HOW TO CODE IT

The definitive guide to error handling in Rust v1.1.1

ARCHITECTURE RUST ERROR HANDLING

Learn to model and handle any error using idiomatic Rust.

Share better Rust!

Contents

Introduction

Are you overwhelmed by the amount of choice Rust gives us for handling errors? Confused about when to return a structured error type or a `Box<dyn Error>`? Intimidated by `Box<dyn Error + Send + Sync + 'static>`'s beefy type signature?

Whether you're building an application or library, this guide will help you make the right decision.

I *love* error handling. I'm obsessed. I work in the finance and space industries, and things go wrong *a lot*.

Failure cases vastly outnumber success cases. Knowing how to communicate what went wrong, to the right audience, in an appropriate amount of detail is a skill that sets you apart from other developers.

Think about how great the Rust compiler's error messages are compared to other programming languages. We want users of our code to have that same reaction, whether they're on our team or using our library. We want them to be impressed when things go wrong!

"Impress your users, even when things go wrong."

Before we dazzle anyone with our error handling skills, though, let's nail the fundamentals.

Part I

Rust error handling basics

What is an error in Rust?

In Rust, an error is any type that implements the `std::error::Error` trait. Here's the definition:

RUST src/core/error.rs

```rust
pub trait Error: Debug + Display {
    // Provided methods
    fn source(&self) -> Option<&(dyn Error + 'static)> { ... }  1
    fn description(&self) -> &str { ... }
    fn cause(&self) -> Option<&dyn Error> { ... }
    fn provide<'a>(&'a self, request: &mut Request<'a>) { ... }
}
```

This is a moderately threatening trait definition, but all four of these methods have default implementations provided for us.

Any type that implements both `Debug` and `Display` can implement `Error`. There's very little manual work required.

In fact, `Error::cause` and `description` are deprecated in favor of `Error::source`

and the `Display` implementation, respectively. You should never have to worry about them, except when working with older code.

`Error::provide` is part of an experimental nightly build, so I won't discuss it here. You won't have to worry about it unless you're working with cutting-edge, unstable code.

`Error::source`

Watch out for the default implementation of `Error::source`. It returns `None`.

If you want a custom error type to return the original error that caused it, you need to provide your own implementation.

The return type of `Error::source` warrants closer examination [1], because we'll see similar types throughout this guide.

You know what `Option` is already. `&(dyn Error + 'static)` simply means "a reference to some error that may live for the whole duration of the program".

Umm, technically... 🤦‍♂️

I frequently refer to types with the `'static` lifetime as being "static". This is convenient shorthand, but subtly incorrect.

They're not static in the sense of a `static` variable. They have the `'static` lifetime, which means that no reference will outlive them, and they would exist until the end of the program if required to.

If the compiler determines that `'static` objects don't *need* to live as long as the program, it's free to drop them sooner.

Please justify my sloppiness by making sure you're clear on the distinction.

The `'static` lifetime is important for error handling, because errors are often

The `'static` lifetime is important for error handling, because errors are often handled long after the code that causes them returns, sometimes on a different thread.

Good luck handling an error that's been dropped unexpectedly! Rust protects us from this scenario.

You'll often see `'static` alongside `Send` and `Sync` bounds. `dyn Error + Send + Sync + 'static` describes "some error that can live as long as the program, be sent between threads by value or shared across threads by immutable reference".

Error::source's return type, `&(dyn Error + 'static)`, doesn't make any promises about thread safety.

In general, standard library code places more relaxed bounds on dynamic errors than you'll see in the broader ecosystem and use in your own projects.

This allows the widest variety of things to behave as errors, with stricter requirements left to the user's discretion.

`Error::source` returns a *non*-static reference to a static `Error`.

How do we make an `Error` type static? Simple – use only owned fields, or fields which specify the `'static` lifetime for references and trait objects.

The following type is only `'static` if the reference assigned to `field` happens to be `'static` itself:

RUST

```
pub struct QuestionablyStatic<'a> {
    field: &'a str,
}
```

These are always `'static`:

RUST

RUST

```rust
pub struct StaticByOwnership {
    field: String
}

pub struct ExplicitlyStatic {
    field: &'static str
}
```

Errors in the context of `Result`

Surprisingly, the type wrapped by `std::result::Result::Err` doesn't need an `Error` bound:

RUST

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

You can use whatever type you want to represent an error inside `Result`.

