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 requires you to acknowledge the possibility of an error and take some action before your code will compile. This requirement makes your program more robust by ensuring that you'll discover errors and handle them appropriately before you've deployed your code to production!

Rust groups errors into two major categories: *recoverable* and *unrecoverable* errors. For a recoverable error, such as a *file not found* error, we most likely just want to report the problem to the user and retry the operation. Unrecoverable errors are always symptoms of bugs, such as trying to access a location beyond the end of an array, and so we want to immediately stop the program.

Most languages don't distinguish between these two kinds of errors and handle both in the same way, using mechanisms such as exceptions. Rust doesn't have exceptions. Instead, it has the type `Result<T, E>` for recoverable errors and the `panic!` macro that stops execution when the program encounters an unrecoverable error. This chapter covers calling `panic!` first and then talks about returning `Result<T, E>` values. Additionally, we'll explore considerations when deciding whether to try to recover from an error or to stop execution.

Unrecoverable Errors with `panic!`

Sometimes bad things happen in your code, and there's nothing you can do about it. In these cases, Rust has the `panic!` macro. There are two ways to cause a panic in practice: by taking an action that causes our code to panic (such as accessing an array past the end) or by explicitly calling the `panic!` macro. In both cases, we cause a panic in our program. By default, these panics will print a failure message, unwind, clean up the stack, and quit. Via an environment variable, you can also have Rust display the call stack when a panic occurs to make it easier to track down the source of the panic.

Unwinding the Stack or Aborting in Response to a 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. However, walking back and cleaning up is a lot of work. Rust, therefore, allows you to choose the alternative of immediately *aborting*, 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 resultant binary as small as possible, you can switch from unwinding to aborting upon a panic by adding `panic = 'abort'` to the appropriate `[profile]` sections in your *Cargo.toml* file. For example, if you want to abort on panic in release mode, add this:

```
[profile.release]
panic = 'abort'
```

Let's try calling `panic!` in a simple program:

Filename: src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```

When you run the program, you'll see something like this:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at src/main.rs:2:5:
crash and burn
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

The call to `panic!` causes the error message contained in the last two lines. The first line shows our panic message and the place in our source code where the panic occurred: *src/main.rs:2:5* indicates that it's the second line, fifth character 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, and 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!` call.

We can use the backtrace of the functions the `panic!` call came from to figure out the part of our code that is causing the problem. To understand how to use a `panic!` backtrace, let's look at another example and 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. Listing 9-1 has some code that attempts to access an index in a vector beyond the range of valid indexes.

Filename: src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

Listing 9-1: Attempting to access an element beyond the end of a vector, which will cause a call to `panic!`

Here, we're attempting to access the 100th element of our vector (which is at index 99 because indexing starts at zero), but the vector has only 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.

In C, attempting to read beyond the end of a data structure is undefined behavior. You might get whatever is at the location in memory that would correspond to that element in the data structure, even though the memory doesn't belong to that structure. This is called a *buffer overread* and can lead to security vulnerabilities if an attacker is able to manipulate the index in such a way as to read data they shouldn't be allowed to that is stored after the data structure.

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 `dev` profile [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at src/main.rs:4:6:
index out of bounds: the len is 3 but the index is 99
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

This error points at line 4 of our *main.rs* where we attempt to access index `99` of the vector in `v`.

The `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. A *backtrace* is a list of all the functions that have been called to get to this point. Backtraces in Rust work as 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 that spot are code that your code has called; the lines below are code that called your code. These before-and-after lines might include core Rust code, standard library code, or crates that you're using. Let's try getting a backtrace by setting the `RUST_BACKTRACE` environment variable to any value except `0`. Listing 9-2 shows output similar to what you'll see.

