Course: Advanced Programming

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

ID: 718228

Name: Ramy Ghoubrial Abdelshahid Ghoubrial

Table of Contents

1. Introduction

Modern software systems require both **high performance** and **reliable correctness**, especially in areas such as operating systems, embedded systems, and performance-critical applications. These requirements place systems programming languages at the center of important design trade-offs.

C, C++, and Rust represent different approaches to systems programming.
C prioritizes low-level control and minimal abstraction, placing full responsibility on the programmer.
C++ adds powerful abstractions and safer programming patterns while preserving low-level access.
Rust enforces memory and concurrency safety at compile time, aiming to prevent common runtime errors without sacrificing performance.

This project analyzes how these languages handle **memory management, type systems, safety guarantees, and compilation models**, and how they balance **low-level control** with **language-enforced correctness**. The goal is not to rank the languages, but to clarify their design choices and appropriate use cases in systems programming.

2. Key Differences Between C, C++, and Rust in Systems Programming

This section explains how **C, C++, and Rust** approach systems programming, with focus on **memory management, type systems, safety guarantees, and compilation models**.
The goal is **not to rank the languages**, but to clarify what each language **allows, prevents, or enforces** at the systems level.

2.1 C programming language

Maximum Control, Minimal Built-in Safety

Memory Management

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

As a result, many incorrect programs compile successfully, with errors appearing only at runtime or silently corrupting memory. C offers maximum freedom, but correctness depends entirely on programmer discipline.

Type System

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

- No generics or templates
- Abstraction is commonly implemented using macros, void*, and coding conventions
- The compiler cannot prevent many forms of incorrect pointer usage or memory misuse

The type system provides limited compile-time guarantees and does not enforce safe abstractions.

Safety Guarantees

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

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

When Undefined Behavior occurs, the compiler is allowed to produce arbitrary results. Responsibility for correctness lies entirely with the programmer.

2.2 C++ programming language

Low-Level Power with Optional Safety

C++ extends C by introducing powerful abstraction mechanisms while preserving low-level control. The language provides tools that enable safer programming, but **safety is not enforced by default**.

Memory Management

C++ still allows manual memory management using new/delete and malloc/free. However, modern C++ strongly encourages safer patterns such as:

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

In well-written modern C++, memory management is **manual in capability but automatic in practice**, as resource lifetimes are tied to object scope.

Type System

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

- Templates for compile-time generic programming
- Function overloading
- Classes and object-oriented programming
- Static compile-time type checking

These features enable powerful abstractions without runtime overhead. However, unsafe constructs such as raw pointers, unchecked casts, and incorrect lifetime handling remain possible.

Safety Guarantees

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

- Undefined Behavior still exists (dangling references, invalid casts, data races, out-of-bounds access)
- Both safe and unsafe code can compile and run correctly

As a result, safety in C++ depends heavily on **coding style, discipline, and established conventions**.

2.3 Rust Programming Language

Strong Compile-Time Safety with Systems-Level Performance

Rust is designed to combine the performance characteristics of low-level systems languages with strong correctness guarantees enforced at compile time. Unlike C and C++, Rust makes safety a core language property rather than a convention.

Memory Management

Rust does not use garbage collection. Instead, it enforces memory safety through a strict ownership model:

- **Ownership**: Each value has a single owner responsible for freeing it.
- **Borrowing**: References can be immutable or mutable, but strict rules ensure no data races or dangling references.
- **Lifetimes**: Compile-time annotations and inference ensure references never outlive the data they point to.

Memory is released automatically when ownership goes out of scope, similar to RAII in C++, but enforced directly by the compiler rather than by convention. This eliminates entire classes of memory errors before the program can run.

Type System

Rust has a strong and expressive type system focused on correctness:

- Generics with trait bounds for abstraction
- Algebraic data types (enum) for expressive modeling
- Pattern matching with exhaustiveness checking
- Explicit handling of optional and error values (Option, Result)

The type system works closely with the borrow checker to ensure memory and concurrency safety at compile time, without introducing runtime overhead.

