



Error Handling and Advanced Testing

after all, you need *some* way to deal with buggy code!

Jack Duvall

Carnegie Mellon University



Outline

- 1 Error Handling
- 2 Panics
- 3 Testing
- 4 Property-Based Testing And Fuzzing
- 5 Homework

Outline

1 Error Handling

2 Panics

3 Testing

4 Property-Based Testing And Fuzzing

5 Homework

In General: Ways Of Signaling Errors

- **Error Return Codes:** Function returns a special value to notify caller that it didn't complete successfully.
- **Exceptions:** Abnormal return path, propogating up callstack until a special exception handler catches it.
- **Signals/Panics:** Program immediately interrupted at the request of the OS, usually leads to termination due to severity.

Exceptions Considered Harmful

How can you guarantee that calling function won't throw an exception?

- **SML:** Exceptional control flow built into the type, see `letcc`.
- **C++:** Function can optionally be annotated with `noexcept` keyword
- **Most Other Languages With Exceptions** (Python, Java, etc.): Sorry bro ur out of luck, read the docs ig

How do you release resources if an exception is thrown?

- **Garbage collected languages:** same as usual
- **C++:** Need to treat every non-`noexcept` function like it could throw and add code to clean up local resources if it does

What Do Exceptions *Really* Do, Anyways?

- Return early from a function
- Let the caller know the operation didn't succeed
- Propagate through layers of the stack
- Stop the program if not handled somewhere

This Can Be Done With Types!

Rust's approach: return type encodes both success and failure possibilities

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

We Have Cool Syntax, Too

The `?` operator is very nice.

```
let x = returns_result()?;
```

will de-sugar to

```
let x = match returns_result() {  
    Ok(v) => v,  
    Err(e) => return Err(e),  
}
```

Note this means you can only use `?` inside a function that *also* returns a `Result<V, E2>` where `E2` `impl From<E>`.

Society If We Didn't Have ?

```
fn parse_input1(s: &str)
-> Result<(i32, i32), std::num::ParseIntError> {
    let v = s.split(" ").collect::<Vec<_>>();
    match v[0].parse::<i32>() {
        Ok(a) => match v[1].parse::<i32>() {
            Ok(b) => Ok((a, b)),
            Err(e) => Err(e),
        },
        Err(e) => Err(e),
    }
}
```

Society Because We Have ?

```
fn parse_input2(s: &str)
-> Result<(i32, i32), std::num::ParseIntError> {
    let mut v = s.split(" ").collect::<Vec<_>>();
    let a = v[0].parse::<i32>()?;
    let b = v[1].parse::<i32>()?;
    Ok((a, b))
}
```

Why This Is Better Than Exceptions

It's always explicit if a function can fail or not! Literally in the return type

- `?` operator lets you know all the places a short-circuit return could happen
- Must manually pattern match on the `Result` to check both cases
- More code generated, similar to C++

The type of the function constraints possible failures. If it's not a `Result` type, the function will always succeed when it returns!

- But, are we guaranteed that a function will return?

Outline

1 Error Handling

2 Panics

3 Testing

4 Property-Based Testing And Fuzzing

5 Homework

Main Classes Of Panics In Rust

- Integer Overflow (debug mode only)
- Out-of-bounds slice index
- Any `panic!` statement

Integer Overflow Panics

- Raised whenever an operation would result in value outside bounds of type
 - `u64::MAX + 98`
 - `i32::MIN * -1`
- *Only checked in debug builds*; release builds will use 2's complement wrapping, usually provided by the hardware
- Divide by zero always panics

Explicitly Allowing Integer Overflow

You can manually use wrapping functions directly on the type:

```
assert_eq!(255u8.wrapping_add(5u8), 4u8);
```

Or, use a transparent `Wrapping<T>` struct that has `std::ops::Add` and such implemented for all numeric `T`:

```
assert_eq!(Wrapping(255u8) + Wrapping(5u8), Wrapping(4u8));
```

This wrapper is zero-cost thanks to `#[repr(transparent)]`

What About Floating Point?

What *about* floating point? :)

All floating point errors result in inf or NaN values, which can be checked with `.is_inf()` or `.is_nan()` if necessary.

This is IEEE 754 compliant, fortunately they realized that crashing due to zero division wasn't the best option in all cases :)

Out-Of-Bounds Panics

```
fn main() {  
    let x = [1, 2, 3];  
    println!("{}", x[99]);  
}
```

- If this was written in C, what would this code do?
- Logically, what *should* this code do?
- Fun note: simple “unconditional panics” like this are detected at compile time

The `panic!` Statement

Use this when you purposely want to cause a panic

- Detect extremely unexpected conditions that would nevertheless result in an error
- Enforce invariants when creating structure or calling function

```
struct Bounded<const LOW: usize, const HIGH: usize>(usize);
impl<const LOW: usize, const HIGH: usize> Bounded<LOW, HIGH> {
    fn new(x: usize) -> Self {
        if !(LOW <= x && x <= HIGH) {
            panic!("{x} was not in the range [{LOW}, {HIGH}]!");
        }
        Self(x)
    }
}
```

Friends of the `panic!` Statement

- `assert!`, `assert_eq!`, and `assert_ne!` for condition panics if invariants not met
- `debug_*` versions of the above for asserts that only happen during debug builds
- `todo!` to signal code isn't finished yet
- `unimplemented!` to signal code will not be implemented
- `unreachable!` to signal to the compiler that it can optimize away this branch/check. Use very judiciously!

