Programmiersprachen im Vergleich

Artur Papoyan

January 6, 2026

Abstract

1 Introduction

- Motivation
- Overview of the paper structure

2 Rust Highlights

2.1 Origin and Design Philosophy

- History and Background
- Goal Memory Safety without Garbage Collector
- Role of the Rust Foundation

2.2 Ownership and Borrowing

- Ownership model: who owns what?
- Borrowing: shared and exclusive references
- Borrow Checker and lifetimes

2.3 Memory Management Without Garbage Collector

Unlike many high-level languages, Rust achieves memory safety without a garbage collector. Memory is managed through the ownership system described above. When an object's lifetime ends, its memory is immediately freed by Rust's runtime (through an automatic call to its destructor, if it has one) at the end of the scope. This approach yields deterministic memory management: the programmer (and compiler) know exactly when an object will be deallocated, which is typically when it goes out of scope. There is no need to wait for a garbage collector to periodically find and reclaim unused memory. As a result, Rust programs avoid the runtime overhead of GC pauses and can often use memory more efficiently.

To illustrate, consider a simple example: if you allocate a buffer inside a function in Rust (e.g., by creating a Vec<u8> inside the function), that buffer's memory will be freed as soon as the function returns and the vector goes out of scope. This is done automatically by Rust's standard library implementation of Vec in its Drop trait. The programmer does not have to manually free the memory (as in Cs free()), nor worry about a garbage collector cleaning it up eventually, the cleanup happens immediately when it should. The strict ownership rules ensure that there are no dangling pointers to that buffer at the time of deallocation, because the compiler would not allow those pointers to exist beyond the scope.

In summary, Rust's memory management can be seen as an automated scope-based memory management (similar to C++ RAII) with compile-time checks to ensure safety. This approach contrasts with garbage-collected languages where memory safety is attained by automatic tracing and collection (with potential runtime costs and nondeterministic timing) and with traditional C/C++ where manual memory management can be efficient but error-prone. Rust manages to largely "have its cake and eat it too" memory safety without garbage collection, and performance without sacrificing safety. As a result, Rust is particularly attractive for systems programming tasks where predictable performance and low-level control are required, but one also wants to avoid the memory errors that have historically caused security vulnerabilities in C/C++ software.

- Automatic deallocation via scope-based drop
- Comparison to Garbage Collector

2.4 Concurrency and Synchronization

- Fearless Concurrency
- Safety + Performance
- Zero-Cost Abstractions
- Send/Sync
- Threads, Channels
- async/await + runtime ecosystem

2.5 Type System and Language Characteristics

- Static typing
- Traits and Generics
- Pattern Matching
- Error handling: Result, Option

2.6 Tooling and Ecosystem

- Cargo (Package Manager and Build System)
- Crates.io (Ecosystem)
- rustup and Toolchains
- rustdoc, Testing, Benchmarking

2.7 Current Developments

2.8 Comparison with Other Languages

- Rust vs. C++

3 Practical Section

4 Conclusion