Error handling in Rust

???

github.com/Kixunil

AWESOME
HARD
ERROR HANDLING IN RUST
JAKE-CLARK.TUMBLR

Error handling in C

- 0 means failure, 1 means success
- 1 means failure, 0 means success
- -1 means failure, 0 means success
- Magic value means failure, anything else success
- errno is set on failure, but not reset on success
- Which file failed to open?
- Annoying ifs
- Can forget to check, may lead to UB

Error handling in C++

- All the C bagage remains if you’re unlucky enough!
- Exceptions are zero-cost on success but painfully expensive on failure
- What are all the possible exceptions a function can throw?
- Forgetting to check crashes the entire application
- Which are all the possible function calls that can throw?

Error handling in Go

- Second parameter of a tuple is an error until it isn’t
- Can forget to check
- Can return BS values such as (nil, nil) or (non-nil, non-nil)
- Annoying ifs
- Error requires heap allocation
- Only a string by default, requires dynamic casting for more

Error handling in Python, Java, …

- Exceptions are heap allocated
- What are all the possible exceptions a function can throw?
- Forgetting to check crashes the entire application
- Which are all the possible function calls that can throw?
- What does the end user do with the stack trace?

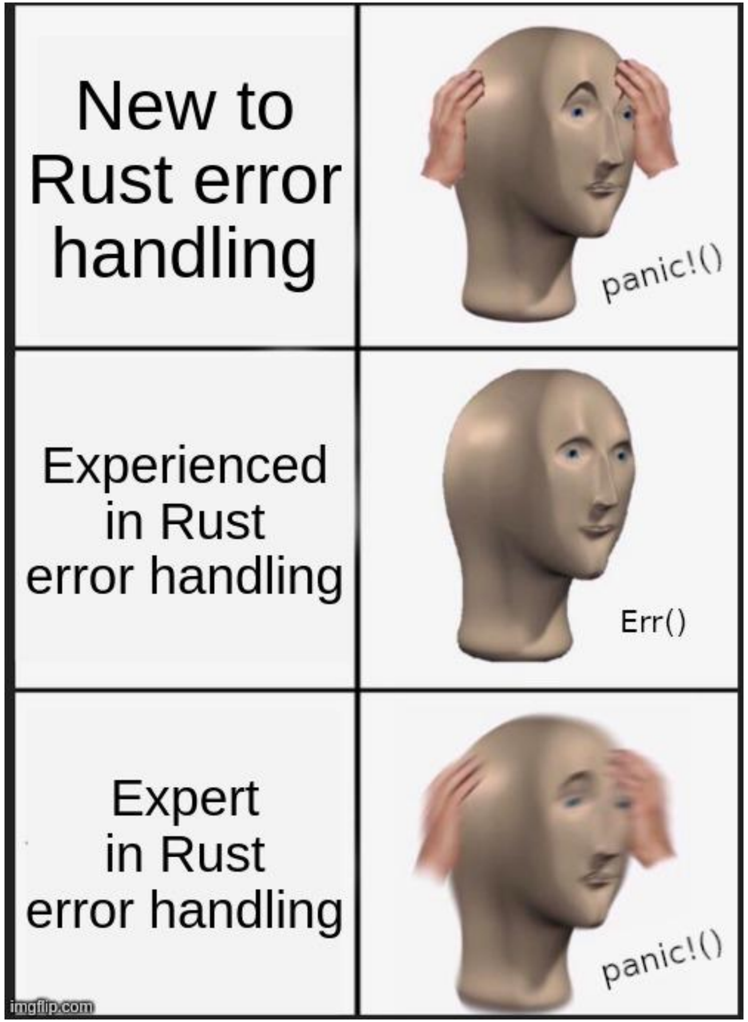
Error handling in Rust

- Standardized type Result
- Result is either Ok or Err, never both or neither.
- Accessing invalid data is impossible (without obviously wrong unsafe)
- Standardized Error trait (already in core! 🎉)
- The ? operator is neither annoying nor invisible
- You can statically inspect the error type in most cases
- You can hide the error type if needed
- Error type may be an enum, giving you the list of possible errors
- Usually on stack but may be on heap if needed
- Forgetting to check throws a warning

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

```rust
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt")?;
    // Early return on error
    file.write_all(format!("name: {}\n", info.name).as_bytes())?;
    file.write_all(format!("age: {}\n", info.age).as_bytes())?;
    file.write_all(format!("rating: {}\n", info.rating).as_bytes())?;
    Ok(())
}
```