The same is true for associated types in many trait definitions, such as `std::str::FromStr`:

RUST

```rust
pub trait FromStr: Sized {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

`Err` isn't bounded by `Error` [2]!

Although you *can* use any types in these contexts, I strongly encourage you to only use `Error` implementations.

Other Rust developers will expect these things to behave like `Error`s, and we should strive to be as unsurprising as possible. That doesn't stop you from implementing additional functionality on your `Error`s, though.

"Be unsurprising. Use `Error`-bounded types in most error contexts, even when not strictly required."

There are exceptions to this rule, often within the standard library itself. Look out for the discussion of `Error::downcast` and `Box<dyn Error>` in the next section.

Okay, we've nailed the essentials. Let's get into the choice that confuses most new Rust developers: should we use dynamic or statically typed errors?

Part II

Dynamic error handling in Rust

When to use `Box<dyn Error>` and friends

`Box<dyn Error>` is Rust's vaguest error type. It's just some object that implements `Error` 🤷.

`Box<dyn Error + Send + Sync + 'static>` is its thread-safe counterpart.

The `Error` is boxed because, as a dynamic trait object, we don't know its size at compile time. We have to allocate it on the heap.

`Box<dyn Error>` simply says "something went wrong, check my message or my optional cause to know more".

This has two key properties:

- It's excellent for quickly communicating that something went wrong.
- It's god-awful at providing structured data for an error handler to act on.

If you would like consumers of your error – whether they're error handlers in your own application or users of your library – to be able to dynamically change their program's behavior based on the internal details of an error, don't use `Box<dyn Error>`.

Parsing error details from messages is fragile and hard to maintain. If you expect people to rely on your error messages to drive program behavior, you've also inadvertently made those error messages part of your public API. If that error message changes, code that parses it may break.

If you know that there's nothing useful a receiving program can do with the error, but that the message is helpful for a human debugger, then `Box<dyn Error>` and related trait objects are very convenient.

"Use `Box<dyn Error>` to quickly communicate that something went wrong to a human debugger or end user."

I work on an astrodynamics library for a space mission simulator funded by the European Space Agency. If someone inputs garbage data, like the time `23:59:60` on a year without leap seconds, there's really no way to recover. In this scenario, it would be perfectly reasonable to return `Box<dyn Error>` with a message that explains how silly they are.

Now, we don't actually do this – that's a story for Part III on structured errors – but it is a valid Rust error handling strategy.

Boxed errors aren't errors

Remember when I told you to be unsurprising and put only types that implement `Error` into `Result::Err`?

Well, surprise! `Box<dyn Error>` doesn't implement `Error` 🙄.

You need to wrap `Box<dyn Error>` in a newtype that *does* implement `Error` to get this functionality. My *Ultimate Guide to Rust Newtypes* has got you covered.

Alternatively, you can use a library like anyhow which provides such a type for you. We'll discuss this shortly.

Handling dynamic errors from other people's code

What if library code you call returns a dynamic error?

Hopefully, you just want to log it for a future debugging session. Surely the thoughtfully crafted error message will give you everything you need to solve the problem 🤡.

But say it doesn't, and you need to find out what's inside the `dyn Error`?

I don't envy you this situation. It's often an indicator of bad library design.

"The Laws of Thermodynamics state that the worst code is written by Other People."

Moaning about it won't help you in the moment, though. You need to downcast.

Downcasting errors in Rust

Did you know that you can get a concrete error type back out of a boxed `dyn Error`?

I'm not going to get into *how* the `std::error` crate does this, because it involves some scary `unsafe` code that has nothing to do with handling errors. That won't stop us from using it.

If you'd like me to walk through the `std::error` internals, leave a comment and I'll write it!

`dyn Error` trait objects have three methods for attempting a transformation into some concrete type `T`:

RUST

```
pub fn downcast<T: Error + 'static>(self: Box<Self>) → Result<Box<T>, B
pub fn downcast_mut<T: Error + 'static>(&mut self) → Option<&mut T>
pub fn downcast_ref<T: Error + 'static>(&self) → Option<&T>
```

Do you see it? The underlying error type must be `'static`, or you can't downcast to it. This is one more reason why it's good practice to design only `'static` error types.

Note also that the `Box<Self>` inside the `Result::Err` returned by `downcast` doesn't implement `Error`, but `Self` does. This is one of the cases where returning a non-`Error` inside `Result::Err` makes sense.

If the `dyn Error` is of type `T`, you'll get a `T` for closer inspection. Whether that `T` is owned or borrowed depends on which method you call.

All of this is useless if the underlying type is private to the crate the `dyn Error` came from. In this scenario, politely explain your predicament to the maintainers, then scream into a pillow.

Avoid forcing callers to downcast

I don't encourage designing your errors to require downcasting to figure out what's gone wrong.

If you choose to return a dynamic error, you are communicating that the internal structure of the error shouldn't matter to callers.

"Dynamic errors communicate that their internal structure shouldn't matter to callers."

Forcing them to dig into your crate's error types, identify the possible culprits, downcast, and react dynamically screams "leaky implementation details".

This is Rust, not Go.

So what's the point of downcasting?

If downcasting isn't an ideal way to handle errors, what is it good for? Let's use Actix Web 4.7.0 as an example.

The primary Actix error struct, `Error`, has a single field, `cause`, that holds a `Box<dyn ResponseError>`.

RUST actix-web src/error/error.rs

