GRQ02 - Programming Rust - Ch 4 & 5

Your Name

Chapter 4. Ownership

1. The first couple of pages of Chapter 4 make a detailed case for Rust's ownership model. In a paragraph of your own words, how would you describe to a friend in CS on the idea of ownwership in Rust?

Your answer here...

2. When is an owned value dropped in Rust? What does it mean for a value to be dropped?

Your answer here...

3. Graphs or cycles of objects are easily constructed in most programming languages, but not in Rust. What is the most typical structural relationship between values in Rust? Why?

Your answer here...

4. In figures 4-5 and 4-6 the reference count field of the `PyListObject` changes from 1 to 3. What is causing that change and what purpose does this field serve in Python?

Your answer here...

5. Besides initialization, what four operations cause a value's ownership to change in Rust? What is a change of ownership called?

Your answer here...

6. When rust is in a `for` loop over an iterable value, what happens to the ownership of the iterable?

Your answer here...

7. Why do primitive types like `u8`, `boolean`, and `char` not change ownership when assigned to another variable or passed as an argument?

Your answer here...

Chapter 5. References

8. When it comes to the relationship between a reference and a referent, what is an extremely important rule to know and follow? Why?

Your answer here...

9. What are Rust's two types of references? How do they differ from one another?

Your answer here...

10. There are two different `show` functions defined in the early pages of the chapter. The first has a parameter type of `Table` while the latter has a parameter type of `&Table`. The `for` loops inside each of the functions have the same code but behave differently. Why?

Your answer here...

11. To access a referent via a reference you typically must use the asterisk/dereference operator on it. In what scenario(s) will Rust implicitly dereference for you?

Your answer here...

12. Google's Java library named Guava is an open-source project that is popular among strong Java engineering teams. One of its first principles is "avoiding `null` *" where possible. Please read the brief introduction, the "Optional" section, and, "What's the point?" at the following URL: https://github.com/google/guava/wiki/UsingAndAvoidingNullExplained. How does Guava's approach to* `null` *compare to Rust's approach with respect to functions/methods that may return the absence of a value?*

Your answer here...

13. What is a lifetime*? Figures 5-3 and 5-4 display the lifetimes of* `&x` *and* `r`*, respectively, to illustrate a piece of code that fails Rust's borrow checker. What is it about these two lifetimes that leads to the error in the code?*

Your answer here...

14. The authors make an interesting statement that in Rust, "a function's signature always exposes the body's behavior." Compare the following two signatures in Java and Rust. Identify what the compiler guarantees *about the function's behavior in Rust that you* could not know with certainty *about the behavior of an equivalent method in Java by looking at its signature alone. Specifically address 1) whether or not* `sum` *can modify* `v`*'s referent's contents, and 2) whether or not* `sum` *can assign another reference to* `v`*'s referent whose lifetime outlives the scope of* `sum`*?*

```
// Rust
fn sum<'a>(v: &'a Vec<i32>) -> i32 { /* ... */ }

// Java
int sum(Vector<Integer> v) { /* ... */ }
```

Your answer here...

15. When are you not *required to specify the lifetimes of references in a function or struct's signature?*

Your answer here...

16. In the section Taking Arms Against a Sea of Objects*, the authors add commentary regarding the software architecture of applications written in memory managed languages versus those written in Rust. "It takes a bit of effort to make cycle in Rust." Provide an example of a data structure or object-oriented design pattern you have implemented in a memory managed language that used a cycle of references between objects.*

Your answer here...