[Unrecoverable Errors with panic! 1](#__RefHeading___Toc9537_1341122361)

[Unwinding the Stack Versus Aborting on Panic 2](#__RefHeading___Toc9539_1341122361)

[Using a panic! Backtrace 3](#__RefHeading___Toc9541_1341122361)

[Recoverable Errors with Result 5](#__RefHeading___Toc9543_1341122361)

[Matching on Different Errors 7](#__RefHeading___Toc9545_1341122361)

[Shortcuts for Panic on Error: unwrap and expect 9](#__RefHeading___Toc9547_1341122361)

[Propagating Errors 10](#__RefHeading___Toc9549_1341122361)

[A Shortcut for Propagating Errors: ? 11](#__RefHeading___Toc9551_1341122361)

[? Can Only Be Used in Functions That Return Result 12](#__RefHeading___Toc9553_1341122361)

[To panic! or Not To panic! 13](#__RefHeading___Toc9555_1341122361)

[Examples, Prototype Code, and Tests: Perfectly Fine to Panic 14](#__RefHeading___Toc9557_1341122361)

[Cases When You Have More Information Than The Compiler 14](#__RefHeading___Toc9559_1341122361)

[Guidelines for Error Handling 15](#__RefHeading___Toc9561_1341122361)

[Creating Custom Types for Validation 16](#__RefHeading___Toc9563_1341122361)

[Summary 18](#__RefHeading___Toc9565_1341122361)

Error Handling

Rust’s commitment to reliability extends to error handling. Errors are a fact of life in software, so Rust has a number of features for handling situations in which something goes wrong. In many cases, Rust will require you to acknowledge the possibility of an error occurring and take some action before your code will compile. This makes your program more robust by ensuring that you won’t only discover errors after you’ve deployed your code to production.

Rust groups errors into two major categories: recoverable and unrecoverable errors. Recoverable errors are situations when it’s usually reasonable to report the problem to the user and retry the operation, like a file not being found. Unrecoverable errors are always symptoms of bugs, like trying to access a location beyond the end of an array.

Most languages don’t distinguish between the two kinds of errors, and handle both in the same way using mechanisms like exceptions. Rust doesn’t have exceptions. Instead, it has the value Result<T, E> for recoverable errors and the panic! macro that stops execution when it encounters unrecoverable errors. This chapter will cover calling panic! first, then talk about returning Result<T, E> values. Finally, we’ll discuss considerations to take into account when deciding whether to try to recover from an error or to stop execution.

Unrecoverable Errors with panic!

Sometimes, bad things happen, and there’s nothing that you can do about it. For these cases, Rust has the panic! macro. When this macro executes, your program will print a failure message, unwind and clean up the stack, and then quit. The most common situation this occurs in is when a bug of some kind has been detected and it’s not clear to the programmer how to handle the error.

PROD: START BOX

Unwinding the Stack Versus Aborting on Panic

By default, when a panic! occurs, the program starts unwinding, which means Rust walks back up the stack and cleans up the data from each function it encounters, but this walking and cleanup is a lot of work. The alternative is to immediately abort, which ends the program without cleaning up. Memory that the program was using will then need to be cleaned up by the operating system. If in your project you need to make the resulting binary as small as possible, you can switch from unwinding to aborting on panic by adding panic = 'abort' to the appropriate [profile] sections in your Cargo.toml. For example, if you want to abort on panic in release mode:

[profile.release]

panic = 'abort'

PROD: END BOX

Let’s try calling panic! with a simple program:

Filename: src/main.rs

fn main() {

panic!("crash and burn");

}

If you run it, you’ll see something like this:

$ cargo run

Compiling panic v0.1.0 (file:///projects/panic)

Finished debug [unoptimized + debuginfo] target(s) in 0.25 secs

Running `target/debug/panic`

thread 'main' panicked at 'crash and burn', src/main.rs:2

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

error: Process didn't exit successfully: `target/debug/panic` (exit code: 101)

The last three lines contain the error message caused by the call to panic!. The first line shows our panic message and the place in our source code where the panic occurred: src/main.rs:2 indicates that it’s the second line of our src/main.rs file.