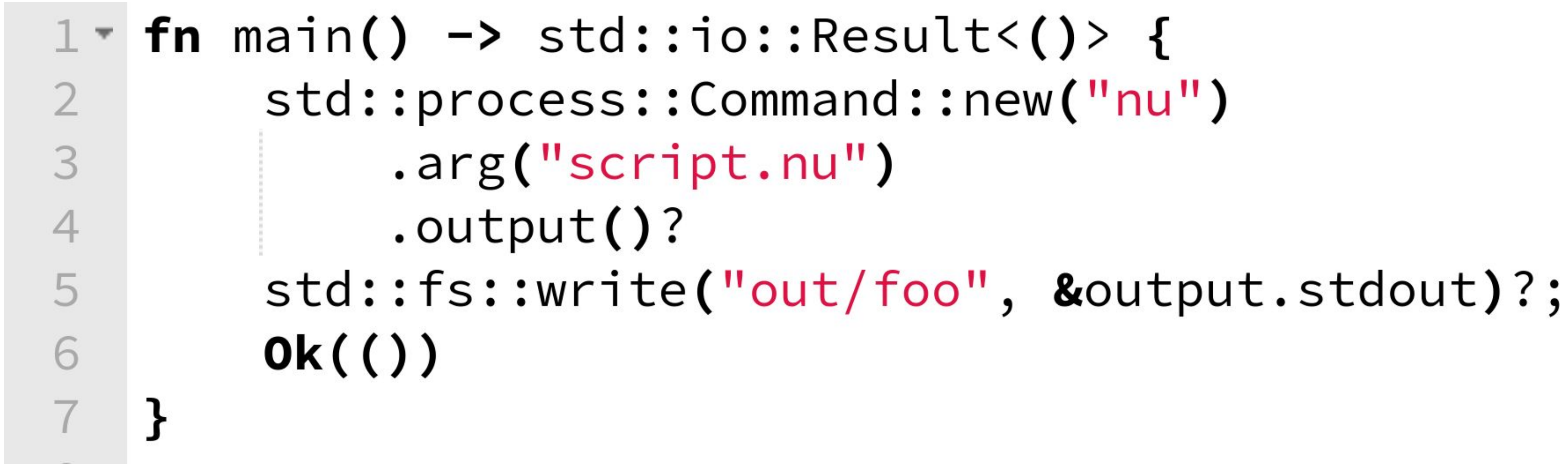
What should my error type be?!


New to Rust error handling
panic!()
Experienced in Rust error handling
Err()
Expert in Rust error handling
panic!()
imgflip.com

Panic on programmer errors

Is it an internal tool?

