The Story of Building a Rust-Style Static Analyzer for C++ with AI

The project is available at: https://github.com/shuaimu/rusty-cpp

As someone who has spent almost 15 years doing systems research with C++, I am deeply troubled by all kinds of failures, especially segmentation faults and memory corruptions. Most of these are caused by memory issues: memory leaks, dangling pointers, use-after-free, and many others. I've had many cases where I have a pointer that ends with an odd number. The last one literally happened last month. It gave me so many sleepless nights. I remember a memory bug that I spent a month but still could not figure out, and I ended up wrapping every raw pointer I could find with shared_ptr.

So I always wished for some mechanical way that can help me eliminate all possible memory failures.

The Rust Dream (and the C++ Reality)

Rust is exactly what I need, and I'm happy to see this language mature. But unfortunately, many of my existing codebases that I deeply rely on are in C++. It's not a practical decision to just drop everything and rewrite everything from scratch in Rust.

One thing I used to hope for is better interop between C++ and Rust—something similar to C++ and the D language, or Swift's limited support of C++, where you can have seamless interop. You can write one more class and that class coexists with the existing C++ codebase. But after closely following discussions in the Rust committee, I do not think this is likely to happen soon.

Bringing Rust to C++

So I was very happy when I came across another option: bring similar memory safety features, the borrow checking and all, from Rust to C++. This is actually not a dead end, because in many ways C++ is a superset of Rust's features.

The first direction I was thinking: can we utilize C++'s overly powerful macro syntax to track the borrows? Imagine if we could achieve this without having to modify the compiler. After doing research for a while, I realized somebody had already tried this approach. Engineers at Google had already tried it, and this is an impossible solution. The impossibility lies in C++ details. You can read their analysis.

So it seems like what we have to do is provide a static analyzer for C++.

Circle C++: So Close, Yet So Far

There are actually efforts on this. The most mature one would be Circle C++ and later the Memory Safe C++ proposal. Circle C++ satisfies almost everything I dreamed of. It provides almost a Rust copy—the borrow check rules from Rust into C++.

But Circle also has its downsides that make it basically unusable to me. It relies on a closed-source compiler. I cannot replace my g++ with an experimental compiler. Additionally, it also has many intrusive features such as changes to C++ grammar, bringing in special syntax for borrow-checked references. The later development of Circle is also concerning. It was rejected by the C++ committee, and it seems like further development on this project has ceased.

Back to Square One: Just Write the Analyzer

So we're back to square one. Everybody tries to fix the language, but nobody tries to just analyze it. There are other efforts, like some say 2025 is the year of inventing alternative languages to C++, but that's not what I want. I want C++ to be safe. I don't have to leave my C++ world yet.

Then I thought: how hard is it to write this C++ static analyzer? Conceptually, I think it's not hard. It requires going through the AST. And since the static analysis is mostly statically scoped, it doesn't require heavy cross-file analysis. It can be single-file based, very limited scope. It should be engineerable, but the amount of engineering is expected to be huge, as it is basically cloning Rust's frontend. Having a full-time professor job, I don't have the time to do it.

I thought about hiring a PhD student to do it. But I had two problems: I don't have the funding, and it's very hard to find a PhD student for this. I don't blame them. I talked about this project to a few students, but they're not interested. Because it sounds like too much engineering and not enough paper-friendly novelty, you probably cannot even invent some cool concepts and put them in a publication, although it think it would be a very impactful paper we can manage to publish it.

So this idea sat for a while.

Enter AI Coding Assistants

Until this year, when AI coding assistants had really great development. I tried out Claude Code, and then I quickly upgraded. I was trying Claude Code for a few simple web dashboards, and then I wondered: how good can we test this with the idea of doing a Rust-style C++ static analyzer?

So I asked Claude. I gave this idea to Claude, and it quickly gave an answer that it's doable and gave me a plan that looked very reasonable to me. I asked it to come up with a prototype, then I asked it to write a few tests. Some tests passed, some failed, then it kept fixing the prototype. I kept asking for more tests. This iteration lasted for a while, to the point where it couldn't write tests that detect bugs anymore.

Then I started to try this tool in my other projects. I started with the rpc component in the Mako project. The refactoring process found more bugs, I fixed them, and this iteration continued.

Now I would say it is actually at a pretty stable, usable state.

