

A decorative graphic on the left side of the slide. It consists of a blue parallelogram and a light green parallelogram, both tilted at an angle. The blue shape is in the foreground, and the green shape is partially behind it. They are set against a dark blue background with faint, lighter blue diagonal stripes.

RUSTikales Rust for advanced coders



Plan for today



Plan for today

1. Lifetimes



1. Lifetimes

- Recap: References are pointers to values



1. Lifetimes

The problem: Memory safety

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

1. Lifetimes

The problem: Memory safety

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

Compiler **moves** the value of **a** into **b**

1. Lifetimes

The problem: Memory safety

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

a	
content	
length	2
capacity	4

heap	
0xabc0	1
0xabc4	2
...	...



1. Lifetimes

The problem: Memory safety

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

a	
content	
length	2
capacity	4

b	
content	
length	2
capacity	4

heap	
0xabc0	1
0xabc4	2
...	...

1. Lifetimes

The problem: Memory safety

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

a	
content	
length	2
capacity	4

b	
content	
length	2
capacity	4

heap	
0xabc0	1
0xabc4	2
...	...

Ownership violation!
Memory would be freed twice at the end!

1. Lifetimes

The problem: Memory safety

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

a	
content	???
length	???
capacity	???

b	
content	
length	2
capacity	4

heap	
0xabc0	1
0xabc4	2
...	...

Solution:

- Move a into b
- Invalidate a

1. Lifetimes

The problem: Memory safety

a	
content	???
length	???
capacity	???

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

Error: a is not initialized, can't use it – It was moved



1. Lifetimes

The solution: References

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", *b);  
}
```

1. Lifetimes

The solution: References

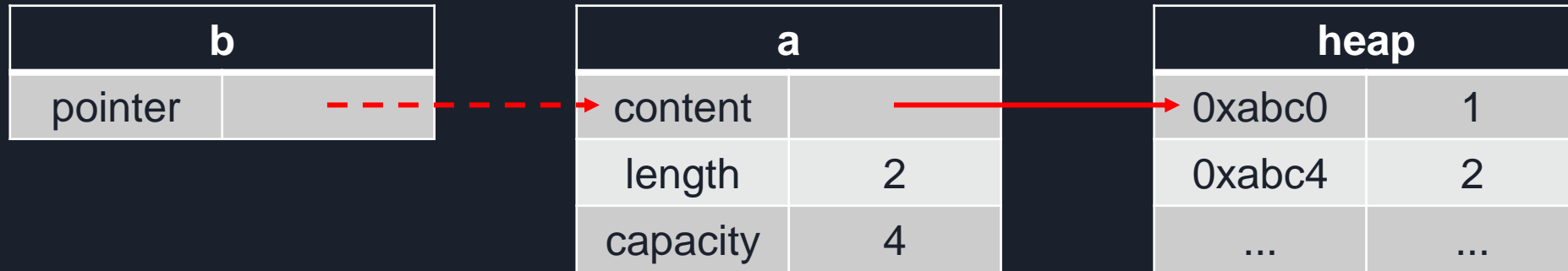
```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", *b);  
}
```



1. Lifetimes

The solution: References

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", *b);  
}
```



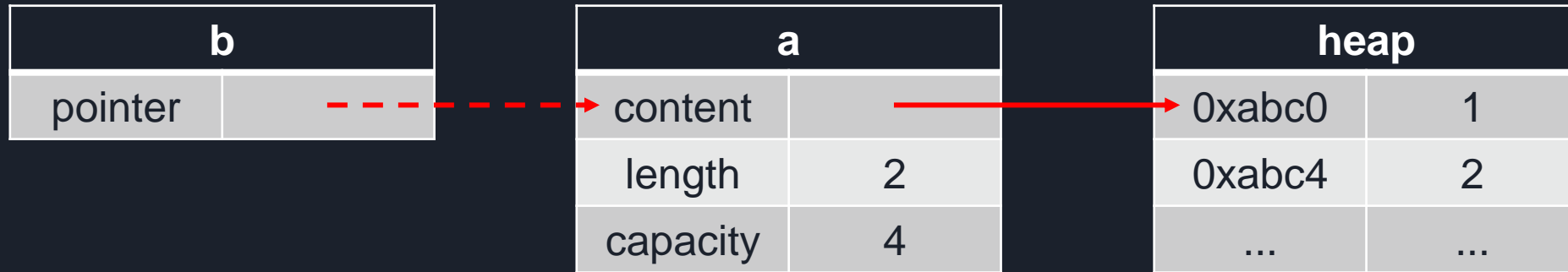
No ownership violation:

- **a** still owns the memory on the heap
- **b** does not own **a**, it just points to it

1. Lifetimes

The solution: References

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", *b);  
}
```



By **dereferencing** **b**, we get access to **a**
→ we print the vector

1. Lifetimes

The solution: References

```
fn example_1() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", *b);  
}
```



By **dereferencing** **b**, we get access to **a**
→ we print the vector
→ most of the time, derefs are implicit

1. Lifetimes

```
struct A { a: [i32; 5000] }  
fn get(a: &A) {  
    println!("a.a is {:?}", a.a);  
}  
fn example_2() {  
    let a: A = A { a: [20; 5000] };  
    get(&a);  
}
```

1. Lifetimes

```
struct A { a: [i32; 5000] } big struct!  
fn get(a: &A) {  
    println!("a.a is {:?}", a.a);  
}  
fn example_2() {  
    let a: A = A { a: [20; 5000] };  
    get(&a);  
}
```

1. Lifetimes

```
struct A { a: [i32; 5000] }  
fn get(a: &A) {  
    println!("a.a is {:?}", a.a);  
}  
fn example_2() {  
    let a: A = A { a: [20; 5000] };  
    get(&a);  
}
```

By passing a reference, we don't need to copy the original value

1. Lifetimes

```
struct B { a: &A }  
fn example_3() {  
    let a: A = A { a: [20; 5000] };  
    let b: B = B { a: &a };  
    get(b.a);  
}
```

1. Lifetimes

