Lifetimes

```
error[E0597]: `x` does not live long enough
```


The Borrowing Rules

1. No borrower cannot live longer than the owner
   - i.e., no dangling references
2. You may have one or the other of these two kinds of borrows, but not both at the same time:
   - one or more shared references (`&T`) to a resource,
   - exactly one mutable reference (`&mut T`).

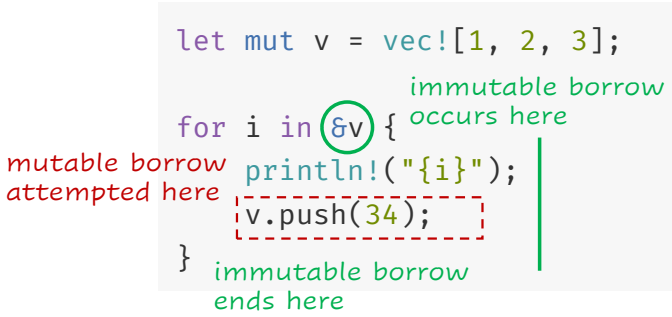
Why have these restrictive rules? These rules prevent data races.

Data Races

There is a 'data race' when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized.

Iterator invalidation

- “Iterator invalidation” happens when you try to mutate a collection that you’re iterating over. Rust’s borrow checker prevents this from happening:




```
error: cannot borrow `v` as mutable because it is also
borrowed as immutable
    v.push(34);
    ^
note: previous borrow of `v` occurs here; the immutable
borrow prevents subsequent moves or mutable borrows of
`v` until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!(“{}”, i);
    v.push(34);
}
^
```

- We can’t modify v because it’s borrowed by the loop.

Use after Free

- **References must not live longer than the resource they refer to**. Rust will check the lifetime of your references to ensure that this is true.

```
let y: &i32;
    {
        let x = 5;
        y = &x;
    }
println!("{y}");
```

```
error: `x` does not live long enough
y = &x;
     ^
```

- y is only valid for the scope where x exists. As soon as x goes away, it becomes invalid to refer to it.

Lifetime and Scope

- A variable's lifetime is **a scope in which it is valid**.
- A variable's lifetime **begins when it is created and ends when it is dropped**.
- *A lifetime is a construct the compiler (or more specifically, its **borrow checker**) uses to ensure all borrows are valid.*
- When we borrow a variable via &, the borrow has a lifetime that is
  - determined by where it is declared, and
  - is valid as long as it ends before the lender is destroyed.

Every reference in Rust has a lifetime.


```
// Lifetimes are annotated below with lines denoting the creation
// and destruction of each variable.
// `i` has the longest lifetime because its scope entirely encloses
// both `borrow1` and `borrow2`. The duration of `borrow1` compared
// to `borrow2` is irrelevant since they are disjoint.
fn main() {
    let i = 3; // Lifetime for `i` starts.
    {
        let borrow1 = &i; // `borrow1` lifetime starts.

        println!("borrow1: {borrow1}");   //
    } // `borrow1 ends.


    {
        let borrow2 = &i; // `borrow2` lifetime starts.

        println!("borrow2: {borrow2}");   //
    } // `borrow2` ends.

}   // i's lifetime ends.
```

Preventing Dangling References with Lifetimes

- The main aim of lifetimes is to prevent *dangling references*.

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
6 |         r = &x;
  |             ^^ borrowed value does not live long enough
7 |     }
  |     - `x` dropped here while still borrowed
8 |
9 |     println!("r: {}", r);
  |                        - borrow later used here
```


Borrow Checker

- The Rust compiler has a *borrow checker* that compares scopes to determine whether all borrows are valid.