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at src/main.rs:4:6:
index out of bounds: the len is 3 but the index is 99
stack backtrace:
   0: rust_begin_unwind
             at
/rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/std/src/panicking.rs:645:5
   1: core::panicking::panic_fmt
             at
/rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src/panicking.rs:72:1
4
   2: core::panicking::panic_bounds_check
             at
/rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src/panicking.rs:208:
5
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at
/rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src/slice/index.rs:25
5:10
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at
/rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src/slice/index.rs:18
:9
   5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at
/rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/alloc/src/vec/mod.rs:2770:
9
   6: panic::main
             at ./src/main.rs:4:6
   7: core::ops::function::FnOnce::call_once
             at
/rustc/07dca489ac2d933c78d3c5158e3f43beefeb02ce/library/core/src/ops/function.rs:2
50:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

Listing 9-2: The backtrace generated by a call to `panic!` displayed when the environment variable `RUST_BACKTRACE` is set

That's a lot of output! The exact output you see might be different depending on your operating system and Rust version. In order to get backtraces with this information, debug symbols must be enabled. Debug symbols are enabled by default when using `cargo build` or `cargo run` without the `--release` flag, as we have here.

In the output in Listing 9-2, line 6 of the backtrace points to the line in our project that's causing the problem: line 4 of *src/main.rs*. If we don't want our program to panic, we should start our investigation at the location pointed to by the first line mentioning a file we wrote. In Listing 9-1, where we deliberately wrote code that would panic, the way to fix the panic is to not request an element beyond the range of the vector indexes. When your code panics in the future, you'll

need to figure out what action the code is taking with what values to cause the panic and what the code should do instead.

We'll come back to `panic!` and when we should and should not use `panic!` to handle error conditions in the "To `panic!` or Not to `panic!`" section later in this chapter. Next, we'll look at how to recover from an error using `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 you can easily interpret and respond to. For example, if you try to open a file and that operation fails because the file doesn't exist, you might want to create the file instead of terminating the process.

Recall from "Handling Potential Failure with `Result`" in Chapter 2 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 discuss 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 defined on it in many different situations where the success 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. In Listing 9-3 we try to open a file.

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");
}
```

Listing 9-3: Opening a file

The return type of `File::open` is a `Result<T, E>`. The generic parameter `T` has been filled in by the implementation of `File::open` 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 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 in the variable `greeting_file_result` will be an instance of `Ok` that contains a file handle. In the case where it fails, the value in `greeting_file_result` will be an instance of `Err` that contains more information about the kind of error that occurred.

We need to add to the code in Listing 9-3 to take different actions depending on the value `File::open` returns. Listing 9-4 shows one way to handle the `Result` using a basic tool, the `match` expression that we discussed in Chapter 6.

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {error:?}"),
    };
}
```

Listing 9-4: Using a `match` expression to handle the `Result` variants that might be returned

Note that, like the `Option` enum, the `Result` enum and its variants have been brought into scope by the prelude, so we don't need to specify `Result::` before the `Ok` and `Err` variants in the `match` arms.

When the result is `Ok`, this code will return the inner `file` value out of the `Ok` variant, and we then assign that file handle value to the variable `greeting_file`. After the `match`, we can 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:

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/error-handling`
thread 'main' panicked at src/main.rs:8:23:
Problem opening the file: Os { code: 2, kind: NotFound, message: "No such file or
directory" }
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As usual, this output tells us exactly what has gone wrong.

Matching on Different Errors

The code in Listing 9-4 will `panic!` no matter why `File::open` failed. However, we want to 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 the code to `panic!` in the same way it did in Listing 9-4. For this, we add an inner `match` expression, shown in Listing 9-5.

