Rust errors explained simply

Rust errors are conceptually simple, but not one-liners. Let's dive in and see how simple they are. In the following example we will create a custom error that can hold a message, can be printed out and can be created from other error types. That will give you your first and complete custom error type.

Custom error type

First let's create a custom error type (enum) with two variants - default and io.

```
#[derive(Debug)]
enum MyErr {
    Default(String),
    Io(io::Error),
}
```

Error printing (Display trait)

To make a struct/enum printable you need to implement `fmt::Display` trait for your object. That is just a few lines of code and since we will print out our error we need to implement that.

```
impl fmt::Display for MyErr {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match &self {
            MyErr::Default(msg) => write!(f, "Default error: {}", msg),
            MyErr::Io(e) => write!(f, "IO error: {}", e),
        }
    }
}
```

Notice that first we decide which error type are we dealing with. Then we print customized error message based on error type. Lastly we append the message hidden in the error type (`MyErr::Default(String)`) or in the inner error (`MyErr::Io(io::Error)`). In both cases the inner type (`String` or `io::Error`) can be printed out as long as they implement `fmt::Display`. So we can treat them the same.

Error conversion from one into another (From trait)

Error is just a type that can be converted from one into another. The thing is we need to tell Rust how. This is done by implementing `From<A>` trait for stuct B if you need to construct B from A. B is our

error `MyErr` and A is the only foreign error we mess with here - `io::Error`. Here is how it's done.

```
impl From<io::Error> for MyErr {
    fn from(err: io::Error) -> Self {
        Self::Io(err)
    }
}
```

What that code does is tell Rust that it's possible to assemble `MyErr` with it's variant `MyErr::Io` that expects exactly an `io::Error`. So basically we wrap our error around already existing one. Simple.

Almost there

That's it for the error itself. Nothing more is needed for `MyErr` to be fully functional just like we need to.

Now let's write some code that actually uses the error - both variants.

Error factories

Here are two dead simple methods that create some errors for us. One for `MyErr::Default` variant (explicitly) and second for `MyErr::Io` variant (automatic conversion).

```
fn test_default_error() -> Result<(), MyErr> {
    Err(MyErr::Default("Dunno, an error".to_string()))
}

fn test_io_error() -> Result<(), MyErr> {
    Err(io::Error::other("Can't open a file"))?
}
```

Last piece

The last piece that we need is to import some already used stuff and then trigger the whole thing. Imports first.

```
use std::fmt;
use std::io;
```

And then the trigger.

```rust
fn main() {
    match test_default_error() {
        Ok(_) => println!("Default test is OK, let's go on ..."),
        Err(e) => eprintln!("Default test has failed, stopping: {}", e),
    }

    match test_io_error() {
        Ok(_) => println!("IO test is OK, let's go on ..."),
        Err(e) => eprintln!("Io test has failed, stopping: {}", e),
    }
}
```

Here is the whole code together with block comments for better understanding.

```rust
// ---------- IMPORT ZONE ----------
use std::fmt;
use std::io;
// ---------- IMPORT ZONE ----------

// ---------- OUR CUSTOM ERROR VARIANTS ----------
#[derive(Debug)]
enum MyErr {
    Default(String),
    Io(io::Error),
}
// ---------- OUR CUSTOM ERROR VARIANTS ----------

// ---------- TEACH ALL VARIANTS TO PRINTABLE ----------
impl fmt::Display for MyErr {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match &self {
            MyErr::Default(msg) => write!(f, "Defaultni chyba: {}", msg),
            MyErr::Io(e) => write!(f, "IO chyba: {}", e),
        }
    }
}
// ---------- TEACH ALL VARIANTS TO PRINTABLE ----------

// ---------- TEACH HOW TO CONVERT io::Error to MyErr type ----------
impl From<io::Error> for MyErr {
    fn from(err: io::Error) -> Self {
        Self::Io(err)
    }
}
```

```rust
// ---------- TEACH HOW TO CONVERT io::Error to MyErr type ----------

// ---------- ERROR FACTORIES ---------
fn test_default_error() -> Result<(), MyErr> {
    Err(MyErr::Default("Nevim, chyba".to_string()))
}

fn test_io_error() -> Result<(), MyErr> {
    Err(io::Error::other("Neumim otevrit fajl").into())
}
// ---------- ERROR FACTORIES ---------

// ---------- MAIN WHEEL - THE TRIGGER ----------
fn main() {
    match test_default_error() {
        Ok(_) => println!("Default test is OK, let's go on ..."),
        Err(e) => eprintln!("Default test has failed, stopping: {}", e),
    }

    match test_io_error() {
        Ok(_) => println!("IO test is OK, let's go on ..."),
        Err(e) => eprintln!("Io test has failed, stopping: {}", e),
    }
}
// ---------- MAIN WHEEL - THE TRIGGER ----------
```

Once we run the whole wheel with `cargo run` we get

```
Default test has failed, stopping: Default error: Dunno, an error
Io test has failed, stopping: IO error: Can't open a file
```

First error is the `MyErr::Default` and contains all the messages wrapped inside just like an onion. Second error is `MyErr::Io` and keeps error message from `std::io::Error`.