

### **CoE 164**

Computing Platforms

02d: Rust Error Handling





#### **ERROR HANDLING**

Rust has two main ways of handling errors or exceptions in code:

- Panicking
- Enumerations





#### **ERRORS: PANIC**

A **panic** is an unrecoverable error - one that cannot be resolved by handling it separately.

We can induce a panic if we write code that will either cause an unrecoverable error at runtime, or force a panic using the panic! macro.

```
fn main() {
    let a = vec![1, 2, 3];
    println!("a[5] = {}", a[5]); // Bug
    panic!("Goodbye world!"); // Forced
}
```

Example

# ERRORS: RECOVERABLE

Most errors encountered are *recoverable*.

Rust provides enums that encapsulate data that may exist or may cause a panic, which we can handle appropriately.



#### **ERRORS: NULL**

Most programming languages have a construct to place a **null value** to denote the absence of a value. Using a null value in a non-null context leads to sometimes expensive errors!

Rust does not have a standalone null value, but *nullity* can be handled using the Option <T> enum.





#### **ERRORS: OPTION ENUM**

The Option <T> enum has two variants - Some (denoting presence) and None (denoting absence). If the enum is of variant Some, then it will have an associated data of type T.

This enum is in the standard library.

```
enum Option<T> {
    None,
    Some(T),
}
let three_boxed = Some(3);
```



#### **ERRORS: OPTION ENUM**

Because it is an enum, the match and if let constructs can be used to handle the different variants. It also has some convenience functions to handle only one or the other variant while panicking otherwise.

```
// These two statements are almost the same!
let next_node = match next_node {
    Some(x) => x,
    None => panic!("No next node!"),
};
let next_node_v2 = next_node.unwrap();
```



#### **ERRORS: OPTION UNWRAP**

The unwrap () and related methods enable getting the value inside the Some variant of an Option. Note that the plain unwrap () method can definitely panic while the others may never do so.

```
let boxed_num = Some(7);
let seven = boxed_num.unwrap(); // 7

let boxed_num_v2: Option <u32> = None;
let a = boxed_num_v2.unwrap_or(42); // 42
let b = boxed_num_v2.unwrap_or_default(); // 0
```



#### **ERRORS: OPTION LOGIC**

Option has logical operation methods to operate against two Options. Depending on the operation, the output will either be None or the second operand.

```
let a = Some(7).and(None).unwrap_or(0); // 0
let b = Some(7).or(None).unwrap_or(0); // 7
let c = Some(7).xor(None).unwrap_or(0); // 0
let a_v2 = None.xor(Some(7)).unwrap_or(0); // 7
```

## ERRORS: EXCEPTIONS

Recoverable errors are called exceptions in most programming languages. They are usually handled using specialized syntax.

Rust does not have a specific construct for exception handling because the Result <T, E>enum already encapsulates exceptions.



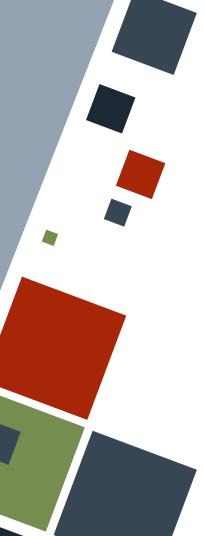


#### **ERRORS: RESULT ENUM**

The Result <T, E> enum has two variants - Ok (denoting successful operation) and Err (denoting failed operation). The enum will have an associated data of type T and E if the enum is of variant Ok and Err, respectively.

This enum is in the standard library.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```



#### **ERRORS: RESULT ENUM**

Because it is an enum, the match and if let constructs can be used to handle the different variants. It also has some convenience functions to handle only one or the other variant while panicking otherwise.

#### Example

```
let mut str in = String::new();
let s = io::stdin()
    .read line(&mut str in);
let ssize = match s {
    Ok(x) => x
    Err( ) => panic!("Invalid"),
let ssize v2 = s
    .expect("Invalid");
```



#### **ERRORS: RESULT ENUM**

If we want to handle specific errors, we can do a match on the error object. For the example below, the data in the error variant has a method kind () which determines what the nature of the error is.

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

if let Err(err_obj) = File::open('hello.txt") {
    match err_obj.kind() {
        Error_Kind::NotFound => {
            println!("File not found!");
        }
        other => {
            panic!("Error encountered: {:?}", other);
        }
    }
}
```



#### **ERRORS: RESULT UNWRAP**

The unwrap () and related methods enable getting the value inside the Ok or Err variant of an Result. Note that the plain unwrap () method can definitely panic while the others may never do so.

```
let boxed_num = Ok(7);
let seven = boxed_num.unwrap(); // 7

let boxed_num_v2 = Err(42);
let a = boxed_num_v2.unwrap_or(24); // 24
let b = boxed_num_v2.unwrap_err(); // 42
```



#### **ERRORS: RESULT LOGIC**

Result has logical operation methods to operate against two Results. Depending on the operation, the output will either be None or the second operand.

```
let a = Ok(7).and(Err(5)); // Err(5)
let a2 = Err(7).and(Ok(5)); // Err(7)

let b = Ok(7).or(Err(5)); // Ok(7)
let b2 = Err(5).or(Ok(7)); // Ok(7)
let b3 = Err(5).or(Err(7)); // Err(7)
```



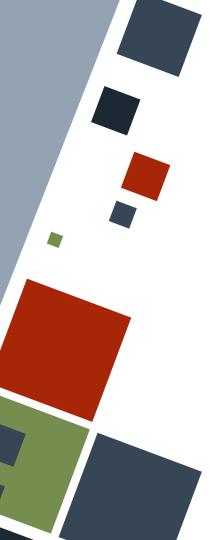
#### **ERRORS: PROPAGATION**

There are cases when functions work on data that have the Result and Option enums. They can opt to send the Err or None variants to the caller by using the ? syntax. The function is required to return a Result enum.

```
use std::error::Error;

fn str_to_i64() -> Result <i64, Box <dyn Error>> {
    let mut str_in = String::new();
    io::stdin().read_line(&mut str_in)?;

    Ok(str_in.trim().parse::<i64>()?)
}
```



#### **ERRORS: PROPAGATION**

To catch all errors, the data type of the Error variant should be Box <dyn Error>.

Sometimes it makes more sense to return an Option instead.

```
use std::error::Error;
```

```
fn str to i64() -> Option <i64> {
    let mut str in = String::new();
    if let None =
io::stdin().read line(&mut str in).ok()
        return None;
    str in.trim().parse:: <464>().ok()
```



#### **RESULT AND OPTION DUALITY**

Since the Result and Option enums are very similar, there exist methods that can convert from one type to another.

```
let my_vec = vec![1, 2, 3];

// Err("Index error")
let a = my_vec.get(3).ok_or("Index error");

// Some([1, 2, 3])
let b: Option <Vec <u64>> = my_vec.try_into().ok();
```

### **RESOURCES**

• The Rust Book





**CoE 164** 

Computing Platforms

02d: Rust Error Handling