Filename: src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {e:?}"),
            },
            other_error => {
                panic!("Problem opening the file: {other_error:?}");
            }
        },
    };
}
```

Listing 9-5: 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. The enum `io::ErrorKind` is provided by the standard library and has variants representing the different kinds of errors that might result from an `io` operation. The variant we want to use is `ErrorKind::NotFound`, which indicates the file we're trying to open doesn't exist yet. So we match on `greeting_file_result`, but we also have an inner match on `error.kind()`.

The condition we want to check in the inner match 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, because `File::create` could also fail, we need a second arm in the inner `match` expression. When the file can't be created, a different error message is printed. The second arm of the outer `match` stays the same, so the program panics on any error besides the missing file error.

Alternatives to Using `match` with `Result<T, E>`

That's a lot of `match`! The `match` expression is very useful but also very much a primitive. In Chapter 13, you'll learn about closures, which are used with many of the methods defined on `Result<T, E>`. These methods can be more concise than using `match` when handling `Result<T, E>` values in your code.

For example, here's another way to write the same logic as shown in Listing 9-5, this time using closures and the `unwrap_or_else` method:

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {error:?}");
            })
        } else {
            panic!("Problem opening the file: {error:?}");
        }
    });
}
```

Although this code has the same behavior as Listing 9-5, it doesn't contain any `match` expressions and is cleaner to read. Come back to this example after you've read Chapter 13, and look up the `unwrap_or_else` method in the standard library documentation. Many more of these methods can clean up huge nested `match` expressions when you're dealing with errors.

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, more specific tasks. The `unwrap` method is a shortcut method implemented just like the `match` expression we wrote in Listing 9-4. 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. Here is an example of `unwrap` in action:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = 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 src/main.rs:4:49:
called `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound,
message: "No such file or directory" }
```

Similarly, the `expect` method lets us also choose the `panic!` error message. 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:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}
```

We use `expect` in the same way as `unwrap`: to return the file handle or call the `panic!` macro. The error message used by `expect` in its call to `panic!` will be the parameter that we pass to `expect`, rather than the default `panic!` message that `unwrap` uses. Here's what it looks like:

```
thread 'main' panicked at src/main.rs:5:10:
hello.txt should be included in this project: Os { code: 2, kind: NotFound,
message: "No such file or directory" }
```

In production-quality code, most Rustaceans choose `expect` rather than `unwrap` and give more context about why the operation is expected to always succeed. That way, if your assumptions are ever proven wrong, you have more information to use in debugging.

Propagating Errors

When a function's implementation calls something that might fail, instead of handling the error within the function itself you can return the error to the calling code so that it 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 9-6 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 the function.

Filename: src/main.rs

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

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

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

This function can be written in a much shorter way, but we're going to start by doing a lot of it manually in order to explore error handling; at the end, we'll show the shorter way. Let's look at the return type of the function first: `Result<String, io::Error>`. This means 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 code that calls 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 calling code 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 with a `match` similar to the `match` in Listing 9-4. If `File::open` succeeds, the file handle in the pattern variable `file` becomes the value in the mutable variable `username_file` and the function continues. In the `Err` case, instead of calling `panic!`, we use the `return` keyword to return early out of the function entirely and pass the error value from `File::open`, now in the pattern variable `e`, back to the calling code as this function's error value.

So, if we have a file handle in `username_file`, the function then creates a new `String` in variable `username` and calls the `read_to_string` method on the file handle in `username_file` to read the contents of the file into `username`. 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 `username` 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`. However, we don't need to explicitly say `return`, because 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`. It's up to the calling code to decide what to do with those values. If the calling code gets an `Err` value, it could call `panic!` and crash the 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 calling code is actually trying to do, so we propagate all the success or error information upward for it to handle appropriately.

This pattern of propagating errors is so common in Rust that Rust provides the question mark operator `?` to make this easier.

A Shortcut for Propagating Errors: the ? Operator

Listing 9-7 shows an implementation of `read_username_from_file` that has the same functionality as in Listing 9-6, but this implementation uses the `?` operator.

Filename: src/main.rs

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

Listing 9-7: A function that returns errors to the calling code using the `?` operator

The `?` placed after a `Result` value is defined to work in almost the same way as the `match` expressions we defined to handle the `Result` values in Listing 9-6. 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 `Err` will be returned from the whole function as if we had used the `return` keyword so the error value gets propagated to the calling code.

