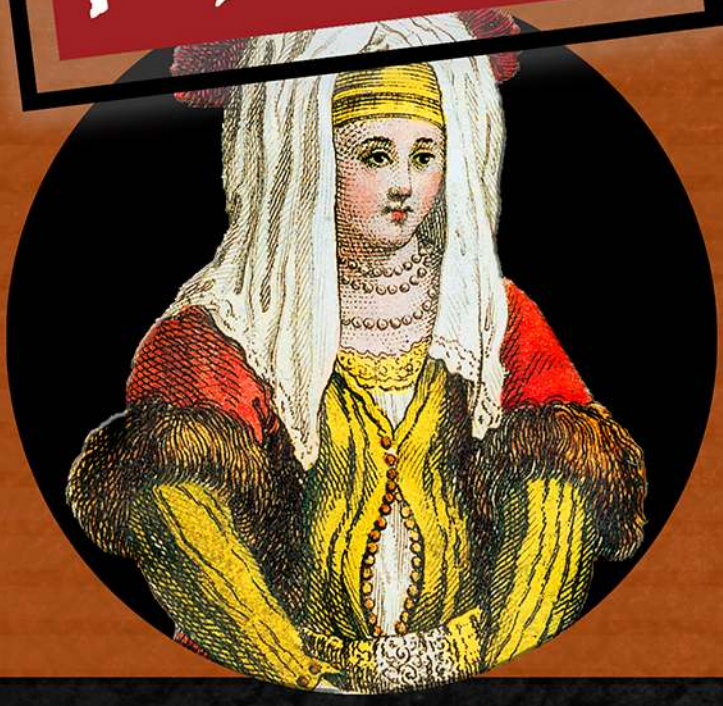
Author Picks

FREE



A Rust Sampler

Carol Nichols and Jake Goulding

manning

A Rust Sampler

by Carol Nichols and Jake Goulding

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Candace Gillhoolley, cagi@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617299162

contents

Introduction

A sampler in needlework is a piece containing many small designs worked as a reference to different patterns or to showcase skills. This eBook is a collection of interesting bits about working with the Rust programming language that will be useful as you progress from the introductory texts into creating projects in Rust. You'll learn about what the 2018 Edition of Rust means and how it'll affect you. You'll work through a Test Driven Development example. You'll experience a concurrency bug in Ruby and see how Rust prevents it at compile time. You'll see why self-referential structs in Rust aren't allowed and what to do instead. And you'll learn a best practice for the types of parameters in functions. These articles will hone your Rust skills and serve as references for your future Rust work.

All examples use Rust 1.35 and the 2018 Edition, but thanks to Rust's stability guarantees, the examples that compile will continue to compile with newer versions of Rust. Error messages may be different, and code that doesn't compile might begin compiling.

What the 2018 Edition of Rust Means to You

Rust version 1.31 introduced a new concept, editions, and specifically added the Rust 2018 Edition. A Rust edition consists of a small number of breaking changes that are opt-in. Let's explore in more detail what editions are and what they mean to you.

What Is an Edition?

Editions in Rust are a way for the language to evolve without breaking backward compatibility. All editions will be supported in all compiler versions released after each edition is created. Switching to a new edition is always opt-in: if you don't change the `edition` key in your project's *Cargo.toml* file, you can upgrade your Rust compiler without breaking your code, according to Rust's stability guarantees. Because there was no concept of editions when Rust 1.0 released, any Rust project without an `edition` key is compiled according to the original Rust syntax. This original syntax is now called Rust 2015 Edition because that's when Rust 1.0 came out.

Editions are supported at the level of translating source code to Rust Mid-level Intermediate Representation (MIR), a structure similar to an Abstract Syntax Tree on which most of the Rust compiler analysis, like borrow checking, happens. Therefore, the breaking changes that can be made using editions are small, such as adding a new keyword (which might cause your code to stop compiling if you were using that keyword as a function or variable name, for example). Editions won't make major changes to Rust, like removing the borrow checker.

Editions are also a way to wrap up all the incremental improvements that are released every six weeks. The community of people working on Rust focuses on polishing and documenting all the features released over a longer period of time on the order of years. The edition presents a nice package of an improved Rust language to those who might not have been following Rust closely, so that they know to perhaps take another look. The Edition Guide details all the changes made in the 2018 Edition, and the official documentation has been updated as well.

Another key factor in the way editions are implemented is that Rust 2015 crates can depend on Rust 2018 crates and vice versa—there's no ecosystem split. Each crate decides the edition it uses, and the compiler builds each crate independently according to that configuration.

Further aspects of editions are more relevant to different groups: people who are starting to learn Rust now that the 2018 Edition has been released, and people who have learned 2015 Edition Rust and are working on a 2015 Edition codebase.

New Rust Developers Learning Rust Post-2018 Edition

Now that the 2018 Edition has been released, this is a great time to start learning Rust! Rust has been stable since 1.0 was released, but there were some rough edges and inconsistencies that developers using Rust discovered and developers working on Rust were able to fix with the new edition. You don't have to worry about being surprised by the way paths work with `use`, or why and where you need to say `extern crate` to use

a library even though you've already put it in your *Cargo.toml* file. Many of the improvements were specifically designed to make Rust easier to learn and more productive to use.

