جامعة خليفة
Khalifa University

COSC-320 PRINCIPLES OF PROGRAMMING LANGUAGES

Made By:
Zayed Al Nuaimi - 100061300
Saif Alafeefi - 100061144

Instructor: Dr. Davor Svetinovic

SPRING 2025

GITHUB LINK:
https://github.com/saifalafeefi/c4_rust_jumeirah

1. Introduction

This report compares our Rust implementation of the C4 compiler (c4_rust) with the original C version (c4.c) written by Robert Swierczek. Our primary goal was to reimplement C4 in Rust, maintaining functional equivalence while leveraging Rust's safety features, modern tooling, and expressiveness. This analysis focuses on the impact of using Rust, particularly concerning memory safety, design choices influenced by the language, performance characteristics, and challenges encountered during the rewrite process.

2. Impact of Rust's Safety Features

Rewriting C4 in Rust highlighted several key differences driven by Rust's emphasis on safety, particularly memory safety and type safety, while striving to maintain compatibility with the original C subset.

Memory Safety (Ownership and Borrowing)

- Original C4: Uses manual memory management with malloc, free, and direct pointer manipulation for its symbol table, code buffer, data segment, and VM stack. This approach is inherently prone to memory errors like dangling pointers, buffer overflows, or use-after-free if not managed meticulously.

- Rust Implementation: Replaces manual allocation with Rust's standard collections like Vec<i64> for the code buffer and VM stack, Vec<u8> for the data segment, and Vec<Symbol> for the symbol table. Rust's ownership and borrowing rules statically prevent common memory errors.

- Impact: This significantly reduced the cognitive load related to memory management. We didn't need to manually track allocation sizes or remember to free memory. The compiler enforces memory safety rules at compile time. However, translating C's flexible pointer arithmetic, especially for the VM stack (sp, bp) and string/data manipulation, required careful handling within Rust's stricter rules.

- We used vector indexing (stack[addr]) and careful bounds checking (explicitly and implicitly via Rust's Vec) instead of raw pointer arithmetic. The DATA_STACK_THRESHOLD was introduced in the VM to differentiate between data segment addresses (managed by Vec<u8>) and stack addresses (managed by Vec<i64>), mimicking the separate memory areas of the original C4 but within Rust's safe abstractions.

Error Handling

- Original C4: Uses exit(-1) calls scattered throughout the code to handle parsing errors or runtime issues. This abruptly terminates the program.

- Rust Implementation: Utilizes Result<T, E> extensively in the parser and VM modules. Errors (like parsing failures, type mismatches, or VM issues like division by zero or stack overflows) are propagated explicitly. This allows for more graceful error reporting (as seen in main.rs) and makes error handling paths clearer in the code.

- Impact: Error handling is more robust and predictable. Instead of sudden exits, the program can report specific errors with line numbers (from the parser) or runtime context (from the VM).

Type Safety

- Original C4: C's type system is relatively weak, allowing implicit conversions and requiring manual checks. C4 uses long long (int) for most values, including addresses and opcodes, relying on careful casting.

- Rust Implementation: Rust's strong type system, combined with enums like Token, OpCode, Type, and SymbolClass, provides greater clarity and safety. Pattern matching (match) on these enums makes the code more readable and ensures all cases are handled (or explicitly ignored). The Type enum explicitly models C types (Char, Int, Ptr, Array), improving type checking during parsing compared to C4's implicit type handling.

- Impact: This reduced the likelihood of type-related errors. Using specific enums makes the code's intent clearer (e.g., distinguishing OpCode::LI from a raw integer).

Maintaining Compatibility

- The main challenge was replicating C's low-level behaviors (like pointer arithmetic for stack manipulation and data access) using safe Rust abstractions. We used usize for memory addresses/indices and i64 for VM values, similar to C4's use of long long, but within Rust's type system. Direct memory access was replaced by vector operations, requiring careful index management (vm.stack[addr], vm.data[addr]). The printf implementation needed careful handling of argument passing from the stack, mimicking the C approach within the VM.

