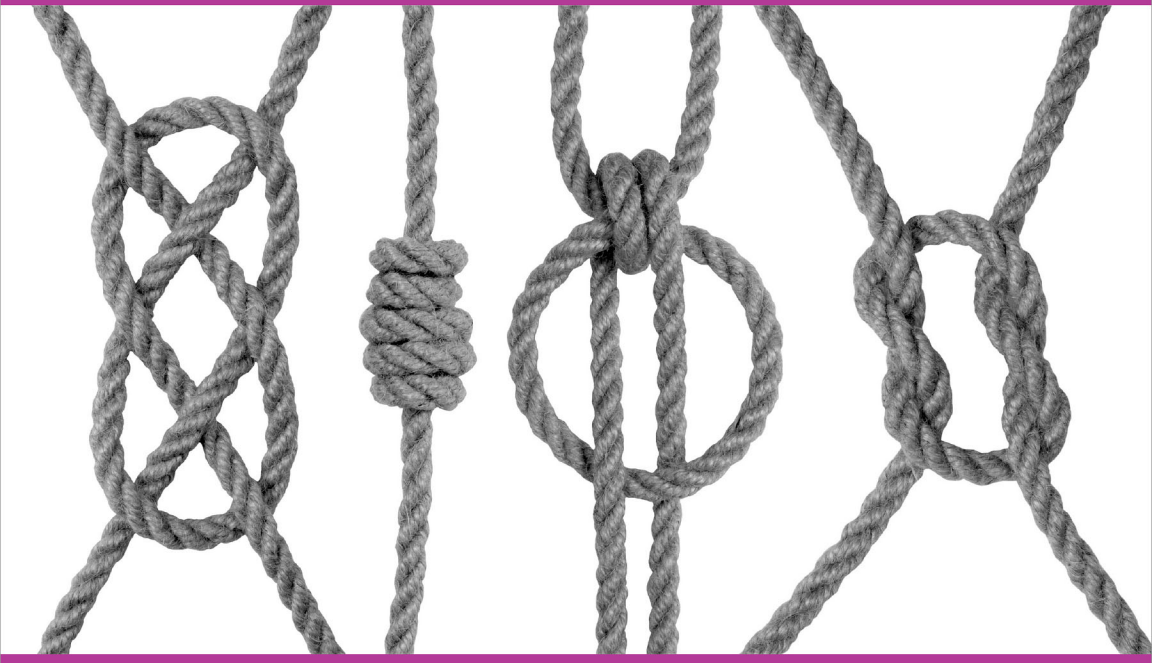
O'REILLY®

# Why Rust?

## Trustworthy, Concurrent Systems Programming



Jim Blandy

O'REILLY®

# Additional Resources

## 4 Easy Ways to Learn More and Stay Current

**Programming Newsletter**
Get programming related news and content delivered weekly to your inbox.
**oreilly.com/programming/newsletter**

**Free Webcast Series**
Learn about popular programming topics from experts live, online.
**webcasts.oreilly.com**

**O'Reilly Radar**
Read more insight and analysis about emerging technologies.
**radar.oreilly.com**

**Conferences**
Immerse yourself in learning at an upcoming O'Reilly conference.
**conferences.oreilly.com**

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. #15305

# Why Rust?

*Jim Blandy*

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

**Why Rust?**
by Jim Blandy

Copyright © 2015 O'Reilly Media. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Editors:** Meghan Blanchette and Rachel Roumeliotis
**Production Editor:** Melanie Yarbrough
**Copyeditor:** Charles Roumeliotis
**Proofreader:** Melanie Yarbrough
**Interior Designer:** David Futato
**Cover Designer:** Randy Comer
**Illustrator:** Rebecca Demarest

September 2015: First Edition

**Revision History for the First Edition**
2015-09-02: First Release
2015-09-014: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491927304* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Why Rust?,* the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-92730-4

[LSI]

# Table of Contents

# Why Rust?

Systems programming languages have come a long way in the 50 years since we started using high-level languages to write operating systems, but two thorny problems in particular have proven difficult to crack:

- It's difficult to write secure code. It's common for security exploits to leverage bugs in the way C and C++ programs handle memory, and it has been so at least since the Morris virus, the first Internet virus to be carefully analyzed, took advantage of a buffer overflow bug to propagate itself from one machine to the next in 1988.
- It's very difficult to write multithreaded code, which is the only way to exploit the abilities of modern machines. Each new generation of hardware brings us, instead of faster processors, more of them; now even midrange mobile devices have multiple cores. Taking advantage of this entails writing multithreaded code, but even experienced programmers approach that task with caution: concurrency introduces broad new classes of bugs, and can make ordinary bugs much harder to reproduce.

These are the problems Rust was made to address.

Rust is a new systems programming language designed by Mozilla. Like C and C++, Rust gives the developer fine control over the use of memory, and maintains a close relationship between the primitive operations of the language and those of the machines it runs on, helping developers anticipate their code's costs. Rust shares the ambitions Bjarne Stroustrup articulates for C++ in his paper "Abstraction and the C++ machine model":

> In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

To these Rust adds its own goals of memory safety and data-race-free concurrency.

The key to meeting all these promises is Rust's novel system of ownership, moves, and borrows, checked at compile time and carefully designed to complement Rust's flexible static type system. The ownership system establishes a clear lifetime for each value, making garbage collection unnecessary in the core language, and enabling sound but flexible interfaces for managing other sorts of resources like sockets and file handles.

These same ownership rules also form the foundation of Rust's trustworthy concurrency model. Most languages leave the relationship between a mutex and the data it's meant to protect to the comments; Rust can actually check at compile time that your code locks the mutex while it accesses the data. Most languages admonish you to be sure not to use a data structure yourself after you've sent it via a channel to another thread; Rust checks that you don't. Rust is able to prevent data races at compile time.

Mozilla and Samsung have been collaborating on an experimental new web browser engine named Servo, written in Rust. Servo's needs and Rust's goals are well matched: as programs whose primary use is handling untrusted data, browsers must be secure; and as the Web is the primary interactive medium of the modern Net, browsers must perform well. Servo takes advantage of Rust's sound concurrency support to exploit as much parallelism as its developers can find, without compromising its stability. As of this writing, Servo is roughly 100,000 lines of code, and Rust has adapted over time to meet the demands of development at this scale.

## Type Safety

But what do we mean by "type safety"? Safety sounds good, but what exactly are we being kept safe from?

Here's the definition of "undefined behavior" from the 1999 standard for the C programming language, known as "C99":

> 3.4.3
>
> **undefined behavior**

> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

Consider the following C program:

```
int main(int argc, char **argv) {
  unsigned long a[1];
  a[3] = 0x7ffff7b36cebUL;
  return 0;
}
```

According to C99, because this program accesses an element off the end of the array `a`, its behavior is undefined, meaning that it can do anything whatsoever. On my computer, this morning, running this program produced the output:

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

Then it crashes. I don't even have a `.netrc` file.

The machine code the C compiler generated for my `main` function happens to place the array `a` on the stack three words before the return address, so storing `0x7ffff7b36cebUL` in `a[3]` changes poor `main`'s return address to point into the midst of code in the C standard library that consults one's `.netrc` file for a password. When my `main` returns, execution resumes not in `main`'s caller, but at the machine code for these lines from the library:

```
warnx(_("Error: .netrc file is readable by others."));
warnx(_("Remove password or make file unreadable by others."));
            goto bad;
```

In allowing an array reference to affect the behavior of a subsequent `return` statement, my C compiler is fully standards-compliant. An "undefined" operation doesn't just produce an unspecified result: it is allowed to cause the program to do *anything at all*.

The C99 standard grants the compiler this carte blanche to allow it to generate faster code. Rather than making the compiler responsible for detecting and handling odd behavior like running off the end of an array, the standard makes the C programmer responsible for ensuring those conditions never arise in the first place.

Empirically speaking, we're not very good at that. The 1988 Morris virus had various ways to break into new machines, one of which entailed tricking a server into executing an elaboration on the tech-

nique shown above; the "undefined behavior" produced in that case was to download and run a copy of the virus. (Undefined behavior is often sufficiently predictable in practice to build effective security exploits from.) The same class of exploit remains in widespread use today. While a student at the University of Utah, researcher Peng Li modified C and C++ compilers to make the programs they translated report when they executed certain forms of undefined behavior. He found that nearly all programs do, including those from well-respected projects that hold their code to high standards.

In light of that example, let's define some terms. If a program has been written so that no possible execution can exhibit undefined behavior, we say that program is *well defined*. If a language's type system ensures that every program is well defined, we say that language is *type safe*.

C and C++ are not type safe: the program shown above has no type errors, yet exhibits undefined behavior. By contrast, Python is type safe. Python is willing to spend processor time to detect and handle out-of-range array indices in a friendlier fashion than C:

```
>>> a = [0]
>>> a[3] = 0x7ffff7b36ceb
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

Python raised an exception, which is not undefined behavior: the Python documentation specifies that the assignment to `a[3]` should raise an `IndexError` exception, as we saw. As a type-safe language, Python assigns a meaning to every operation, even if that meaning is just to raise an exception. Java, JavaScript, Ruby, and Haskell are also type safe: every program those languages will accept at all is well defined.

**NOTE**

Note that being type safe is mostly independent of whether a language checks types at compile time or at run time: C checks at compile time, and is not type safe; Python checks at runtime, and is type safe. Any practical type-safe language must do at least some checks (array bounds checks, for example) at runtime.

It is ironic that the dominant systems programming languages, C and C++, are not type safe, while most other popular languages are. Given that C and C++ are meant to be used to implement the foundations of a system, entrusted with implementing security boundaries and placed in contact with untrusted data, type safety would seem like an especially valuable quality for them to have.

This is the decades-old tension Rust aims to resolve: it is both type safe and a systems programming language. Rust is designed for implementing those fundamental system layers that require performance and fine-grained control over resources, yet still guarantees the basic level of predictability that type safety provides. We'll look at how Rust manages this unification in more detail in later parts of this report.

Type safety might seem like a modest promise, but it starts to look like a surprisingly good deal when we consider its consequences for multithreaded programming. Concurrency is notoriously difficult to use correctly in C and C++; developers usually turn to concurrency only when single-threaded code has proven unable to achieve the performance they need. But Rust's particular form of type safety guarantees that concurrent code is free of data races, catching any misuse of mutexes or other synchronization primitives at compile time, and permitting a much less adversarial stance towards exploiting parallelism. We'll discuss this more in the final section of the report.

**NOTE**

Rust does provide for *unsafe code*, functions or lexical blocks that the programmer has marked with the `unsafe` keyword, within which some of Rust's type rules are relaxed. In an unsafe block, you can use unrestricted pointers, treat blocks of raw memory as if they contained any type you like, call any C function you want, use inline assembly language, and so on.

Whereas in ordinary Rust code the compiler guarantees your program is well defined, in unsafe blocks it becomes the programmer's responsibility to avoid undefined behavior, as in C and C++. As long as the programmer succeeds at this, unsafe blocks don't affect the safety of the rest of the program. Rust's standard library uses unsafe blocks to implement features that are themselves safe to use, but which the compiler isn't able to recognize as such on its own.

The great majority of programs do not require unsafe code, and Rust programmers generally avoid it, since it must be reviewed with special care. The rest of this report covers only the safe portion of the language.

## Reading Rust

Before we get into the details of Rust's semantics, let's take a look at Rust's syntax and types. For the most part, Rust tries to avoid originality; much will be familiar, so we'll focus on what's unusual. The types are worth some close attention, since they're the key not only to Rust's performance and safety, but also to making the language palatable and expressive.

Here's a function that returns the greatest common divisor of two numbers:

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m; m = n; n = t;
        }
        m = m % n;
    }
    n
}
```

