🦀

rustlings

intro2.rs

The purpose of this exercise is to introduce to Rust's `println!` macro, which is used to print text to the console.

Initial code :

```
fn main() {
    printline!("Hello there!");
}
```

In this code, the `printline!` macro is incorrectly used to print the text "Hello there!" to the console. However, Rust's standard macro for printing is `println!`, not `printline!`. As a result, this code will produce a compilation error because `printline!` is not a recognized macro.

```
fn main() {
    println!("Hello there!");
}
```

With this change, the code will now successfully print "Hello there!" to the console when executed.

variable6.rs

This exercise show the importance of specifying the correct data type when declaring constants in Rust.

Initial code :

```
const NUMBER = 3;

fn main() {
```

```
    println!("Number {}", NUMBER);
}
```

In this code, a constant named `NUMBER` is declared without explicitly specifying its data type. In Rust, it's crucial to provide an explicit type for constants to ensure type safety and clarity in the code.

Here is how it should be done :

```
const NUMBER: u32 = 3;

fn main() {
    println!("Number {}", NUMBER);
}
```

In the corrected code, we've defined the constant `NUMBER` with an explicit type annotation `: u32`, indicating that it is an unsigned 32-bit integer.

function5.rs

In this exercice, the goal is to to demonstrate the correct usage of Rust functions and return statements.

Initial code :

```
fn main() {
    let answer = square(3);
    println!("The square of 3 is {}", answer);
}

fn square(num: i32) -> i32 {
    num * num;
}
```

In this code, a function named `square` is defined to calculate the square of a given number. However, the function body contains a semicolon `;` after the expression `num * num`, which makes it an expression statement instead of a return statement. As a result, no result is returned.

To fix this, I've removed the semicolon `;` after the expression `num * num` in the `square` function. This change converts the expression into a return statement, ensuring that the computed value is correctly returned as the result of the function.

```
fn square(num: i32) -> i32 {
    num * num
}
```

quizz1.rs

This quiz focuses on understanding variables, functions, and conditional statements in Rust and it asks to create a function that calculates the price of a quantity of apples based on certain conditions.

The initial code provides a brief explanation of the task and declares a function `calculate_price_of_apples`, but it's left incomplete.

```
//Mary is buying apples. The price of an apple is calculated 
// - An apple costs 2 rustbucks.
// - If Mary buys more than 40 apples, each apple only costs 
// Write a function that calculates the price of an order of 
// quantity bought.


//Put your function here!
// fn calculate_price_of_apples {
```

To calculate the price of apples based on the quantity bought we define the `calculate_price_of_apples` function. Here's an explanation of the code solution:

```
fn calculate_price_of_apples(apple: u32) -> u32 {
```

This line declares the function `calculate_price_of_apples` that takes a single argument `apple` of type `u32` (unsigned 32-bit integer) representing the quantity of apples bought. It also specifies that the function returns an unsigned 32-bit integer (`u32`) representing the total price.

```
    if apple > 40 {
        // If more than 40 apples are bought, each apple cost
        apple
    } else {
        // If 40 or fewer apples are bought, each apple costs
        apple * 2
    }
}
```

This block of code contains the conditional statement that determines the price of the apples based on the quantity bought. If the quantity of apples is greater than 40, each apple costs 1 rustbuck; otherwise, each apple costs 2 rustbucks.

Final code :

```
fn calculate_price_of_apples (apple: u32) -> u32{
    if apple > 40 {
        apple
    } else {
        apple*2
    }
}

// Don't modify this function!
#[test]
fn verify_test() {
    let price1 = calculate_price_of_apples(35);
    let price2 = calculate_price_of_apples(40);
    let price3 = calculate_price_of_apples(41);
    let price4 = calculate_price_of_apples(65);

    assert_eq!(70, price1);
    assert_eq!(80, price2);
    assert_eq!(41, price3);
    assert_eq!(65, price4);
}
```

if3.rs

As shown in the tests, the expected behavior of the `animal_habitat` function are these one :

If the input is “gopher”, the output should be “Burrow”

If the input is “snake”, the output should be “Desert”

If the input is “crab”, the output should be “Beach”

If the input is “dinosaur”, the output should be “Unknown”

```
mod tests {
    use super::*;

    #[test]
    fn gopher_lives_in_burrow() {
        assert_eq!(animal_habitat("gopher"), "Burrow")
    }

    #[test]
    fn snake_lives_in_desert() {
        assert_eq!(animal_habitat("snake"), "Desert")
    }

    #[test]
    fn crab_lives_on_beach() {
        assert_eq!(animal_habitat("crab"), "Beach")
    }

    #[test]
    fn unknown_animal() {
        assert_eq!(animal_habitat("dinosaur"), "Unknown")
    }
}
```

Now, let's examine the code to see how it achieves these expected outcomes.

```
pub fn animal_habitat(animal: &str) -> &'static str {
    let identifier = if animal == "crab" {
```

```
        1
    } else if animal == "gopher" {
        2.0
    } else if animal == "snake" {
        3
    } else {
        "Unknown"
    };

    // DO NOT CHANGE THIS STATEMENT BELOW
    let habitat = if identifier == 1 {
        "Beach"
    } else if identifier == 2 {
        "Burrow"
    } else if identifier == 3 {
        "Desert"
    } else {
        "Unknown"
    };

    habitat
}
```

The `animal_habitat` function in the original code has issues with inconsistent data types and logic. For instance, it assigns different types to the `identifier` variable based on the animal, leading to compilation errors. Additionally, it wrongly assigns a string ("Unknown") to an integer variable, which is also incorrect.

```
pub fn animal_habitat(animal: &str) -> &'static str {
    let identifier = if animal == "crab" {
        1
    } else if animal == "gopher" {
        2
    } else if animal == "snake" {
        3
    } else {
        4
```

```
    };

    // DO NOT CHANGE THIS STATEMENT BELOW
    let habitat = if identifier == 1 {
        "Beach"
    } else if identifier == 2 {
        "Burrow"
    } else if identifier == 3 {
        "Desert"
    } else {
        "Unknown"
    };

    habitat
}
```

In the corrected code, the `identifier` variable consistently uses integer values for each animal type. It also correctly assigns an integer value (4) to the "Unknown" case to maintain consistency. With these corrections, the function now behaves as expected, returning the correct habitat for each animal and handling unknown animals.

primitive_types6.rs

This exercise focuses on understanding tuple indexing in Rust. It asks to access the second element of a tuple using tuple indexing syntax.

