A half-hour to learn Rust

Jan 27, 2020 · 51 minute read · rust

In order to increase fluency in a programming language, one has to read a lot of it. But how can you read a lot of it if you don't know what it means?

In this article, instead of focusing on one or two concepts, I'll try to go through as many Rust snippets as I can, and explain what the keywords and symbols they contain mean.

Ready? Go!

`let` introduces a variable binding:

Rust code

```rust
let x; // declare "x"
x = 42; // assign 42 to "x"
```

This can also be written as a single line:

Rust code

```rust
let x = 42;
```

You can specify the variable's type explicitly with `:`, that's a type annotation:

Rust code

```rust
let x: i32; // `i32` is a signed 32-bit integer
x = 42;

// there's i8, i16, i32, i64, i128
//    also u8, u16, u32, u64, u128 for unsigned
```

This can also be written as a single line:

Rust code

```rust
let x: i32 = 42;
```

If you declare a name and initialize it later, the compiler will prevent you from using it before it's initialized.

Rust code

```rust
let x;
foobar(x); // error: borrow of possibly-uninitialized variable: `x`
x = 42;
```

However, doing this is completely fine:

Rust code

```rust
let x;
x = 42;
foobar(x); // the type of `x` will be inferred from here
```

The underscore `_` is a special name - or rather, a "lack of name". It basically means to throw away something:

Rust code

```rust
// this does *nothing* because 42 is a constant
let _ = 42;

// this calls `get_thing` but throws away its result
let _ = get_thing();
```

Names that *start* with an underscore are regular names, it's just that the compiler won't warn about them being unused:

Rust code

```rust
// we may use `_x` eventually, but our code is a work-in-progress
// and we just wanted to get rid of a compiler warning for now.
let _x = 42;
```

Separate bindings with the same name can be introduced - you can *shadow* a variable binding:

Rust code

```rust
let x = 13;
let x = x + 3;
// using `x` after that line only refers to the second `x`,
// the first `x` no longer exists.
```

Rust has tuples, which you can think of as "fixed-length collections of values of different types".

Rust code

```rust
let pair = ('a', 17);
pair.0; // this is 'a'
pair.1; // this is 17
```

If we really wanted to annotate the type of `pair`, we would write:

Rust code

```rust
let pair: (char, i32) = ('a', 17);
```

Tuples can be *destructured* when doing an assignment, which means they're broken down into their individual fields:

Rust code

```rust
let (some_char, some_int) = ('a', 17);
// now, `some_char` is 'a', and `some_int` is 17
```

This is especially useful when a function returns a tuple:

Rust code

```rust
let (left, right) = slice.split_at(middle);
```

Of course, when destructuring a tuple, `_` can be used to throw away part of it:

Rust code

```
let (_, right) = slice.split_at(middle);
```

The semi-colon marks the end of a statement:

Rust code

```
let x = 3;
let y = 5;
let z = y + x;
```

Which means statements can span multiple lines:

Rust code

```
let x = vec![1, 2, 3, 4, 5, 6, 7, 8]
    .iter()
    .map(|x| x + 3)
    .fold(0, |x, y| x + y);
```

(We'll go over what those actually mean later).

`fn` declares a function.

Here's a void function:

Rust code

```
fn greet() {
    println!("Hi there!");
}
```

And here's a function that returns a 32-bit signed integer. The arrow indicates its return type:

Rust code

```
fn fair_dice_roll() -> i32 {
    4
}
```

A pair of brackets declares a block, which has its own scope:

Rust code

```
// This prints "in", then "out"
fn main() {
    let x = "out";
    {
        // this is a different `x`
        let x = "in";
        println!("{}", x);
    }
    println!("{}", x);
}
```

Blocks are also expressions, which mean they evaluate to.. a value.

Rust code

```
// this:
let x = 42;

// is equivalent to this:
let x = { 42 };
```

Inside a block, there can be multiple statements:

Rust code

```
let x = {
    let y = 1; // first statement
    let z = 2; // second statement
    y + z // this is the *tail* - what the whole block will evaluate to
};
```

And that's why "omitting the semicolon at the end of a function" is the same as returning, ie. these are equivalent:

Rust code

```rust
fn fair_dice_roll() -> i32 {
    return 4;
}

fn fair_dice_roll() -> i32 {
    4
}
```

`if` conditionals are also expressions:

Rust code

```rust
fn fair_dice_roll() -> i32 {
    if feeling_lucky {
        6
    } else {
        4
    }
}
```

A `match` is also an expression:

Rust code

```rust
fn fair_dice_roll() -> i32 {
    match feeling_lucky {
        true => 6,
        false => 4,
    }
}
```

Dots are typically used to access fields of a value:

Rust code

```rust
let a = (10, 20);
a.0; // this is 10

let amos = get_some_struct();
amos.nickname; // this is "fasterthanlime"
```

Or call a method on a value:

Rust code

```rust
let nick = "fasterthanlime";
nick.len(); // this is 14
```

The double-colon, `::`, is similar but it operates on namespaces.