```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

r's lifetime = lifetime(a') > lifetime(b') = x's lifetime


```
fn main() {
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); // --+       |
}                         // ----------+
```

r's lifetime(a') < x's lifetime(b')

Lifetime Annotation Syntax

- Lifetime annotations describe **the relationships of the lifetimes** of multiple references to each other without affecting the lifetimes.

```
&i32          // a reference
&'a i32       // a reference with an explicit lifetime
&mut i32      // a mutable reference
&'a mut i32   // a mutable reference with an explicit lifetime
```

the lifetime a

```
fn foo<'a>(...)
fn foo<'a, 'b>(...)
fn foo<'a>(x: &'a i32, y: &'a mut i32) -> &'a i32
```


What could be wrong?
What should be guaranteed?

```
fn f(s: &str, t: &str) -> &str {
    todo!();
}
```

should not outlive smaller of ( , )

What is wrong? What should be guaranteed?

```
fn f(s: &str, t: &str) -> &str {
    todo!();
}
```

```
error[E0106]: missing lifetime specifier
 --> src/lib.rs:1:27
  |
1 | fn f(s: &str, t: &str) -> &str {
  |         ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `s` or `t`
help: consider introducing a named lifetime parameter
  |
1 | fn f<'a>(s: &'a str, t: &'a str) -> &'a str {
  |     ++++     ++          ++
```


What does this lifetime annotation imply?

Hey caller, make sure to **pass only arguments that lives at least as long as the returned reference**. Otherwise you are doomed.

```
fn f<'a>(s: &'a str, t: &'a str) -> &'a str {
    todo!();
}
```

In other words, the returned reference is valid only as long as both parameters are alive.

Lifetime Annotations in Function Signatures

- If we want to express the following constraint: *"the returned reference will be valid as long as both the parameters are valid."*

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

The lifetime of the reference returned by the "longest()" function is **the same as the smaller of the lifetimes** of the values referred to by the function arguments.

Lifetime Annotations in Function Signatures

```
fn demo1() {
    let s1 = String::from("abcd");
    let s2 = String::from("xyz");

    let result = longest(&s1, &s2);
    println!("The longest string is {result}");
}
```

Which one is OK?

```
fn demo2() {
    let s1 = String::from("abcd");

    let result;
    {
        let s2 = String::from("xyz");
        result = longest(&s1, &s2);
    }
    println!("The longest string is {result}");
}
```

Thinking in Terms of Lifetimes

- How to specify lifetime parameters depends on what your function is doing.

```
fn first<'a>(x: &'a str, _: &str) -> &'a str {
    x
}
```

- When returning a reference from a function, the lifetime parameter for the return type needs to match the lifetime parameter for one of the parameters. Otherwise,

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

What will happen?

```
fn f<'a>(s: &'a str) -> &'a str { // 'a can be omitted
    s
}

let r;
{
    let s = String::from("hello");
    r = f(&s); // by the time `r` is assigned, it is guaranteed
               // that `s` is still valid
}
println!("r: {}", r);
```

error: `s` does not live long enough
label: borrow later used here

Multiple lifetimes

- If you have multiple references, you can use the same lifetime multiple times:

```
fn x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str {
```

This says that x, y, and return value are all alive for the same scope.

- If you wanted x and y to have different lifetimes, you can use multiple lifetime parameters:

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
```

In this example, x and y have different valid scopes, but the return value has the same lifetime as x.

Structs with references

```
#[derive(Debug)]
struct City<'a> { // City has lifetime 'a
    name: &'a str, // and name also has lifetime 'a.
    date_founded: u32,
}
```

We need to ensure that any reference to a City cannot outlive the reference to a name it contains.

- It means

"*please only take an input for name if it lives at least as long as City*".

- It does not mean:

"*I will make the input for name live as long as City*".

Lifetime declaration is mandatory for impl

```
struct Adventurer<'a> {
    name: &'a str,
    hit_points: u32,
}

impl Adventurer {
  fn take_damage(&mut self) {
    self.hit_points -= 20;
    println!("{} has {} hit points left!", self.name, self.hit_points);
  }
}
```