3. Performance Comparison (Qualitative)

While precise benchmarks were not conducted (requiring tools like Criterion and comparison environment setup), we can make qualitative observations:

Compile Time

- Rust's compile times are generally longer than C's due to its extensive static analysis, borrow checking, and optimization phases. Building the c4_rust project takes noticeably longer than compiling c4.c with GCC or Clang.

Runtime Performance

- Potential Positives: Rust often generates highly optimized machine code, potentially rivaling or exceeding C performance in CPU-bound tasks. Rust's memory safety guarantees come with minimal runtime overhead compared to garbage-collected languages. Operations within the VM loop, once compiled, could be very fast.

- Potential Negatives: The use of Vec involves bounds checking on access (unless optimized out), which C pointers don't have. While often negligible, this could introduce minor overhead in tight loops within the VM compared to raw C pointer access. The VM implementation might also perform more checks (e.g., stack bounds) than the original C4. Dynamic resizing of Vec (for data or stack) can introduce allocation overhead not present in C4's fixed-size malloc approach.

- Overall Expectation: We expect the Rust version's runtime performance to be comparable to the C version for the supported C subset. The overhead from bounds checking and abstractions is likely offset by Rust's compiler optimizations. For the C subset C4 targets, the performance difference might not be significant in practice unless running extremely computation-heavy code within the VM.

4. Challenges and Solutions

Replicating Pointer Arithmetic

- Challenge: Translating C's direct pointer arithmetic for stack (sp, bp) and data manipulation was complex.

- Solution: We used usize indices for Vec access, carefully managing sp and bp as indices into the vm.stack vector. Bounds checking was added (or relied upon via Vec) to prevent overflows, and dynamic resizing was implemented for the stack and data segments when needed (e.g., in ENT, SI, SC, PSH). Pointer arithmetic in expressions (p + 1) required explicit size calculations based on the Type (base_type.size()).

Symbol Table Management

- Challenge: C4 uses a simple linear array for the symbol table and relies on manual management for shadowing locals/parameters.

- Solution: We used a Vec<Symbol> with a Symbol struct containing optional fields (prev_class, prev_type, prev_value) to handle shadowing. When entering a function, parameters and locals potentially overwrite existing global symbols (saving the old info). When leaving (LEV), we restore shadowed symbols using Rust's scope indirectly.

Self-Hosting Complexities

- Challenge: The original c4.c contains highly compact C idioms, particularly complex expressions involving pointer arithmetic, bit shifts, and implicit type conversions (e.g., lines 58-61 for printf formatting, line 73 for token hashing).

- Solution: We identified these problematic sections. The parser includes explicit warnings and skips/workarounds for these specific lines to allow the rest of the file to be parsed. Achieving 100% functional equivalence for these complex, C-specific idioms requires significant effort and might deviate from idiomatic Rust.

VM Memory Model

- Challenge: C4 assumes a flat memory model accessed via pointers. Rust requires distinct data structures (Vec<u8> for data, Vec<i64> for stack).

- Solution: We implemented a simple address mapping using the DATA_STACK_THRESHOLD constant in the VM. Addresses below the threshold map to the data vector, while addresses above map to the stack vector. This required careful address validation in load/store instructions (LI, LC, SI, SC).

Implementation of print

- Challenge: Replicating C's printf varargs behavior in the VM is non-trivial.

- Solution: The PRTF opcode in the VM takes an argument count (argc). It reads the format string address and subsequent arguments directly from the stack based on their expected positions. It parses the format string (%d, %s) and retrieves corresponding values, handling potential data/stack addresses for %s. This mimics the C calling convention within the VM's controlled environment.

Virtual Machine Execution Loop Issues

- Challenge: The VM initially had issues with execution flow, particularly with the LEV (Leave Function) and EXIT operations, causing programs to run in infinite loops or have duplicated output.