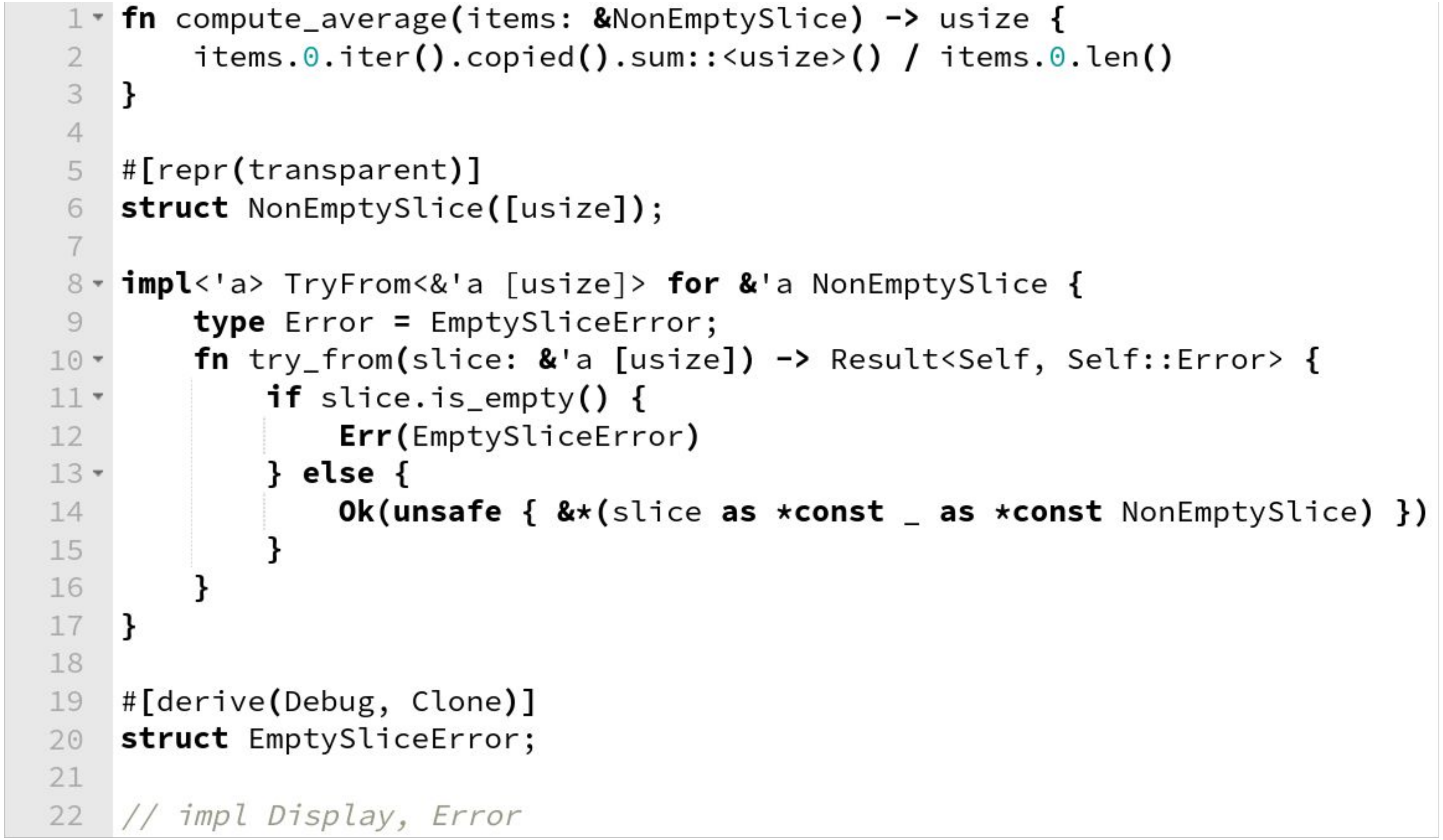
panic!()



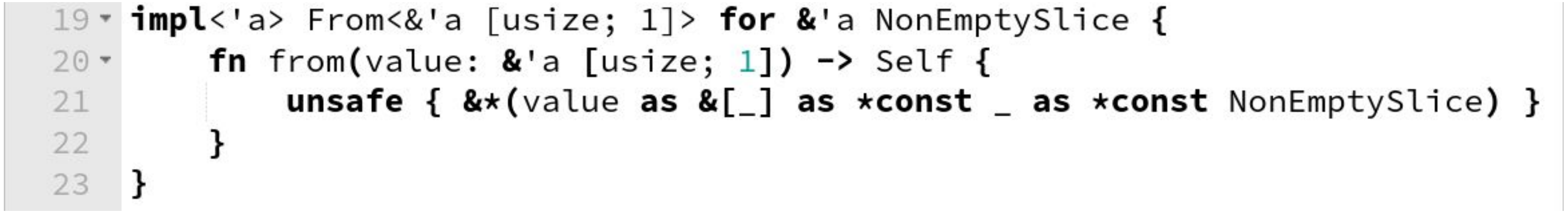
```rust
fn main() -> std::io::Result<()> {
    std::process::Command::new("nu")
        .arg("script.nu")
        .output()?
    std::fs::write("out/foo", &output.stdout)?;
    Ok(())
}
```

The best error type is no error type

```rust
fn compute_average(items: &[usize]) -> Result<usize, EmptySliceError> {
    items.iter().copied().sum::<usize>().checked_div(items.len()).ok_or(EmptySliceError)
}

#[derive(Debug, Clone)]
struct EmptySliceError;

// impl Display, Error
```