In this example, `std` is a *crate* (~ a library), `cmp` is a *module* (~ a source file), and `min` is a *function*:

Rust code

```rust
let least = std::cmp::min(3, 8); // this is 3
```

`use` directives can be used to "bring in scope" names from other namespace:

Rust code

```rust
use std::cmp::min;

let least = min(7, 1); // this is 1
```

Within `use` directives, curly brackets have another meaning: they're "globs". If we want to import both `min` and `max`, we can do any of these:

Rust code

```rust
// this works:
use std::cmp::min;
use std::cmp::max;

// this also works:
use std::cmp::{min, max};

// this also works!
use std::{cmp::min, cmp::max};
```

A wildcard (`*`) lets you import every symbol from a namespace:

Rust code

```
// this brings `min` and `max` in scope, and many other things
use std::cmp::*;
```

Types are namespaces too, and methods can be called as regular functions:

Rust code

```
let x = "amos".len(); // this is 4
let x = str::len("amos"); // this is also 4
```

`str` is a primitive type, but many non-primitive types are also in scope by default.

Rust code

```
// `Vec` is a regular struct, not a primitive type
let v = Vec::new();

// this is exactly the same code, but with the *full* path to `Vec`
let v = std::vec::Vec::new();
```

This works because Rust inserts this at the beginning of every module:

Rust code

```
use std::prelude::v1::*;
```

(Which in turns re-exports a lot of symbols, like `Vec`, `String`, `Option` and `Result`).

Structs are declared with the `struct` keyword:

Rust code

```rust
struct Vec2 {
    x: f64, // 64-bit floating point, aka "double precision"
    y: f64,
}
```

They can be initialized using *struct literals*:

Rust code

```rust
let v1 = Vec2 { x: 1.0, y: 3.0 };
let v2 = Vec2 { y: 2.0, x: 4.0 };
// the order does not matter, only the names do
```

There is a shortcut for initializing the *rest of the fields* from another struct:

Rust code

```rust
let v3 = Vec2 {
    x: 14.0,
    ..v2
};
```

This is called "struct update syntax", can only happen in last position, and cannot be followed by a comma.

Note that *the rest of the fields* can mean *all the fields*:

Rust code

```rust
let v4 = Vec2 { ..v3 };
```

Structs, like tuples, can be destructured.

Just like this is a valid `let` pattern:

Rust code

```rust
let (left, right) = slice.split_at(middle);
```

So is this:

Rust code

```rust
let v = Vec2 { x: 3.0, y: 6.0 };
let Vec2 { x, y } = v;
// `x` is now 3.0, `y` is now `6.0`
```

And this:

Rust code

```rust
let Vec2 { x, .. } = v;
// this throws away `v.y`
```

`let` patterns can be used as conditions in `if`:

Rust code

```rust
struct Number {
    odd: bool,
    value: i32,
}

fn main() {
    let one = Number { odd: true, value: 1 };
    let two = Number { odd: false, value: 2 };
    print_number(one);
    print_number(two);
}

fn print_number(n: Number) {
    if let Number { odd: true, value } = n {
        println!("Odd number: {}", value);
    } else if let Number { odd: false, value } = n {
        println!("Even number: {}", value);
    }
}

// this prints:
// Odd number: 1
// Even number: 2
```

`match` arms are also patterns, just like `if let`:

Rust code

```rust
fn print_number(n: Number) {
    match n {
        Number { odd: true, value } => println!("Odd number: {}", value),
        Number { odd: false, value } => println!("Even number: {}", value),
    }
}

// this prints the same as before
```

A `match` has to be exhaustive: at least one arm needs to match.

Rust code

```rust
fn print_number(n: Number) {
    match n {
        Number { value: 1, .. } => println!("One"),
        Number { value: 2, .. } => println!("Two"),
        Number { value, .. } => println!("{}", value),
        // if that last arm didn't exist, we would get a compile-time error
    }
}
```

If that's hard, `_` can be used as a "catch-all" pattern:

Rust code

```rust
fn print_number(n: Number) {
    match n.value {
        1 => println!("One"),
        2 => println!("Two"),
        _ => println!("{}", n.value),
    }
}
```

You can declare methods on your own types:

Rust code

```rust
struct Number {
    odd: bool,
    value: i32,
}

impl Number {
    fn is_strictly_positive(self) -> bool {
        self.value > 0
    }
}
```

And use them like usual:

Rust code

```rust
fn main() {
    let minus_two = Number {
        odd: false,
        value: -2,
    };
    println!("positive? {}", minus_two.is_strictly_positive());
    // this prints "positive? false"
}
```

Variable bindings are immutable by default, which means their interior can't be mutated:

Rust code

```rust
fn main() {
    let n = Number {
        odd: true,
        value: 17,
    };
    n.odd = false; // error: cannot assign to `n.odd`,
                   // as `n` is not declared to be mutable
}
```

And also that they cannot be assigned to:

Rust code

```rust
fn main() {
    let n = Number {
        odd: true,
        value: 17,
    };
    n = Number {
        odd: false,
        value: 22,
    }; // error: cannot assign twice to immutable variable `n`
}
```

`mut` makes a variable binding mutable:

Rust code

```rust
fn main() {
    let mut n = Number {
        odd: true,
        value: 17,
    }
    n.value = 19; // all good
}
```

Traits are something multiple types can have in common:

Rust code

```rust
trait Signed {
    fn is_strictly_negative(self) -> bool;
}
```

You can implement:

- one of your traits on anyone's type
- anyone's trait on one of your types
- but not a foreign trait on a foreign type

These are called the "orphan rules".

