Rust Programming Fundamentals Labs

Module 1: Introduction to Rust

Lab 1.1: Installing Rust

1. Go to the Rust website (https://www.rust-lang.org/).
2. Click the **Install** link and follow the instructions to install Rust on your system.
3. Open a new command window and type **`cargo --version`** to verify Rust is installed successfully.

Lab 1.2: Creating a Simple Application

1. Open a command window and navigate to a folder of your choosing.
2. Create a new Rust binary project with **`cargo new hello`**.
3. Change to the hello directory. List the files, notice the **cargo.toml** file and the **src** directory.
4. Build the application by entering **`cargo build`**.
5. Run the application by typing **`cargo run`**.
6. Run the application directly by navigating to the output directory (under **target**).
7. Build a release version of the application and run it.

Lab 1.3: Using Visual Studio Code

1. Install VS Code if you haven't already.
2. Install the Rust and Rust (rls) extensions.
3. Open the folder from lab 1.2. Note that the folder you want is where **cargo.toml** is located, and **not** the **src** folder.
4. Use the command palette (Ctrl+Shift+P) and type **cargo** and look for **build**.
5. Set a breakpoint on the **println** call and press F5 to start debugging. In the current extensions, the project should be automatically recognized and configured for debugging (a message box appears to confirm).
6. If not, you may need to configure **launch.json** to something like this:

```
    "configurations": [
        {
            "name": "(Windows) Launch",
            "type": "cppvsdbg",
            "request": "launch",
            "program": "${workspaceFolder}/target/debug/hello.exe",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
```

```
        }
    ]
```

7. If you managed to debug, you're good to go!

Module 2: Rust Fundamentals

Lab 2.1: Fahrenheit to Celsius Converter

1. Create project named **temp_converter**.
2. Get input from the user for a temperature in Fahrenheit to be converted. Use `std::io::stdin().read_line()`.
3. Convert the result to Celsius and display. (C = 5*(F – 32) / 9)

Lab 2.2: Binary Display

1. Create a new project named **binary_display**.
2. Input an integer number from the user and display it in binary.
3. Calculate the number of "1"s in the number using the AND (&) operator and the right shift operator (>>). Display the result.
4. Alternatively, you can use **count_ones** instance method for the same purpose.

Lab 2.3: Dollar Stairs

1. Create a new project named **dollar_stairs**.
2. Input a number n from the user and display n stairs of $ signs, like so (n=5 in this example):

```
$
$$
$$$
$$$$
$$$$$
```

Lab 2.4: Guessing Game

1. Create a new project named **guess_game**.
2. The computer should select a secret number in the range 1-100 like so:
    a. Add a dependency on the **rand** crate (search in https://crates.io)
    b. Use the example code from the slides
3. The user should try to guess the number. The program should respond with "too big" or "too small".
4. When the user finally guesses correctly, display the number of guesses she took.

Lab 2.5: Importing a Crate

1. Create a new project named **complex_calc**.
2. The project should use complex numbers, available in a crate called **num_complex**. Locate it in crates.io and add it as a dependency.

3. In main, get two values from the user, being the real and imaginary part of a complex number.
4. Create a complex number (**Complex::new** associated method) from the user's input.
5. Display the magnitude of the complex number and its angle (search the crate's docs for the required functions).

Module 3: Ownership

Lab 3.1: Primes

1. Create a new project named **primes**.
2. Create a function called **is_prime** that returns true if a number is prime.
3. Create a function named **calc_primes** that accepts two numbers and returns a **Vec<u32>** with the prime numbers in that range.
4. Create a function named **add_primes** that accepts two numbers and an existing **Vec<u32>** and appends to it all the prime numbers in the range.
5. Create a function called **count_primes** that accepts a number slice and returns the count of primes in that range.
6. Write code in the main function to test all the above functions.
7. Optional: write tests for these functions.

Module 4: Compound Types

Lab 4.1: Simple Struct

1. Create a new project named **shapes**.
2. Create a struct that represents a rectangle with width and height.
3. Create a function named **calc_area** that accepts a rectangle and returns its area.
4. Do the same for calculating a rectangle's circumference.
5. Create some rectangles in the **main** function and call the other functions to test your code.

Lab 4.2: Methods and Associated functions

1. Continue from the previous exercise.
2. Turn the area and circumference calculations into methods.
3. Add an associated **new** function that allows easy construction of a rectangle based on its width and height.
4. Replace the code in **main** to use these new constructs.
5. Build and test your code.