There is a difference between what the `match` expression from Listing 9-6 does and what the `?` operator does: error values that have the `?` operator called on them go through the `from` function, defined in the `From` trait in the standard library, which is used to convert values from one type into another. When the `?` operator calls the `from` function, the error type received is converted into the error type defined in the return type of the current function. This is useful when a function returns one error type to represent all the ways a function might fail, even if parts might fail for many different reasons.

For example, we could change the `read_username_from_file` function in Listing 9-7 to return a custom error type named `OurError` that we define. If we also define `impl From<io::Error> for OurError` to construct an instance of `OurError` from an `io::Error`, then the `?` operator calls in the body of `read_username_from_file` will call `from` and convert the error types without needing to add any more code to the function.

In the context of Listing 9-7, the `?` at the end of the `File::open` call will return the value inside an `Ok` to the variable `username_file`. If an error occurs, the `?` operator will return early out of the whole function and give any `Err` value to the calling code. The same thing applies to the `?` at the end of the `read_to_string` call.

The `?` operator 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 `?`, as shown in Listing 9-8.

Filename: src/main.rs

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

Listing 9-8: Chaining method calls after the `?` operator

We've moved the creation of the new `String` in `username` to the beginning of the function; that part hasn't changed. Instead of creating a variable `username_file`, 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 `username` when both `File::open` and `read_to_string` succeed rather than returning errors. The functionality is again the same as in Listing 9-6 and Listing 9-7; this is just a different, more ergonomic way to write it.

Listing 9-9 shows a way to make this even shorter using `fs::read_to_string`.

Filename: src/main.rs

```
use std::fs;
use std::io;

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

Listing 9-9: Using `fs::read_to_string` instead of opening and then reading the file

Reading a file into a string is a fairly common operation, so the standard library provides the convenient `fs::read_to_string` function that opens the file, creates a new `String`, reads the contents of the file, puts the contents into that `String`, and returns it. Of course, using `fs::read_to_string` doesn't give us the opportunity to explain all the error handling, so we did it the longer way first.

Where The ? Operator Can Be Used

The `?` operator can only be used in functions whose return type is compatible with the value the `?` is used on. This is because the `?` operator is defined to perform an early return of a value out of the function, in the same manner as the `match` expression we defined in Listing 9-

6. In Listing 9-6, the `match` was using a `Result` value, and the early return arm returned an `Err(e)` value. The return type of the function has to be a `Result` so that it's compatible with this `return`.

In Listing 9-10, let's look at the error we'll get if we use the `?` operator in a `main` function with a return type that is incompatible with the type of the value we use `?` on.

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")?;
}
```

Listing 9-10: Attempting to use the `?` in the `main` function that returns `()` won't compile.

This code opens a file, which might fail. The `?` operator follows the `Result` value returned by `File::open`, but this `main` function has the return type of `()`, not `Result`. When we compile this code, we get the following error message:

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns
`Result` or `Option` (or another type that implements `FromResidual`)
 --> src/main.rs:4:48
  |
3 | fn main() {
  | --------- this function should return `Result` or `Option` to accept `?`
4 |     let greeting_file = File::open("hello.txt")?;
  |                                                ^ cannot use the `?` operator
in a function that returns `()`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not
implemented for `()`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` (bin "error-handling") due to 1 previous
error
```

This error points out that we're only allowed to use the `?` operator in a function that returns `Result`, `Option`, or another type that implements `FromResidual`.

To fix the error, you have two choices. One choice is to change the return type of your function to be compatible with the value you're using the `?` operator on as long as you have no restrictions preventing that. The other choice is to use a `match` or one of the `Result<T, E>` methods to handle the `Result<T, E>` in whatever way is appropriate.

The error message also mentioned that `?` can be used with `Option<T>` values as well. As with using `?` on `Result`, you can only use `?` on `Option` in a function that returns an `Option`. The behavior of the `?` operator when called on an `Option<T>` is similar to its behavior when called on a `Result<T, E>`: if the value is `None`, the `None` will be returned early from the function at that point. If the value is `Some`, the value inside the `Some` is the resultant value of the expression, and the function continues. Listing 9-11 has an example of a function that finds the last character of the first line in the given text.

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

Listing 9-11: Using the `?` operator on an `Option<T>` value

This function returns `Option<char>` because it's possible that there is a character there, but it's also possible that there isn't. This code takes the `text` string slice argument and calls the `lines` method on it, which returns an iterator over the lines in the string. Because this function wants to examine the first line, it calls `next` on the iterator to get the first value from the iterator. If `text` is the empty string, this call to `next` will return `None`, in which case we use `?` to stop and return `None` from `last_char_of_first_line`. If `text` is not the empty string, `next` will return a `Some` value containing a string slice of the first line in `text`.

The `?` extracts the string slice, and we can call `chars` on that string slice to get an iterator of its characters. We're interested in the last character in this first line, so we call `last` to return the last item in the iterator. This is an `Option` because it's possible that the first line is the empty string; for example, if `text` starts with a blank line but has characters on other lines, as in `"\nhi"`. However, if there is a last character on the first line, it will be returned in the `Some` variant. The `?` operator in the middle gives us a concise way to express this logic, allowing us to implement the function in one line. If we couldn't use the `?` operator on `Option`, we'd have to implement this logic using more method calls or a `match` expression.

Note that you can use the `?` operator on a `Result` in a function that returns `Result`, and you can use the `?` operator on an `Option` in a function that returns `Option`, but you can't mix and match. The `?` operator won't automatically convert a `Result` to an `Option` or vice versa; in those cases, you can use methods like the `ok` method on `Result` or the `ok_or` method on `Option` to do the conversion explicitly.

So far, all the `main` functions we've used return `()`. The `main` function is special because it's the entry point and exit point of an executable program, and there are restrictions on what its return type can be for the program to behave as expected.

Luckily, `main` can also return a `Result<(), E>`. Listing 9-12 has the code from Listing 9-10, but we've changed the return type of `main` to be `Result<(), Box<dyn Error>>` and added a

return value `Ok(())` to the end. This code will now compile.

Filename: src/main.rs

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt")?;

    Ok(())
}
```

Listing 9-12: Changing `main` to return `Result<(), E>` allows the use of the `?` operator on `Result` values.

The `Box<dyn Error>` type is a *trait object*, which we'll talk about in the "Using Trait Objects that Allow for Values of Different Types" section in Chapter 17. For now, you can read `Box<dyn Error>` to mean "any kind of error." Using `?` on a `Result` value in a `main` function with the error type `Box<dyn Error>` is allowed because it allows any `Err` value to be returned early. Even though the body of this `main` function will only ever return errors of type `std::io::Error`, by specifying `Box<dyn Error>`, this signature will continue to be correct even if more code that returns other errors is added to the body of `main`.

When a `main` function returns a `Result<(), E>`, the executable will exit with a value of `0` if `main` returns `Ok(())` and will exit with a nonzero value if `main` returns an `Err` value. Executables written in C return integers when they exit: programs that exit successfully return the integer `0`, and programs that error return some integer other than `0`. Rust also returns integers from executables to be compatible with this convention.

The `main` function may return any types that implement the `std::process::Termination` trait, which contains a function `report` that returns an `ExitCode`. Consult the standard library documentation for more information on implementing the `Termination` trait for your own types.

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 call `panic!` and when you should return `Result`? When code panics, there's no way to recover. You could call `panic!` for any error situation, whether there's a possible way to recover or not, but then you're making the decision that a situation is unrecoverable on behalf of the calling code. When you choose to return a `Result` value, you give the calling code options. The calling code could choose to attempt to recover in a way that's appropriate for its situation, or it could decide that an `Err` value in this case is unrecoverable, so it 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.

