CS 241: Systems Programming
Lecture 21. Lifetimes

Fall 2025

Prof. Stephen Checkoway

Data must live longer than references

Data must live longer than references

```rust
fn main() {
    let some_ref: &i32 = {
        let x = 28;
        &x
    };
    println!("The value of x is {some_ref}");
}
```

Data must live longer than references

```
fn main() {
    let some_ref: &i32 = {
        let x = 28;
        &x
    };
    println!("The value of x is {some_ref}");
}
error[E0597]: `x` does not live long enough
 --> lifetimes.rs:4:9
  |
2 |     let some_ref: &i32 = {
  |         -------- borrow later stored here
3 |         let x = 28;
  |             - binding `x` declared here
4 |         &x
  |         ^^ borrowed value does not live long enough
5 |     };
  |     - `x` dropped here while still borrowed
```

Checking references in a function

Inside a single function, the BorrowChecker checks that all data outlive references to the data

The **lifetime of data** is how long the data will live

- until the end of the function
- until the end of a block
- until the end of the program
- until it is moved (e.g., by calling a function that takes ownership)

The **lifetime of a reference** is from its creation until its last use

Checking references between functions

Passing a reference to a function or returning one makes it more complicated to check lifetimes

Options

- Whole program analysis
- **Annotations on functions specifying lifetime information**

Whole-program analysis has some drawbacks (including compilation time and nonlocal errors)

Consider this function which returns a reference to an i32 named x

```rust
fn foo() -> &i32 {
    // Some code here
    &x
}
```

How long must x live for this code to avoid having a reference to data that is no longer alive?

A. Until the end of foo

B. Until the end of the block x is declared in

C. Until the end of the program

D. Until x is moved

Returning a reference

We can’t return a reference to a local variable

We must return a reference to something else

- A global variable
- A literal value (like &83 or `"a literal string"`)
- **Some data passed by reference to the function**

This is the interesting case!

Trying to return a reference

Trying to return a reference

This function returns a string literal, but it doesn’t actually compile

```
fn day_of_week(day: i32) -> &str {
    match day {
        0 => "Sunday",
        1 => "Monday",
        2 => "Tuesday",
        3 => "Wednesday",
        4 => "Thursday",
        5 => "Friday",
        6 => "Saturday",
        _ => panic!("Not a valid day of the week!"),
    }
}
```

Trying to return a reference

This function returns a string literal, but it doesn’t actually compile

```
fn day_of_week(day: i32) -> &str {
    match day {
        0 => "Sunday",
        1 => "Monday",
        2 => "Tuesday",
        3 => "Wednesday",
        4 => "Thursday",
        5 => "Friday",
        6 => "Saturday",
        _ => panic!("Not a valid day of the week!"),
    }
}
```

```
error[E0106]: missing lifetime specifier
 --> lifetimes.rs:9:29
  |
9 | fn day_of_week(day: i32) -> &str {
  |                             ^ expected named lifetime parameter
```

'static lifetime specifier

```
error[E0106]: missing lifetime specifier
 --> lifetimes.rs:9:29
  |
9 | fn day_of_week(day: i32) -> &str {
  |                             ^ expected named lifetime
parameter
  |
  = help: this function's return type contains a borrowed
value, but there is no value for it to be borrowed from
help: consider using the `'static` lifetime
  |
9 | fn day_of_week(day: i32) -> &'static str {
  |                              +++++++
```

Trying to return a reference

```rust
fn day_of_week(day: i32) -> &'static str {
    match day {
        0 => "Sunday",
        1 => "Monday",
        2 => "Tuesday",
        3 => "Wednesday",
        4 => "Thursday",
        5 => "Friday",
        6 => "Saturday",
        _ => panic!("Not a valid day of the week!"),
    }
}
```

The 'static in &'static str is a lifetime specifier that indicates the reference is valid until the end of the program

Returning a reference to non-string literals

```
fn this_seems_useless_but_it_works() -> &'static i32 {
    &83
}
```