```
struct B { a: &A }  
fn example_3() {  
    let a: A = A { a: [20; 5000] };  
    let b: B = B { a: &a };  
    get(b.a);  
}
```

Don't want to move/copy 20KB of data every time?
Just pass a reference!



2. Next time

- Slices
- Smart Pointers

1. Lifetimes

```
struct B { a: &A }  
fn example_3() {  
    let a: A = A { a: [20; 5000] };  
    let b: B = B { a: &a };  
    get(b.a);  
}
```

Oh no

1. Lifetimes

```
error[E0106]: missing lifetime specifier
--> src\ref_ex.rs:15:15
   |
15 | struct B { a: &A }
   |               ^ expected named lifetime parameter
help: consider introducing a named lifetime parameter
15 | struct B<'a> { a: &'a A }
   |           ++++   ++
```

Well, not quite.



1. Lifetimes

- References in its simplest form are memory addresses
 - can point to any memory, to the stack, to the heap



1. Lifetimes

- References in its simplest form are memory addresses
 - can point to any memory, to the stack, to the heap
- Pointers are easy to mishandle
 - dangling pointers
 - race conditions

1. Lifetimes

```
int *f() {  
    int x = 10;  
    return &x;  
}  
int main(void) {  
    int *hehe = f();  
    printf("%d\n", *hehe);  
    somethingElse();  
    printf("%d\n", *hehe);  
}
```

3/3

1. Lifetimes

```
int *f() {  
    int x = 10;  
    return &x;  
}  
  
int main(void) {  
    int *hehe = f();  
    printf("%d\n", *hehe);  
    somethingElse();  
    printf("%d\n", *hehe);  
}
```

3/3

What does this C code print?

1. Lifetimes

```
int *f() {  
    int x = 10;  
    return &x;  
}  
  
int main(void) {  
    int *hehe = f();  
    printf("%d\n", *hehe);  
    somethingElse();  
    printf("%d\n", *hehe);  
}
```

3/3

What does this C code print?

- We can't know.
- The first `printf()` prints 10
- `somethingElse()` may **overwrite** the memory the pointer is pointing to

1. Lifetimes

```
int *f() {  
    int x = 10;  
    return &x;  
}  
  
int main(void) {  
    int *hehe = f();  
    printf("%d\n", *hehe);  
    somethingElse();  
    printf("%d\n", *hehe);  
}
```

3/3

What does this C code print?

→ We can't know.

→ The first `printf()` prints 10

→ `somethingElse()` may **overwrite** the memory the pointer is pointing to

```
void somethingElse() {  
    int a = 420;  
}
```

```
>main.exe  
10  
420
```



1. Lifetimes

- References in its simplest form are memory addresses
- Pointers are easy to mishandle
- More is required to make them memory safe, and fit for Rust's goals



1. Lifetimes

- References in its simplest form are memory addresses
- Pointers are easy to mishandle
- More is required to make them memory safe, and fit for Rust's goals
- The compiler needs to analyze when and how references are valid
 - Part One: Borrow Checker
 - Part Two: Lifetimes



1. Lifetimes

- What *is* a lifetime?



1. Lifetimes

- What *is* a lifetime?
 - Rust docs

A *lifetime* is a construct the compiler (or more specifically, its *borrow checker*) uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. While lifetimes and scopes are often referred to together, they are not the same.

<https://doc.rust-lang.org/rust-by-example/scope/lifetime.html>



1. Lifetimes

- What *is* a lifetime?

- Rust docs

- A lifetime is the time between creating and destroying a variable
 - Different from scopes
 - Used by the Borrow Checker

A *lifetime* is a construct the compiler (or more specifically, its *borrow checker*) uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. While lifetimes and scopes are often referred to together, they are not the same.



1. Lifetimes

- What *is* a lifetime?
 - Rust docs
 - Rustonomicon

Rust enforces these rules through *lifetimes*. Lifetimes are named regions of code that a reference must be valid for. Those regions may be fairly complex, as they correspond to paths of execution in the program. There may even be holes in these paths of execution, as it's possible to invalidate a reference as long as it's reinitialized before it's used again. Types which contain references (or pretend to) may also be tagged with lifetimes so that Rust can prevent them from being invalidated as well.

<https://doc.rust-lang.org/nomicon/lifetimes.html>



1. Lifetimes

- What *is* a lifetime?

- Rust docs
- Rustonomicon
 - Lifetimes enforce Borrow Checker rules
 - Named regions of code that a reference must be valid for
 - Tags for references

Rust enforces these rules through *lifetimes*. Lifetimes are named regions of code that a reference must be valid for. Those regions may be fairly complex, as they correspond to paths of execution in the program. There may even be holes in these paths of execution, as it's possible to invalidate a reference as long as it's reinitialized before it's used again. Types which contain references (or pretend to) may also be tagged with lifetimes so that Rust can prevent them from being invalidated as well.



1. Lifetimes

- What *is* a lifetime?
 - Rust docs
 - Rustonomicon
 - „Effective Rust“

The lifetime of an item on the stack is the period where that item is guaranteed to stay in the same place; in other words, this is exactly the period where a *reference* (pointer) to the item is guaranteed not to become invalid.

This starts at the point where the item is created, and extends to where it is either *dropped* (Rust's equivalent to object destruction in C++) or *moved*.

<https://www.lurklurk.org/effective-rust/lifetimes.html>



1. Lifetimes

The lifetime of an item on the stack is the period where that item is guaranteed to stay in the same place; in other words, this is exactly the period where a *reference* (pointer) to the item is guaranteed not to become invalid.

- What *is* a lifetime?

- Rust docs
- Rustonomicon
- „Effective Rust“

This starts at the point where the item is created, and extends to where it is either *dropped* (Rust's equivalent to object destruction in C++) or *moved*.

- A lifetime is the time between creating and destroying an item on the stack
- Lifetimes relate to memory locations
- Periods in time where a reference is guaranteed to be valid



1. Lifetimes

- What *is* a lifetime?
 - Rust docs
 - A lifetime is the time between creating and destroying a variable
 - Different from scopes
 - Used by the Borrow Checker
 - Rustonomicon
 - Lifetimes enforce Borrow Checker rules
 - Named regions of code that a reference must be valid for
 - Tags for references
 - „Effective Rust“
 - A lifetime is the time between creating and destroying an item on the stack
 - Lifetimes relate to memory locations
 - Periods in time where a reference is guaranteed to be valid



1. Lifetimes

Rust enforces these rules through *lifetimes*. Lifetimes are named regions of code that a reference must be valid for. Those regions may be fairly complex, as they correspond to paths of execution in the program. There may even be holes in these paths of execution, as it's possible to invalidate a reference as long as it's reinitialized before it's used again. Types which contain references (or pretend to) may also be tagged with lifetimes so that Rust can prevent them from being invalidated as well.

Introducing lifetimes

Lifetimes are what the Rust compiler uses to keep track of how long references are valid for. Checking references is one of the borrow checker's main responsibilities. Lifetimes help the borrow checker ensure that you never have invalid references.

<https://blog.logrocket.com/understanding-lifetimes-in-rust/>

When we talked about references in Chapter 4, we left out an important detail: every reference in Rust has a *lifetime*, which is the scope for which that reference is valid. Most of the time lifetimes are implicit and inferred, just like most of the time types are inferred. Similarly to when we have to annotate types because multiple types are possible, there are cases where the lifetimes of references could be related in a few different ways, so Rust needs us to annotate the relationships using generic lifetime parameters so that it can make sure the actual references used at runtime will definitely be valid.

https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/second-edition/ch10-03-lifetime-syntax.html

The lifetime of an item on the stack is the period where that item is guaranteed to stay in the same place; in other words, this is exactly the period where a *reference* (pointer) to the item is guaranteed not to become invalid.

This starts at the point where the item is created, and extends to where it is either *dropped* (Rust's equivalent to object destruction in C++) or *moved*.

Lifetimes are a way of tracking the scope of a reference to an object in memory.

In Rust, every value has one owner, and when the owner goes out of scope, the value is dropped, and its memory is freed. Lifetimes allow Rust to ensure that a reference to an object remains valid for as long as it's needed.

<https://earthly.dev/blog/rust-lifetimes-ownership-burrowing/>

A lifetime is a construct the compiler (or more specifically, its *borrow checker*) uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. While lifetimes and scopes are often referred to together, they are not the same.

1. Lifetimes