In this case, the line indicated is part of our code, and if we go to that line we see the panic! macro call. In other cases, the panic! call might be in code that our code calls. The filename and line number reported by the error message will be someone else’s code where the panic! macro is called, not the line of our code that eventually led to the panic!. We can use the backtrace of the functions the panic! call came from to figure this out.

Using a panic! Backtrace

Let’s look at another example to see what it’s like when a panic! call comes from a library because of a bug in our code instead of from our code calling the macro directly:

Filename: src/main.rs

fn main() {

let v = vec![1, 2, 3];

v[100];

}

We’re attempting to access the hundredth element of our vector, but it only has three elements. In this situation, Rust will panic. Using [] is supposed to return an element, but if you pass an invalid index, there’s no element that Rust could return here that would be correct.

Other languages like C will attempt to give you exactly what you asked for in this situation, even though it isn’t what you want: you’ll get whatever is at the location in memory that would correspond to that element in the vector, even though the memory doesn’t belong to the vector. This is called a buffer overread, and can lead to security vulnerabilities if an attacker can manipulate the index in such a way as to read data they shouldn’t be allowed to that is stored after the array.

In order to protect your program from this sort of vulnerability, if you try to read an element at an index that doesn’t exist, Rust will stop execution and refuse to continue. Let’s try it and see:

$ cargo run

Compiling panic v0.1.0 (file:///projects/panic)

Finished debug [unoptimized + debuginfo] target(s) in 0.27 secs

Running `target/debug/panic`

thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 100', /stable-dist-rustc/build/src/libcollections/vec.rs:1362

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

error: Process didn't exit successfully: `target/debug/panic` (exit code: 101)

This points at a file we didn’t write, libcollections/vec.rs. That’s the implementation of Vec<T> in the standard library. The code that gets run when we use [] on our vector v is in libcollections/vec.rs, and that is where the panic! is actually happening.

The next note line tells us that we can set the RUST\_BACKTRACE environment variable to get a backtrace of exactly what happened to cause the error. Let’s try that. Listing 9-1 shows the output:

$ RUST\_BACKTRACE=1 cargo run

Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/panic`

thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 100', /stable-dist-rustc/build/src/libcollections/vec.rs:1395

stack backtrace:

1: 0x10922522c - std::sys::imp::backtrace::tracing::imp::write::h61cce32efcf6a3d0

2: 0x10922649e - std::panicking::default\_hook::{{closure}}::hdac93beb64eaf365

3: 0x109226140 - std::panicking::default\_hook::h4a7f61136a9004ca

4: 0x109226897 - std::panicking::rust\_panic\_with\_hook::hdf5cd951b8d6fa36

5: 0x1092266f4 - std::panicking::begin\_panic::h1204ab053b688140

6: 0x109226662 - std::panicking::begin\_panic\_fmt::h7d4fffc79f986d3b

7: 0x1092265c7 - rust\_begin\_unwind

8: 0x1092486f0 - core::panicking::panic\_fmt::he6eb92dab4407c61

9: 0x109248668 - core::panicking::panic\_bounds\_check::h37b4772a417ae8c7

10: 0x1092205b5 - <collections::vec::Vec<T> as core::ops::Index<usize>>::index::hbc2823add66bc839

11: 0x10922066a - aggregator::main::h977e018a69ea4690

12: 0x1092282ba - \_\_rust\_maybe\_catch\_panic

13: 0x109226b16 - std::rt::lang\_start::h5196b70c908371ed

14: 0x1092206e9 - main

Listing 9-1: The backtrace generated by a call to panic! displayed when the environment variable RUST\_BACKTRACE is set

That’s a lot of output! Line 11 of the backtrace points to the line in our project causing the problem: src/main.rs, line four. A backtrace is a list of all the functions that have been called to get to this point. Backtraces in Rust work like they do in other languages: the key to reading the backtrace is to start from the top and read until you see files you wrote. That’s the spot where the problem originated. The lines above the lines mentioning your files are code that your code called; the lines below are code that called your code. These lines might include core Rust code, standard library code, or crates that you’re using.

