Course: Advanced Programming

Project: C12 - C, C++, and Rust: A Comparative Study of Systems Programming Models

ID: 718228

Name: Ramy Ghoubrial Abdelshahid Ghoubrial

Part 1 — Key Differences Between C, C++, and Rust in Systems Programming

This section analyzes how **C, C++, and Rust** approach systems programming, focusing on **memory management**, **type systems**, **safety guarantees**, **compilation models**, and the fundamental **trade-off between low-level control and language-enforced correctness**. The goal is not to rank the languages, but to clarify *what each language allows, prevents, or enforces* at the systems level.

1. C — Maximum Control, Minimal Built-in Safety

Memory Management

C relies entirely on **manual memory management**. Programmers explicitly allocate and deallocate memory using functions such as malloc() and free(), and the language does not track ownership or lifetimes.

As a result, many incorrect programs still compile successfully, with errors manifesting only at runtime—or silently corrupting memory. C provides maximal freedom, but correctness depends entirely on programmer discipline.

Type System

C has a relatively **simple and weak type system** compared to C++ and Rust:

- No generics or templates
- Abstraction is often achieved using macros, void*, and coding conventions

The compiler does not prevent many forms of incorrect pointer usage or memory misuse.

Safety Guarantees

C provides **no memory safety guarantees**. Many common errors lead to **Undefined Behavior (UB)**, including:

- Use-after-free
- Double free
- Out-of-bounds memory access
- Invalid pointer arithmetic
- Type-punning violations

When UB occurs, the compiler is allowed to generate arbitrary behavior. C therefore places full responsibility for correctness on the programmer.

Compilation Model

C is compiled directly to machine code with a **minimal runtime**, offering:

- Predictable performance
- Excellent portability
- Simple interoperability with hardware and other languages

However, correctness heavily relies on external practices such as code review, testing, and tooling (e.g., sanitizers and static analysis).

Trade-off Summary

Advantages

- Absolute low-level control
- Minimal runtime overhead
- Ideal for kernels, embedded systems, and low-level libraries

Disadvantages

- High long-term correctness cost
- Error-prone in large or evolving codebases

2. C++ — Low-Level Power with Optional Safety

C++ extends C by adding powerful abstraction mechanisms while preserving low-level control. Safety is **enabled by the language**, but not enforced.

Memory Management

C++ still allows manual memory management (new/delete, malloc/free), but modern C++ strongly encourages:

- **RAII (Resource Acquisition Is Initialization)**
- Standard containers (std::vector, std::string)
- Smart pointers (std::unique_ptr, std::shared_ptr)

In well-written modern C++, memory management is *manual in capability but automatic in practice*.

Type System

C++ has a **much stronger and more expressive type system** than C:

- Templates (compile-time generics)
- Function overloading
- Classes and object-oriented programming
- Compile-time type checking

However, unsafe constructs such as raw pointers, casts, and lifetime misuse remain possible.

Safety Guarantees

C++ improves safety by providing **safer patterns**, not by enforcing them:

- Undefined Behavior still exists (dangling references, invalid casts, data races, out-of-bounds access)
- Both highly safe and highly unsafe code can be written in valid C++

Safety therefore depends heavily on coding style, discipline, and conventions.

Compilation Model

C++ is compiled ahead-of-time like C and emphasizes **zero-cost abstractions**, meaning abstractions are designed to compile away without runtime overhead.

However, the language's size and complexity increase:

- Compile times
- Error message complexity
- Risk of subtle bugs

Trade-off Summary

Advantages

- Excellent balance between performance and abstraction
- Mature ecosystem and tooling
- Suitable for large systems when used correctly

Disadvantages

- Safety is optional, not guaranteed
- High language complexity

3. Rust — Strong Compile-Time Safety with Systems-Level Performance

Rust is designed to provide **systems performance with correctness enforced by the compiler**.

Memory Management

Rust does not use garbage collection. Instead, it enforces:

- **Ownership**: each value has exactly one owner
- **Borrowing**: references must not outlive the data they point to
- **Lifetimes**: compile-time rules used to validate references

Memory is released automatically when ownership ends, similar to RAII, but enforced more strictly at compile time.

Type System

Rust's type system is both **strong and expressive**:

- Generics and trait bounds

- Algebraic data types (enum)
- Option and Result for explicit handling of absence and failure

This design reduces silent error patterns and forces many correctness decisions to be explicit.

Safety Guarantees

In **safe Rust**, the compiler prevents entire classes of bugs:

- Use-after-free
- Double free
- Null-pointer dereferencing (for references)
- Data races in concurrent code

Undefined Behavior is largely confined to unsafe blocks, where the programmer explicitly opts out of compiler guarantees and assumes responsibility.

Compilation Model

Rust is compiled ahead-of-time (LLVM-based) and strongly emphasizes **zero-cost abstractions**.

The borrow checker can make initial development slower, but significantly reduces debugging effort and runtime failures.

Trade-off Summary

Advantages

- Strong safety guarantees by default
- No garbage collector
- High-performance systems code with enforced correctness

Disadvantages

- Steeper learning curve
- Requires redesigning code to satisfy ownership rules

4. Direct Comparison: Control vs. Correctness

Error Prevention Philosophy

- **C**: “You are trusted.” → Errors are possible and often become UB.
- **C++**: “You can choose safety.” → Safe patterns exist, but unsafe ones remain valid.
- **Rust**: “Prove it’s safe.” → The compiler enforces correctness unless explicitly bypassed.

Core Trade-Off

- **C** maximizes expressiveness and minimizes restrictions.
- **Rust** maximizes correctness by restricting what is allowed in safe code.
- **C++** occupies the middle ground, capable of resembling either extreme depending on usage.

5. Relation to the Practical Examples (Part 2)

The code examples in Part 2 directly reflect the concepts discussed here:

- C macros vs. C++ templates vs. Rust generics → abstraction mechanisms
- Return codes vs. exceptions vs. Result → error handling strategies
- Manual allocation vs. RAII vs. ownership → memory and lifetime management
- Pointer arithmetic vs. bounds-checked access vs. borrowing rules → safety boundaries

These examples concretely demonstrate how each language’s design choices impact correctness and control in real systems code.