Error Handling

Performant Software Systems with Rust — Lecture 6

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

Two Kinds of Errors

- **Recoverable**: just report the problem to the user and retry the operation
  - Example: `File not found`
- **Unrecoverable**: Runtime bugs, need to terminate immediately
  - Example: accessing a location beyond the end of an collection

Rust Does Not Use Exceptions

- Exceptions in most other languages handle both kinds of errors the same way, using **exceptions**
- But our code typically has a lot more **recoverable** errors that should be handled in a better, more **graceful**, way
- For **unrecoverable** errors, we (or the Rust runtime) can just `panic!`, causing a runtime crash immediately

`Panic!`

- Caused by
  - Taking an action, such as accessing a collection past its end, that causes the code to panic
  - Calling `panic!` macro directly
- Prints a message, walks back the stack and cleans up the data from each function, and quits
  - `RUST_BACKTRACE=1 cargo run`

Live Demo

Recoverable Errors with `Result`

```
1 enum Result<T, E> {
2     Ok(T),
3     Err(E),
4 }
```

where `T` and `E` are generic type parameters

Example: Opening a File

```rust
use std::fs::File;

fn main() {
    // File::open returns `Result<T, E>`, where `T` is `std::fs::file`,
    // the file handle, and E is `std::io::Error`
    let greeting_file_result = File::open("llms.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {error:?}"),
    };
}
```

Live Demo

Shortcuts for Panic on Error: unwrap

```
1 use std::fs::File;
2
3 fn main() {
4     let greeting_file = File::open("llms.txt").unwrap();
5 }
```

Shortcuts for Panic on Error: expect

```rust
use std::fs::File;

fn main() {
    let greeting_file = File::open("llms.txt")
        .expect("llms.txt should be included in this project");
}
```

Propagating Errors

- A function runs into an **error**
- Instead of handling the **error** by itself, the function returns the **error** to the caller for it to decide what to do

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

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("llms.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
```

```
16          Err(e) ⇒ Err(e),
17      }
```

The ? Operator

```rust
fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("llms.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

```rust
fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();
    File::open("llms.txt")?.read_to_string(&mut username)?;
    Ok(username)
}
```

```rust
fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("llms.txt")
}
```

The ? Operator Also Works with Option<T> Values

```
1 fn last_char_of_first_line(text: &str) → Option<char> {
2     text.lines().next()?.chars().last()
3 }
```

`main()` Can Also Return `Result`

```rust
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), io::Error> {
    let greeting_file = File::open("llms.txt")?;

    Ok(())
}
```

`main()` Can Also Return `Result`

```rust
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("llms.txt")?;

    Ok(())
}
```

When Should We `panic!`?

- `panic!` — or `unwrap()` and `expect()` — when we
  - have information and confidence that the `panic!` condition will not be valid
  - design a library for others to use and need valid data to process

```
1  let home: IpAddr = "127.0.0.1"
2      .parse()
3      .expect("Hardcoded IP address should be valid");
```

- Otherwise, return `Result` for the caller to process

Create Custom Types for Data Validation

```
 1 loop {
 2     let guess: i32 = match guess.trim().parse() {
 3         Ok(num) ⇒ num,
 4         Err(_) ⇒ continue,
 5     };
 6
 7     if guess < 1 || guess > 100 {
 8         println!("The secret number will be between 1 and 100.");
 9         continue;
10     }
11
12     match guess.cmp(&secret_number) {
13         // . . .
14 }
```

Create Custom Types for Data Validation

```rust
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess must be between 1 and 100, got {value}.");
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Required Additional Reading

The Rust Programming Language, Chapter 9