If you have experience with C, C++, Java, or JavaScript, you'll probably be able to fake your way through most of this. The interesting parts in brief:

- The `fn` keyword introduces a function definition. The `->` token after the argument list indicates the return type.
- Variables are immutable by default in Rust; the `mut` keyword marks our parameters `n` and `m` as mutable, so we can assign to them.
- In a variable or parameter declaration, the name being declared isn't nestled inside the syntax of the type, as it would be in C and C++. A Rust declaration has a name followed by a type, with a colon as a separator.
- A `u64` value is an unsigned 64-bit integer; `i32` is the type of 32-bit signed integers; and `f32` and `f64` are the usual floating-point types. Rust also has `isize` and `usize` types, which are 32-bit integers on 32-bit machines and 64-bit integers on 64-bit machines, in signed and unsigned varieties.
- The `!` in the use of `assert!` marks that as a macro invocation, rather than a function call. Rust has a flexible macro system that is carefully integrated into the language's grammar. (Unfortunately, we don't have space to do more than mention it in this report.)
- The type of a numeric literal like `0` is inferred from context; in our `gcd` function, those are `u64` zeros. You can specify a literal's type by providing a suffix: `1729i16` is a signed 16-bit integer. If neither inference nor suffix determines a literal's type, Rust assigns it the type `i32`.
- The `let` keyword introduces local variables. Rust infers types within functions, so there's no need for us to state a type for our temporary variable `t`: Rust infers that it must be `u64`.
- The conditions of `if` and `while` expressions need no parenthesis, but curly brackets are required around the expressions they control.
- Rust has a `return` statement, but we didn't need one to return our value here. In Rust, a block surrounded by curly braces can be an expression; its value is that of the last expression it contains. The body of our function is such a block, and its last

expression is `n`, so that's our return value. Likewise, `if` is an expression whose value is that of the branch that was taken. Rust has no need for a separate `?:` conditional operator as in C; one just writes the `if-else` structure right into the expression.

There's much more, but hopefully this covers enough of the syntax to get you oriented. Now let's look at a few of the more interesting aspects of Rust's type system: generics, enumerations, and traits.

## Generics

It is very common for functions in Rust to be *generic*—that is, to operate on an open-ended range of argument types, rather than just a fixed selection, much as a function template does in C++. For example, here is the `std::cmp::min` function from Rust's standard library, which returns the lesser of its two arguments. It can operate on integers of any size, strings, or really any type in which one value can be said to be less than another:

```
fn min<T: Ord>(a: T, b: T) -> T {
    if a <= b { a } else { b }
}
```

Here, the text `<T: Ord>` after the function's name marks it as a generic function: we're defining it not just for one specific type, but for any type `T`, which we'll use as the type of our arguments and return value. By writing `T : Ord`, we've said that not just any type will do: `T` must be a type that is `Ord`, meaning that it supports a comparison ordering all values of that type. If a type is `Ord`, we can use the `<=` operator on its values. `Ord` is an example of a *trait*, which we'll cover in detail below.

With this definition, we can apply `min` to values of any type we want, as long as the type orders its values:

```
min(10i8,  20)    == 10;    // T is i8
min(10,    20u32) == 10;    // T is u32
min("abc", "xyz") == "abc"; // strings are Ord, so this works
```

Since the definition uses `T` for both arguments, calls to `min` must pass two values of the same type:

```
min(10i32, "xyz");          // error: mismatched types.
```

The C++ analogue of our `min` function would be:

```
template<typename T>
T min(T a, T b) {
    return a <= b ? a : b;
}
```

However, the analogy isn't exact: where the Rust `min` stipulates that its argument type `T` must be `Ord`, the C++ function template says nothing about its requirements for `T`. In C++, for each call to `min`, the compiler must take the specific argument type at hand, substitute it for `T` in `min`'s definition, and see if the result is meaningful. Rust can check `min`'s definition in its own right, once, and can check a call to `min` using only the function's stated type: if the arguments have the same type, and that type is `Ord`, the call is well typed. This allows Rust to produce error messages that locate problems more precisely than those you can expect from a C++ compiler. Rust's design also forces programmers to state their requirements up front, which has its benefits and drawbacks.

One can have generic types as well as functions:

```
struct Range<Idx> {
    start: Idx,
    end: Idx,
}
```

This is the `std::ops::Range` type from Rust's standard library, which represents the value of range expressions like `0..10`; these appear in iterations, expressions denoting portions of arrays and strings, and so on. As in the definition of our generic function `min`, the text `<Idx>` after the name `Range` indicates that we're defining a structure that is generic in one type, `Idx`, which we use as the type of the structure's `start` and `end` fields.

Making `Range` generic allows us to handle all these expressions as `Range<T>` values for different types `T`:

```
-10i32..10          // a Range<i32>
-2.0..0.25f64       // a Range<f64>
200..800            // a Range<T>, for the integer type T
                    // determined from context
```

Rust has a more general expression syntax for writing instances of any struct type. For example, the last range above could also be written:

```
Range { start: 200, end: 800 }
```

Rust compiles generic functions by producing a copy of their code specialized for the exact types they're applied to, much as C++ generates specialized code for function template instantiations. As a result, generic functions are as performant as the same code written with specific types used in place of the type variables: the compiler can inline method calls, take advantage of other aspects of the type, and perform other optimizations that depend on the types.

## Enumerations

Rust's enumerated types are a departure from C and C++ `enum` types, but users of functional languages will recognize them as *algebraic datatypes*. A Rust enumerated type allows each variant to carry a distinct set of data values along with it. For example, the standard library provides an `Option` type, defined as follows:

```
enum Option<T> {
    None,
    Some(T)
}
```

This says that, for any type `T`, an `Option<T>` value may be either of two variants: `None`, which carries no value; or `Some(v)`, which carries the value `v` of type `T`. Enumerated types resemble unions in C and C++, but a Rust `enum` remembers which alternative is live, preventing you from writing to one variant of the enum and then reading another. C and C++ programmers usually accomplish the same purpose by pairing a union type with an `enum` type, calling the combination a "tagged union."

Since `Option` is a generic type, you can use it with any value type you want. For example, here's a function that returns the quotient of two numbers, but declines to divide by zero:

```
fn safe_div(n: i32, d: i32) -> Option<i32> {
    if d == 0 {
        return None;
    }
    return Some(n / d);
}
```

This function takes two arguments `n` and `d`, both of type `i32`, and returns an `Option<i32>`, which is either `None` or `Some(q)` for some signed 32-bit integer `q`. If the divisor is zero, `safe_div` returns `None`; otherwise it does the division and returns `Some(the quotient)`.

The only way to retrieve a value carried by an enumerated type is to check which variant it is, and handle each case, using a `match` statement. For example, we can call `safe_div` like this:

```
match safe_div(num, denom) {
    None => println!("No quotient."),
    Some(v) => println!("quotient is {}", v)
}
```

You can read the `match` here as something like a `switch` statement that checks which variant of `Option<T> safe_div` returned. The `Some` branch assigns the value the variant carries to the variable `v`, which is local to its branch of the `match` statement. (The `None` variant carries no values, so it doesn't set any local variables.)

In some cases a full-blown `match` statement is more than we need, so Rust offers several alternatives with varying ergonomics. The `if let` and `while let` statements use matching as the condition for branching or looping; and the `Option` type itself provides several convenience methods, which use `match` statements under the hood.

Rust's standard libraries make frequent use of enumerations, to great effect; we'll see two more real-world examples later in the section on memory safety.

## Traits

When we defined our generic `min` function above, we didn't simply define `min<T>(a: T, b: T) -> T`. One could read that as "the lesser of two values of any type `T`," but not every type is well-ordered. It's not meaningful to ask, say, which of two network sockets is the lesser. Instead we defined `min<T: Ord>(...)`, indicating that `min` only works on types whose values fall in some order relative to each other. Here, the constraint `Ord` is a *trait*: a collection of functionality that a type can implement.

The `Ord` trait itself is pretty involved, so let's look at a simpler (but quite useful) example: the standard library's `IntoIterator` and `Iterator` traits. Suppose we have a table of the names of the seasons in the United States' Pacific Northwest:

```
let seasons = vec!["Spring", "Summer", "Bleakness"];
```

This declares `seasons` to be a value of type `Vec<&str>`, a vector of references to statically allocated strings. Here's a loop that prints the contents of `seasons`:

```
for elt in seasons {
    println!("{}", elt);
}
```

Perhaps obviously, this prints:

```
Spring
Summer
Bleakness
```

Rust's `for` loop isn't limited to vectors: it can iterate over any type that meets a few key requirements. Rust captures those requirements as two traits:

- Types implementing the `Iterator` trait can produce a sequence of values for a `for` loop to iterate over, and decide when to exit the loop. `Iterator` values hold the loop state.
- Types implementing the `IntoIterator` trait have an `into_iter` method that returns an `Iterator` traversing them in whatever way is natural. To be permitted as the `E` in `for V in E { ... }`, a type must implement `IntoIterator`.

The standard library's container types like `Vec`, `HashMap`, and `LinkedList` all implement `IntoIterator` out of the box. But as an example, let's look at what it would take to implement iteration for our `Vec<&str>` type ourselves.