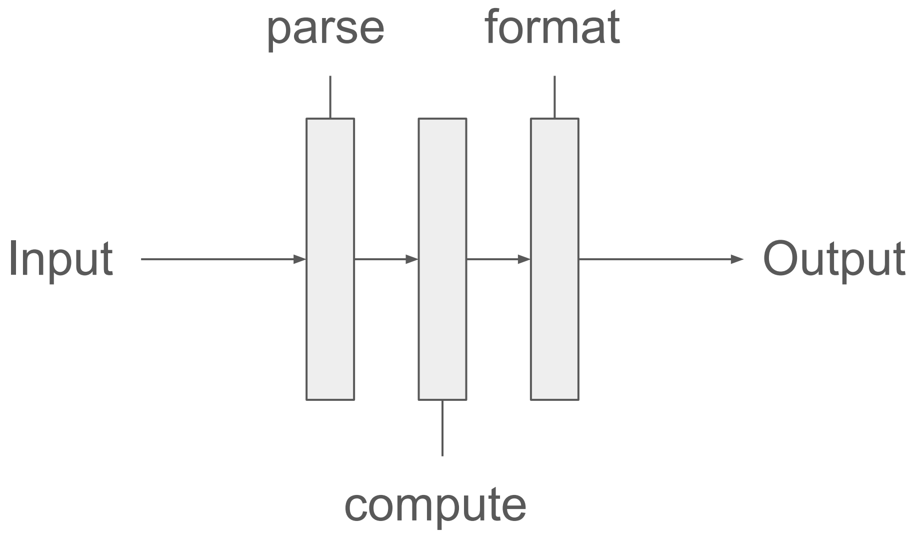


```rust
fn compute_average(items: &NonEmptySlice) -> usize {
    items.0.iter().copied().sum::<usize>() / items.0.len()
}

#[repr(transparent)]
struct NonEmptySlice([usize]);

impl<'a> TryFrom<&'a [usize]> for &'a NonEmptySlice {
    type Error = EmptySliceError;
    fn try_from(slice: &'a [usize]) -> Result<Self, Self::Error> {
        if slice.is_empty() {
            Err(EmptySliceError)
        } else {
            Ok(unsafe { &*(slice as *const _ as *const NonEmptySlice) })
        }
    }
}

#[derive(Debug, Clone)]
struct EmptySliceError;

// impl Display, Error
```



```rust
impl<'a> From<&'a [usize; 1]> for &'a NonEmptySlice {
    fn from(value: &'a [usize; 1]) -> Self {
        unsafe { &*(value as &[_] as *const _ as *const NonEmptySlice) }
    }
}
```

Input
Output


parse
format
Input
Output
compute

serde

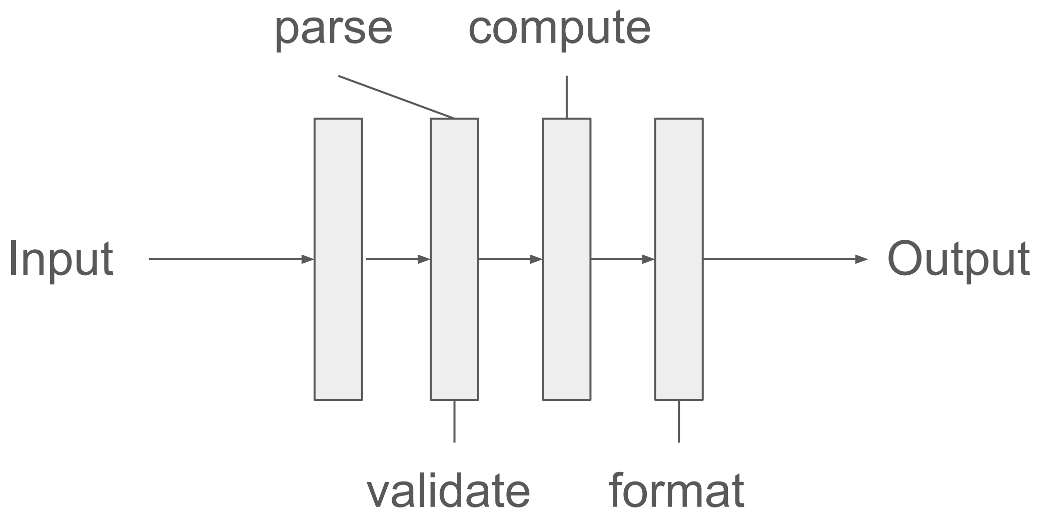
Example: Where it breaks (configure_me crate)

- An application parameter can be optional or mandatory
- An application parameter can have a default value or not
- A mandatory parameter must not have default value
- Settings are given in two toml fields

```
default = "42"

optional = false
```

```rust
pub enum Optionality {
    Mandatory,
    Optional,
    DefaultValue(String),
}
```

```rust
fn validate_optionality(optional: Option<Spanned<bool>>, default_optional: bool, default: Option<Spanned<String>>) -> Result<Optionality, FieldError> {
    match (optional, default_optional, default) {
        (Some(opt), _, None) if !opt.get() => Ok(Optionality::Mandatory),
        (Some(opt), _, Some(default)) if !opt.get() => Err(FieldError::MandatoryWithDefault { optional_span: opt.to_span(), default_span: default.to_span(), }),
        (Some(_), _, None) => Ok(Optionality::Optional),
        (_, _, Some(default)) => Ok(Optionality::DefaultValue(default.into_inner())),
        (None, true, None) => Ok(Optionality::Optional),
        (None, false, None) => Ok(Optionality::Mandatory),
    }
}
```


parse
compute
Input
Output
validate
format

Are you writing a binary?