Lab 4.3: Enums

1. Create a new project named **turtle_graphics**.
2. Create a struct called **Turtle** that represents an entity that can move around (this is from the LOGO language days in the 80s 😊).

3. The turtle has a position and a heading.
4. Create an enum called **TurtleCommand** that has the following variants:
    a. Rotate with an angle
    b. Move forward a specified number of units
    c. Move backwards a specified number of units
    d. Rotate right (90 degrees from current heading)
    e. Rotate left (90 degrees from current heading)
5. Create an **execute_command** method for the turtle type that executes the command given. Use pattern matching.
6. Add an associated function to **turtle** to create one positioned at (0,0) and heading 0 degrees.
7. In the main function:
    a. Create a turtle
    b. Issue several commands.
    c. Print the turtle's position and orientation after each command.
8. Optional: allow the user to input commands and execute appropriately.
9. Optional: Add tests to ensure the functionality is correct.

(*) Lab 4.4: The Builder Pattern

1. Continue from the lab 4.3. Implement a **TurtleBuilder** struct to allow building a **Turtle** object like so:
   let mut t = TurtleBuilder::new().with_position(3.0, 4.0).with_heading(45.0).build();

Module 5: Common Collections

Lab 5.1: Command line arguments

1. Create a new project named **quad**.
2. Use the command line arguments (**std::env::args**), that are supposed to be the 3 coefficients in the quadratic equation **a*x*x+b*x+c=0**
3. Solve the equation (if possible), notifying the user of any errors.

Lab 5.2: Word histogram

1. Create a new project named **word_hist**.
2. Accept a file name from the command line.
3. Read the file contents into a string (**std::fs::read_to_string**).
4. Count the number of occurrences of words in the file (split words by whitespace with **String::split_whitespace**). Use a **HasMap<>** for counting occurrences.
5. Display the results.
6. (*) Sort by the occurrence number and display the result again.

Lab 5.3: Game of Life

1. Create a new application named **gameoflife**.
2. The program should simulate the well-known John Conway's Game of Life.
3. Input a number from the user, indicating board size (in the range of 7 to 50).

4. Create a two-dimensional square array with the given size. Use an appropriate data structure. Each call can be empty or filled.
5. Fill the grid of cells with random values (empty or filled).
6. Loop over the following actions:
    a. Show the grid to the user (display '.' For empty cells and 'X' for filled ones).
    b. The user then presses any key.
    c. Calculate the next generation of cells based on the following rules:
        i. Calculate the number of living cells around a given cell (8 neighbors).
        ii. If the number is 0 or 1 – the cell dies in the next generation (loneliness).
        iii. If the number is 2 – the cell state remains as it was.
        iv. If the number is 3 – a new cell is born.
        v. If the number is 4 or higher – the cell dies (overpopulation)
    d. Repeat until the user presses 'q'.
    e. Optionally, when showing the grid, show it over the previous grid.
7. As an alternative for waiting for the user to press a key, wait one second between loop iterations. Use the **std::thread::sleep** method.
8. Optional: find a graphics crate and use it for the display.

Module 6: Managing Projects

Lab 6.1: Modules

1. Go back to lab 4.3 (or create a new project), and make changes so that the **Turtle** type is placed in a library module and the **main** function uses that module.

Module 7: Error Handling

Lab 7.1: Basic Error Handling

1. Make changes to lab 2.4 so that if the user provides bad input, the process doesn't panic and instead displays an appropriate error and lets the user try again.
2. Modify lab 5.2 so that **main** returns a **Result** type, making the necessary code changes.

Module 8: Generics and Traits

Lab 8.1: Generics

1. Create a new library project named **generics**.
2. Create a generic struct named **Stack<>** that represents a stack of any arbitrary type.
3. Add associated function(s) for creating an instance.
4. Add the following methods: **push**, **pop**, **is_empty**, **len**.
5. Implement as needed.
6. Create test(s) for your implementation.
7. Add a **Queue<T>** class in a similar way.

Lab 8.2: Implementing Traits

1. Create a new library project named **rationals**.
2. Create a struct called **Rational** that represents a rational number (numerator and denominator).
3. Decorate the type with the traits: **Copy**, **Clone** and **Debug**.
4. Implement the following traits: **Display**, **PartialEq**, **PartialOrd**, **std::ops::Add**, **std::ops::Sub**, **std::ops::Mul**.
5. Implement any other methods you see fit.
6. Write code to test your implementation.
7. (*) Add a **compact** method that reduces the rational to its simplest form.