The initial code provides a tuple named `numbers` and a placeholder for accessing the second element of the tuple.

```
// Replace below ??? with the tuple indexing syntax.
let second = ???;
```

The placeholder `???` indicates where the expression for accessing the second element of the tuple should be placed.

The final version of the code correctly accesses the second element of the tuple `numbers` using tuple indexing syntax.

```
let second = numbers.1;
```

Here, `.1` is used to access the second element of the tuple `numbers`. In Rust, tuple elements are indexed starting from 0, so `.1` refers to the second element of the tuple.

Here is the final code :

```
#[test]
fn indexing_tuple() {
    let numbers = (1, 2, 3);
    // Replace below ??? with the tuple indexing syntax.
    let second = numbers.1;

    assert_eq!(2, second,
        "This is not the 2nd number in the tuple!")
}
```

vec2.rs

This exercise focuses on understanding and applying operations on vectors (Vec) in Rust. It requires doubling each element of a given vector.

The initial code defines two functions: `vec_loop` and `vec_map`. Both functions are incomplete, with placeholders indicating where the necessary operations should be performed to double each element of the vector.

```
// TODO: Fill this up so that each element in the Vec `v` is
???
```

These placeholders indicate where the operation to double each element should be placed.

```
// TODO: Do the same thing as above - but instead of mutating
// Vec, you can just return the new number!
???
```

Similarly, this placeholder indicates where the operation to double each element should be placed, but this time without mutating the original vector.

`vec_loop` **Function:**

In Rust, when we want to modify elements of a vector in place, we often use a mutable reference to each element.

We use `iter_mut()` to obtain a mutable iterator over the elements of the vector. This iterator provides mutable references to each element, enabling us to modify them directly.

The `for` loop is a construct in Rust used for iteration. In this context, it iterates over the mutable references obtained from `iter_mut()`, allowing us to mutate each element individually.

To access the value of each mutable reference, we dereference it using `*`. This allows us to modify the value of each element in place.

```
fn vec_loop(mut v: Vec<i32>) -> Vec<i32> {
    for element in v.iter_mut() {
        *element = *element * 2; // element is an adress.
                                 //If we want to double it, w
    }
    v
}
```

`vec_map` **Function:**

In Rust, when we want to transform elements of a vector without modifying the original vector, we often use iterators and the `map` function.

We use `iter()` to obtain an iterator over the elements of the vector. This iterator provides immutable references to each element, allowing us to access them without modifying the original vector.

The `map` function is a higher-order function in Rust that applies a closure to each element of an iterator, producing a new iterator with the transformed values. In this context, we use `map` to double each element, without altering the original vector.

```
fn vec_map(v: &Vec<i32>) -> Vec<i32> {
    v.iter().map(|element| {
        element * 2
```

```
    }).collect()
}
```

move_semantics6.rs

The purpose of this exercise is to reinforce understanding of Rust move semantics and ownership rules. The exercice involves ensuring that functions take ownership of data appropriately to avoid unnecessary copying and adhere to Rust's ownership principles.

The initial code provides two functions, `get_char` and `string_uppercase`, which are both wrong in the logic of ownerships. The task is to modify the functions to ensure proper ownership semantics:

```
fn get_char(data: String) -> char {
    data.chars().last().unwrap()
}

fn string_uppercase(mut data: &String) {
    data = &data.to_uppercase();

    println!("{}", data);
}
```

In `get_char`, the function should not take ownership of the `data` parameter, while in `string_uppercase`, the function should take ownership of `data`.

`get_char` Function:

In Rust, we should pass references (`&`) to functions whenever possible to avoid taking ownership of data unnecessarily. Since `get_char` only needs to read the data and not modify it, it should take a reference to the `String` parameter.

```
fn get_char(data: &String) -> char {
    data.chars().last().unwrap()
}
```

By changing the parameter type to `&String`, we indicate that the function borrows the `String` data without taking ownership. We also use methods to access the characters of the `String` without modifying its ownership. The `last()` method returns an `Option` containing the last character, and `unwrap()` is used here to retrieve the character.

`string_uppercase` Function:

In `string_uppercase`, the function needs to modify the `String` data by converting it to uppercase. Since the function should take ownership of the `String`, it should receive it by value (without a reference).

```
fn string_uppercase(mut data: String) {
    data = data.to_uppercase();
    println!("{}", data);
}
```

By receiving the `String` parameter directly (without a reference), the function takes ownership of the data. Inside the function, we use the `to_uppercase()` method to convert the `String` to uppercase. Since the function owns the `String`, it can modify it directly.

structs3.rs

The purpose of this exercise is to demonstrate how to define a struct in Rust with associated methods and to implement logic related to the struct. in this exercise, we are working with a `Package` struct representing a shipping package and implementing methods to determine if the package is international and calculate shipping fees.

The initial code defines a `Package` struct with fields for sender and recipient countries, and weight in grams.

```
struct Package {
    sender_country: String,
    recipient_country: String,
    weight_in_grams: u32,
}
```

It also includes methods `new`, `is_international`, and `get_fees`, but they are incomplete. The `new` method panics if the weight of the package is below 10 grams.

```
fn is_international(&self) -> ??? {
    // Something goes here...
}

fn get_fees(&self, cents_per_gram: u32) -> ??? {
    // Something goes here...
}
```

The `is_international` method should return a boolean indicating if the package is international, and `get_fees` should calculate shipping fees based on the weight of the package and a given fee per gram.

