Now you have a tiny game and know some basics of Rust. Congrats!

Pro tips

Read the error messages!
Be patient!
You'll be okay!!

Resources

Rust book: **https://doc.rust-lang.org/book/**

Rust documentation: **https://doc.rust-lang.org/std/index.html**

Crates (packages): **https://crates.io**

Crate documentation: **https://docs.rs**

Terminal tutorials: **https://blog.balthazar-rouberol.com/discovering-the-terminal**, **https://www.learnshell.org/**, **https://coolguy.website/map-is-the-territory/introduction.html**

More in-depth explanations of ownership and references: **https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html**, **https://www.youtube.com/watch?v=8M0QfLUDaaA** (no captions but good)

Credits

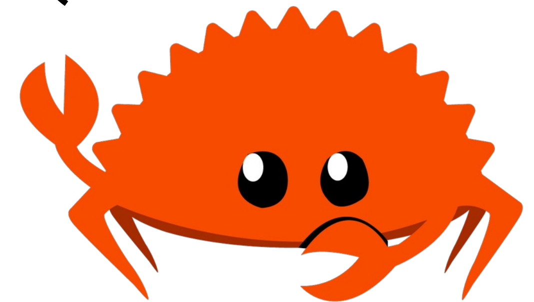
Ferris image from **rustacean.net** // draws inspiration from the Rust book and Becca Turner's RustConf 2020 talk

serif font is Elstob // `monospace font is Victor Mono`

written by Cynthia Li // made with 🦀💙

An Introduction to Rust

if you don't know systems programming



Why Rust?

Rust is a fast and memory-safe programming language with a strong type system and declarative memory management. It's very focused on letting you write *correct* code, but can be hard to learn because of the restrictions it imposes to ensure this.

Rust also has *really* good error messages, well-written documentation, and excellent tools that make writing code and debugging easier. I like it because it helps me make sure that my program is doing exactly what I want it to. I hope you like it too!

Borrowing

But sometimes you want values to be accessible or modifiable from different locations. Here you would borrow `game` with a reference using the `&` symbol, which points to the data at a variable without moving it.

```
let mut game = Game::new();
{
    let game2 = &game;
    // do things with game2
}
// ...
```

And this would compile! If you wanted to use `game2` to modify `game`, you would make the reference mutable:

```
let game2 = &mut game;
```

You can have as many immutable references to the same value as you want, but only one mutable reference. Like ownership, Rust hates it when multiple things are able to modify the same data at the same time.

Rules of thumb[1]

Every value can only ever have either:

1. No borrows
2. One or more immutable references (`&`)
3. Exactly one mutable reference (`&mut`)

Ownership and references are tricky concepts, so don't worry if you don't understand them at first. You'll get it!

---

1 this list was brought to you by everything Joe Osborn has told me about Rust ownership. Thanks Prof Osborn

What's up with the mut keyword?

All variables in Rust are immutable by default, so we have to explicitly tell Rust that we want to be able to change it. This might seem weird and restrictive, but Rust is very cautious: it doesn't want you accidentally modifying data that shouldn't be changed.

Likewise, when defining our game's update method, we had to add a `mut` keyword to let us modify the game state.

```
fn update(&mut self) { ... }
```

But what about the &?

Detour: Ownership

Every value in Rust can only owned by a single variable. For example, if we try to assign our game to another variable,

```
let mut game = Game::new();
{
    let mut game2 = game;
    // do things with game2
}
loop {
    game.draw();
    game.update();
}
```

This won't compile because the game now belongs to `game2` after assigning the value again, so the first `game` is now invalid. Rust doesn't want multiple variables to own the same data at the same time. This has to do with how memory management works in Rust, but I won't get into that here (on the final page there are links to people who explain it far better than I could).

Getting started

Where do I write Rust? Rust has two language servers that attach to your editor to analyze your code while you write it, `rls` and `rust-analyzer` (most people use the second). If you're not already attached to a text editor, VS Code's `rust-analyzer` extension is really good!

Where do I find documentation? Documentation for Rust's standard library is at **doc.rust-lang.org/std/index.html**. Documentation for crates like macroquad is at **docs.rs/crate_name**.

What's cargo? Cargo is Rust's package manager and build system. It downloads crates from **crates.io**, runs projects, runs tests, generates documentation, manages dependencies, and more! You use it from the terminal---if you're not familiar with it, there's a tutorial linked on the last page.

An example

I'm going to walk through an example of an extremely minimal 2D game using the library Macroquad. First, you want to make a new crate (a project) with cargo. In the terminal, navigate to the folder where you want to make your project, and run the command:

```
cargo new --bin cool-game
```

This will create a new folder in that directory, `cool-game`. It will contain a `src/` directory, where your Rust files will go, and a Cargo.toml file, where you'll list the dependencies of the crate---in this case, just Macroquad (version 0.3.10).

