Name: Sushant Yadav
Reg No: Ch.en.u4cys22067
Course Code: 20cys312
Date: 2025/03/22

Implementing Structured Error Handling in File I/O

Write a Rust program that does the following:

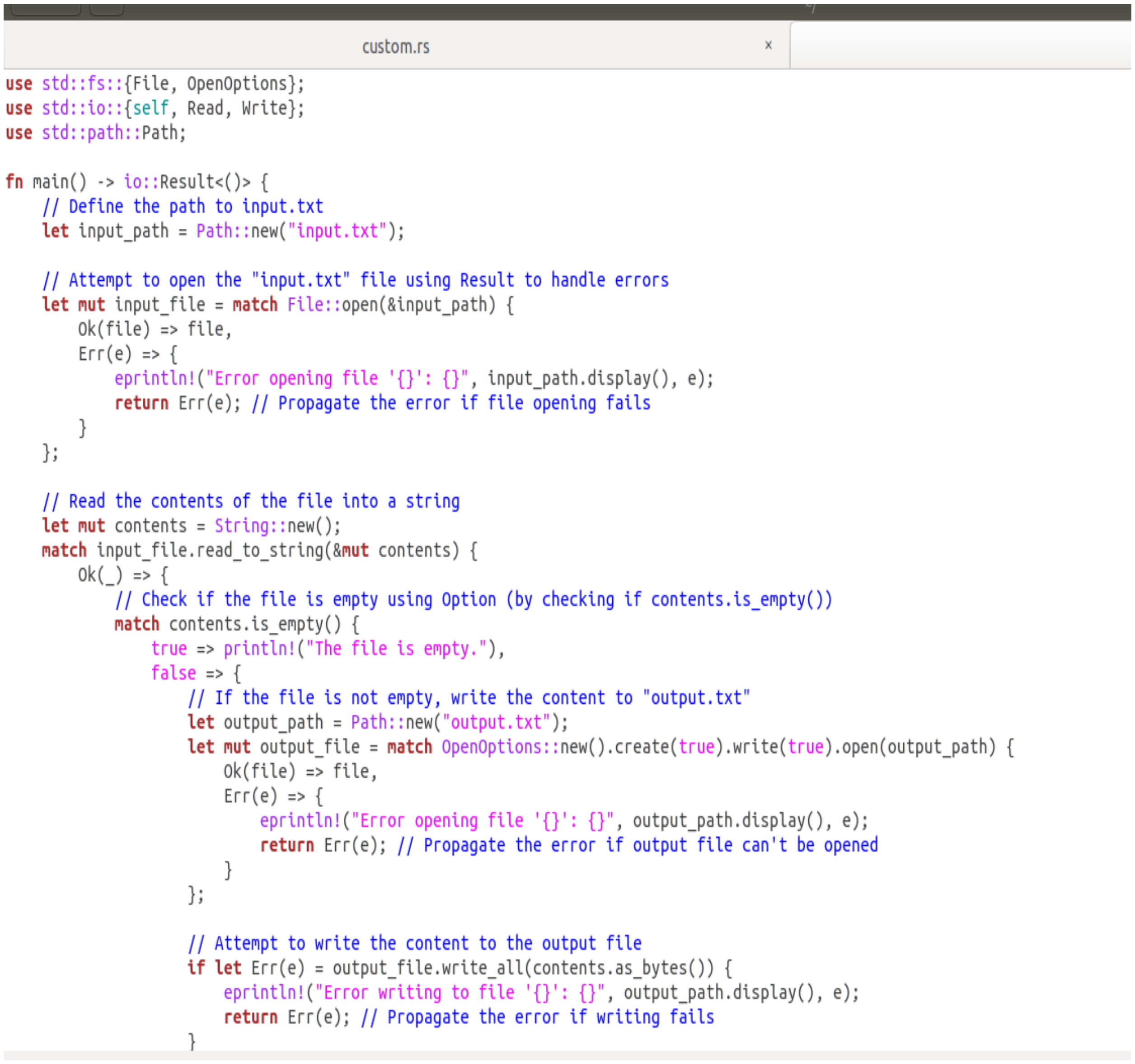
1. **Reads** the contents of a file named "input.txt".
2. **Handles possible errors** (file not found, permission denied, etc.) using Result<T, E>.
3. **Writes** the content to a new file named "output.txt".
4. Uses Option<T> to check if the file is empty and prints an appropriate message.

The objectives of the program are:

1. **Read File Contents**: Open and read the contents of input.txt.
2. **Handle Errors**: Use Result<T, E> for proper error handling (e.g., file not found, permission issues).
3. **Check if File is Empty**: Use Option<T> to determine if the file is empty and display a message.
4. **Write to New File**: Write the contents to output.txt.
5. **Display Clear Errors**: Provide informative error messages for any issues encountered.