`is_international` Method:

In Rust, methods that do not mutate the struct's state typically take a reference to the struct as their first parameter. This convention is denoted by the `&self` parameter. In the `is_international` method, `&self` indicates that the method does not consume or modify the `Package` struct; it only borrows it for reading purposes.

To determine if a package is international, we compare the sender and recipient countries. If they are different, the package is international; otherwise, it is local.

```
fn is_international(&self) -> bool {
    self.sender_country != self.recipient_country
}
```

We compare the sender and recipient countries stored in the `Package` struct. If they are not equal, it indicates that the package is international.

`get_fees` Method:

Similarly, in the `get_fees` method, `&self` indicates that the method does not modify the `Package` struct's state. It only borrows the struct to access its fields and calculate shipping fees based on the provided parameters.

To calculate the shipping fees, we multiply the weight of the package (in grams) by the given fee per gram.

```
fn get_fees(&self, cents_per_gram: u32) -> u32 {
    self.weight_in_grams * cents_per_gram
}
```

We multiply the weight of the package stored in the `Package` struct by the given fee per gram to calculate the total shipping fees.

enum3.rs

The purpose of this exercise is to demonstrate the usage of enums in Rust, particularly in the context of message handling within a state struct. The goal is to define an enum `Message` containing different message variants and implement logic in the `State` struct to process these messages appropriately.

The initial code provides a skeleton structure with an empty `enum Message` and incomplete implementation of the `State` struct. The `Message` enum needs to be defined with variants representing different types of messages such as change color, echo, move, and quit. Additionally, the `State` struct methods need to be updated to handle these message variants using pattern matching.

```
enum Message {
    // TODO: implement the message variant types based on the
}

impl State {
    fn process(&mut self, message: Message) {
        // TODO: create a match expression to process the dif
    }
}
```

Defining `Message` Enum:

The `Message` enum needs to be defined with variants corresponding to different types of messages that the `State` struct can handle. It will look like this :

```
enum Message {
    ChangeColor(u8, u8, u8),
```

```
    Echo(String),
    Move(Point),
    Quit,
}
```

Implementing `State::process` Method:

The `process` method of the `State` struct is updated to handle different message variants using pattern matching.

```
fn process(&mut self, message: Message) {
        ///
}
```

The `match` expression is Rust's powerful construct for pattern matching. It allows us to match different patterns of values and execute corresponding code blocks based on these patterns. Here, we're matching the `message` parameter against each variant of the `Message` enum.

```
  match message {
      Message::ChangeColor(r, g, b) => {
          self.change_color((r, g, b));
      }
      Message::Echo(s) => {
          self.echo(s);
      }
      Message::Move(p) => {
          self.move_position(p);
      }
      Message::Quit => {
          self.quit();
      }
  }
```

Handling `Message` Variants:

When the `message` is of type `Message::ChangeColor`, the tuple `(r, g, b)` is extracted from the variant. Then, the `change_color` method of the `State` struct is invoked with the RGB values passed as arguments.

```
fn change_color(&mut self, color: (u8, u8, u8)) {
        self.color = color;
    }

Message::ChangeColor(r, g, b) => {
          self.change_color((r, g, b));
 }
```

If the `message` is an `Echo` variant, the associated `String` `s` is extracted. The `echo` method of the `State` struct is called with the extracted `String` argument.

```
fn echo(&mut self, s: String) {
        self.message = s
    }

Message::Echo(s) => {
          self.echo(s);
      }
```

When the `message` is a `Move` variant, the `Point` `p` is extracted. The `move_position` method of the `State` struct is called with the extracted `Point` argument.

```
fn move_position(&mut self, p: Point) {
        self.position = p;
    }

Message::Move(p) => {
          self.move_position(p);
      }
```

Finally, If the `message` is a `Quit` variant, no additional data needs to be extracted. The `quit` method of the `State` struct is invoked.

```
fn quit(&mut self) {
        self.quit = true;
    }
```

```
Message::Quit => {
                self.quit();
            }
```

strings4.rs

In this exercise, we're provided with various values, some being string literals (`&str`) and others being owned strings (`String`). We need to determine the appropriate function to call (`string_slice` or `string`) for each value.

We define a function `string_slice` that takes a reference to a string slice (`&str`) as its argument.

```
fn string_slice(arg: &str) {
    println!("{}", arg);
}
```

We define another function `string` that takes ownership of a `String` as its argument.

```
fn string(arg: String) {
    println!("{}", arg);
}
```

Difference beetween string slice and strings :

String Slice (`&str`): is an immutable sequence of UTF-8 bytes of dynamic length somewhere in memory. Since the size is unknown, one can only handle it behind a pointer. This means that `str` most commonly appears as `&str`: a reference to some UTF-8 data, normally called a "string slice" or just a "slice".

String (`String`): `String` is a growable, heap-allocated data structure that owns a string of UTF-8 characters. Like `Vec`: we use it when you need to own or modify your string data.

- `"blue"`: This is a string slice (string literal), so we call `string_slice`.
- `"red".to_string()`: This creates a new `String`, so we call `string`.
- `String::from("hi")`: Similarly, this creates a new `String`, so we call `string`.

- `"rust is fun!".to_owned()`: This also creates a new `String`, so we call `string`.
- `"nice weather".into()`: This converts the string literal into a `String`, so we call `string`.
- `format!("Interpolation {}", "Station")`: This macro creates a new `String`, so we call `string`.
- `&String::from("abc")[0..1]`: This takes a reference to a substring, so we call `string_slice`.
- `" hello there ".trim()`: This returns a string slice, so we call `string_slice`.
- `"Happy Monday!".to_string().replace("Mon", "Tues")`: This creates a new `String`, so we call `string`.
- `"mY sHiFt KeY iS sTiCkY".to_lowercase()`: This creates a new `String`, so we call `string`.