If we don’t want our program to panic, the location pointed to by the first line mentioning a file we wrote is where we should start investigating in order to figure out how we got to this location with values that caused the panic. In our example where we deliberately wrote code that would panic in order to demonstrate how to use backtraces, the way to fix the panic is to not try to request an element at index 100 from a vector that only contains three items. When your code panics in the future, you’ll need to figure out for your particular case what action the code is taking with what values that causes the panic and what the code should do instead.

We’ll come back to panic! and when we should and should not use these methods later in the chapter. Next, we’ll now look at how to recover from an error with Result.

Recoverable Errors with Result

Most errors aren’t serious enough to require the program to stop entirely. Sometimes, when a function fails, it’s for a reason that we can easily interpret and respond to. For example, if we try to open a file and that operation fails because the file doesn’t exist, we might want to create the file instead of terminating the process.

Recall from Chapter 2 the section on “Handling Potential Failure with the Result Type” that the Result enum is defined as having two variants, Ok and Err, as follows:

enum Result<T, E> {

Ok(T),

Err(E),

}

The T and E are generic type parameters; we’ll go into generics in more detail in Chapter 10. What you need to know right now is that T represents the type of the value that will be returned in a success case within the Ok variant, and E represents the type of the error that will be returned in a failure case within the Err variant. Because Result has these generic type parameters, we can use the Result type and the functions that the standard library has defined on it in many different situations where the successful value and error value we want to return may differ.

Let’s call a function that returns a Result value because the function could fail: opening a file, shown in Listing 9-2.

Filename: src/main.rs

use std::fs::File;

fn main() {

let f = File::open("hello.txt");

}

Listing 9-2: Opening a file

How do we know File::open returns a Result? We could look at the standard library API documentation, or we could ask the compiler! If we give f a type annotation of some type that we know the return type of the function is not, then we try to compile the code, the compiler will tell us that the types don’t match. The error message will then tell us what the type of f is! Let’s try it: we know that the return type of File::open isn’t of type u32, so let’s change the let f statement to:

let f: u32 = File::open("hello.txt");

Attempting to compile now gives us:

error[E0308]: mismatched types

--> src/main.rs:4:18

|

4 | let f: u32 = File::open("hello.txt");

| ^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum `std::result::Result`

|

= note: expected type `u32`

= note: found type `std::result::Result<std::fs::File, std::io::Error>`

This tells us the return type of the File::open function is a Result<T, E>. The generic parameter T has been filled in here with the type of the success value, std::fs::File, which is a file handle. The type of E used in the error value is std::io::Error.

This return type means the call to File::open might succeed and return to us a file handle that we can read from or write to. The function call also might fail: for example, the file might not exist, or we might not have permission to access the file. The File::open function needs to have a way to tell us whether it succeeded or failed, and at the same time give us either the file handle or error information. This information is exactly what the Result enum conveys.

In the case where File::open succeeds, the value we will have in the variable f will be an instance of Ok that contains a file handle. In the case where it fails, the value in f will be an instance of Err that contains more information about the kind of error that happened.

We need to add to the code from Listing 9-2 to take different actions depending on the value File::open returned. Listing 9-3 shows one way to handle the Result with a basic tool: the match expression that we learned about in Chapter 6.

Filename: src/main.rs

use std::fs::File;

fn main() {

let f = File::open("hello.txt");

let f = match f {

Ok(file) => file,

Err(error) => {

panic!("There was a problem opening the file: {:?}", error)

},

};

}

Listing 9-3: Using a match expression to handle the Result variants we might have

Note that, like the Option enum, the Result enum and its variants have been imported in the prelude, so we don’t need to specify Result:: before the Ok and Err variants in the match arms.