Here's the definition of the `Iterator` trait from the standard library:

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;

    fn size_hint(&self) -> (usize, Option<usize>) { ... }
    fn count(self) -> usize { ... }
    fn last(self) -> Option<Self::Item> { ... }
    fn nth(&mut self, n: usize) -> Option<Self::Item> { ... }
    // ... some thirty-odd other methods omitted ...
}
```

There's a lot there, but only the first two items actually concern us. This definition says that, in order to implement this trait, a type must provide at least two things:

- Its `Item` type: the type of value the iteration produces. When iterating over a vector, this would be the type of the vector's elements.
- A `next` method, which returns `Option<Item>`: either `Some(v)`, where `v` is the next value in the iteration, or `None` if we should exit the loop.

When defining methods, the `self` argument is special: it refers to the value on which we're invoking the method, like `this` in C++. The `Iterator` trait's `next` method takes a `&mut self` argument, meaning that it takes its self value by reference, and is allowed to modify it. A method can also take its self value by shared reference (`&self`), which does not permit modification, or by value (simply `self`).

Other than `next`, all the methods in `Iterator` have default definitions (shown as `{ ... }` above, omitting their code) which build on the `Item` and `next` definitions we provide, so we don't need to write them ourselves (although we could if we liked).

To implement `Iterator` for our vector of strings, we must first define a type to represent the current loop state: the vector we're iterating over, and the index of the element whose value we should produce in the next iteration:

```
struct StrVecIter {
    v: Vec<&'static str>,
    i: usize
}
```

The type `&'static str` is a reference to a string literal, like the names of the seasons in our example. (We'll cover lifetimes like `'static` in more detail later, but for now, take it to mean that our vectors hold only string literals, not dynamically allocated strings.)

Now that we have the `StrVecIter` type to hold our loop state, we can implement the `Iterator` trait for it:

```
impl Iterator for StrVecIter {

    type Item = &'static str;

    fn next(&mut self) -> Option<&'static str> {
        if self.i >= self.v.len() {
            return None;
```

```
            }
            self.i += 1;
            return Some(self.v[self.i - 1]);
        }
    }
```

We've provided an `Item` type: each iteration gets another `&'static str` value from the vector. And we've provided a `next` method, which produces either `Some(s)`, where `s` is the value for the next iteration, or `None`, indicating that we should exit the loop. This is all we need: all the other methods appearing in the `Iterator` trait definition will fall back to their default definitions.

With that in place, we can implement the `IntoIterator` trait. Here's the trait's definition, from the standard library:

```
trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item=Self::Item>;
    fn into_iter(self) -> Self::IntoIter;
}
```

This says that any type implementing `IntoIterator` must provide:

- A type `Item`, the type of the values produced for each iteration of the loop.
- A type `IntoIter`, which holds the loop state. This must implement `Iterator`, with the same `Item` type as our own.
- A method `into_iter`, which produces a value of our `IntoIter` type.

Here's how we could implement `IntoIterator` for our type `Vec<&str>`:

```
impl IntoIterator for Vec<&'static str> {
    type Item = &'static str;
    type IntoIter = StrVecIter;
    fn into_iter(self) -> StrVecIter {
        return StrVecIter { v: self, i: 0 };
    }
}
```

This defines the `into_iter` method for `Vec<&str>` to construct a value of the `StrVecIter` type we defined above, pointing to our vector and ready to start iteration at the first element; accordingly, `StrVecIter` is our `IntoIter` type. And finally, our `Item` type is `&str`: each iteration of the loop gets a string.

We could improve on this definition by passing it the vector by reference, not by value; as written, the `for` loop will move the vector into the `StrVecIter` value, meaning that it can no longer be used after we've iterated over it. We can fix this readily by having `StrVecIter` *borrow* the vector instead of taking it by value; we'll cover borrowed references later in the report.

Like functions and types, trait implementations can be generic. Rust's standard library uses a single implementation of `IntoIterator` to handle vectors of any type:

```
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> {
        ...
    }
}
```

Iterators are a great example of Rust's commitment to zero-cost abstractions. While Rust's `for` loop requires the type representing the loop state to implement the `Iterator` trait, this doesn't imply that any sort of virtual dispatch is taking place each time the `for` loop invokes the iterator's `next` method. As long as the compiler knows the exact type of the iterator value, it can inline the type's `next` method, and we'll get the same machine code we'd expect from a handwritten loop.

Implementing `Iterator` does more than just allow us to connect to `for` loops. The default method definitions on `Iterator` offer a nice collection of operations on sequences of values. For example, since ranges implement `IntoIterator`, here's a function that sums the integers in the range `1..n` using `Iterator`'s `fold` method:

```
fn triangle(n: i32) -> i32 {
    (0..n+1).fold(0, |sum, i| sum + i)
}
```

Here, the expression `|sum, i| sum + i` is a Rust *closure*: an anonymous function that takes two arguments, `sum` and `i`, and returns `sum + i`. We pass this closure as `fold`'s second argument; `fold` calls it for each value the iterator produces, passing the running total and the iterator's value as arguments. The closure's return value is taken as the new running total, which `fold` returns when the iteration is complete.

As with the `for` loop, this is a zero-cost abstraction: the `fold` method can be inlined into `triangle`, and the closure can be inlined into `fold`. The machine code generated for this definition is as good as that for the same loop written out by hand.

Traits usually appear in Rust code as bounds on type parameters, just as the trait `Ord` bounded the type variable `T` in our definition of `min` earlier. Since Rust compiles generic functions by specializing them to the actual types they're being applied to, the compiler always knows exactly which implementation of the bounding traits to use. It can inline method definitions, and in general optimize the code for the types at hand.

However, you can also use traits to refer to values whose specific type isn't determined until runtime. Here, Rust must use dynamic dispatch to find the traits' implementations, retrieving the relevant method definition from a table at runtime, much as C++ does when calling a virtual member function.

For example, the following function reads four bytes from an input stream `stream`, and compares them against a given sequence of bytes. One might use a function like this to check the "magic number" bytes at the beginning of a binary file:

```
use std::io::Read;
use std::io::Result;

fn check_magic(stream: &mut Read, magic: &[u8])
               -> Result<bool> {
    let mut buffer = [0; 4];
    if try!(stream.read(&mut buffer)) < 4 {
        return Ok(false);
    }
    return Ok(&buffer == magic);
}
```

The standard library defines `std::io::Read` as a trait with methods for reading from a stream of bytes, akin to `std::istream` in C++. This trait's `read` method accepts a buffer, tries to fill it with bytes from the stream, and returns the number of bytes it transferred on success, or an error code on failure.

Our `stream` argument's type, `&mut Read`, is interesting: rather than being a mutable reference to some specific type, it is a mutable reference to a value of *any* type that implements `Read`. This sort of reference is called a *trait object*, and supports all the trait's methods and

operations. This allows us to use a reference to any value that implements `Read` as the first argument to `check_magic`.

**NOTE**

At runtime, Rust represents a trait object as a pair of pointers: one to the value itself, and the other to a table of implementations of the trait's methods for that value's type. Our call to `stream.read` consults this table to find the `read` implementation for `stream`'s true type, and calls that, passing along the trait object's pointer to the value as the `self` argument.

Trait objects allow data structures to hold values of mixed types, where the set of possible types is open-ended. For example, the following function takes a vector of values, and joins them all into a string, knowing nothing about their types other than that they implement the trait `ToString`:

```
fn join(v: &Vec<&ToString>, sep: char) -> String {
    let mut s = String::new();
    for i in 0..v.len() {
        s.push_str(&v[i].to_string());
        if i + 1 < v.len() {
            s.push(sep);
        }
    }
    s
}
```

We can pass this a vector containing an arbitrary mix of types:

```
assert_eq!(join(&vec![&0,
                      &std::net::Ipv4Addr::new(192,168,0,1),
                      &"trilobite"],
                ','),
           "0,192.168.0.1,trilobite");
```

When used in this way, traits are analogous to C++'s abstract base classes or Java's interfaces: they use dynamic dispatch to allow code to operate on values whose types can vary at runtime. But the analogy doesn't extend much further:

- When traits serve as bounds on type parameters to generic functions, there's no dynamic dispatch involved. This is the most common use of traits in Rust.
- Whereas a type's base classes and interfaces are fixed when it is defined, the set of traits a type implements is not. You can

define your own traits and implement them on types defined elsewhere; you can even implement your traits on primitive types like `i32`.

# Memory Safety in Rust

Now that we've sketched Rust's syntax and types, we're ready to look at the heart of the language, the foundation for Rust's claims to memory safety and trustworthy concurrency. We'll focus on three key promises Rust makes about every program that passes its compile-time checks:

- *No null pointer dereferences.* Your program will not crash because you tried to dereference a null pointer.
- *No dangling pointers.* Every value will live as long as it must. Your program will never use a heap-allocated value after it has been freed.
- *No buffer overruns.* Your program will never access elements beyond the end or before the start of an array.

Rust uses a different technique to back up each of these promises, and each technique has its own impact on the way you write programs. Let's take a look at each one in turn.

## No Null Pointer Dereferences

The simplest way to ensure that one never dereferences a null pointer is to never permit one to be created in the first place; this is the approach Rust takes. There is no equivalent in Rust to Java's `null`, C++'s `nullptr`, or C's `NULL`. Rust does not convert integers to pointers, so one can't just use `0`, as in C++. The default value of a pointer is irrelevant: Rust requires you to initialize each variable before using it. And so on: the language simply doesn't provide a way to introduce null pointers.

But all those other languages include explicit support for null pointers for a good reason: they're extremely useful. A lookup function can return a null pointer to indicate that the requested value was not found. A null pointer can indicate the absence of optional information. Functions that always return a pointer to an object under normal circumstances can return null to indicate failure.

The problem with null pointers is that it's easy to forget to check for them. And since null pointers are often used to signal rare conditions like running out of memory, missing checks can go unnoticed for a long time, leaving us with the worst possible combination: checks that are easy to forget, used in a way that makes the omission usually asymptomatic. If we could ensure that our programs never neglect to check for null pointers, then we could have our optional values and our error results, without the crashes.

Rust's solution is to separate the concept of a pointer (a kind of value that refers to some other value in memory) from the concept of an optional value, handling the latter using the standard library's `Option` enumerated type, presented earlier. When we need an optional argument, return value, or so on of some pointer type `P`, we use `Option<P>`. A value of this type is not a pointer; trying to dereference it or call methods on it is a type error. Only if we check which variant we have in hand using a `match` statement and find `Some(p)` can we extract the actual pointer `p`—which is now guaranteed not to be null, simply because pointers are never null.

The punchline is that, under the hood, this all turns back into null pointers in the compiled code. The Rust language permits the compiler to choose any representation it likes for `enum` types like `Option`. In the case of `Option<P>`, the compiler chooses to represent `None` as a zero value, and `Some(p)` as the pointer value `p`, since the one can't be mistaken for the other. Thus, after the compiler has ensured that all the necessary checks are present in the source code, it translates it to the same representation at the machine level that C++ would use for nullable pointers.

For example, here's a definition of a singly linked list whose nodes carry a value of some type `T`:

```
struct Node<T> {
    value: T,
    next: Option<Box<Node<T>>>
}

type List<T> = Option<Box<Node<T>>>;
```

The `Box` type is Rust's simplest form of heap allocation: a `Box<T>` is an owning pointer to a heap-allocated value of type `T`. When the box is dropped, the value in the heap it refers to is freed along with it. So `Option<Box<Node<T>>>` is either `None`, indicating the end of the list, or `Some(b)` where `b` is a box pointing to the next node.

Once we've disentangled the concept of a pointer from the concept of an optional value, we begin to notice other common situations in C and C++ interfaces where special sentinel values that mark errors or other corner conditions might be mistaken for legitimate values. For example, many POSIX system calls return `-1` to indicate that an error occurred, and return nonnegative integers on success. One might discover C code like this:

```
ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
process_bytes(buffer, bytes_read);
```

If the two arguments to `process_bytes` are the address of a buffer and the length of the data it holds in bytes, then we have a bug here: the `read` system call returns `-1` when an error occurs, and the programmer forgot to check for that case. This code will blithley pass the `-1` along to `process_bytes` as a nonsensical length. Just as a null pointer is a legitimate pointer value in C, our error marker `-1` is a legitimate integer value, so nothing in the language requires the programmer to check the result from `read` before treating it as a buffer length.

In the Rust standard library, functions that can fail, such as those that do input or output, return a value of the following type, declared in the `std::result` module:

```
#[must_use]
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

This definition looks very similar to `Option`, but here we have two type parameters: `T`, representing whatever sort of value the opera-

tion returns on success; and `E`, representing an error type. The `std::io` module defines the following type alias for its own use:

```
type Result<T> = std::result::Result<T, Error>;
```

where `Error` is `std::io::Error`, the module's own error type. Thus, a `std::io::Result<T>` value carries either a successful result value of type `T`, or a `std::io::Error` error code.

