20CYS312 – PRINCIPLES OF PROGRAMMING LANGUAGES

LAB EXERCISE 10

Name: Naren S
Roll No: CH.EN.U4CYS22034
Date: 21/03/2025
Year / Branch : 3rd / CYS

Lab Exercise 10: 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.

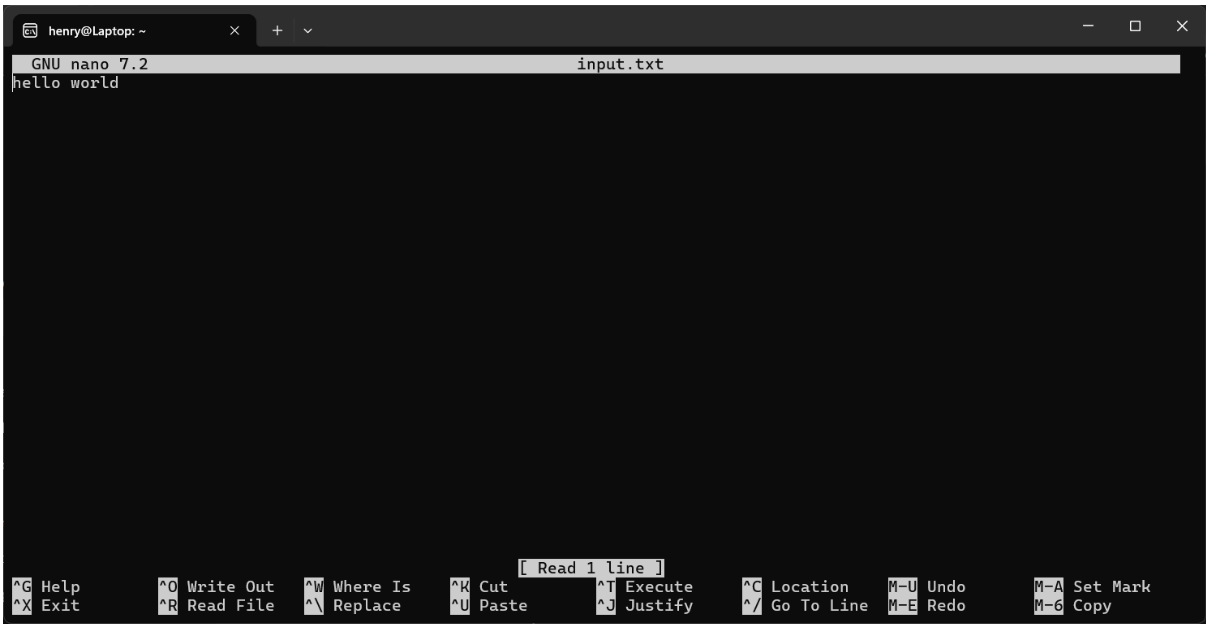
Objective

The objective of this lab exercise is to understand and implement **structured error handling** in **File I/O operations** using **Rust**. This includes handling potential errors (such as "file not found" or "permission denied"), safely reading from and writing to files, and using **Option<T>** to check if the file is empty.

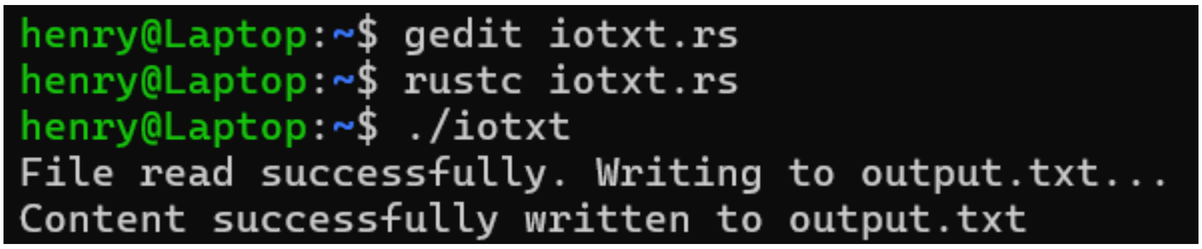
Code

```
use std::fs::File;
use std::io::{self, Read, Write};

fn main() {
    let input_path = "input.txt";
    let output_path = "output.txt";

    // Attempt to read the contents of "input.txt"
    match read_file(input_path) {
        Ok(content) => {
            if content.is_empty() {
                println!("The input file is empty.");
            } else {
                println!("File read successfully. Writing to output.txt...");
                if let Err(e) = write_file(output_path, &content) {
                    eprintln!("Error writing to output file: {}", e);
                } else {
                    println!("Content successfully written to output.txt");
                }
            }
        }
        Err(e) => eprintln!("Error reading the input file: {}", e),
    }
}

// Function to read the contents of a file
fn read_file(path: &str) -> Result<String, io::Error> {
    let mut file = File::open(path)?;
    let mut content = String::new();
    file.read_to_string(&mut content)?;
    Ok(content)
}

// Function to write content to a new file
fn write_file(path: &str, content: &str) -> Result<(), io::Error> {
    let mut file = File::create(path)?;
    file.write_all(content.as_bytes())?;
    Ok(())
}
```