Watching AI Evolve in Real-Time

Something else about the AI coding development: it's really evolving quickly. Initially I was using Sonnet 3.7, and it was giving me a lot of errors in a behavior very much like a first-year student. I had to manually re-run tests because it wasn't doing that. When I upgraded to Sonnet 4.5, it became less often that it gave phantom answers. But it still sometimes wasn't able to handle some complex problems. We'd go back and forth and I'd try to give it hints. But with Opus 4.5, that happens much less often now. I even started trying out the fully autonomous coding: instead of examining its every action, I just write a TODO list with many tasks, and ask it to finish the tasks one by one.

This is a very interesting experience. Six months ago, I would never have thought AI coding assistants could be this powerful.

It's amazing. But it actually worries me a little bit. Just in terms of this small project, it demonstrates more powerful engineering skills than most of my PhD students, and probably stronger than me. Looking at the code it wrote, I think it would take me about a few years full-time to reach this point, if I'm being optimistic, because I am not a compiler expert at all.

I can see in my first-hand experience that Claude keeps evolving. It's stronger and stronger, less likely to give me phantom results. If it keeps growing like this, I'm very concerned about the future shape of the systems engineering market. Maybe inevitably, someday we actually won't need hard-trained system hackers, just someone who's conceptually familiar with things and can sort of read the code.

I never had to fully understand the code. What I had to do is: I asked it to give me a plan of changes before implementation, it gave me a few options, and then I chose the option that seemed most reasonable to me. Remember, I'm not an expert on this. I think most of the time, anybody who has taken some undergraduate compiler class would probably make the right choice.

Interestingly, among the three options it usually gives me, Option A is usually not the correct option, usually Option B is. I was wondering if it's just trying to give me more options and the first option is always just a placeholder, like my students sometimes do.

Technical Design Choices

Let me talk about the technical design for a minute.

Comment-Based Syntax: To be compatible with any C++ compiler, we use comment-based annotations, and we don't introduce any new grammar to actual code. You have `@safe` to mark safe functions and `@unsafe` to mark unsafe functions. All unannotated code, including all existing code in your codebase and STL—is treated as `@unsafe` by default.

The rule is simple: `@safe` code can only call other `@safe` code directly. To call anything else (STL, external libraries, unannotated legacy code), you need to wrap it in an `@unsafe` block. This creates a clean audit boundary—code is either safe or it isn't.

This doesn't require any changes to existing code. If you have a legacy codebase and want to write one new function that's safe, it's totally fine—just mark it `@safe` and the analyzer will check it.

```
// Namespace-level: makes all functions in the namespace safe by default
// @safe
namespace myapp {

    void func1() {
        int value = 42;
        int& ref1 = value;
        int& ref2 = value;  // ERROR: multiple mutable borrows
    }

    // @unsafe
    void unsafe_func() {
        // Explicitly unsafe, no checking here
        int value = 42;
        int& ref1 = value;
        int& ref2 = value;  // OK - not checked
    }
}

// Function-level annotation
// @safe
void checked_func() {
    int value = 42;
    int& ref1 = value;

    // Need to call STL or external code? Use an unsafe block
    // @unsafe
    {
        std::vector<int> vec;  // OK in unsafe block
        vec.push_back(value);
    }
}
```

Const as Non-Mut: C++ has a perfect match for Rust's mutability: `const` and non- `const` . Const variables and const member functions are just non-mutable variables and functions. Non-const variables and functions are mutable. The only difference is that the default is reversed—we just need to put `const` in front of everything.

Borrow Checking: The core feature is Rust-style borrow checking. Multiple immutable borrows are fine, but you can't have multiple mutable borrows, or mix mutable and immutable borrows to the same variable:

```
// @safe
void borrow_rules() {
    int value = 42;

    // Multiple immutable borrows - OK
    const int& ref1 = value;
    const int& ref2 = value;
    int sum = ref1 + ref2;  // Fine

    // Multiple mutable borrows - ERROR
    int& mut1 = value;
    int& mut2 = value;  // ERROR: already mutably borrowed

    // Mixing mutable and immutable - ERROR
    const int& immut = value;
    int& mut = value;  // ERROR: already immutably borrowed
}
```

External Annotations: Something I had to do is support existing STL and third-party libraries. Those libraries are already there—Boost, STL, everything. What can we do? We can't modify system headers. So what we did is external annotations: we allow annotating existing functions as unsafe and giving their lifetime. This allows us to use any system headers without having to modify them.

```
// External annotations go in a header file
// @external: {
//   strlen: [safe, (const char* str) -> owned]
//   strcpy: [unsafe, (char* dest, const char* src) -> char*]
//   strchr: [safe, (const char* str, int c) -> const char* where str: 'a, return: 'a]
//
//   // Third-party libraries work the same way
//   sqlite3_column_text: [safe, (sqlite3_stmt* stmt, int col) -> const char* where stmt: 'a,
//   nlohmann::json::parse: [safe, (const string& s) -> owned json]
// }
```

The `where` clause specifies lifetime relationships—like `where stmt: 'a, return: 'a` means the returned pointer lives as long as the statement handle. This lets the analyzer catch dangling pointers from external APIs.