```
/// Perform the actual borrow checking.
///
/// Use `consumer_options: None` for the default behavior of returning
/// [`BorrowCheckResult`] only. Otherwise, return [`BodyWithBorrowckFacts`] according
/// to the given [`ConsumerOptions`].
#[instrument(skip(infcx, input_body, input_promoted), fields(id=?input_body.source.def_id()), level = "debug")]
fn do_mir_borrowck<'tcx>(
    infcx: &InferCtxt<'tcx>,
    input_body: &Body<'tcx>,
    input_promoted: &IndexSlice<Promoted, Body<'tcx>>,
    consumer_options: Option<ConsumerOptions>,
) -> (BorrowCheckResult<'tcx>, Option<Box<BodyWithBorrowckFacts<'tcx>>>) {
```

```
/// Computes the (non-lexical) regions from the input MIR.
///
/// This may result in errors being reported.
pub(crate) fn compute_regions<'cx, 'tcx>(
    infcx: &BorrowckInferCtxt<'_, 'tcx>,
    universal_regions: UniversalRegions<'tcx>,
    body: &Body<'tcx>,
    promoted: &IndexSlice<Promoted, Body<'tcx>>,
    location_table: &LocationTable,
    param_env: ty::ParamEnv<'tcx>,
    flow_inits: &mut ResultsCursor<'cx, 'tcx, MaybeInitializedPlaces<'cx, 'tcx>>,
    move_data: &MoveData<'tcx>,
    borrow_set: &BorrowSet<'tcx>,
    upvars: &[&ty::CapturedPlace<'tcx>],
    consumer_options: Option<ConsumerOptions>,
) -> NllOutput<'tcx> {
```

https://github.com/rust-lang/rust/blob/master/compiler/rustc_borrowck/src/nll.rs
https://github.com/rust-lang/rust/blob/master/compiler/rustc_borrowck/src/lib.rs

```
// Compute non-lexical lifetimes.
let nll::NllOutput {
    regioncx,
    opaque_type_values,
    polonius_input,
    polonius_output,
    opt_closure_req,
    nll_errors,
} = nll::compute_regions(
```

```
/// The output of `nll::compute_regions`. This includes the computed `Regioncx`
/// closure requirements to propagate, and any generated errors.
pub(crate) struct NllOutput<'tcx> {
    pub regioncx: RegionInferenceContext<'tcx>,
    pub opaque_type_values: FxIndexMap<LocalDefId, OpaqueHiddenType<'tcx>>,
    pub polonius_input: Option<Box<AllFacts>>,
    pub polonius_output: Option<Rc<PoloniusOutput>>,
    pub opt_closure_req: Option<ClosureRegionRequirements<'tcx>>,
    pub nll_errors: RegionErrors<'tcx>,
}
```

1. Lifetimes

```
/// Perform the actual borrow checking.
///
/// Use `consumer_options: None` for the default behavior of returning
/// [`BorrowCheckResult`] only. Otherwise, return [`BodyWithBorrowckFacts`] according
/// to the given [`ConsumerOptions`].
#[instrument(skip(infcx, input_body, input_promoted), fields(id=?input_body.id))
fn do_mir_borrowck<'tcx>(
    infcx: &InferCtxt<'tcx>,
    input_body: &Body<'tcx>,
    input_promoted: &IndexSlice<Promoted, Body<'tcx>>,
    consumer_options: Option<ConsumerOptions>,
) -> (BorrowCheckResult<'tcx>, Option<Box<BodyWithBorrowckFacts<'tcx>>>) {
```