So many flavors to choose from! yummy

Outline

1 Error Handling

2 Panics

3 Testing

4 Property-Based Testing And Fuzzing

5 Homework

The `#[test]` Annotation

This is a compiler macro, marking a function defined *anywhere in a crate* to be run as part of a test suite during `cargo test`

```
#[test]
fn test1() {
    assert_eq!(9 + 10, 21);
}
```

Tests pass if they run to completion without panicking; conversely, panics signal test failure.

Using `#[test]` With Results

This is a thing you can do!

```
#[test]
fn test2() -> Result<(), String> {
    Err("oh no! my test! it's broken!".to_string())
}
```

What cargo test Looks Like When This Is Run

```
running 2 tests
```

```
test test2 ... FAILED
```

```
test test1 ... FAILED
```

```
failures:
```

```
---- test2 stdout ----
```

```
Error: "oh no! my test! it's broken!"
```

```
thread 'test2' panicked at 'assertion failed: `(left == right)`
```

```
left: `1`,
```

```
right: `0`: the test returned a termination value with a non-zero status c
```

```
note: run with `RUST_BACKTRACE=1` environment variable to display a backtr
```

```
---- test1 stdout ----
```

```
thread 'test1' panicked at 'assertion failed: `(left == right)`
```

```
left: `19`,
```

```
right: `21`, src/lib.rs:3:5
```

Sometimes, You `#[should_panic]`

You can use this annotation to test for error cases where you expect panics:

```
#[test]
#[should_panic]
fn test3() {
    let x: u64 = None.unwrap();
}
```


Recommended Practice: Making A “Test Module”

```
#[cfg(test)]
mod test {
    use super::*;
    #[test]
    fn test1() { ... }
}
```

- Why? Test functions shouldn't be used in other code (because they could panic), so if not compiled with `cargo test`, these test functions will generate “unused function” warnings.
- Adding `#[cfg(test)]` makes the entire module and all functions inside only ever defined in test mode, easier than annotating all of them.

Things You Generally Want To Test

- Simple cases that work as expected (“unit testing”)
- Edge cases handled gracefully
- Serialization/deserialization is invertible
- Internally/externally asserted invariants hold

There's a whole field about Test Driven Development and other best testing practices and I don't really know enough to say much confidently on this subject :P

Upgrading To cargo nexttest

- From the `cargo-nexttest` binary crate, install with `cargo install cargo-nexttest` or put a binary release in your path
- Runs tests in parallel, nicer interface
- Allows checking for flaky tests
 - Flaky test: test that sometimes succeeds, sometimes fails (!)
 - Yes, this does mean Rust doesn't solve all ur software dev woes
- See docs at <https://nexte.st/index.html>

Outline

- 1 Error Handling
- 2 Panics
- 3 Testing
- 4 Property-Based Testing And Fuzzing**
- 5 Homework

What Are Those??

- **Property-Based Testing:** given arbitrary input satisfying certain properties, test that the output will satisfy certain properties as well
 - Requires a way of describing the input properties
 - Frameworks usually have a way of “simplifying” crashing input to a minimal crashing test case
- **Fuzzing:** test that code doesn't crash on all possible traces through code paths
 - Requires some way of enumerating code paths, usually has to be done after compilation that may optimize some branches away
- Very similar concepts! Both deal with somewhat arbitrary inputs, have “panic means test failure”

Crates for Property-Based Testing

- `proptest`
 - Inspired by [Hypothesis](#) (Python)
 - Can make different input ranges/properties per-value
- `quickcheck`
 - Inspired by [QuickCheck](#) (Haskell)
 - Can make different input ranges/properties per-type, often leads to a lot of wrapper types
- `bolero`
 - More fuzzing-like, also has generators similar to `quickcheck`

Note: I have not used any of these :P

proptest Example With Strategy Chaining

```
proptest! {  
    fn grade_range() -> impl Strategy<Value = (u8, u8)>> {  
        (0..=100, 0..=100)  
        .prop_filter("need min<=max", |(min, max)| min<=max)  
        .prop_map(|(min, max)| Range { min, max })  
    }  
    #[test]  
    fn test_create_distribution(range in grade_range()) {  
        let dist: Result<_> = create_distribution(range);  
        prop_assert!(dist.is_ok());  
    }  
}
```

proptest Example With String Regexes

```
proptest! {  
    #[test]  
    fn test_i32_parse_err(s in "[^0-9]+") {  
        let x = s.parse::<i32>();  
        prop_assert!(x.is_err());  
    }  
}
```


Crates For Fuzzing

- `cargo-fuzz`: based on LLVM's LibFuzzer
- `afl`: relies on "American Fuzzy Lop", and old yet popular fuzzing library
- `bo1ero`: supports both those backends, plus Honggfuzz

All of these are x86_64 Linux or x86_64 MacOs only, and need Rust nightly features enabled :(

Outline

- 1 Error Handling
- 2 Panics
- 3 Testing
- 4 Property-Based Testing And Fuzzing
- 5 Homework**

Error Handling Puzzle

There will be different functions returning different errors, handle them all!