Libclang Challenges: I'm using libclang for parsing the AST, and it looks like it doesn't always give me simple names with proper qualifiers attached. This means if I have some name collision from different namespaces, it will likely mismatch the names to the proper entity in the analysis. I have to keep fixing them—basically keep asking Claude to fix it. This is a really painful process, but it looks like I'm getting to a stable state where I see much fewer bugs. That's a good thing.

Rust Standard Library Types: A lot of Rust idioms come from its standard library types such as `Box` , `Arc` , and `Option` . So I also wrote C++ equivalents to them. Although many say that `unique_ptr` is equivalent to `Box` , it actually isn't— `unique_ptr` allows null pointers, but `Box` doesn't. And similar for `Arc` vs `shared_ptr` . So I wrote C++ alternatives with exactly the same API as Rust.

```
#include "rusty/rusty.hpp"

// @safe
void rust_types_demo() {
    // Box - heap allocation, single owner, never null
    auto box1 = rusty::make_box<int>(42);
    auto box2 = std::move(box1);
    // *box1 = 100;  // ERROR: use after move

    // Arc - thread-safe reference counting
    auto arc1 = rusty::make_arc<int>(100);
    auto arc2 = arc1.clone();  // Explicit clone, ref count increases
    // Both can read: *arc1, *arc2

    // Vec - dynamic array with ownership
    rusty::Vec<int> vec;
    vec.push(10);
    vec.push(20);
    int last = vec.pop();  // Returns 20

    // Option - no more null pointer surprises
    rusty::Option<int> maybe = rusty::Some(42);
    if (maybe.is_some()) {
        int val = maybe.unwrap();
    }
    rusty::Option<int> nothing = rusty::None;

    // Result - explicit error handling
    auto divide = [](int a, int b) -> rusty::Result<int, const char*> {
        if (b == 0) return rusty::Result<int, const char*>::Err("div by zero");
        return rusty::Result<int, const char*>::Ok(a / b);
    };

    auto result = divide(10, 2);
    if (result.is_ok()) {
        int val = result.unwrap();  // 5
    }
}
```

Send and Sync: We also tried to create multi-threading safety by copying Rust's idea of marking types as `Send` and `Sync` . Only sendable types are allowed in thread spawning. Right now we're using C++ concepts to mark types as `Send` or `Sync` . The downside for now is we're doing manual marking—you have to mark types yourself. I haven't tried if I can deduce that automatically, but even if I can't, manually marking types as `Send` or `Sync` is fine for now.

Usage

Basic usage is straightforward:

```
$ rusty-cpp-checker myfile.cpp

Rusty C++ Checker
Analyzing: myfile.cpp
✗ Found 3 violation(s) in myfile.cpp:
Cannot create mutable reference to 'value': already mutably borrowed
Cannot create mutable reference to 'value': already immutably borrowed
Use after move: variable 'ptr' has been moved
```

For larger projects, you can use `compile_commands.json` (generated by CMake or other build systems):

```
# Generate compile_commands.json with CMake
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..

# Run the analyzer with it
$ rusty-cpp-checker src/main.cpp --compile-commands build/compile_commands.json
```

We also have CMake integration that supports automatic checking at compile time. You can check the Mako project as an example.

Conclusion

This project to me personally is a 15-year itch finally being scratched. Not by hiring a team of compiler engineers, not by waiting for the C++ committee to adopt memory safety, but just by having a conversation with an AI that turned my half-baked ideas into working code. This is unimaginable to me 6 months ago. I expected experiencing a few more iPhone-moments in my life, but I never thought it would be in this way-it shows a future where all my (programming) skills are probably not needed any more.

Anyway, check out the project. Try it on your codebase. And maybe, like me, you'll finally get some peace of mind about those mysterious segfaults.