Assignment 3: "Julia Set"

Develop a Rust program that generates an image of a Jula Set and saves is to disk in a **PPM P6** format.

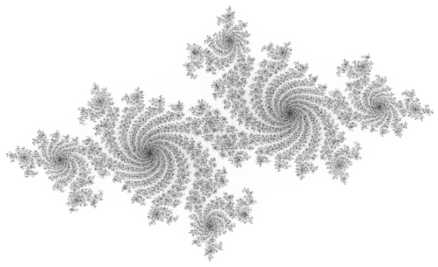

Figure 1

The resulting image (see Figure 1) should display the Julia Set at the specified resolution. Pixels that belong to the set must represented in black (rgb: 0, 0, 0) and those outside the set should be colorized based on the escape condition. An interactive description of how Julia Set is generated can be found at the following link: https://complex-analysis.com/content/julia_set.html. A detailed algorithm description is also provided in this document.

Requirements summary:

- Use **Rust** to develop an application that generates the Julia Set image.
- Save the generated image to disk in "PPM P6" format.
- The resulting image resolution (e.g., 600x600) is provided via input arguments.
- Implement the required features according to the specification below.

Color Representation:

- Pixels belonging to the Juia Set should be represented in black (rgb: 0, 0, 0).
- Pixel outside the Jula Set should be visualized with a grayscale gradient based on the provided function (see details below).

Implementation Details:

- The application must accept input parameters for:
  - **width** and **height** – integer numbers; reprenent the image size in pixels
  - **max_iterations** - maximum number of iterations
  - **c** - a complex number; represents a Julia set constant that defines the fractal's shape
  - **Center** – a complex number; indicates the center point of the region being visualized
  - **Zoom** - floating-point value; controls magnification of the fractal view (zooms in)
- For each pixel at coordinates (x, y), determine whether it belongs to the Julia Set using the 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 Moped.

Detailed Description

1. "Pixel" struct [implement in pixel.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 ) -> Self`
  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. Since this method is used only for testing, annotate it with "#[allow(dead_code)]" unless you are using it in other methods.

  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_black_pixel_count(&self) -> u32`
  Returns the number of pixels in the image 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 Julia Set 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 derive and/or implement Clone, Copy, Debug, Add and Mul traits as well as a method for calculating magnitude squared:

- **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_squared` method** – calculates the squared 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_squared = c.mag_squared(); // c_mag needs to have a value of 34.0
```

4. Julia Set Fractal part [implement in fractal.rs]

The Fractal struct defines the complete configuration needed to generate a Julia set image. It encapsulates both the rendering parameters (such as *resolution*, *iteration depth*, and *coloring*) and the viewport settings (*zoom* and *center*) that determine which part of the fractal is visible. Together, these fields specify what fractal to compute, how to color it, and where to save the result.

The following fields should be created:

- `width` – unsigned integer (usize); the width of the output image in pixels.
- `height` – unsigned integer (usize); the height of the output image in pixels.
- `max_iter` – unsigned integer (usize); the maximum number of iterations used to determine whether a point escapes (controls the level of detail).
- `c` – Complex number; the Julia set constant *c*that defines the fractal's shape (different values produce different fractal shapes).
- `zoom` – floating-point value (f32); controls magnification of the fractal view (higher values zoom in).
- `center` – Complex number; the coordinates of the center point in the complex plane, used to pan the view.
- `color_fn` – function pointer (fn(usize, usize) -> Pixel); maps iteration counts to a Pixel color, allowing different gradients or scheme.

Where color_fn is an **Option** enum of the **ColorFn** type, defined as:

```
pub type ColorFn = fn(usize, usize) -> Pixel;
```

Implement following functions:

```
pub fn render(&self) -> Image
```

This method computes Julia set fractal based on the parameters stored in the Fractal struct, and returns the generated image. First, it **creates a new Image data structure** with the given **width** and **height** dimensions. 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:

```
let zx = self.center.re + (x as f64 - self.width as f64 / 2.0) * scale_x;
let zy = self.center.im + (y as f64 - self.height as f64 / 2.0) * scale_y;
```

where

- scale_x is **3.0** divided by **zoom** divided by **width** (3.0 / zoom / width) and
- scale_y is **3.0** divided by **zoom** divided by **width** (3.0 / zoom / height)

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

Here is a pseudo algorithm for the code part of this function:

```
For each pixel (x, y) in the image:
    Map the pixel coordinates (x, y) to the corresponding complex number (z_x, z_y) in the complex plane
    using the given formula. This represents the initial value z0 passed to check_pixel function.