```rust
pub struct Error {
    cause: Box<dyn ResponseError>,
}
```

`ResponseError` is a trait with identical bounds to `std::error::Error`, but specifies methods to return a status code and an HTTP response body:

RUST actix-web src/error/response_error.rs

```rust
pub trait ResponseError: fmt::Debug + fmt::Display {
    fn status_code(&self) -> StatusCode
    fn error_response(&self) -> HttpResponse<BoxBody>
}
```

It has default implementations for both of these methods, but they're not important here.

What is important is the large number of concrete error types that Actix provides `ResponseError` implementations for: `Box<dyn std::error::Error + 'static>`, `Infallible`, `serde_json::Error`, `std::io::Error`, and many more.

Naturally, Actix users can implement `ResponseError` for their own types too, so `actix_web::error::Error` chooses a dynamic error type to wrap a theoretically infinite variety of `ResponseError`s.

Actix itself doesn't care about the internal structure of any particular `ResponseError`. It just needs a way to get a status code and response body when something goes wrong. This is a scenario where dynamic errors shine.

But you know who might care? *The team whose code produced the error*.

If an Actix user converts an error into Actix's opaque error format, they should reasonably expect to be able to get it out again. That's why `actix_web::error::Error` provides the `as_error` method, which downcasts to the user's original error type.

RUST actix-web src/error/error.rs

```rust
impl Error {
    pub fn as_error<T: ResponseError + 'static>(&self) -> Option<&T> {
        <dyn ResponseError>::downcast_ref(self.cause.as_ref())
    }
}
```

The implementation of `ResponseError::downcast_ref` is specific to Actix. It's not the same as `<dyn std::error::Error>::downcast_ref` – these are methods of distinct trait objects. However, the concept is the same.

(If you're confused by the `<dyn Trait>::method` syntax, it means that the method is defined on the dynamic trait object type, and not as part of the trait itself.)

There are no leaky abstractions here, because the caller of `as_error` also owns the code that created the error in the first place.

Actix never calls `downcast_ref` itself. It doesn't use `downcast_ref` to handle errors. Rather, it provides `as_error` as a means for external parties using Actix's wrapper type to inspect their own implementation details.

`downcast_ref` in tests

Ok, Actix *does* call `downcast_ref`, but only in tests.

Tests are one of the few scenarios where you *should* care that some dynamic error you're

returning has a specific underlying type.

Handling Rust errors with anyhow

What discussion of dynamic error handling in Rust would be complete without talking about anyhow?

anyhow is Rust's most-loved crate for handling errors in the laziest way possible.

`anyhow::Error` is effectively a `Box<dyn Error + Send + Sync + 'static>` with bells on. It always gives you a backtrace, and, unlike `Box`, it takes up only one machine word, not two (a "narrow pointer").

anyhow comes with a selection of macros, methods and blanket implementations to make wrapping and adding context to any `Display + Send + Sync + 'static` type a breeze.

Just like `actix_web::error::Error`, `anyhow::Error` is a wrapper for user-provided types. Seeing as those users might want their types back, it provides `downcast` methods in your three favorite flavors: owned, `&` and `&mut`.

I use anyhow often, and I find it's a better fit for applications than libraries.

If you return a concrete `anyhow::Error` across a crate boundary, you force the caller to depend directly on anyhow, and not everyone will want to.

Also, if you make anyhow part of your public interface, you can't upgrade to new major versions of anyhow without bumping the major version of your own crate.

As a general rule, return only your own or standard library error types across crate boundaries to minimize leakage of your implementation details into other people's code.

"Return only your own or standard library error types across crate boundaries."

across crate boundaries.

Who is your audience and what will they do with your error?

I hope it's becoming clear that how you choose to handle your errors depends on two key things:

- Who the audience for the error is.
- What they should be able to do with an error you give them.

Dynamic errors are great for consolidating a wide range of error types and returning them in a format where the only reasonable thing to do is write to output, whether that's a logger or an HTTP connection.

In Part III, we'll look at structured, statically typed errors as carriers of data that we can handle programmatically. More than that though, we'll see how they serve as invaluable, innate documentation for other developers.

When we understand both of these error handling styles, we'll bring them together, equipping ourselves with the knowledge to handle any kind of error that might arise, and avoid some nasty footguns.

Part III

Structured error handling in Rust

Sane APIs support programmatic error handling

Knock knock. It's Hyrum's Law.

Hyrum's Law

> "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody."
>
> *hyrumslaw.com*

In other words, someone, somewhere *will* end up depending on your error messages. You might not say these messages are part of your public API, but the public has access to them, and if they've got no better way to handle your errors, they're going to `if-else` your strings.

Changing an error message in the popular library you maintain is going to fuck someone up – and they will end up at your door. Knock knock.

If you're thinking that this is a low-impact edge-case, consider that error strings from deep within the Go standard library are depended on by programs of real consequence.

Here's a sample from Go's `http` package:

GO `go src/net/http/request.go`