Here is the final code :

```
fn string_slice(arg: &str) {
    println!("{}", arg);
}
fn string(arg: String) {
    println!("{}", arg);
}

fn main() {
    string_slice("blue");
    string("red".to_string());
    string(String::from("hi"));
    string("rust is fun!".to_owned());
    string("nice weather".into());
    string(format!("Interpolation {}", "Station"));
    string_slice(&String::from("abc")[0..1]);
    string_slice("  hello there ".trim());
    string("Happy Monday!".to_string().replace("Mon", "Tues")
    string("mY sHiFt KeY iS sTiCkY".to_lowercase());
}
```

module3.rs

The purpose of this exercise is to demonstrate how to use modules in Rust. More specifically the `SystemTime` struct and `UNIX_EPOCH` constant from the `std::time` module to calculate the duration since the Unix epoch and print the result.

The initial code attempts to use the `SystemTime` struct and `UNIX_EPOCH` constant, but it lacks the necessary imports, resulting in compilation errors.

```
use ???

fn main() {
    match SystemTime::now().duration_since(UNIX_EPOCH) {
        Ok(n) => println!("1970-01-01 00:00:00 UTC was {} sec
        Err(_) => panic!("SystemTime before UNIX EPOCH!"),
    }
}
```

We just had to add the required import statement `use std::time::{SystemTime, UNIX_EPOCH};` to bring the `SystemTime` struct and `UNIX_EPOCH` constant into scope.

```
use std::time::{SystemTime, UNIX_EPOCH};

fn main() {
    match SystemTime::now().duration_since(UNIX_EPOCH) {
        Ok(n) => println!("1970-01-01 00:00:00 UTC was {} sec
        Err(_) => panic!("SystemTime before UNIX EPOCH!"),
    }
}
```

Now we can calculate the duration since the Unix time (01/01/1970 00h00) and print the result

hashmap3.rs

We need to create a Rust program that processes a list of soccer match scores. Each score entry consists of two team names and the respective goals scored by each team. The program is tasked with building a scores table, which contains the name of each team along with the number of goals they scored and conceded. To accomplish this, the program utilizes a HashMap to store the team names as keys and a custom Team struct to store the goals scored and conceded by each team.

```
use std::collections::HashMap;

// Define the structure to store the goal details of a team.
struct Team {
    goals_scored: u8,
    goals_conceded: u8,
}

// Define the function to build the scores table.
fn build_scores_table(results: String) -> HashMap<String, Tea
    let mut scores: HashMap<String, Team> = HashMap::new();

    // Iterate over each line of the results.
    for r in results.lines() {
        // Split each line into parts using commas as delimit
        let v: Vec<&str> = r.split(',').collect();
        // Extract team names and scores.
        let team_1_name = v[0].to_string();
        let team_1_score: u8 = v[2].parse().unwrap();
        let team_2_name = v[1].to_string();
        let team_2_score: u8 = v[3].parse().unwrap();
        // TODO: Populate the scores table with details extra
    }
    scores
}
```

The `build_scores_table` function is responsible for processing the input results and populating the scores table. However, the function is incomplete. The provided tests validate the correctness of the build_scores_table function by ensuring that it correctly constructs the scores table.

To complete the solution, we need to populate the scores table with the details extracted from each line of the results.

```
for r in results.lines() {
```

This loop iterates through each line of the provided results string.

```
    let v: Vec<&str> = r.split(',').collect();
```

Here, each line is split into parts using the `split` method, which splits the string into substrings using a delimiter (`,` in this case), returning them as a vector.

```
    let team_1_name = v[0].to_string();
    let team_1_score: u8 = v[2].parse().unwrap();
    let team_2_name = v[1].to_string();
    let team_2_score: u8 = v[3].parse().unwrap();
```

The split parts are then used to extract the team names and scores. `to_string()` is used to convert the `&str` slices into owned `String` instances. The scores are parsed into `u8` using the `parse` method.

```
    let team_1_goals = scores.entry(team_1_name.clone()).or_i
    team_1_goals.goals_scored += team_1_score;
    team_1_goals.goals_conceded += team_2_score;

    let team_2_goals = scores.entry(team_2_name.clone()).or_i
    team_2_goals.goals_scored += team_2_score;
    team_2_goals.goals_conceded += team_1_score;
```

This part updates the goals scored and conceded for both teams. The `entry` method is used to either insert a new team or update an existing one in the HashMap. The scores are then updated accordingly for each team.

quizz2.rs

The purpose of this exercise is to implement a function called `transformer` that processes a vector of tuples containing strings and commands. Each command specifies an action to be performed on the corresponding string. The actions can be:

- Uppercasing the string

- Trimming the string
- Appending "bar" to the string a specified number of times

```
pub enum Command {
    Uppercase,
    Trim,
    Append(usize),
}
```

To solve the exercise, we have to import the `transformer` function from the `my_module` module into the scope of the test module. It allows the test module to call and test the `transformer` function.

```
use super::my_module::transformer;
```

The `transformer` function initializes an empty vector `output` to store the modified strings and it will iterates over each tuple `(string, command)` in the input vector using `input.iter()`.

```
let mut output: Vec<String> = vec![];
    for (string, command) in input.iter() {
```

For each tuple, it matches the command and performs these actions on the string:

If the command is `Uppercase`, it converts the string to uppercase using `string.to_uppercase()`.

If the command is `Trim`, it trims leading and trailing whitespace from the string using `string.trim().to_string()`.

If the command is `Append`, it appends "bar" to the string a specified number of times using `format!("{}{}", string, "bar".repeat(*u))`.

```
let string_to_modify = match command {
            Command::Uppercase => string.to_uppercase(),
            Command::Trim => string.trim().to_string(),
            Command::Append(u) => format!("{}{}", string, "ba
        };
```

Finally, the function pushes the modified string `string_to_modify` to the `output` vector using `output.push(string_to_modify)` and return `output`

```
pub fn transformer(input: Vec<(String, Command)>) -> Vec<Stri
    let mut output: Vec<String> = vec![];
    for (string, command) in input.iter() {
        let string_to_modify = match command {
            Command::Uppercase => string.to_uppercase(),
            Command::Trim => string.trim().to_string(),
            Command::Append(u) => format!("{}{}", string, "ba
        };
        output.push(string_to_modify);
    }
    output
}
```

option3.rs

```
match &y {
    Some(p) => println!("Co-ordinates are {},{} ", p.x, p.y),
    _ => panic!("no match!"),
}
```

In the original code, the `y` variable was directly matched with `Some(p)` which consumes `y`, making it unavailable for further use. To retain access to `y` after the match, a reference to `y` is used instead by matching on `&y`. This way, ownership is not transferred, and `y` remains accessible in the subsequent code.