In situations such as examples, prototype code, and tests, it's more appropriate to write code that panics instead of returning a `Result`. Let's explore why, then discuss situations in which the compiler can't tell that failure is impossible, but you as a human can. The chapter will conclude with some general guidelines on how to decide whether to panic in library code.

Examples, Prototype Code, and Tests

When you're writing an example to illustrate some concept, also including robust error-handling code 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 you'd want 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, you'd want the whole test to fail, even if that method isn't the functionality under test. Because `panic!` is how a test is marked as a failure, calling `unwrap` or `expect` is exactly what should happen.

Cases in Which You Have More Information Than the Compiler

It would also be appropriate to call `unwrap` or `expect` 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's perfectly acceptable to call `unwrap`, and even better to document the reason you think you'll never have an `Err` variant in the `expect` text. Here's an example:

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1"
    .parse()
    .expect("Hardcoded IP address should be valid");
```

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 `expect` 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 a possibility because 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 rather than 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. Mentioning the assumption that this IP address is hardcoded will prompt us to change `expect` to better error-handling code if, in the future, we need to get the IP address from some other source instead.

Guidelines for Error Handling

It's advisable to have your code panic when it's possible that your code could end up in a bad state. In this context, a *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 something that is unexpected, as opposed to something that will likely happen occasionally, like a user entering data in the wrong format.
- Your code after this point needs to rely on not being in this bad state, rather than checking for the problem at every step.
- There's not a good way to encode this information in the types you use. We'll work through an example of what we mean in the "Encoding States and Behavior as Types" section of Chapter 17.

If someone calls your code and passes in values that don't make sense, it's best to return an error if you can so the user of the library can decide what they want to do in that case. However, in cases where continuing could be insecure or harmful, the best choice might be to call `panic!` and alert the person using your library to the bug in their code so 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.

However, when failure is expected, it's more appropriate to return a `Result` than to make a `panic!` call. Examples 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, returning a `Result` indicates that failure is an expected possibility that the calling code must decide how to handle.

When your code performs an operation that could put a user at risk if it's called using invalid 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 the standard library will call `panic!` if you attempt an out-of-bounds memory 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's not a kind of error you want the calling code 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.

However, having lots of error checks in all of your functions would be verbose and annoying. Fortunately, you can use Rust's type system (and thus the type checking done by the compiler) to do many 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 such as `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 in which our code asked the user to guess a number between 1 and 100. We never validated that the user's guess was between those numbers before checking it against our secret number; we only validated that the guess was positive. In this case, the consequences were not very dire: our output of "Too high" or "Too low" would still be correct. But it would be a useful enhancement to guide the user toward valid guesses and have different behavior when the user guesses a number that's out of range versus when the 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, and then add a check for the number being in range, like so:

Filename: src/main.rs

```
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 whether 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 were absolutely critical that the program only operated on values between 1 and 100, and it had many functions with this requirement, having a check like this in every function would be tedious (and might impact performance).

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 9-13 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.

Filename: src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {value}.");
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Listing 9-13: 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 an `i32`. 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 `i32` and to return a `Guess`. The code in the body of the `new` function tests `value` to make sure it's between 1 and 100. If `value` doesn't pass this test, we make a `panic!` call, which will alert the programmer who is writing the calling code that they have a bug they need to fix, because 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 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 an `i32`. This kind of method is sometimes called a *getter* because 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 be private so code using the `Guess` struct is not allowed to set `value` directly: code outside the module *must* use the `Guess::new` function to create an instance of `Guess`, thereby ensuring 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 an `i32` 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 you've seen useful ways that the standard library uses generics with the `Option` and `Result` enums, we'll talk about how generics work and how you can use them in your code.