Safety Guarantees

Rust provides strong safety guarantees by default:

- No use-after-free
- No double free
- No data races in safe code
- No invalid memory access

Undefined Behavior is largely eliminated from safe Rust code. Unsafe operations are still possible through the unsafe keyword, but they are explicitly marked and isolated, making safety boundaries visible and auditable.

3- Trade-Offs Between Control and Language-Enforced Correctness

---

C, C++, and Rust represent different points along the spectrum between unrestricted low-level control and compiler-enforced correctness.

- **C** maximizes control and flexibility but provides no safety guarantees. Errors are easy to introduce and difficult to detect.
- **C++** offers powerful abstractions and safer patterns while preserving low-level access, but correctness depends heavily on developer discipline.
- **Rust** restricts certain patterns to enforce safety, reducing flexibility in exchange for strong compile-time guarantees.

These trade-offs reflect different design philosophies and make each language suitable for different problem domains within systems programming.

4- Relation to the Practical Examples

4.1 Abstraction Mechanisms: C vs C++ vs Rust

This example compares how abstraction is implemented in C, C++, and Rust using a simple "maximum" function.

C – Macro-Based Abstraction (Preprocessor Level)

In C, abstraction is commonly implemented using macros. The MAX(a, b) macro provides generic behavior through text substitution before compilation. This results in a zero-overhead abstraction, as no function calls or runtime checks are introduced.

However, this abstraction is unsafe. The macro performs no type checking and does not follow function semantics. As shown in the example, passing expressions with side effects (such as ++x) leads to double evaluation, producing incorrect results. This demonstrates a classic C pitfall where abstraction prioritizes simplicity and control over safety.

```c
#include <stdio.h>

// C macros: text substitution, no type checking
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main(void) {
    // Safe usage
    int m = MAX(3, 5);
    printf("max=%d\n", m);

    // Unsafe usage: side effects in arguments
    int x = 10;
    int y = 5;

    int result = MAX(++x, y);  // Expanded as: ((++x) > (y) ? (++x) : (y))

    printf("x=%d, result=%d\n", x, result); // x is incremented twice
    return 0;
}
```

C++ – Template-Based Abstraction (Type-Safe, Not Memory-Safe)

C++ replaces macros with templates, providing a type-safe abstraction mechanism. The max_t template is checked at compile time and follows normal function semantics, ensuring that arguments like ++x are evaluated exactly once.

Templates are implemented via monomorphization, where the compiler generates a specialized version of the function for each used type. This preserves zero runtime overhead while eliminating macro-related bugs.

However, the example also highlights that C++ does not enforce memory safety. The use of raw pointers allows dangling pointer errors (delete p followed by dereference), demonstrating that while abstraction is type-safe, memory correctness still depends on programmer discipline.

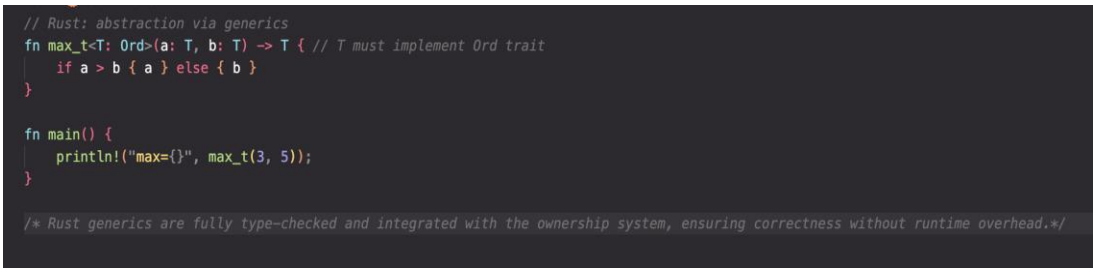
```cpp
#include <iostream>

// C++ abstraction: templates (type-checked at compile time)
template <typename T>
T max_t(T a, T b) {
    return (a > b) ? a : b;
}
int main() {
    // ✅ Safe abstraction: no macro expansion, no double evaluation
    int x = 10, y = 5;
    int r = max_t(++x, y);   // ++x evaluated once
    std::cout << "x=" << x << ", result=" << r << "\n";

    // ⚠️ Still allowed in C++: raw pointer misuse
    int* p = new int(5);
    delete p;        // memory freed
    *p = 10;         // ❌ dangling pointer (undefined behavior)

    return 0;
}
```