Final code :

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let y: Option<Point> = Some(Point { x: 100, y: 200 });
```

```
    match &y {
        Some(p) => println!("Co-ordinates are {},{} ", p.x, p
        _ => panic!("no match!"),
    }
    y; // Fix without deleting this line.
}
```

error6.rs

The exercise aims to demonstrate the importance of defining custom error types in library code, allowing callers to make decisions based on the error content rather than simply printing or propagating errors further.

The initial code defines a custom error type `ParsePosNonzeroError`, which encompasses errors occurring during the creation of a value and errors encountered during integer parsing. However, it lacks a conversion function for `ParseIntError` to `ParsePosNonzeroError`.

Added Conversion Function for `ParseIntError`:

```
fn from_parseint(err: ParseIntError) -> ParsePosNonzeroError
    ParsePosNonzeroError::ParseInt(err)
}
```

This function is implemented as an associated function of the `ParsePosNonzeroError` enum.

It takes a `ParseIntError` as input and returns a `ParsePosNonzeroError`.

Inside the function, `ParseIntError` is wrapped in the `ParsePosNonzeroError::ParseInt` variant, allowing for uniform error handling within the `ParsePosNonzeroError` enum.

Modified Error Handling in `parse_pos_nonzero`:

```
fn parse_pos_nonzero(s: &str) -> Result<PositiveNonzeroIntege
    let x: Result<i64, ParseIntError> = s.parse();
    let x = match x {
        Ok(value) => value,
        Err(err) => return Err(ParsePosNonzeroError::from_par
    };
```

```
    PositiveNonzeroInteger::new(x).map_err(ParsePosNonzeroErr
}
```

The function signature remains unchanged, still returning a `Result` containing either a `PositiveNonzeroInteger` or a `ParsePosNonzeroError`.

The parsing operation (`s.parse()`) now explicitly specifies the result type as `Result<i64, ParseIntError>`. This means that upon successful parsing, the result will contain an integer value (`i64`), and in case of parsing failure, it will contain a `ParseIntError`.

In the match statement, if parsing succeeds (`Ok(value)`), the parsed value is extracted and stored in `x`. If parsing fails (`Err(err)`), the function immediately returns an error, converting the `ParseIntError` into a `ParsePosNonzeroError` using the `from_parseint` conversion function.

After handling the parsing error, the function attempts to create a `PositiveNonzeroInteger` from the parsed value (`x`). If successful, it returns the `PositiveNonzeroInteger`. If an error occurs during creation, it is mapped to a `ParsePosNonzeroError` using the `from_creation` conversion function.

Full code before the non-modification line :

```
use std::num::ParseIntError;
use std::str::FromStr;

#[derive(PartialEq, Debug)]
enum ParsePosNonzeroError {
    Creation(CreationError),
    ParseInt(ParseIntError),
}

impl ParsePosNonzeroError {
    fn from_creation(err: CreationError) -> ParsePosNonzeroEr
        ParsePosNonzeroError::Creation(err)
    }

    fn from_parseint(err: ParseIntError) -> ParsePosNonzeroEr
        ParsePosNonzeroError::ParseInt(err)
    }
}
```

```
fn parse_pos_nonzero(s: &str) -> Result<PositiveNonzeroIntege
    let x: Result<i64, ParseIntError> = s.parse();
    let x = match x {
        Ok(value) => value,
        Err(err) => return Err(ParsePosNonzeroError::from_par
    };
    PositiveNonzeroInteger::new(x).map_err(ParsePosNonzeroErr
}
```

generics2.rs

he purpose of this exercise is to enhance the flexibility and reusability of the `Wrapper` struct by converting it to use generics. By using generics, the `Wrapper` struct will be able to wrap and store values of any type, rather than being limited to only storing `u32` values as in the initial code.

Initial code :

```
struct Wrapper {
    value: u32,
}

impl Wrapper {
    pub fn new(value: u32) -> Self {
        Wrapper { value }
    }
}
```

Added Generic Type Parameter to `Wrapper` Struct:

```
struct Wrapper<T> {
    value: T,
}
```

The `Wrapper` struct is modified to include a type parameter `T`. This type parameter `T` represents the type of the value that the `Wrapper` struct will contain. It is a generic type, meaning it can be any valid Rust type.

Modified Constructor Method to Accept Generic Type:

```
impl<T> Wrapper<T> {
    pub fn new(value: T) -> Self {
        Wrapper { value }
    }
}
```

The `impl` block for `Wrapper` now includes the generic type parameter `<T>` to indicate that the methods within this block are associated with the generic version of the `Wrapper` struct.

The `new` method signature is updated to accept a value of type `T` instead of `u32`.

Inside the `new` method, the `value` field of the `Wrapper` struct is initialized with the provided `value`, which can be of any type `T`.

traits5.rs

The exercise aims to demonstrate the use of traits and generics in Rust. The task is to modify the given code to ensure it compiles correctly while adhering to the constraints provided.

The initial code defines two traits, `SomeTrait` and `OtherTrait`, each with a single method. Additionally, two empty structs, `SomeStruct` and `OtherStruct`, are declared. The code then implements both traits for both structs. Finally, there's a function `some_func` that currently accepts an unspecified type parameter.

