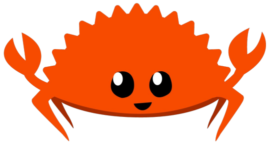
Rust Bootcamp 2021

Day 8



Day 8 Activities

- ✓ More about Errors
- ✓ Macros
- ✓ Unsafe code
- ✓ Using a C library

Combining Errors

We've talked about error handling in Rust on Day 3, and how to propagate errors.

But, what happens when you have errors of different types?

Previously, this was a compiler error with mismatched types. But this can be handled.

Returning an Error trait

The simplest way to handle multiple error types is to use the *Error* trait.

```
use std::error::Error;

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // run some code here, which calls other functions
    // that return different error types
}
```

Because we don't know the specific type, we need to *Box* the trait object.

This allows us to use the ? operator with impunity, and let Rust handle the details.

https://doc.rust-lang.org/book/ch12-03-improving-error-handling-and-modularity.h

Creating our own Error type

The other way is to create an Error type that encapsulates the possible errors. This is common in library crates.

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

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

We first use an enum to combine the io and num error types.

https://doc.rust-lang.org/1.4.0/book/error-handling.html#the-error-trait

```
impl error::Error for MyError {
    fn description(&self) -> &str {
        // Both underlying errors already impl `Error`, so we defer to their
        // implementations.
        match *self {
            MyError::Io(ref err) => err.description(),
            // Normally we can just write `err.description()`, but the error
            // type has a concrete method called `description`, which conflicts
            // with the trait method. For now, we must explicitly call
            // `description` through the `Error` trait.
            MyError::Parse(ref err) => error::Error::description(err),
        }
    }
    fn cause(&self) -> Option<&error::Error> {
        match *self {
            // N.B. Both of these implicitly cast `err` from their concrete
            // types (either `&io::Error` or `&num::ParseIntError`)
            // to a trait object `&Error`. This works because both error types
            // implement `Error`.
            MyError::Io(ref err) => Some(err),
            MyError::Parse(ref err) => Some(err),
        }
    }
}
```

Creating our own Error type

We also need some conversion functionality:

```
impl From<io::Error> for MyError {
    fn from(err: io::Error) -> MyError {
        MyError::Io(err)
    }
}
impl From<num::ParseIntError> for MyError {
    fn from(err: num::ParseIntError) -> MyError {
        MyError::Parse(err)
    }
}
```

https://doc.rust-lang.org/1.4.0/book/error-handling.html#the-error-trait

Creating our own Error type

With all that code out of the way, we can use our Error type.

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

The try! macro does automatic conversion of Error types for us.

https://doc.rust-lang.org/1.4.0/book/error-handling.html#the-error-trait

Macros

Some of the common macros we've already used include *println!* and *try!*.

But it is also possible to create your own macros with some basic metaprogramming.

Basic declarative macros are actually created using another macro, called *macro_rules!*

https://doc.rust-lang.org/book/ch19-06-macros.html

Creating our own Macro

Here is a simplified definition of the *vec!* macro.

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

https://doc.rust-lang.org/book/ch19-06-macros.html

Creating our own Macro

Here is a simplified definition of the *vec!* macro.

Breaking this down, we first declare the name of the macro as *vec*.

Then we have a match expression, which in this case has one pattern, assigning each value in a list to *x*.

The code inside $()* is called for every match, while the surrounding code is called once.

