C4 Rust Compiler – Comparison Report

Team: Jabal Ali Village

1. Safety Features

Rust's memory safety model gave our compiler an immediate upgrade.
Thanks to ownership, borrowing, and strict compile-time checks, we avoided common C issues like:

- Null pointer dereferencing
- Use-after-free bugs
- Memory leaks

These protections made our Rust version more stable and prevented entire classes of runtime errors without needing a garbage collector.

2. Design Differences

Rust’s modern language design made our compiler much cleaner to implement.
Here are a few stand-out differences:

- **Enums + Pattern Matching**: Let us write concise and expressive logic in the parser and lexer.
- **Option / Result Types**: Helped us handle errors safely instead of crashing.
- **Structs + Borrowing**: Made it easy to add position tracking for tokens (a bonus feature).

What felt complex in C became elegant in Rust.

3. Performance

In early testing, our Rust compiler performed on par with the original C4.
Even though Rust enforces safety, it doesn't sacrifice speed:

- Fast tokenizing, parsing, and evaluation
- Zero-cost abstractions kept performance snappy
- If needed, we can benchmark precisely using tools like Criterion.rs

4. Challenges Faced

Rewriting from C to Rust wasn’t copy–paste. We hit a few key challenges:

- Translating C-style pointers and mutable global state into Rust’s strict ownership model
- Designing the AST and virtual machine to work without garbage collection
- Managing lifetimes and avoiding borrow-checker conflicts when building the parser

We skipped self-hosting tests (compiling itself) to focus on building a stable MVP but it's a future goal.

5. Bonus Feature: Better Error Reporting

In Rust, we added position tracking for every token line and column numbers.
This allowed us to report syntax errors like this:

Error at line 1, column 10: Expected ';'

Instead of crashing like C4, our Rust version guides the user to the problem.

6. Additional Improvements

Our Rust compiler improved on the original by:

- Supporting all basic binary operations: +, -, *, /
- Parsing variable assignments like x = 5;
- Adding helpful error messages throughout the parser and VM
- Making the codebase modular and readable every component lives in its own file

Overall, the rewrite gave us the chance to rethink and improve the original design.

7. Conclusion

Rewriting C4 in Rust was a challenge, but the results were worth it.

- The compiler is safer, more maintainable, and just as fast
- The language features made our implementation more expressive
- We now have a strong foundation to build on (floating point support, more C syntax, etc.)

Rust didn't just help us reimplement the compiler it helped us improve it.