Use anyhow

```rust
use anyhow::{Context, Result};

fn main() -> Result<()> {
    ...
    it.detach().context("Failed to detach the important thing")?;

    let content = std::fs::read(path)
        .with_context(|| format!("Failed to read instrs from {}", path))?;
    ...
}
```

HTTP server?

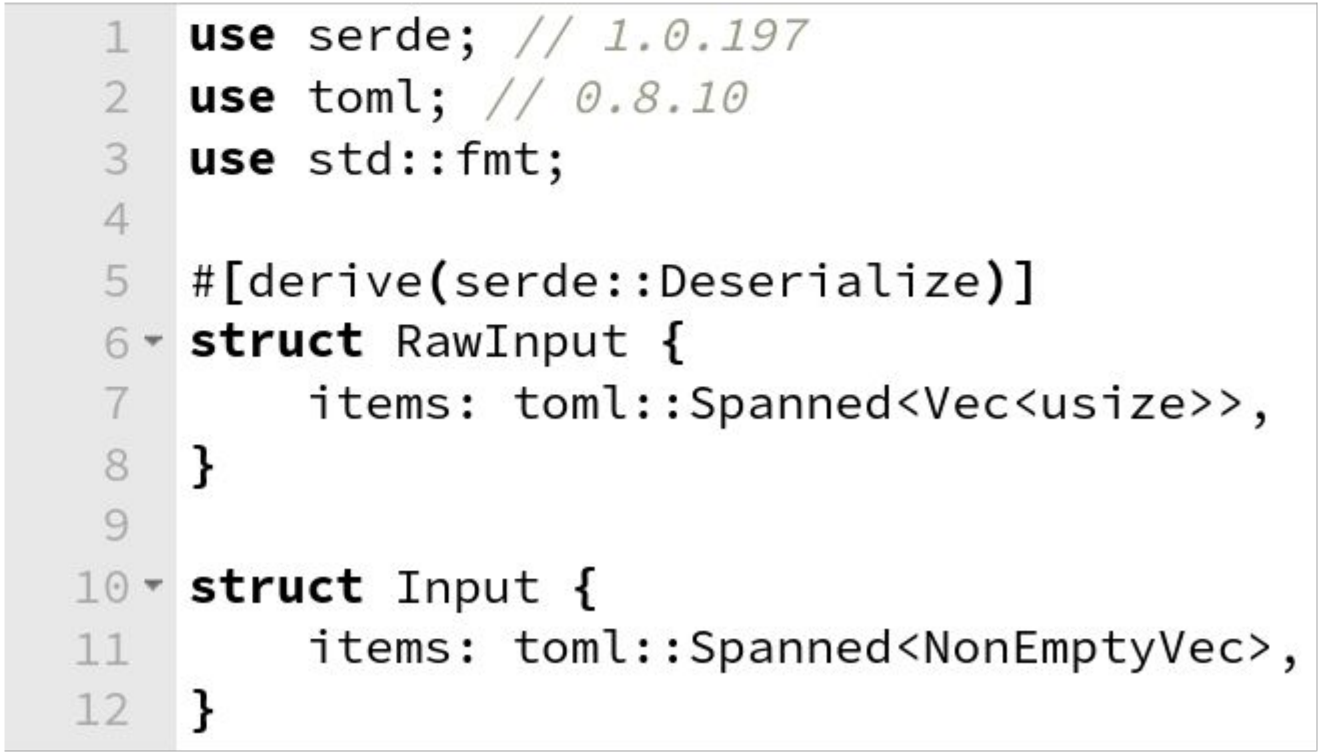
```rust
enum Error {
    NotAuthorized,
    Forbidden(&'static str),
    InvalidData(&'static str),
    NotFound,
    Internal,
    RedirectToLogin(LoginReason),
    RedirectToRegistration,
}
```

```rust
impl From<&'_ app::OpenError> for Error {
    fn from(value: &app::OpenError) -> Self {
        use app::OpenError;

        match value {
            OpenError::NonAdmin => Error::Forbidden("Non-admins are not authorized to open admin-only apps"),
            OpenError::RejectedWithMessage(_) | OpenError::RejectedWithInvalidMessage => Error::Forbidden("You are not allowed to open this application"),
            OpenError::EntryPointExec { .. } | OpenError::EntryPointFailedWithMessage { .. } |  OpenError::EntryPointFailedWithInvalidMessage { .. } |
            OpenError::SystemUserNotFound | OpenError::TaskJoin(_) | OpenError::EntryPointKilledWithMessage { .. } |
            OpenError::EntryPointKilledWithInvalidMessage | OpenError::EntryPointWaitFailed { .. } | OpenError::ReadingStdoutFailed { .. } => Error::Internal,
        }
    }
}
```

```rust
fn log_and_convert<E>(logger: &slog::Logger) -> impl '_ + FnOnce(E) -> Error where E: 'static + std::error::Error, for<'a> &'a E: Into<Error> {
    move |error| {
        let ret = (&error).into();
        error!(logger, "request failed"; "error" => #error);
        ret
    }
}
```

```rust
match (component, request.method()) {
    ("", HttpMethod::Get) | ("/", HttpMethod::Get) => {
        // There's nothing secret here, but redirecting the user immediately is a better
        // UX.
        crate::login::auth_request::<_, S>(&mut user_db, request, logger.clone()).await.map_err(view_auth)?;
        Ok(serve_static::<S, _>(&SafeResourcePath::from_literal("index.html"), Some("text/html"), logger))
    },
    ("/static", HttpMethod::Get) => {
        let path = SafeResourcePath::try_from(remaining.to_owned())
            .map_err(log_and_convert(&logger))?;

        Ok(serve_static::<S, _>(&path, None, logger))
    },
    ("/icons", HttpMethod::Get) => {
        let icon_path = SafeResourcePath::<&str>::try_from(remaining)
            .map_err(log_and_convert(&logger))?;

        let icon_path = icon_path.prefix(app::config::DIRS.app_icons);
        Ok(serve_static_abs::<S, _>(&icon_path, None, logger))
    },
```

