Assignment 3: "Mandelbrot"

Develop a program in Rust that generates the Mandelbrot set image and save is to disk in a "PPM" format.

The resulting image (see Figure 1) should display the Mandelbrot set in for the given resolution, where pixels within the set are represented in black (rgb: 0, 0, 0) and those outside in white (rgb: 255, 255, 255).

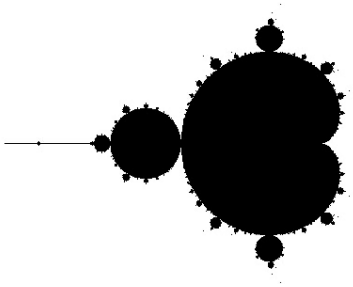

Figure 1

A nice and interactive description of how Mandelbrot set is generated can be found on this link: https://complex-analysis.com/content/mandelbrot_set.html. The algorithm description is also described later in the text in this document.

Requirements summary:

- Use **Rust** to develop an application that generates the Mandelbrot set.
- Save the generated image to disk in "PPM" format.
- The resulting image should display the Mandelbrot set in a 525x300 resolution, which is provided to the program via input arguments.
- Implement the required features according to the specification below.

Color Representation:

- Pixels belonging to the Mandelbrot set should be represented in black (rgb: 0, 0, 0).
- Pixels not part of the Mandelbrot set should be displayed in white (rgb: 255, 255, 255).

Implementation Details:

- Your application should accept input parameters for the image size (**width** and **height**) to determine the resulting image dimensions, as well as the maximum number of iterations (**max_iterations**) needed for the Mandelbrot algorithm.
- Determine if a pixel at coordinates (x, y) belongs to the Mandelbrot set using the provided pseudo-algorithms in Section 4.

Data Handling:

- Store image data in a one-dimensional “vector” of pixels.
- Implement Matrix and Pixel structs with all fields and methods made public for ease of usage and testing on the Moped.

Detailed Description

1. "Pixel" struct [implement in image.rs]

The Pixel struct is used to store information about a single pixel, denoted by the values for red, green and blue colors (RGB) in the [0-255] range, therefore the values for the colors should use unsigned 8-bit integers. The following fields should be created:

- `r` – unsigned integer allowing [0-255] range representing the value for the red color channel.
- `g` – unsigned integer allowing [0-255] range representing the value for the green color channel.
- `b` – unsigned integer allowing [0-255] range representing the value for the blue color channel.

This struct should implement the Debug, Copy, Clone and PartialEq traits.

The `Pixel` struct should also implement Display trait, when printed, the output should have r, g and b values separated by a single space `' '`, e.g., white pixel would be printed as:

255 255 255

Code examples:

```
let p1 = Pixel{r: 255, g: 0, b: 0}; // create a pixel with red channel set to 255
let p2 = p1; // will take ownership unless copied

println!("{}", p1); // will work only if copy and display traits are imlpemented
```

2. "Image" struct [implement in image.rs]

The Image struct is used to allow easier traversal through matrix and the vector holding its data values. The following fields should be created:

- `width` – an unsigned integer value.
- `height` – an unsigned integer value.
- `data` – a vector of pixels for storing `width*height` values.

This struct needs to implement the following methods:

- `pub fn new(width: usize, height: usize ) -> Image`
  Allocates a vector that holds `width*height` pixels and initializes it so that each pixel value is set to `Pixel{r: 0, g: 0, b: 0}`.

  Code example:
  ```
  let img1 = Image::new(100, 200); // creates an image that allocated a vector of 100*200 elements
  ```

- `pub fn get(&self, x: usize, y: usize) -> Option<&Pixel>`
  Returns an immutable reference to a Pixel wrapped in an `Option` enum, which contains some data if the pixel exists, or it returns None if the given x and y values are out of range.

  Code example:
  ```
  let img1 = Image::new(100, 200); // creates an image that allocated a vector of 100*200 elements
  ```

```
let element_0_0 = img1.get(0, 0).unwrap(); // unwrapped reference to element in the image
```

- `pub fn get_mut(&mut self, x: usize, y: usize) -> Option<&mut Pixel>`
  Same as above, excepts it returns a mutable reference.

- `pub fn get_mandelbrot_pixels(&self) -> usize`
  Returns the number of pixels in the Mandelbrot set by iterating over the data **vector using iterators and closures**.

3. Complex numbers [implement in complex.rs]

The "`Complex`" struct is used to define a type that helps us work with complex numbers needed for the Mandelbrot algorithm (`check_pixel`) function. Every complex number can be represented in the form of $a + bi$, where i is an imaginary unit, a is called the real part and b is called the imaginary part. We represent this number with a structure.

The following fields should be created:
- **`re`** – a floating point value representing the real part of the complex number.
- **`im`** – a floating point value representing the imaginary part of the complex number.

This struct should implement Clone, Copy, Add and Mul traits as well as a method for calculating "magnitude":

- **`Add` trait** – performs addition of two Complex numbers.
  Addition of 2 complex numbers means simply adding the real parts of the number with the complex parts of the number, example:

```
c1 =           ( 5.0 + 3.0i)
c2 =           ( 2.0 + 4.0i)
c3 = c1 + c2 = ( 7.0 + 7.0i)

=> Complex {re: 7.0, im: 7.0}
```

- **`Mul` trait** – performs multiplication of two complex numbers.
  Multiplication of 2 complex numbers is defined with:
  $$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i$$
  For example:

```
c1 = (5.0 + 3.0i)
c2 = (2.0 + 4.0i)
c3 = c1 * c2 = (5.0*2.0 – 3.0*4.0) + (5.0*4.0 + 3.0*2.0)i

=> Complex {re: -2.0, im: 26.0}
```

