Exercises

Exercise 1: Variables and Mutability

Goal: Practice declaring immutable and mutable variables, and understand their scope.

Instructions:

1. Declare an immutable variable x with the value 10.
2. Attempt to reassign x to 20 and observe the compiler error.
3. Declare a mutable variable y with the value 15.
4. Reassign y to 25 and print its value.

Hints:

- Use let for immutable variables.
- Use let mut for mutable variables.

Expected Outcome:

- Students should see a compiler error when trying to reassign x.
- Students should successfully reassign and print y.

Exercise 2: Ownership and Borrowing

Goal: Understand ownership transfer and borrowing.

Instructions:

1. Create a String with the value "hello" and assign it to s1.
2. Transfer ownership of s1 to s2 and print s2.
3. Attempt to print s1 after transferring ownership and observe the compiler error.
4. Create a function print_string that takes an immutable reference to a String and prints it.
5. Call print_string with s2.

Hints:

- Use String::from to create a String.
- Use & to create an immutable reference.

Expected Outcome:

- Students should see a compiler error when trying to print s1 after transferring ownership.
- Students should successfully print s2 and pass it to print_string.

Exercise 3: Data Processing Pipeline

Goal: Implement a data processing pipeline that filters, transforms, and aggregates data using Rust’s core concepts.

Scenario:

You are given a dataset of temperature readings (in Celsius) from multiple sensors over a 24-hour period. Your task is to:

1. Filter out invalid readings (values below -50°C or above 60°C).
2. Convert valid readings from Celsius to Fahrenheit.
3. Calculate the average temperature for each sensor.
4. Identify the sensor with the highest average temperature.

Data:

Use the following dataset (simulated as a vector of tuples):

```
let sensor_data: Vec<(&str, Vec<f64>)> = vec![
    (“sensor_1”, vec![22.5, 23.0, 22.8, -60.0, 23.3]),
    (“sensor_2”, vec![18.0, 19.5, 18.7, 20.0, 19.2]),
    (“sensor_3”, vec![25.0, 24.8, 25.2, 25.1, 24.9]),
];
```

Instructions:

1. Define a function filter_invalid_readings that takes a vector of f64 and returns a new vector with only valid readings.
2. Define a function �ahrenh_to_fahrenheit that converts a Celsius value to Fahrenheit.
3. Define a function calculate_average that calculates the average of a vector of f64.
4. Define a function process_sensor_data that:
   - Filters invalid readings for each sensor.
   - Converts valid readings to Fahrenheit.
     1. F = (C × 9/5) + 32
   - Calculates the average temperature for each sensor.
   - Returns the sensor name with the highest average temperature.
5. Print the results for each sensor and the sensor with the highest average temperature.

Hints:

- Use filter and map to process the data.
- Use a match expression or if statements to filter invalid readings.
- Use a loop or iterator to calculate the average for each sensor.
- Use a struct or tuple to store intermediate results (e.g., sensor name and average temperature).

Expected Outcome:

- Students should print the average temperature for each sensor in Fahrenheit.
- Students should identify and print the sensor with the highest average temperature.

Example Output:

```
Sensor sensor_1: Average temperature = 74.14°F
Sensor sensor_2: Average temperature = 66.74°F
Sensor sensor_3: Average temperature = 76.22°F
Sensor with the highest average temperature: sensor_3
```

Exercise 4: Order Fulfillment System

Goal: Simulate an order fulfillment system for a small store using vectors and basic control flow.

Scenario:

You are tasked with managing product inventory and processing customer orders. The system should:

1. Track the quantity of each product in stock (using a vector of tuples: (&str, u32)).
2. Process orders by reducing the stock quantity if available.
3. Handle cases where the ordered quantity exceeds the available stock.
4. Generate a report of products that need restocking (quantity < 3).

Data:

Use the following initial inventory:

```rust
let mut inventory: Vec<(&str, u32)> = vec![
    ("apple", 10),
    ("banana", 5),
    ("orange", 8),
    ("pear", 2),
];
```

Instructions:

1. Define a function process_order that takes a product name, ordered quantity, and inventory (as &mut Vec<(&str, u32)>):
    - If the product exists and the ordered quantity is available, reduce the stock.
    - If the product does not exist or the quantity is insufficient, print an error message.
2. Define a function generate_restock_report that returns a vector of product names with quantities below 3.
3. Process the following orders:
    - Order 4 apples.
    - Order 6 bananas.
    - Order 3 oranges.
    - Order 2 pears.
4. Print the updated inventory and the restock report.

Hints:

- Use a loop to iterate over the inventory and find the product by name.
- Use pattern matching or conditionals to handle stock updates and errors.
- Use a mutable vector to update inventory quantities.
- Use a loop to generate the restock report.

Expected Outcome:

Students should print the following (example output):

```
Order for 6 bananas cannot be fulfilled. Insufficient stock.
Updated Inventory:
apple: 6
banana: 5
orange: 5
pear: 0
Restock Report: ["pear", "banana"]
```