Compiler-like application?



```rust
use serde; // 1.0.197
use toml; // 0.8.10
use std::fmt;

#[derive(serde::Deserialize)]
struct RawInput {
    items: toml::Spanned<Vec<usize>>,
}

struct Input {
    items: toml::Spanned<NonEmptyVec>,
}
```

```rust
enum ValidationErrorSource {
    InvalidField { name: String, span: Span, kind: FieldError },
    Duplicates { name: String, first_span: Span, duplicate_spans: Vec<Span> },
    InvalidIdentifier(ident::Error),
    InvalidProgramName { input: String, span: Span }
}
```

Real usage

- configure_me_codegen
- See also github.com/Kixunil/debcrafter

Library?

Give up

DON'T
PANIC!()

Early stage of development?

```rust
#[derive(Debug)]
pub struct ValidationError {
    internal: InternalValidationError,
}

#[derive(Debug)]
pub(crate) enum InternalValidationError {
    Decode(bitcoin::consensus::encode::Error),
    InvalidInputType(InputTypeError),
    InvalidProposedInput(crate::psbt::PrevTxOutError),
    VersionsDontMatch { proposed: i32, original: i32, },
    LockTimesDontMatch { proposed: u32, original: u32, },
    SenderTxinSequenceChanged { proposed: u32, original: u32, },
    SenderTxinContainsNonWitnessUtxo,
    SenderTxinContainsWitnessUtxo,
    SenderTxinContainsFinalScriptSig,
    SenderTxinContainsFinalScriptWitness,
    TxInContainsKeyPaths,
    ContainsPartialSigs,
    ReceiverTxinNotFinalized,
    ReceiverTxinMissingUtxoInfo,
    MixedSequence,
    MixedInputTypes { proposed: InputType, original: InputType, },
    MissingOrShuffledInputs,
    TxOutContainsKeyPaths,
    FeeContributionExceedsMaximum,
    DisallowedOutputSubstitution,
    OutputValueDecreased,
    MissingOrShuffledOutputs,
    Inflation,
    AbsoluteFeeDecreased,
    PayeeTookContributedFee,
    FeeContributionPaysOutputSizeIncrease,
    FeeRateBelowMinimum,
}
```

```rust
/// Error while generating address from script.
#[derive(Debug, Clone, PartialEq, Eq)]
#[non_exhaustive]
pub enum FromScriptError {
    /// Script is not a p2pkh, p2sh or witness program.
    UnrecognizedScript,
    /// A witness program error.
    WitnessProgram(witness_program::Error),
    /// A witness version construction error.
    WitnessVersion(witness_version::TryFromError),
}
```

```rust
/// Hex decoding error.
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum HexToArrayError {
    /// Non-hexadecimal character.
    InvalidChar(InvalidCharError),
    /// Tried to parse fixed-length hash from a string with the wrong length.
    InvalidLength(InvalidLengthError),
}
```