Here's an implementation of our trait on our type:

Rust code

```rust
impl Signed for Number {
    fn is_strictly_negative(self) -> bool {
        self.value < 0
    }
}

fn main() {
    let n = Number { odd: false, value: -44 };
    println!("{}", n.is_strictly_negative()); // prints "true"
}
```

Our trait on a foreign type (a primitive type, even):

Rust code

```rust
impl Signed for i32 {
    fn is_strictly_negative(self) -> bool {
        self < 0
    }
}

fn main() {
    let n: i32 = -44;
    println!("{}", n.is_strictly_negative()); // prints "true"
}
```

A foreign trait on our type:

Rust code

```rust
// the `Neg` trait is used to overload `-`, the
// unary minus operator.
impl std::ops::Neg for Number {
    type Output = Number;

    fn neg(self) -> Number {
        Number {
            value: -self.value,
            odd: self.odd,
        }
    }
```

```
}

fn main() {
    let n = Number { odd: true, value: 987 };
    let m = -n; // this is only possible because we implemented `Neg`
    println!("{}", m.value); // prints "-987"
}
```

An `impl` block is always *for* a type, so, inside that block, `Self` means that type:

Rust code

```rust
impl std::ops::Neg for Number {
    type Output = Self;

    fn neg(self) -> Self {
        Self {
            value: -self.value,
            odd: self.odd,
        }
    }
}
```

Some traits are *markers* - they don't say that a type implements some methods, they say that certain things can be done with a type.

For example, `i32` implements trait `Copy` (in short, `i32` *is* `Copy`), so this works:

Rust code

```rust
fn main() {
    let a: i32 = 15;
    let b = a; // `a` is copied
    let c = a; // `a` is copied again
}
```

And this also works:

Rust code

```rust
fn print_i32(x: i32) {
    println!("x = {}", x);
```

```
}

fn main() {
    let a: i32 = 15;
    print_i32(a); // `a` is copied
    print_i32(a); // `a` is copied again
}
```

But the `Number` struct is not `Copy`, so this doesn't work:

Rust code

```rust
fn main() {
    let n = Number { odd: true, value: 51 };
    let m = n; // `n` is moved into `m`
    let o = n; // error: use of moved value: `n`
}
```

And neither does this:

Rust code

```rust
fn print_number(n: Number) {
    println!("{} number {}", if n.odd { "odd" } else { "even" }, n.value);
}

fn main() {
    let n = Number { odd: true, value: 51 };
    print_number(n); // `n` is moved
    print_number(n); // error: use of moved value: `n`
}
```

But it works if `print_number` takes an immutable reference instead:

Rust code

```rust
fn print_number(n: &Number) {
    println!("{} number {}", if n.odd { "odd" } else { "even" }, n.value);
}

fn main() {
    let n = Number { odd: true, value: 51 };
    print_number(&n); // `n` is borrowed for the time of the call
```

```
    print_number(&n); // `n` is borrowed again
}
```

It also works if a function takes a *mutable* reference - but only if our variable binding is also `mut`.

Rust code

```
fn invert(n: &mut Number) {
    n.value = -n.value;
}

fn print_number(n: &Number) {
    println!("{} number {}", if n.odd { "odd" } else { "even" }, n.value);
}

fn main() {
    // this time, `n` is mutable
    let mut n = Number { odd: true, value: 51 };
    print_number(&n);
    invert(&mut n); // `n is borrowed mutably - everything is explicit
    print_number(&n);
}
```

Trait methods can also take `self` by reference or mutable reference:

Rust code

```
impl std::clone::Clone for Number {
    fn clone(&self) -> Self {
        Self { ..*self }
    }
}
```

When invoking trait methods, the receiver is borrowed implicitly:

Rust code

```
fn main() {
    let n = Number { odd: true, value: 51 };
    let mut m = n.clone();
    m.value += 100;
```

```
    print_number(&n);
    print_number(&m);
}
```

To highlight this: these are equivalent:

Rust code

```
let m = n.clone();

let m = std::clone::Clone::clone(&n);
```

Marker traits like `Copy` have no methods:

Rust code

```
// note: `Copy` requires that `Clone` is implemented too
impl std::clone::Clone for Number {
    fn clone(&self) -> Self {
        Self { ..*self }
    }
}

impl std::marker::Copy for Number {}
```

Now, `Clone` can still be used:

Rust code

```
fn main() {
    let n = Number { odd: true, value: 51 };
    let m = n.clone();
    let o = n.clone();
}
```

But `Number` values will no longer be moved:

Rust code

```
fn main() {
    let n = Number { odd: true, value: 51 };
    let m = n; // `m` is a copy of `n`
```

```
    let o = n; // same. `n` is neither moved nor borrowed.
}
```

Some traits are so common, they can be implemented automatically by using the `derive` attribute:

Rust code

```rust
#[derive(Clone, Copy)]
struct Number {
    odd: bool,
    value: i32,
}

// this expands to `impl Clone for Number` and `impl Copy for Number` blocks.
```

Functions can be generic:

Rust code

```rust
fn foobar<T>(arg: T) {
    // do something with `arg`
}
```

They can have multiple *type parameters*, which can then be used in the function's declaration and its body, instead of concrete types:

Rust code

```rust
fn foobar<L, R>(left: L, right: R) {
    // do something with `left` and `right`
}
```

Type parameters usually have *constraints*, so you can actually do something with them.