Rust – Generic Abstraction with Enforced Safety

Rust implements abstraction using generics combined with trait bounds. The max_t<T: Ord> function ensures that only comparable types are allowed, preventing logical errors at compile time.

Unlike C and C++, Rust's abstraction is fully integrated with its ownership and borrowing system. This guarantees both type safety and memory safety, preventing use-after-free and dangling pointer errors by construction.

Rust also uses monomorphization, generating specialized code at compile time. As a result, the abstraction provides zero runtime overhead while enforcing strong correctness guarantees.



```rust
// Rust: abstraction via generics
fn max_t<T: Ord>(a: T, b: T) -> T { // T must implement Ord trait
    if a > b { a } else { b }
}

fn main() {
    println!("max={}", max_t(3, 5));
}

/* Rust generics are fully type-checked and integrated with the ownership system, ensuring correctness without runtime overhead.*/
```

Key Takeaway

This example illustrates how abstraction evolves across the three languages:
C offers maximum freedom with minimal safety, **C++** provides type-safe abstractions without enforcing memory safety, and **Rust** enforces both logical and memory correctness at compile time without sacrificing performance.

4.2 Error Handling Mechanisms: C vs C++ vs Rust

This example compares how C, C++, and Rust represent and enforce error handling when validating a “positive input” condition.

C – Return Codes + Output Parameters (Manual Discipline)

In C, error handling is typically done with **status return codes**. The function parse_positive(x, out) returns 0/1 to indicate failure/success, while the computed value is written through the out pointer. This pattern is lightweight and predictable, with no runtime machinery.

However, correctness depends entirely on the caller. The compiler does **not** enforce checking the return value, and if the caller forgets to test it, the program may continue with an invalid state (silent error propagation). This shows that C’s model is efficient but relies heavily on programmer discipline.

```c
// C: error handling via return codes
#include <stdio.h>

int parse_positive(int x, int *out) {
    if (x <= 0) return 0;   // failure
    *out = x;
    return 1;              // success
}

int main() {
    int value = 0;
    if (!parse_positive(-3, &value)) {
        printf("Error: input must be positive\n");
        return 1;
    }
    printf("ok: %d\n", value);
    return 0;
}
/* The compiler does not enforce error handling. Forgetting to check return values can silently propagate errors. */
```

C++ – Exceptions (Separate Error Path, Potential Runtime Cost)

In C++, parse_positive(x) signals failure by **throwing an exception** (std::runtime_error). This separates the normal flow (“happy path”) from error handling using try/catch, improving readability since callers don’t need to check a status after every call.

The trade-off is that exceptions introduce a different control-flow model: when an exception is thrown, the runtime performs **stack unwinding** to find a matching handler and destructors run along the way. This can add runtime cost and complexity compared to plain return codes, and exception usage is sometimes avoided in low-level or performance-critical contexts depending on project constraints.

```cpp
// C++: error handling via exceptions
#include <iostream>
#include <stdexcept>

int parse_positive(int x) {
    if (x <= 0)
        throw std::runtime_error("input must be positive");
    return x;
}

int main() {
    try {
        int v = parse_positive(-3);
        std::cout << "ok: " << v << "\n";
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << "\n";
        return 1;
    }
    return 0;
}
/*Exceptions separate normal logic from error handling but may introduce runtime overhead and complexity.*/
```

Rust – Result<T, E> (Explicit, Compiler-Enforced Handling)

Rust uses the Result type to represent either success (Ok(value)) or failure (Err(error)), as shown in parse_positive(x) -> Result<i32, String>. The caller must explicitly handle both cases (e.g., via match), and the compiler prevents treating the result as a valid value without acknowledging the error branch.