This will use things you know about traits, standard library, general Rust syntax

- And it's OK to look these things up too!

Backup: Why Panic When We Have Result?

Some Moral Reasons

- Sometimes, it's just *really* obnoxious
 - Having to check *every single addition* for overflow?
 - Every single allocation?
 - C/C++ people be like: ya ofc (or maybe not)
 - miss me with that tyvm
- Sometimes, the error state is so irrecoverable that we shouldn't bother handling anyways
 - Allocations are *usually* a good example
 - When do you *actually* run out of memory on a modern system?
- Some of Rust's panics are ugly though (on indexing? really?) and libraries sometimes over-use imo
- See the [official Rust Book section](#) for a more balanced view

Panics Are Sometimes Proved Away

The following code will (should, really) not have a panic check:

```
fn main() {  
    let x = vec![1, 2, 3, 4];  
    println!("{}", x[3]);  
}
```

This isn't a feature of Rust, but rather a feature of LLVM, so relying on this can be fickle.

Not Actually A Reason: Runtime Cost

Both panics and `Results` need to be checked for!

- panic: if condition doesn't hold, jump to panic handler (often there are a bunch with different source info and messages and stuff)
- `Result`: branch depending on whether its `Ok` or `Err`.

Sometimes, all these extra panic handlers can result in *more code* than `Results`!
[citation needed]

Backup: The `Try` Trait

So How Does `?` Work, Exactly?

- What does it “desugar” to?
- Can I add more types for it to work with?
- Unfortunately we can't answer either of these questions: currently, it's an internal compiler operation, only for `Option` and `Result` types
- This is different from nearly every other operator! `+` and `>>` and `|` have overloads, even `Deref`!

Motivating Example: A Neat Type

A proposed type that ? could work with:

```
enum ControlFlow<B, C = ()> {  
    /// Exit the operation without running subsequent phases.  
    Break(B),  
    /// Move on to the next phase of the operation as normal.  
    Continue(C),  
}
```

Motivating Example: Some Clean Code

```
impl<T> TreeNode<T> {
    fn traverse_inorder<B>(
        &self,
        mut f: impl FnMut(&T) -> ControlFlow<B>,
    ) -> ControlFlow<B> {
        if let Some(left) = &self.left {
            left.traverse_inorder(&mut f)?;
        }
        f(&self.value)?;
        if let Some(right) = &self.right {
            right.traverse_inorder(&mut f)?;
        }
        ControlFlow::Continue(())
    }
}
```

Terminology

At its core, the `?` operator is about splitting a type and control flow into two parts:

- The **output** that will be returned from the `?`, where control flow continues as normal, and
- The **residual** that will be returned to calling code, as an early exit from the normal flow.

Source for all this: <https://rust-lang.github.io/rfcs/3058-try-trait-v2.html>

Try Is Actually Two Traits

```
trait FromResidual<Residual = <Self as Try>::Residual> {  
    fn from_residual(r: Residual) -> Self;  
}  
  
trait Try: FromResidual {  
    type Output;  
    type Residual;  
    fn branch(self) -> ControlFlow<Self::Residual, Self::Output>;  
    fn from_output(o: Self::Output) -> Self;  
}
```

Why Have Two Traits?

This allows the residual of one erroring type to easily be turned into another output error type, without also having to convert the outputs! Probably a common usecase:

```
impl<T, E: From<String>> FromResidual<ResultCodeResidual> for
    Result<T, E> {
    fn from_residual(r: ResultCodeResidual) -> Self {
        Err(format!(
            "Something fancy about {} at {:?}",
            r.0,
            std::time::SystemTime::now()
        ).into())
    }
}
```

Formalizing Desugaring: Sugared

```
fn<T1, T2> f(g: impl FnOnce() -> T2) -> T1
  where T1: Try,
         T2: FromResidual<T1::Residual>
{
  let x = g();
  let y = x?;
  ...
}
```

Formalizing Desugaring: Desugared

```
fn<T1, T2> f(g: impl FnOnce() -> T2) -> T1
  where T1: Try,
         T2: FromResidual<T1::Residual>
{
  let x = g();
  let y = match T1::branch(x) {
    ControlFlow::Continue(c) => c,
    ControlFlow::Break(b) => { return T2::from_residual(b) }
  };
  ...
}
```