**Simplify Rust error handling with anyhow**

[[Davide Ferrero](https://deid84.medium.com/?source=post_page-----f680410e70f9--------------------------------)](https://deid84.medium.com/?source=post_page-----f680410e70f9--------------------------------)

[Davide Ferrero](https://deid84.medium.com/?source=post_page-----f680410e70f9--------------------------------)

·

Follow

3 min read

·

16 hours ago

50



Photo by [Brett Jordan](https://unsplash.com/@brett_jordan?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Rust has a rather unconventional way of handling errors, and especially at the beginning it can be a bit tricky to deal with it. Fortunately in the myriad of crates available today, I discovered that [anyhow](https://crates.io/crates/anyhow) can help us in this job. If you are interested in learning more about the topic, I recommend you read the part dedicated to error management in [The Rust Programming Language](https://doc.rust-lang.org/book/ch09-00-error-handling.html).

**But why use anyhow?**

In the first days of discovering rust, all of us came across its two most famous enums:

* Option<T>
* Result<T, E>

The first can be used as a return value from functions that may fail, where None can be returned to indicate failure, while the second has another variant Err that we use to indicate why the operation failed. These two enums are really powerful tools, however, the need to always make both parameters explicit in the standard library Result, can sometimes be laborious, particularly when we have to handle errors of different types within, for example, a function.

Let’s take a look at the following code

fn numeric\_error() -> Result<(), usize> {  
 Ok(())  
}  
  
fn string\_error() -> Result<(), String> {  
 Ok(())  
}  
  
fn io\_error() -> Result<(), std::io::Error> {  
 Ok(())  
}  
  
fn main() -> Result<(), Box<dyn Error>> {  
 numeric\_error()?;  
 string\_error()?;  
 io\_error()?;  
 Ok(())  
}

These functions have no real use, but they serve to point out one thing, each returns a different type of error.

* numeric\_error returnts a *usize*
* string\_error returns a *String*
* io\_error returns a *std::io::Error*

Let’s now take a look at the main function. As you can see, I chose to use another fundamental rust tool, the ? operator. In this way we can avoid repeating a match statement for each function in order to handle the errors of the three functions called within the main function.

However, since each function returns a different type of error, the main function has to adapt and use a trait object as the error to satisfy the compiler. Having to specify Result<(), Box<dyn Error>> as the return value of our functions is verbose and becomes tedious if we have to use it frequently. This is one of the reasons that can push us to use a crate like [anyhow](https://crates.io/crates/anyhow). (there are also others that we will see later).

Using anyhow we can modify the code as follows:

use anyhow::Result;  
  
fn numeric\_error() -> Result<()> {  
 Ok(())  
}  
  
fn string\_error() -> Result<()> {  
 Ok(())  
}  
  
fn io\_error() -> Result<()> {  
 Ok(())  
}  
  
fn main() -> Result<()> {  
 numeric\_error()?;  
 string\_error()?;  
 io\_error()?;  
 Ok(())  
}

*anyhow* allows us to use Result<T> instead of Result<T, E>. This way we can focus on the returned value without worrying about the error type.

**Return more information about the error**

Giving more information about errors can be of great help for debugging purposes. anyhow provides a Context trait that gives access to a context method you can use on Result (and Option) types. This method will add the information you specify in the error.

For example:

fn main() {  
 error\_with\_context().unwrap()  
}  
  
fn error\_info\_with\_context() -> Result<()> {  
 Err(anyhow!("Error!")).context("You called a dumb function!")  
}

**Early return**

anyhow provides a utility macro to return early from your methods with an error: the bail! macro. It can be thought of as the equivalent of return Err();

fn another\_error() -> Result<()> {  
 // Could be the short form of: return Err(anyhow!("Early error!"));  
 bail!("Early error!");  
}

**Conclusion**

While we can certainly get by without this crate, I have found anyhow to be very helpful in reducing the repetitiveness that comes with error handling in my code. I hope you too can find it a valid help in your work.