Here, then, is the signature of the Rust library method for reading bytes from a stream, our analogue to the POSIX `read` system call:

```
fn read(&mut self, buf: &mut [u8]) -> std::io::Result<usize>;
```

Here the `self` argument is the stream we're reading bytes from, and `buf` is a *slice* of some writable byte array, carrying its start address and size in a single parameter. But what we're interested in is the type of the return value: `std::io::Result<usize>`, meaning that we either get a value `Ok(size)`, indicating that we successfully read `size` bytes, or a value `Err(error)`, indicating that an error occurred, as described by the `std::io::Error` value `error`.

As with `Option`, there is no way to retrieve the byte count that `read` may have returned without first checking that we indeed have an `Ok` result, rather than an `Err`. The `read` method's return type is distinct from its successful completion type, forcing the developer to check for errors.

All fallible operations in Rust's `std::io` library use `std::io::Result` in this manner. And as with `Option`, Rust provides more ergonomic ways to handle `Result` values than a `match` statement. If one must call a function that returns a `Result` from within a function that itself returns `Result`, the `try!` macro is very convenient:

```
// Print the contents of the directory `dir`, and return
// the number of entries or a std::io::Error.
fn list_directory(dir: &Path) -> std::io::Result<usize> {
    let mut count = 0;
    let entries = try!(std::fs::read_dir(dir));

    for entry_or_error in entries {
        let entry = try!(entry_or_error);
        println!("{:?}", entry.path());
        count += 1;
    }
```

```
    return Ok(count);
}
```

Here, the call to `std::fs::read_dir` may fail (for example, if the directory doesn't exist), so we use the `try!` macro to check its result. If it returns some `Err(e)`, then `try!` returns immediately from our `list_directory` function, providing that `Err` as the value. Otherwise, the result must be `Ok(v)`, and `v` becomes the value of the `try!` expression—in our case, an iterator over the directory's entries.

Since errors may also occur in the process of reading individual entries, the iterator yields not `DirEntry` values (the type of a directory entry), but `std::io::Result<DirEntry>` values. We check each result in the loop with a second `try!` expression, and print and count the result if all is well.

You may have noticed the `#[must_use]` annotation on the definition of `std::result::Result`; that directs the compiler to produce a warning if a program leaves a return value of this type unused. Although it is easy enough to subvert, the warning does help catch error checks that have been accidentally omitted when calling functions that return no interesting value.

## No Dangling Pointers

Rust programs never try to access a heap-allocated value after it has been freed. This is not an unusual promise; any practical type-safe language must ensure this. What is unusual is that Rust does so without resorting to garbage collection or reference counting.

The Rust design FAQ explains:

> A language that requires a garbage collector is a language that opts into a larger, more complex runtime than Rust cares for. Rust is usable on bare metal with no extra runtime. Additionally, garbage collection is frequently a source of non-deterministic behavior.

Instead of garbage collection, Rust has three rules that specify when each value is freed, and ensure all pointers to it are gone by that point. Rust enforces these rules entirely at compile time; at runtime, your program uses plain old pointers—dumb addresses in memory—just like pointers and references in C and C++, or references in Java.

The three rules are as follows:

- *Rule 1: Every value has a single owner at any given time.* You can move a value from one owner to another, but when a value's owner goes away, the value is freed along with it.
- *Rule 2: You can borrow a reference to a value, so long as the reference doesn't outlive the value (or equivalently, its owner).* Borrowed references are temporary pointers; they allow you to operate on values you don't own.
- *Rule 3: You can only modify a value when you have exclusive access to it.*

We'll look at each of these in turn, and explore their consequences.

*Rule 1: Every value has a single owner at any given time.* Variables own their values, as do fields of structures and enums, and elements of arrays and tuples. Every heap-allocated value has a single pointer that owns it; when its owning pointer is dropped, the value is dropped along with it. Values can be moved from one owner to another, with the source relinquishing ownership to the destination.

As an example, Rust's `String` type is a growable array of Unicode characters (encoded using UTF-8). A `String` stores its text on the heap; it is the owner of that heap-allocated memory. So suppose we create a `String` value by copying a statically allocated string literal, and store that in a variable:

```
{
    let s = "Chez Rutabaga".to_string();
} // s goes out of scope here; text is freed
```

Here, `s` owns the `String`, and the `String` owns a heap-allocated buffer holding the text `"Chez Rutabaga"`. When `s` goes out of scope, the `String` will be dropped, and its heap-allocated buffer will be dropped along with it.

Suppose we add some code:

```
{
    let s = "Chez Rutabaga".to_string();
    let t1 = s;
    let t2 = s;
}
```

What should this do?

If this were C++ code using `std::string`, each assignment would create a fresh copy of the string. This is simple: each of the resulting

three strings is entirely independent of the others. But what if the string were large? A simple assignment could do unbounded amounts of work, and allocate unbounded amounts of memory. A systems language should make costs apparent; in the end, the Rust designers felt that implicit copying wasn't the right approach.

If this were Python or Java code, those languages handle their strings by reference, so this would result in all three variables, `s`, `t1`, and `t2`, pointing to the same string object. This approach makes the assignments efficient, because we only copy pointers, and no new allocation is necessary. But now, how do we decide when to free this shared string? We either need reference counting or garbage collection, neither of which make sense for a primitive type in a systems programming language. (C++ used to permit reference-counted string implementations, but the 2011 version of the C++ spec forbids them.)

Midway between those two strategies would be to copy only the `String` structure itself byte-for-byte. But this would result in three `String` values all pointing to the same buffer holding the text, leaving us in a situation similar to the Python/Java case. It's not clear who owns the buffer, nor how to manage its lifetime without complex machinery.

Rust approaches this situation starting with an observation from the C++ community about what a programmer really intends when she writes an assignment. Sometimes she really needs a copy. But often, the source of the assigment isn't going to be used anymore anyway, so *moving* the value, leaving the source unusable, is good enough. So in Rust, for most types (including any type that owns resources like a heap-allocated buffer), assignment *moves* the value: the destination takes ownership, and the source is no longer considered initialized. Hence:

```
let t1 = s;     // `t1` takes ownership from `s`
let t2 = s;     // compile-time error: use of moved value: `s`
```

Passing arguments to a function and returning values from a function are handled like assignment: they also move such types, rather than copying. Moves leave the source unusable even if they're only possible, not certain:

```
let x = "might be used by f".to_string();
if flip_coin() {
    f(x);       // value of `x` moved to `f`
```

```
}
x;              // compile-time error: use of moved value: `x`
```

For simple types like primitive integers and characters, however, there's no meaningful difference between a move and a copy. Consider code like this:

```
let pi = 3.1415926f32;
let one_eighty = pi;
let circumference_to_diameter = pi;
```

We're assigning `pi` twice, but there's no good reason this shouldn't work; the concerns about ownership we raised above don't apply to simple types like `f64`.

So, Rust divides all types into one of two kinds:

- Some types can be copied bit-for-bit, without the need for any special treatment; Rust permits such types to implement the special trait `Copy`. Assignment copies `Copy` types, rather than moving them: the source of the assignment retains its value. The primitive numeric types are `Copy`, as are `char`, `bool`, and a number of others.
- All other types are moved by assignment, never implicitly copied.

Rust doesn't implicly attach the `Copy` trait to newly defined types. You must implement it explicitly, assuming you feel copy semantics are appropriate for your type. Doing so is easy, since the `Copy` trait has no methods:

```
impl Copy for MyCopyableType { }
```

However, Rust permits `Copy` implementations only for types that qualify: all the values your type comprises must be `Copy` themselves, and your type must not require custom behavior when it is dropped (it mustn't have a "destructor," in C++ terms).

The Rust standard library includes a related trait, `Clone`, for types that can be copied explicitly. Its definition is simple:

```
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

The `clone` method must return an independent copy of `self`. The `clone_from` method does the reverse: it changes `self` into a clone of its `source` argument. That is nothing we couldn't do simply with a call to `clone` and an assignment; in fact, this is the default definition. However, some types can implement `clone_from` more efficiently, perhaps by reusing resources the destination already owns.

Most of the standard library types implement `Clone` where possible. Cloning a vector entails cloning each of its elements in turn, so `Vec<T>` implements `Clone` whenever `T` does so; the other container types behave similarly. `String` implements `Clone`, so we could make our original code work as follows:

```
{
    let s = "Chez Rutabaga".to_string();
    let t1 = s.clone();
    let t2 = s.clone();
}
```

Now each of the three variables holds its own independent copy of the text.

Obviously, any type that implements `Copy` can trivially implement `Clone` as well:

```
impl Clone for Color {
    fn clone(&self) -> Self { *self }
}
```

Since `MyCopyableType` is `Copy`, simply returning it from the `clone` method by value performs the copy required. So, for consistency, Rust requires a type to implement `Clone` if it implements `Copy`.

Since these traits are both ubiquitous and tedious to implement, the Rust compiler will write them for you, if you ask. The following defines a type `Color`, which is copied by assignment, not moved:

```
#[derive(Copy, Clone)]
struct Color { r: u8, g: u8, b: u8 }
```

The `#[derive(Copy, Clone)]` attribute asks the compiler to automatically implement the `Copy` and `Clone` traits for `Color`, guided by the structure of the type.

Where does all this leave us? We have a model for assignment that respects the single-owner rule, and avoids implicit copies that might have surprising costs, while allowing implicit copies where they make sense. And the single-owner rule brings us big benefits: every

value is guaranteed to be dropped, at a well-defined point in the code, without our having to write a line to make it happen.

You can take advantage of the fact that Rust drops values at well-defined times to design interfaces for resource management that are both trustworthy and easy to use. The Rust standard library uses this technique to ensure that files and sockets are always closed, mutexes are released, threads complete their tasks, and so on. This design pattern resembles C++'s "Resource Acquisition Is Initialization" idiom, but Rust's careful integration of moves into the language mean that ownership need not be fixed to a specific scope: the resource-owning values can be passed around between functions and threads, stored in data structures, and so on.

However, the single-owner rule is rather restrictive. Suppose we want to write a function that adds an exclamation point to the end of a `String`:

```
fn add_oomph(mut t: String) {
    t.push('!');
}
```

We might try to use it like this:

```
let mut s = "Hello, world".to_string();
add_oomph(s);
println!("{}", s);
```

Of course, this doesn't compile. Passing parameters to functions moves non-`Copy` arguments, just as assignment does, so the call to `add_oomph` moves ownership of the string to the function, leading the Rust compiler to reject the calling code with this error message:

```
error: use of moved value: `s`
println!("{}", s);
                ^
note: `s` moved here because it has type
`collections::string::String`, which is non-copyable
add_oomph(s);
          ^
```

The definition of `add_oomph` is hopeless anyway: `t` receives ownership of the string, we add the exclamation point, and then `t` goes out of scope, so the `String` we just modified gets dropped.

We could fix all this by having `add_oomph` return the string to its caller after modifying it:

```
fn add_oomph(mut t: String) -> String {
    t.push('!');
    t
}

let mut s = "Hello, world".to_string();
s = add_oomph(s);
println!("{}", s);
```

This works, but it's circuitous: passing `s` to `add_oomph` moves the string to `t`. We append the exclamation point there, and then returning `t` moves the string to the caller, which stores it back in `s`.

The escape from this comedy of errors is our second rule.

*Rule 2: You can borrow a reference to a value, so long as the reference doesn't outlive the value (or equivalently, its owner).* Although each value has only a single owner, you can *borrow references* to a value temporarily. A reference is a pointer that does not own its referent. Rust restricts the use of references to ensure that they all disappear before the value they refer to is dropped or moved, so references are never dangling pointers.

Under this rule, we can make `add_oomph` *borrow the string mutably*, rather than taking ownership of it. Instead of passing `s` directly, we pass a *mutable reference*, written `&mut`, to the string. We must change both the type of the function parameter, and the way we pass it:

```
fn add_oomph(t: &mut String) {
    t.push('!');
}

let mut s = "Hello, world".to_string();
add_oomph(&mut s);
println!("{}", s);
```

This time the program works as we had originally intended: `add_oomph` receives a mutable reference to the string, adds an exclamation point to it, and returns. When `t` goes out of scope, the borrow has ended. Ownership of the string remains with `s` the entire time—`add_oomph` only borrowed it—so the modified value is still there when we reach the call to `println!`. We print the string `"Hello, world!"`, exclamation point included, and then when `s` goes out of scope, we drop the string.

When we don't need to modify a value, we can *borrow a shared reference to it*, with a simple `&` instead of `&mut`:

```
fn print_in_quotes(t: &String) {
    println!("'{}'", t);
}

let mut s = "Hello, world".to_string();
add_oomph(&mut s);
print_in_quotes(&s);
```

Here, `print_in_quotes` takes a shared reference to our string, which is sufficient for its needs. This program prints the text:

```
'Hello, world!'
```

While Rule 1 explains values' lifetimes, Rule 2 constrains borrowed references from outliving their referents. When we take a reference to a value, Rust looks at how we use it (Is it passed to function? Stored in a local variable? Saved in a data structure?) and assesses how long it could live. (Since these checks are all done at compile time, Rust can't always be exact, and must sometimes err on the side of caution.) Then, it compares the lifetime of the value with the lifetime of the reference: if the reference could possibly outlive the value it points to, Rust complains, and refuses to compile the program.

Some simple examples:

```
let x = String::new();
let borrow = &x;
let y = x;      // error: cannot move out of `x` because
                // it is borrowed
```

While a value is borrowed, it musn't be moved to a new owner: the reference would still point at the original owner, which the move leaves uninitialized. Similarly, a variable must not go out of scope while it's borrowed:

```
let borrow;
let x = String::new();
borrow = &x;        // error: `x` does not live long enough
```

Since `x` is declared after `borrow`, it has a shorter lifetime; it's as if we had written:

```
{
    let borrow;
    {
        let x = String::new();
        borrow = &x;    // error: `x` does not live long enough
    }
}
```

This is equivalent, but makes it clearer that `borrow` outlasts its referent, which is forbidden. Reversing the two, however, is fine:

```
let x = String::new();
let borrow = &x;
```

Now the lifetime of `x` encloses the lifetime of `borrow`, so all is well.

Rust checks borrows of values in data structures, too. Here's an example using the `vec!` macro to construct a fresh instance of Rust's `Vec` type, analogous to a C++ `std::vector` or a Python list. Suppose we borrow a reference to a vector's element:

```
// create a new vector with one element
let mut v = vec!["hemlock"];

let borrow = &v[0];     // borrow that first element
v = vec!["wormwood"];   // error: cannot assign to `v`
                        // because it is borrowed
```

When we assign the second vector to `v`, that causes its prior value to be dropped—but `borrow` is holding a reference to that first vector's element. If Rust were to let the assignment proceed, `borrow` would become a dangling pointer, referring to the freed storage of the first vector.

But that error message is surprising: it says that all assignment to `v` is forbidden for as long as the borrowed reference to one of its elements exists. Which brings us to the next rule.

*Rule 3: You can only modify a value when you have exclusive access to it.* In other words:

- While you borrow a shared reference to a value, nothing can modify it or cause it to be dropped.
- While you borrow a mutable reference to a value, that reference is the only way to access that value at all.

Our hemlock/wormwood example above breaks the "shared reference" clause: we tried to modify `v` while (a portion of) it was borrowed. The borrow means that both `v[0]` and `*borrow` refer to the same value: the vector's first element. Since `v` does not have exclusive access to the value, we can't modify it.

We could fix our code by restricting the lifetime of the borrow:

```
let mut v = vec!["hemlock"];
{
    let borrow = &v[0];
    ... // do things with borrow here
}
v = vec!["wormwood"]; // no error; borrow is gone by this point
```

As the term "shared reference" suggests, it's fine to borrow as many shared references to a value as you like:

```
fn sum_refs(a: &i32, b: &i32) -> i32 {
    *a + *b
}

let x: i32 = 128;
assert_eq!(sum_refs(&x, &x), 256);
```

But borrowing a mutable reference forbids any other access to the value, whether through direct use of the variable:

```
let mut x = 128;
let b1 = &mut x;
x;      // error: cannot use `x`
        // because it was mutably borrowed
x += 1; // error: cannot assign to `x` because it is borrowed
```

or though some other reference:

```
let mut x = 128;
let b1 = &x;
let b2 = &mut x;    // error: cannot borrow `x` as mutable
                    // because it is also borrowed as shareable
```

Note that these exclusion rules may apply to an entire data structure, not just individual values. Suppose I have a vector of vectors of characters:

```
let mut v = Vec::new();
v.push(vec![' ', 'o', 'x']);
v.push(vec![' ', 'x', 'x']);
v.push(vec!['o', ' ', ' ']);
```

Borrowing an element of any of these four vectors causes Rust, in an overabundance of caution, to freeze the entire structure:

```
// multiple shared borrows are fine, as before
let borrow = &v[2][2];
let borrow2 = &v[0][0];

// borrow is shared, so reads are fine
assert_eq!(v[1][0], ' ');
v[1][0] = 'o';  // error: cannot borrow `v` as mutable because
                // it is also borrowed as shared
```

The error message refers to a mutable borrow, rather than the assignment we wrote, because Rust treats the expression `v[1][0]` here as a chain of mutable borrows: first `v`, then `v[1]`, and finally `v[1][0]`. Rust's analysis isn't detailed enough to realize that we won't actually use that mutable borrow to interfere with our other shared borrows from the row vectors, so it forbids the mutable borrow from taking place at all.

Rust's borrow checking has improved over time, and will continue to do so. But any analysis carried out at compile time must inevitably make conservative approximations: it would be impossible to accept all well-defined programs without inadvertently permitting some ill-defined programs. While many developers find themselves "fighting with the borrow checker" while learning Rust, over time they generally develop an intuition for how to work within its constraints.

So that's what Rule 3 means. The motivation for it may be hard to see, so let's look at two examples of the problems it avoids. A direct demonstration involves enumerations:

```
let mut opt = Some("cogito".to_string());
match opt {
    Some(ref r) => {
        opt = None;
        // What does r refer to now?
    },
    None => ()
}
```

We create an `Option<String>` that is a `Some` holding the `String` `"cogito"`. Then we match on that option. The pattern `ref r` means that the variable `r` should be assigned a borrowed reference to the value the `Some` is carrying, not the value itself. In this case, `r` becomes a reference to the field of `opt` holding the string `"cogito"`. Then, while this reference still exists, we change `opt` to `None`. What happens to the string we're borrowing? In general, what happens to the values carried by one variant of an enumeration when we replace it with another variant? The memory set aside for the enumeration has been reinitialized to a new collection of values; we can't have references to the old values hanging around.

The Rust compiler agrees:

```
error: cannot assign to `opt` because it is borrowed
    opt = None;
```

```
    ^~~~~~~~~~
note: borrow of `opt` occurs here:
    Some(ref r) => {
         ^~~~
```

But there's really something more general going on here. The code responsible for modifying some data structure must respect the structure's own rules, but it can't reasonably accomodate arbitrary references pointing into the midst of the structure while it does its work.

Other languages acknowledge this in different ways. Regarding modifying Java `Hashtable` objects while iterating through them, the documentation for the class says:

> [I]f the Hashtable is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException.

Similarly, the C++ Standard Template Library documentation for "unordered associative containers" (known to the rest of us as "hash tables") says that operations that might rehash the table invalidate all iterators into the table, and dereferencing an invalidated iterator is undefined behavior. (There's that "undefined behavior" again; break this rule, and your program is allowed to do anything at all.)

There's a wonderful example due to Aaron Turon of this problem in action. Consider the following function, which appends elements to the end of a vector:

```
fn append<T: Clone>(to: &mut Vec<T>, from: &[T]) {
    for elt in from {
        to.push(elt.clone());
    }
}
```

The syntax `&[T]` is new: this is a *slice*, a borrowed reference to a range of elements in some array or vector. Slices can be mutable, allowing you to modify their elements; a mutable slice is written `&mut [T]`. Concretely, a slice is simply a pointer to the first element in the range it refers to, together with a length. Like other forms of borrowed references, slices don't own the elements they refer to, and all the restrictions on ordinary borrowed references apply.

Our `append` function takes a slice of elements borrowed from somewhere, iterates over its elements, and pushes a copy of each one onto the end of a given `Vec`. This works fine:

```
let v1 = vec![5, 13, 17, 29, 37, 41, 53];
let mut v2 = vec![];

append(&mut v2, &v1);   // v2 was empty,
                        // so becomes a copy of v1
assert_eq!(v1, v2);
```

(The expression `&v1` evaluates to a slice referring to all of `v1`'s elements.)

But what if we try to append a vector to itself?