- **`mag` method** – calculates the magnitude of the current complex number.
  Instead of calculating the actual absolute value (since we are comparing against the value 4 in our algorithm), we calculate the following:
  $$(a^2 + b^2)$$
  Example:
```
let c = Complex {Re: 5.0, Im: 3.0}
let c_mag = c.mag(); // c_mag needs to have a value of 34.0
```

4. Mandelbrot part [implement in mandelbrot.rs]

Implement following functions:

```rust
pub fn generate_image(width: usize, height: usize, max_iterations: usize) -> Image
```

This function **creates a new Image data structure** with the input dimensions (**width** and **height**). It then iterates over all pixels in this image. For each pixel, it maps the pixel coordinates (x, y) to the corresponding complex number (cx, cy) in the complex plane, using the following formula:

```rust
let cx = (x as f64/image.width as f64 - 0.75) * 3.5;
let cy = (y as f64/image.height as f64 - 0.5) * 2.0;

let c = Complex {re: cx, im: cy};
```

Here is a pseudo algorithm for this function:

```
For each pixel (x, y) in the image:
    Map the pixel coordinates (x, y) to the corresponding complex number (c_x, c_y) in the complex plane
        where the real part c_x ranges from -2.5 to 1 and the imaginary part c_y ranges from -1 to 1.

    Call the check_pixel function

    If the check_pixel function returns "Some" value:
        The pixel (x, y) is not in the Mandelbrot set.
        Assign a color representing that it is not within the set (white - RGB: 255, 255, 255).

    If the check_pixel function return "None":
        Assign a color representing that it diverges (black - RGB: 0, 0, 0).
```

Note that you need to use the Image struct and implemented traits to access the values in associated vector (e.g., `image.get(x, y)` or `image.get_mut(x, y)`).

Note that this function calls the `check_pixel` function, which checks if the pixel belongs to the Mandelbrot set. If the `check_pixel` function returns "`Some`" value, we set the color of that pixel to white, otherwise we set it to black (since it means it is in the Mandelbrot set).

```rust
pub fn check_pixel(c: Complex, max_iterations: usize) -> Option<usize>
```

For each given complex number, this function computes a complex number z by incrementing it from its initial value (`Complex { re: 0, im: 0 }`). The number is incremented using the following formula:

$$z_{n+1} = z_n^2 + c$$

Where c is the input value provided by the **`generate_image`** function.

Here is the pseudo code for the `check_pixel` function:

```
Set initial values:
    z = 0 + 0i
    iteration = 0

While iteration < max_iterations:
    Calculate the new value of z using the Mandelbrot formula:
        z = z^2 + c

    If the magnitude (absolute value) squared of z becomes greater than 4:
        The pixel (x, y) is not in the Mandelbrot set.

        Break out of the loop / return the number of iterations (use Option enum)

    Increment the iteration counter.

If the loop completes without z becoming greater than 4:
    The pixel (x, y) is in the Mandelbrot set.
    Return "None"
```

Note that the **max_iterations** value is provided as an input parameter (default: **1024**).
This function returns an Option enum. If a pixel does not belong to the Mandelbrot set, this function returns the number of iterations, otherwise it returns None, signifying that the pixel is in the Mandelbrot set.

Moreover, this function needs to use Add and Mul traits when dealing with complex numbers.

5. Parsing input [implement in client.rs]

Your program should accept the following three parameters:

- **width** – an integer representing the width of the image (required argument).
- **height** – an integer representing the width of the image (required argument).
- **max_iterations** – an integer representing the max number of iterations the algorithm should use (if not provided, or in case of a parsing error the default value of 1024 is used).

If the number of user-provided input arguments is different than 2*, your program needs to print the following usage message and exit**:

```
println!("Usage: {} <width> <height> <max_iterations>", args[0]);
```

*You can use `env::args().collect()` to get the arguments. Note that the first element of this collection is the name of the binary of your program, that is, if we want two user-provided input arguments, we need to test if this collection size is equal to 3.
**Note that you can exit using "`std::process::exit(1);`".

If the number of command-line arguments is correct, their values should be converted to unsigned integral values (according to the function signature). The return value of the function is a `Result` type containing the parsed values or parsing errors.

This functionality needs to be implemented in the following method in the client.rs:

```
pub fn parse_args() -> Result<(usize, usize, usize), ParseIntError>
```

The function needs to process input arguments and return parsed values as the Result type, in case of parsing error, the ParseIntError is returned.

6. Saving files

```rust
pub fn save_to_file(image: &Image, filename: &str)
```

This function has already been implemented, you need to call it to save data to .ppm file by passing the image struct and the output filename, which need to be "mandelbrot.ppm" on Moped.

7. Expected Behavior

Assuming your executable is called “mandelbrot”, your code will be executed with the following input parameters:

`./mandelbrot 525 300 1024` (or cargo run 525 300 1024)
The expected output is the following:

```
Generating Mandelbrot for 525x300 image (max_iterations: 1024)
Pixels in the set: 34062
```

Note that additional tests are performed (see tests.rs file), testing different image sizes, structures and traits (for Pixels, Images, and Complex) for various input values.

Your program should not produce any warnings, no additional info about bad input parameters, or other print outs. The output and test results should achieve a match on Moped.

To compile at home use:

- rustc main.rs
- or use Cargo (create a new project and put source files in the src folder, or specify alternative location in Cargo.toml)

To run tests at home use:

- rustc --crate-name tests tests.rs –test
- or run cargo test

8. Submission Guidelines

The program has to be submitted before the deadline on the online platform after it has passed all checks. Further information is provided in the lectures, tutorials and on Moodle.

Deadline: Wednesday, 11.12.2024 23:59 on the online platform.