3_Result type and errors_en

May 2, 2023

1 Handling potential failure with the *Result* type

[ ]:
```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
main();
```

```
Guess the number!
Please input your guess.
```

> Note: Executing the cell above will hang the kernel, restart it with the menu Kernel/restart

`read_line` puts what the user types into the string we're passing it, but it also returns a value — in this case, an `io::Result`. Rust has a number of types named `Result` in its standard library: a generic `Result` as well as specific versions for submodules, such as `io::Result`.

The `Result` types are enumerations with two variants: **`Ok`** or **`Err`**. The `Ok` variant indicates the operation was successful, and inside `Ok` is the successfully generated value. The `Err` variant means the operation failed, and `Err` contains information about how or why the operation failed.

An instance of `io::Result` has an `expect` method that you can call. If this instance of `io::Result` is an `Err` value, `expect` will cause the program to crash and display the message that you passed as an argument to `expect`.

If this instance of `io::Result` is an `Ok` value, `expect` will take the inner value that `Ok` is holding and return just that value to you so you can use it. In this case, that value is the number of bytes in what the user entered into standard input.

[ ]:
```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
main();
```

2 Recoverable Errors with Result

2.1 Matching on Different Errors

[2]:
```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Tried to create file but there was a problem:␣
 ↪{:?}", e),
            },
            other_error => panic!("There was a problem opening the file: {:?}",␣
 ↪other_error),
        },
    };
}
main();
```

The type of the value that `File::open` returns inside the Err variant is **`io::Error`**, which is a struct provided by the standard library. This struct has a method kind that we can call to get an **`io::ErrorKind` value**. The enum `io::ErrorKind` is provided by the standard library and has variants representing the different kinds of errors that might result from an io operation. The variant we want to use is **`ErrorKind::NotFound`**, which indicates the file we're trying to open doesn't exist yet. Because `File::create` could also fail, we need a second arm in the inner match expression.

2.2 Propagating errors

When you're writing a function whose implementation calls something that might fail, instead of handling the error within this function, you can return the error to the calling code so that it can decide what to do. This is known as *propagating the error* and gives more control to the calling code, where there might be more information or logic that dictates how the error should be handled than what you have available in the context of your code.

[3]:
```rust
use std::io;
use std::io::Read;
use std::fs::File;

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

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

    let mut s = String::new();

    match f.read_to_string(&mut s) { // read the content of the file into s
        Ok(_) => Ok(s),
        Err(e) => Err(e), //we don't need to explicitly say return, because␣
 ↪this is the last expression in the function
    }
}
```

The function is returning a value of the type `Result< T, E >` where the generic parameter `T` has been filled in with the concrete type `String` and the generic type `E` has been filled in with the concrete type `io::Error`. If this function succeeds without any problems, the code that calls this function will receive an `Ok` value that holds a `String` — the username that this function read from the file. If this function encounters any problems, the code that calls this function will receive an `Err` value that holds an instance of `io::Error` that contains more information about what the problems were.

2.3 A shortcut for propagating Errors: the *?* Operator

The `?` placed after a `Result` value is defined to work in almost the same way as the match expressions. The difference is that the error values that have the `?` operator called on them go through the **`from`** function, defined in the **`From`** trait in the standard library, which is used to convert errors from one type into another.

When the `?` operator calls the `from` function, the error type received is converted into the error type defined in the return type of the current function. As long as each error type implements the from function to define how to convert itself to the returned error type, the `?` operator takes care of the conversion automatically.

[5]:
```rust
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

Note : The `?` operator can only be used in functions that have a return type of Result.