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
    {
        let mut temp_vec = Vec::new();
        $(
            temp_vec.push($x);
        )*
        temp_vec
    }
    };
}
```

https://doc.rust-lang.org/book/ch19-06-macros.html

Macros for common code

```
macro_rules! impl_ops { // create a macro to implement common code for a type
     ($ty:ty) => {
          impl<T: Distance> Add<T> for $ty {
               type Output = Self;
               fn add(self, other: T) -> Self {
                    Self((self.to_base().val() + other.to_base().val())/Self::factor())
               }
          }
          impl<T: Distance> Sub<T> for $ty {
               type Output = Self;
               fn sub(self, other: T) -> Self {
                    Self((self.to_base().val() - other.to_base().val())/Self::factor())
               }
          }
     };
}
impl_ops!(Meters);
impl_ops!(Kilometers);
```

https://doc.rust-lang.org/book/ch19-06-macros.html

Procedural Macros

Another type of macro is the procedural macro, like *derive.*

```
#[derive(debug)]
struct Meters(f32);
```

A procedural macro can generate code from attributes, and operates more like a function than matching against patterns.

Unfortunately, these are very complicated to create, requiring a separate crate just to hold the macro. You can see an example of this here:

https://doc.rust-lang.org/book/ch19-06-macros.html#how-to-write-a-custom-derive-macro

Unsafe Code

There is in fact a way to tell the Rust compiler to ignore all of its safety guarantees, and say, “Trust me, I know what I’m doing.” While it should be a goal to never write unsafe code, sometimes it is needed to interface with the system or hardware. Here are the possible unsafe operations:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of unions

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

Unsafe Code

The *unsafe* operator is used to denote a block of unsafe code.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Note that dereferencing raw pointers is an unsafe operation, even if their creation is safe.

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

Unsafe Functions

We can also designate an entire function as *unsafe.*

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

Unsafe functions must always be called from unsafe blocks.

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

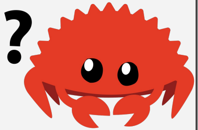
Unsafe Example - Splitting a Slice

Splitting a mutable slice should be safe, since it's returning references to different parts of slice, not the same part.

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut
[i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid], &mut slice[mid..])
}
```



But, the Rust compiler isn't quite smart enough to determine that.

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

Unsafe Example - Splitting a Slice

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut
[i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

External Code

Sometimes you need to call code that isn't Rust, such as things from the C standard library.

```
extern "C" {
    fn abs(input: i32) -> i32;
}

unsafe {
    println!("Absolute value of -3 according to C: {}", abs(-3));
}
```

This is where *extern* blocks come in. They are always unsafe to call, since Rust has no way to guarantee that Rust's rules are being followed.

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#using-extern-functions-to-call-external-code

External Code

Here we'll use the snappy compression library.

```
use libc::size_t;
#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

let x = unsafe { snappy_max_compressed_length(100) };
println!("max compressed length of a 100 byte buffer: {}", x);
```

Note the usage of #[link] to get the compiler to link to a non-standard library.

There are many more notes in the link below, if you want to explore this.

https://doc.rust-lang.org/nomicon/ffi.html

External Code

The other option is creating an interface to Rust code for other languages to call.

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

This code is safe, and creates shared library with the C ABI with this function.

Note that *#[no_mangle]* is needed to keep the function name exactly as written, instead of adding compiler hints to it.

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#calling-rust-functions-from-other-languages

Odds and Ends - Tools

- rustfmt: formats your code according to community style rules

```
$ rustup component add rustfmt

$ cargo fmt
```

- rustfix: fixes your code via compiler suggestions

```
$ cargo fix
```

- clippy: a Rust linter

```
$ rustup component add clippy

$ cargo clippy
```

Odds and Ends - Cleanup action

The *Drop* trait can be used to implement a cleanup action for a struct.

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer: {}", self.data);
    }
}
```

https://doc.rust-lang.org/book/ch15-03-drop.html#running-code-on-cleanup-with-the-drop-trait

Odds and Ends - WASM

Rust can be used to generate code that runs in the browser.

You need a few extra tools to build WASM binaries, which can be read by the browser (a javascript “assembly” code).

Note that while most regular Rust code works with WASM, some crates *do not* work, requiring various features (system library dependencies, file i/o, threads) that do not exist in a browser.

https://rustwasm.github.io/docs/book/

End of Bootcamp!