Here we tell Rust that when the result is Ok, return the inner file value out of the Ok variant, and we then assign that file handle value to the variable f. After the match, we can then use the file handle for reading or writing.

The other arm of the match handles the case where we get an Err value from File::open. In this example, we’ve chosen to call the panic! macro. If there’s no file named hello.txt in our current directory and we run this code, we’ll see the following output from the panic! macro:

thread 'main' panicked at 'There was a problem opening the file: Error { repr: Os { code: 2, message: "No such file or directory" } }', src/main.rs:8

Matching on Different Errors

The code in Listing 9-3 will panic! no matter the reason that File::open failed. What we’d really like to do instead is take different actions for different failure reasons: if File::open failed because the file doesn’t exist, we want to create the file and return the handle to the new file. If File::open failed for any other reason, for example because we didn’t have permission to open the file, we still want to panic! in the same way as we did in Listing 9-3. Let’s look at Listing 9-4, which adds another arm to the match:

Filename: src/main.rs

use std::fs::File;

use std::io::ErrorKind;

fn main() {

let f = File::open("hello.txt");

let f = match f {

Ok(file) => file,

Err(ref error) if error.kind() == ErrorKind::NotFound => {

match File::create("hello.txt") {

Ok(fc) => fc,

Err(e) => {

panic!(

"Tried to create file but there was a problem: {:?}",

e

)

},

}

},

Err(error) => {

panic!(

"There was a problem opening the file: {:?}",

error

)

},

};

}

Listing 9-4: Handling different kinds of errors in different ways

The type of the value that File::open returns inside the Err variant is io::Error, which is a struct provided by the standard library. This struct has a method kind that we can call to get an io::ErrorKind value. io::ErrorKind is an enum provided by the standard library that has variants representing the different kinds of errors that might result from an io operation. The variant we’re interested in is ErrorKind::NotFound, which indicates the file we’re trying to open doesn’t exist yet.

The condition if error.kind() == ErrorKind::NotFound is called a match guard: it’s an extra condition on a match arm that further refines the arm’s pattern. This condition must be true in order for that arm’s code to get run; otherwise, the pattern matching will move on to consider the next arm in the match. The ref in the pattern is needed so that error is not moved into the guard condition but is merely referenced by it. The reason ref is used to take a reference in a pattern instead of & will be covered in detail in Chapter 18. In short, in the context of a pattern, & matches a reference and gives us its value, but ref matches a value and gives us a reference to it.

The condition we want to check in the match guard is whether the value returned by error.kind() is the NotFound variant of the ErrorKind enum. If it is, we try to create the file with File::create. However, since File::create could also fail, we need to add an inner match statement as well! When the file can’t be opened, a different error message will be printed. The last arm of the outer match stays the same so that the program panics on any error besides the missing file error.

Shortcuts for Panic on Error: unwrap and expect

Using match works well enough, but it can be a bit verbose and doesn’t always communicate intent well. The Result<T, E> type has many helper methods defined on it to do various things. One of those methods, called unwrap, is a shortcut method that is implemented just like the match statement we wrote in Listing 9-3. If the Result value is the Ok variant, unwrap will return the value inside the Ok. If the Result is the Err variant, unwrap will call the panic! macro for us.

use std::fs::File;

fn main() {

let f = File::open("hello.txt").unwrap();

}

If we run this code without a hello.txt file, we’ll see an error message from the panic! call that the unwrap method makes:

thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error { repr: Os { code: 2, message: "No such file or directory" } }', /stable-dist-rustc/build/src/libcore/result.rs:868