At the end of the file, under [dependencies], add:

```
macroquad = "0.3.10"
```

and save the file.

Now, open the `main.rs` file in the `src/` folder and replace its contents with some setup code for Macroquad, taken from its documentation page: **docs.rs/macroquad/0.3.10**

```
use macroquad::prelude::*;

#[macroquad::main("cool game")]
async fn main() {
    loop {
        next_frame().await;
    }
}
```

import Macroquad's functions to this file

a macro/attribute that tells Macroquad what the title of the game is, and to use the function below to run the game

Macroquad uses async/await to let its games run in the browser, but you don't need to worry about it here

You can run this from the terminal with the command `cargo run`, and a window with a black background should pop up. This isn't particularly interesting, but Macroquad provides functions we can draw with:

```
loop {
    clear_background(DARKBLUE);
    draw_rectangle(350.0, 250.0, 100.0, 100.0, PINK);
    next_frame().await;
}
```

Now when we run this there'll be a blue background and a pink square in the middle.

```
    fn draw(&self) {
        clear_background(DARKBLUE);

        draw_rectangle(self.player.x, self.player.y,
            self.player.w, self.player.h, PINK);
    }
}
```

no `this` like in Java; need to explicitly include `self` parameter like in Python

Then replace the contents of our async main function with

```
let mut game = Game::new();
loop {
    if is_key_down(KeyCode::Q) {
        println!("thanks for playing!");
        break;
    }
    game.draw();
    game.update();
    next_frame().await;
}
```

if a function doesn't have a `self` parameter, call it with `Type::function_name()`

if it does, call it with a [.] on a variable of the type the function is implemented for

And we'll have a pink square on a blue background that we can move with arrow keys. Yay :)

Data Structures

Games are usually interactive, so let's make a player character. We can define a struct---like a Java/Python class---to hold data about our game.

```
struct Game {
    player: Rect,
}
```

Macroquad gives us the type `Rect`, which has an x and y position, a width, and a height

We can implement functions for our `Game` with an `impl` block.

```
impl Game {
    fn new() -> Self {
        Self { player: Rect::new(50.0, 50.0, 25.0, 25.0) }
    }

    fn update(&mut self) {
        if is_key_down(KeyCode::Up) {
            self.player.y -= 1.0;
        }
        if is_key_down(KeyCode::Down) {
            self.player.y += 1.0;
        }
        if is_key_down(KeyCode::Right) {
            self.player.x += 1.0;
        }
        if is_key_down(KeyCode::Left) {
            self.player.x -= 1.0;
        }
    }
```

`Self` is shorthand for the type of the impl block

a rectangle at (50, 50) with width 25 and height 25

A tour of some basic syntax

We can add an if statement at the beginning of the loop to break out of it (then close the window) when the player presses the Q key:

```
if is_key_down(KeyCode::Q) {
    println!("thanks for playing!");
    break;
}
```

no parentheses needed around the condition

A keycode is an enum, or a type that can be one of several variants. For example, defining an `enum Pet { Cat, Dog }` allows you to say that your pet is a `Pet::Cat` or a `Pet::Dog`, but nothing else.

`println!` is a macro (you can tell because of the !) used to format and print things to the console

What about drawing many things?

Here's a for loop:

```
for i in 0..5 {
    draw_rectangle(
        150.0 * i as f32,
        250.0, 100.0, 100.0, PINK
    );
}
```

a range from 0 to 5 (not including 5)

`i` is an integer, so we use `as` to cast between number types (floats to integers, etc)

You can iterate over a range or a collection.

```
let positions = vec![100.0, 250.0, 400.0, 550.0];
```

declare variables with `let`. You don't need to say what type it is unless it's ambiguous (if the compiler gets confused and tells you to)

`vec!` is a macro that gives a shorthand to declare `Vec`s, which are like arrays

```
for x in positions.iter() {
    draw_rectangle(x, 250.0, 100.0, 100.0, PINK);
}
```

There are always options

Rust doesn't allow null variables, and you have to define what value a variable holds before using it. For example, this won't compile:

```
let positions;
for x in positions {
    draw_rectangle(x, 250.0, 100.0, 100.0, PINK);
}
```

If you want a variable to maybe hold a value, you can use the enum `Option`, and use a match statement to check if it's `Some(value)` or `None`.

```
let maybe_positions = Some(vec![100.0, 250.0, 400.0,
550.0]);

match maybe_positions {
    Some(positions) => {
        // loop through the x positions in rects
        for x in positions {
            draw_rectangle(x, 250.0, 100.0, 100.0, PINK);
        }
    },
    None => { /* do nothing */ }
}
```

a match statement is an if statement that compares one variable against the values on its branches. Often it's used to check what variant of an enum a variable is, but can also match on other types like numbers or strings.