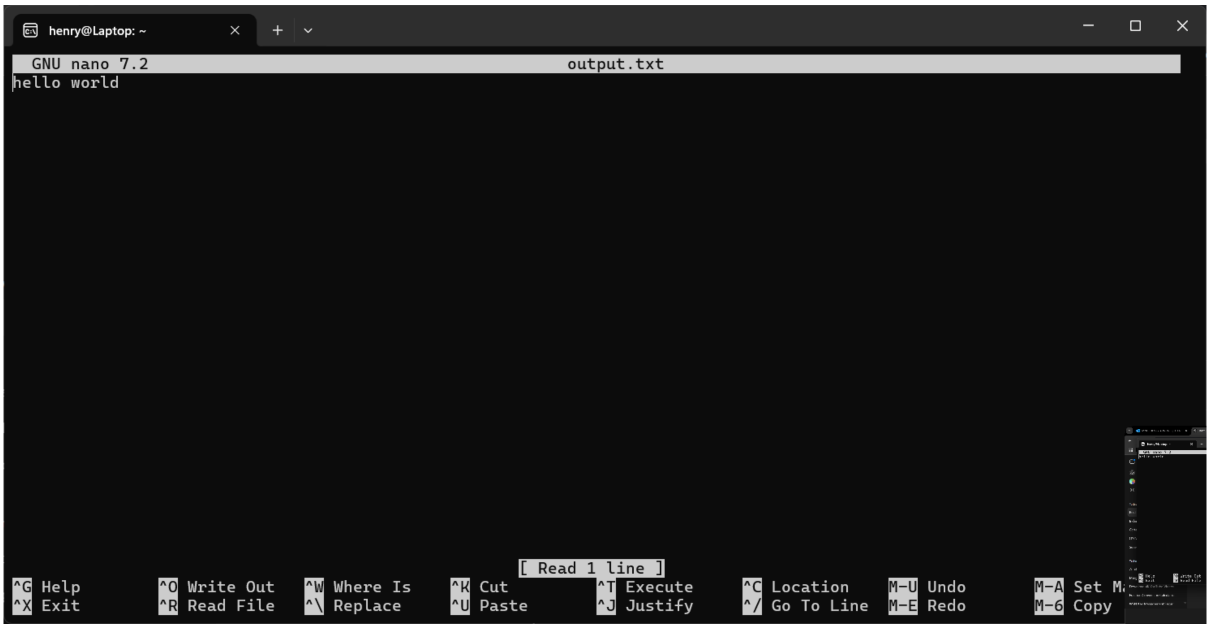
Output



```
henry@Laptop: ~
  GNU nano 7.2                                              input.txt
hello world

                                                    [ Read 1 line ]
^G Help          ^O Write Out     ^W Where Is      ^K Cut           ^T Execute       ^C Location      M-U Undo         M-A Set Mark
^X Exit          ^R Read File     ^\ Replace       ^U Paste         ^J Justify       ^/ Go To Line    M-E Redo         M-6 Copy
```



```
henry@Laptop:~$ gedit iotxt.rs
henry@Laptop:~$ rustc iotxt.rs
henry@Laptop:~$ ./iotxt
File read successfully. Writing to output.txt...
Content successfully written to output.txt
```



```
henry@Laptop: ~
  GNU nano 7.2                                              output.txt
hello world

                                                    [ Read 1 line ]
^G Help          ^O Write Out     ^W Where Is      ^K Cut           ^T Execute       ^C Location      M-U Undo         M-A Set M
^X Exit          ^R Read File     ^\ Replace       ^U Paste         ^J Justify       ^/ Go To Line    M-E Redo         M-6 Copy
```

Explanation

1. **Imports and Setup:**
    - fs, File, Read, and Write modules are imported for file operations.
    - We define input and output file paths.
2. **Main Function:**
    - Calls read_file function to read from input.txt.
    - If successful:
        - Checks if the file is **empty** using Option<T>.
        - If not empty, calls write_file to save the content to output.txt.
    - If an error occurs, it is printed with an appropriate message.
3. **read_file Function:**
    - Uses File::open() to open the file and handles errors.
    - Reads the content using read_to_string() and returns a Result<String, io::Error>.
4. **write_file Function:**
    - Uses File::create() to create the output file.
    - Writes the content and returns a Result<(), io::Error>.
5. **Error Handling:**
    - match expressions handle different outcomes:
        - **Ok(content)** – Process the file contents.
        - **Err(e)** – Print the error message with specific reasons (file not found, permission denied, etc.).

Conclusion

This program demonstrates how to:

- Use Result<T, E> to handle errors in file operations.
- Use Option<T> to check for empty content.
- Perform **robust** and **safe** file reading and writing in **Rust**.
- Handle **common file errors** like missing files or permission issues with detailed messages.