Parsing errors

```rust
/// Error with rich context returned when a string can't be parsed as an integer.
///
/// This is an extension of [`core::num::ParseIntError`], which carries the input that failed to
/// parse as well as type information. As a result it provides very informative error messages that
/// make it easier to understand the problem and correct mistakes.
///
/// Note that this is larger than the type from `core` so if it's passed through a deep call stack
/// in a performance-critical application you may want to box it or throw away the context by
/// converting to `core` type.
#[derive(Debug, Clone, PartialEq, Eq)]
#[non_exhaustive]
pub struct ParseIntError {
    pub(crate) input: String,
    // for displaying - see Display impl with nice error message below
    bits: u8,
    // We could represent this as a single bit but it wouldn't actually derease the cost of moving
    // the struct because String contains pointers so there will be padding of bits at least
    // pointer_size - 1 bytes: min 1B in practice.
    is_signed: bool,
    pub(crate) source: core::num::ParseIntError,
}
```

Error type size

no_std?

```rust
#[cfg(feature = "std")]
impl std::error::Error for FromScriptError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        use FromScriptError::*;

        match *self {
            UnrecognizedScript => None,
            WitnessVersion(ref e) => Some(e),
            WitnessProgram(ref e) => Some(e),
        }
    }
}
```

```rust
/// Formats error.
///
/// If `std` feature is OFF appends error source (delimited by `: `). We do this because
/// `e.source()` is only available in std builds, without this macro the error source is lost for
/// no-std builds.
#[macro_export]
macro_rules! write_err {
    ($writer:expr, $string:literal $(, $args:expr)*; $source:expr) => {
        {
            #[cfg(feature = "std")]
            {
                let _ = &$source;   // Prevents clippy warnings.
                write!($writer, $string $(, $args)*)
            }
            #[cfg(not(feature = "std"))]
            {
                write!($writer, concat!($string, ": {}") $(, $args)*, $source)
            }
        }
    }
}
```

```rust
impl fmt::Display for FromScriptError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        use FromScriptError::*;

        match *self {
            WitnessVersion(ref e) => write_err!(f, "witness version construction error"; e),
            WitnessProgram(ref e) => write_err!(f, "witness program error"; e),
            UnrecognizedScript => write!(f, "script is not a p2pkh, p2sh or witness program"),
        }
    }
}
```

no-alloc?

```rust
use storage::Storage;

/// Conditionally stores the input string in parse errors.
///
/// This type stores the input string of a parse function depending on whether `alloc` feature is
/// enabled. When it is enabled, the string is stored inside as `String`. When disabled this is a
/// zero-sized type and attempt to store a string does nothing.
///
/// This provides two methods to format the error strings depending on the context.
#[derive(Debug, Clone, Eq, PartialEq, Hash, Ord, PartialOrd)]
pub struct InputString(Storage);
```

```rust
impl fmt::Display for ParseError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // Outputs "failed to parse '<input string>' as foo"
        write_err!(f, "{}", self.input.display_cannot_parse("foo"); self.error)
    }
}
```