The simplest constraints are just trait names:

Rust code

```rust
fn print<T: Display>(value: T) {
    println!("value = {}", value);
}

fn print<T: Debug>(value: T) {
    println!("value = {:?}", value);
}
```

There's a longer syntax for type parameter constraints:

Rust code

```rust
fn print<T>(value: T)
where
    T: Display,
{
    println!("value = {}", value);
}
```

Constraints can be more complicated: they can require a type parameter to implement multiple traits:

Rust code

```rust
use std::fmt::Debug;

fn compare<T>(left: T, right: T)
where
    T: Debug + PartialEq,
{
    println!("{:?} {} {:?}", left, if left == right { "==" } else { "!=" },
}

fn main() {
    compare("tea", "coffee");
    // prints: "tea" != "coffee"
}
```

Generic functions can be thought of as namespaces, containing an infinity of functions with different concrete types.

Same as with crates, and modules, and types, generic functions can be "explored" (navigated?) using `::`

Rust code

```
fn main() {
    use std::any::type_name;
    println!("{}", type_name::<i32>()); // prints "i32"
    println!("{}", type_name::<(f64, char)>()); // prints "(f64, char)"
}
```

This is lovingly called [turbofish syntax](), because `::<>` looks like a fish.

Structs can be generic too:

Rust code

```
struct Pair<T> {
    a: T,
    b: T,
}

fn print_type_name<T>(_val: &T) {
    println!("{}", std::any::type_name::<T>());
}

fn main() {
    let p1 = Pair { a: 3, b: 9 };
    let p2 = Pair { a: true, b: false };
    print_type_name(&p1); // prints "Pair<i32>"
    print_type_name(&p2); // prints "Pair<bool>"
}
```

The standard library type `Vec` (~ a heap-allocated array), is generic:

Rust code

```
fn main() {
    let mut v1 = Vec::new();
    v1.push(1);
    let mut v2 = Vec::new();
    v2.push(false);
    print_type_name(&v1); // prints "Vec<i32>"
```

```
    print_type_name(&v2); // prints "Vec<bool>"
}
```

Speaking of `Vec`, it comes with a macro that gives more or less "vec literals":

Rust code

```
fn main() {
    let v1 = vec![1, 2, 3];
    let v2 = vec![true, false, true];
    print_type_name(&v1); // prints "Vec<i32>"
    print_type_name(&v2); // prints "Vec<bool>"
}
```

All of `name!()`, `name![]` or `name!{}` invoke a macro. Macros just expand to regular code.

In fact, `println` is a macro:

Rust code

```
fn main() {
    println!("{}", "Hello there!");
}
```

This expands to something that has the same effect as:

Rust code

```
fn main() {
    use std::io::{self, Write};
    io::stdout().lock().write_all(b"Hello there!\n").unwrap();
}
```

`panic` is also a macro. It violently stops execution with an error message, and the file name / line number of the error, if enabled:

Rust code

```
fn main() {
    panic!("This panics");
}
// output: thread 'main' panicked at 'This panics', src/main.rs:3:5
```

Some methods also panic. For example, the `Option` type can contain something, or it can contain nothing. If `.unwrap()` is called on it, and it contains nothing, it panics:

Rust code

```
fn main() {
    let o1: Option<i32> = Some(128);
    o1.unwrap(); // this is fine

    let o2: Option<i32> = None;
    o2.unwrap(); // this panics!
}

// output: thread 'main' panicked at 'called `Option::unwrap()` on a `None`
```

`Option` is not a struct - it's an `enum`, with two variants.

Rust code

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

impl<T> Option<T> {
    fn unwrap(self) -> T {
        // enums variants can be used in patterns:
        match self {
            Self::Some(t) => t,
            Self::None => panic!(".unwrap() called on a None option"),
        }
    }
}

use self::Option::{None, Some};

fn main() {
    let o1: Option<i32> = Some(128);
```

```
    let o1: Option<i32> = Some(128);
    o1.unwrap(); // this is fine

    let o2: Option<i32> = None;
    o2.unwrap(); // this panics!
}

// output: thread 'main' panicked at '.unwrap() called on a None option', sr
```

`Result` is also an enum, it can either contain something, or an error:

Rust code

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

It also panics when unwrapped and containing an error.

Variables bindings have a "lifetime":

Rust code

```rust
fn main() {
    // `x` doesn't exist yet
    {
        let x = 42; // `x` starts existing
        println!("x = {}", x);
        // `x` stops existing
    }
    // `x` no longer exists
}
```

Similarly, references have a lifetime:

Rust code

```rust
fn main() {
    // `x` doesn't exist yet
    {
        let x = 42; // `x` starts existing
        let x_ref = &x; // `x_ref` starts existing - it borrows `x`
```