```
fn some_func<T: SomeTrait + OtherTrait>(item: T) -> bool {
    item.some_function() && item.other_function()
}
```

The `some_func` function now has a generic type parameter `T`. This type parameter `T` is bounded by the traits `SomeTrait` and `OtherTrait` using the `+` syntax.

By making `some_func` generic, it can accept any type that implements both `SomeTrait` and `OtherTrait`. This ensures that the function can work with a wide range of types, as long as they support the required trait methods.

quiz3.rs

Initially, the system only supports numeric grades, but it needs to be extended to handle alphabetical grades as well.

The initial code defines a `ReportCard` struct with fields for the student's grade, name, and age. It provides a method `print` to generate a report card string. Two tests are provided, one for generating a numeric report card and another for an alphabetic report card.

This is the code modified :

```
use std::fmt::Display;

pub struct ReportCard<T: Display> {
    pub grade: T,
    pub student_name: String,
    pub student_age: u8,
}

impl<T: Display> ReportCard<T> {
    pub fn print(&self) -> String {
        format!("{} ({}) - achieved a grade of {}", &self.stu
    }
}
```

The `ReportCard` struct was modified to accept a generic type `T`. The type `T` is bounded by the `Display` trait, allowing any type that can be displayed as a string to be used as the `grade`.

The implementation block for `ReportCard` now specifies `T` as a generic type. All methods within this block are updated to work with the generic type `T`.

The `Display` trait bound is applied to `T`, ensuring that any type used for the `grade` field can be converted to a string for printing.

```
fn generate_alphabetic_report_card() {
        // TODO: Make sure to change the grade here after you
        let report_card = ReportCard {
            grade: "A+".to_string(),
            student_name: "Gary Plotter".to_string(),
```

```
            student_age: 11,
        };
        assert_eq!(
            report_card.print(),
            "Gary Plotter (11) - achieved a grade of A+"
        );
    }
```

In the test function `generate_alphabetic_report_card`, the grade is changed to `"A+"`, a string, to demonstrate the support for alphabetical grades. This adjustment confirms that the changes made to the `ReportCard` implementation successfully enable the system to generate report cards with alphabetical grades.

lifetime3.rs

The purpose of this exercise is to demonstrate the importance of lifetimes in Rust, particularly when structs hold references. By modifying the `Book` struct to include lifetime annotations, we ensure that references to strings (`author` and `title`) within the `Book` struct have a valid duration.

Initial code :

```
struct Book {
    author: &str,
    title: &str,
}

fn main() {
    let name = String::from("Jill Smith");
    let title = String::from("Fish Flying");
    let book = Book { author: &name, title: &title };

    println!("{} by {}", book.title, book.author);
}
```

Changes made :

```
struct Book<'a> {
    author: &'a str,
```

```
    title: &'a str,
}
```

Lifetimes are introduced to ensure that references in the `Book` struct have a valid duration. The `'a` syntax denotes a lifetime parameter, indicating that both `author` and `title` references in the `Book` struct have the same lifetime.

The `Book` struct is annotated with a lifetime parameter `'a`, which is applied to both the `author` and `title` fields. This annotation specifies that the references stored in `author` and `title` must live for at least as long as the lifetime `'a`.

The references `author` and `title` are now tied to the lifetime `'a`, ensuring they remain valid as long as the `Book` struct is in scope.

In the `main` function, a `Book` instance is created with references to `name` and `title`. The lifetime parameter `'a` ensures that the references in `author` and `title` remain valid throughout the scope of `book`.

test4.rs

In this exercise, we are tasked with writing tests for a `Rectangle` struct that ensures correct behavior when creating rectangles with positive dimensions and panics when attempting to create rectangles with negative dimensions.

```
#[test]
fn correct_width_and_height() {
    // This test should check if the rectangle is the size th
    let rect = Rectangle::new(10, 20);
    assert_eq!(rect.width, 10); // check width
    assert_eq!(rect.height, 20); // check height
}
```

In the `correct_width_and_height`, we test if the `Rectangle` is created with the correct width and height. To achieve this, we construct a `Rectangle` instance using the `Rectangle::new` constructor with known positive dimensions (e.g., 10 and 20). We use the `assert_eq!` macro to compare the width and height fields of the created `Rectangle` instance with the expected values (10 for width and 20 for height).

```
#[test]
#[should_panic]
```

```
fn negative_width() {
    // This test should check if program panics when we try t
    let _rect = Rectangle::new(-10, 10);
}
```

In the function `negative_width` we test if the program panics when attempting to create a `Rectangle` with a negative width. To do this, we use the `should_panic` attribute on the test function to indicate that it is expected to panic.

Inside the test, we call the `Rectangle::new` constructor with a negative width value (e.g., -10). If the program panics as expected due to a negative width, the test passes; otherwise, it fails.

```
#[test]
#[should_panic]
fn negative_height() {
    // This test should check if program panics when we try t
    let _rect = Rectangle::new(10, -10);
}
```

In the `negative_height` function, similar to the negative width test, we test if the program panics when attempting to create a `Rectangle` with a negative height. We use the `should_panic` attribute on the test function to indicate that it is expected to panic.

Inside the test, we call the `Rectangle::new` constructor with a negative height value (e.g., -10). If the program panics as expected due to a negative height, the test passes; otherwise, it fails.

iterator5.rs

We're given a model to track exercise progress, where the name of the exercise is the key and the progress is the value (represented by an enum `Progress`).

We need to count the number of exercises with a given progress (`None`, `Some`, or `Complete`).

Two counting functions are provided:

`count_for`: This function uses imperative loops to count the progress.

`count_collection_for`: This function counts progress across a collection of hashmaps using imperative loops.

```
fn count_for(map: &HashMap<String, Progress>, value: Progress
    let mut count = 0;
    for val in map.values() {
        if val == &value {
            count += 1;
        }
    }
    count
}

fn count_collection_for(collection: &[HashMap<String, Progres
    let mut count = 0;
    for map in collection {
        for val in map.values() {
            if val == &value {
                count += 1;
            }
        }
    }
    count
}
```

We need to implement two counting functions (`count_iterator` and `count_collection_iterator`) using iterators instead of imperative loops and the solution should count progress efficiently and produce the same results as the provided counting functions.

```
fn count_iterator(map: &HashMap<String, Progress>, value: Pro
    map.values().filter(|&&progress| progress == value).count
}
```

For `count_iterator` we iterate over hashmap values using `map.values()` then filter the values that match the given progress using `filter` and count the filtered values using `count`.