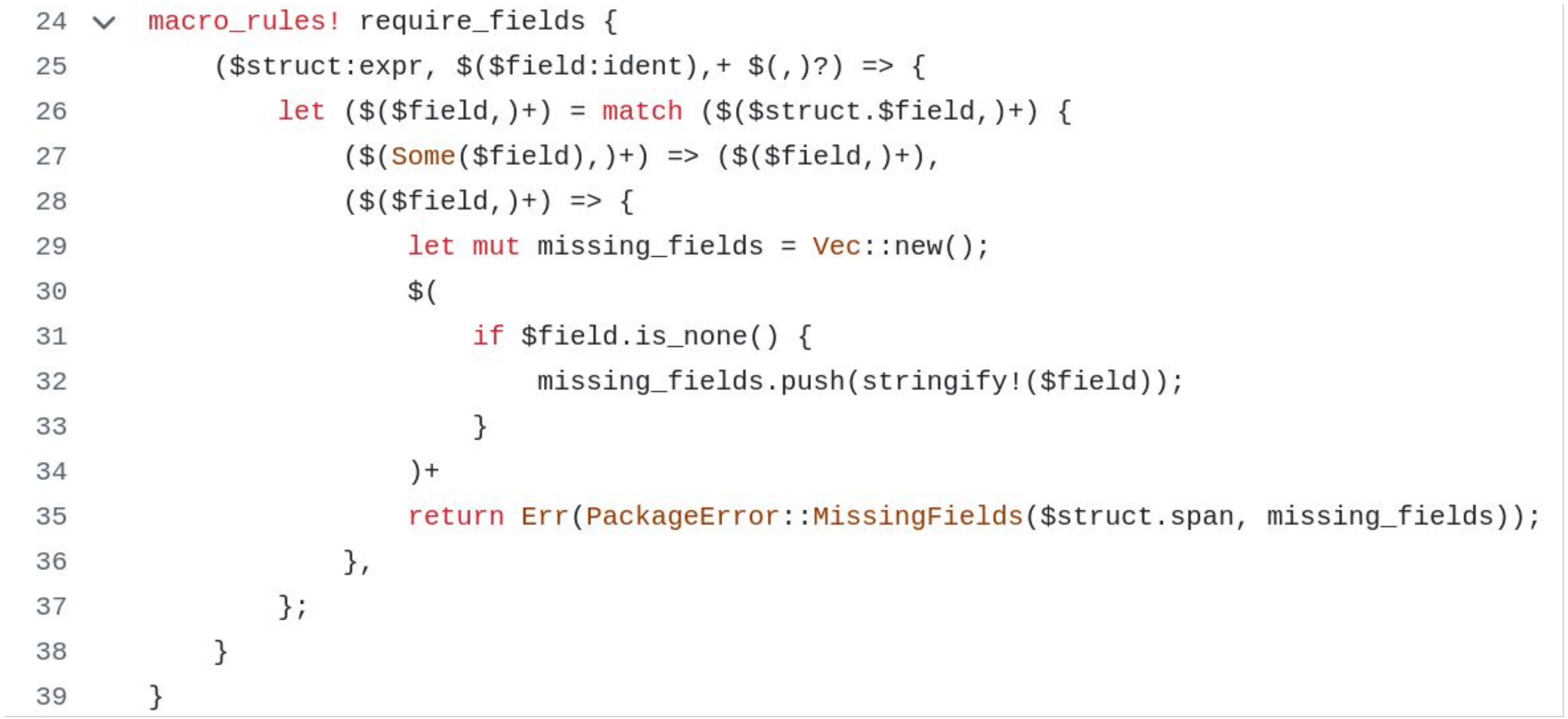
Limit string length?

7172f4021133f04608a79ee78a0592a0O9e9d127bd101c881537409b05e573e6

… 92a0O9e9d1 …

Multiple errors

Vec<Error>



```rust
macro_rules! require_fields {
    ($struct:expr, $($field:ident),+ $(,)?) => {
        let ($($field,)+) = match ($($struct.$field,)+) {
            ($(Some($field),)+) => ($($field,)+),
            ($($field,)+) => {
                let mut missing_fields = Vec::new();
                $(
                    if $field.is_none() {
                        missing_fields.push(stringify!($field));
                    }
                )+
                return Err(PackageError::MissingFields($struct.span, missing_fields));
            },
        };
    }
}
```

```rust
pub struct Error {
    first: ErrorKind,
    #[cfg(feature = "multi-error")]
    remaining: Vec<ErrorKind>,
}

impl Error {
    pub fn first(&self) -> &ErrorKind {
        &self.first
    }

    pub fn all(&self) -> impl Iterator<Item=&ErrorKind> + '_ {
        #[cfg(not(feature = "multi-error"))]
        {
            core::iter::once(&self.first)
        }
        #[cfg(feature = "multi-error")]
        {
            core::iter::once(&self.first).chain(&self.remaining)
        }
    }
}

#[non_exhaustive]
pub enum ErrorKind {
    InvalidChar(InvalidCharError),
    UnknownName(UnknownNameError),
}

pub struct InvalidCharError {
    c: char,
    pos: usize,
}

pub struct UnknownNameError {
    name: String,
    pos: usize,
}
```

```rust
fn validate_opt<T: TryInto<U>, U>(field: Option<T>, errors: &mut Vec<T::Error>) -> Option<U> {
    field.map(TryInto::try_into).transpose().unwrap_or_else(|error| {
        errors.push(error);
        None
    })
}
```