```
/// Computes the (non-lexical) regions from the input MIR.
///
/// This may result in errors being reported.
pub(crate) fn compute_regions<'cx, 'tcx>(
    infcx: &BorrowckInferCtxt<'_, 'tcx>,
    universal_regions: UniversalRegions<'tcx>,
    body: &Body<'tcx>,
    promoted: &IndexSlice<Promoted, Body<'tcx>>,
    location_table: &LocationTable,
    param_env: ty::ParamEnv<'tcx>,
    flow_inits: &mut ResultsCursor<'cx, 'tcx, MaybeInitializedPlaces<'cx, 'tcx>>,
    move_data: &MoveData<'tcx>,
    borrow_set: &BorrowSet<'tcx>,
    upvars: &[&ty::CapturedPlace<'tcx>],
    consumer_options: Option<ConsumerOptions>,
) -> NllOutput<'tcx> {
```

https://github.com/rust-lang/rust/blob/master/compiler/rustc_borrowck/src/nll.rs
https://github.com/rust-lang/rust/blob/master/compiler/rustc_borrowck/src/lib.rs



```
// Compute non-lexical lifetimes.
let nll::NllOutput {
    regioncx,
    opaque_type_values,
    polonius_input,
    polonius_output,
    opt_closure_req,
    nll_errors,
} = nll::compute_regions(
```

compute_regions`. This includes the computed `Region` to propagate, and any generated errors.

```
pub(crate) struct NllOutput<'tcx> {
    pub regioncx: RegionInferenceContext<'tcx>,
    pub opaque_type_values: FxIndexMap<LocalDefId, OpaqueHiddenType<'tcx>>,
    pub polonius_input: Option<Box<AllFacts>>,
    pub polonius_output: Option<Rc<PoloniusOutput>>,
    pub opt_closure_req: Option<ClosureRegionRequirements<'tcx>>,
    pub nll_errors: RegionErrors<'tcx>,
}
```



1. Lifetimes

- Lifetimes are a construct of the compiler
 - Technically *everything* is a construct of the compiler



1. Lifetimes

- Lifetimes are a construct of the compiler
 - Technically *everything* is a construct of the compiler
- Lifetimes allow the compiler to analyze and optimize the final code
 - Lifetimes don't get into the final executable
 - Memory Safety guarantees



1. Lifetimes

- A lifetime describes two things



1. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the reference may point into



1. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the reference may point into
- Many sources only explain lifetimes as **liveness (code)**, but **aliasing (memory)** is also important
 - Alias: „variables and pointers alias if they refer to overlapping regions of memory.“

1. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

1. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

region of code where **b** must be valid

1. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

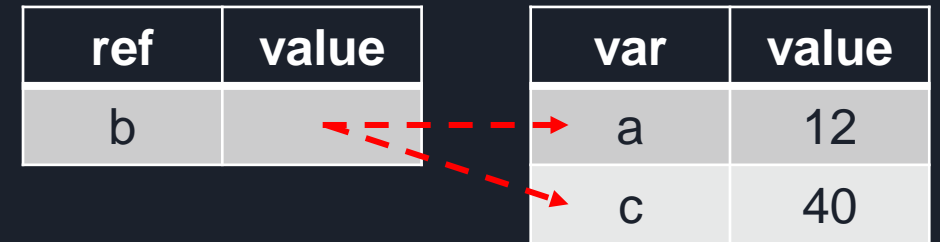
region of code where **b** must be valid

ref	value	var	value
b		a	12
		c	40

1. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

region of code where **b** must be valid



Depending on **RNG**, **b** may point to either **a** or **c**
→ When we ***b**, both memory locations must be alive

1. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

c is alive here and then dropped



1. Lifetimes

```
pub fn main()
let a: i32
let mut b
if rng() {
    let c
    b = &c;
}
println!("{}", *b);
}
```

error[E0597]: `c` does not live long enough
--> src\ex2.rs:10:13

```
    let c = 40;
    - binding `c` declared here
    b = &c;
    ^^ borrowed value does not live long enough
}
- `c` dropped here while still borrowed
println!("{}", *b);
    -- borrow later used here
```



1. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the original value must live in
- The regions of code don't have to be contiguous



1. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the original value must live in
- The regions of code don't have to be contiguous
 - References must only be valid between uses



1. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the original value must live in
- The regions of code don't have to be contiguous
 - References must only be valid between uses
 - **Non-Lexical Lifetimes**
 - The compiler is *very* good at figuring out the shortest required lifetimes for references

1. Lifetimes

```
pub fn main() {  
    let mut r: &Vec<i32>;  
    {  
        let x: Vec<i32> = vec![2];  
        r = &x;  
        println!("{:?}", *r);  
    }  
    {  
        let y: Vec<i32> = vec![2];  
        r = &y;  
        println!("{:?}", *r);  
    }  
}
```

1. Lifetimes

```
pub fn main() {  
    let mut r: &Vec<i32>;  
    {  
        let x: Vec<i32> = vec![2];  
        r = &x;  
        println!("{:?}", *r);  
    }  
    {  
        let y: Vec<i32> = vec![2];  
        r = &y;  
        println!("{:?}", *r);  
    }  
}
```

region of code where r must be valid

region of code where r must be valid

1. Lifetimes

```
pub fn main() {  
    let mut r: &Vec<i32>;  
    {  
        let x: Vec<i32> = vec![2];  
        r = &x;  
        println!("{:?}", *r);  
    }  
    {  
        let y: Vec<i32> = vec![2];  
        r = &y;  
        println!("{:?}", *r);  
    }  
}
```

region of code where r must be valid

We don't care about this section

region of code where r must be valid



1. Lifetimes

- Lifetimes get complicated when you cross the function border



1. Lifetimes

- Lifetimes get complicated when you cross the function border