```
        println!("x_ref = {}", x_ref);
        // `x_ref` stops existing
        // `x` stops existing
    }
    // `x` no longer exists
}
```

The lifetime of a reference cannot exceed the lifetime of the variable binding it borrows:

Rust code

```
fn main() {
    let x_ref = {
        let x = 42;
        &x
    };
    println!("x_ref = {}", x_ref);
    // error: `x` does not live long enough
}
```

A variable binding can be immutably borrowed multiple times:

Rust code

```
fn main() {
    let x = 42;
    let x_ref1 = &x;
    let x_ref2 = &x;
    let x_ref3 = &x;
    println!("{} {} {}", x_ref1, x_ref2, x_ref3);
}
```

While borrowed, a variable binding cannot be mutated:

Rust code

```
fn main() {
    let mut x = 42;
    let x_ref = &x;
    x = 13;
    println!("x_ref = {}", x_ref);
```

```
    // error: cannot assign to `x` because it is borrowed
}
```

While immutably borrowed, a variable cannot be *mutably borrowed*:

Rust code

```
fn main() {
    let mut x = 42;
    let x_ref1 = &x;
    let x_ref2 = &mut x;
    // error: cannot borrow `x` as mutable because it is also borrowed as imr
    println!("x_ref1 = {}", x_ref1);
}
```

References in function arguments also have lifetimes:

Rust code

```
fn print(x: &i32) {
    // `x` is borrowed (from the outside) for the
    // entire time this function is called.
}
```

Functions with reference arguments can be called with borrows that have different lifetimes, so:

- All functions that take references are generic
- Lifetimes are generic parameters

Lifetimes' names start with a single quote, `'`:

Rust code

```
// elided (non-named) lifetimes:
fn print(x: &i32) {}

// named lifetimes:
fn print<'a>(x: &'a i32) {}
```

This allows returning references whose lifetime depend on the lifetime of the arguments:

Rust code

```
struct Number {
    value: i32,
}

fn number_value<'a>(num: &'a Number) -> &'a i32 {
    &num.value
}

fn main() {
    let n = Number { value: 47 };
    let v = number_value(&n);
    // `v` borrows `n` (immutably), thus: `v` cannot outlive `n`.
    // While `v` exists, `n` cannot be mutably borrowed, mutated, moved, etc.
}
```

When there is a *single* input lifetime, it doesn't need to be named, and everything has the same lifetime, so the two functions below are equivalent:

Rust code

```
fn number_value<'a>(num: &'a Number) -> &'a i32 {
    &num.value
}

fn number_value(num: &Number) -> &i32 {
    &num.value
}
```

Structs can also be *generic over lifetimes*, which allows them to hold references:

Rust code

```
struct NumRef<'a> {
    x: &'a i32,
}

fn main() {
    let x: i32 = 99;
```

```rust
    let x_ref = NumRef { x: &x };
    // `x_ref` cannot outlive `x`, etc.
}
```

The same code, but with an additional function:

Rust code

```rust
struct NumRef<'a> {
    x: &'a i32,
}

fn as_num_ref<'a>(x: &'a i32) -> NumRef<'a> {
    NumRef { x: &x }
}

fn main() {
    let x: i32 = 99;
    let x_ref = as_num_ref(&x);
    // `x_ref` cannot outlive `x`, etc.
}
```

The same code, but with "elided" lifetimes:

Rust code

```rust
struct NumRef<'a> {
    x: &'a i32,
}

fn as_num_ref(x: &i32) -> NumRef<'_> {
    NumRef { x: &x }
}

fn main() {
    let x: i32 = 99;
    let x_ref = as_num_ref(&x);
    // `x_ref` cannot outlive `x`, etc.
}
```

`impl` blocks can be generic over lifetimes too:

Rust code

```
impl<'a> NumRef<'a> {
    fn as_i32_ref(&'a self) -> &'a i32 {
        self.x
    }
}

fn main() {
    let x: i32 = 99;
    let x_num_ref = NumRef { x: &x };
    let x_i32_ref = x_num_ref.as_i32_ref();
    // neither ref can outlive `x`
}
```

But you can do elision ("to elide") there too:

Rust code

```
impl<'a> NumRef<'a> {
    fn as_i32_ref(&self) -> &i32 {
        self.x
    }
}
```

You can elide even harder, if you never need the name:

Rust code

```
impl NumRef<'_> {
    fn as_i32_ref(&self) -> &i32 {
        self.x
    }
}
```

There is a special lifetime, named `'static`, which is valid for the entire program's lifetime.

String literals are `'static`:

Rust code

```
struct Person {
    name: &'static str,
```

```
}

fn main() {
    let p = Person {
        name: "fasterthanlime",
    };
}
```

But *owned strings* are not static:

Rust code

```rust
struct Person {
    name: &'static str,
}

fn main() {
    let name = format!("fasterthan{}", "lime");
    let p = Person { name: &name };
    // error: `name` does not live long enough
}
```

In that last example, the local `name` is not a `&'static str`, it's a `String`. It's been allocated dynamically, and it will be freed. Its lifetime is *less* than the whole program (even though it happens to be in `main`).

To store a non-`'static` string in `Person`, it needs to either:

A) Be generic over a lifetime:

Rust code