    Call the check_pixel(z0)

    If the check_pixel function returns "Some" value:
        The pixel (x, y) escaped the Julia set.
        Apply a gradient function using the value returned by the check_pixel function.

    If the check_pixel function return "None":
        Assign the black color.
```

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 Julia set. If the `check_pixel` function returns "`Some`" value, we use this result to either set the color of that pixel to black, or a apply gradient function using the value returned by check pixel function.

`pub fn check_pixel(&self) -> 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 the initial value of Z (Z_0) is passed by the **`render()`** function and C is the constant given by user via input arguments (or defaults to `(-0.7 + 0.27015i)`), and saved in the Fractal struct's **c** field.

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 Julia formula:
        z = z*z + c

    If the magnitude squared of z becomes greater than 4:
        The pixel (x, y) is not in the Julia 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 Julia set.
    Return "None"
```

Note that the **max_iterations** value is available in the Fractal struct (default: **300**).
This function returns an Option enum. If a pixel does not belong to the Julia set, this function returns the number of iterations, otherwise it returns None, signifying that the pixel is in the Julia set.

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

`fn default_color_fn(iter: usize, max_iter: usize) -> Pixel`
The default color function used to assign a visual gradient to escaped points, where faster escapes appear brighter and slower escapes appear darker. Takes the number of iterations iter it took for a point to escape, and the maximum allowed iterations max_iter.

It calculates a brightness value v by linearly scaling the iteration count to a range between 0 and 255.

- Points that escape quickly (low iter) will have high brightness (closer to white).
- Points that escape slowly (high iter) will have low brightness (closer to black).

This function returns a Pixel representing the grayscale color for the escaped point. It has already been implemented in fractal.rs and you just need to uncomment it and use it.

```rust
fn default_color_fn(iter: usize, max_iter: usize) -> Pixel {
    let v = 255 - (255.0 * iter as f64 / max_iter as f64) as u8;
    Pixel::new(v, v, v)
}
```

5. Parsing input [implement in client.rs]

Your program should accept the following parameters:

- `--width` – integer; the width of the image (required argument).
- `--height` – integer; the height of the image (required argument).
- `--max-iter` – integer; the max number of iterations used to determine if a point escapes the set (default: 300).
- `--c` – complex number; given as a comma-separated pair "-0.4,1" without quotes (default: -0.7, 0.27015). Represents a Julia set constant c that defines the fractal's shape.
- `--center` – complex number; given as a comma-separated pair "-0.4,1" without quotes (default: 0.0,0.0). Represents the coordinates of the center point in the complex plane, used to pan the view.
- `--zoom` – floating-point value; controls magnification of the fractal view (default: 1.0).
- `--o` – string; the filename where the generated image will be saved in PPM P6 format.

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

```
USAGE: julia [OPTION]

Options:
  --width=INT         width of the image in pixels (required)
  --height=INT        height of the image in pixels (required)
  --o=FILE            filename where the image will be saved (PPM P6 format, default: julia.ppm)
  --max-iter=INT      maximum number of iterations used to determine escape (default: 300)
  --c=REAL,REAL       coordinates of the center point in the complex plane (default: 0.0,0.0)
  --center=REAL,REAL  coordinates of the center point in the complex plane (default: 0.0,0.0)
  --zoom=FLOAT        magnification factor for the fractal view (default: 1.0)
```

*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.