```
let mut v3 = vec![0, 1, 0, -1];
append(&mut v3, &v3);
```

The `for` loop in our `append_vector` function will iterate through the elements of the slice, pushing each one onto the end of `v3`. But when a growable vector type like `Vec` runs out of space for new elements, it must allocate a larger buffer, move the extant elements into it, and then free the original buffer. When this occurs, any slices referring to the original buffer become dangling pointers.

This is exactly what can happen in our second case: if a call to `push` causes `v3` to relocate its elements to a larger buffer, the slice we're iterating over becomes a pointer into freed memory: disaster.

The Rust compiler agrees:

```
error: cannot borrow `v3` as immutable because it is also
borrowed as mutable
    append(&mut v3, &v3);
                    ^~
note: previous borrow of `v3` occurs here; the mutable
borrow prevents subsequent moves, borrows, or modification
of `v3` until the borrow ends
    append(&mut v3, &v3);
                ^~
```

Now that we have the exclusion rules in hand, let's look at an interesting case. Returning a reference to one of our own local variables would clearly earn us a "does not live long enough" error, but if a function were to return a reference to one of its arguments, or some part of one of its arguments, then that ought to be OK. Indeed, the following is just fine:

```
fn first(v: &Vec<i32>) -> &i32 {
    return &v[0];
}
```

Rust reasons that, since the caller was apparently able to pass in a reference to the vector, the vector must be alive for the duration of the call: the borrow prevents anyone from moving, freeing, or modifying it. So, if we return a reference into the vector, that must be all right.

Our `first` function might be used like this:

```
fn call_first() {
    let v = vec![42, 1729];
    let borrow = first(&v);
    assert_eq!(*borrow, 42);
}
```

Rust checks borrows in each function separately, using only the types of the functions it calls, without looking into their definitions. So all we know here is that `first` takes a reference to a vector, and returns a reference to an `i32`. Rust assumes that, if a function returns a reference, that reference must be to (some part of) something we passed it, and thus should be treated like a borrowed reference to one of the arguments.

In simple cases like `first`, it's obvious which argument is being borrowed from, since there's only one: `v`. In more complicated cases, it's not so clear. Suppose we had a double-barreled version of `first` that returns a pair of references to the first elements of each of the two vectors it is passed:

```
fn firsts(x: &Vec<i32>,
          y: &Vec<i32>)
          ->
          (&i32, &i32) {
    return (&x[0], &y[0]);
}
```

Here, we have two borrowed references being passed in, and two being returned. It's obvious from the definition which borrows from which, but remember that callers of this function consider only its type when checking borrows, without looking into its definition. So callers work with only this:

```
fn firsts(x: &Vec<i32>, y: &Vec<i32>) -> (&i32, &i32);
```

With this view, we can still presume that the returned references are borrowed from the arguments—but which from which? Rust rejects the definition as ambiguous.

In definitions like this, Rust requires that we place *explicit lifetimes* on the references to spell out the relationships. Whereas a reference type with an inferred lifetime is written `&i32` or `&mut i32`, reference types with explicit lifetimes look like `&'a i32` or `&'x mut i32`: the `'a` or `'x` are the lifetime names, traditionally kept to a single character. Adding these annotations to our double-barreled `firsts` function gives us:

```
fn firsts<'a, 'b>(x: &'a Vec<i32>,
                  y: &'b Vec<i32>)
                  ->
                  (&'a i32, &'b i32) {
    return (&x[0], &y[0]);
}
```

We add `<'a, 'b>` after the function name, since the function is generic over any pair of lifetimes `'a` and `'b`. Then, we label each reference passed into the function with an explicit lifetime: `'a` represents the lifetime of the first vector, `'b` that of the second. Finally, we label the references we return: the first `&i32` must be borrowed from the first vector, the second `&i32` from the second vector. Now, when callers consider the type of `firsts`, they see:

```
fn firsts<'a, 'b>(x: &'a Vec<i32>,
                  y: &'b Vec<i32>)
                  ->
                  (&'a i32, &'b i32);
```

and they have all the information they need to ensure that the returned references don't outlive their referents.

References always have lifetimes associated with them; Rust simply lets us omit them when the situation is unambigous. We could have spelled out the lifetimes when defining our original `first` function, yielding:

```
fn first<'a>(v: &'a Vec<i32>) -> &'a i32 {
    return &v[0];
}
```

This second definition is identical to the original; we've simply made explicit what Rust assumed on our behalf.

Rust's ownership, moves, and borrows end up being a reasonably pleasant way to express resource management. The best way to see this is to walk through a more complex example. Here is a complete Rust program that counts the number of times each line appears in its input stream:

```
use std::collections::BTreeMap;
use std::io::BufRead;

fn main() {
    let mut counts = BTreeMap::new();
    let stdin = std::io::stdin();
    for line_or_error in stdin.lock().lines() {
        let line = line_or_error.unwrap();
        *counts.entry(line).or_insert(0) += 1;
    }

    for (line, count) in counts.iter() {
        println!("{} {}", count, line);
    }
}
```

This short program provides several examples of the ownership rules at work, so let's break it down.

```
let mut counts = BTreeMap::new();
```

The call to `BTreeMap::new()` constructs a fresh instance of `BTreeMap`, an ordered map type from Rust's standard library. We'll use this map to count the number of times we've seen each distinct line in our input stream, associating strings with counts. Note that, although Rust is a statically typed language, and this map has fixed key and value types, we don't need to write them out here; Rust is able to infer the map's type from the way it is used.

The constructor function returns the new map by value, not a pointer to the map. And since `BTreeMap` is a non-`Copy` type, that means that this initialization of `counts` is a move: `counts` takes ownership of the map, and will drop it when it goes out of scope.

```
let stdin = std::io::stdin();
```

This call constructs a handle referring to our standard input stream, and returns it by value. This is an instance of the `Stdin` type. `Stdin` is also non-`Copy`, so as before, our variable `stdin` becomes its owner.

```
for line_or_error in stdin.lock().lines() {
```

Here, `stdin.lock().lines()` evaluates to an `Iterator`, which reads from `stdin` and produces each line as a `std::io::Result<String>` value, carrying either a line of text read from the stream, or an error result if the read failed. This `Result` is also a non-`Copy` type, so on each iteration of the `for` loop, `line_or_error` takes ownership of the latest `Result`. (The expression `stdin.lock().lines()` does some interesting things with ownership itself, but for now, take that expression as a set phrase that ensures other threads won't steal data from the standard input stream while we loop over its lines.)

```
let line = line_or_error.unwrap();
```

The `unwrap` method of a `std::io::Result<String>` checks for errors, exiting the program if it finds any. On success, it returns the `Result`'s `Ok` value. The signature for the `unwrap` method of `std::result::Result<T, E>` is:

```
fn unwrap(self) -> T
```

Note that `self` isn't being passed by reference here: there's no `&` or `&mut` prefix on it. It's being passed by value, which means the call moves `line_or_error`'s value to `unwrap`: after this call, `line_or_error` is no longer initialized. When the result is an `Err`, this doesn't matter; the program exits anyway. But when the result is an `Ok(line)`, this is very important: since `unwrap` has taken ownership of the value, it is free to return the `Ok` variant's string by value. So if `unwrap` returns at all, the variable `line` takes ownership of the `String`.

Let's pause a moment to consider what's happening to the memory that actually holds the text of each line. As the iterator reads each line from our input stream, it allocates heap space to hold it. Ownership of that space moves from the iterator to `line_or_error`, then to `unwrap`'s `self`, and then to `line`. But no copies of the text itself have taken place; we've just moved the pointer to it around. And at every point, the text has an single owner, apparent from the text of the program. No reference counting, heap tracing, or other sophisticated techniques are needed to ensure it will be freed if an error occurs in the process.

Furthermore, since Rust supports inlining across crates (especially for generic functions like `unwrap`), optimization can reduce all this to machine code that simply accepts the `Result` from the iterator, checks whether it is `Ok` or `Err`, and then deals with the `String`. It's

the same code you would write in C++, except that the compiler can check that it's been done correctly.

```
*counts.entry(line).or_insert(0) += 1;
```

Here we look up the entry for `line` in our `BTreeMap`, and bump its count, providing zero as the initial value for new entries. The `BTreeMap::entry` method takes ownership of the key we pass; the method now owns the string. If `entry` needs to create a new entry in the map, we'll use this string as the key; no copy takes place. Otherwise, we already have an equivalent key in the table, and we drop the string we were passed. In either case, `entry` returns a `Entry` value pointing to the map's entry for our line, vacant or occupied. Calling `or_insert` ensures the entry is occupied, supplying an initial value as needed. It returns a mutable reference to the entry's value, which we increment. Our line has been counted.

Again, notice that we have managed our dynamic storage with no more machinery than we would have in idiomatic C++, but without the opportunity to make accidental copies, dangling references, or invalidated iterators. We pass around pointers to the text we're handling, carefully transferring ownership from one site to the next, until we have done the map lookup. At this point, we decide whether to retain the line string as a key in the map or drop its storage because the map already has another copy.

When we leave the first `for` loop, we drop the iterator. Our lock on `stdin` goes away, and other threads can read from it again.

