safe interfaces.
- Apply safe code writing techniques across various system domains.
- Understand safety principles that can be applied even outside Rust

Course Resources

- Course home page:
  - https://www.cs.cmu.edu/~aldrich/courses/17-350-fa26/
- Syllabus and policies:
  - https://www.cs.cmu.edu/~aldrich/courses/17-350-fa26/syllabus.html
- Recommended Rust learning resource:
  - https://rust-book.cs.brown.edu/

Lecture Topics and Roadmap

- Ownership, borrowing, and lifetimes
- Safe concurrency and async
- Error handling and traits
- Modules/crates, macros, and FFI/unsafe encapsulation
- Systems domains: interpreters, distributed systems, embedded, WebAssembly
- Beyond types: verification in Rust with Verus
- Research on safety in Rust & similar languages

Assignments and Grading

- Grade breakdown:
  - 60% assignments
  - 20% in-class participation
  - 20% final project
  - Standard percentages (90-80-70-60), +/- for graduate students and at mid-semester
- Course rhythm:
  - Frequent assignments and in-class exercises
  - ~12 units: about 3 hours in class + 9 hours outside class each week

What you will build

- Warm-up: a basic interpreter for (a fraction of) WebAssembly
- A fast interpreter or JIT compiler
- A WASM image processing framework
- A distributed key-value store
- An embedded safety monitor
- A verified Rust systems program
- A final project limited by your imagination (and 3.5 weeks of time)

Core Course Policies

- Late work:
  - 5 automatic free late days
  - Then 10% penalty/day up to 5 days late
  - More than 5 days late: feedback, no credit
- Collaboration and integrity:
  - Individual work unless explicitly marked group work
  - Cite sources; no copying or unauthorized sharing
  - AI tools allowed, but you must understand and own submitted work
    - we may audit this
- Support:
  - Accommodations and health resources are available

Course introductions

Please introduce yourself and share why you are interested in the course

History of Rust

- Created to combine systems-level performance with stronger safety guarantees
- Started at Mozilla Research and grew through open-source adoption
- Designed for modern software realities:
  - Concurrency everywhere
  - Large codebases with long maintenance lifetimes

How Rust Achieves Memory + Concurrency Safety

- Make common memory/concurrency bugs compile-time errors
- Key static checks:
  - Ownership and move semantics
  - Borrowing rules (`&T` and `&mut T`)
  - Lifetimes that prevent dangling references
  - `Send` / `Sync` constraints for thread-safe sharing
  - `Option` / `Result` for explicit state and failure handling

Rust Mechanisms for Safe Systems Programming

- Ownership + borrowing + lifetimes
  - Prevent use-after-free, double free, dangling pointers
- RAII + deterministic drop
  - Ensure reliable cleanup of resources
- Type-driven concurrency
  - Prevent data races by construction
- `unsafe` isolation patterns
  - Keep low-level power behind safe APIs

Let's look at a simple Rust program.

Demo: a simple interpreter in Rust

Takeways and Next Steps

- Rust is an awesome language for systems programming
- We'll learn about how to write systems programs safely--in Rust and other languages
- Several super cool systems coding projects
- Next steps
  - Read the syllabus
  - First assignment released tonight, due in 1 week
    - Rust warm-up
  - Invite your friends to join the class
- Next lecture: Writing Fast Interpreters