```
fn f(v: &Vec<i32>) -> &i32 {  
    &v[0]  
}  
  
pub fn main() {  
    let v: Vec<i32> = vec![1, 2, 3];  
    let first: &i32 = f(&v);  
    println!("{}", *first);  
}
```

1. Lifetimes

- Lifetimes get complicated when you cross the function border

```
fn f(v: &Vec<i32>) -> &i32 {  
    &v[0] How long is this reference alive? :^)  
}  
  
pub fn main() {  
    let v: Vec<i32> = vec![1, 2, 3];  
    let first: &i32 = f(&v);  
    println!("{}", *first);  
}
```

1. Lifetimes

- Lifetimes get complicated when you cross the function border

```
fn f(v: &Vec<i32>) -> &i32 {  
    &v[0]  
}  
pub fn main() {  
    let v: Vec<i32> = vec![1, 2, 3];  
    let first: &i32 = f(&v);  
    println!("{}", *first);  
}
```

Lifetime elision, rule 2:
If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters

1. Lifetimes

- Lifetimes get complicated when you cross the function border

```
fn f(v: &Vec<i32>) -> &i32 {  
    &v[0]  
}  
  
pub fn main() {  
    let v: Vec<i32> = vec![1, 2, 3];  
    let first: &i32 = f(&v);  
    println!("{}", *first);  
}
```

points to `v[0]`, the compiler didn't complain!
`first` is valid as long as `v` is alive.

1. Lifetimes

- Lifetimes get complicated when you cross the function border

```
fn g(v1: &Vec<i32>, v2: &Vec<i32>) -> &i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    let v2: Vec<i32> = vec![4, 5];  
    let first: &i32 = g(&v1, &v2);  
    println!("{}", *first);  
}
```

1. Lifetimes

- Lifetimes get complicated when you cross the function border

```
fn g(v1: &Vec<i32>, v2: &Vec<i32>) -> &i32 { Swiggly lines of doom!  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    let v2: Vec<i32> = vec![4, 5];  
    let first: &i32 = g(&v1, &v2);  
    println!("{}", *first);  
}
```

1. Lifetimes

- Lifetimes get complicated when you cross the function border

```
error[E0106]: missing lifetime specifier
```

```
--> src\ex3.rs:10:39
```

```
10 | fn g(v1: &Vec<i32>, v2: &Vec<i32>) -> &i32 {
```

```
          ^ 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 `v1` or `v2`
```

```
help: consider introducing a named lifetime parameter
```

```
10 | fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {
```

```
      ++++
```

```
      ++
```

```
      ++
```

```
      ++
```



1. Lifetimes

- Lifetimes get complicated when you cross the function border
- What's the problem?



1. Lifetimes

- Lifetimes get complicated when you cross the function border
- What's the problem?
 - According to the code model, both vectors are borrowed for some amount of time



1. Lifetimes

- Lifetimes get complicated when you cross the function border
- What's the problem?
 - According to the code model, both vectors are borrowed for some amount of time
 - According to the memory model, both vectors are somewhere in memory



1. Lifetimes

- Lifetimes get complicated when you cross the function border
- What's the problem?
 - According to the code model, both vectors are borrowed for some amount of time
 - According to the memory model, both vectors are somewhere in memory
 - The compiler knows* that the return value is from those vectors



1. Lifetimes

- Lifetimes get complicated when you cross the function border
- What's the problem?
 - According to the code model, both vectors are borrowed for some amount of time
 - According to the memory model, both vectors are somewhere in memory
 - The compiler knows* that the return value is from those vectors
 - It just doesn't know which vector it is from

1. Lifetimes

- Lifetimes get complicated when you cross the function border
- What's the problem?
 - According to the code model, both vectors are borrowed for some amount of time
 - According to the memory model, both vectors are somewhere in memory
 - The compiler knows* that the return value is from those vectors
 - It just doesn't know which vector it is from
 - Randomly picking a lifetime doesn't work, because the lifetimes may be different

```
fn a(v: &Vec<i32>) -> &i32 {  
    let v1: Vec<i32> = vec![4, 5];  
    g(v1: v, v2: &v1)  
}
```

1. Lifetimes

- Lifetimes get complicated when you cross the function border
- What's the problem?
 - According to the code model, both vectors are borrowed for some amount of time
 - According to the memory model, both vectors are somewhere in memory
 - The compiler knows* that the return value is from those vectors
 - It just doesn't know which vector it is from
 - Randomly picking a lifetime doesn't work, because the lifetimes may be different

```
fn a(v: &Vec<i32>) -> &i32 {  
    let v1: Vec<i32> = vec![4, 5];  
    g(v1: v, v2: &v1) g could return a reference to elements  
    of v1, which is dropped here  
}
```



1. Lifetimes

- Lifetimes get complicated when you cross the function border
- What's the problem?
 - According to the code model, both vectors are borrowed for some amount of time
 - According to the memory model, both vectors are somewhere in memory
 - The compiler knows* that the return value is from those vectors
 - It just doesn't know which vector it is from
 - Randomly picking a lifetime doesn't work, because the lifetimes may be different
- We have to provide more information about the lifetimes

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    let v2: Vec<i32> = vec![4, 5];  
    let first: &i32 = g(&v1, &v2);  
    println!("{}", *first);  
}
```

1. Lifetimes

named lifetime parameter

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    let v2: Vec<i32> = vec![4, 5];  
    let first: &i32 = g(&v1, &v2);  
    println!("{}", *first);  
}
```

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}
```

This tells the compiler:

- There exists a region of memory **'a** such that:
 - The original **v1** is in that region
 - The original **v2** is in that region
 - The ***return value** is in that region
 - **Union** of all three regions