```
for (line, count) in counts.iter() {
    println!("{} {}", count, line);
}
```

The iterator returned by `counts.iter()` produces *references* to the key and value. The loop body is just borrowing them; the map retains ownership at all times. The `println!` macro knows enough to automatically dereference its arguments, so we don't need to write `*count` and `*line`.

At the end of the function, we first drop `stdin`, and then `counts`. Dropping the `BTreeMap` frees its keys and values. Every line we read has its storage dropped either when we look it up in the map, or when we drop the map, so there are no leaks.

Rust's three rules combine to enforce a memory model with a combination of properties difficult to find elsewhere:

- Dangling pointers do not occur. Rust avoids other similar sorts of reference-invalidation errors as well.
- Rust frees resources automatically and predictably.
- Rust accomodates programming with a direct imperative style, with minimal reallocation and copying, and without garbage collection.

We'll see even more benefits of this system later when we discuss concurrency.

## No Buffer Overruns

Back in 1988, the Morris virus broke into Internet-connected computers by exploiting buffer overflows permitted by the `gets` library function. In 2015, security flaws caused by programs writing untrusted data beyond the ends of their own buffers remain common: according to the Open Source Vulnerability Database, buffer overruns have constituted a steady 10%-15% of all reported vulnerabilities over the last eight years.

Why are buffer overruns so common? In C and C++, you don't actually index arrays; you index pointers, which carry no information about the start and end of the array or object they point into. (Arrays are implicitly converted into pointers to their first element.) Bounds checking is left entirely up to the programmer, and as we've observed before, programmers often make minor mistakes; bounds checking code is no exception. At various times, people have modified C and C++ compilers to actually check for invalid memory references, using pointer representations that carry bounds along with the actual address, but these modifications have been dropped over time in favor of less precise tools.

In Rust, one does not index pointers. Instead, one indexes arrays and slices, both of which have definite bounds.

A Rust array type is written `[T; n]`, designating an array of `n` elements of type `T`. When a Rust program indexes an array `a` with an expression like `a[i]`, the program first checks that `i` falls within the array's size `n`. Sometimes the compiler recognizes that this check can be safely omitted, but when it can't, Rust generates code to check the array's index at runtime. If the index is out of bounds, the thread panics.

As explained earlier, a Rust slice is a borrowed pointer to a section of some other array that actually owns the elements. A slice is represented as a pointer to the first element included in the slice, together with a count of the number of elements it comprises. If `a` is an array, the expression `&a[i..j]` evaluates to a slice referring to the `i`th through `j-1`th elements of `a`. When we create a slice, we check that its start and end fall within the bounds of the array it borrows from; and when we index a slice, we check that the index falls within the slice.

So, for example:

```
fn fill(s: &mut[i32], n: i32) {
    for i in 0..s.len() {
        s[i] = n;
    }
}

let mut a = [6, 2, 7, 3, 1, 8, 5];
fill(&mut a[3..5], 0);
assert_eq!(a, [6, 2, 7, 0, 0, 8, 5]);
```

Here the function `fill` takes a mutable slice, and sets all its elements to `n`. Below that, we declare an array of type `[i32; 7]`, create a slice referring to its fourth and fifth elements, pass that slice to `fill` to be mutated, and then check that it now has the value one would expect.

This code performs bounds checks in two places. First, obviously, each assignment in `fill` to `s[i]` must check that `i` is a valid index for the slice `s`. Second, when we create the slice of `a` that we pass to `fill`, we check that the requested range actually falls within the array we're borrowing from. Naturally, in this toy example, the checks borrowing the slice can be optimized out, since the array's size and the slice's range are all known at compile time. Less obviously, in `fill`, the compiler may be able to recognize that `i` will always be less than `s.len()`, and thus omit the bounds checks there as well.

There is a better way to write `fill`, however. A `for` loop iterating over a mutable slice produces a mutable reference to each element of the slice in turn, allowing us to say:

```
fn fill(a: &mut[i32], n: i32) {
    for i in a {
        *i = n;
```

```
        }
    }
```

Here, there is no bounds check needed when we store `n`; `i` is already a mutable reference to the correct element. In producing that reference, it's the iterator itself that must index the slice; the bounds check should occur at that point. But notice that the iterator is also responsible for deciding when to exit the loop altogether; both the bounds check and the end-of-loop check compare the current index to the length of the slice. It seems a shame to do the same comparison twice!

So, internally, the iterator `iter_mut` checks whether there are more elements to handle, and if there are, uses unsafe code to produce the reference to the element that skips the bounds check. The end result is a loop that performs one comparison per iteration, just as you would write in C or C++. Again, Rust has provided a comfortable, safe abstraction at no cost.

Rust's standard library includes many forms of iterators, written with similar optimizations where appropriate, so that these off-the-shelf iterators are usually both more legible and faster than loops that index the array themselves. (Faster, that is, if the compiler isn't able to optimize out the check—which often it can.)

# Multithreaded Programming

Now that we've outlined Rust's rules regarding ownership and memory safety, we're finally able to present the crown jewel of Rust's design: concurrency without data races. In most languages, programmers try to squeeze as much performance as they can out of their single-threaded code, and turn to concurrency only when there's no alternative. In Rust, concurrency is much safer to use, making it a technique you can design into your code from the beginning: a method of first resort, not last resort. Rust also provides high-level abstractions like channels and worker thread pools to make concurrency convenient to use.

## Creating Threads

Before we can show off Rust's synchronization features, we need to create some threads. The Rust `std::thread::spawn` function takes a closure and runs it in a new thread. For example:

```
let thread1 = std::thread::spawn(|| {
    println!("Alphonse");
    return 137;
});
let thread2 = std::thread::spawn(|| {
    println!("Gaston");
    return 139;
});
assert_eq!(try!(thread1.join()), 137);
assert_eq!(try!(thread2.join()), 139);
```

This creates two threads, each of which prints a distinctive string and exits. The `std::thread::spawn` call returns a `JoinHandle`, a value whose `join` method waits for the thread to finish, and provides whatever value the thread returned. After starting up the threads, the main thread waits for each one to finish, and checks for the expected value. (If the main thread exits first, all other threads shut down immediately, so without the call to `join` we might not see any output from them at all.)

Since the two threads don't communicate at all, this program might print the two strings in either order. However, the `println!` macro locks the standard output stream while printing its text, so each thread's output will appear without being interleaved with any other's.

Since closures capture their environment, what happens if these two threads try to share a local variable?

```
let mut x = 1;
let thread1 = std::thread::spawn(|| { x += 8 });
let thread2 = std::thread::spawn(|| { x += 27 });
```

Rust forbids this, making the following complaint about each call:

```
error: closure may outlive the current function, but it
borrows `x`, which is owned by the current function
```

Since our closure uses `x` from the surrounding environment, Rust treats the closure as a data structure that has borrowed a mutable reference to `x`. The error message complains that Rust can't be sure that the function to which `x` belongs won't return while the threads are still running; if it did, the threads would be left writing to a popped stack frame.

Fair enough. But under such pessimistic rules, threads could never be permitted to access local variables. It's common for a function to want to use concurrency as an implementation detail, with all

threads finishing before the function returns, and in such a case the local variables are guaranteed to live long enough. If we promise to join our threads while `x` is still in scope, it seems like this isn't sufficient reason to reject the program.

And indeed, Rust offers a second function, `std::thread::scoped`, used very much like `spawn`, but willing to create a thread running a closure that touches local variables, in a manner that ensures safety. The `scoped` function has an interesting type, which we'll summarize as:

```
fn scoped<'a, F>(f: F) -> JoinGuard<'a>
  where F: 'a, ...
```

As with `spawn`, we expect a closure `f` as our sole argument. But instead of returning a `JoinHandle`, `scoped` returns a `JoinGuard`. Both types have `join` methods that return the result from the thread's closure, but they differ in their behavior when dropped: whereas a `JoinHandle` lets its thread run freely, dropping a `JoinGuard` blocks until its thread exits. A thread started by `scoped` never outlives its `JoinGuard`.

But now let's consider how the lifetimes here nest within each other:

- Dropping `JoinGuard` waits for the thread to return; the thread cannot outlive the `JoinGuard`.
- The `JoinGuard` that `scoped` returns takes lifetime `'a`; the `JoinGuard` must not outlive `'a`.
- The clause `where F: 'a` in the type of `scoped` says that `'a` is the closure's lifetime.
- Closures of this form borrow the variables they use; Rust won't let our closure outlive `x`.

Following this chain of constraints from top to bottom, `scoped` has ensured that the thread will always exit before the variables it uses go out of scope. Rust's compile-time checks guarantee that `scoped` threads' use of the surrounding variables is safe.

So, let's try our program again, using `scoped` instead of `spawn`:

```
let mut x = 1;
let thread1 = std::thread::scoped(|| { x += 8; });
let thread2 = std::thread::scoped(|| { x += 27; });
```

We've solved our lifetime problems, but this is still buggy, because we have two threads manipulating the same variable. Rust agrees:

```
error: cannot borrow `x` as mutable more than once at a time
    let thread2 = std::thread::scoped(|| { x += 27; });
                                       ^~~~~~~~~~~~~~~
note: borrow occurs due to use of `x` in closure
    let thread2 = std::thread::scoped(|| { x += 27; });
                                         ^
note: previous borrow of `x` occurs here due to use in closure;
the mutable borrow prevents subsequent moves, borrows, or
modification of `x` until the borrow ends
    let thread1 = std::thread::scoped(|| { x += 8; });
                                       ^~~~~~~~~~~~~~
```

What's happened here is pretty amazing: the error here is simply a consequence of Rust's generic rules about ownership and borrowing, but in this context they've prevented us from writing unsafe multithreaded code. Rust doesn't actually know anything about threads; it simply recognizes that this code breaks Rule 3: "You can only modify a value when you have exclusive access to it." Both closures modify `x`, yet they do not have exclusive access to it. Rejected.

Indeed, if we rewrite our code to remove the modification of `x`, so that the closures can borrow shared references to it, all is well. This code works perfectly:

```
let mut x = 1;
let thread1 = std::thread::scoped(|| { x + 8 });
let thread2 = std::thread::scoped(|| { x + 27 });
assert_eq!(thread1.join() + thread2.join(), 37);
```

But what if we really did want to modify `x` from within our threads? Can that be done?

## Mutexes

When several threads need to read and modify some shared data structure, they must take special care to ensure that these accesses are synchronized with each other. According to C++, failing to do so is undefined behavior; after defining its terms carefully, the 2011 C++ standard says:

> The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

This is an extremely broad class of behavior to leave undefined: if any thread modifies a value, and another thread reads that value, and no appropriate synchronization operation took place to mediate

between the two, your program is allowed to do anything at all. Not only is this rule difficult to follow in practice, but it magnifies the effect of any other bugs that might cause your program to touch data you hadn't intended.

One way to protect a data structure is to use a mutex. Only one thread may lock a mutex at a time, so if threads access the structure only while locking the mutex, the lock and unlock steps each thread performs serve as the synchronization operations we need to avoid undefined behavior.

Unfortunately, C and C++ leave the relationship between a mutex and the data it protects entirely implicit in the structure of the program. It's up to the developers to write comments that explain which threads can touch which data structures, and what mutexes must be held while doing so. Breaking the rules is a silent failure, and often one whose symptoms are difficult to reproduce reliably.

Rust's mutex type, `std::sync::Mutex`, leverages Rust's borrowing rules to ensure that threads never use a data structure without holding the mutex that protects it. Each mutex owns the data it protects; threads can borrow a reference to the data only by locking the mutex.

Here's how we can use `std::sync::Mutex` to let our scoped threads share access to our local variable `x`:

```
let x = std::sync::Mutex::new(1);
let thread1 = std::thread::scoped(|| {
    *x.lock().unwrap() += 8;
});
let thread2 = std::thread::scoped(|| {
    *x.lock().unwrap() += 27;
});
thread1.join();
thread2.join();
assert_eq!(*x.lock().unwrap(), 36);
```

Compared to our prior version, we've changed the type of `x` from `i32` to `Mutex<i32>`. Rather than sharing mutable access to a local `i32` as attempted above, the closures now share immutable access to the mutex. The expression `x.lock().unwrap()` locks the mutex, checks for errors, and returns a `MutexGuard` value. Dereferencing a `MutexGuard` borrows a reference (mutable or shareable, depending on the context) to the value the mutex protects—in this case, our

`i32` value. When the `MutexGuard` value is dropped, it automatically releases the mutex.

Taking a step back, let's look at what this API gives us:

- The only way to access the data structure a mutex protects is to lock it first.
- Doing so gives us a `MutexGuard`, which only lets us borrow a reference to the protected data structure. Rust's Rule 2 ("You can borrow a reference to a value, so long as the reference doesn't outlive the value") ensures that we must end the borrow before the `MutexGuard` is dropped.
- By Rust's Rule 3 ("You can only modify a value when you have exclusive access to it"), if we're modifying the value, we can't share it with other threads. If we share it with other threads, none of us can modify it. And recall that borrows affect the entire data structure up to the final owner (here, the mutex). So while our example mutex here only protects a simple integer, the same solution can protect structures of any size and complexity.
- Rust's Rule 1 ("Every value has a single owner at any given time") ensures that we will drop the `MutexGuard` at some well-defined point in the program. We cannot forget to unlock the mutex.

The result is a mutex API that grants threads access to shared mutable data, while ensuring at compile time that your program remains free of data races. As before, Rust's ownership and borrowing rules, innocent of any actual knowledge of threads, have provided exactly the checks we need to make mutex use sound.

**NOTE**

The absence of data races (and hence the absence of undefined behavior that they can cause) is critical, but it's not the same as the absence of nondeterministic behavior. We have no way of knowing which thread will add its value to `x` first; it could jump to 9 and then 36, or 28 and then 36. Similarly, we can only be sure the threads have completed their work after both have been joined. If we were to move our assertion before either of the `join` calls, the value it saw would vary from one run to the next.

**NOTE**

The `std::thread::scoped` function used here is undergoing some redesign, because it turns out to be unsafe in some (rare) circumstances. However that problem is resolved, Rust will continue to support concurrency patterns like those shown here in some form or another.

## Channels

Another popular approach to multithreaded programming is to let threads exchange messages with each other representing requests, replies, and the like. This is the approach the designers of the Go language advocate; the "Effective Go" document offers the slogan:

> Do not communicate by sharing memory; instead, share memory by communicating.

Rust's standard library includes a channel abstraction that supports this style of concurrency. One creates a channel by calling the `std::sync::mpsc::channel` function:

```
fn channel<T>() -> (Sender<T>, Receiver<T>)
```

This function returns a tuple of two values, representing the back and front of a message queue carrying values of type `T`: the `Sender<T>` enqueues values, and the `Receiver<T>` removes them from the queue.

The initialism "MPSC" here stands for "multiple producer, single consumer": the `Sender` end of a channel can be cloned and used by as many threads as you like to enqueue values; but the `Receiver` end cannot be cloned, so only a single thread is allowed to extract values from the queue.

Let's work through an example that uses channels to perform filesystem operations on a separate thread. We'll spawn a worker thread to carry out the requests, and then send it filenames to check. Here's a function that holds the worker's main loop:

```
// These declarations allow us to use these standard library
// definitions without writing out their full module path.
use std::fs::Metadata;
use std::io::Result;
use std::path::PathBuf;
use std::sync::mpsc::{Sender, Receiver};

fn worker_loop(files:   Receiver<PathBuf>,
               results: Sender<(PathBuf, Result<Metadata>)>) {
```

```
        for path_buf in files {
            let metadata = std::fs::metadata(&path_buf);
            results.send((path_buf, metadata)).unwrap();
        }
    }
```

This function takes two channel endpoints as arguments: we'll receive filenames on `files`, and send back results on `results`.

We represent the filenames we process as `std::path::PathBuf` values. A `PathBuf` resembles a `String`, except that whereas a `String` is always valid UTF-8, a `PathBuf` has no such scruples; it can hold any string the operating system will accept as a filename. `PathBuf` also provides cross-platform methods for operating on filenames. The standard library functions for working with the filesystem accept references to `PathBuf` values as filenames.

The `Receiver` type works nicely with `for` loops: writing `for path_buf in files` gives us a loop that iterates over each value received from the channel, and exits the loop when the sending end of the channel is closed.

For each `PathBuf` we receive, we call `std::fs::metadata` to look up the given file's metadata (modification time, size, permissions, and so on). Whether the call succeeds or fails, we send back a tuple containing the `PathBuf` and the result from the `metadata` call on our reply channel, `results`. Sending a value on a channel can fail if the receiving end has been dropped, so we must call `unwrap` on the result from the `send` to check for errors.

Before we look at the code for the client side, we should take note of how the `PathBuf` ownership is being handled here. A `PathBuf` owns a heap-allocated buffer that holds the path's text, so the `PathBuf` type cannot implement the `Copy` trait. Following Rust's Rule 1, that means that assigning, passing, or returning a `PathBuf` moves the value, rather than copying it. The source of the move is left with no value.

The client's sending end has type `Sender<PathBuf>`, which means that when we send a `PathBuf` on that channel, it is moved into the channel, which takes ownership. By Rust's Rule 2, there can't be any borrowed references to the `PathBuf` when this move occurs, so the sender has well and truly lost all access to the `PathBuf` and the heap-allocated buffer it owns. At the other end, receiving a `PathBuf` from the channel moves ownership from the channel to the caller. Each

iteration of the `for` loop in `worker_loop` takes ownership of the next `PathBuf` received, lets `std::fs::metadata` borrow it, and then sends it back to the main thread, along with the results of the `metadata` call. At no point do we ever need to copy the `PathBuf`s heap-allocated buffer; we just move the owning structure from client to server, and then back again.

Once again, Rust's rules for ownership, moves, and borrowing have let us construct a simple and flexible interface that enforces isolation between threads at compile time. We've allowed threads to exchange values without opening up any opportunity for data races or other undefined behavior.

Now we can turn to examine the client side:

```
use std::sync::mpsc::channel;
use std::thread::spawn;

let paths = vec!["/home/jimb/.bashrc",
                 "/home/jimb/.emacs",
                 "/home/jimb/nonesuch",
                 "/home/jimb/.cargo",
                 "/home/jimb/.golly"];

let worker;

// Create a channel the worker thread can use to send
// results to the main thread.
let (worker_tx, main_rx) = channel();

{
    // Create a channel the main thread can use to send
    // filenames to the worker.
    let (main_tx, worker_rx) = channel();

    // Start the worker thread.
    worker = spawn(move || {
        worker_loop(worker_rx, worker_tx);
    });

    // Send paths to the worker thread to check.
    for path in paths {
        main_tx.send(PathBuf::from(path)).unwrap();
    }

    // main_tx is dropped here, which closes the channel.
    // The worker will exit after it has received everything
    // we sent.
}
```

```
    // We could do other work here, while waiting for the
    // results to come back.
    for (path, result) in main_rx {
        match result {
            Ok(metadata) =>
            println!("Size of {:?}: {}", &path, metadata.len()),
            Err(err) =>
                println!("Error for {:?}: {}", &path, err)
        }
    }

    worker.join().unwrap();
```

We start with a list of filenames to process; these are statically allocated strings, from which we'll construct `PathBuf` values. We create two channels, one carrying filenames to the worker, and the other conveying results back. The way we spawn the worker thread is new:

```
    worker = spawn(move || {
        worker_loop(worker_rx, worker_tx);
    });
```

This may look like a use of the logical "or" operator, `||`, but `move` is actually a keyword: `move || { ... }` is a closure, and `||` is its empty argument list. The `move` indicates that this closure should capture the variables it uses from its environment by *moving* them into the closure value, not by borrowing them. In our present case, that means that this closure takes ownership of the `worker_rx` and `worker_tx` channel endpoints. Using a `move` closure here has two practical consequences:

- The closure has an unrestricted lifetime, since it doesn't depend on local variables located in any stack frame; it's carrying around its own copy of all the values it needs. This makes it suitable for use with `std::thread::spawn`, which doesn't necessarily guarantee that the thread it creates will exit at any particular time.
- When we create this closure, the variables `worker_rx` and `worker_tx` become uninitialized in the outer function; the main thread can no longer use them.

Having started the worker thread, the client then loops over our array of paths, creating a fresh `PathBuf` for each one, and sending it to the worker thread. When we reach the end of that block, `main_tx`

goes out of scope, dropping its `Sender` value. Closing the sending end of the channel signals `worker_loop`'s `for` loop to stop iterating, allowing the worker thread to exit.

Just as the worker function uses a `for` loop to handle requests, the main thread uses a `for` loop to process each result sent by the worker thread, using a `match` statement to handle the success and error cases, printing the results to our standard output.

Once we've processed all our results, we join on the worker thread and check the `Result`; this ensures that if the worker thread panicked, the main thread will panic as well, so that failures are not ignored.

On my machine, this program produces the following output:

```
Size of "/home/jimb/.bashrc": 259
Size of "/home/jimb/.emacs": 34210
Error for "/home/jimb/nonesuch": No such file or directory
(os error 2)
Size of "/home/jimb/.cargo": 4096
Size of "/home/jimb/.golly": 4096
```

It would be easy to extend our worker thread to receive not simple filenames but an enumeration of different sorts of requests it could handle: reading and writing files, deleting files, and so on. Or, we could simply send it closures to call, turning it into a completely open-ended worker thread. But no matter how we extend this structure, Rust's type safety and ownership rules ensure that our code will be free of data races and heap corruption.

At Mozilla, there is a sign on the wall behind one of our engineer's desks. The sign has a dark horizontal line, below which is the text, "You must be this tall to write multi-threaded code." The line is roughly nine feet off the ground. We created Rust to allow us to lower that sign.

## More Rust

Despite its youth, Rust is not a small language. It has many features worth exploring that we don't have space to cover here:

- Rust has a full library of collection types: sequences, maps, sets, and so on.

- Rust has reference-counted pointer types, `Rc` and `Arc`, which let us relax the "single owner" rules.
- Rust has support for `unsafe` blocks, in which one can call C code, use unrestricted pointers, reinterpret a value's bytes according to a different type, and generally wreak havoc. But safe interfaces with `unsafe` implementations turn out to be an effective technique for extending Rust's concept of safety.
- Rust's macro system is a drastic departure from the C and C++ preprocessor's macros, providing identifier hygiene and body parsing that is both extremely flexible and syntactically sound.
- Rust's module system helps organize large programs.
- Rust's package manager, Cargo, interacts with a shared public repository of packages, helping the community share code and growing the ecosystem of libraries (called "crates") available to use in Rust.

You can read more about all these on Rust's primary website, *http://www.rust-lang.org*, which has extensive library documentation, examples, and even an entire book about Rust.

# About the Author

**Jim Blandy** works for Mozilla on Firefox's tools for web developers. He is a committer to the SpiderMonkey JavaScript engine, and has been a maintainer of GNU Emacs, GNU Guile, and GDB. He is one of the original designers of the Subversion version control system.