```
fn count_collection_iterator(collection: &[HashMap<String, Pr
    collection
        .iter()
        .map(|map| count_iterator(map, value))
        .sum()
}
```

For the `count_collection_iterator` function we iterate over the collection of hashmaps using `collection.iter()`.

For each hashmap, apply `count_iterator` to count progress and sum up the counts using `sum`.

rc1.rs

The purpose of this exercise is to demonstrate the use of reference counting (Rc) in Rust, specifically to model multiple owners of a shared resource. The initial code defines a solar system simulation where planets take ownership of the sun using Rc pointers. The goal is to ensure that the sun has multiple owners without causing memory leaks or ownership issues.

This is the part of the code we had to change to make it works :

```
// TODO
    let saturn = Planet::Saturn(Rc::new(Sun {}));
    println!("reference count = {}", Rc::strong_count(&sun));
    saturn.details();

    // TODO
    let uranus = Planet::Uranus(Rc::new(Sun {}));
    println!("reference count = {}", Rc::strong_count(&sun));
    uranus.details();

    // TODO
    let neptune = Planet::Neptune(Rc::new(Sun {}));
    println!("reference count = {}", Rc::strong_count(&sun));
    neptune.details();
```

How did we approched it :

We introduced `Rc` pointers to the `Planet` enum variants to indicate multiple ownership of the `Sun`. In each variant of the `Planet` enum, we cloned the reference to the `Sun` using `Rc::clone` to increment the reference count.

We dropped the planets in reverse order of their creation to decrement the reference count and ensure proper cleanup.

```
// TODO
    let saturn = Planet::Saturn(Rc::clone(&sun));
    println!("reference count = {}", Rc::strong_count(&sun));
    saturn.details();

    // TODO
    let uranus = Planet::Uranus(Rc::clone(&sun));
    println!("reference count = {}", Rc::strong_count(&sun));
    uranus.details();

    // TODO
    let neptune = Planet::Neptune(Rc::clone(&sun));
    println!("reference count = {}", Rc::strong_count(&sun));
    neptune.details();
```

In Rust, the `Rc` (Reference Counting) smart pointer allows multiple ownership of data by keeping track of the number of references to a value. In the final code, each `Planet` variant holds an `Rc` pointer to the `Sun`, indicating shared ownership. By using `Rc::clone`, we create additional references to the `Sun`, incrementing the reference count each time a planet is created. When a planet is dropped, the reference count decreases, and the memory is deallocated when the reference count reaches zero, ensuring memory safety and preventing memory leaks.

threads3.rs

⚠⚠ This correction is not mine. 😔 I wasn't able to find it myself but i'll try my best to explain it to you how he did it ! ⚠⚠

The purpose of this exercise is to demonstrate the utilization of multi-threading in Rust with the help of channels. Specifically, we are tasked with splitting a queue into two halves and sending each half concurrently through separate threads. Afterwards, we need to receive the values from these threads through a channel and assert that all values were received.

The genius of the solve is here :

```
fn send_tx(q: Queue, tx: mpsc::Sender<u32>) -> () {
    let qc = Arc::new(q);
    let qc1 = Arc::clone(&qc);
    let qc2 = Arc::clone(&qc);


    thread::spawn(move || {
        for val in &qc1.first_half {
            println!("sending {:?}", val);
            tx.send(*val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
```

```
fn send_tx(q: Queue, tx: mpsc::Sender<u32>) -> () {
    let qc = Arc::new(q);
    let qc1 = Arc::clone(&qc);
    let qc2 = Arc::clone(&qc);
    let tx1 = tx.clone();

    thread::spawn(move || {
        for val in &qc1.first_half {
            println!("sending {:?}", val);
            tx1.send(*val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
```

To the left the inital code. To the right the modified code to make the exercise work.

In Rust, when you pass a variable to a closure that will be executed on a different thread, it needs to implement the `Send` trait to allow it to be sent safely to other threads. This is because Rust ensures memory safety and thread safety at compile time through its ownership system.

In the initial code, the closure passed to `thread::spawn` captures the `tx` sender channel. However, since `tx` is moved into the closure, Rust no longer allows us to use `tx` afterward because it's considered "moved" and can't be accessed anymore in the main thread.

To overcome this limitation, he uses `Arc` (atomic reference counting) to create a reference-counted pointer to the `Queue` struct, allowing multiple ownership across threads. However, we still need to ensure that each thread has its own sender channel to communicate with the receiving end without causing data races or ownership conflicts.

```
let tx1 = tx.clone();
```

By calling `clone()` on the `tx` sender channel, he creates a new instance of the sender channel `tx1`. This new instance can be safely passed to the closure because it's a separate, independent copy of the original sender channel. Both `tx` and `tx1` now have their own ownership, allowing them to be used independently in separate threads.

And then, modifying the `tx.send(*val).unwrap();` line to `tx1.send(*val).unwrap();` ensures that each thread uses its own sender channel (`tx1`) to send values. This prevents ownership conflicts and ensures thread safety, as each thread now has exclusive access to its own sender channel for communication.

In summary, by cloning the sender channel and using separate instances (`tx` and `tx1`) in each thread, he ensures that each thread can safely send values without worrying about ownership issues or data races, thus achieving thread safety and preventing potential runtime errors.

🤓🤓🤓🤓

macros4.rs

This exercise involves fixing a syntax error in a given macro definition to ensure that it compiles successfully and behaves as intended when invoked.

The initial code provides a macro named `my_macro`, which is defined using the `macro_rules!` macro provided by Rust. This macro has two rules: one for an empty invocation and another for an invocation with an expression. However, the macro definition is missing semicolons at the end of each rule, resulting in a syntax error.

The primary change made was to correct the syntax error in the macro definition. In Rust, macros require semicolons (`;`) at the end of each rule to terminate them properly. Without these semicolons, the compiler would encounter a syntax error.