```go
// MaxBytesError is returned by [MaxBytesReader] when its read limit is
type MaxBytesError struct {
    Limit int64
}

func (e *MaxBytesError) Error() string {
    // Due to Hyrum's law, this text cannot be changed.  3
    return "http: request body too large"
}
```

I didn't write the comment at `3`. One of the Go team did. Good thing, too, because here's Grafana depending on it.

Credit goes to Abenezer Belachew for finding these examples, and his fascinating write-up on Hyrum's Law in Go

Hyrum's Law in Go.

I'm calling out Go because it was famously unergonomic to discern whether a specific type of error was present in a long chain of errors. Things improved in Go 1.13, but Hyrum's Law had already had its way with the Go codebase.

In fact, `MaxBytesError` was only added to Go's public API in 2022, replacing the anonymous error that forced Grafana and others to depend on the error string. The message it outputs can't change without breaking their code.

Shouldn't they have known better than to depend on an undocumented implementation detail? Are they software engineers or kindergarteners?

Kids need *structure*, and Go didn't give them any. There was no stable way to identify this error.

This is precisely why you should avoid forcing callers to downcast your Rust errors. Whenever there's the slightest possibility that someone might want to react to your error programmatically, a dynamic error type won't do.

Luckily, Rust makes it simple to build strong, beautiful errors into our API contracts.

Build expressive Rust errors with enums

Consider a simple, Gregorian `Date` type:

RUST

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
pub struct Date {
    year: i32,
    month: u8,
    day: u8,
}

impl Date {
    pub fn new(year: i32, month: u8, day: u8) -> Result<Self, ???> {
        todo!()
    }
}
```

When deciding what type of error to return [4], start by listing all the ways someone might lose their mind when calling your function. In our case:

- The month may be outside the range `1..=12`.
- The day may be zero, or greater than the number of days in the given month.
- The caller requests February 29th on a non-leap year.

Expressing the constructor return type as `Result<Self, Box<dyn Error>>` is convenient – just box a string explaining the problem. Convenient, that is, until Hyrum wants his pound of flesh. We can't change these strings because we've forced people to depend on them.

"Codify all possible error states in your public API."

In Rust, our weapon of choice is the `enum`:

RUST

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum DateError {
    InvalidMonth(u8),
    InvalidDay { month: u8, day: u8 },
    NonLeapYear(i32),
}

impl Display for DateError {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        use DateError::*;
        match self {
            InvalidMonth(month) => write!(f, "{} is not a valid month",
            InvalidDay{ month, day } => {
                write!(f, "{} is not a valid day for month {}", day, mon
            },
            NonLeapYear(year) => write!(f, "{} is not a leap year", year
        }
    }
}

impl Error for DateError {}
```

```
impl Error for DateError {}
```

`DateError` gives us two massive benefits:

1. The entire universe of errors that the caller needs to handle is obvious from the function signature. There's no need to dig through the `Date` constructor call chain to figure out what errors it might return. This is a key shortcoming with dynamic errors or, God forbid, exceptions in other languages.
2. It encodes our list of problem states in a way that callers can respond to programmatically. They match each variant of interest to act on the cause, supported by structured data describing the invalid fields.

`DateError`'s variants are a documented part of our public API. Adding or removing variants or their fields are still breaking changes, but, unlike string error messages, they're governed by an explicit contract between us and our users.

If your users still choose to depend on your messages rather than your enum variants, that's very much a *them* problem, not a *you* problem, which is the best kind of problem.

"Good library developers give users recipes for perfection. Some people can't cook."

Composing structured error types

So far, so simple. But in real-life code, fallible functions call other fallible functions, and each failure may be represented by a different error type. We need to compose these errors into a single return type.

Umbrella errors

Some crates and modules choose to compose every error their code produces into a single public error type. `std::io::Error` is the most prominent example (you'll hear more about it later).

I strongly discourage you from doing this if many different things can go wrong when calling your code.

If your module exposes 10 functions that fail in different ways, don't be tempted to define:

RUST

