Error Handling in Rust

May 14, 2015

Like most programming languages, Rust encourages the programmer to handle errors in a particular way. Generally speaking, error handling is divided into two broad categories: exceptions and return values. Rust opts for return values.

In this article, I intend to provide a comprehensive treatment of how to deal with errors in Rust. More than that, I will attempt to introduce error handling one piece at a time so that you'll come away with a solid working knowledge of how everything fits together.

When done naively, error handling in Rust can be verbose and annoying. This article will explore those stumbling blocks and demonstrate how to use the standard library to make error handling concise and ergonomic.

Target audience: Those new to Rust that don't know its error handling idioms yet. Some familiarity with Rust is helpful. (This article makes heavy use of some standard traits and some very light use of closures and macros.)

Update (2018/04/14): Examples were converted to `?`, and some text was added to give historical context on the change.

Update (2020/01/03): A recommendation to use `failure` was removed and replaced with a recommendation to use either `Box<Error + Send + Sync>` or `anyhow`.

Brief notes

All code samples in this post compile with Rust `1.0.0-beta.5`. They should continue to work as Rust 1.0 stable is released.

All code can be found and compiled in my blog's repository.

The Rust Book has a section on error handling. It gives a very brief overview, but doesn't (yet) go into enough detail, particularly when working with some of the more recent additions to the standard library.

Run the code!

If you'd like to run any of the code samples below, then the following should work:

```
$ git clone git://github.com/BurntSushi/blog
$ cd blog/code/rust-error-handling
$ cargo run --bin NAME-OF-CODE-SAMPLE [ args ... ]
```

Each code sample is labeled with its name. (Code samples without a name aren't available to be run this way. Sorry.)

Table of Contents

This article is very long, mostly because I start at the very beginning with sum types and combinators, and try to motivate the way Rust does error handling incrementally. As such, programmers with experience in other expressive type systems may want to jump around. Here's my very brief guide:

- If you're new to Rust, systems programming and expressive type systems, then start at the beginning and work your way through. (If you're brand new, you should probably read through the Rust book first.)
- If you've never seen Rust before but have experience with functional languages ("algebraic data types" and "combinators" make you feel warm and fuzzy), then you can probably skip right over the basics and start by skimming multiple error types, and work your way into a full read of standard library error traits. (Skimming the basics might be a good idea to just get a feel for the syntax if you've really never seen Rust before.) You may need to consult the Rust book for help with Rust closures and macros.
- If you're already experienced with Rust and just want the skinny on error handling, then you can probably skip straight to the end. You may find it useful to skim the case study for examples.

The Basics

I like to think of error handling as using *case analysis* to determine whether a computation was successful or not. As we will see, the key to ergonomic error handling is reducing the amount of explicit case analysis the programmer has to do while keeping code composable.

Keeping code composable is important, because without that requirement, we could `panic` whenever we come across something unexpected. (`panic`

causes the current task to unwind, and in most cases, the entire program aborts.) Here's an example:

panic-simple

```
// Guess a number between 1 and 10.
// If it matches the number I had in mind, return true. Else
fn guess(n: i32) -> bool {
    if n < 1 || n > 10 {
        panic!("Invalid number: {}", n);
    }
    n == 5
}

fn main() {
    guess(11);
}
```

(If you like, it's easy to run this code.)

If you try running this code, the program will crash with a message like this:

```
thread '<main>' panicked at 'Invalid number: 11', src/bin/pa
```

Here's another example that is slightly less contrived. A program that accepts an integer as an argument, doubles it and prints it.

unwrap-double

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // error 1
    let n: i32 = arg.parse().unwrap(); // error 2
    println!("{}", 2 * n);
}

// $ cargo run --bin unwrap-double 5
```

```
// 10
```

If you give this program zero arguments (error 1) or if the first argument isn't an integer (error 2), the program will panic just like in the first example.

I like to think of this style of error handling as similar to a bull running through a china shop. The bull will get to where it wants to go, but it will trample everything in the process.

Unwrapping explained

In the previous example (`unwrap-double`), I claimed that the program would simply panic if it reached one of the two error conditions, yet, the program does not include an explicit call to `panic` like the first example (`panic-simple`). This is because the panic is embedded in the calls to `unwrap`.

To "unwrap" something in Rust is to say, "Give me the result of the computation, and if there was an error, just panic and stop the program." It would be better if I just showed the code for unwrapping because it is so simple, but to do that, we will first need to explore the `Option` and `Result` types. Both of these types have a method called `unwrap` defined on them.

The `Option` type

The `Option` type is defined in the standard library:

`option-def`

```
enum Option<T> {
    None,
    Some(T),
}
```

The `Option` type is a way to use Rust's type system to express the *possibility of absence*. Encoding the possibility of absence into the type system is an important concept because it will cause the compiler to force the programmer to handle that absence. Let's take a look at an example that tries to find a character in a string:

option-ex-string-find

```
// Searches `haystack` for the Unicode character `needle`. I
// byte offset of the character is returned. Otherwise, `Non
fn find(haystack: &str, needle: char) -> Option<usize> {
    for (offset, c) in haystack.char_indices() {
        if c == needle {
            return Some(offset);
        }
    }
    None
}
```

(Pro-tip: don't use this code. Instead, use the `find` method from the standard library.)

Notice that when this function finds a matching character, it doesn't just return the `offset`. Instead, it returns `Some(offset)`. `Some` is a variant or a *value constructor* for the `Option` type. You can think of it as a function with the type `fn<T>(value: T) -> Option<T>`. Correspondingly, `None` is also a value constructor, except it has no arguments. You can think of `None` as a function with the type `fn<T>() -> Option<T>`.

This might seem like much ado about nothing, but this is only half of the story. The other half is *using* the `find` function we've written. Let's try to use it to find the extension in a file name.

option-ex-string-find

```
fn main_find() {
    let file_name = "foobar.rs";
    match find(file_name, '.') {
        None => println!("No file extension found."),
        Some(i) => println!("File extension: {}", &file_name
    }
}
```

This code uses pattern matching to do *case analysis* on the `Option<usize>` returned by the `find` function. In fact, case analysis is the

only way to get at the value stored inside an `Option<T>`. This means that you, as the programmer, must handle the case when an `Option<T>` is None instead of `Some(t)`.

But wait, what about unwrap used in `unwrap-double`? There was no case analysis there! Instead, the case analysis was put inside the unwrap method for you. You could define it yourself if you want:

option-def-unwrap

```
enum Option<T> {
    None,
    Some(T),
}

impl<T> Option<T> {
    fn unwrap(self) -> T {
        match self {
            Option::Some(val) => val,
            Option::None =>
              panic!("called `Option::unwrap()` on a `None`
        }
    }
}
```

The unwrap method *abstracts away the case analysis*. This is precisely the thing that makes unwrap ergonomic to use. Unfortunately, that `panic!` means that unwrap is not composable: it is the bull in the china shop.

Composing `Option<T>` values

In `option-ex-string-find` we saw how to use `find` to discover the extension in a file name. Of course, not all file names have a `.` in them, so it's possible that the file name has no extension. This *possibility of absence* is encoded into the types using `Option<T>`. In other words, the compiler will force us to address the possibility that an extension does not exist. In our case, we just print out a message saying as such.

Getting the extension of a file name is a pretty common operation, so it

makes sense to put it into a function:

option-ex-string-find

