GRQ03 - Programming Rust - Ch 6, 9, & 10

Your Name

Chapter 6. Expressions

1. Rust is an expression language. *Give an example different from the book's that illustrates assigning to a variable with an* `if` *expression.*

Your answer here...

2. The `match` *expression and* `if let` *expressions are related and interchangeable in some common scenarios. When would you choose* `match` *over the* `if-let` *and vice-versa?*

Your answer here...

3. Imagine a friend has a vector of integers and they'd like to use a `for` *loop to iterate through each element and* double *its value* without *using indices. Their first attempt is as follows. The* `rust` *compiler gives errors about mutability, moved values, and more. Correct the code below by adding* `mut`, `&mut`, *and the dereferencing operator in the necessary places such that when* `a` *is printed its values are* `[2, 4, 6]` *(*without *using indicies).*

```
let a = vec![1, 2, 3];
for x in a {
    x = x * 2;
}
println!("{:?}", a);
```

4. The turbofish *operator is unique to Rust. When is it needed? Now that you've written a tokenizer for* `thdc`*, explain* why *the* turbofish *operator is necessary in terms of tokenizing* operator *lexemes.*

Your answer here...

5. What is an lvalue *in Rust? Do other programming languages embrace the same name and concept of* lvalues*? (searching for the latter question is encouraged)*

Chapter 9. Structs

6. Given the `struct` *below, complete the example by initializing the variable* `person` *with your name and age.*

```
struct Person {
    name: String,
    age: u8
}
// later in the code
let you: Person;
```

7. How are methods defined so that they're associated with a `struct`*?*

Your answer here...

8. Three different ways to define a method in Rust include: 1) not having a `self` *parameter, 2) having an* `&self` *parameter, and 3) having an* `&mut self` *parameter. What are the implications of each?*

Your answer here...

9. What do you need to add to a `struct` in order for it to be displayed properly in a `println!` macro call such as `println!("{:?}", point)`'?

Your answer here...

Chapter 10. `enum` and `match`

10. How are `enum`s in Rust different from `enum`s in most other languages you've encountered? What is the "price" of using an `enum`?

Your answer here...

11. Given an `enum`, how can you tell how much memory is required to store any *value of that `enum`'s type?*

Your answer here...

12. What about the `enum Json` example demonstrates an `enum`'s ability to "quickly implement[..] tree-like data structures?"

Your answer here...

13. How does Rust's pattern matching work when matching against enums, structs, or tuples?

Your answer here...

14. In the figure 10-6, "Pattern matching with structs", the matching pattern is `Point { x: x, y: y }`. It is shown in the context of a `match balloon.location {...}` on the previous page. Explain the different meanings of the two `x`'s in this pattern? What about this example also allows you to specify the same pattern as `Point {x, y}`?

Your answer here...

15. Consider the `impl` for `BinaryTree` in the section on "Populating a Binary Tree". Notice `self` is being reassigned to a different *variant of the `BinaryTree` enum! Why do you think this is possible with an `enum` in Rust? Why is it* not *possible to reassign `this` in Java to another object sharing the same superclass? (Hint: consider their representations in memory and what the concrete* type *`this` is in Java versus `&mut self` must be in a Rust `enum`'s method.)*

Your answer here...

This isn't a question, but a note to call out the last section of this chapter as one that is worth reading twice. This sentence is especially worth lingering on: "For cases when a value may be one thing, or another thing, or perhaps nothing at all, enums are better than class hierarchies on every axis: faster, safer, less code, easier to document." The following paragraph about the limitations in flexibility is also very important to think deeply about.