There’s another method similar to unwrap that lets us also choose the panic! error message: expect. Using expect instead of unwrap and providing good error messages can convey your intent and make tracking down the source of a panic easier. The syntax of expect looks like this:

use std::fs::File;

fn main() {

let f = File::open("hello.txt").expect("Failed to open hello.txt");

}

We use expect in the same way as unwrap: to return the file handle or call the panic! macro. The error message that expect uses in its call to panic! will be the parameter that we pass to expect instead of the default panic! message that unwrap uses. Here’s what it looks like:

thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code: 2, message: "No such file or directory" } }', /stable-dist-rustc/build/src/libcore/result.rs:868

Propagating Errors

When writing a function whose implementation calls something that might fail, instead of handling the error within this function, you can choose to let your caller know about the error so they can decide what to do. This is known as propagating the error, and gives more control to the calling code where there might be more information or logic that dictates how the error should be handled than what you have available in the context of your code.

For example, Listing 9-5 shows a function that reads a username from a file. If the file doesn’t exist or can’t be read, this function will return those errors to the code that called this function:

use std::io;

use std::io::Read;

use std::fs::File;

fn read\_username\_from\_file() -> Result<String, io::Error> {

let f = File::open("hello.txt");

let mut f = match f {

Ok(file) => file,

Err(e) => return Err(e),

};

let mut s = String::new();

match f.read\_to\_string(&mut s) {

Ok(\_) => Ok(s),

Err(e) => Err(e),

}

}

Listing 9-5: A function that returns errors to the calling code using match

Let’s look at the return type of the function first: Result<String, io::Error>. This means that the function is returning a value of the type Result<T, E> where the generic parameter T has been filled in with the concrete type String, and the generic type E has been filled in with the concrete type io::Error. If this function succeeds without any problems, the caller of this function will receive an Ok value that holds a String — the username that this function read from the file. If this function encounters any problems, the caller of this function will receive an Err value that holds an instance of io::Error that contains more information about what the problems were. We chose io::Error as the return type of this function because that happens to be the type of the error value returned from both of the operations we’re calling in this function’s body that might fail: the File::open function and the read\_to\_string method.

The body of the function starts by calling the File::open function. Then we handle the Result value returned with a match similar to the match in Listing 9-3, only instead of calling panic! in the Err case, we return early from this function and pass the error value from File::open back to the caller as this function’s error value. If File::open succeeds, we store the file handle in the variable f and continue.

Then we create a new String in variable s and call the read\_to\_string method on the file handle in f in order to read the contents of the file into s. The read\_to\_string method also returns a Result because it might fail, even though File::open succeeded. So we need another match to handle that Result: if read\_to\_string succeeds, then our function has succeeded, and we return the username from the file that’s now in s wrapped in an Ok. If read\_to\_string fails, we return the error value in the same way that we returned the error value in the match that handled the return value of File::open. We don’t need to explicitly say return, however, since this is the last expression in the function.

The code that calls this code will then handle getting either an Ok value that contains a username or an Err value that contains an io::Error. We don’t know what the caller will do with those values. If they get an Err value, they could choose to call panic! and crash their program, use a default username, or look up the username from somewhere other than a file, for example. We don’t have enough information on what the caller is actually trying to do, so we propagate all the success or error information upwards for them to handle as they see fit.

This pattern of propagating errors is so common in Rust that there is dedicated syntax to make this easier: ?.

A Shortcut for Propagating Errors: ?

Listing 9-6 shows an implementation of read\_username\_from\_file that has the same functionality as it had in Listing 9-5, but this implementation uses the question mark operator:

use std::io;

use std::fs::File;

fn read\_username\_from\_file() -> Result<String, io::Error> {

let mut f = File::open("hello.txt")?;

let mut s = String::new();

f.read\_to\_string(&mut s)?;

Ok(s)

}

Listing 9-6: A function that returns errors to the calling code using ?

