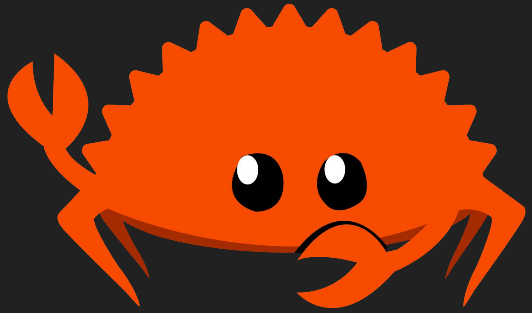
Mandelbrot Plotter

Daniel Bank

https://github.com/danielbank/mandelbrot



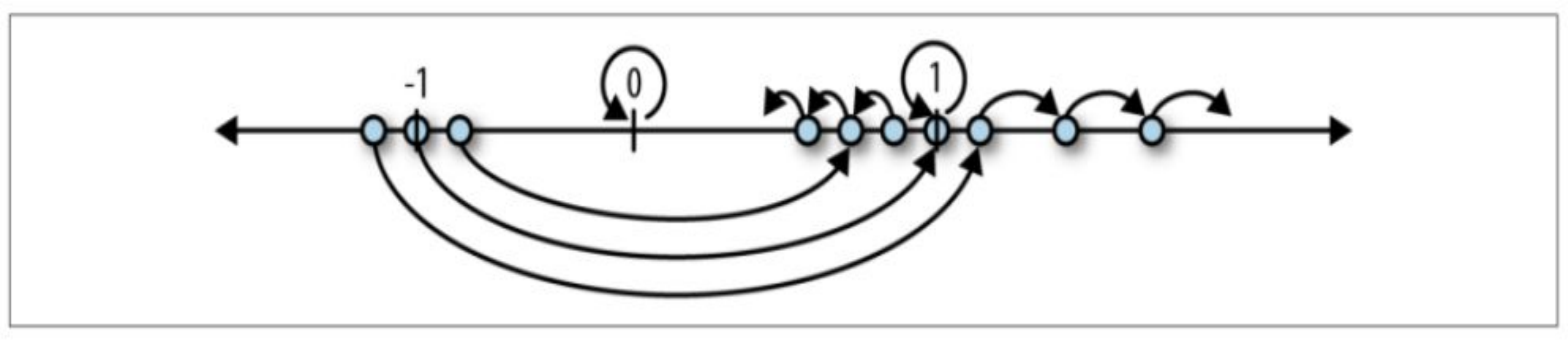
Programming Rust (“The Crab Book”)

I’m working through the Programming Rust book by Jim Blandy and Jason Orendorff. This Mandelbrot Plotter project and notes come from this book.

What is the Mandelbrot Set?

The Mandelbrot set is defined as the set of complex numbers **c** for which **z** does not fly out to infinity.

```
fn complex_square_add(c:
Complex<f64>) {
    let mut z = Complex { re:
0.0, im: 0.0 };
    loop {
        z = z * z + c;
    }
}
```



Mandelbrot Calculation

Option is an *enumerated* type (enum). Option<T> is either Some(v) where v is a value of type T, or None

- Limit the number of iterations using the **limit** parameter.
- If the value wanders out of a circle of radius 2, we know it will blow up. So we can return early in that case.

- Idiomatic Rust: Use **return** statements for explicit early returns, use an expression (without semicolon!) for the function’s value when control falls off the end.

```
use num::Complex;

fn escape_time(c: Complex<f64>, limit: u32)
-> Option<u32> {
    let mut z = Complex { re: 0.0, im: 0.0
};
    for i in 0..limit {
        z = z * z + c;
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
    }

    None
}
```

Complex<T> is a Generic Struct

<T> can be read as “for any type T”:

- Complex<f64>
- Complex<f32>
- … etc

```
struct Complex<T> {
  // Real portion of the complex number
  re: T,
  // Imaginary portion of the complex number
  im: T
}
```

Parsing Pairs

- **<T: FromStr>** can be read as “for any type T that implements the FromStr trait”
- Argument to match expression is a tuple expression. Pattern only matches if both elements of the tuple are Ok variants of the Result type.

```
use std::str::FromStr;

fn parse_pair<T: FromStr>(s: &str, separator:
char) -> Option<(T, T)> {
    match s.find(separator) {
        None => None,
        Some(index) => match
(T::from_str(&s[..index]),
T::from_str(&s[index + 1..])) {
            (Ok(l), Ok(r)) => Some((l, r)),
            _ => None,
        },
    }
}
```