Literals are valid for the entire program

Returning a reference to a global variable

```rust
static SOME_GLOBAL_INT: i32 = 42;

fn foo(which: bool) -> &'static i32 {
    static GLOBAL_BUT_ONLY_ACCESSIBLE_IN_FOO: i32 = 8;
    if which {
        &SOME_GLOBAL_INT
    } else {
        &GLOBAL_BUT_ONLY_ACCESSIBLE_IN_FOO
    }
}

fn main() {
    println!("{} {}", foo(false), foo(true));
}
```

References based on arguments

```
  = help: this function's return type contains a
borrowed value, but there is no value for it to be
borrowed from
```

The error message’s help hints that to return a non 'static reference, the function needs **some other data** to base the reference on

That other data must come from function arguments

Consider this function which returns a reference to a &str

```rust
fn foo(arg: &str) -> &str {
    todo!()
}
```

What can foo return?

A. Only string literals

B. arg or string literals

C. string literals or slices of string literals

D. arg or slices of arg

E. arg, slices of arg, string literals, or slices of string literals

Reference arguments

```
fn foo(arg: &i32) -> &i32
```

Consider this code

```
fn main() {
    let r = {
        let x = 1005;
        foo(&x)
    };
    println!("{r}");
}
```

If this code ran, would it be safe? Could foo() return a reference that doesn’t live long enough?

Lifetime parameters

Lifetime parameters are a way to relate the **lifetimes of returned references** to the **lifetimes of reference arguments**

Lifetime parameters

- Start with a `'` (e.g., `'a`, `'b`, `'c`, `'foo`)
- Are specified along with generic arguments inside <angle brackets>

Lifetime parameter example

Declares a lifetime parameter

```
fn foo<'a>(arg: &'a i32) -> &'a i32 {
    todo!()
}
```

Specifies that arg has the lifetime 'a

Specifies that the return value lives at least as long as lifetime 'a

When `foo(&x)` is called, Rust uses the lifetime of x for `'a` so the returned reference can be used as long as x is alive

```
fn foo<'a>(arg: &'a i32) -> &'a i32 { /* … */ }

fn main() {
    let r: &i32 = {
        let x = 1005;
        foo(&x)
    };
    println!("{r}");
}
```

Is this code valid? Put another way, can the compiler guarantee that the r reference doesn't outlive the data it points to? Why or why not?

A. The code is valid

B. The code is invalid

C. It depends on what foo returns

```rust
fn foo<'a>(arg: &'a i32) -> &'a i32 { /* … */ }

fn main() {
    let r: &i32 = { foo(&1005) };
    println!("{r}");
}
```

Can the compiler guarantee that the r reference doesn’t outlive the data it points to? What is the lifetime of the returned reference?

A. Yes. The lifetime is until the end of main

B. Yes. The lifetime is until the end of the program

C. No. The lifetime is until the end of block foo() is called in which isn’t long enough

D. No. The lifetime of &1005 isn’t 'static

Returning a reference based on a reference argument

Returning a reference based on a reference argument

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Returning a reference based on a reference argument

```
fn first_word<'a>(s: &'a str) -> &'a str {
    if let Some(idx) = s.find(' ') {
        &s[..idx]
    } else {
        s
    }
}
```

Returning a reference based on a reference argument

```rust
fn first_word<'a>(s: &'a str) -> &'a str {
    if let Some(idx) = s.find(' ') {
        &s[..idx]
    } else {
        s
    }
}
```

In both cases, first_word() returns a reference (a string slice) that is valid for as long as the string pointed to by s is valid

Returning a reference based on a reference argument

```rust
fn first_word<'a>(s: &'a str) -> &'a str {
    if let Some(idx) = s.find(' ') {
        &s[..idx]
    } else {
        s
    }
}

fn main() {
    let sentence = String::from("This is complicated!");
    let word = first_word(&sentence);
    println!("{word}");
}
```

In both cases, first_word() returns a reference (a string slice) that is valid for as long as the string pointed to by s is valid

Returning reference to a struct member

Returning reference to a struct member

```
struct Foo {
    name: String,
}
```

Returning reference to a struct member

```
struct Foo {
    name: String,
}

impl Foo {
    fn name<'a>(&'a self) -> &'a str {
        &self.name
    }
```

Returning reference to a struct member

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

impl Foo {
    fn name<'a>(&'a self) -> &'a str {
        &self.name
    }

    fn name_mut<'a>(&'a mut self) -> &'a mut String {
        &mut self.name
    }
}
```

Returning reference to a struct member