```rust
struct Person<'a> {
    name: &'a str,
}

fn main() {
    let name = format!("fasterthan{}", "lime");
    let p = Person { name: &name };
    // `p` cannot outlive `name`
}
```

or

B) Take ownership of the string

Rust code

```rust
struct Person {
    name: String,
}

fn main() {
    let name = format!("fasterthan{}", "lime");
    let p = Person { name: name };
    // `name` was moved into `p`, their lifetimes are no longer tied.
}
```

Speaking of: in a struct literal, when a field is set to a variable binding of the same name:

Rust code

```rust
    let p = Person { name: name };
```

It can be shortened like this:

Rust code

```rust
    let p = Person { name };
```

For many types in Rust, there are owned and non-owned variants:

- Strings: `String` is owned, `&str` is a reference
- Paths: `PathBuf` is owned, `&Path` is a reference
- Collections: `Vec<T>` is owned, `&[T]` is a reference

Rust has slices - they're a reference to multiple contiguous elements.

You can borrow a slice of a vector, for example:

Rust code

```rust
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    let v2 = &v[2..4];
    println!("v2 = {:?}", v2);
}

// output:
// v2 = [3, 4]
```

The above is not magical. The indexing operator (`foo[index]`) is overloaded with the `Index` and `IndexMut` traits.

The `..` syntax is just range literals. Ranges are just a few structs defined in the standard library.

They can be open-ended, and their rightmost bound can be inclusive, if it's preceded by `=`.

Rust code

```rust
fn main() {
    // 0 or greater
    println!("{:?}", (0..).contains(&100)); // true
    // strictly less than 20
    println!("{:?}", (..20).contains(&20)); // false
    // 20 or less than 20
    println!("{:?}", (..=20).contains(&20)); // true
    // only 3, 4, 5
    println!("{:?}", (3..6).contains(&4)); // true
}
```

Borrowing rules apply to slices.

Rust code

```rust
fn tail(s: &[u8]) -> &[u8] {
  &s[1..]
}

fn main() {
```

```rust
    let x = &[1, 2, 3, 4, 5];
    let y = tail(x);
    println!("y = {:?}", y);
}
```

This is the same as:

Rust code

```rust
fn tail<'a>(s: &'a [u8]) -> &'a [u8] {
  &s[1..]
}
```

This is legal:

Rust code

```rust
fn main() {
    let y = {
        let x = &[1, 2, 3, 4, 5];
        tail(x)
    };
    println!("y = {:?}", y);
}
```

...but only because `[1, 2, 3, 4, 5]` is a `'static` array.

So, this is illegal:

Rust code

```rust
fn main() {
    let y = {
        let v = vec![1, 2, 3, 4, 5];
        tail(&v)
        // error: `v` does not live long enough
    };
    println!("y = {:?}", y);
}
```

...because a vector is heap-allocated, and it has a non-`'static` lifetime.

`&str` values are really slices.

Rust code

```
fn file_ext(name: &str) -> Option<&str> {
    // this does not create a new string - it returns
    // a slice of the argument.
    name.split(".").last()
}

fn main() {
    let name = "Read me. Or don't.txt";
    if let Some(ext) = file_ext(name) {
        println!("file extension: {}", ext);
    } else {
        println!("no file extension");
    }
}
```

...so the borrow rules apply here too:

Rust code

```
fn main() {
    let ext = {
        let name = String::from("Read me. Or don't.txt");
        file_ext(&name).unwrap_or("")
        // error: `name` does not live long enough
    };
    println!("extension: {:?}", ext);
}
```

Functions that can fail typically return a `Result`:

Rust code

```
fn main() {
    let s = std::str::from_utf8(&[240, 159, 141, 137]);
    println!("{:?}", s);
    // prints: Ok("🍉")

    let s = std::str::from_utf8(&[195, 40]);
    println!("{:?}", s);
    // prints: Err(Utf8Error { valid_up_to: 0, error_len: Some(1) })
}
```

If you want to panic in case of failure, you can `.unwrap()`:

Rust code

```
fn main() {
    let s = std::str::from_utf8(&[240, 159, 141, 137]).unwrap();
    println!("{:?}", s);
    // prints: "🍉"

    let s = std::str::from_utf8(&[195, 40]).unwrap();
    // prints: thread 'main' panicked at 'called `Result::unwrap()`
    // on an `Err` value: Utf8Error { valid_up_to: 0, error_len: Some(1) }',
    // src/libcore/result.rs:1165:5
}
```

Or `.expect()`, for a custom message:

Rust code

```
fn main() {
    let s = std::str::from_utf8(&[195, 40]).expect("valid utf-8");
    // prints: thread 'main' panicked at 'valid utf-8: Utf8Error
    // { valid_up_to: 0, error_len: Some(1) }', src/libcore/result.rs:1165:5
}
```

Or, you can `match`:

Rust code

```
fn main() {
    match std::str::from_utf8(&[240, 159, 141, 137]) {
        Ok(s) => println!("{}", s),
        Err(e) => panic!(e),
    }
    // prints 🍉
}
```

Or you can `if let`:

Rust code