```
// Returns the extension of the given file name, where the e
// as all characters succeeding the first `.`.
// If `file_name` has no `.`, then `None` is returned.
fn extension_explicit(file_name: &str) -> Option<&str> {
    match find(file_name, '.') {
        None => None,
        Some(i) => Some(&file_name[i+1..]),
    }
}
```

(Pro-tip: don't use this code. Use the `extension` method in the standard library instead.)

The code stays simple, but the important thing to notice is that the type of `find` forces us to consider the possibility of absence. This is a good thing because it means the compiler won't let us accidentally forget about the case where a file name doesn't have an extension. On the other hand, doing explicit case analysis like we've done in `extension_explicit` every time can get a bit tiresome.

In fact, the case analysis in `extension_explicit` follows a very common pattern: *map* a function on to the value inside of an `Option<T>`, unless the option is None, in which case, just return None.

Rust has parametric polymorphism, so it is very easy to define a combinator that abstracts this pattern:

option-map

```
fn map<F, T, A>(option: Option<T>, f: F) -> Option<A> where
    match option {
        None => None,
        Some(value) => Some(f(value)),
    }
}
```

Indeed, `map` is defined as a method on `Option<T>` in the standard library.

Armed with our new combinator, we can rewrite our `extension_explicit` method to get rid of the case analysis:

`option-ex-string-find`

```
// Returns the extension of the given file name, where the e
// as all characters succeeding the first `.`.
// If `file_name` has no `.`, then `None` is returned.
fn extension(file_name: &str) -> Option<&str> {
    find(file_name, '.').map(|i| &file_name[i+1..])
}
```

One other pattern that I find is very common is assigning a default value to the case when an `Option` value is None. For example, maybe your program assumes that the extension of a file is `rs` even if none is present. As you might imagine, the case analysis for this is not specific to file extensions—it can work with any `Option<T>`:

`option-unwrap-or`

```
fn unwrap_or<T>(option: Option<T>, default: T) -> T {
    match option {
        None => default,
        Some(value) => value,
    }
}
```

The trick here is that the default value must have the same type as the value that might be inside the `Option<T>`. Using it is dead simple in our case:

`option-ex-string-find`

```
fn main() {
    assert_eq!(extension("foobar.csv").unwrap_or("rs"), "csv
    assert_eq!(extension("foobar").unwrap_or("rs"), "rs");
}
```

(Note that `unwrap_or` is defined as a method on `Option<T>` in the standard library, so we use that here instead of the free-standing function we defined above. Don't forget to check out the more general `unwrap_or_else` method.)

There is one more combinator that I think is worth paying special attention to: and_then. It makes it easy to compose distinct computations that admit the *possibility of absence*. For example, much of the code in this section is about finding an extension given a file name. In order to do this, you first need the file name which is typically extracted from a file *path*. While most file paths have a file name, not *all* of them do. For example, `.`, `..` or `/`.

So, we are tasked with the challenge of finding an extension given a file *path*. Let's start with explicit case analysis:

`option-ex-string-find`

```
fn file_path_ext_explicit(file_path: &str) -> Option<&str> {
    match file_name(file_path) {
        None => None,
        Some(name) => match extension(name) {
            None => None,
            Some(ext) => Some(ext),
        }
    }
}

fn file_name(file_path: &str) -> Option<&str> {
  // implementation elided
  unimplemented!()
}
```

You might think that we could just use the `map` combinator to reduce the case analysis, but its type doesn't quite fit. Namely, `map` takes a function that does something only with the inner value. The result of that function is then *always* rewrapped with Some. Instead, we need something like `map`, but which allows the caller to return another `Option`. Its generic implementation is even simpler than `map`:

option-and-then

```
fn and_then<F, T, A>(option: Option<T>, f: F) -> Option<A>
        where F: FnOnce(T) -> Option<A> {
    match option {
        None => None,
        Some(value) => f(value),
    }
}
```

Now we can rewrite our `file_path_ext` function without explicit case analysis:

option-ex-string-find

```
fn file_path_ext(file_path: &str) -> Option<&str> {
    file_name(file_path).and_then(extension)
}
```

The `Option` type has many other combinators defined in the standard library. It is a good idea to skim this list and familiarize yourself with what's available—they can often reduce case analysis for you. Familiarizing yourself with these combinators will pay dividends because many of them are also defined (with similar semantics) for `Result`, which we will talk about next.

Combinators make using types like `Option` ergonomic because they reduce explicit case analysis. They are also composable because they permit the caller to handle the possibility of absence in their own way. Methods like `unwrap` remove choices because they will panic if `Option<T>` is None.

The `Result` type

The `Result` type is also defined in the standard library:

result-def

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

```
    Err(E),
}
```

The `Result` type is a richer version of `Option`. Instead of expressing the possibility of *absence* like `Option` does, `Result` expresses the possibility of *error*. Usually, the *error* is used to explain why the result of some computation failed. This is a strictly more general form of `Option`. Consider the following type alias, which is semantically equivalent to the real `Option<T>` in every way:

option-as-result

```
type Option<T> = Result<T, ()>;
```

This fixes the second type parameter of `Result` to always be `()` (pronounced "unit" or "empty tuple"). Exactly one value inhabits the `()` type: `()`. (Yup, the type and value level terms have the same notation!)

The `Result` type is a way of representing one of two possible outcomes in a computation. By convention, one outcome is meant to be expected or "`Ok`" while the other outcome is meant to be unexpected or "`Err`".

Just like `Option`, the `Result` type also has an unwrap method defined in the standard library. Let's define it:

result-def

```
impl<T, E: ::std::fmt::Debug> Result<T, E> {
    fn unwrap(self) -> T {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) =>
              panic!("called `Result::unwrap()` on an `Err`
        }
    }
}
```

This is effectively the same as our definition for `Option::unwrap`, except it includes the error value in the `panic!` message. This makes debugging

easier, but it also requires us to add a Debug constraint on the E type parameter (which represents our error type). Since the vast majority of types should satisfy the Debug constraint, this tends to work out in practice. (Debug on a type simply means that there's a reasonable way to print a human readable description of values with that type.)

OK, let's move on to an example.

Parsing integers

The Rust standard library makes converting strings to integers dead simple. It's so easy in fact, that it is very tempting to write something like the following:

result-num-unwrap

```
fn double_number(number_str: &str) -> i32 {
    2 * number_str.parse::<i32>().unwrap()
}

fn main() {
    let n: i32 = double_number("10");
    assert_eq!(n, 20);
}
```

At this point, you should be skeptical of calling unwrap. For example, if the string doesn't parse as a number, you'll get a panic:

```
thread '<main>' panicked at 'called `Result::unwrap()` on an
```

This is rather unsightly, and if this happened inside a library you're using, you might be understandably annoyed. Instead, we should try to handle the error in our function and let the caller decide what to do. This means changing the return type of double_number. But to what? Well, that requires looking at the signature of the parse method in the standard library:

```
impl str {
```

```
    fn parse<F: FromStr>(&self) -> Result<F, F::Err>;
}
```

Hmm. So we at least know that we need to use a `Result`. Certainly, it's possible that this could have returned an `Option`. After all, a string either parses as a number or it doesn't, right? That's certainly a reasonable way to go, but the implementation internally distinguishes *why* the string didn't parse as an integer. (Whether it's an empty string, an invalid digit, too big or too small.) Therefore, using a `Result` makes sense because we want to provide more information than simply "absence." We want to say *why* the parsing failed. You should try to emulate this line of reasoning when faced with a choice between `Option` and `Result`. If you can provide detailed error information, then you probably should. (We'll see more on this later.)

OK, but how do we write our return type? The `parse` method as defined above is generic over all the different number types defined in the standard library. We could (and probably should) also make our function generic, but let's favor explicitness for the moment. We only care about `i32`, so we need to find its implementation of `FromStr` (do a `CTRL-F` in your browser for "FromStr") and look at its associated type `Err`. We did this so we can find the concrete error type. In this case, it's `std::num::ParseIntError`. Finally, we can rewrite our function:

`result-num-no-unwrap`

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntEr
    match number_str.parse::<i32>() {
        Ok(n) => Ok(2 * n),
        Err(err) => Err(err),
    }
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
```

```
    }
}
```

This is a little better, but now we've written a lot more code! The case analysis has once again bitten us.