6. **Propagate Errors Safely**: Use return Err(e) to propagate errors when necessary.
7. **Structured Error Handling**: Demonstrate Rust's structured error handling using Result and Option.

Code:



custom.rs

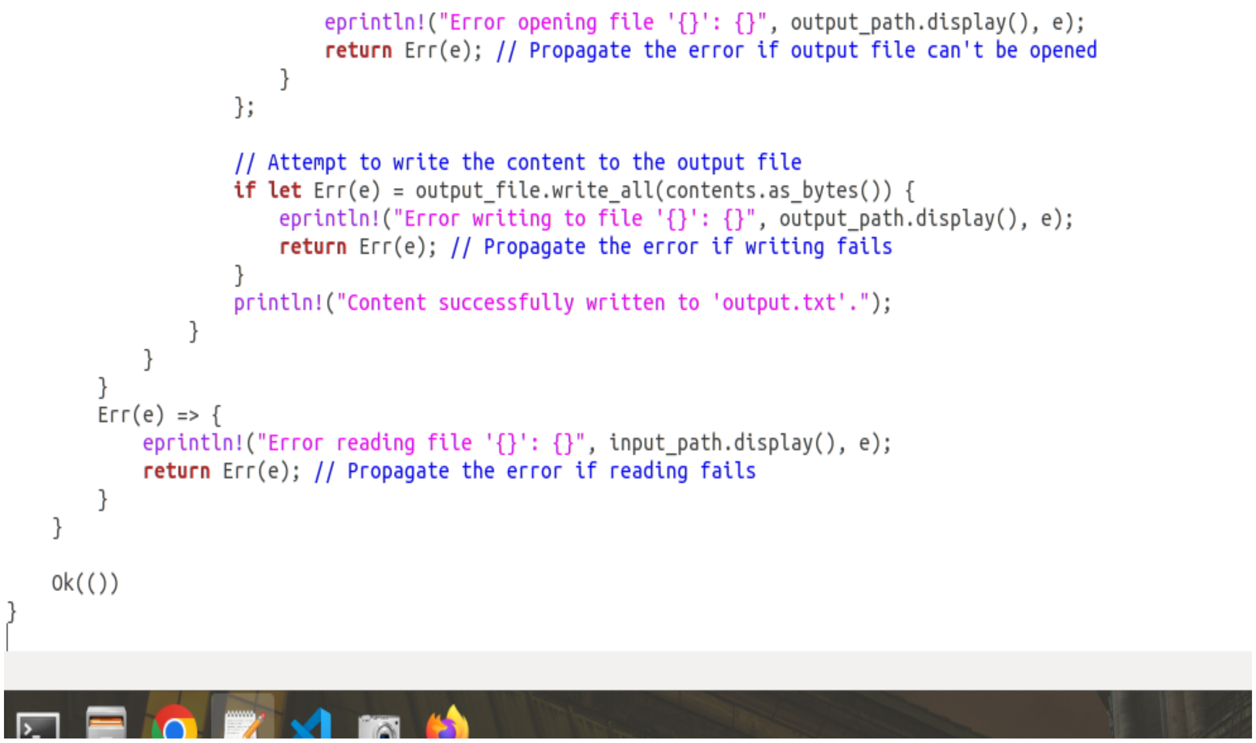
```rust
use std::fs::{File, OpenOptions};
use std::io::{self, Read, Write};
use std::path::Path;

fn main() -> io::Result<()> {
    // Define the path to input.txt
    let input_path = Path::new("input.txt");

    // Attempt to open the "input.txt" file using Result to handle errors
    let mut input_file = match File::open(&input_path) {
        Ok(file) => file,
        Err(e) => {
            eprintln!("Error opening file '{}': {}", input_path.display(), e);
            return Err(e); // Propagate the error if file opening fails
        }
    };

    // Read the contents of the file into a string
    let mut contents = String::new();
    match input_file.read_to_string(&mut contents) {
        Ok(_) => {
            // Check if the file is empty using Option (by checking if contents.is_empty())
            match contents.is_empty() {
                true => println!("The file is empty."),
                false => {
                    // If the file is not empty, write the content to "output.txt"
                    let output_path = Path::new("output.txt");
                    let mut output_file = match OpenOptions::new().create(true).write(true).open(output_path) {
                        Ok(file) => file,
                        Err(e) => {
                            eprintln!("Error opening file '{}': {}", output_path.display(), e);
                            return Err(e); // Propagate the error if output file can't be opened
                        }
                    };

                    // Attempt to write the content to the output file
                    if let Err(e) = output_file.write_all(contents.as_bytes()) {
                        eprintln!("Error writing to file '{}': {}", output_path.display(), e);
                        return Err(e); // Propagate the error if writing fails
                    }
```



```
                    eprintln!("Error opening file '{}': {}", output_path.display(), e);
                    return Err(e); // Propagate the error if output file can't be opened
                }
            };

            // Attempt to write the content to the output file
            if let Err(e) = output_file.write_all(contents.as_bytes()) {
                eprintln!("Error writing to file '{}': {}", output_path.display(), e);
                return Err(e); // Propagate the error if writing fails
            }
            println!("Content successfully written to 'output.txt'.");
        }
    }
}
Err(e) => {
    eprintln!("Error reading file '{}': {}", input_path.display(), e);
    return Err(e); // Propagate the error if reading fails
}
}

Ok(())
}
```

