[EXAMPLE TEMPLATE]

**WEEK 9**

**KU ID: 100058537**

**NAME: Mariam Alzaabi**

|  |
| --- |
| Q1. Rustlings exercise GitHub link: |
| ANSWER: |
| Q2.  A. Find and explain two traits used in C2RUST? (also mention the URLs)  B. Explain where those are used, and why?  C. Explain the alternatives to using traits? |
| ANSWER:  WEEK 9  KU ID: [Your KU ID here]  NAME: [Your Name here]  Q1. Rustlings exercise GitHub link:  ANSWER: The GitHub repository for the Rustlings exercises is located at: https://github.com/rust-lang/rustlings.  Q2.  A. Find and explain two traits used in C2RUST? (also mention the URLs)  ANSWER:  1. Clone Trait  - Explanation: The `Clone` trait enables a type to create a duplicate of itself through the `clone(&self) -> Self` method. It is designed for explicit copying, supporting both simple types and complex structures requiring heap allocation, such as strings or vectors. While not directly used in the Rustlings traits exercises, it is a fundamental trait in Rust.  - URL: https://doc.rust-lang.org/std/clone/trait.Clone.html  2. Debug Trait  - Explanation: The `Debug` trait allows a type to be formatted for debugging purposes using the `{:?}` specifier in print statements. It defines the `fmt(&self, &mut Formatter) -> Result` method, providing a readable representation of a value. Though not implemented in the Rustlings traits exercises, it is widely applicable for inspecting data.  - URL: https://doc.rust-lang.org/std/fmt/trait.Debug.html  B. Explain where those are used, and why?  ANSWER:  1. Clone Trait  - Usage: In C2RUST, which converts C code to Rust, the `Clone` trait could be applied when translating C structures that require duplication, such as arrays or linked lists. For example, in the Rustlings exercise `traits2.rs`, a `Vec<String>` is modified by appending "Bar". In a similar context, C2RUST might use `Clone` to duplicate data structures to ensure independent ownership.  - Purpose: Rust enforces strict ownership rules, unlike C’s pointer-based memory management. `Clone` provides a safe mechanism to replicate data, preventing multiple references to the same memory, which aligns with Rust’s safety guarantees.  2. Debug Trait  - Usage: In C2RUST, `Debug` might be implemented for Rust structs generated from C code to facilitate validation during transpilation. For instance, in `traits3.rs`, structs like `SomeSoftware` and `OtherSoftware` are defined with licensing information. Debug could be used to print these structs’ contents for comparison with their C originals.  - Purpose: During development, `Debug` enables developers to inspect the internal state of translated structures efficiently, ensuring accuracy without requiring custom output logic.  C. Explain the alternatives to using traits?  ANSWER:  1. Alternative to Clone:  - Description: Instead of using `Clone`, a custom duplication function can be defined. For example, in `traits1.rs`, where `append\_bar` modifies a `String`, a manual function could copy the string’s contents explicitly (e.g., `fn duplicate(s: &String) -> String { s.to\_string() }`) rather than invoking `clone()`.  - Rationale: This approach avoids trait dependencies, offering flexibility for specific copying needs. However, it requires manual implementation for each type, reducing reusability compared to `Clone`.  2. Alternative to Debug:  - Description: Rather than implementing `Debug`, a bespoke printing method can be created. In `traits3.rs`, for the `SomeSoftware` struct, a function like `fn print\_info(&self) { println!("Version: {}", self.version\_number); }` could be written to display its fields without relying on `Debug`.  - Rationale: This method provides precise control over output formatting and eliminates the need for a trait. However, it demands individual implementation for each type, making it less efficient than the standardized `Debug` trait for widespread use. |