```
fn main() {
    if let Ok(s) = std::str::from_utf8(&[240, 159, 141, 137]) {
        println!("{}", s);
    }
    // prints 🍉
}
```

Or you can bubble up the error:

Rust code

```
fn main() -> Result<(), std::str::Utf8Error> {
    match std::str::from_utf8(&[240, 159, 141, 137]) {
        Ok(s) => println!("{}", s),
        Err(e) => return Err(e),
    }
    Ok(())
}
```

Or you can use `?` to do it the concise way:

Rust code

```
fn main() -> Result<(), std::str::Utf8Error> {
    let s = std::str::from_utf8(&[240, 159, 141, 137])?;
    println!("{}", s);
    Ok(())
}
```

The `*` operator can be used to *dereference*, but you don't need to do that to access fields or call methods:

Rust code

```
struct Point {
    x: f64,
    y: f64,
}

fn main() {
    let p = Point { x: 1.0, y: 3.0 };
    let p_ref = &p;
    println!("({}, {})", p_ref.x, p_ref.y);
```

```
}

// prints `(1, 3)`
```

And you can only do it if the type is `Copy`:

Rust code

```rust
struct Point {
    x: f64,
    y: f64,
}

fn negate(p: Point) -> Point {
    Point {
        x: -p.x,
        y: -p.y,
    }
}

fn main() {
    let p = Point { x: 1.0, y: 3.0 };
    let p_ref = &p;
    negate(*p_ref);
    // error: cannot move out of `*p_ref` which is behind a shared reference
}
```

Rust code

```rust
// now `Point` is `Copy`
#[derive(Clone, Copy)]
struct Point {
    x: f64,
    y: f64,
}

fn negate(p: Point) -> Point {
    Point {
        x: -p.x,
        y: -p.y,
    }
}

fn main() {
    let p = Point { x: 1.0, y: 3.0 };
    let p_ref = &p;
```

```
    negate(*p_ref); // ...and now this works
}
```

Closures are just functions of type `Fn`, `FnMut` or `FnOnce` with some captured context.

Their parameters are a comma-separated list of names within a pair of pipes (`|`). They don't *need* curly braces, unless you want to have multiple statements.

Rust code

```rust
fn for_each_planet<F>(f: F)
    where F: Fn(&'static str)
{
    f("Earth");
    f("Mars");
    f("Jupiter");
}

fn main() {
    for_each_planet(|planet| println!("Hello, {}", planet));
}

// prints:
// Hello, Earth
// Hello, Mars
// Hello, Jupiter
```

The borrow rules apply to them too:

Rust code

```rust
fn for_each_planet<F>(f: F)
    where F: Fn(&'static str)
{
    f("Earth");
    f("Mars");
    f("Jupiter");
}

fn main() {
    let greeting = String::from("Good to see you");
    for_each_planet(|planet| println!("{}, {}", greeting, planet));
```

```
    // our closure borrows `greeting`, so it cannot outlive it
}
```

For example, this would not work:

Rust code

```
fn for_each_planet<F>(f: F)
    where F: Fn(&'static str) + 'static // `F` must now have "'static" lifeti
{
    f("Earth");
    f("Mars");
    f("Jupiter");
}

fn main() {
    let greeting = String::from("Good to see you");
    for_each_planet(|planet| println!("{}, {}", greeting, planet));
    // error: closure may outlive the current function, but it borrows
    // `greeting`, which is owned by the current function
}
```

But this would:

Rust code

```
fn main() {
    let greeting = String::from("You're doing great");
    for_each_planet(move |planet| println!("{}, {}", greeting, planet));
    // `greeting` is no longer borrowed, it is *moved* into
    // the closure.
}
```

An `FnMut` needs to be mutably borrowed to be called, so it can only be called once at a time.

This is legal:

Rust code

```
fn foobar<F>(f: F)
    where F: Fn(i32) -> i32
```

```
{
    println!("{}", f(f(2)));
}

fn main() {
    foobar(|x| x * 2);
}

// output: 8
```

This isn't:

Rust code

```
fn foobar<F>(mut f: F)
    where F: FnMut(i32) -> i32
{
    println!("{}", f(f(2)));
    // error: cannot borrow `f` as mutable more than once at a time
}

fn main() {
    foobar(|x| x * 2);
}
```

This is legal again:

Rust code

```
fn foobar<F>(mut f: F)
    where F: FnMut(i32) -> i32
{
    let tmp = f(2);
    println!("{}", f(tmp));
}

fn main() {
    foobar(|x| x * 2);
}

// output: 8
```

`FnMut` exists because some closures *mutably borrow* local variables:

Rust code

```rust
fn foobar<F>(mut f: F)
    where F: FnMut(i32) -> i32
{
    let tmp = f(2);
    println!("{}", f(tmp));
}

fn main() {
    let mut acc = 2;
    foobar(|x| {
        acc += 1;
        x * acc
    });
}

// output: 24
```

Those closures cannot be passed to functions expecting `Fn`:

Rust code

```rust
fn foobar<F>(f: F)
    where F: Fn(i32) -> i32
{
    println!("{}", f(f(2)));
}

fn main() {
    let mut acc = 2;
    foobar(|x| {
        acc += 1;
        // error: cannot assign to `acc`, as it is a
        // captured variable in a `Fn` closure.
        // the compiler suggests "changing foobar
        // to accept closures that implement `FnMut`"
        x * acc
    });
}
```

`FnOnce` closures can only be called once. They exist because some closure move out variables that have been moved when captured:

Rust code