```
struct Foo {
    name: String,
}

impl Foo {
    fn name<'a>(&'a self) -> &'a str {
        &self.name
    }

    fn name_mut<'a>(&'a mut self) -> &'a mut String {
        &mut self.name
    }
}
```

We can return mutable references

Returning reference to a struct member

```
struct Foo {
    name: String,
}

impl Foo {
    fn name<'a>(&'a self) -> &'a str {
        &self.name
    }

    fn name_mut<'a>(&'a mut self) -> &'a mut String {
        &mut self.name
    }
}

fn main() {
    let mut x = Foo { name: String::from("Thing") };
    x.name_mut().push_str(" One");
    println!("{}", x.name());
}
```

We can return mutable references

```
fn append<'a,'b>(target: &'a mut String, s: &’b str)
  -> &'X mut String
{
  target.push_str(s);
  target
}
```

What lifetime specifier does X need to be?

A. 'a

B. 'b

C. ‘static

D. Something else

Multiple lifetime parameters

```
fn append<'a, 'b>(target: &'a mut String, s: &'b str) -> &'a mut String {
    target.push_str(s);
    target
}

fn main() {
    let mut s = String::new();

    append(append(&mut s, "foo"), "bar");
    println!("{s}");
}
```

Prints out: foobar

Using the same lifetime parameter for multiple reference parameters

```
fn smallest<'a>(x: &'a mut i32, y: &'a mut i32) -> &'a mut i32 {
    if *x < *y {
        x
    } else {
        y
    }
}
```

When called, `'a` will be the smallest lifetime satisfied by both `x` and `y`

The return value must live as long as both of them

Implicit lifetime parameters or lifetime elision

When the function has one reference argument and one reference return value, no explicit lifetime parameter is required

- `fn foo(x: &i32) -> &i32`
- The lifetime of the return value is the same as the lifetime of the argument

If the function is a method with a `&self` or `&mut self` parameter, then the lifetime of the returned references is the lifetime of `self`

Otherwise the lifetime parameters must be specified

first_word with lifetime elision

```
fn first_word(s: &str) -> &str {
    if let Some(idx) = s.find(' ') {
        &s[..idx]
    } else {
        s
    }
}
```

Methods with lifetime elision

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

impl Foo {
    fn name(&self) -> &str {
        &self.name
    }

    fn name_mut(&mut self) -> &mut String {
        &mut self.name
    }
}
```

Which of these methods requires explicit lifetimes?

A. fn foo(i: &i32) -> &i32

B. fn foo(i: &i32, j: &i32) -> &i32

C. fn foo(&self) -> &i32

D. None of the above

Structs containing references

```
struct CounterRef<'a> {
    counter: &'a mut usize
}

impl<'a> CounterRef<'a> {
    fn count_zeros(&mut self,
                   nums: &[i32])
    {
        for num in nums {
            if *num == 0 {
                *self.counter += 1;
            }
        }
    }
}
```

Structs containing references

```
struct CounterRef<'a> {
    counter: &'a mut usize
}

impl<'a> CounterRef<'a> {
    fn count_zeros(&mut self,
                   nums: &[i32])
    {
        for num in nums {
            if *num == 0 {
                *self.counter += 1;
            }
        }
    }
}
```

```
fn main() {
    let mut count: usize = 0;
    let mut cr = CounterRef {
        counter: &mut count
    };

    cr.count_zeros(&[0, 3, 0, 4]);
    cr.count_zeros(&[-1, 0, 1, 2]);
    println!("{count}");
}
```

What will this print?

```
struct CounterRef<'a> {
    counter: &'a mut usize
}

impl<'a> CounterRef<'a> {
    fn count_zeros(&mut self,
                   nums: &[i32])
    {
        for num in nums {
            if *num == 0 {
                *self.counter += 1;
            }
        }
    }
}
```

```
fn main() {
    let mut count: usize = 0;
    let mut cr = CounterRef {
        counter: &mut count
    };

    cr.count_zeros(&[0, 3, 0, 4]);
    cr.count_zeros(&[-1, 0, 1, 2]);
    println!("{count}");
}
```

A. 1

B. 2

C. 3

D. Does not compile