Combinators to the rescue! Just like `Option`, `Result` has lots of combinators defined as methods. There is a large intersection of common combinators between `Result` and `Option`. In particular, `map` is part of that intersection:

`result-num-no-unwrap-map`

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntEr
    number_str.parse::<i32>().map(|n| 2 * n)
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

The usual suspects are all there for `Result`, including `unwrap_or` and `and_then`. Additionally, since `Result` has a second type parameter, there are combinators that affect only the error type, such as `map_err` (instead of `map`) and `or_else` (instead of `and_then`).

The `Result` type alias idiom

In the standard library, you may frequently see types like `Result<i32>`. But wait, we defined `Result` to have two type parameters. How can we get away with only specifying one? The key is to define a `Result` type alias that *fixes* one of the type parameters to a particular type. Usually the fixed type is the error type. For example, our previous example parsing integers could be rewritten like this:

result-num-no-unwrap-map-alias

```
use std::num::ParseIntError;
use std::result;

type Result<T> = result::Result<T, ParseIntError>;

fn double_number(number_str: &str) -> Result<i32> {
    unimplemented!();
}
```

Why would we do this? Well, if we have a lot of functions that could return `ParseIntError`, then it's much more convenient to define an alias that always uses `ParseIntError` so that we don't have to write it out all the time.

The most prominent place this idiom is used in the standard library is with `io::Result`. Typically, one writes `io::Result<T>`, which makes it clear that you're using the `io` module's type alias instead of the plain definition from `std::result`. (This idiom is also used for `fmt::Result`.)

A brief interlude: unwrapping isn't evil

If you've been following along, you might have noticed that I've taken a pretty hard line against calling methods like `unwrap` that could `panic` and abort your program. *Generally speaking*, this is good advice.

However, `unwrap` can still be used judiciously. What exactly justifies use of `unwrap` is somewhat of a grey area and reasonable people can disagree. I'll summarize some of my *opinions* on the matter.

- **In examples and quick 'n' dirty code.** Sometimes you're writing examples or a quick program, and error handling simply isn't important. Beating the convenience of `unwrap` can be hard in such scenarios, so it is very appealing.
- **When panicking indicates a bug in the program.** When the invariants of your code should prevent a certain case from happening (like, say, popping from an empty stack), then panicking can be permissible. This is because it exposes a bug in your program. This can be explicit, like from an `assert!` failing, or it could be because your index into an array

was out of bounds.

This is probably not an exhaustive list. Moreover, when using an `Option`, it is often better to use its `expect` method. `expect` does exactly the same thing as `unwrap`, except it prints a message you give to `expect`. This makes the resulting panic a bit nicer to deal with, since it will show your message instead of "called unwrap on a None value."

My advice boils down to this: use good judgment. There's a reason why the words "never do X" or "Y is considered harmful" don't appear in my writing. There are trade offs to all things, and it is up to you as the programmer to determine what is acceptable for your use cases. My goal is only to help you evaluate trade offs as accurately as possible.

Now that we've covered the basics of error handling in Rust, and I've said my piece about unwrapping, let's start exploring more of the standard library.

Working with multiple error types

Thus far, we've looked at error handling where everything was either an `Option<T>` or a `Result<T, SomeError>`. But what happens when you have both an `Option` and a `Result`? Or what if you have a `Result<T, Error1>` and a `Result<T, Error2>`? Handling *composition of distinct error types* is the next challenge in front of us, and it will be the major theme throughout the rest of this article.

Composing `Option` and `Result`

So far, I've talked about combinators defined for `Option` and combinators defined for `Result`. We can use these combinators to compose results of different computations without doing explicit case analysis.

Of course, in real code, things aren't always as clean. Sometimes you have a mix of `Option` and `Result` types. Must we resort to explicit case analysis, or can we continue using combinators?

For now, let's revisit one of the first examples in this article:

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // error 1
    let n: i32 = arg.parse().unwrap(); // error 2
    println!("{}", 2 * n);
}

// $ cargo run --bin unwrap-double 5
// 10
```

Given our new found knowledge of `Option`, `Result` and their various combinators, we should try to rewrite this so that errors are handled properly and the program doesn't panic if there's an error.

The tricky aspect here is that `argv.nth(1)` produces an `Option` while `arg.parse()` produces a `Result`. These aren't directly composable. When faced with both an `Option` and a `Result`, the solution is *usually* to convert the `Option` to a `Result`. In our case, the absence of a command line parameter (from `env::args()`) means the user didn't invoke the program correctly. We could just use a `String` to describe the error. Let's try:

`error-double-string`

```
use std::env;

fn double_arg(mut argv: env::Args) -> Result<i32, String> {
    argv.nth(1)
        .ok_or("Please give at least one argument".to_owned(
        .and_then(|arg| arg.parse::<i32>().map_err(|err| err
        .map(|i| i * 2)
}

fn main() {
    match double_arg(env::args()) {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
```

```
    }
}
```

There are a couple new things in this example. The first is the use of the `Option::ok_or` combinator. This is one way to convert an `Option` into a `Result`. The conversion requires you to specify what error to use if `Option` is None. Like the other combinators we've seen, its definition is very simple:

option-ok-or-def

```
fn ok_or<T, E>(option: Option<T>, err: E) -> Result<T, E> {
    match option {
        Some(val) => Ok(val),
        None => Err(err),
    }
}
```

The other new combinator used here is `Result::map_err`. This is just like `Result::map`, except it maps a function on to the *error* portion of a `Result` value. If the `Result` is an `Ok(...)` value, then it is returned unmodified.

We use `map_err` here because it is necessary for the error types to remain the same (because of our use of and_then). Since we chose to convert the `Option<String>` (from `argv.nth(1)`) to a `Result<String, String>`, we must also convert the `ParseIntError` from `arg.parse()` to a `String`.

The limits of combinators

Doing IO and parsing input is a very common task, and it's one that I personally have done a lot of in Rust. Therefore, we will use (and continue to use) IO and various parsing routines to exemplify error handling.

Let's start simple. We are tasked with opening a file, reading all of its contents and converting its contents to a number. Then we multiply it by 2 and print the output.

Although I've tried to convince you not to use `unwrap`, it can be useful to first write your code using `unwrap`. It allows you to focus on your problem instead of the error handling, and it exposes the points where proper error

handling need to occur. Let's start there so we can get a handle on the code, and then refactor it to use better error handling.

io-basic-unwrap

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> i32 {
    let mut file = File::open(file_path).unwrap(); // error
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap(); // error 2
    let n: i32 = contents.trim().parse().unwrap(); // error
    2 * n
}

fn main() {
    let doubled = file_double("foobar");
    println!("{}", doubled);
}
```

(N.B. The `AsRef<Path>` is used because those are the same bounds used on `std::fs::File::open`. This makes it ergnomic to use any kind of string as a file path.)