```rust
pub enum Error {
    Fn1Error,
    Fn2Error,
    // ...,
    Fn10Error,
}

pub fn fn1() -> Result<(), Error> {}
pub fn fn2() -> Result<(), Error> {}
// ...
pub fn fn10() -> Result<(), Error> {}
```

For each function call that results in an error, callers would have to filter out the noise of nine, unrelated error cases.

As we'll soon see, you can't always eliminate noise completely, but our aim is to design error types that prioritize *relevant* information.

Let's extend our budding time library with a new struct and a corresponding error:

RUST

```rust
#[derive(Debug, Clone, Copy, Default, PartialEq, Eq, PartialOrd, Ord)]
struct UtcTimestamp {
    hour: u8,
    minute: u8,
    second: u8,
}

impl UtcTimestamp {
    pub fn new(hour: u8, minute: u8, second: u8)
    -> Result<UtcTimestamp, UtcTimestampError> {
        todo!()
    }
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum UtcTimestampError {
    InvalidHour(u8),
    InvalidMinute(u8),
```

```
    InvalidSecond(u8),
    InvalidLeapSecond { hour: u8, minute: u8, second: u8 },
}

impl Display for UtcTimestampError {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        use UtcTimestampError::*;
        match self {
            InvalidHour(hour) => write!(f, "{} is not a valid hour", hou
            InvalidMinute(minute) => {
                write!(f, "{} is not a valid minute", minute)
            }
            InvalidSecond(second) => {
                write!(f, "{} is not a valid second", second)
            }
            InvalidLeapSecond {
                hour,
                minute,
                second,
            } => write!(
                f,
                "{}:{}:{} is not a valid leap second",
                hour, minute, second
            ),
        }
    }
}

impl Error for UtcTimestampError {}
```

The `UtcTimestampError` variants for hour-, minute- and second-related errors are obvious. However, the International Earth Rotation and Reference Systems Service (IERS – they hold the best parties) occasionally adds leap seconds to keep UTC in sync with the rotation of the Earth.

This is why – and I say this as an author of an astronomical time library – UTC is the Devil's Timescale.

Leap seconds always occur at `23:59:60`. If we have a `second` field of `60`, and `hour` and `minute` fields that aren't `23` and `59`, respectively, someone's messed up. We capture this with `UtcTimestampError::InvalidLeapSecond`.

Now, leap seconds don't happen every year, praise be to IERS. And they only occur in

June or December. So when we define a `UtcDateTime` time, we need to account for three things:

1. `DateError` s.
2. `UtcTimestampError` s.
3. Leap seconds with valid timestamps, but which fall on a year or month in which there was no leap second.

How do we compose three errors that occur in the course of a single function call? That's right – with another enum.

RUST

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
struct UtcDateTime {
    date: Date,
    time: UtcTimestamp,
}

impl UtcDateTime {
    fn new(year: i32, month: u8, day: u8, hour: u8, minute: u8)
    -> Result<Self, UtcDateTimeError> {
        todo!()
    }
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum UtcDateTimeError {
    Date(DateError),
    Time(UtcTimestampError),
    InvalidLeapSecond(Date),
}

impl Display for UtcDateTimeError {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        use UtcDateTimeError::*;
        match self {
            Date(err) => write!(f, "invalid date: {}", err),
            Time(err) => write!(f, "invalid time: {}", err),
            InvalidLeapSecond(date) => {
                write!(f, "no leap second occurs on {}", date)
            },
        }
```

```
        }
    }

    impl Error for UtcDateTimeError {}
```

`DateError` and `UtcTimestampError` are thinly wrapped in `UtcDateTime`-specific equivalents. Their messages carry a little more context for human readers.

Having access to both a date and a time, the `UtcDateTime` constructor can also validate whether a leap second timestamp falls on a leap second date. `UtcDateTimeError::InvalidLeapSecond` is a new variant specific to the compound struct.