If the number of command-line arguments is correct, their values should be converted to corresponding values (according to the Fractal struct signature). If a value is not provided, client should set the default values. 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(args: Vec<String>) -> Result<(Fractal, String), String>
```

The function needs to process input arguments and construct a Fractal object with the provided parameters along with the output filename. In case of parsing error, the Err(String) with an error message is returned.

In case of **an unknown argument**, it should print (without quotes) the name of the first unknown argument:

```
"Unknown argument: <argument_name>"
```

where <argument_name> is the name of the unknown argument

In case of parsing errors for width (usize), height (usize), max-iter (usize) and zoom (f64) arguments, the function should return the following error:

```
Invalid <argument_name>
```

where <argument_name> is the name of the argument (width, height, max-iter or zoom).

Example: ./julia --width 800 --height 600 --max-iter 1024 --constant -0.8,0.156 --center -0.5,0.3 --zoom 30.0 --output julia.ppm

```
Unknown argument: --constant
```

Complex number arguments are accepted in format "re,im", if parsing has too few or too many parts, the function should return the following error:

```
Invalid format for <argument_name>. Use --<argument_name> re,im
```

where <argument_name> is the name of the argument (`c`, or `center`).

For example: ./julia --width 800 --height 600 --max-iter 1024 --c -0.8,0.156

```
Invalid format for --c. Use --c re,im
```

Both imaginary and real part of the provided complex number must parse to a valid f64 value, otherwise function should return:

```
Invalid real part for --<argument_name>
Invalid imaginary part for --<argument_name>
```

where <argument_name> is the name of the argument (`c`, or `center`).

6. Saving files

- `pub fn save_ppm(image: &Image, filename: &str, format: &str) -> std::io::Result<()>`
  This function has already been implemented, you need to call it to save data to .ppm file by passing the image struct, the output filename (which needs to be "julia.ppm" on Moped), and the desired format. While saving in the P3 format is implemented, you need to implement the P6 version below.

- `pub fn save_ppm_p3 image: &Image, filename: &str) -> std::io::Result<()>`
  This function has already been implemented, it is from the function above.

- `pub fn save_ppm_p6(image: &Image, filename: &str) -> std::io::Result<()>`
  Saves the rendered fractal image to disk in the **PPM P6 (binary)** format. This format is more compact and faster to write/read than the text-based P3 format, which is already implemented in save_ppm_p3() method. Use this format for the final version of your program.

  This function operates as follows:

  - Opens (or creates) a file with the given filename
  - Writes the PPM header:
    - **P6** → indicates binary RGB format
    - **width height** → the image dimensions
    - **255** → the maximum color value
    - A single newline after the header
  - Iterates over all pixels in the image row by row
  - For each pixel, writes its red, green, and blue values as raw bytes (not text)
  - Closes the file when finished

  You need to call it to save data to .ppm file by passing the image struct and the output filename, which need to be "julia.ppm" on Moped.

  Hint: to save binary data you can use (file is of std::fs::File type - you need to use std::io::Write):

  ```
  file.write_all(&[p.r, p.g, p.b])?;
  ```

  Returns `Ok(())` if the file was successfully written, or an error (Err(std::io::Error)) if something went wrong (e.g., invalid filename, permission denied).

7. Expected Behavior

Assuming your executable is called “julia” and your program is executed with the following parameters:

`./julia --width 600 --height 600 --max-iter 1024` (or cargo run -- --width 600 --height 600 --max-iter 1024)

The expected output is the following:

```
Julia set saved to julia.ppm (pixel check: 826)

./julia
Usage: julia [OPTION]...
Generate a Julia set fractal image.

Options:
  --width=INT           width of the image in pixels (required)
  --height=INT          height of the image in pixels (required)
  --o=FILE              filename where the image will be saved (PPM P6 format, required)
  --max-iter=INT        maximum number of iterations used to determine escape (default: 300)
  --c=REAL,REAL         Julia set constant c as comma-separated pair (default: -0.7,0.27015)
  --center=REAL,REAL    coordinates of the center point in the complex plane (default: 0.0,0.0)
  --zoom=FLOAT          magnification factor for the fractal view (default: 1.0)
```

Another example (default on Moped):

`./julia --width 400 --height 400 --max-iter 300` (or cargo run -- --width 400 --height 400 --max-iter 1024)

```
Julia set saved to julia.ppm (pixel check: 9984)
```

In your main.rs you need to get user input arguments, call client to construct the Fractal struct, and then call render() method to generate the Jula set image. You can then count the number of pixels and save the data to the output file.

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.

Although not a requirement for the assignment, each struct can also implement "new" method if you need it in your implementation.

Deadline: Wednesday, 10.12.2025 12:00 on the online platform.