C4: Rust vs. C Comparison Report

Meera Al Wadhi

Hamda

May 2025

Introduction

Welcome to our report comparing the classic `C4.c`, a tiny C compiler written in C, with our fresh take on it in Rust. Our project, dubbed `c4_rust_Alain`, aimed to recreate the core features of the C4 subset of C—like integers, functions, `if`/`while` statements, and `return`, while tapping into Rust's superpowers: memory safety, modularity, and a slick modern toolchain. You can check out our work on GitHub at https://github.com/meeraalawadhi/c4_rust_team_alfaqa. Let's dive into how these two versions stack up!

Architecture Comparison

We broke down the key differences between the C and Rust implementations to see how they handle the same tasks. Here's a quick look:

How C and Rust compare in their approach to building C4.

Aspect	**C (c4.c)**	**Rust (main.rs)**
Language Features	Classic procedural C with manual memory management	Safe, statically typed Rust with no memory worries
Lexer	Built into `next()` function	Standalone `tokenizer()` for clarity
Parser	Single-pass, all in `main()`	Multi-function recursive-descent parser
Code Generation	Spits out bytecode into a global `e` buffer	Builds an instruction vector via AST traversal
Virtual Machine	Manual registers and heap management	Stack-based VM using Rust's enums
Error Handling	Prints error and exits	Uses `panic!` For clear, early exits
Extensibility	Tricky due to its one-big-file design	Super flexible with enums, pattern matching, and modules

Feature Parity

Did we bring over all the cool stuff from C to Rust? Here's the rundown:

We brought features from C to Rust, and we're at.

Feature	C	Rust	Notes
`int` type			Handled by `Token::Int` and `Expr::Number`
Functions			Captured in `FunctionDef` in the AST
`if/else`			Supported via the `If` node in AST
`while`			Loops work in both AST and VM
`return`			Maps to `PSH` + `EXIT` in VM
Arithmetic Ops			Translated to VM instructions
Comparisons			Handled with dedicated opcodes
Pointers	Partial		Not yet in Rust—coming soon!
`char`, `enum`			Not in Rust's scope yet
Self-hosting		Partial	Rust parses `c4.c`, but full self-hosting is a work in progress.

Testing and Compatibility

We've been busy making sure our Rust version holds up! We wrote over 20 unit tests in Rust to check the tokenizer, parser, and VM. The Rust version nails arithmetic expressions, branching, loops, and function calls. We're still working on getting it to parse c4.c fully for complete self-hosting, but we've added some bonuses like logical operators, memory load/store, and function calls with parameters. Pretty neat, right?

Why Rust Shines

Switching to Rust brought some serious perks:

- **Memory Safety**: No more worrying about buffer overflows or rogue pointers!
- **Modularity**: The lexer, parser, and VM are neatly separated, making changes a breeze.
- **Clear Errors**: Rust's `panic!` Gives us detailed stack traces to debug faster.
- **Awesome Tools**: `cargo test`, `cargo doc`, and auto-formatting make development smooth.
- **Future-Proof**: Adding new features or optimizations is way easier with Rust's design.

Challenges We Faced

It wasn't all smooth sailing. Here's what gave us a bit of a headache:

- Translating C's low-level control flow (like jump tables) into Rust's idiomatic style.
- Dynamically resolving variables and functions without leaning on global state.

- Avoiding mutable global variables, which Rust doesn't like.
- Building a parser that handles nested expressions and operator precedence cleanly.

Wrapping Up

Our Rust version captures most of `c4.c`'s core functionality while being safer, easier to maintain, and more testable. We haven't tackled pointers or full self-hosting yet, but the foundation is solid for adding those. Rust has made this project feel like a playground for future ideas!

==========