Ok, next question: what error type should this alternative `UtcDateTime` constructor return?

RUST

```
impl UtcDateTime {
    fn from_date_and_time(date: Date, time: UtcTimestamp)
    → Result<Self, ???> {
        todo!()
    }
}
```

With the onus on the caller to construct valid `Date`s and `UtcTimestamp`s and handle their errors, the constructor's error space shrinks to just `InvalidLeapSecond`, which could plausibly become its own `struct` error type.

What's the proper way to support *both* constructors? This?

RUST

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct InvalidLeapSecondDateError(Date);

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum UtcDateTimeError {
    Date(DateError),
    Time(UtcTimestampError),
```

```
    InvalidLeapSecond(InvalidLeapSecondDateError),
}

// Error implementations omitted

impl UtcDateTime {
    fn new(year: i32, month: u8, day: u8, hour: u8, minute: u8)
    -> Result<Self, UtcDateTimeError> {
        todo!()
    }

    fn from_date_and_time(date: Date, time: UtcTimestamp)
    -> Result<Self, InvalidLeapSecondDateError> {
        todo!()
    }
}
```

Maybe.

This approach succeeds in giving the caller only the most relevant information about the issue, at a cost to you, the developer. All this nesting creates a lot of code. We want to avoid module-scale umbrella errors, but while a bespoke error per domain type is one thing, you may think that a bespoke error per *function* is excessive.

Ultimately, you decide whether it's reasonable for your users to handle unrelated error variants. Trust me, they'll let you know if not. Stick to our rule of thumb and you'll be fine:

"Design error types that prioritize *relevant* information. Minimize unrelated noise."

How to improve the ergonomics of your Rust errors

Manually implementing errors is boilerplatey. In this section, we'll remove that barrier to implementing robust error types for every occasion.

thiserror

A titan among error handling crates, thiserror dramatically simplifies the process of

defining and constructing situational error types.

It's the order to anyhow's dynamic chaos. Perfectly balanced, as all things should be.

Let's reimplement `UtcDateTimeError` with thiserror:

RUST

```rust
#[derive(thiserror::Error, Debug, Clone, Copy, PartialEq, Eq)]  5
pub enum UtcDateTimeError {
    #[error(transparent)]  6
    Date(#[from] DateError),  7
    #[error(transparent)]
    Time(#[from] UtcTimestampError),
    #[error("no leap second occurs on {0}")]  8
    InvalidLeapSecond(Date),
}
```

First off, the manual `Display` implementation is gone, replaced by annotations. `thiserror::Error` is a derive macro that handles the legwork for us [5].

At [6], we take advantage of the `transparent` annotation to make thiserror forward the error message from the wrapped `DateError`. This is useful when the wrapping enum doesn't have any additional context that could clarify the problem for users.

Next, we generate an implementation of `From<DateError>` for `UtcDateTimeError::Date`, and `From<UtcTimestampError>` for `UtcDateTimeError::Time` [7]. This makes constructing the `UtcDateTimeError` wrapper from its causes trivial.

Best of all, `Result`s containing either `DateError` or `UtcTimestampError` will be transparently morphed into `Result<T, UtcDateTimeError>` when returned with the try operator, `?`:

RUST

```rust
fn some_utc_datetime_func() -> Result<(), UtcDateTimeError> {
    Err(DateError::InvalidMonth(13))?
}
```

Unlike `DateError` and `UtcTimestampError`, `UtcDateTimeError::InvalidLeapSecond`

has no `Display` implementation of its own, so the final step is to generate one at [8], interpolating the wrapped `Date`.

If you'd prefer not to take a dependency on thiserror, you can still get try-operator ergonomics by manually implementing `From` for your error as you would with any other type.

Structured error handling examples from the Rust ecosystem

Don't take my word for it. Here are prime examples of structured errors from two popular Rust crates.

tracing

tracing is the number-one framework for instrumenting your Rust applications. Collecting the events you emit requires a collector – some implementation of `tracing_core::collect::Collect`. As the name suggests, there can be only one global default collector. What happens if you try to set it twice?

RUST tracing tracing-core/src/dispatch.rs

```rust
/// Returned if setting the global dispatcher fails.
pub struct SetGlobalDefaultError {  9
    _no_construct: (),
}

impl SetGlobalDefaultError {
    const MESSAGE: &'static str = "a global default trace dispatcher has
}

impl fmt::Debug for SetGlobalDefaultError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_tuple("SetGlobalDefaultError")
            .field(&Self::MESSAGE)
            .finish()
    }
}
```