This avoids silent failure propagation while keeping performance predictable: Result is a **zero-cost abstraction** in the sense that it is represented as a normal value (no stack unwinding mechanism like exceptions). Rust therefore combines C-like efficiency with language-enforced correctness.

```rust
// Rust: error handling via Result
fn parse_positive(x: i32) -> Result<i32, String> {
    if x <= 0 {
        Err("input must be positive".into())
    } else {
        Ok(x)
    }
}

fn main() {
    match parse_positive(-3) {
        Ok(v) => println!("ok: {}", v),
        Err(e) => {
            println!("Error: {}", e);
            std::process::exit(1);
        }
    }
}
/*  Errors must be explicitly handled.
The compiler prevents ignoring failure cases. */
```

Key Takeaway

C provides minimal-overhead error signaling but does not enforce handling; C++ uses exceptions to separate normal logic from failures with possible runtime/control-flow complexity; Rust encodes errors in the type system (Result), forcing explicit handling at compile time while remaining efficient.

4.3 Memory Management: C vs C++ vs Rust

This example compares how the three languages manage heap memory and how strongly they prevent lifetime-related bugs.

C – Fully Manual Heap Management (Maximum Control, No Enforcement)

In C, heap memory is explicitly allocated with malloc and must be explicitly released with free. The pointer p is just an address; the compiler does not track whether it is valid, owned, or already freed. This makes the model simple and efficient, but correctness is entirely the programmer's responsibility.

The example shows the key risk: after free(p), the pointer still exists and could still be dereferenced at the syntax level. Doing so would be **use-after-free**, which is **Undefined Behavior**. Similarly, double-free and memory leaks are possible if the programmer forgets or repeats deallocation.

```c
// C: manual memory management
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = malloc(sizeof *p);
    if (!p) return 1; // Must in C to handle allocation failure

    *p = 42;
    printf("value=%d\n", *p);

    free(p); // Manually free memory to prevent leaks

    // Using p after free would be undefined behavior
    return 0;
}

/* The programmer is fully responsible for allocation and deallocation.
Use-after-free errors are not detected by the compiler. */
```

C++ – RAII for Safe Patterns, but Unsafe Operations Still Allowed

C++ improves memory/resource handling through **RAII**: objects acquire resources in constructors and release them deterministically in destructors. In the example, Resource r; is

automatically cleaned up when leaving scope, and std::vector<int> manages dynamic memory safely without manual new/delete.

However, C++ still allows manual heap allocation (new) and explicit deallocation (delete). The second part demonstrates that raw pointer misuse remains possible: dereferencing p after delete creates a **dangling pointer** and leads to **Undefined Behavior**. So, C++ provides safer abstractions and patterns, but does not enforce memory safety by default.

```cpp
#include <iostream>
#include <vector>

struct Resource {
    Resource() { std::cout << "acquire resource\n"; } // constructor
    ~Resource() { std::cout << "release resource\n"; } // destructor
};

int main() {
    // ✅ RAII: automatic deterministic cleanup
    Resource r;

    std::vector<int> v;
    v.push_back(42);
    std::cout << "value=" << v[0] << "\n";

    // ⚠️ Still possible in C++: dangling pointer (use-after-free)
    int* p = new int(5);
    delete p;        // freed
    *p = 10;         // ❌ undefined behavior (dangling pointer)

    return 0;
}
```

Rust – Ownership and Drop (Compiler-Enforced Memory Safety)

Rust manages heap memory through **ownership**, enforced at compile time. In the example, Box::new(42) allocates on the heap, and the value is automatically freed when it goes out of scope via Rust's Drop mechanism (RAII-like behavior, but enforced by the language rules).