```
error[E0726]: implicit elided lifetime not allowed here
 --> src/lib.rs:5:6
  |
5 | impl Adventurer {
  |      ^^^^^^^^^^ expected lifetime parameter
  |
help: indicate the anonymous lifetime
  |
5 | impl Adventurer<'_> {
  |                 ++++
```


Lifetime declaration is mandatory for impl

```
struct Adventurer<'a> {
    name: &'a str,
    hit_points: u32,
}

impl<'a> Adventurer<'a> {
  fn take_damage(&mut self) {
    self.hit_points -= 20;
    println!("{} has {} hit points left!", self.name, self.hit_points);
  }
}
```

We repeat 'a twice, like on functions: impl<'a> defines a lifetime 'a, and Adventure<'a> uses it.

- The lifetime parameter declaration after `impl` and use after the type name is required.

Anonymous Lifetime (`'_`)

- "Anonymous lifetime" is an indicator that references are being used.

```
struct Adventurer<'a> {
    name: &'a str,
    hit_points: u32,
}

impl Adventurer<'_> {
    fn take_damage(&mut self) {
        self.hit_points -= 20;
        println!("{} has {} hit points left!", … );
    }
}
```


Lifetimes and Subtypes

- Lifetimes are just regions of code, and regions can be partially ordered with the *contains* (*outlives*) relationship.
- Subtyping on lifetimes is in terms of that relationship:

  if `'big: 'small` ("big *contains* small" or "big *outlives* small"), then `'big` is a *subtype* of `'small`.

subtype

supertype

```
struct OutlivesExample<'a, 'b: 'a> {
    a_str: &'a str,
    b_str: &'b str,
}
```

Lifetimes and Subtypes (Cont'd)

- If someone wants a reference that lives for `'small`, usually what they actually mean is that they want a reference that lives for *at least* `'small`.
- `'static`, the forever lifetime (i.e., the lifetime of the entire program), is **a subtype of every lifetime** because by definition it outlives everything.

```
fn bar<'a>() {
    let s: &'static str = "hi";
    let t: &'a str = s;
}
```

- Since `'static` outlives the lifetime parameter `'a`, `&'static str` is a subtype of `&'a str`.

'static Lifetime

- String literals and globals have the type `'static` lifetime : they are baked into the data segment of the final binary.

```
let x: &'static str = "Hello, world.";
```

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

- You may also encounter `'static` as part of a trait bound:

```
fn generic<T>(x: T)
where
    T: 'static {}
```

Lifetime Variances

For a type `T<'a>`, `'a` may be:

- **Covariant**: `'b: 'a` => `T<'b>: T<'a>` (default for immutable data)
- **Contravariant**: `'b: 'a => T<'a>: T<'b>` (only to `fn` parameters)
- **Invariant**: even if `'b: 'a`, nothing can be said about `T<'b>` and `T<'a>`

1. If the lifetime is present "inside" some sort of `mutable` context
   – whether inside a `&mut` reference, or *interior mutability* like `` `Cell`/`RefCell`/`Mutex` ``.
2. If the lifetime is used in multiple spots where the variances conflict.

Lifetime and Type Parameter Variances

	'a	T	U
&'a T	covariant	covariant	
&'a mut T	covariant	invariant	
Box<T>		covariant	
Vec<T>		covariant	
UnsafeCell<T>		invariant	
Cell<T>		invariant	
fn(T) -> U		contravariant	covariant
*const T		covariant	
*mut T		invariant	

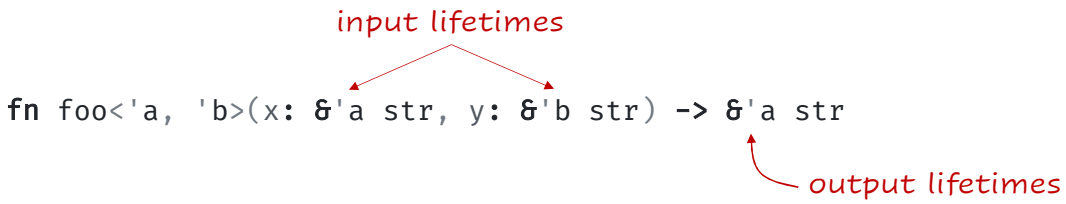
What are the variances of each lifetime parameter?