Lab 8.3. Polymorphism

1. Create a new project named **drawing**.
2. Define a trait named **Shape** that declares the following:
    a. A function called **area** that should return the shape's area.
    b. A function called **bounds** that returns the 2D bounds of the shape.
3. Create a few structs implementing the **Shape** trait, such as: **Rectangle**, **Ellipse**, **Circle**.
4. Create a **Vec<>** holding on to various shapes polymorphically.
5. Add several shapes and test calls to **area** and **bounds**.

Module 9: Smart Pointers

Lab 9.1: Polymorphism (take 2)

1. Augment lab 8.4 to use **Box<>** to store shapes in a **Vec<>**, so that that shape objects' lifetime is not constrained by the lifetime of any Vec<>.

Lab 9.2: (**) Generic Tree

1. Create a library project named **generic_tree**.
2. Implement a generic **Tree<T>** struct, that holds reference counted objects to its items. Each node holds its children, and a child points back to its parent.
3. Some Guidance
    a. Use **RefCell** for interior mutability.
    b. Create a **Tree<T>** struct and a **TreeNode<T>** struct. Tree<T> should only hold a root **TreeNode<T>** (combine as needed with **RefCell<>** and **Rc<>**)
    c. Implement the following functions/methods on the Tree struct:
        i. Associated function, **new(T)** builds a new Tree with a root item holding the data provided.
        ii. **add_child** method to add a child to a given node.
        iii. **add_sibling** method, to add a sibling node to a given node
        iv. **root** method, returning the root node.

4. Add other functions/methods as you see fit.
5. Test your implementation.

Module 10: Functional Programming

Lab 10.1: Using Iterators

1. Create a new project named **sorter**.
2. Read the contents of a file (from the command line) into a string.
3. Split the string into words.
4. Sort the words in ascending Unicode order (and display results). (use **collect** first)
5. Sort the words in descending order of length (display results).
6. Display the words that are at least 4 bytes in length.
7. Display the sum of all words' lengths (use **fold**).
8. Transform every word into its length (use **map**).

Lab 10.2: Implementing an Iterator

1. Create a new project named **primes**.
2. Write a function named **calc_primes**, that accepts two numbers (**from** and **to**) and returns an iterator that returns the next prime in the range between from and to.
3. Implement a struct to serve as the iterator.
4. Test your implementation.

Module 11: Concurrency

Lab 11.1: The Mandelbrot Set

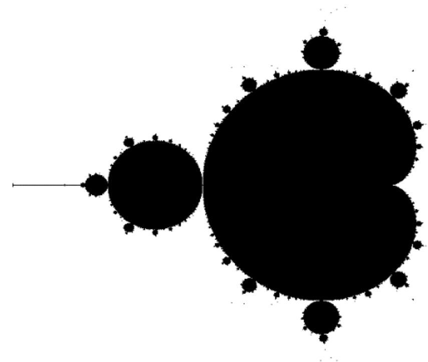
1. Create a project named **mandelbrot**.
2. Add dependencies for the crates **num-complex** and **png**.
3. Create a function named **build_mandelbrot** that accepts two **Complex** objects that form the range over which to build the Mandelbrot Set, and dimensions in pixels.
4. Implement the function by iterating over each pixel and calculating a pixel value of black (0) or white (255). Use the following function to get the final pixel color:

```
fn mandelbrot_color(c : &Complex) -> u8 {
    const ITERATIONS : u32 = 1000;
    let mut z = Complex::new(0.0, 0.0);

    for _ in 0..ITERATIONS {
        z = z * z + c;
        if z.norm_sqr() > 4.0 {
            break;
        }
    }
```

```
    if z.norm_sqr() > 4.0 { 0xff } else { 0 }
}
```

5. Save the resulting pixels to a PNG file with the help of the **png** crate (look at an example in its docs).
6. Use the following coordinates for a nice Mandelbrot set production: (-1.75,-1.0)-(0.75, 1.0).
7. Here is the image you should produce:



8. Create a function named **build_mandelbrot_mt** that uses threads to fork/join the work by giving each thread a set of rows to work on.
9. Compare execution times with different thread counts (input from the command line).

Lab 11.2: Thread Safe Queue and Stack

1. Create a library that implements a thread-safe queue and a thread-safe stack.
2. Test your implementation.