The ? placed after a Result value is defined to work the exact same way as the match expressions we defined to handle the Result values in Listing 9-5. If the value of the Result is an Ok, the value inside the Ok will get returned from this expression and the program will continue. If the value is an Err, the value inside the Err will be returned from the whole function as if we had used the return keyword so that the error value gets propagated to the caller.

In the context of Listing 9-6, the ? at the end of the File::open call will return the value inside an Ok to the variable f. If an error occurs, ? will return early out of the whole function and give any Err value to our caller. The same thing applies to the ? at the end of the read\_to\_string call.

The ? eliminates a lot of boilerplate and makes this function’s implementation simpler. We could even shorten this code further by chaining method calls immediately after the ?:

use std::io;

use std::io::Read;

use std::fs::File;

fn read\_username\_from\_file() -> Result<String, io::Error> {

let mut s = String::new();

File::open("hello.txt")?.read\_to\_string(&mut s)?;

Ok(s)

}

We’ve moved the creation of the new String in s to the beginning of the function; that part hasn’t changed. Instead of creating a variable f, we’ve chained the call to read\_to\_string directly onto the result of File::open("hello.txt")?. We still have a ? at the end of the read\_to\_string call, and we still return an Ok value containing the username in s when both File::open and read\_to\_string succeed rather than returning errors. The functionality is again the same as in Listing 9-5 and Listing 9-6, this is just a different, more ergonomic way to write it.

? Can Only Be Used in Functions That Return Result

The ? can only be used in functions that have a return type of Result, since it is defined to work in exactly the same way as the match expression we defined in Listing 9-5. The part of the match that requires a return type of Result is return Err(e), so the return type of the function must be a Result to be compatible with this return.

Let’s look at what happens if use ? in the main function, which you’ll recall has a return type of ():

use std::fs::File;

fn main() {

let f = File::open("hello.txt")?;

}

When we compile this, we get the following error message:

error[E0308]: mismatched types

-->

|

3 | let f = File::open("hello.txt")?;

| ^^^^^^^^^^^^^^^^^^^^^^^^^ expected (), found enum `std::result::Result`

|

= note: expected type `()`

= note: found type `std::result::Result<\_, \_>`

This error is pointing out that we have mismatched types: the main function has a return type of (), but the ? might return a Result. In functions that don’t return Result, when you call other functions that return Result, you’ll need to use a match or one of the Result methods to handle it, instead of using ? to potentially propagate the error to the caller.

Now that we’ve discussed the details of calling panic! or returning Result, let’s return to the topic of how to decide which is appropriate to use in which cases.

To panic! or Not To panic!

So how do you decide when you should panic! and when you should return Result? When code panics, there’s no way to recover. You could choose to call panic! for any error situation, whether there’s a possible way to recover or not, but then you’re making the decision for your callers that a situation is unrecoverable. When you choose to return a Result value, you give your caller options, rather than making the decision for them. They could choose to attempt to recover in a way that’s appropriate for their situation, or they could decide that actually, an Err value in this case is unrecoverable, so they can call panic! and turn your recoverable error into an unrecoverable one. Therefore, returning Result is a good default choice when you’re defining a function that might fail.

There are a few situations in which it’s more appropriate to write code that panics instead of returning a Result, but they are less common. Let’s discuss why it’s appropriate to panic in examples, prototype code, and tests, then situations where you as a human can know a method won’t fail that the compiler can’t reason about, and conclude with some general guidelines on how to decide whether to panic in library code.

Examples, Prototype Code, and Tests: Perfectly Fine to Panic

When you’re writing an example to illustrate some concept, having robust error handling code in the example as well can make the example less clear. In examples, it’s understood that a call to a method like unwrap that could panic! is meant as a placeholder for the way that you’d actually like your application to handle errors, which can differ based on what the rest of your code is doing.

Similarly, the unwrap and expect methods are very handy when prototyping, before you’re ready to decide how to handle errors. They leave clear markers in your code for when you’re ready to make your program more robust.