```rust
impl fmt::Display for SetGlobalDefaultError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.pad(Self::MESSAGE)
    }
}

#[cfg(feature = "std")]
#[cfg_attr(docsrs, doc(cfg(feature = "std")))]
impl error::Error for SetGlobalDefaultError {}
```

Since there's only one way setting the global default can fail – when it's already been set – this is neatly represented by an empty struct: `SetGlobalDefaultError` [9].

wgpu

Here's an all-singing, all-dancing example from wgpu, a cross-platform graphics API based on the WebGPU standard. Creating compute shader pipelines is fraught with danger:

RUST wgpu wgpu-core/src/pipeline.rs

```rust
#[derive(Clone, Debug, Error)]
#[non_exhaustive]  10
pub enum CreateComputePipelineError {
    #[error(transparent)]
    Device(#[from] DeviceError),
    #[error("Unable to derive an implicit layout")]
    Implicit(#[from] ImplicitLayoutError),  11
    #[error("Error matching shader requirements against the pipeline")]
    Stage(#[from] validation::StageError),
    #[error("Internal error: {0}")]
    Internal(String),
    #[error("Pipeline constant error: {0}")]
    PipelineConstants(String),  12
    #[error(transparent)]
    MissingDownlevelFlags(#[from] MissingDownlevelFlags),
    #[error(transparent)]
    InvalidResource(#[from] InvalidResourceError),
}
```

`CreateComputePipelineError` showcases a thiserror-derived enum error. It includes variants composed from granular, low-level errors [11], and new errors exclusive to the creation of the pipeline [12].

If you'd like to see more examples from `wgpu`, which adopts the maximalist approach of having distinct error types for each operation, `wgpu_core/src/ray_tracing.rs` contains several error definitions, including one 27-variant monster!

Non-exhaustive errors

Note that `CreateComputePipelineError` is marked `non_exhaustive` [10]. This is the wgpu devs saying "we reserve the right for other things to go wrong in future".

When you match a non-exhaustive enum error, rustc will force you to add a catch-all pattern. This will mop up any new variants that you don't explicitly match.

If the devs hadn't done this, adding a new error variant would be a breaking change.

`std::io::Error`, Rust's most challenging error type

`std::io::Error` isn't the prettiest part of the Rust standard library. It's trying to solve a very hard problem – to represent any possible IO error, on all supported operating systems, with the smallest possible overhead. In doing so, it ends up being too low-level for some use cases, and too high-level for others.

We'll scavenge what looks tasty, and leave the bits that look off. Like vultures.

Here's the implementation (for clarity, I've left out the bit-packing optimization used on 64-bit systems):

RUST rust library/std/src/io/error.rs|repr_unpacked.rs

```rust
pub struct Error {
    repr: Repr,
}

struct Repr(Inner);
```

```rust
type Inner = ErrorData<Box<Custom>>;

struct Custom {
    kind: ErrorKind,
    error: Box<dyn error::Error + Send + Sync>,
}

enum ErrorData<C> {  13
    Os(RawOsError),
    Simple(ErrorKind),
    SimpleMessage(&'static SimpleMessage),
    Custom(C),
}
```

Aha! Four error representations wearing a trench coat! And they would have gotten away with it if it wasn't for us meddling crabs.

`ErrorData` specifies four, broad forms of error [13]:

- `Os` wraps error codes returned by the operating system. `RawOsError` is a `usize` alias.
- `SimpleMessage` is, simply, an error message.
- `Simple` wraps an `ErrorKind` – another enum, which we'll discuss imminently.
- `Custom` is a catch-all variant for anything that isn't covered by the other three. Specifically, `std::io::Error` uses an `ErrorData<Box<Custom>>`, meaning `ErrorData::Custom` holds a `Box<Custom>`. `Custom` itself combines an `ErrorKind` and a boxed, dynamic error. Capeesh?

I won't reproduce `ErrorKind` in full – it has more variants than Covid. Here's a sample of the many, many ways IO goes wrong:

RUST rust library/std/src/io/error.rs

```rust
#[derive(Clone, Copy, Debug, Eq, Hash, Ord, PartialEq, PartialOrd)]
#[non_exhaustive]
pub enum ErrorKind {
    // ...

    #[stable(feature = "rust1", since = "1.0.0")]
    ConnectionRefused,  14
    #[stable(feature = "rust1", since = "1.0.0")]
```

```
    #[stable(feature = "rust1", since = "1.0.0")]
    ConnectionReset,

    // ...