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}
```

This tells the compiler:

- There exists a region of memory **'a** such that:
 - The original **v1** is in that region
 - The original **v2** is in that region
 - The ***return value** is in that region
 - **Union** of all three regions

'v1 [1 – 2 – 3]

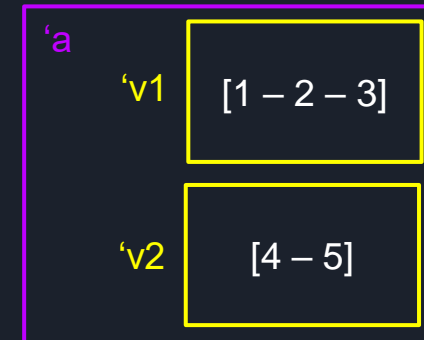
'v2 [4 – 5]

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}
```

This tells the compiler:

- There exists a region of memory **'a** such that:
 - The original **v1** is in that region
 - The original **v2** is in that region
 - The ***return value** is in that region
 - **Union** of all three regions



1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}
```

This tells the compiler:

- There exists a region of memory **'a** such that:
 - The original **v1** is in that region
 - The original **v2** is in that region
 - The ***return value** is in that region
 - **Union** of all three regions
- There exists a region of code **'a** such that:
 - The reference **v1** is live in that region
 - The reference **v2** is live in that region
 - The **return value** is live in that region
 - **Intersection** of all three regions

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    let v2: Vec<i32> = vec![4, 5];  
    let first: &i32 = g(&v1, &v2);  
    println!("{}", *first);  
}
```

The intersection of all three lifetimes is this block

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    let v2: Vec<i32> = vec![4, 5];  
    let first: &i32 = g(&v1, &v2);  
    println!("{}", *first);  
}
```

2/3

Why, if we only borrow in this line?
Shouldn't the region only be this line?

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    let v2: Vec<i32> = vec![4, 5];  
    let first: &i32 = g(&v1, &v2);  
    println!("{}", *first);  
}
```

2/3

Why, if we only borrow in this line?
Shouldn't the region only be this line?

`first` is a reference into those vectors, so **by proxy** we also extended the borrow of `v1` and `v2` → **Aliasing**

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    let v2: Vec<i32> = vec![4, 5];  
    let first: &i32 = g(&v1, &v2);  
    println!("{}", *first);  
}
```

2/3

Why, if we only borrow in this line?
Shouldn't the region only be this line?

Mutably borrowing either **v1** or **v2** in
this block would throw a compiler error

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}
```

This tells the compiler:

- There exists a region of memory **'a** such that:
 - The original **v1** is in that region
 - The original **v2** is in that region
 - The ***return value** is in that region
 - **Union** of all three regions
- There exists a region of code **'a** such that:
 - The reference **v1** is live in that region
 - The reference **v2** is live in that region
 - The **return value** is live in that region
 - **Intersection** of all three regions
- It does **NOT** mean:
 - **v1** and **v2** have the same lifetime

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(v: &Vec<i32>) {
    let v1: Vec<i32> = vec![3, 4];
    let first: &i32 = g(v1: v, v2: &v1);
    println!("{}", *first);
}

pub fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    a(&v1);
}
```


1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
fn a(v: &Vec<i32>) {  
    let v1: Vec<i32> = vec![3, 4];  
    let first: &i32 = g(v1: v, v2: &v1);  
    println!("{}", *first);  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    a(&v1);  
}
```

It is easy to see that `v` lives longer than `&v1`, yet this still works

1. Lifetimes

```
fn g<'a>(v1: &'a Vec<i32>, v2: &'a Vec<i32>) -> &'a i32 {  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}  
  
fn a(v: &Vec<i32>) {  
    let v1: Vec<i32> = vec![3, 4];  
    let first: &i32 = g(v1: v, v2: &v1);  
    println!("{}", *first);  
}  
  
pub fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
    a(&v1);  
}
```

The intersection of all three lifetimes is this block



1. Lifetimes

- You can list as many lifetime parameters as you want, as long as you use them

1. Lifetimes

```
fn g<'v1, 'v2, 'r>(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)  
-> &'r i32  
where  
    'v1: 'r,  
    'v2: 'r  
{  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}
```

1. Lifetimes

```
fn g<'v1, 'v2, 'r>(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)  
-> &'r i32
```

v1 and v2 are totally unrelated, there's no connection

where

```
    'v1: 'r,  
    'v2: 'r  
{  
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }  
}
```

1. Lifetimes

```
fn g<'v1, 'v2, 'r>(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)  
-> &'r i32
```

```
where
```

```
    'v1: 'r,
```

```
    'v2: 'r
```

However, both 'v1 and 'v2 must **outlive** 'r

```
{
```

```
    if v1.len() > v2.len() { &v1[0] }
```

```
    else { &v2[0] }
```

```
}
```

1. Lifetimes

```
fn g<'v1, 'v2, 'r>(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)  
-> &'r i32
```

where

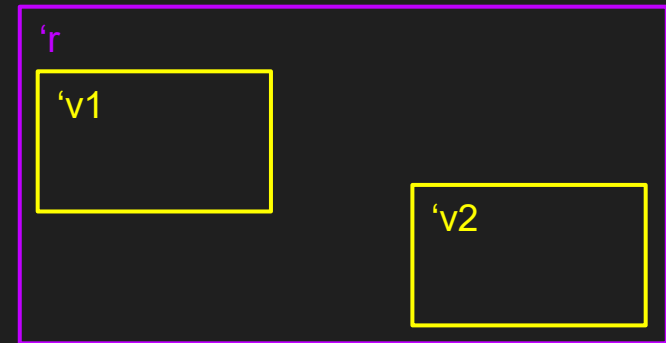
```
'v1: 'r,  
'v2: 'r
```

```
{
```

```
    if v1.len() > v2.len() { &v1[0] }  
    else { &v2[0] }
```

```
}
```

regions of code → Intersection of all regions → 'r
regions of memory → Union of all regions → 'v1&'v2
→ For a short period, 'r can borrow data from both vectors



1. Lifetimes

```
fn g<'v1, 'v2, 'r>(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)  
-> &'r i32
```

```
where
```