If a method call fails in a test, we’d want the whole test to fail, even if that method isn’t the functionality under test. Because panic! is how a test gets marked as a failure, calling unwrap or expect is exactly what makes sense to do.

Cases When You Have More Information Than The Compiler

It would also be appropriate to call unwrap when you have some other logic that ensures the Result will have an Ok value, but the logic isn’t something the compiler understands. You’ll still have a Result value that you need to handle: whatever operation you’re calling still has the possibility of failing in general, even though it’s logically impossible in your particular situation. If you can ensure by manually inspecting the code that you’ll never have an Err variant, it is perfectly acceptable to call unwrap. Here’s an example:

use std::net::IpAddr;

let home = "127.0.0.1".parse::<IpAddr>().unwrap();

We’re creating an IpAddr instance by parsing a hardcoded string. We can see that 127.0.0.1 is a valid IP address, so it’s acceptable to use unwrap here. However, having a hardcoded, valid string doesn’t change the return type of the parse method: we still get a Result value, and the compiler will still make us handle the Result as if the Err variant is still a possibility since the compiler isn’t smart enough to see that this string is always a valid IP address. If the IP address string came from a user instead of being hardcoded into the program, and therefore did have a possibility of failure, we’d definitely want to handle the Result in a more robust way instead.

Guidelines for Error Handling

It’s advisable to have your code panic! when it’s possible that you could end up in a bad state—in this context, bad state is when some assumption, guarantee, contract, or invariant has been broken, such as when invalid values, contradictory values, or missing values are passed to your code—plus one or more of the following:

The bad state is not something that’s expected to happen occasionally

Your code after this point needs to rely on not being in this bad state

There’s not a good way to encode this information in the types you use

If someone calls your code and passes in values that don’t make sense, the best thing might be to panic! and alert the person using your library to the bug in their code so that they can fix it during development. Similarly, panic! is often appropriate if you’re calling external code that is out of your control, and it returns an invalid state that you have no way of fixing.

When a bad state is reached, but it’s expected to happen no matter how well you write your code, it’s still more appropriate to return a Result rather than calling panic!. Examples of this include a parser being given malformed data, or an HTTP request returning a status that indicates you have hit a rate limit. In these cases, you should indicate that failure is an expected possibility by returning a Result in order to propagate these bad states upwards so that the caller can decide how they would like to handle the problem. To panic! wouldn’t be the best way to handle these cases.

When your code performs operations on values, your code should verify the values are valid first, and panic! if the values aren’t valid. This is mostly for safety reasons: attempting to operate on invalid data can expose your code to vulnerabilities. This is the main reason that the standard library will panic! if you attempt an out-of-bounds array access: trying to access memory that doesn’t belong to the current data structure is a common security problem. Functions often have contracts: their behavior is only guaranteed if the inputs meet particular requirements. Panicking when the contract is violated makes sense because a contract violation always indicates a caller-side bug, and it is not a kind of error you want callers to have to explicitly handle. In fact, there’s no reasonable way for calling code to recover: the calling programmers need to fix the code. Contracts for a function, especially when a violation will cause a panic, should be explained in the API documentation for the function.

Having lots of error checks in all of your functions would be verbose and annoying, though. Luckily, you can use Rust’s type system (and thus the type checking the compiler does) to do a lot of the checks for you. If your function has a particular type as a parameter, you can proceed with your code’s logic knowing that the compiler has already ensured you have a valid value. For example, if you have a type rather than an Option, your program expects to have something rather than nothing. Your code then doesn’t have to handle two cases for the Some and None variants, it will only have one case for definitely having a value. Code trying to pass nothing to your function won’t even compile, so your function doesn’t have to check for that case at runtime. Another example is using an unsigned integer type like u32, which ensures the parameter is never negative.