Use parse_pair() to Parse Complex Numbers

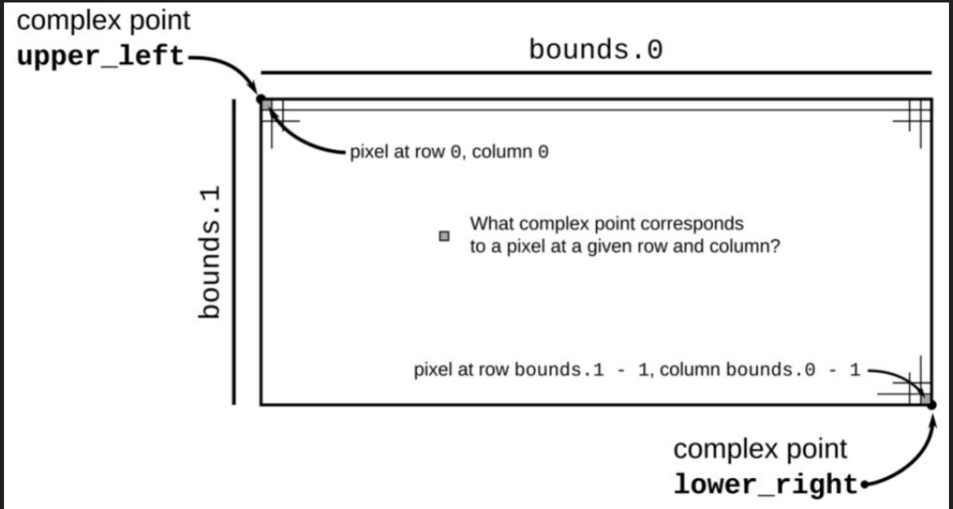
- **parse_pair()** can use any separator character and just returns a tuple.
- Using “,” as the separator, if we yield a tuple, we can initialize a Complex type
- **Complex { re, im }** is shorthand for **Complex { re: re, im: im }**

```
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None,
    }
}
```

Mapping from Pixels to Complex Numbers

- Won’t explain the math in the **pixel_to_point()** fn.
- **pixel.0** refers to the first element of the tuple **pixel**
- Rust generally refuses to convert between numeric types implicitly so you need to write it out:

```
pixel.0 as f64
```


complex point
upper_left
bounds.0
bounds.1
pixel at row 0, column 0
What complex point corresponds to a pixel at a given row and column?
pixel at row bounds.1 - 1, column bounds.0 - 1
complex point
lower_right

Plotting the Set

- If **escape_time()** yields None, we color the pixel black (the number is in the set)
- If **escape_time()** yields a number (u32), we color it a shade of gray based on how larger that number is (how long it took to fall out).

The size of the **usize** primitive is how many bytes it takes to reference any location in memory. For example, on a 32 bit target, this is 4 bytes and on a 64 bit target, this is 8 bytes.

```
fn render(
    pixels: &mut [u8],
    bounds: (usize, usize),
    upper_left: Complex<f64>,
    lower_right: Complex<f64>,
) {
    assert!(pixels.len() == bounds.0 * bounds.1);

    for row in 0..bounds.1 {
        for column in 0..bounds.0 {
            let point = pixel_to_point(bounds, (column,
row), upper_left, lower_right);
            pixels[row * bounds.0 + column] = match
escape_time(point, 255) {
                None => 0,
                Some(count) => 255 - count as u8,
            };
        }
    }
}
```

Writing an Image

- **()** is the *unit* type, akin to **void** in C
- The **?** operator is shorthand for making a check that returns the **Ok(f)** or the **Err(e)** of a **Result**
- It's a common beginner mistake to use **?** in the **main** function, but this won't work because **main** does not have a return value
- The **?** operator is only useful in functions that themselves return **Result**
- In **main()**, use you can use **expect()**

```
use image::png::PNGEncoder;
use image::ColorType;

fn write_image(
    filename: &str,
    pixels: &[u8],
    bounds: (usize, usize),
) -> Result<(), std::io::Error> {
    let output = File::create(filename)?;

    let encoder = PNGEncoder::new(output);
    encoder.encode(
        &pixels,
        bounds.0 as u32,
        bounds.1 as u32,
        ColorType::Gray(8),
    )?;

    Ok(())
}
```

Non-Concurrent Example

```
let args: Vec<String> = std::env::args().collect();

let bounds = parse_pair(&args[2], 'x').expect("error parsing image dimensions");
let upper_left = parse_complex(&args[3]).expect("error parsing upper left corner point");
let lower_right = parse_complex(&args[4]).expect("error parsing lower right corner
point");

let mut pixels = vec![0; bounds.0 * bounds.1];
render(&mut pixels, bounds, upper_left, lower_right);
write_image(&args[1], &pixels, bounds).expect("error writing PNG file");
```

- **vec![v; n]** is a macro call that creates a vector **n** elements long whose elements are initialized to **v**