```rust
fn foobar<F>(f: F)
    where F: FnOnce() -> String
{
    println!("{}", f());
}

fn main() {
    let s = String::from("alright");
    foobar(move || s);
    // `s` was moved into our closure, and our
    // closures moves it to the caller by returning
    // it. Remember that `String` is not `Copy`.
}
```

This is enforced naturally, as `FnOnce` closures need to be *moved* in order to be called.

So, for example, this is illegal:

Rust code

```rust
fn foobar<F>(f: F)
    where F: FnOnce() -> String
{
    println!("{}", f());
    println!("{}", f());
    // error: use of moved value: `f`
}
```

And, if you need convincing that our closure *does* move `s`, this is illegal too:

Rust code

```rust
fn main() {
    let s = String::from("alright");
    foobar(move || s);
    foobar(move || s);
    // use of moved value: `s`
}
```

But this is fine:

Rust code

```rust
fn main() {
    let s = String::from("alright");
    foobar(|| s.clone());
    foobar(|| s.clone());
}
```

Here's a closure with two arguments:

Rust code

```rust
fn foobar<F>(x: i32, y: i32, is_greater: F)
    where F: Fn(i32, i32) -> bool
{
    let (greater, smaller) = if is_greater(x, y) {
        (x, y)
    } else {
        (y, x)
    };
    println!("{} is greater than {}", greater, smaller);
}

fn main() {
    foobar(32, 64, |x, y| x > y);
}
```

Here's a closure ignoring both its arguments:

Rust code

```rust
fn main() {
    foobar(32, 64, |_, _| panic!("Comparing is futile!"));
}
```

Here's a slightly worrying closure:

Rust code

```rust
fn countdown<F>(count: usize, tick: F)
    where F: Fn(usize)
{
    for i in (1..=count).rev() {
        tick(i);
```

```
    }
}

fn main() {
    countdown(3, |i| println!("tick {}...", i));
}

// output:
// tick 3...
// tick 2...
// tick 1...
```

And here's a toilet closure:

Rust code

```
fn main() {
    countdown(3, |_| ());
}
```

Called thusly because `|_| ()` looks like a toilet.

Anything that is iterable can be used in a `for in` loop.

We've just seen a range being used, but it also works with a `Vec`:

Rust code

```
fn main() {
    for i in vec![52, 49, 21] {
        println!("I like the number {}", i);
    }
}
```

Or a slice:

Rust code

```
fn main() {
    for i in &[52, 49, 21] {
        println!("I like the number {}", i);
    }
```

```
}

// output:
// I like the number 52
// I like the number 49
// I like the number 21
```

Or an actual iterator:

Rust code

```rust
fn main() {
    // note: `&str` also has a `.bytes()` iterator.
    // Rust's `char` type is a "Unicode scalar value"
    for c in "rust".chars() {
        println!("Give me a {}", c);
    }
}

// output:
// Give me a r
// Give me a u
// Give me a s
// Give me a t
```

Even if the iterator items are filtered and mapped and flattened:

Rust code

```rust
fn main() {
    for c in "SuRPRISE INbOUND"
        .chars()
        .filter(|c| c.is_lowercase())
        .flat_map(|c| c.to_uppercase())
    {
        print!("{}", c);
    }
    println!();
}

// output: UB
```

You can return a closure from a function:

Rust code

```rust
fn make_tester(answer: String) -> impl Fn(&str) -> bool {
    move |challenge| {
        challenge == answer
    }
}

fn main() {
    // you can use `.into()` to perform conversions
    // between various types, here `&'static str` and `String`
    let test = make_tester("hunter2".into());
    println!("{}", test("******"));
    println!("{}", test("hunter2"));
}
```

You can even move a reference to some of a function's arguments, into a closure it returns:

Rust code

```rust
fn make_tester<'a>(answer: &'a str) -> impl Fn(&str) -> bool + 'a {
    move |challenge| {
        challenge == answer
    }
}

fn main() {
    let test = make_tester("hunter2");
    println!("{}", test("*******"));
    println!("{}", test("hunter2"));
}

// output:
// false
// true
```

Or, with elided lifetimes:

Rust code

```rust
fn make_tester(answer: &str) -> impl Fn(&str) -> bool + '_ {
    move |challenge| {
        challenge == answer
    }
}
```

And with that, we hit the 30-minute estimated reading time mark, and you should be able to read *most* of the Rust code you find online.

Writing Rust is a very different experience from reading Rust. On one hand, you're not reading the *solution* to a problem, you're actually solving it. On the other hand, the Rust compiler helps out a *lot*.

For all of the intentional mistakes made above ("this code is illegal", etc.), rustc always has very good error messages *and* insightful suggestions.

And when there's a hint missing, the compiler team is not afraid to add it.

For more Rust material, you may want to check out:

- The Rust Book
- Rust By Example
- Read Rust
- This Week In Rust

I also blog about Rust and tweet about Rust a lot, so if you liked this article, you know what to do.

Have fun!