Creating Custom Types for Validation

Let’s take the idea of using Rust’s type system to ensure we have a valid value one step further, and look at creating a custom type for validation. Recall the guessing game in Chapter 2, where our code asked the user to guess a number between 1 and 100. We actually never validated that the user’s guess was between those numbers before checking it against our secret number, only that it was positive. In this case, the consequences were not very dire: our output of “Too high” or “Too low” would still be correct. It would be a useful enhancement to guide the user towards valid guesses, though, and have different behavior when a user guesses a number that’s out of range versus when a user types, for example, letters instead.

One way to do this would be to parse the guess as an i32 instead of only a u32, to allow potentially negative numbers, then add a check for the number being in range:

loop {

// snip

let guess: i32 = match guess.trim().parse() {

Ok(num) => num,

Err(\_) => continue,

};

if guess < 1 || guess > 100 {

println!("The secret number will be between 1 and 100.");

continue;

}

match guess.cmp(&secret\_number) {

// snip

}

The if expression checks to see if our value is out of range, tells the user about the problem, and calls continue to start the next iteration of the loop and ask for another guess. After the if expression, we can proceed with the comparisons between guess and the secret number knowing that guess is between 1 and 100.

However, this is not an ideal solution: if it was absolutely critical that the program only operated on values between 1 and 100, and it had many functions with this requirement, it would be tedious (and potentially impact performance) to have a check like this in every function.

Instead, we can make a new type and put the validations in a function to create an instance of the type rather than repeating the validations everywhere. That way, it’s safe for functions to use the new type in their signatures and confidently use the values they receive. Listing 9-8 shows one way to define a Guess type that will only create an instance of Guess if the new function receives a value between 1 and 100:

struct Guess {

value: u32,

}

impl Guess {

pub fn new(value: u32) -> Guess {

if value < 1 || value > 100 {

panic!("Guess value must be between 1 and 100, got {}.", value);

}

Guess {

value: value,

}

}

pub fn value(&self) -> u32 {

self.value

}

}

Listing 9-8: A Guess type that will only continue with values between 1 and 100

First, we define a struct named Guess that has a field named value that holds a u32. This is where the number will be stored.

Then we implement an associated function named new on Guess that creates instances of Guess values. The new function is defined to have one parameter named value of type u32 and to return a Guess. The code in the body of the new function tests value to make sure it is between 1 and 100. If value doesn’t pass this test, we call panic!, which will alert the programmer who is calling this code that they have a bug they need to fix, since creating a Guess with a value outside this range would violate the contract that Guess::new is relying on. The conditions in which Guess::new might panic should be discussed in its public-facing API documentation; we’ll cover documentation conventions around indicating the possibility of a panic! in the API documentation that you create in Chapter 14. If value does pass the test, we create a new Guess with its value field set to the value parameter and return the Guess.

Next, we implement a method named value that borrows self, doesn’t have any other parameters, and returns a u32. This is a kind of method sometimes called a getter, since its purpose is to get some data from its fields and return it. This public method is necessary because the value field of the Guess struct is private. It’s important that the value field is private so that code using the Guess struct is not allowed to set value directly: callers must use the Guess::new function to create an instance of Guess, which ensures there’s no way for a Guess to have a value that hasn’t been checked by the conditions in the Guess::new function.

A function that has a parameter or returns only numbers between 1 and 100 could then declare in its signature that it takes or returns a Guess rather than a u32, and wouldn’t need to do any additional checks in its body.

Summary

Rust’s error handling features are designed to help you write more robust code. The panic! macro signals that your program is in a state it can’t handle, and lets you tell the process to stop instead of trying to proceed with invalid or incorrect values. The Result enum uses Rust’s type system to indicate that operations might fail in a way that your code could recover from. You can use Result to tell code that calls your code that it needs to handle potential success or failure as well. Using panic! and Result in the appropriate situations will make your code more reliable in the face of inevitable problems.

Now that we’ve seen useful ways that the standard library uses generics with the Option and Result enums, let’s talk about how generics work and how you can make use of them in your code.