    #[unstable(feature = "io_error_more", issue = "86442")]
    FilesystemQuotaExceeded,  15
    #[stable(feature = "io_error_a_bit_more", since = "1.83.0")]
    FileTooLarge,

    // ...

    #[stable(feature = "io_error_a_bit_more", since = "1.83.0")]
    ArgumentListTooLong,  16
    #[stable(feature = "rust1", since = "1.0.0")]
    Interrupted,

    // ...

    #[stable(feature = "rust1", since = "1.0.0")]
    Other,  17
    #[unstable(feature = "io_error_uncategorized", issue = "none")]  18
    #[doc(hidden)]  19
    Uncategorized,
}
```

`ErrorKind` is a smash-up of network failures [14], filesystem errors [15] and OS process complaints [16]. There are write-only error cases, like `ReadOnlyFilesystem`, in an enum that's shared by read operations. This is not the tight error definition we're used to.

Down in the basement of your program, `std::io` doesn't know what sort of operation you're attempting. It shovels bytes into the OS via the `Write` trait, and gets bytes out via the `Read` trait. `std::io::Error` is baked into their definitions.

What are the consequences? Since `Read` and `Write` depend on `std::io::Error`, these traits must live in `std`, not `core`. `std::io::Error` presumes the presence of an operating system. But if you're running `no_std`, there's a chance you *are* the operating system! `no_std` programs have to reinvent these traits without this dependency.

" `no_std` programs have to reinvent `Read` and `Write` without `std::io::Error` "

There's strangeness for `std` programs too. `Read` and `Write` are the basis for higher-level readers and writers. If you design an HTTP connection, a database connection, a packet library, a logger, or anything else with sophisticated IO, odds are that you'll define specialized readers and writers based on lower-level implementations of `Read` and `Write`.

Since specialist implementations must return `std::io::Error` to satisfy the IO trait signatures, the Rust devs had to give `std::io::Error` a way to represent errors that `std::io` doesn't know about.

That's what `Custom` is for. It's built from any `ErrorKind` variant – probably `Other` – and a `Box<dyn Error + Send + Sync>`. In other words, custom readers and writers are forced to represent their custom errors dynamically. In this mirror world, the more specialized the use case, the more vague `std::io::Error` becomes.

"The more specialized the use case, the more vague `std::io::Error` becomes."

What's up with `Other` and `Uncategorized`?

Ever get that creeping feeling – late at night, long after the world has gone to sleep – of something lurking just beyond the corner of your eye? That's Hyrum.

He comes for all of us, just like he came for `std::io`.

That link directs to a Rust language tracking issue, in which a number of Rust Nightly users complain of failing tests following the addition of several new `ErrorKind` variants. But `ErrorKind` is non-exhaustive, so how did this happen?

Hyrum's Law.

`Other` [17] was formerly a catch-all variant not just for Rust users, but for the Rust standard library itself. For example, there is no `ErrorKind` representing a failure to write to `stdout`. Instead, a message describing the problem was bundled into `Other`.

Did the `ErrorKind` documentation explicitly warn users that this was *not* a stable contract, and that these "other" errors may be replaced as time went on? Yes, it did.

Did Rust users depend on this anyway? Naturally.

When these vague errors became bespoke `ErrorKind` variants, code that expected to find them in `Other` stopped working.

Enter `Uncategorized`. Reason can't stop developers from depending on implicit behavior, but rustc can.

`Uncategorized` is the new home for errors the Rust team hasn't figured out what to do with. The standard library no longer assigns errors to `Other`. Since `Uncategorized` is marked as `unstable` [18], you can't match it without enabling an unstable feature yourself – you know what you're getting yourself in for.

For good measure, `Uncategorized` is also hidden from the docs [19], but that's the Hyrum's Law equivalent of trying to hold back the tide.

That's `std::io::Error`. Pros: enum-based variants for every error kind Rust knows about. A valiant, workable solution to an unforgiving problem. Cons: everything else.

When designing your own error types, consider these pitfalls carefully, and plan your escape route.

Now that you're equipped with the strengths and weaknesses of both dynamic and structured errors in Rust, it should be clear that you're not faced with a binary choice to adopt one or the other.

This isn't *Highlander*. `anyhow` and `thiserror` serve different purposes and may happily coexist within the same codebase.

Choose how to represent each error on a case-by-case basis, guided by what you expect users to do with your error.

And keep an eye out for Hyrum.

He hunts at night.

Part IV

Error handling in exceptional circumstances: `panic!`, `no_std` and FFIs

Let me cook.