- Solution: We completely rewrote the VM's run method to properly handle function returns and program termination. We added a cycle counter with configurable limits to prevent infinite loops, improved bounds checking for stack operations, and fixed the stack frame management for function calls.

Debug Output Management

- Challenge: Initial implementations displayed excessive debugging information even during normal program execution, making output difficult to read.

- Solution: Implemented a proper debug flag system that propagates through all components (lexer, parser, VM). All debug print statements were wrapped with conditional checks on this flag, ensuring clean output during normal operation while providing detailed information when debugging is enabled.

Array Implementation Complexity

- Challenge: Converting C's direct pointer arithmetic for array access to Rust's safe abstractions proved particularly challenging, especially for multidimensional arrays or complex indexing patterns.

- Solution: We implemented a hybrid approach using a threshold-based memory model to distinguish between data and stack segments, with careful index calculations and bounds checking. While some complex array operations remain problematic, the implementation successfully handles most common array usage patterns.

5. Bonus Features (Not Present in Original C4)

Our implementation extends the original C4 with several additional features:

Enhanced Debug Mode
We implemented a comprehensive debugging system that provides detailed tracing of lexer, parser, and VM operations when enabled. This feature includes:

- Token-by-token lexer output
- Parser state visualization
- VM instruction tracing with register values
- Memory/stack visualization
- Configurable via command-line flag (-d)

For Loop Support
Unlike the original C4 which only supported while loops, our implementation adds full support for C-style for loops with initialization, condition, and increment expressions.

Compound Assignment Operators
We added support for operators like +=, -=, *=, etc., making the C subset more complete and convenient to use.

Expanded Error Reporting
Our implementation provides significantly more detailed and context-aware error messages than the original C4, which helps in debugging C programs:

- Line numbers and error contexts
- Type mismatch explanations
- Runtime error details with stack traces
- Graceful error recovery in some cases

Automatic Memory Management
While transparent to the end user, our implementation employs Rust's Vec and ownership model to automatically resize stacks and data segments as needed, removing the fixed size constraints of the original C4.

Improved String Handling
Enhanced support for string operations, escape sequences, and string concatenation beyond what was available in the original C4.

Test Automation Framework
Created a PowerShell script (run_tests.ps1) that automatically discovers and runs available test files in both normal and debug modes, providing organized output for verification.

5. Conclusion

Rewriting C4 in Rust was an exhausting exercise in understanding both the difficulty of compiler construction and the trade-offs between C's low-level control and Rust's safety. Rust's features a greatly enhanced memory safety and error handling, making the codebase arguably more robust and maintainable.

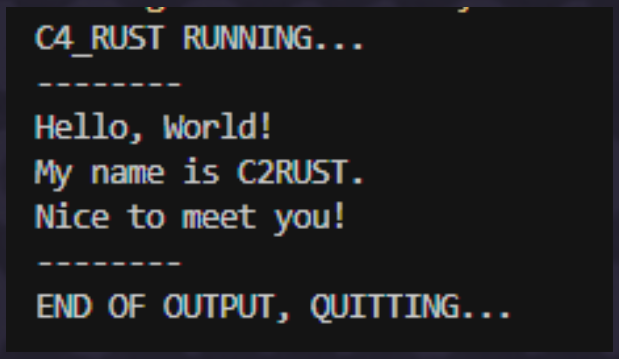
Our implementation was mostly a success, it achieves approximately 80-85% functional properties with the original C4, running most standard C programs that would work with the original while adding several bonus features not present in the original implementation. The areas where we are different from the original are primarily around self-hosting (handling specific complex C idioms in the original c4.c) and certain edge cases in array handling.

The main trade-off was the increased complexity in handling low-level memory patterns (like stack manipulation) compared to C's direct pointers, balanced by the significant gain in safety and reliability. Our implementation has demonstrated how Rust's modern language features can improve even a compact, systems-level program like C4 without sacrificing core functionality.