This is also what the compiler does in the background when we only specify `'a!`
→ Subtyping and Variance

```
    'v1: 'r,  
    'v2: 'r  
{  
    if v1.len() > v2.len() { &v1[0] }  
    else                    { &v2[0] }  
}
```


1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---

start of 'v1

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner		
	first		

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first		

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first		

start of 'v2

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first		
g	v1	'v1	orig
	v2	'v2	inner
	result		

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first		
g	v1	'v1	orig
	v2	'v2	inner
	result		

'r is the smallest lifetime such that 'v1 and 'v2 are both also alive inside it

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first		
g	v1	'v1	orig
	v2	'v2	inner
	result		

'r is the smallest lifetime such that 'v1 and 'v2 are both also alive inside it
→ orig lives in e3 → 'v1 live in e3

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
	outer	'v1	orig
	inner	[3,4]	---
g	first		
	v1	'v1	orig
	v2	'v2	inner
	result		

'r is the smallest lifetime such that 'v1 and 'v2 are both also alive inside it
→ orig lives in e3 → 'v1 live in e3
→ inner lives in a → 'v2 live in a

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
	outer	'v1	orig
	inner	[3,4]	---
g	first		
	v1	'v1	orig
	v2	'v2	inner
	result		

'r is the smallest lifetime such that 'v1 and 'v2 are both also alive inside it

→ orig lives in e3 → 'v1 live in e3

→ inner lives in a → 'v2 live in a

→ inner does **not** live in e3 → 'v2 stuck in a

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first		
g	v1	'v1	orig
	v2	'v2	inner
	result	'r	both

'r is the smallest lifetime such that 'v1 and 'v2 are both also alive inside it

→ orig lives in e3 → 'v1 live in e3

→ inner lives in a → 'v2 live in a

→ inner does **not** live in e3 → 'v2 stuck in a

→ return reference 'r **alive** in a

→ not allowed to escape to e3

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first	'r	both

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first	'r	both

This is 'r, first can be from both inner and orig here

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
	outer	'v1	orig
a	inner	[3,4]	---
	first	'r	both

Use is valid, both original values are alive

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

end of 'v2 and 'r

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---
a	outer	'v1	orig
	inner	[3,4]	---
	first	'r	both

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

Stackframe			
function	variable	value	refers to
e3	orig	[1,2,3]	---

1. Lifetimes

```
fn g<'v1, 'v2, 'r>
(v1: &'v1 Vec<i32>, v2: &'v2 Vec<i32>)
-> &'r i32 where 'v1: 'r, 'v2: 'r {
    if v1.len() > v2.len() { &v1[0] }
    else { &v2[0] }
}

fn a(outer: &Vec<i32>) {
    let inner = vec![3, 4];
    let first = g(outer, &inner);
    println!("{}", *first);
}

fn e3() {
    let orig = vec![1, 2, 3];
    a(&orig);
}
```

end of 'v1

Stackframe

function	variable	value	refers to
----------	----------	-------	-----------



1. Lifetimes

- Structs can also have references as fields
 - For that, you *have to* specify lifetime parameters



1. Lifetimes

- Structs can also have references as fields
 - For that, you *have to* specify lifetime parameters
 - Structs are way more flexible than functions, you can use them literally everywhere
 - Inferring lifetimes is way harder :^)



1. Lifetimes

- Structs can also have references as fields
- Many situations arise when you need that, and many where you want that



1. Lifetimes

- Structs can also have references as fields
- Many situations arise when you need that, and many where you want that
 - Less memory usage
 - Less computation (cloning is expensive)



1. Lifetimes

- Structs can also have references as fields
- Many situations arise when you need that, and many where you want that
- At the same time, in many situations you do not want that



1. Lifetimes

- Structs can also have references as fields
- Many situations arise when you need that, and many where you want that
- At the same time, in many situations you do not want that
 - Explicit lifetimes makes your code harder to use and maintain
 - All lifetimes have to line up, in every case, or it falls apart quickly



1. Lifetimes

- Structs can also have references as fields
- Many situations arise when you need that, and many where you want that
- At the same time, in many situations you do not want that
 - Explicit lifetimes makes your code harder to use and maintain
 - Lifetime infection will make your code unreadable

```
struct A<'b> {  
    b: B<'b>,  
}  
  
struct B<'b> {  
    i: &'b i32,  
}
```



1. Lifetimes

- Structs can also have references as fields
- Many situations arise when you need that, and many where you want that
- At the same time, in many situations you do not want that
 - Explicit lifetimes makes your code harder to use and maintain
 - Lifetime infection will make your code unreadable

A itself doesn't need a lifetime parameter!
But you have to propagate it, it's now also
A's business

```
struct A<'b> {  
    b: B<'b>,  
}  
struct B<'b> {  
    i: &'b i32,  
}
```



1. Lifetimes

- Structs can also have references as fields
- Many situations arise when you need that, and many where you want that
- At the same time, in many situations you do not want that
- If things get out of hand:
 - Read about Ownership again, do they need references? Isn't it better to own the data?
 - Use Smart Pointers

1. Lifetimes

```
struct Flags {  
    do_stuff: bool,  
    something: bool,  
}
```

1 implementation

```
struct Parser<'f, 's> {  
    flags: &'f Flags,  
    original: &'s str,  
}
```

1. Lifetimes

```
struct Flags {  
    do_stuff: bool,  
    something: bool,  
}
```

Two lifetime parameters

1 implementation

```
struct Parser<'f, 's> {  
    flags: &'f Flags,  
    original: &'s str,  
}
```

1. Lifetimes

```
struct Flags {  
    do_stuff: bool,  
    something: bool,  
}
```

1 implementation

```
struct Parser<'f, 's> {  
    flags: &'f Flags,  
    original: &'s str,  
}
```