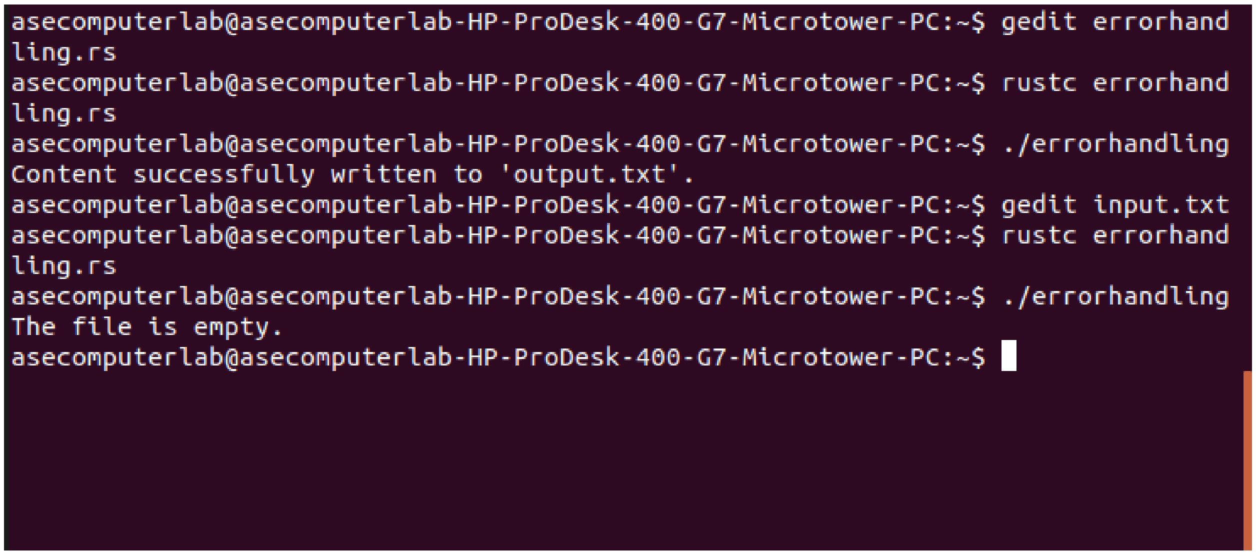
Explanation

1. **Opening the Input File**:
   a. We use File::open() to attempt to open the file input.txt. This returns a Result<File, std::io::Error>. If successful, we proceed to read the file. If there's an error (like the file not being found), we handle it with a match expression and return the error.
2. **Reading File Contents**:
   a. We use the read_to_string() method to read the contents of the file into a String. The read_to_string() method returns a Result<usize, std::io::Error>, which we also handle with a match statement. If an error occurs during reading, we print an error message and propagate the error using return Err(e).
3. **Checking if the File is Empty**:

    a. We check whether the contents of the file are empty using contents.is_empty(). If the file is empty, a message "The file is empty." is printed. If it's not empty, we proceed to write the content to output.txt.

4. **Opening the Output File**:
    a. We use OpenOptions::new().create(true).write(true).open() to open or create the output.txt file for writing. This allows us to create a new file or overwrite an existing one. We handle any errors with a match expression.
5. **Writing to the Output File**:
    a. We use write_all() to write the content to the output file. If an error occurs during writing, we print an error message and return the error.
6. **Error Handling**:
    a. Rust's Result<T, E> type is used throughout the program to handle various errors (file not found, permission denied, etc.). We handle errors gracefully by printing error messages and propagating them using return Err(e) to exit the program when necessary.

Output:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit errorhand
ling.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc errorhand
ling.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./errorhandling
Content successfully written to 'output.txt'.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit input.txt
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc errorhand
ling.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./errorhandling
The file is empty.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Conclusion:

This program demonstrates structured error handling in Rust, effectively using the Result<T, E> type for managing potential I/O errors and the Option<T> type to handle cases where the file might be empty. The use of match ensures that all error cases are accounted for, and appropriate messages are printed, making the program robust and easy to debug.

In Rust, leveraging these types for error handling is a powerful way to write safe, reliable code, especially when dealing with file operations that can fail due to various reasons (e.g., missing files, permission issues). This example follows Rust's philosophy of explicit error handling, ensuring that errors are never ignored and can be properly managed.