```
struct Multi<'a, 'b, 'c, 'd1, 'd2> {
    a: &'a str,                  // covariant
    b: Cell<&'b str>,            // invariant
    c: fn(&'c str) -> usize,     // contra-variant
    d: &'d1 mut &'d2 str,        // d1: covariant, d2: invariant
}

struct TwoSpots<'a> {            // invariant
    foo: &'a str,
    bar: Cell<&'a str>,
}
```


Lifetime Elision

- The patterns programmed into Rust's analysis of references are called the *lifetime elision rules*.
- The elision rules do *not* capture every possible case for lifetimes.
- Lifetimes on function or method parameters are called *input lifetimes*, and lifetimes on return values are called *output lifetimes*.



Three rules for elision

1. The compiler assigns a lifetime to each parameter that's a reference.
   ```
   fn foo<'a>(x: &'a i32)
   fn foo<'a, 'b>(x: &'a i32, y: &'b i32)
   ```

2. If there is exactly one input lifetime, that lifetime is assigned to all output lifetimes.
   ```
   fn foo<'a>(x: &'a i32) -> &'a i32
   ```

3. If there are multiple input lifetimes, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all output lifetimes.
   ```
   fn foo<'a, 'b>(&'a self, y: &'b i32) -> &'a str
   ```


Lifetimes in Functions and Traits

- Function signatures with lifetimes have a few constraints:
  - any reference must have an annotated lifetime.
  - any reference being returned must have the same lifetime as an input or be **static**.

- Additionally, note that returning references without input is banned if it would result in returning references to invalid data.

- Annotation of lifetimes in trait methods basically are similar to functions. Note that `impl` may have annotation of lifetimes too.

Lifetime Coercion

- A *longer lifetime* can be *coerced into* a *shorter one*, so a longer lifetime may be used in place of the shorter lifetime.

- This comes in the form of
  - inferred coercion by the Rust compiler, and also
  - declaring a lifetime difference:


```
// Here, Rust infers a lifetime that is as short as possible.
// The two references are then coerced to that lifetime.
fn longest<'a>(first: &'a i32, second: &'a i32) -> &'a i32 {
    if first > send { first } else { second }
}

// `<'a: 'b, 'b>` reads as lifetime `'a` is at least as long as `'b`.
// Here, we take in an `&'a i32` and return a `&'b i32` as a result of coercion.
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}

fn fooz() {
    let first = 2; // Longer lifetime

    {
        let second = 3; // Shorter lifetime

        println!("The longest is {}", longest(&first, &second));
        println!("{} is the first", choose_first(&first, &second));
    };
}
```

Tips to avoid getting too stressed about

- You can stay with owned types, use clones etc. if you want to avoid them for the time being.
- Much of the time, when the compiler wants a lifetime you will just end up writing <'a> here and there and then it will work.
  - *"Don't worry, I won't give you anything that doesn't live long enough".*
- You can explore lifetimes just a bit at a time.
  - Write some code with owned values, then make one a reference.
  - The compiler will start to complain, but also give some suggestions.
  - And if it gets too complicated, you can undo it and try again next time.

Lifetimes Summary

- Rust keeps its references under control by assigning lifetimes to them.
- They ensure that types containing references don't outlive their values, which basically prevents us from writing code that produces dangling pointers.
- In many cases, lifetime definitions can be omitted and Rust fills in the gaps for us.
- It's also possible to have types with multiple distinct lifetime parameters.
- Lifetimes are a compile-time only feature and don't exist at runtime.