Crucially, safe Rust prevents the dangerous cases shown in C/C++: once a value is moved or dropped, it cannot be accessed again, making **use-after-free and double-free impossible in safe code**. The compiler enforces these lifetime and ownership rules statically, eliminating a large class of memory bugs without garbage collection.

```rust
// Rust: ownership-based memory management
fn main() {
    let x = Box::new(42);
    println!("value={}", x);
    // memory automatically freed at end of scope
}

/* Rust enforces memory safety at compile time.
Use-after-free and double-free are impossible in safe Rust. */
```

Key Takeaway

C provides direct manual control but no safety enforcement; C++ offers deterministic cleanup via RAII yet still permits unsafe raw-pointer patterns; Rust enforces correct lifetimes via ownership, giving deterministic cleanup with strong compile-time guarantees against common memory errors.

4.4 Pointer and Index Access: C vs C++ vs Rust

This example compares how the three languages allow access to contiguous memory and how strongly they prevent out-of-bounds and dangling-reference bugs.

C – Raw Pointers + Pointer Arithmetic (Maximum Flexibility, No Bounds Safety)

In C, arrays decay into pointers and pointer arithmetic is a core feature. In the code, int *p = arr; points to the first element, and p = p + 1; moves the pointer by sizeof(int) bytes to reach the next element. This model gives direct, predictable low-level control and maps closely to machine memory.

The trade-off is that C performs no bounds checking. The commented line p = p + 10 would move the pointer outside the valid array region, and any dereference after that becomes **Undefined Behavior**. This can lead to memory corruption, crashes, or security vulnerabilities (e.g., buffer overflows). Correctness is entirely the programmer's responsibility.

```c
// C: pointer arithmetic
#include <stdio.h>

int main() {
    int arr[3] = {10, 20, 30};
    int *p = arr;

    printf("%d\n", *p);   // 10
    p = p + 1;
    printf("%d\n", *p);   // 20

    // p = p + 10; // undefined behavior

    return 0;
}
/* C allows unrestricted pointer arithmetic with no runtime or compile-time safety checks. */
```

C++ – Safer Containers with Optional Low-Level Pointer Access

C++ supports raw pointers like C, but also provides safer abstractions. In the provided C++ code, std::vector<int> manages memory automatically, and v.at(1) performs **bounds checking** at runtime, preventing out-of-range access by throwing an exception.

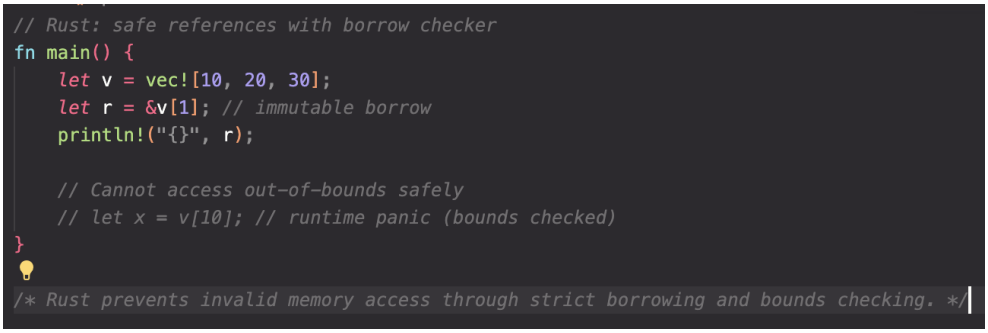
At the same time, the code shows that low-level pointer access is still possible (int* p = &v[0];). This is useful for performance or interoperability, but it reintroduces risks: if the vector reallocates (e.g., due to resizing), such pointers can become **dangling**, leading again to Undefined Behavior. The smart pointer (std::unique_ptr) portion demonstrates how C++ can enforce ownership for heap objects using RAII, but it does not eliminate pointer hazards unless developers consistently adopt these safer constructs.

```cpp
#include <iostream>
#include <vector>
#include <memory>

int main() {
    std::vector<int> v = {10, 20, 30};

    // Low-level pointer access is still possible
    int* p = &v[0];
    std::cout << *p << "\n";

    // Safer access using bounds-checked method
    std::cout << v.at(1) << "\n";

    /* ------------------------------
     * Fixing ownership with RAII
     * ------------------------------ */

    // Raw pointer (problematic ownership)
    // int* raw = new int(42);        // ❌ must remember to delete

    // Smart pointer (RAII)
    std::unique_ptr<int> safe = std::make_unique<int>(42);
    std::cout << *safe << "\n";      // automatically freed

    return 0;
}
// C++ provides safer abstractions like vectors with bounds checking and smart pointers for automatic memory management, reducing risks of
pointer misuse.
```

