**What does the Question Mark (?) operator do in Rust?**

[[Vennila Pugazhenthi](https://medium.com/@vennilapugazhenthi?source=post_page-----581fe7bc4b0e--------------------------------)](https://medium.com/@vennilapugazhenthi?source=post_page-----581fe7bc4b0e--------------------------------)

[Vennila Pugazhenthi](https://medium.com/@vennilapugazhenthi?source=post_page-----581fe7bc4b0e--------------------------------)

·

Follow

3 min read

·

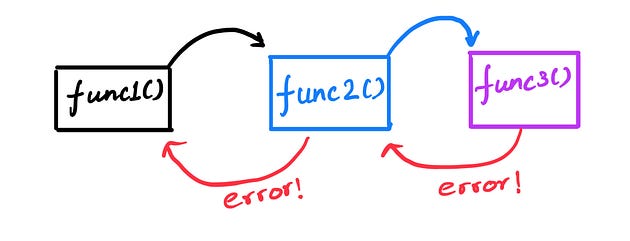
Aug 14

6

Rust's question mark (?) operator is used for propagating errors.

Let’s say there are three functions: func1(), func2(), and func3() where func1() calls func2() and func2() in turn calls func3(). If there’s an error returned from func3() to func2(), func2() can either handle the error or it can return the error to func1().

Returning the error to the caller function (in this case func1()) is known as the propagation of error and this is where the question mark (?) operator is helpful. Propagation of error is used when the caller function has more context and placing the logic of error handling in the caller function makes sense.



Propagation of error

Let’s take a look at a code example. Let’s say you are writing a function to read a username from a file.

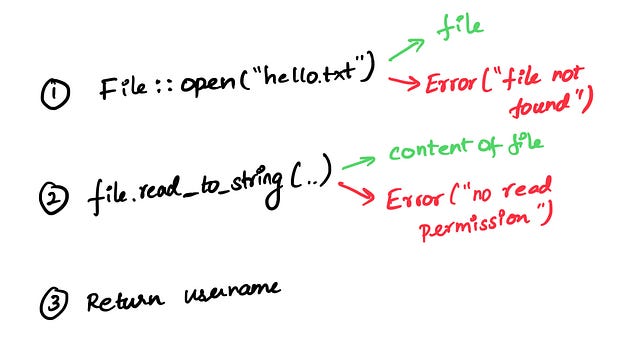
To read a username from a file, we need to do the following:

1. Open the file

2. Read the content of the file into a variable

3. Return the variable

We also need to think about possible failures or errors in these steps. Opening a file that is not present can throw an error or trying to read a file without having access to it can also throw an error.



Possible Errors

Here, we will be propagating any errors, that is to return any errors. Since the function will either return the username of type ‘String’ or return errors, the return type in the function signature will be ‘Result<String, io::Error>’.

fn read\_username\_from\_file() -> Result<String, io::Error> {}

Note: Result<T, E> is an enum with two variants: Ok(T) and Err(E). Ok(T) represents success and contains a value and Err(E) represents error and contains an error value

use std::fs::File;  
use std::io::{self, Read};  
  
fn read\_username\_from\_file() -> Result<String, io::Error> {  
 let username\_file\_result = File::open("hello.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),  
 Err(e) => Err(e),  
 }  
}

Opening a file and reading a file to a string can either succeed or fail. So, we use ‘match’ to handle both cases. If encountered an error, it returns the error early out of the function. If it succeeds, it returns Ok(username).

We can rewrite the same function using the ‘?’ operator.

use std::fs::File;  
use std::io::{self, Read};  
  
fn read\_username\_from\_file() -> Result<String, io::Error> {  
 let mut username\_file = File::open("hello.txt")?;  
 let mut username = String::new();  
 username\_file.read\_to\_string(&mut username)?;  
 Ok(username)  
}

The ‘?’ operator works in a similar way as ‘match’ to handle the ‘Result’ enum.

* If the ‘Result’ has an ‘Ok(T)’ variant, then the ‘?’ operator will return the value inside the ‘Ok’ to the expression and the program will continue.
* If the ‘Result’ has an ‘Err(E)’ variant, then the ‘?’ operator will return the error ‘Err(E)’ to the caller function.

The ‘?’ operator allows chaining of operation. We can rewrite the same function as:

use std::fs::File;  
use std::io::{self, Read};  
  
fn read\_username\_from\_file() -> Result<String, io::Error> {  
 let mut username = String::new();  
  
 File::open("hello.txt")?.read\_to\_string(&mut username)?;  
  
 Ok(username)  
}

The functionality is the same. The ‘?’ allows you to write more ergonomic code.

So far, we have used the ‘?’ operator with the ‘Result<T, E>’ type to propagate errors. Rust also allows you to use the ‘?’ operator with ‘Option<T>’.

Option<T> is an enum with two variants: ‘Some(T)’ contains a value and ‘None’ doesn’t contain a value. This is useful in cases where the value could be present or not.

* If the ‘Option<T>’ has a ‘Some(T)’ variant, then the ‘?’ operator will return the value inside the ‘Some(T)’ to the expression, and the program will continue.
* If the ‘Option<T>’ has a ‘None’ variant, then the ‘?’ operator will return the ‘None’ to the caller functions.

Note: The ‘?’ operator can be used on the ‘Result’ type in a function that returns ‘Result’. Likewise, the ‘?’ operator can be used on the ‘Option’ type in a function that returns ‘Option’. However, the ‘?’ operator cannot be mixed with both the ‘Result’ and ‘Option’ types.