There are three different errors that can occur here:

1. A problem opening the file.
2. A problem reading data from the file.
3. A problem parsing the data as a number.

The first two problems are described via the `std::io::Error` type. We know this because of the return types of `std::fs::File::open` and `std::io::Read::read_to_string`. (Note that they both use the `Result` type alias idiom described previously. If you click on the `Result` type, you'll see the type alias, and consequently, the underlying `io::Error` type.) The third problem is described by the `std::num::ParseIntError` type. The `io::Error` type in particular is *pervasive* throughout the standard library. You will see it again and again.

Let's start the process of refactoring the `file_double` function. To make this function composable with other components of the program, it should *not* panic if any of the above error conditions are met. Effectively, this means that the function should *return an error* if any of its operations fail. Our problem is that the return type of `file_double` is `i32`, which does not give us any useful way of reporting an error. Thus, we must start by changing the return type from `i32` to something else.

The first thing we need to decide: should we use `Option` or `Result`? We certainly could use `Option` very easily. If any of the three errors occur, we could simply return None. This will work *and it is better than panicking*, but we can do a lot better. Instead, we should pass some detail about the error that occurred. Since we want to express the *possibility of error*, we should use `Result<i32, E>`. But what should `E` be? Since two *different* types of errors can occur, we need to convert them to a common type. One such type is `String`. Let's see how that impacts our code:

`io-basic-error-string`

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32,
    File::open(file_path)
        .map_err(|err| err.to_string())
        .and_then(|mut file| {
            let mut contents = String::new();
            file.read_to_string(&mut contents)
                .map_err(|err| err.to_string())
                .map(|_| contents)
        })
        .and_then(|contents| {
            contents.trim().parse::<i32>()
                    .map_err(|err| err.to_string())
        })
        .map(|n| 2 * n)
}
```

```
fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

This code looks a bit hairy. It can take quite a bit of practice before code like this becomes easy to write. The way I write it is by *following the types*. As soon as I changed the return type of `file_double` to `Result<i32, String>`, I had to start looking for the right combinators. In this case, we only used three different combinators: and_then, `map` and `map_err`.

and_then is used to chain multiple computations where each computation could return an error. After opening the file, there are two more computations that could fail: reading from the file and parsing the contents as a number. Correspondingly, there are two calls to and_then.

`map` is used to apply a function to the `Ok(...)` value of a `Result`. For example, the very last call to `map` multiplies the `Ok(...)` value (which is an `i32`) by 2. If an error had occurred before that point, this operation would have been skipped because of how `map` is defined.

`map_err` is the trick the makes all of this work. `map_err` is just like `map`, except it applies a function to the `Err(...)` value of a `Result`. In this case, we want to convert all of our errors to one type: `String`. Since both `io::Error` and `num::ParseIntError` implement `ToString`, we can call the `to_string()` method to convert them.

With all of that said, the code is still hairy. Mastering use of combinators is important, but they have their limits. Let's try a different approach: early returns.

Early returns

I'd like to take the code from the previous section and rewrite it using *early returns*. Early returns let you exit the function early. We can't return early in `file_double` from inside another closure, so we'll need to revert back to explicit case analysis.

io-basic-error-string-early-return

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32,
    let mut file = match File::open(file_path) {
        Ok(file) => file,
        Err(err) => return Err(err.to_string()),
    };
    let mut contents = String::new();
    if let Err(err) = file.read_to_string(&mut contents) {
        return Err(err.to_string());
    }
    let n: i32 = match contents.trim().parse() {
        Ok(n) => n,
        Err(err) => return Err(err.to_string()),
    };
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

Reasonable people can disagree over whether this code is better than the code that uses combinators, but if you aren't familiar with the combinator approach, this code looks simpler to read to me. It uses explicit case analysis with `match` and `if let`. If an error occurs, it simply stops executing the function and returns the error (by converting it to a string).

Isn't this a step backwards though? Previously, I said that the key to ergonomic error handling is reducing explicit case analysis, yet we've reverted back to explicit case analysis here. It turns out, there are *multiple*

ways to reduce explicit case analysis. Combinators aren't the only way.

The `try!` macro/`?` operator

In older versions of Rust (Rust 1.12 or older), a cornerstone of error handling in Rust is the `try!` macro. The `try!` macro abstracts case analysis just like combinators, but unlike combinators, it also abstracts *control flow*. Namely, it can abstract the *early return* pattern seen above.

Here is a simplified definition of a `try!` macro:

`try-def-simple`

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

(The real definition is a bit more sophisticated. We will address that later.)

Using the `try!` macro makes it very easy to simplify our last example. Since it does the case analysis and the early return for us, we get tighter code that is easier to read:

`io-basic-error-try`

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32,
    let mut file = try!(File::open(file_path).map_err(|e| e.
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to
    let n = try!(contents.trim().parse::<i32>().map_err(|e|
    Ok(2 * n)
}
```

```
fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

The `map_err` calls are still necessary given our definition of `try!`. This is because the error types still need to be converted to `String`. The good news is that we will soon learn how to remove those `map_err` calls! The bad news is that we will need to learn a bit more about a couple important traits in the standard library before we can remove the `map_err` calls.

In newer versions of Rust (Rust 1.13 or newer), the `try!` macro was replaced with the `?` operator. While it is intended to grow new powers that we won't cover here, using `?` instead of `try!` is simple:

`io-basic-error-question`

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32,
    let mut file = File::open(file_path).map_err(|e| e.to_st
    let mut contents = String::new();
    file.read_to_string(&mut contents).map_err(|e| e.to_stri
    let n = contents.trim().parse::<i32>().map_err(|e| e.to_
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

Defining your own error type

Before we dive into some of the standard library error traits, I'd like to wrap up this section by removing the use of `String` as our error type in the previous examples.

Using `String` as we did in our previous examples is convenient because it's easy to convert errors to strings, or even make up your own errors as strings on the spot. However, using `String` for your errors has some downsides.

The first downside is that the error messages tend to clutter your code. It's possible to define the error messages elsewhere, but unless you're unusually disciplined, it is very tempting to embed the error message into your code. Indeed, we did exactly this in a previous example.

The second and more important downside is that `String`s are *lossy*. That is, if all errors are converted to strings, then the errors we pass to the caller become completely opaque. The only reasonable thing the caller can do with a `String` error is show it to the user. Certainly, inspecting the string to determine the type of error is not robust. (Admittedly, this downside is far more important inside of a library as opposed to, say, an application.)

For example, the `io::Error` type embeds an `io::ErrorKind`, which is *structured data* that represents what went wrong during an IO operation. This is important because you might want to react differently depending on the error. (e.g., A `BrokenPipe` error might mean quitting your program gracefully while a `NotFound` error might mean exiting with an error code and showing an error to the user.) With `io::ErrorKind`, the caller can examine the type of an error with case analysis, which is strictly superior to trying to tease out the details of an error inside of a `String`.

Instead of using a `String` as an error type in our previous example of reading an integer from a file, we can define our own error type that represents errors with *structured data*. We endeavor to not drop information from underlying errors in case the caller wants to inspect the details.