```
#[rustfmt::skip]
macro_rules! my_macro {
    () => {
        println!("Check out my macro!");
    };
    ($val:expr) => {
        println!("Look at this other macro: {}", $val);
    };
}
```

In the corrected version, each rule in the macro definition is now terminated with a semicolon (`;`). This ensures that each rule is properly delineated, allowing the macro parser to identify and process them correctly.

The usage of the macro in the `main` function remains unchanged.

```
fn main() {
    my_macro!();
```

```
    my_macro!(7777);
}
```

clippy3.rs

In this exercise, the goal is to apply fixes to the initial code provided, addressing several common issues that Clippy can detect.

Original code :

```
#[allow(unused_variables, unused_assignments)]
fn main() {
    let my_option: Option<()> = None;
    if my_option.is_none() {
        my_option.unwrap();
    }

    let my_arr = &[
        -1, -2, -3
        -4, -5, -6
    ];
    println!("My array! Here it is: {:?}", my_arr);

    let my_empty_vec = vec![1, 2, 3, 4, 5].resize(0, 5);
    println!("This Vec is empty, see? {:?}", my_empty_vec);

    let mut value_a = 45;
    let mut value_b = 66;
    // Let's swap these two!
    value_a = value_b;
    value_b = value_a;
    println!("value a: {}; value b: {}", value_a, value_b);
}
```

The changes made are removing the unnecessary call to `unwrap()` on the `Option` type, as it can cause a panic if the value is `None`. Since the value is not used afterward, the call is simply omitted.

I corrected the formatting of the array initialization by adding a comma after each element and properly aligning the elements on separate lines. This improves readability and prevents potential syntax errors.

I changed the initialization of the empty vector `my_empty_vec` by calling `Vec::new()` instead of `vec![].resize()`. This creates an empty vector directly without unnecessary resizing operations.

And I replaced the manual variable swapping with the safer `std::mem::swap()` function, which exchanges the values of two mutable references. This ensures correct behavior and prevents potential bugs associated with manual swapping.

Final code :

```
#[allow(unused_variables, unused_assignments)]
fn main() {
    let my_option: Option<()> = None;
    if my_option.is_none() {
    }

    let my_arr = &[
        -1, -2, -3,
        -4, -5, -6
    ];
    println!("My array! Here it is: {:?}", my_arr);

    let my_empty_vec: Vec<i32> = Vec::new();
    println!("This Vec is empty, see? {:?}", my_empty_vec);

    let mut value_a = 45;
    let mut value_b = 66;
    std::mem::swap(&mut value_a, &mut value_b);
    println!("value a: {}; value b: {}", value_a, value_b);
}
```

using_as.rs

This exercise is here to demonstrate the importance of handling data types properly and ensuring consistent type conversions to avoid potential errors and ensure the correctness of computations.

```
fn average(values: &[f64]) -> f64 {
    let total = values.iter().sum::<f64>();
    total / values.len()
}

fn main() {
    let values = [3.5, 0.3, 13.0, 11.7];
    println!("{}", average(&values));
}
```

The initial code defines a function named `average` that calculates the average of a slice of `f64` values. It then calls this function in the `main` function, passing an array of `f64` values to calculate and print the average.

In the final code, a modification is made to ensure proper type conversion. Specifically, the length of the slice is converted to a `f64` type before performing the division operation. This ensures that the division operation results in a floating-point value, which is consistent with the calculation of the total sum. Without this conversion, there could be a mismatch in data types, leading to unexpected results or compiler errors.

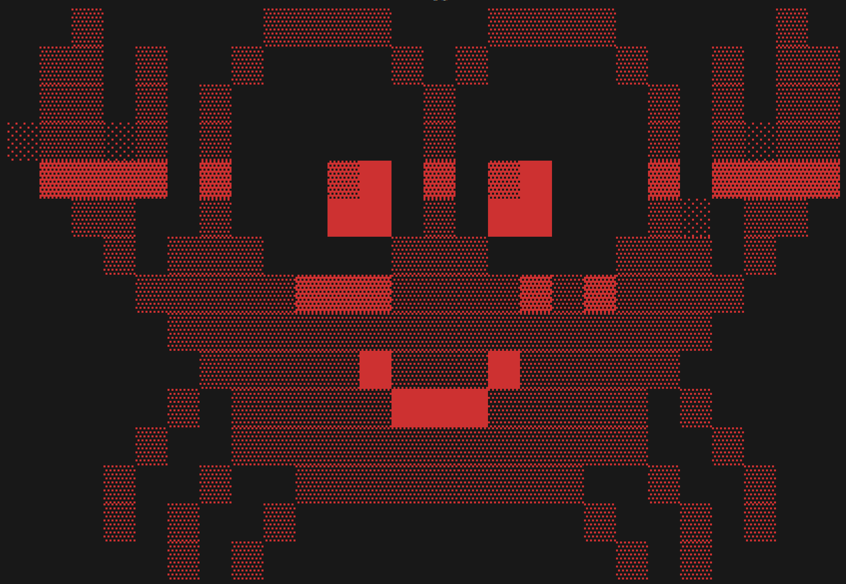
```
fn average(values: &[f64]) -> f64 {
    let total = values.iter().sum::<f64>();
    total / values.len() as f64
}

fn main() {
    let values = [3.5, 0.3, 13.0, 11.7];
    println!("{}", average(&values));
}
```

The change made in the final code involves explicitly converting the length of the slice (`values.len()`) to a `f64` type using the `as` keyword. By converting the length to a `f64` type, we ensure that the division operation (`total / values.len() as f64`) operates on compatible data types.

```
Progress: [##############################################
🎉 All exercises completed! 🎉

+--------------------------------------------------+
|          You made it to the Fe-nish line!        |
+--------------------------  ----------------------+
                            \/
```



```
We hope you enjoyed learning about the various aspects of
Rust!
If you noticed any issues, please don't hesitate to report
 them to our repo.
You can also contribute your own exercises to help the gre
ater community!

Before reporting an issue or contributing, please read our
 guidelines:
https://github.com/rust-lang/rustlings/blob/main/CONTRIBUT
ING.md
```