Rust: I/O

Mustakimur R. Khandaker

Introduction

There are four different popular I/O systems:

- Command-line Arguments (in this chapter)
- Standard I/O (in this chapter)
- File I/O (covered)
- Network I/O (in this chapter)

Command-line Argument

Introduction

To read the values of command line arguments, we'll use a function provided in Rust's standard library, which is std::env::args.

- The function returns an iterator of the command line arguments.
  - iterators produce a series of values, and call to the collect method on an iterator turn it into a collection (i.e. a vector of String).

Note that std::env::args will panic if any argument contains invalid Unicode.

- *So, if the program needs to accept arguments containing invalid Unicode, use std::env::args_os instead.*
  - *The function returns an iterator that produces OsString values instead of String values.*

Accessing Command-line Arguments

In the vector, the program's name takes up the first value in the vector at args[0]. So, to access first command-line argument, we would use args[1].

This will cause panic error. Valid index are from 0 to args.len()-1.

```
fn cmd_arg() {
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);

    let query = &args[1];
    let filename = &args[2];
    //let last = &args[args.len()];

    // recommendation
    /* if args.len() < 3 {
        panic!("Invalid number of arguments");
    } else {
        let query = args[1].clone();
        let filename = args[2].clone();
    } */

    println!("Searching for {}", query);
    println!("In file {}", filename);
}
```

Standard I/O

User Input

Rust std::io has standard functions to read user input from terminal. However, they only read them as String (same as command-line arguments).

- So, we depend on parse function to collect user data as we expected.
- Error handling is critical in such scenario.

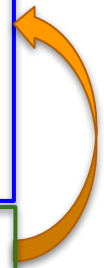
Alternatively, there are other crates that can lower our burden on handling user input.

```rust
fn std_io_01() -> io::Result<()> {
    let mut input = String::new();

    io::stdin().read_line(&mut input)?;
    //io::stdin().read_line(&mut input).unwrap();
    /* io::stdin()
    .read_line(&mut input)
    .expect("Failed to read line"); */

    println!("You typed: {}", input.trim());

    Ok(())
}
```

Parsing

```
fn std_io_02() {
    let mut input = String::new();

    io::stdin().read_line(&mut input).unwrap();

    let result = input.trim().parse();

    let u_int = match result {
        Ok(num) => num,
        Err(err) => {
            eprintln!("Please type a number!\nError
Message: {}", err);
            -1
        }
    };

    println!("Input number: {}", u_int);
}
```

```
fn std_io_03() -> io::Result<()> {
    let mut val = String::new();

    io::stdin().read_line(&mut val)?;
    let mut substr_iter = val.split_whitespace();
    let mut next_num = || -> usize {
        substr_iter
            .next()
            .expect("Not enough input numbers")
            .parse()
            .expect("Input is not a number")
    };
    let val1 = next_num();
    let val2 = next_num();

    println!("Values are {} and {}", val1, val2);

    Ok(())
}
```



User Output

There are two different types of user output.

- Regular user message:
  - We have so far uses println!() macros to format and print string in user terminal.
  - Alternative, we can also use std::io::stdout().write function.

```
use std::io::Write;
std::io::stdout().write("Tutorials ".as_bytes()).unwrap();
```

- Error message:
  - eprintln!() is an excellent macro to print error message.

```
let result = input.trim().parse();

let u_int = match result {
    Ok(num) => num,
    Err(err) => {
        eprintln!("Please type a number!\nError Message: {}", err);
        -1
    }
};
```

```
Standard input:
asd
Please type a number!
Error Message: invalid digit found in
string
```

Network I/O

TCP Connection

The standard library std::net offer supports for TCP connection.

Following code will listen at the address 127.0.0.1:7878 for incoming TCP stream.

- When it gets an incoming stream, it will print Connection established!.

```
fn tcp_conn_01() {
    let listener =
TcpListener::bind("127.0.0.1:7878").unwrap();
    match listener.accept() {
        Ok((_socket, addr)) => println!("new client:
{:?}", addr),
        Err(e) => println!("couldn't get client:
{:?}", e),
    }
}
```

```
fn tcp_conn_02() {
    let addrs = [
        SocketAddr::from(([127, 0, 0, 1], 7878)),
        SocketAddr::from(([127, 0, 0, 1], 7070)),
    ];
    let listener =
TcpListener::bind(&addrs[..]).unwrap();

    println!("Connected to: {}",
listener.local_addr().unwrap());

    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                println!("new client!");
            }
            Err(e) => { /* connection failed */ }
        }
    }
}
```

Reading from Listener

When TcpListener get connected, we should create a new thread for the new instance.

- The new thread can be used to receive and send messages through the TcpStream.

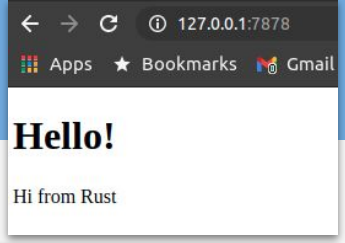
Crate std::io::prelude let us read from and write to the stream.

The TcpStream instance need to be maintained mutable.

- Because it keeps track of what data it returns to us internally. It might read more data.
- Call to stream.read to read bytes from the TcpStream and put them in a buffer.
  - The String::from_utf8_lossy function takes a &[u8] and produces a String from it.

```
fn tcp_conn_03() {
    let listener =
TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];

    stream.read(&mut buffer).unwrap();

    println!("Request: {}",
String::from_utf8_lossy(&buffer[..]));
}
```

Returning a Response



Responses have the following format:

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

The first line is a status line that contains the HTTP version used in the response, a numeric status code that summarizes the result of the request, and a reason phrase that provides a text description of the status code. After the CRLF sequence are any headers, another CRLF sequence, and the body of the response.

We call as_bytes on response to convert the string data to bytes.

- The write method on TcpStream takes a &[u8] and sends those bytes directly down the connection.

Finally, flush will wait and prevent the program from continuing until all the bytes are written to the connection.

- TcpStream contains an internal buffer to minimize calls to the underlying operating system.

```rust
fn handle_conn_rw(mut stream: TcpStream) {
    let mut buffer = [0; 1024];

    stream.read(&mut buffer).unwrap();

    let contents =
fs::read_to_string("content.html").unwrap();

    let response = format!(
        "HTTP/1.1 200 OK\r\nContent-Length:
{}\r\n\r\n{}",
        contents.len(),
        contents
    );

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

< Rust: I/O />