The ideal way to represent *one of many possibilities* is to define our own sum type using `enum`. In our case, an error is either an `io::Error` or a `num::ParseIntError`, so a natural definition arises:

io-basic-error-custom

```
use std::io;
use std::num;

// We derive `Debug` because all types should probably deriv
// This gives us a reasonable human readable description of
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

Tweaking our code is very easy. Instead of converting errors to strings, we simply convert them to our `CliError` type using the corresponding value constructor:

io-basic-error-custom

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32,
    let mut file = File::open(file_path).map_err(CliError::I
    let mut contents = String::new();
    file.read_to_string(&mut contents).map_err(CliError::Io)
    let n: i32 = contents.trim().parse().map_err(CliError::P
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

The only change here is switching `map_err(|e| e.to_string())` (which converts errors to strings) to `map_err(CliError::Io)` or `map_err(CliError::Parse)`. The *caller* gets to decide the level of detail to report to the user. In effect, using a `String` as an error type removes choices from the caller while using a custom `enum` error type like `CliError` gives the caller all of the conveniences as before in addition to *structured data* describing the error.

A rule of thumb is to define your own error type, but a `String` error type will do in a pinch, particularly if you're writing an application. If you're writing a library, defining your own error type should be strongly preferred so that you don't remove choices from the caller unnecessarily.

Standard library traits used for error handling

The standard library defines two integral traits for error handling: `std::error::Error` and `std::convert::From`. While `Error` is designed specifically for generically describing errors, the `From` trait serves a more general role for converting values between two distinct types.

The `Error` trait

The `Error` trait is defined in the standard library:

error-def

```
use std::fmt::{Debug, Display};

trait Error: Debug + Display {
  /// A short description of the error.
  fn description(&self) -> &str;

  /// The lower level cause of this error, if any.
  fn cause(&self) -> Option<&Error> { None }
}
```

This trait is super generic because it is meant to be implemented for *all*

types that represent errors. This will prove useful for writing composable code as we'll see later. Otherwise, the trait allows you to do at least the following things:

- Obtain a Debug representation of the error.
- Obtain a user-facing `Display` representation of the error.
- Obtain a short description of the error (via the `description` method).
- Inspect the causal chain of an error, if one exists (via the `cause` method).

The first two are a result of `Error` requiring impls for both Debug and `Display`. The latter two are from the two methods defined on `Error`. The power of `Error` comes from the fact that all error types impl `Error`, which means errors can be existentially quantified as a trait object. This manifests as either `Box<Error>` or `&Error`. Indeed, the `cause` method returns an `&Error`, which is itself a trait object. We'll revisit the `Error` trait's utility as a trait object later.

For now, it suffices to show an example implementing the `Error` trait. Let's use the error type we defined in the previous section:

`error-impl`

```
use std::io;
use std::num;

// We derive `Debug` because all types should probably deriv
// This gives us a reasonable human readable description of
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

This particular error type represents the possibility of two types of errors occurring: an error dealing with I/O or an error converting a string to a number. The error could represent as many error types as you want by adding new variants to the `enum` definition.

Implementing `Error` is pretty straight-forward. It's mostly going to be a lot

explicit case analysis.

`error-impl`

```
use std::error;
use std::fmt;

impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            // Both underlying errors already impl `Display`
            // their implementations.
            CliError::Io(ref err) => write!(f, "IO error: {}
            CliError::Parse(ref err) => write!(f, "Parse err
        }
    }
}

impl error::Error for CliError {
    fn description(&self) -> &str {
        // Both underlying errors already impl `Error`, so w
        // implementations.
        match *self {
            CliError::Io(ref err) => err.description(),
            // Normally we can just write `err.description()
            // type has a concrete method called `descriptio
            // with the trait method. For now, we must expli
            // `description` through the `Error` trait.
            CliError::Parse(ref err) => error::Error::descri
        }
    }

    fn cause(&self) -> Option<&error::Error> {
        match *self {
            // N.B. Both of these implicitly cast `err` from
            // types (either `&io::Error` or `&num::ParseInt
            // to a trait object `&Error`. This works becaus
            // implement `Error`.
            CliError::Io(ref err) => Some(err),
```

```
            CliError::Parse(ref err) => Some(err),
        }
    }
}
```

I note that this is a very typical implementation of `Error`: match on your different error types and satisfy the contracts defined for `description` and `cause`.

The `From` trait

The `std::convert::From` trait is defined in the standard library:

`from-def`

```
trait From<T> {
    fn from(T) -> Self;
}
```

Deliciously simple, yes? `From` is very useful because it gives us a generic way to talk about conversion *from* a particular type T to some other type (in this case, "some other type" is the subject of the impl, or `Self`). The crux of `From` is the set of implementations provided by the standard library.

Here are a few simple examples demonstrating how `From` works:

`from-examples`

```
let string: String = From::from("foo");
let bytes: Vec<u8> = From::from("foo");
let cow: ::std::borrow::Cow<str> = From::from("foo");
```

OK, so `From` is useful for converting between strings. But what about errors? It turns out, there is one critical impl:

```
impl<'a, E: Error + 'a> From<E> for Box<Error + 'a>
```

This impl says that for *any* type that impls `Error`, we can convert it to a trait

object Box<Error>. This may not seem terribly surprising, but it is useful in a generic context.

Remember the two errors we were dealing with previously? Specifically, `io::Error` and `num::ParseIntError`. Since both impl `Error`, they work with `From`:

`from-examples-errors`

```
use std::error::Error;
use std::fs;
use std::io;
use std::num;

// We have to jump through some hoops to actually get error
let io_err: io::Error = io::Error::last_os_error();
let parse_err: num::ParseIntError = "not a number".parse::<i

// OK, here are the conversions.
let err1: Box<Error> = From::from(io_err);
let err2: Box<Error> = From::from(parse_err);
```

There is a really important pattern to recognize here. Both `err1` and `err2` have the *same type*. This is because they are existentially quantified types, or trait objects. In particular, their underlying type is *erased* from the compiler's knowledge, so it truly sees `err1` and `err2` as exactly the same. Additionally, we constructed `err1` and `err2` using precisely the same function call: `From::from`. This is because `From::from` is overloaded on both its argument and its return type.

This pattern is important because it solves a problem we had earlier: it gives us a way to reliably convert errors to the same type using the same function.

Time to revisit an old friend; the `try!` macro/`?` operator.

The real `try!` macro/`?` operator

Previously, I presented this definition of `try!`:

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

This is not it's real definition. It's real definition is in the standard library:

try-def

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(::std::convert::From::from(er
    });
}
```

There's one tiny but powerful change: the error value is passed through `From::from`. This makes the `try!` macro a lot more powerful because it gives you automatic type conversion for free. This is also very similar to how the ? operator works, which is defined slightly differently. Namely, `x?` desugars to something like the following:

questionmark-def

```
match ::std::ops::Try::into_result(x) {
    Ok(v) => v,
    Err(e) => return ::std::ops::Try::from_error(From::from(
}
```

The `Try` trait is still unstable and beyond the scope of this article, but the essence of it is that it provides a way to abstract over many different types of success/failure scenarios, without being tightly coupled to `Result<T, E>`. As you can see though, the `x?` syntax still calls `From::from`, which is how we achieve automatic error conversion.

Since most code written today uses ? instead of `try!`, we will use ? for the

remainder of this post.

Let's take a look at code we wrote previously to read a file and convert its contents to an integer:

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32,
    let mut file = File::open(file_path).map_err(|e| e.to_st
    let mut contents = String::new();
    file.read_to_string(&mut contents).map_err(|e| e.to_stri
    let n = contents.trim().parse::<i32>().map_err(|e| e.to_
    Ok(2 * n)
}
```

Earlier, I promised that we could get rid of the `map_err` calls. Indeed, all we have to do is pick a type that `From` works with. As we saw in the previous section, `From` has an impl that let's it convert any error type into a `Box<Error>`:

`io-basic-error-try-from`

```
use std::error::Error;
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32,
    let mut file = File::open(file_path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let n = contents.trim().parse::<i32>()?;
    Ok(2 * n)
}
```

We are getting very close to ideal error handling. Our code has very little

overhead as a result from error handling because the ? operator encapsulates three things simultaneously:

1. Case analysis.
2. Control flow.
3. Error type conversion.

When all three things are combined, we get code that is unencumbered by combinators, calls to unwrap or case analysis.

