**IST543 – Software Security**

**Final Project**

**c. based on the experience in b, would you like to use rust in your future development? why?**

Reasons for using Rust for future developments:

1. **Memory Safety:** The Rust code showcases various features that ensure memory safety, such as the use of safe abstractions like `*VecDeque*` instead of raw pointers, and the absence of manual memory management. For instance, in the `*build\_vc\_from\_parent*` function, Rust's ownership system ensures that memory is managed safely when creating a new `*VisitedCell*` struct, mitigating the risk of memory leaks or undefined behaviour. This project demonstrates how Rust's memory safety features contribute to writing secure and reliable code.

In the process of rewriting the C++ program to Rust, developers likely encountered and addressed memory-related issues such as buffer overflows and dangling pointers. Rust's ownership system ensures memory safety by enforcing strict rules at compile-time, preventing common memory errors that are prevalent in languages like C or C++ [1]. This enhanced memory safety reduces the risk of security vulnerabilities, making Rust a favourable choice for future development projects, especially in security-critical domains.

A computer screen shot of a code

Description automatically generated

This code from our project demonstrates how Rust's ownership system ensures memory safety by managing memory allocations and deallocations correctly. The use of *clone()* for the *ancestors* vector creates a deep copy, preventing unintended mutations to the original vector. This approach mitigates the risk of memory leaks or data corruption, showcasing Rust's memory safety guarantees in practice.

1. **Productivity and Efficiency:** The Rust code demonstrates how Rust's compiler helps improve productivity and efficiency by catching potential errors at compile-time. For example, when accessing elements in a 2D array, Rust's compiler ensures that the indices are within bounds, preventing common errors like buffer overflows. Additionally, Rust's pattern matching feature simplifies code readability and reduces the likelihood of logical errors. These are few examples for understanding our code highlights how Rust's compiler checks and language features contribute to faster development cycles and fewer runtime errors.

Rust's syntax and compiler checks enhance productivity and efficiency. Despite its learning curve, Rust's compiler ensures code correctness, reducing runtime errors and speeding up development [2]. This allows developers to focus more on feature development and less on debugging memory-related issues.

A screen shot of a computer program

Description automatically generated

This above piece of *Maze.rs* rust code showcases Rust's syntax and pattern matching capabilities, which enhance code readability and maintainability. The concise *match* statement improves code expressiveness by replacing nested *if-else* conditions, making it easier to understand and modify the code. Additionally, Rust's compiler ensures exhaustiveness checking, reducing the likelihood of logical errors and improving code quality, leading to greater productivity and efficiency.

1. **Community and Ecosystem:** While not explicitly shown in the Rust code from our project, the Rust community and ecosystem play a crucial role in supporting developers and providing valuable resources. Developers can leverage community-maintained libraries and tools, such as `*serde*` for serialization or `*hyper*` for building HTTP clients and servers, to accelerate development and ensure code quality. The practical availability of these libraries and tools demonstrates how Rust's ecosystem contributes to the success of future development projects.

Rewriting a program involves engaging with the Rust community and its ecosystem of libraries and tools. The growing community and rich ecosystem provide extensive support, documentation, and reusable components for efficient software development. Utilizing Rust's ecosystem can speed up future projects and provide reliable solutions for various use cases.





Although these statements in the *Maze* rust code from our project doesn't directly interact with community-maintained libraries, it illustrates Rust's standard library (*std*) and ecosystem support. Rust's standard library provides comprehensive functionality for common tasks, such as file I/O operations (*fs::read\_to\_string*). By leveraging Rust's standard library, developers can build reliable and efficient software without relying on third-party dependencies, showcasing the practical benefits of Rust's ecosystem.

1. **Future Development Considerations:** Our analysis on the Rust code from our project provided keen insights on secure code practices, developers can gain insights into how Rust's features and ecosystem align with the requirements of future development projects. The practical experience of working with Rust's memory safety features, productivity enhancements, and community support reinforces the decision to use Rust in security-sensitive domains. Developers can confidently adopt Rust knowing that it offers practical solutions to common challenges in software development, especially in terms of security and reliability.

The experience from rewriting the program with Rust, its memory safety, productivity gains, and community support suggest Rust's potential for future projects. Rust's focus on safety, performance, and concurrency is ideal for developing secure software, especially in security-sensitive areas. Adopting Rust can reduce memory-related risks and ensure high-standard, secure, and reliable software.

A screen shot of a computer code

Description automatically generated

This code from *Maze.rs* demonstrates how Rust's type system enables developers to design robust and maintainable data structures. By defining the *Maze* struct with explicit field types and sizes, Rust ensures type safety and memory layout predictability, reducing the risk of runtime errors and optimizing performance. This practical example highlights how Rust's features support future development considerations, such as code maintainability and scalability, making it a suitable choice for long-term software projects.

1. **Concurrency without Data Races:** Concurrency is a complex topic in many programming languages, and race conditions are a common issue when multiple threads access and modify shared data simultaneously [3]. Rust addresses this problem by preventing data races at compile time. It does this by enforcing strict rules about how and when data is accessed. This makes it easier to write correct concurrent code, reducing the risk of subtle, hard-to-detect bugs that can lead to serious problems in concurrent applications.

A screen shot of a computer program

Description automatically generated

This code from the *Maze.rs* demonstrates how Rust's ownership system and concurrency model help avoid race conditions. The use of mutable references (*&mut self*) ensures that only one thread can modify *self* at a time, preventing data races. Additionally, the *VecDeque* and *Vec* data structures are inherently thread-safe for single-threaded access, further reducing the risk of race conditions. By leveraging Rust's ownership and concurrency features, this code snippet ensures thread safety and avoids potential race conditions, making it suitable for concurrent execution in future development projects.

1. **Interoperability with C:** Rust’s ability to interoperate with C libraries is a significant advantage when working with existing codebases or libraries. Rust’s foreign function interface (FFI) allows it to call C functions directly from Rust code [4]. This means you can leverage existing C libraries without having to rewrite them in Rust. Additionally, this feature allows for a gradual migration of a C or C++ codebase to Rust, which can be beneficial in large, complex projects.
2. **Performance:** Rust is designed to provide control over low-level details such as memory layout, like C and C++. This means that Rust programs can be as fast as their C or C++ counterparts. Rust also allows fine-grained control over memory management, which can lead to more efficient programs. Rust achieves this through zero-cost abstractions, where higher-level constructs compile into as efficient a form as if they were written in lower-level code [5]. For example, Rust’s ownership model allows for automatic memory management without the need for a garbage collector, which can lead to performance improvements in certain scenarios.

**References**

1. Xu, H., Chen, Z., Sun, M., Zhou, Y., & Lyu, M. R. (2021). Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(1), 1-25.
2. Qin, B., Chen, Y., Liu, H., Zhang, H., Wen, Q., Song, L., & Zhang, Y. (2024). Understanding and Detecting Real-World Safety Issues in Rust. *IEEE Transactions on Software Engineering*.
3. Hu, S., Hua, B., Xia, L., & Wang, Y. (2022, December). Crust: Towards a unified cross-language program analysis framework for rust. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)* (pp. 970-981). IEEE.
4. Maiga, A., Artho, C., Gilcher, F., & Moy, Y. (2023, October). Does Rust SPARK Joy? Safe Bindings from Rust to SPARK, Applied to the BBQueue Library. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems* (pp. 37-47).
5. Holk, E., Pathirage, M., Chauhan, A., Lumsdaine, A., & Matsakis, N. D. (2013, May). GPU programming in rust: Implementing high-level abstractions in a systems-level language. In 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (pp. 315-324). IEEE.