Keep in mind, though, that older tutorials may be using Rust 2015 syntax. Even older tutorials may be using pre-1.0 Rust syntax, which was very different from what got stabilized (tildes everywhere)! If you see Rust code that looks different from what you learned, it might be from an older edition; consult The Edition Guide for explanations of changes.

Rust Developers Working on Pre-2018 Edition Code

The most important feature of editions for existing developers is that switching to a new edition is entirely opt-in. Stability is a very important value of Rust, so editions were designed such that each project gets to decide when to switch editions independently of upgrading the Rust compiler. People working on Rust want to ensure that you never fear upgrading to the next stable version of the compiler.

Even with the small number of breaking changes that are allowed, the hope is you won't fear upgrading to a new edition either. The `rustfix` tool available through Cargo with the `cargo fix` command can do a lot of the conversion work for you.

Deciding whether and when to switch to a newer edition involves trade-offs that each project might weigh differently. The newer edition will have nicer syntax and will align with newer tutorials. Developers who have only used 2018 Edition Rust might be more comfortable in your project if it has switched to the newer idioms. And even though the hope is that new features are supported in as many editions as possible, sometimes a feature will be implemented in the new edition first (non-lexical lifetimes is one such feature).

If your project is a library used by people on many different Rust compiler versions, keep in mind that when you switch to a newer edition, you are implicitly saying your library only supports versions of the compiler starting from when that edition became available. That is, if you upgrade to Rust 2018 Edition, your library will only work in versions of Rust greater than 1.31. This may be desirable for your project, so that you aren't spending time supporting problems that might only happen with older Rust versions. Or this may be undesirable if your goal is to support those older versions.

And, of course, any change to your code has the potential to introduce bugs or changes in behavior. The Rust compiler has an extensive test suite to prevent this, but no test suite is perfect. You should absolutely take into account the time needed to review and test edition-related code changes when deciding about switching to a new edition.

Overall, Rust editions are designed to avoid the need for a Rust 2.0 that breaks backward compatibility in major ways, splits the ecosystem, or drops support for existing Rust code. Rust is prepared to maintain its stability guarantees for the foreseeable future to be a reliable platform on which to build.

A Test Driven Development Example in Rust

This is an example of how to work through the Prime Factors Kata using Test Driven Development. The kata, as we're going to interpret it, is:

> *Write a function that takes an integer greater than 1 and returns a vector of its prime factors, which are the prime numbers that divide that integer exactly, in ascending order. For example, the prime factors of 12 are 2, 2, 3.*

Test Driven Development (TDD) is a software development method where you repeat these three steps:

1. Red: Write a failing test.
2. Green: Write just enough code to make the test pass.
3. Refactor: Keeping the tests green, improve the code as much as you can.

These steps enable you to explore the design space of a solution guided by the constraints of getting and keeping the test cases passing. Let's see what Test Driven Development feels like in Rust by implementing a program to find the prime factors of a number.

If you'd like to try it yourself without spoilers, here is where you should stop reading and pick up again when you're ready!

Let's generate a new library using `cargo new --lib prime_factors`, go into the created *prime_factors* directory, then start by adding a failing test in *src/lib.rs*:

```
#[cfg(test)]
mod tests {
    #[test]
    fn prime_factors_of_two() {
        assert_eq!(prime_factors(2), [2]);
    }
}
```

Next, we run `cargo test` to check that the test fails. It does, in a way, by failing to compile:

```
$ cargo test
   Compiling prime_factors v0.1.0 (/projects/prime_factors)
error[E0425]: cannot find function `prime_factors` in this scope
 --> src/lib.rs:5:20
  |
5 |         assert_eq!(prime_factors(2), [2]);
  |                    ^^^^^^^^^^^^^ not found in this scope
```

One way to think about TDD in Rust is that it has two "red" states—failing to compile and failing tests. The example is in the first kind of red right now. We won't necessarily hit both red states in every TDD cycle, nor do they always happen in the same order. It doesn't really matter whether it's the compiler or the tests that are helping to make the code better. The next step is the same: read the first problem in the output of running the tests and do the simplest thing to respond to that problem.

The first message here is `cannot find function `prime_factors` in this scope`, and the simplest thing to do is define that function without any parameters and with the implicit unit type return value:

```
pub fn prime_factors() {}

#[cfg(test)]
mod tests {
    // ...snip...
}
```

We're defining this function as public because this function would be in our library's public API if we were going to release this library for other projects to use. If we don't use the `pub` keyword, we'll get unused code warnings because we won't have any non-test code using it.