Rust – References + Bounds-Checked Indexing with Borrow Checker

Rust discourages raw pointer arithmetic in safe code and instead uses references that are verified by the borrow checker. In the example, &v[1] creates an immutable borrow, and Rust enforces strict aliasing rules (many immutable references or one mutable reference, but not both), preventing data races.

For bounds safety, indexing a vector performs a runtime bounds check. Attempting v[10] does not read arbitrary memory; it triggers a controlled panic instead of silently corrupting memory. This means Rust protects memory integrity by design: **out-of-bounds reads/writes are not allowed in safe Rust**, and the compiler enforces lifetime/aliasing rules that prevent common pointer misuse.



Key Takeaway

C exposes pointer arithmetic directly with no safety checks; C++ offers safer container-based access but still permits raw pointers and UB if misused; Rust enforces safe referencing and bounds-checked indexing in safe code, preventing invalid memory access while maintaining systems-level performance.

5- Compilation Models

C, C++, and Rust all rely on ahead-of-time (AOT) compilation, producing native machine code without the use of virtual machines or managed runtimes.
This shared compilation strategy is fundamental to systems programming, as it enables predictable performance, deterministic resource usage, and close interaction with hardware.

C and C++

Both C and C++ compile directly to machine code with minimal runtime support.
C offers the simplest compilation model, resulting in fast compilation times, small binaries, and transparent mapping between source code and generated instructions

C++ builds on this model by introducing zero-cost abstractions, where higher-level constructs such as templates and RAII are designed to be resolved entirely at compile time. While this preserves runtime performance, it increases compilation complexity, longer build times, and more difficult-to-interpret compiler diagnostics.

Rust

Rust is also compiled ahead-of-time using an LLVM-based backend, but its compilation model places significantly more emphasis on compile-time verification.
The borrow checker, lifetime analysis, and trait resolution introduce additional compile-time cost, but shift many error detections—such as memory misuse and data races—from runtime to compilation.
This design choice increases initial compilation time and development friction, but substantially improves runtime reliability and reduces debugging effort in production systems.

Compilation Model Trade-Off

- C and C++ favor faster compilation and simpler mental models, placing more responsibility on runtime testing and developer discipline.
- Rust deliberately invests more effort at compile time to enforce correctness, trading longer compilation and stricter constraints for safer and more predictable execution.
  This reflects a broader design philosophy difference: whether errors should be discovered late at runtime or early during compilation.

6- Conclusion

This study compared C, C++, and Rust as systems programming languages through both theoretical analysis and practical code examples. The comparison showed how each language balances low-level control, abstraction, and safety in fundamentally different ways. C provides maximum control and minimal abstraction, relying entirely on programmer discipline for correctness. C++ builds on this foundation by introducing powerful abstractions such as RAII, templates, and standard containers, improving safety without enforcing it. Rust takes a different approach by embedding memory and error safety directly into the language through ownership, borrowing, and strong type enforcement, preventing many classes of bugs at compile time. Rather than identifying a single “best” language, this project highlights that each language reflects a distinct design philosophy and is suitable for different systems programming contexts, depending on the required balance between control, performance, and correctness.