There's one little nit left: the Box<Error> type is *opaque*. If we return a Box<Error> to the caller, the caller can't (easily) inspect underlying error type. The situation is certainly better than String because the caller can call methods like description and cause, but the limitation remains: Box<Error> is opaque. (N.B. This isn't entirely true because Rust does have runtime reflection, which is useful in some scenarios that are beyond the scope of this article.)

It's time to revisit our custom CliError type and tie everything together.

Composing custom error types

In the last section, we looked at the real ? operator and how it does automatic type conversion for us by calling From::from on the error value. In particular, we converted errors to Box<Error>, which works, but the type is opaque to callers.

To fix this, we use the same remedy that we're already familiar with: a custom error type. Once again, here is the code that reads the contents of a file and converts it to an integer:

io-basic-error-custom-from

```
use std::fs::File;
use std::io::{self, Read};
use std::num;
use std::path::Path;

// We derive `Debug` because all types should probably deriv
// This gives us a reasonable human readable description of
#[derive(Debug)]
```

```
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}

fn file_double_verbose<P: AsRef<Path>>(file_path: P) -> Resu
    let mut file = File::open(file_path).map_err(CliError::I
    let mut contents = String::new();
    file.read_to_string(&mut contents).map_err(CliError::Io)
    let n: i32 = contents.trim().parse().map_err(CliError::P
    Ok(2 * n)
}
```

Notice that we still have the calls to `map_err`. Why? Well, recall the definitions of the `?` operator and `From`. The problem is that there is no `From` impl that allows us to convert from error types like `io::Error` and `num::ParseIntError` to our own custom `CliError`. Of course, it is easy to fix this! Since we defined `CliError`, we can impl `From` with it:

`io-basic-error-custom-from`

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

impl From<num::ParseIntError> for CliError {
    fn from(err: num::ParseIntError) -> CliError {
        CliError::Parse(err)
    }
}
```

All these impls are doing is teaching `From` how to create a `CliError` from other error types. In our case, construction is as simple as invoking the corresponding value constructor. Indeed, it is *typically* this easy.

We can finally rewrite `file_double`:

io-basic-error-custom-from

```
fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32,
    let mut file = File::open(file_path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let n: i32 = contents.trim().parse()?;
    Ok(2 * n)
}
```

The only thing we did here was remove the calls to `map_err`. They are no longer needed because the ? operator invokes `From::from` on the error value. This works because we've provided `From` impls for all the error types that could appear.

If we modified our `file_double` function to perform some other operation, say, convert a string to a float, then we'd need to add a new variant to our error type:

```
enum CliError {
    Io(io::Error),
    ParseInt(num::ParseIntError),
    ParseFloat(num::ParseFloatError),
}
```

To reflect this change we need to update the previous `impl From<num::ParseIntError> for CliError` and add the new `impl From<num::ParseFloatError> for CliError`:

```
impl From<num::ParseIntError> for CliError {
    fn from(err: num::ParseIntError) -> CliError {
        CliError::ParseInt(err)
    }
}

impl From<num::ParseFloatError> for CliError {
    fn from(err: num::ParseFloatError) -> CliError {
```

```
            CliError::ParseFloat(err)
        }
    }
```

And that's it!

Advice for library writers

Idioms for Rust libraries are still forming, but if your library needs to report custom errors, then you should probably define your own error type. It's up to you whether or not to expose its representation (like `ErrorKind`) or keep it hidden (like `ParseIntError`). Regardless of how you do it, it's usually good practice to at least provide some information about the error beyond just its `String` representation. But certainly, this will vary depending on use cases.

At a minimum, you should probably implement the `Error` trait. This will give users of your library some minimum flexibility for composing errors. Implementing the `Error` trait also means that users are guaranteed the ability to obtain a string representation of an error (because it requires impls for both `fmt::Debug` and `fmt::Display`).

Beyond that, it can also be useful to provide implementations of `From` on your error types. This allows you (the library author) and your users to compose more detailed errors. For example, `csv::Error` provides `From` impls for both `io::Error` and `byteorder::Error`.

Finally, depending on your tastes, you may also want to define a `Result` type alias, particularly if your library defines a single error type. This is used in the standard library for `io::Result` and `fmt::Result`.

Case study: A program to read population data

This article was long, and depending on your background, it might be rather dense. While there is plenty of example code to go along with the prose, most of it was specifically designed to be pedagogical. While I'm not quite smart enough to craft pedagogical examples that are also *not* toy examples,

I certainly can write about a case study.

For this, I'd like to build up a command line program that lets you query world population data. The objective is simple: you give it a location and it will tell you the population. Despite the simplicity, there is a lot that can go wrong!

The data we'll be using comes from the Data Science Toolkit. I've prepared some data from it for this exercise. You can either grab the world population data (41MB gzip compressed, 145MB uncompressed) or just the US population data (2.2MB gzip compressed, 7.2MB uncompressed).

Up until now, I've kept the code limited to Rust's standard library. For a real task like this though, we'll want to at least use something to parse CSV data, parse the program arguments and decode that stuff into Rust types automatically. For that, we'll use the `csv`, `docopt` and `rustc-serialize` crates.

It's on Github

The final code for this case study is on Github. If you have Rust and Cargo installed, then all you need to do is:

```
git clone git://github.com/BurntSushi/rust-error-handling-ca
cd rust-error-handling-case-study
cargo build --release
./target/release/city-pop --help
```

We'll build up this project in pieces. Read on and follow along!

Initial setup

I'm not going to spend a lot of time on setting up a project with Cargo because it is already covered well in the Rust book and Cargo's documentation.

To get started from scratch, run `cargo new --bin city-pop` and make sure your `Cargo.toml` looks something like this:

```
[package]
name = "city-pop"
version = "0.1.0"
authors = ["Andrew Gallant <jamslam@gmail.com>"]

[[bin]]
name = "city-pop"

[dependencies]
csv = "0.*"
docopt = "0.*"
rustc-serialize = "0.*"
```

You should already be able to run:

```
cargo build --release
./target/release/city-pop
#Outputs: Hello, world!
```

Argument parsing

Let's get argument parsing out of the way. I won't go into too much detail on Docopt, but there is a nice web page describing it and documentation for the Rust crate. The short story is that Docopt generates an argument parser *from the usage string*. Once the parsing is done, we can decode the program arguments into a Rust struct. Here's our program with the appropriate `extern crate` statements, the usage string, our `Args` struct and an empty `main`:

```
extern crate docopt;
extern crate rustc_serialize;

static USAGE: &'static str = "
Usage: city-pop [options] <data-path> <city>
       city-pop --help
```

```
Options:
    -h, --help     Show this usage message.
";

struct Args {
    arg_data_path: String,
    arg_city: String,
}

fn main() {

}
```

Okay, time to get coding. The docs for Docopt say we can create a new parser with `Docopt::new` and then decode the current program arguments into a struct with `Docopt::decode`. The catch is that both of these functions can return a `docopt::Error`. We can start with explicit case analysis:

```
// These use statements were added below the `extern` statem
// I'll elide them in the future. Don't worry! It's all on G
// https://github.com/BurntSushi/rust-error-handling-case-st
//use std::io::{self, Write};
//use std::process;
//use docopt::Docopt;

fn main() {
    let args: Args = match Docopt::new(USAGE) {
        Err(err) => {
            writeln!(&mut io::stderr(), "{}", err).unwrap();
            process::exit(1);
        }
        Ok(dopt) => match dopt.decode() {
            Err(err) => {
                writeln!(&mut io::stderr(), "{}", err).unwra
                process::exit(1);
            }
```

```
            Ok(args) => args,
        }
    };
}
```

This is not so nice. One thing we can do to make the code a bit clearer is to write a macro to print messages to `stderr` and then exit:

`fatal-def`

```
macro_rules! fatal {
    ($($tt:tt)*) => {{
        use std::io::Write;
        writeln!(&mut ::std::io::stderr(), $($tt)*).unwrap()
        ::std::process::exit(1)
    }}
}
```

The unwrap is probably OK here, because if it fails, it means your program could not write to `stderr`. A good rule of thumb here is that it's OK to abort, but certainly, you could do something else if you needed to.

The code looks nicer, but the explicit case analysis is still a drag:

```
let args: Args = match Docopt::new(USAGE) {
    Err(err) => fatal!("{}", err),
    Ok(dopt) => match dopt.decode() {
        Err(err) => fatal!("{}", err),
        Ok(args) => args,
    }
};
```

Thankfully, the `docopt::Error` type defines a convenient method `exit`, which effectively does what we just did. Combine that with our knowledge of combinators, and we have concise, easy to read code:

```
let args: Args = Docopt::new(USAGE)
```

```
                            .and_then(|d| d.decode())
                            .unwrap_or_else(|err| err.exit());
```

If this code completes successfully, then `args` will be filled from the values provided by the user.

Writing the logic

We're all different in how we write code, but when I'm not sure how to go about coding a problem, error handling is usually the last thing I want to think about. This isn't very good practice for good design, but it can be useful for rapidly prototyping. In our case, because Rust forces us to be explicit about error handling, it will also make it obvious what parts of our program can cause errors. Why? Because Rust will make us call `unwrap`! This can give us a nice bird's eye view of how we need to approach error handling.

In this case study, the logic is really simple. All we need to do is parse the CSV data given to us and print out a field in matching rows. Let's do it. (Make sure to add `extern crate csv;` to the top of your file.)

```
// This struct represents the data in each row of the CSV fi
// Type based decoding absolves us of a lot of the nitty gri
// handling, like parsing strings as integers or floats.
struct Row {
    country: String,
    city: String,
    accent_city: String,
    region: String,

    // Not every row has data for the population, latitude o
    // So we express them as `Option` types, which admits th
    // absence. The CSV parser will fill in the correct valu
    population: Option<u64>,
    latitude: Option<f64>,
    longitude: Option<f64>,
}
```

```
fn main() {
    let args: Args = Docopt::new(USAGE)
                             .and_then(|d| d.decode())
                             .unwrap_or_else(|err| err.exit())

    let file = fs::File::open(args.arg_data_path).unwrap();
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
        let row = row.unwrap();
        if row.city == args.arg_city {
            println!("{}, {}: {:?}",
                     row.city, row.country,
                     row.population.expect("population count
        }
    }
}
```

Let's outline the errors. We can start with the obvious: the three places that `unwrap` is called:

1. `fs::File::open` can return an `io::Error`.
2. `csv::Reader::decode` decodes one record at a time, and decoding a record (look at the `Item` associated type on the `Iterator` impl) can produce a `csv::Error`.
3. If `row.population` is None, then calling `expect` will panic.

Are there any others? What if we can't find a matching city? Tools like `grep` will return an error code, so we probably should too. So we have logic errors specific to our problem, IO errors and CSV parsing errors. We're going to explore two different ways to approach handling these errors.

I'd like to start with `Box<Error>`. Later, we'll see how defining our own error type can be useful.

Error handling with `Box<Error>`

`Box<Error>` is nice because it *just works*. You don't need to define your own error types and you don't need any `From` implementations. The downside is that since `Box<Error>` is a trait object, it *erases the type*,

which means the compiler can no longer reason about its underlying type.

Previously we started refactoring our code by changing the type of our function from `T` to `Result<T, OurErrorType>`. In this case, `OurErrorType` is just `Box<Error>`. But what's `T`? And can we add a return type to `main`?

The answer to the second question is no, we can't. That means we'll need to write a new function. But what is `T`? The simplest thing we can do is to return a list of matching `Row` values as a `Vec<Row>`. (Better code would return an iterator, but that is left as an exercise to the reader.)

Let's refactor our code into its own function, but keep the calls to `unwrap`. Note that we opt to handle the possibility of a missing population count by simply ignoring that row.

```
struct Row {
    // unchanged
}

struct PopulationCount {
    city: String,
    country: String,
    // This is no longer an `Option` because values of this
    // constructed if they have a population count.
    count: u64,
}

fn search<P: AsRef<Path>>(file_path: P, city: &str) -> Vec<P
    let mut found = vec![];
    let file = fs::File::open(file_path).unwrap();
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
        let row = row.unwrap();
        match row.population {
            None => { } // skip it
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
```

```
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    found
}

fn main() {
    let args: Args = Docopt::new(USAGE)
                             .and_then(|d| d.decode())
                             .unwrap_or_else(|err| err.exit()

    for pop in search(&args.arg_data_path, &args.arg_city) {
        println!("{}, {}: {:?}", pop.city, pop.country, pop.
    }
}
```

While we got rid of one use of `expect` (which is a nicer variant of `unwrap`), we still should handle the absence of any search results.

To convert this to proper error handling, we need to do the following:

1. Change the return type of `search` to be `Result<Vec<PopulationCount>, Box<Error>>`.
2. Use the `?` operator so that errors are returned to the caller instead of panicking the program.
3. Handle the error in `main`.

Let's try it:

```
fn search<P: AsRef<Path>>
          (file_path: P, city: &str)
          -> Result<Vec<PopulationCount>, Box<Error+Send+Sync
    let mut found = vec![];
    let file = fs::File::open(file_path)?;
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
```

```
        let row = row?;
        match row.population {
            None => { } // skip it
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    if found.is_empty() {
        Err(From::from("No matching cities with a population
    } else {
        Ok(found)
    }
}
```

Instead of `x.unwrap()`, we now have `x?`. Since our function returns a `Result<T, E>`, the `?` operator will return early from the function if an error occurs.

There is one big gotcha in this code: we used `Box<Error + Send + Sync>` instead of `Box<Error>`. We did this so we could convert a plain string to an error type. We need these extra bounds so that we can use the corresponding `From` impls:

```
// We are making use of this impl in the code above, since w
// on a `&'static str`.
impl<'a, 'b> From<&'b str> for Box<Error + Send + Sync + 'a>

