Student Name: ______________________________ Student ID: ________________ TA: _____________________

Elelementary System Programming

16/10/2024

Rules for Programming Examinations

1. **No Communication:**
   - **Do not talk to or interact with other examinees.**
   - **No looking at others' screens or work.**
2. **Device Restrictions:**
   - **Use only the computer provided for the exam.**
   - **All other electronic devices (notebook phones, tablets, smartwatches, etc.) are strictly prohibited.**

Exam 1: (10 points) Create a simple program that spawns multiple threads to perform calculations.

Create a program that spawns multiple threads to perform calculations and collects their results in the main thread.

Requirements:

1) Create a vector to store thread handles
2) Spawn 5 threads, each calculating the square of its index (0-4)
3) Each thread should **return** its calculated result
4) Store each thread handle in the vector
5) In the main thread, collect all results by joining the threads
6) Sum all the collected results
7) Print each individual result and the final sum

Example Output:

```
Thread 0 calculated: 0² = 0
Thread 1 calculated: 1² = 1
Thread 2 calculated: 2² = 4
Thread 3 calculated: 3² = 9
Thread 4 calculated: 4² = 16
Sum of all results: 30
All threads completed!
```

TA Scoring: ___ / 10

___ (1 pt) Creates a Vec<JoinHandle<T>> to store thread handles with appropriate type
___ (2 pts) Correctly spawns 5 threads using thread::spawn with proper closure syntax
___ (1 pt) Each thread calculates the square of its index (i * i)
___ (2 pts) Each thread returns its calculated result (not just printing)
___ (1 pt) Stores all thread handles in the vector (using .push() or collect)
___ (2 pts) Properly joins all threads and collects results using .join().unwrap()
___ (1 pt) Correctly sums all collected results (expected sum: 30)

Exam 2: CLI Word and Line Counter

Create a command-line interface (CLI) program in Rust that counts words and/or lines in a text file based on the provided flag.

Requirements:

1) The program should accept two command-line arguments:
   a) A required file path to a text file
   b) An optional flag: -w for word count, -l for line count, or no flag for both
2) The program should output:
   a) Word count if -w flag is used
   b) Line count if -l flag is used

    c) Both word and line count if no flag is provided
3) Implement basic error handling for:
    a) File not found
    b) Invalid arguments (e.g., incorrect flag, missing file path)
4) Define a word as any sequence of characters separated by whitespace.
5) Empty lines should be included in the line count.

Example usage:

```
$ ./word_line_counter file.txt
Words: 150
Lines: 20

$ ./word_line_counter file.txt -w
Words: 150

$ ./word_line_counter file.txt -l
Lines: 20

$ ./word_line_counter
Error: No file path provided.

$ ./word_line_counter nonexistent.txt
Error: File not found: nonexistent.txt

$ ./word_line_counter file.txt -x
Error: Invalid flag. Use -w for word count, -l for line count, or no flag for both.
```

TA Score: _____ / 10

___ (2 pts) Correctly parses command-line arguments using std::env::args() (file path and optional flag)

___ (1 pt) Successfully reads file content using std::fs::read_to_string() or similar

___ (1 pt) Implements correct word counting logic using split_whitespace() or equivalent

___ (1 pt) Implements correct line counting logic using .lines() or equivalent (includes empty lines)

___ (2 pts) Correctly handles flag logic: -w (words only), -l (lines only), no flag (both)

___ (2 pts) Implements error handling for file not found with appropriate error message

___ (1 pt) Implements error handling for invalid arguments (wrong flag, missing file path)

Exam 3: Traits - Vehicle System

Create a Rust program that demonstrates the use of traits to represent different types of vehicles with multiple characteristics.
You are asked to create a system to represent the characteristics of three vehicles: an Electric Car, a Mountain Bike, and a Cargo Drone. These vehicles use different power sources, are designed for different terrains, and have different cargo capacities. You will define multiple traits to describe these characteristics and implement them accordingly.

Requirements:

1) Define a trait named `PowerSource` with:
    a) A method `fuel()` that returns a string indicating what powers the vehicle
    b) A method `max_range(&self)` that returns an `i32` representing maximum travel range in kilometers
2) Define a trait named `Terrain` with a method `surface()` that returns a string indicating where the vehicle is best suited to travel.
3) Define a trait named CargoCapacity with:
    a) A method `max_load(&self)` that returns an i32 representing maximum cargo weight in kilograms
    b) A default implementation of `can_carry(&self, weight: i32) -> bool` that returns true if the weight is less than or equal to `max_load()`
4) Create three struct types with appropriate fields:
    a) `ElectricCar` with field `battery_kwh: i32`
    b) `MountainBike` (no fields needed)
    c) `CargoDrone` with field `payload_kg: i32`
5) Implement the required traits for each vehicle:
    a) ElectricCar:

        i) Powered by "rechargeable battery"
        ii) Max range calculated as `battery_kwh * 5 kilometers`
        iii) Best suited for "paved roads and highways"
        iv) Max load: 400 kg
    b) MountainBike:
        i) Powered by "human pedaling"
        ii) Max range: 50 kilometers
        iii) Best suited for "rough trails and off-road paths"
        iv) Does NOT implement CargoCapacity trait
    c) CargoDrone:
        i) Powered by "lithium battery"
        ii) Max range: 30 kilometers
        iii) Best suited for "air routes and urban areas"
        iv) Max load: based on the `payload_kg` field value

6) Write a generic function `describe_vehicle<T: PowerSource + Terrain>(vehicle: &T)` that takes any vehicle implementing both traits and prints its power source, max range, and terrain.
7) Write a generic function `check_cargo<T: CargoCapacity>(vehicle: &T, weight: i32)` that prints whether the vehicle can carry the specified weight.
8) In the main function:
    a) Create an instance of each vehicle (use battery_kwh: 75 for ElectricCar, payload_kg: 25 for CargoDrone)
    b) Call `describe_vehicle()` for all three vehicles
    c) Call `check_cargo()` for ElectricCar with 350 kg and CargoDrone with 30 kg

Example Output:

```
Electric Car:
Power source: rechargeable battery
Max range: 375 km
Best terrain: paved roads and highways

Mountain Bike:
Power source: human pedaling
Max range: 50 km
Best terrain: rough trails and off-road paths

Cargo Drone:
Power source: lithium battery
Max range: 30 km
Best terrain: air routes and urban areas

Cargo Check:
Electric Car can carry 350 kg: Yes
Cargo Drone can carry 30 kg: No
```

TA Score: _____ / 10

___ (1 pt) Defines PowerSource trait with both fuel() and max_range(&self) methods
___ (1 pt) Defines Terrain trait with surface() method
___ (1 pt) Defines CargoCapacity trait with max_load(&self) and default can_carry(&self, weight: i32) implementation
___ (1 pt) Creates all three structs (ElectricCar with battery_kwh, MountainBike, CargoDrone with payload_kg)
___ (2 pts) Correctly implements PowerSource trait for all three vehicles with proper logic (especially ElectricCar range calculation)
___ (1 pt) Correctly implements Terrain trait for all three vehicles
___ (1 pt) Correctly implements CargoCapacity trait for ElectricCar and CargoDrone only (not MountainBike)
___ (1 pt) Implements describe_vehicle generic function with correct trait bounds and output
___ (1 pt) Implements check_cargo generic function with correct trait bound and logic

---

Total Score: _______ / 30