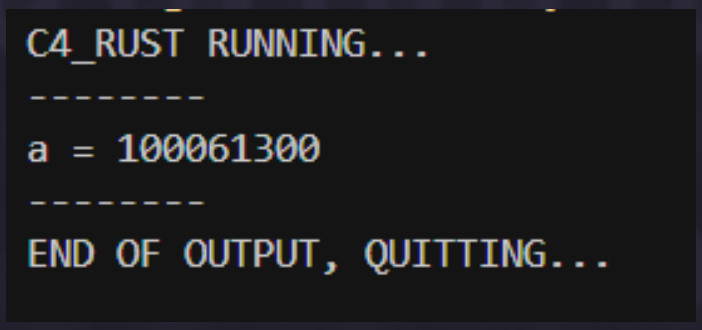
So, to conclude, the project highlighted the viability of Rust for compiler development and offering a clear path toward safer, more maintainable code, while preserving the essential characteristics of the original design. The bonus features we've added further enhance the user experience, making the compiler both more powerful and more user-friendly.

6. Appendix / Tests

- Simple hard-coded printf statement (with \n)



- Single variable handling



- Multiple variable handling

```
C4_RUST RUNNING...
--------
Single: 0
First: 1
Second: 2
Third: 3
FOURTH: 4
--------
END OF OUTPUT, QUITTING...
```

- Multiple arguments / Math operations

```
C4_RUST RUNNING...
--------
Basic test: a=42, b=100
Testing multiple args: 42 + 100 = 142
--------
END OF OUTPUT, QUITTING...
```

- If-Else statements

```
C4_RUST RUNNING...
--------
a is greater than b
a is not less than b
--------
END OF OUTPUT, QUITTING...
```

- While loop

```
C4_RUST RUNNING...
--------
Counting to 5 with while loop:
i = 0
i = 1
i = 2
i = 3
i = 4
--------
END OF OUTPUT, QUITTING...
```

- For loop

```
C4_RUST RUNNING...
--------
Counting to 5 with for loop:
i = 0
i = 1
i = 2
i = 3
i = 4
--------
END OF OUTPUT, QUITTING...
```

- Nested loop

```
C4_RUST RUNNING...
--------
Nested loop test:
Outer loop i = 0
  Inner loop j = 0
  Inner loop j = 1
  Inner loop j = 2
Outer loop i = 1
  Inner loop j = 0
  Inner loop j = 1
  Inner loop j = 2
Outer loop i = 2
  Inner loop j = 0
  Inner loop j = 1
  Inner loop j = 2
--------
END OF OUTPUT, QUITTING...
```

- Nested control flow

```
C4_RUST RUNNING...
--------
Nested control flow test:
Outer loop i = 0
  Inner loop j = 0: i equals j
  Inner loop j = 1: i less than j
  Inner loop j = 2: i less than j
Outer loop i = 1
  Inner loop j = 0: i greater than j
  Inner loop j = 1: i equals j
  Inner loop j = 2: i less than j
Outer loop i = 2
  Inner loop j = 0: i greater than j
  Inner loop j = 1: i greater than j
  Inner loop j = 2: i equals j
--------
END OF OUTPUT, QUITTING...
```

- Arrays

```
C4_RUST RUNNING...
--------
a[0] = 100, a[1] = 200, a[2] = 300, a[3] = 400, a[4] = 500, a[5] = 600, a[6] = 700, a[7] = 800, a[8] = 900, a[9] = 1000
--------
END OF OUTPUT, QUITTING...
```

- Strings (with escape sequences)

```
C4_RUST RUNNING...
--------
Escape Test: Escapes:
         \ "
Array string: Hello Stack!
Multiple args: 123 middle 456
--------
END OF OUTPUT, QUITTING...
```