// But this is also useful when you need to allocate a new s
// error message, usually with `format!`.
impl From<String> for Box<Error + Send + Sync>
```

Now that we've seen how to do proper error handling with `Box<Error>`,

let's try a different approach with our own custom error type. But first, let's take a quick break from error handling and add support for reading from `stdin`.

Reading from stdin

In our program, we accept a single file for input and do one pass over the data. This means we probably should be able to accept input on stdin. But maybe we like the current format too—so let's have both!

Adding support for stdin is actually quite easy. There are only two things we have to do:

1. Tweak the program arguments so that a single parameter—the city—can be accepted while the population data is read from stdin.
2. Modify the `search` function to take an *optional* file path. When None, it should know to read from stdin.

First, here's the new usage and `Args` struct:

```
static USAGE: &'static str = "
Usage: city-pop [options] [<data-path>] <city>
       city-pop --help

Options:
    -h, --help     Show this usage message.
";

struct Args {
    arg_data_path: Option<String>,
    arg_city: String,
}
```

All we did is make the `data-path` argument optional in the Docopt usage string, and make the corresponding struct member `arg_data_path` optional. The `docopt` crate will handle the rest.

Modifying `search` is slightly trickier. The `csv` crate can build a parser out of any type that implements `io::Read`. But how can we use the same code

over both types? There's actually a couple ways we could go about this. One way is to write `search` such that it is generic on some type parameter `R` that satisfies `io::Read`. Another way is to just use trait objects:

```
fn search<P: AsRef<Path>>
          (file_path: &Option<P>, city: &str)
          -> Result<Vec<PopulationCount>, Box<Error+Send+Sync
    let mut found = vec![];
    let input: Box<io::Read> = match *file_path {
        None => Box::new(io::stdin()),
        Some(ref file_path) => Box::new(fs::File::open(file_
    };
    let mut rdr = csv::Reader::from_reader(input);
    // The rest remains unchanged!
}
```

Error handling with a custom type

Previously, we learned how to compose errors using a custom error type. We did this by defining our error type as an `enum` and implementing `Error` and `From`.

Since we have three distinct errors (IO, CSV parsing and not found), let's define an `enum` with three variants:

```
enum CliError {
    Io(io::Error),
    Csv(csv::Error),
    NotFound,
}
```

And now for impls on `Display` and `Error`:

```
impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
```

```
            CliError::Io(ref err) => err.fmt(f),
            CliError::Csv(ref err) => err.fmt(f),
            CliError::NotFound => write!(f, "No matching cit
                                            population were
        }
    }
}

impl Error for CliError {
    fn description(&self) -> &str {
        match *self {
            CliError::Io(ref err) => err.description(),
            CliError::Csv(ref err) => err.description(),
            CliError::NotFound => "not found",
        }
    }
}
```

Before we can use our `CliError` type in our `search` function, we need to provide a couple `From` impls. How do we know which impls to provide? Well, we'll need to convert from both `io::Error` and `csv::Error` to `CliError`. Those are the only external errors, so we'll only need two `From` impls for now:

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

impl From<csv::Error> for CliError {
    fn from(err: csv::Error) -> CliError {
        CliError::Csv(err)
    }
}
```

The `From` impls are important because of how the ? operator is defined. In

particular, if an error occurs, `From::from` is called on the error, which in this case, will convert it to our own error type `CliError`.

With the `From` impls done, we only need to make two small tweaks to our `search` function: the return type and the "not found" error. Here it is in full:

```
fn search<P: AsRef<Path>>
          (file_path: &Option<P>, city: &str)
          -> Result<Vec<PopulationCount>, CliError> {
    let mut found = vec![];
    let input: Box<io::Read> = match *file_path {
        None => Box::new(io::stdin()),
        Some(ref file_path) => Box::new(fs::File::open(file_
    };
    let mut rdr = csv::Reader::from_reader(input);
    for row in rdr.decode::<Row>() {
        let row = row?;
        match row.population {
            None => { } // skip it
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    if found.is_empty() {
        Err(CliError::NotFound)
    } else {
        Ok(found)
    }
}
```

No other changes are necessary.

Adding functionality

If you're anything like me, writing generic code feels good because generalizing stuff is cool! But sometimes, the juice isn't worth the squeeze. Look at what we just did in the previous step:

1. Defined a new error type.
2. Added impls for `Error`, `Display` and two for `From`.

The big downside here is that our program didn't improve a whole lot. I'm personally fond of it because I like using enums for representing errors, but there is quite a bit of overhead to doing so, especially in short programs like this.

One useful aspect of using a custom error type like we've done here is that the `main` function can now choose to handle errors differently. Previously, with `Box<Error>`, it didn't have much of a choice: just print the message. We're still doing that here, but what if we wanted to, say, add a `--quiet` flag? The `--quiet` flag should silence any verbose output.

Right now, if the program doesn't find a match, it will output a message saying so. This can be a little clumsy, especially if you intend for the program to be used in shell scripts.

So let's start by adding the flags. Like before, we need to tweak the usage string and add a flag to the `Args` struct. The `docopt` crate does the rest:

```
static USAGE: &'static str = "
Usage: city-pop [options] [<data-path>] <city>
       city-pop --help

Options:
    -h, --help     Show this usage message.
    -q, --quiet    Don't show noisy messages.
";

struct Args {
    arg_data_path: Option<String>,
    arg_city: String,
    flag_quiet: bool,
```

```
}
```

Now we just need to implement our "quiet" functionality. This requires us to tweak the case analysis in `main`:

```
match search(&args.arg_data_path, &args.arg_city) {
    Err(CliError::NotFound) if args.flag_quiet => process::e
    Err(err) => fatal!("{}", err),
    Ok(pops) => for pop in pops {
        println!("{}, {}: {:?}", pop.city, pop.country, pop.
    }
}
```

Certainly, we don't want to be quiet if there was an IO error or if the data failed to parse. Therefore, we use case analysis to check if the error type is `NotFound` *and* if `--quiet` has been enabled. If the search failed, we still quit with an exit code (following `grep`'s convention).

If we had stuck with `Box<Error>`, then it would be pretty tricky to implement the `--quiet` functionality.

This pretty much sums up our case study. From here, you should be ready to go out into the world and write your own programs and libraries with proper error handling.

The short story

Since this article is long, it is useful to have a quick summary for error handling in Rust. These are my "rules of thumb." They are emphatically *not* commandments. There are probably good reasons to break every one of these heuristics!

- If you're writing short example code that would be overburdened by error handling, it's probably just fine to use `unwrap` (whether that's `Result::unwrap`, `Option::unwrap` or preferably `Option::expect`). Consumers of your code should know to use proper error handling. (If they don't, send them here!)
- If you're writing a quick 'n' dirty program, don't feel ashamed if you use

unwrap. Be warned: if it winds up in someone else's hands, don't be surprised if they are agitated by poor error messages!

- If you're writing a quick 'n' dirty program and feel ashamed about panicking anyway, then you should probably use `Box<Error>` (or `Box<Error + Send + Sync>`) as shown in examples above. Another promising alternative is the `anyhow` crate and its `anyhow::Error` type. When using `anyhow`, your errors will automatically have backtraces attached to them when using nightly Rust.
- Otherwise, in a program, define your own error types with appropriate `From` and `Error` impls to make the `?` operator macro more ergnomic.
- If you're writing a library and your code can produce errors, define your own error type and implement the `std::error::Error` trait. Where appropriate, implement `From` to make both your library code and the caller's code easier to write. (Because of Rust's coherence rules, callers will not be able to impl `From` on your error type, so your library should do it.)
- Learn the combinators defined on `Option` and `Result`. Using them exclusively can be a bit tiring at times, but I've personally found a healthy mix of the `?` operator and combinators to be quite appealing. `and_then`, `map` and `unwrap_or` are my favorites.