Two lifetime parameters

→ The original `Flags` struct has to `outlive` the `Parser`

1. Lifetimes

```
struct Flags {  
    do_stuff: bool,  
    something: bool,  
}
```

1 implementation

```
struct Parser<'f, 's> {  
    flags: &'f Flags,  
    original: &'s str,  
}
```

Two lifetime parameters

- The original Flags struct has to outlive the Parser
- The original source text has to outlive the Parser

1. Lifetimes

```
struct Flags {  
    do_stuff: bool,  
    something: bool,  
}
```

1 implementation

```
struct Parser<'f, 's> {  
    flags: &'f Flags,  
    original: &'s str,  
}
```

Two lifetime parameters

- The original Flags struct has to outlive the Parser
- The original source text has to outlive the Parser
- The original Flags struct is unrelated to the original source text

1. Lifetimes

```
impl<'f, 's> Parser<'f, 's> {  
    fn new(flags: &'f Flags, original: &'s str) -> Self {  
        Self { flags, original }  
    }  
    fn next(&mut self) -> Option<&str> {  
        let tkn: &str = self.original.split_whitespace().next()?;  
        self.original = &self.original[(tkn.len()+1)..];  
        Some(tkn)  
    }  
}
```

1. Lifetimes

```
impl<'f, 's> Parser<'f, 's> { Declares lifetime parameters for the whole struct
    fn new(flags: &'f Flags, original: &'s str) -> Self {
        Self { flags, original }
    }
    fn next(&mut self) -> Option<&str> {
        let tkn: &str = self.original.split_whitespace().next()?;
        self.original = &self.original[(tkn.len()+1)..];
        Some(tkn)
    }
}
```

1. Lifetimes

```
impl<'f, 's> Parser<'f, 's> { Declares lifetime parameters for the whole struct
    fn new(flags: &'f Flags, original: &'s str) -> Self {
        Self { flags, original } Can be used in all methods and associated functions
        → No need to annotate new() with them again
    }
    fn next(&mut self) -> Option<&str> {
        let tkn: &str = self.original.split_whitespace().next()?;
        self.original = &self.original[(tkn.len()+1)..];
        Some(tkn)
    }
}
```

1. Lifetimes

```
impl<'f, 's> Parser<'f, 's> {  
    fn new(flags: &'f Flags, original: &'s str) -> Self {  
        Self { flags, original }  
    }  
    fn next(&mut self) -> Option<&str> {  
        let tkn: &str = self.original.split_whitespace().next()?;  
        self.original = &self.original[(tkn.len()+1)..];  
        Some(tkn)  
    }  
}
```

Lifetime elision, rule 3:

if there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because this is a method, the lifetime of `self` is assigned to all output lifetime parameters

1. Lifetimes

```
pub fn main() {
    let args: Args = std::env::args();
    let flags: Flags = Flags::parse_arg(args);
    let content: String = match fs::read_to_string(path: "./foo.txt") {
        Ok(content: String) => content,
        Err(e: Error) => panic!("{}", e)
    };
    let mut parser: Parser = Parser::new(&flags, original: &content);
    while let Some(tkn: &str) = parser.next() {
        println!("{}", tkn);
    }
}
```

1. Lifetimes

```
pub fn main() {  
    let args: Args = std::env::args();  
    let flags: Flags = Flags::parse_arg(args);  
    let content: String = match fs::read_to_string(path: "./foo.txt") {  
        Ok(content: String) => content,  
        Err(e: Error) => panic!("{}", e)  
    };  
    let mut parser: Parser = Parser::new(&flags, original: &content);  
    while let Some(tkn: &str) = parser.next() {  
        println!("{}", tkn);  
    }  
}
```

flags are alive here

1. Lifetimes

```
pub fn main() {  
    let args: Args = std::env::args();  
    let flags: Flags = Flags::parse_arg(args);  
    let content: String = match fs::read_to_string(path: "./foo.txt") {  
        Ok(content: String) => content,  
        Err(e: Error) => panic!("{}", e)  
    };  
    let mut parser: Parser = Parser::new(&flags, original: &content);  
    while let Some(tkn: &str) = parser.next() {  
        println!("{}", tkn);  
    }  
}
```

flags are alive here
content is alive here

1. Lifetimes

```
pub fn main() {  
    let args: Args = std::env::args();  
    let flags: Flags = Flags::parse_arg(args);  
    let content: String = match fs::read_to_string(path: "./foo.txt") {  
        Ok(content: String) => content,  
        Err(e: Error) => panic!("{}", e)  
    };  
    let mut parser: Parser = Parser::new(&flags, original: &content);  
    while let Some(tkn: &str) = parser.next() {  
        println!("{}", tkn);  
    }  
}
```

flags are alive here
content is alive here
parser is alive here

1. Lifetimes

```
pub fn main() {  
    let args: Args = std::env::args();  
    let flags: Flags = Flags::parse_arg(args);  
    let content: String = match fs::read_to_string(path: "./foo.txt") {  
        Ok(content: String) => content,  
        Err(e: Error) => panic!("{}", e)  
    };  
    let mut parser: Parser = Parser::new(&flags, original: &content);  
    while let Some(tkn: &str) = parser.next() {  
        println!("{}", tkn); tkn is a reference into the content  
    }  
}
```

flags are alive here
content is alive here
parser is alive here
tkn is alive here

1. Lifetimes

```
pub fn main() {      Borrowed data does not outlive original → Lifetimes valid, Borrow Checker passed!
    let args: Args = std::env::args();
    let flags: Flags = Flags::parse_arg(args);
    let content: String = match fs::read_to_string(path: "./foo.txt") {
        Ok(content: String) => content,
        Err(e: Error) => panic!("{}", e)
    };
    let mut parser: Parser = Parser::new(&flags, original: &content);
    while let Some(tkn: &str) = parser.next() {
        println!("{}", tkn); tkn is a reference into the content
    }
}
```

flags are alive here
content is alive here
parser is alive here
tkn is alive here



2. Next time

- Slices
- Smart Pointers