Now when we run `cargo test`, we get:

```
error[E0425]: cannot find function `prime_factors` in this scope
 --> src/lib.rs:7:20
  |
7 |         assert_eq!(prime_factors(2), [2]);
  |                    ^^^^^^^^^^^^^ not found in this scope
help: possible candidate is found in another module, you can import it into scope
  |
1 | use crate::prime_factors;
```

Interesting! Same error message, but there's different help text. The `prime_factors_of_two` test function is in a `tests` module, and the `prime_factors` function is in the parent module of the `tests` module. So we need to bring the function into the scope of the `tests` module, and the suggested code will do so:

```
#[cfg(test)]
mod tests {
    use crate::prime_factors;
    // ...snip...
}
```

Let's try `cargo test` again:

```
error[E0061]: this function takes 0 parameters but 1 parameter was supplied
 --> src/lib.rs:9:20
  |
1 | pub fn prime_factors() {}
  | ---------------------- defined here
...
9 |         assert_eq!(prime_factors(2), [2]);
  |                    ^^^^^^^^^^^^^^^^^ expected 0 parameters

error[E0308]: mismatched types
 --> src/lib.rs:9:9
  |
9 |         assert_eq!(prime_factors(2), [2]);
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected (), found array of 1
    elements
  |
  = note: expected type `()`
             found type `[{integer}; 1]`
```

Easy enough; we do indeed want the function to take one parameter of type `i32`:

```
pub fn prime_factors(num: i32) {}
```

And now `cargo test` results in a warning and a different error:

```
warning: unused variable: `num`
 --> src/lib.rs:1:22
  |
1 | pub fn prime_factors(num: i32) {}
  |                      ^^^ help: consider prefixing with an underscore: `_num`
  |
  = note: #[warn(unused_variables)] on by default

error[E0308]: mismatched types
 --> src/lib.rs:9:9
  |
9 |     assert_eq!(prime_factors(2), [2]);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected (), found array of 1
    elements
  |
  = note: expected type `()`
             found type `[{integer}; 1]`
```

We're going to let this warning continue to happen for now because we will be using this variable eventually, but we're not going to use it this step. We're more concerned with the first error in the test that tells us the values we're trying to assert are equal aren't the same type. The left side is the unit type, and the right side is an array of one element.

So we're going to fix this error first by specifying the return type as `Vec<i32>`:

```
pub fn prime_factors(num: i32) -> Vec<i32> {}
```

which gets us:

```
error[E0308]: mismatched types
 --> src/lib.rs:1:35
  |
1 | pub fn prime_factors(num: i32) -> Vec<i32> {}
  |        -------------              ^^^^^^^^ expected struct
    `std::vec::Vec`, found ()
  |         |
  |         this function's body doesn't return
  |
  = note: expected type `std::vec::Vec<i32>`
             found type `()`
```

The function body is expecting to return a `Vec<i32>`, but the empty body returns the unit type, `()`. Let's get this code compiling and the test passing with the simplest implementation that could possibly work:

```
pub fn prime_factors(num: i32) -> Vec<i32> {
    vec![2]
}
```

And we're green (with some warnings that we're going to continue to ignore)! There isn't anything to refactor here, so on to another failing test that we'll add to the `tests` module:

```
#[test]
fn prime_factors_of_three() {
    assert_eq!(prime_factors(3), [3]);
}
```

This test compiles but fails:

```
running 2 tests
test tests::prime_factors_of_two ... ok
test tests::prime_factors_of_three ... FAILED

failures:

---- tests::prime_factors_of_three stdout ----
thread 'tests::prime_factors_of_three' panicked at 'assertion failed: `(left
     == right)`
  left: `[2]`,
 right: `[3]`', src/lib.rs:16:9
note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.

failures:
      tests::prime_factors_of_three

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Yep, as expected, the hardcoded vector containing 2 that the function returns doesn't equal the expected value of a vector containing 3. The next simplest thing to get both these tests to pass is returning the parameter in a vector, which should produce the correct answer for any prime number:

```
pub fn prime_factors(num: i32) -> Vec<i32> {
    vec![num]
}
```

This compiles, passes the tests, *and* gets rid of those warnings about not using `num`. Onward to a new failing test!

```
#[test]
fn prime_factors_of_four() {
    assert_eq!(prime_factors(4), [2, 2]);
}
```

Aha, something a bit more interesting because 4 isn't prime. This fails as expected:

```
---- tests::prime_factors_of_four stdout ----
thread 'tests::prime_factors_of_four' panicked at 'assertion failed: `(left
     == right)`
  left: `[4]`,
 right: `[2, 2]`', src/lib.rs:21:9
```

Let's go with a recursive solution here: if the number is divisible by 2 and greater than 2, then 2 is one of its prime factors and the rest of its prime factors are the prime factors of the number divided by 2. We'll create a new vector holding 2, then combine that with the result of recursing by calling `prime_factors` with the current number divided by 2.

If the number is odd or 2, then we're in the base case where we can return a vector containing the number. Here's the code:

```
pub fn prime_factors(num: i32) -> Vec<i32> {
    if num % 2 == 0 && num > 2 {
        let mut answer = vec![2];
        answer.extend(&prime_factors(num / 2));
        answer
    } else {
        vec![num]
    }
}
```

There are a lot of conditionals and a lot of duplication of the number 2. We could refactor some at this point, but we're not going to. This is where experience and judgement calls come into putting TDD into practice: this isn't the general solution, so there's going to be a lot of change coming up. Refactoring at this point would potentially do more harm than good and just have to be undone later. But don't let this become an excuse to not refactor!

If we add tests for 5, 6, 7, and 8, they'll all pass at this point without changing the code. You could argue that skipping 5 and 7 is acceptable if you're confident enough that all prime numbers will be correct if 2 and 3 are correct, thus making more tests for prime numbers uninteresting and unnecessary. Another judgement call.

It's also fine to write a whole bunch of tests while test driving, but once you understand the domain better, delete some tests that are covering the same cases before calling your implementation complete. That way, you'll have fewer tests to wait for to run and fewer tests that will all break at once if one part of the logic changes. There's no part of TDD that says you have to keep all the tests written during the process!

The next interesting test case that fails is 9 because it isn't prime and 2 is not any of its prime factors. Here's the test:

```
#[test]
fn prime_factors_of_nine() {
    assert_eq!(prime_factors(9), [3, 3]);
}
```

And the test failure message:

```
---- tests::prime_factors_of_nine stdout ----
thread 'tests::prime_factors_of_nine' panicked at 'assertion failed: `(left
     == right)`
  left: `[9]`,
 right: `[3, 3]`', src/lib.rs:32:9
```

Because the recursive solution isn't great, let's switch to an iterative approach. This means we'll make more parts of the code mutable. Now, Rust tries to discourage you from making things mutable by making all variables immutable by default, but that doesn't mean it's *bad* or *wrong* necessarily. Here's an iterative approach that keeps track of the prime factors it's found. While `num` isn't 1, it starts looking for candidate factors from 2 and increases until it finds a factor. When it does, it adds that factor to the `answer` vector, divides `num` by that factor, and continues.

```
pub fn prime_factors(mut num: i32) -> Vec<i32> {
    let mut answer = vec![];

    while num != 1 {
        let mut candidate = 2;

        while candidate <= num {
            if num % candidate == 0 {
                answer.push(candidate);
                num = num / candidate;
                break;
            } else {
                candidate += 1;
            }
      }
    }
    answer
}
```

This is starting to feel like a correct general solution: the test for 9 passes. Let's add two more tests: one for a larger prime (101 => [101]) and one for a larger non-prime (48 => [2, 2, 2, 2, 3]). Again, these aren't really necessary since they aren't covering any cases that aren't already covered (and they do pass right away). But in this case, they don't add much time to the test run and they could have caught problems that might not show up with smaller numbers. Adding more test cases that might have the same coverage as other test cases is a trade-off between the work it takes to write them, the time it takes to run them, and the likelihood that the test cases might end up having different and important coverage from the other test cases.

Refactoring time! Let's try to get rid of the mutable `candidate` variable by switching the `while` loop to a `for` loop over a range:

```
pub fn prime_factors(mut num: i32) -> Vec<i32> {
    let mut answer = vec![];
    while num != 1 {
        for candidate in 2..=num {
            if num % candidate == 0 {
                answer.push(candidate);
                num = num / candidate;
                break;
            }
        }
    }
    answer
}
```

This code still passes all the tests and is a bit more concise. However, there's a clever solution in Java that only uses three lines of code for the algorithm, excluding the function signature and initialization of a list to hold the results! Can we get the Rust solution to be that small? Let's peek at this short Java solution:

```
for (int candidate = 2; num > 1; candidate++)
    for (; num % candidate == 0; num /= candidate)
        result.add(candidate);
```

This feels a bit too clever, with the `for` loop conditions not matching the iterator index. It's neat code golf, though. Rust doesn't have this `for` loop syntax, but here's a close translation:

```
pub fn prime_factors(mut num: i32) -> Vec<i32> {
    let mut answer = vec![];
    let mut candidate = 2;

    while num > 1 {
        while num % candidate == 0 {
            answer.push(candidate);
            num /= candidate;
        }
        candidate += 1;
    }

    answer
}
```

This is better than the previous solution we ended up with, in that the code isn't going back out to the outer loop if it can continue to divide by the candidate factor over and over.

But is this the most idiomatic Rust solution? It doesn't really feel like it, with all the mutable variables still in there. Let's give the recursive solution a try again, using the knowledge we gained looking at iterative solutions:

```
pub fn prime_factors(num: i32) -> Vec<i32> {
    for candidate in 2..num {
        if num % candidate == 0 {
            let mut answer = vec![candidate];
            answer.extend(&prime_factors(num / candidate));
            return answer;
        }
    }
    vec![num]
}
```

Down to just the mutable `answer` vector. To get this to be correct where the previous recursive solution was incorrect, we had to add the outer `for` loop to increment the candidate factor. When `num` is 2, the range that the `for` loop iterates over will be empty, so it won't do any work and will return a vector containing 2.

An interesting bit about code that solves this problem is that it produces one value at a time, and the subsequent values don't need to know about the previously pro-

duced values at all. That is, at no point in the iterative solutions did we need to look at what we pushed in to the `answer` vector at all; and at no point in the recursive solution did the recursive calls to `prime_factors` need to know about the `vec![candidate]`.

Rust has a special name for something that produces one value in a sequence at a time: an Iterator. We can solve this problem by implementing the Iterator trait: rather than keeping the state in variables in a function, we'll keep the state in fields of a struct. First, let's define a public struct named `PrimeFactorsIterator` that will have private fields for `num` and the current candidate:

```
pub struct PrimeFactorsIterator {
    num: i32,
    candidate: i32,
}
```

Next, let's create a public associated function that constructs instances of `PrimeFactorsIterator` and starts the candidate factor at 2:

```
impl PrimeFactorsIterator {
    pub fn new(num: i32) -> PrimeFactorsIterator {
        PrimeFactorsIterator { num, candidate: 2 }
    }
}
```

And now we're going to implement the `Iterator` trait with `i32` as the type of values the iterator will return and a definition for the required `next` method that will produce one factor or indicate that there are no more factors by returning `None`:

```
impl Iterator for PrimeFactorsIterator {
    type Item = i32;

    fn next(&mut self) -> Option<i32> {
        while self.num > 1 {
            while self.num % self.candidate == 0 {
                self.num /= self.candidate;
                return Some(self.candidate);
            }
            self.candidate += 1;
        }
        None
    }
}
```

While this solution is longer than the other solutions in general, to get all the factors in a vector, the `prime_factors` function now only needs to create a new instance of the iterator and collect all the values it produces, which is kind of neat:

```
pub fn prime_factors(num: i32) -> Vec<i32> {
    PrimeFactorsIterator::new(num).collect()
}
```

Out of all these solutions, which do you like the best? Which is the easiest to understand? Which is the most idiomatic Rust? Which was driven the most by the tests? Ultimately, TDD is a practice that lets you consider tradeoffs as you explore solutions guided by tests that ensure correctness, and Rust has many tradeoffs to consider!

Comparison of Concurrency in Ruby and Rust

Rust promises that data races are compile-time errors. If you're used to programming in a language without such guarantees, such as Ruby, you've probably experienced pain in trying to implement multi-threaded code without data races or in trying to debug a subtle data race. Or perhaps you avoided writing concurrent code at all because of the difficulty! In Rust, the compiler helps you to get concurrency right, and there are no debugging data races at runtime because Rust won't even let them compile.

Let's explore what a data race is, what a data race looks like in Ruby, what it looks like if we try to port that Ruby to Rust, and what error messages we get that prevent us from making the same mistake.

By "data race" we mean a condition in code where some code is reading a value from a particular location and other code is writing to the same location at about the same time. Sometimes when running the program, the read happens before the write, which causes one result. Other times the write happens before the read, which causes another result. A data race can also happen when both parts of the code are writing to the same location in potentially different orders. There can also be more than two parts of the code involved, as long as at least one part is writing to the shared location. This nondeterministic behavior can be the source of bugs that are hard to detect, reproduce, and fix.

Here's a trivial data race in Ruby (we're using v2.6.3): using multiple threads to increment one number from 1 to 500. This is contrived and not a task you would ever want to actually implement in this way, but this will work nicely for demonstration purposes. Imagine this code representing multiple threads updating one aggregate value with the results of each thread's computation:

```
THREADS = 10
COUNT   = 50

$x = 1

THREADS.times.map { |t|
  Thread.new {
      COUNT.times { |c|
      a  = $x + 1
      sleep 0.000001
      puts "Thread #{t} wrote #{a}"
      $x =  a
      }
  }
}.each(&:join)

if $x != THREADS * COUNT + 1
  puts "Got $x = #{$x}."
  puts "Expected to get #{THREADS * COUNT + 1}."
else
  puts "Did not reproduce the issue."
end
```

This code has a global variable named `$x`. Each thread does the following:

- Reads the value from the global variable `$x`
- Adds 1 to the value it read
- Stores the result in a local variable named `a`
- Sleeps for a tiny amount of time
- Prints to the screen which thread this is and what value it has in a, so that we can see the order in which reads and writes are happening
- Writes the resulting value in local variable a to the global variable `$x`

In the time that one thread sleeps, another thread might read or write to the global variable, causing the reads and writes to happen in an order that produces a result other than what we expect. Here are some parts of the output from one run:

```
Thread 0 wrote 2
Thread 0 wrote 3
...
Thread 1 wrote 78
Thread 3 wrote 29
...
Thread 3 wrote 59
Thread 4 wrote 29
...
Thread 4 wrote 78
Thread 3 wrote 60
...
Thread 0 wrote 50
Thread 0 wrote 51
Got $x = 51.
Expected to get 501
```

The number 51 is certainly not the 501 we expected to get; we've succeeded in implementing a data race in Ruby.

Let's see what it would look like to port this code to Rust as faithfully as possible:

```
use std::thread;
use std::time::Duration;

const THREADS: usize = 10;
const COUNT: usize = 50;

fn main() {
    let mut x = 1;

    let mut handles = vec![];

    for t in 0..THREADS {
        handles.push(thread::spawn(|| {
            for _ in 0..COUNT {
                let a = x + 1;
                thread::sleep(Duration::from_micros(1));
                println!("Thread {} wrote {}", t, a);
                x = a;
            }
        }))
```

```
    }

    for handle in handles {
        handle.join().expect("Thread shouldn't have panicked");
    }

    if x != THREADS * COUNT + 1 {
        println!("Got x = {}", x);
        println!("Expected to get {}", THREADS * COUNT + 1);
    } else {
        println!("Did not reproduce the issue.");
    }
}
```

Sure enough, we get compiler errors! Thanks, Rust! Here's the first error:

```
error[E0499]: cannot borrow `x` as mutable more than once at a time
  --> src/main.rs:13:36
   |
13 |           handles.push(thread::spawn(|| {
   |                        -              ^^ mutable borrow starts here in
    previous iteration of loop
   |  _________________________|
   | |
14 | |             for _ in 0..COUNT {
15 | |                 let a = x + 1;
   | |                         - borrows occur due to use of `x` in closure
16 | |                 thread::sleep(Duration::from_micros(1));
...  |
19 | |             }
20 | |         }))
   | |__________- argument requires that `x` is borrowed for `'static`
```

The main message of this error is `cannot borrow `x` as mutable more than once at a time`. The error goes on to point out that the mutable borrow starts at the beginning of the closure passed to the thread in a previous iteration of the loop. So this is saying the multiple threads created by the outer `for` loop can't all borrow `x` mutably. Rust has prevented one of the causes of a data race: multiple threads trying to write to the same location!

There are more problems with this code! Here's the next compiler error:

```
error[E0373]: closure may outlive the current function, but it borrows `t`,
    which is owned by the current function
  --> src/main.rs:13:36
   |
13 |         handles.push(thread::spawn(|| {
   |                                    ^^ may outlive borrowed value `t`
...
17 |                 println!("Thread {} wrote {}", t, a);
   |                                                - `t` is borrowed here
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:13:22
   |
```

```
13 |           handles.push(thread::spawn(|| {
   |  _________________________^
14 | |             for _ in 0..COUNT {
15 | |                 let a = x + 1;
16 | |                 thread::sleep(Duration::from_micros(1));
...  |
19 | |             }
20 | |         }))
   | |_________^
help: to force the closure to take ownership of `t` (and any other referenced
     variables), use the `move` keyword
   |
13 |         handles.push(thread::spawn(move || {
   |                                    ^^^^^^^
```

This is saying the closure passed to the thread could continue running after the current function (meaning `main`). The compiler doesn't have any knowledge of the `join` function waiting for threads to finish, so even though we are ensuring `main` will outlive each thread, the compiler can't see that. By default, closures borrow values from their environment.

The help text after the error message says we can use the `move` keyword to force the closure to take ownership of all referenced variables. Let's try it by changing `thread::spawn(|| {` to `thread::spawn(move || {`.

Interestingly, this compiles! And the threads are printing out that they're incrementing numbers:

```
Thread 0 wrote 2
Thread 2 wrote 2
Thread 1 wrote 2
Thread 2 wrote 3
Thread 1 wrote 3
Thread 0 wrote 3
Thread 3 wrote 2
Thread 2 wrote 4
Thread 1 wrote 4
Thread 0 wrote 4
...
```

And at the end of the output, we see this:

```
...
Got x = 1
Expected to get 501
```

The reason this now compiles is the `move` keyword makes *all* the values referenced by the closure– so `t` *and* `x`. Because both are `Copy` types, each thread then owns a copy of `t` and `x`.

Because each thread increments the value in their `x` variable 50 times, the `x` in `main` doesn't get incremented at all. What we've changed here is that the threads are no longer writing to a shared location, so we've eliminated the data race. But this program isn't really doing what we want it to do.

So how *can* we use threads to increment a shared value in Rust? We are leaving that as an exercise for the reader! Here are some ideas on directions to try:

- Use a static atomic type for the shared value.
- Use an Arc<Mutex<T>> around the shared value.
- Use the crossbeam crate or the rayon crate, which provide safe abstractions for concurrent programming.

Rust's ownership and borrowing model prevents us from creating data races at compile time. In other languages, we can easily end up with data races that don't become noticeable until runtime. Rust gives us compiler errors so that we can fix data races earlier in our development process.

Why You Can't Store a Value and a Reference to that Value in the Same Struct

Let's say we have a value and we want to store that value and a reference to something inside that value in our own type:

```
struct Thing {
    count: u32,
}

struct Combined<'a>(Thing, &'a u32);

fn make_combined<'a>() -> Combined<'a> {
    let thing = Thing { count: 42 };

    Combined(thing, &thing.count)
}
```

Or say we have a value and we want to store that value and a reference to that value in the same structure:

```
struct Combined<'a>(Thing, &'a Thing);

fn make_combined<'a>() -> Combined<'a> {
    let thing = Thing::new();

    Combined(thing, &thing)
}
```

Or a case when we're not even taking a reference of the value:

```
struct Combined<'a>(Parent, Child<'a>);

fn make_combined<'a>() -> Combined<'a> {
    let parent = Parent::new();
    let child = parent.child();

    Combined(parent, child)
}
```

In each of these cases, we get an error that one of the values "does not live long enough". What does this error mean?

Let's look at a simple implementation of this:

```
struct Parent {
    count: u32,
}

struct Child<'a> {
    parent: &'a Parent,
}

struct Combined<'a> {
    parent: Parent,
    child: Child<'a>,
}

impl<'a> Combined<'a> {
    fn new() -> Self {
        let parent = Parent { count: 42 };
        let child = Child { parent: &parent };

        Combined { parent, child }
    }
}
```

This will fail to compile with the errors:

```
error[E0515]: cannot return value referencing local variable `parent`
  --> src/lib.rs:19:9
   |
17 |         let child = Child { parent: &parent };
   |                                     ------- `parent` is borrowed here
18 |
19 |         Combined { parent, child }
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^ returns a value referencing data owned
    by the current function

error[E0505]: cannot move out of `parent` because it is borrowed
  --> src/lib.rs:19:20
   |
14 | impl<'a> Combined<'a> {
   |      -- lifetime `'a` defined here
...
17 |         let child = Child { parent: &parent };
   |                                     ------- borrow of `parent` occurs here
18 |
19 |         Combined { parent, child }
   |         -----------^^^^^^---------
   |         |          |
   |         |          move out of `parent` occurs here
   |         returning this value requires that `parent` is borrowed for `'a`
```

To completely understand these errors, we have to think about how the values are represented in memory and what happens when we *move* those values. Let's annotate

`Combined::new` with some hypothetical memory addresses that show where values are located:

```
let parent = Parent { count: 42 };
// `parent` lives at address 0x1000 and takes up 4 bytes
// The value of `parent` is 42

let child = Child { parent: &parent };
// `child` lives at address 0x1010 and takes up 4 bytes
// The value of `child` is 0x1000

Combined { parent, child }
// The return value lives at address 0x2000 and takes up 8 bytes
// `parent` is moved to 0x2000
// `child` is ... ?
```

What should happen to `child`? If the value was just moved like `parent` was, then it would refer to memory that no longer is guaranteed to have a valid value in it. Any other piece of code is allowed to store values at memory address 0x1000. Accessing that memory assuming it was an integer could lead to crashes and/or security bugs, and it's one of the main categories of errors that Rust prevents.

This is exactly the problem that *lifetimes* prevent. A lifetime is a bit of metadata that allows programmers and the compiler to know how long a value will be valid at its **current memory location**. That's an important distinction as it's a common mistake Rust newcomers make. Rust lifetimes are *not* the time period between when an object is created and when it is destroyed!

As an analogy, think of it this way: During a person's life, they will reside in many different locations, each with a distinct address. A Rust lifetime is concerned with the address a person *currently resides at*, not about the address at the point they die in the future (although dying also changes their address). Every time they move it's relevant because their address is no longer valid.

It's also important to note that lifetime parameters *do not* change our code; our code controls the lifetimes—our lifetimes don't control the code. The pithy saying is "lifetimes are descriptive, not prescriptive".

Let's annotate `Combined::new` with some line numbers which we will use to highlight lifetimes:

```
{                                          // 0
    let parent = Parent { count: 42 };     // 1
    let child = Child { parent: &parent }; // 2
                                           // 3
    Combined { parent, child }             // 4
}                                          // 5
```

The *concrete lifetime* of `parent` is from 1 to 4, inclusive (which I'll represent as `[1,4]`). The concrete lifetime of `child` is `[2,4]`, and the concrete lifetime of the return value is `[4,5]`. It's possible to have concrete lifetimes that start at zero—that would represent the lifetime of a parameter to a function or something that existed outside of the block.

Note that the lifetime of `child` itself is `[2,4]`, but that it **refers to** a value with a lifetime of `[1,4]`. This is fine as long as the referring value becomes invalid before the referred-to value does. The problem occurs when we try to return `child` from the block. This would "over-extend" the lifetime beyond its natural length.

This new knowledge should explain the first two examples. The third one requires looking at the implementation of `Parent::child`. Chances are it will look something like this:

```
impl Parent {
    fn child(&self) -> Child { /* ... */ }
}
```

This uses *lifetime elision* to avoid writing explicit lifetime parameters. It is equivalent to:

```
impl Parent {
    fn child<'a>(&'a self) -> Child<'a> { /* ... */ }
}
```

In both cases, the method says that a `Child` structure will be returned that has been parameterized with the concrete lifetime of `self`. Said another way, the `Child` instance contains a reference to the `Parent` that created it, and thus cannot live longer than that `Parent` instance.

This also lets us recognize that something is really wrong with our creation function:

```
fn make_combined<'a>() -> Combined<'a> { /* ... */ }
```

Although we're more likely to see this written in a different form:

```
impl<'a> Combined<'a> {
    fn new() -> Combined<'a> { /* ... */ }
}
```

In both cases, there is no lifetime parameter being provided via an argument. This means the lifetime that `Combined` will be parameterized with isn't constrained by anything—it can be whatever the caller wants it to be. This is nonsensical because the caller could specify the `'static` lifetime, and there's no way to meet that condition.

How Do We Fix These Errors?

The easiest and most recommended solution is to not attempt to put a value and a reference to that value in the same structure together. Instead, place types that own data into a structure together and then provide methods that allow access to get references to the owned types or objects containing references as needed.

There is a special case where the lifetime tracking is overzealous: when we have something placed on the heap. This occurs when we use a `Box<T>`, for example. In this case, the structure that is moved contains a pointer into the heap. The pointed-at value will remain stable, but the address of the pointer itself will move. In practice, this doesn't matter, as we always follow the pointer.

The rental crate or the owning_ref crate are ways of representing this case, but they require that the base address *never move*. This rules out mutating vectors, which may cause a reallocation and a move of the heap-allocated values.

Why References Aren't Kept in Sync with the Values' Locations

After moving `parent` into the struct, why is the compiler not able to get a new reference to `parent` and assign it to `child` in the struct?

While it is theoretically possible to do this, doing so would introduce a large amount of complexity and overhead. Every time that the object is moved, the compiler would need to insert code to "fix up" the reference. This would mean that copying a struct is no longer a very cheap operation that just moves some bits around. It could even mean that code like this is expensive, depending on how good a hypothetical optimizer would be:

```
let a = Object::new();
let b = a;
let c = b;
```

Instead of forcing this to happen for *every* move, the programmer gets to *choose* when this will happen by creating methods that will take the appropriate references only when we call them.

There's one specific case where we *can* create a type with a reference to itself. We need to use something like `Option` to make it in two steps, though:

```
#[derive(Debug)]
struct WhatAboutThis<'a> {
    name: String,
    nickname: Option<&'a str>,
}

fn main() {
    let mut tricky = WhatAboutThis {
        name: "Annabelle".to_string(),
        nickname: None,
    };
    tricky.nickname = Some(&tricky.name[..4]);

    println!("{:?}", tricky);
}
```

This does work, in the sense that we have successfully created a struct that contains a reference to itself, but the created value is highly restricted—it can *never* be moved. Notably, this means it cannot be returned from a function or passed by-value to anything. A constructor function shows the same problem with the lifetimes as above:

```
fn creator<'a>() -> WhatAboutThis<'a> { /* ... */ }
```

Why It's Discouraged to Accept a Reference to a String, Vec, or Box as a Function Parameter

Here's some Rust code that takes a `&String` as a parameter:

```
fn awesome_greeting(name: &String) {
    println!("Wow, you are awesome, {}!", name);
}
```

And here's code that takes a reference to a `Vec`:

```
fn total_price(prices: &Vec<i32>) -> i32 {
    prices.iter().sum()
}
```

And code that takes a reference to a `Box`:

```
fn is_even(value: &Box<i32>) -> bool {
    **value % 2 == 0
}
```

These function definitions work, but they're not idiomatic Rust. The reason is that we can instead define functions to use `&str`, `&[T]` or `&T` as parameter types with no loss of genericity. Let's explore further.

One of the main reasons to use a `String` or a `Vec` is because they allow increasing or decreasing the capacity. However, when we use an immutable reference as a parameter type in a function definition, that function can't use any of those interesting methods on the `Vec` or `String`.

Accepting a `&String`, `&Vec` or `&Box` also **requires** an allocation before we can call the method. Unnecessary allocation is a performance loss. This is usually exposed right away when we try to call these methods in a test or a `main` method:

```
fn main() {
    awesome_greeting(&String::from("Anna"));

    total_price(&vec![42, 13, 1337])

    is_even(&Box::new(42))
}
```

We aren't able to call these functions with a static string literal, an array, or an integer literal already stored in the program's code. We're required to perform an extra allocation.

Another performance consideration is that `&String`, `&Vec` and `&Box` introduce an unnecessary layer of indirection as we have to dereference the `&String` to get a `String` and then perform a second dereference to end up at `&str`.

Instead, we should accept a *string slice* (`&str`), a *slice* (`&[T]`), or just a reference (`&T`). A `&String`, `&Vec<T>` or `&Box<T>` will be automatically coerced to a `&str`, `&[T]` or `&T`, respectively.

```
fn awesome_greeting(name: &str) {
    println!("Wow, you are awesome, {}!", name);
}
```

```
fn total_price(prices: &[i32]) -> i32 {
    prices.iter().sum()
}

fn is_even(value: &i32) -> bool {
    *value % 2 == 0
}
```

Now we can call these methods with a broader set of types. For example, `awesome_greeting` can be called with a string literal (`"Anna"`) *or* an allocated `String`. `total_price` can be called with a reference to an array (`&[1, 2, 3]`) *or* an allocated `Vec`.

If we'd like to add or remove items from the `String` or `Vec<T>`, we can take a *mutable reference* (`&mut String` or `&mut Vec<T>`):

```
fn add_greeting_target(greeting: &mut String) {
    greeting.push_str("world!");
}

fn add_candy_prices(prices: &mut Vec<i32>) {
    prices.push(5);
    prices.push(25);
}
```

Specifically for slices, we can also accept a `&mut [T]` or `&mut str`. This allows us to mutate a specific value inside the slice, but we can't change the number of items inside the slice (which means it's very restricted for strings):

```
fn reset_first_price(prices: &mut [i32]) {
    prices[0] = 0;
}

fn lowercase_first_ascii_character(s: &mut str) {
    if let Some(f) = s.get_mut(0..1) {
        f.make_ascii_lowercase();
    }
}
```

And that's why it's idiomatic to take a string slice, slice, or reference as a parameter rather than a String, Vec, or Box!

We hope you've enjoyed learning about various aspects of working with Rust and the ways it differs from other languages you may be used to working with. Learning about editions can give you confidence that your code will continue to compile even as Rust evolves. Test Driven Development, while a bit different in Rust, is just as useful a technique for exploring possible designs. Rust ensures that you don't have data races at compile time, which is not the case in languages like Ruby. Taking a string slice, slice, or reference as a parameter is idiomatic because that parameter can accept more types. And Rust's compile-time guarantees make your code memory safe without compromising on runtime performance. As you've seen in the examples in this sampler, working with Rust can feel a bit different than working with other languages. But we hope you agree that the benefits of programming in Rust make embracing these differences worthwhile, and that you're inspired to use Rust more!