CoE 164

Computing Platforms

03c: Lifetimes

REFERENCES

A **reference** to a data enables *lending* of such data. When a reference is "passed" to a function or any variable, the function *does not* own the data.

During the course of the program, references should point to *valid* data.

REFERENCES: DANGLES

If the reference points to data that will become out of scope soon, it is called a **dangling reference**. Rust implements a *borrow checker* to ensure that such references will never happen.

Example

```
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```

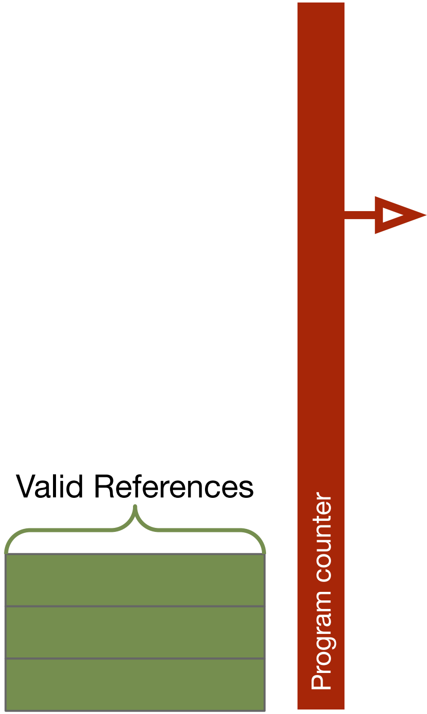
REFERENCES: DANGLES

Example

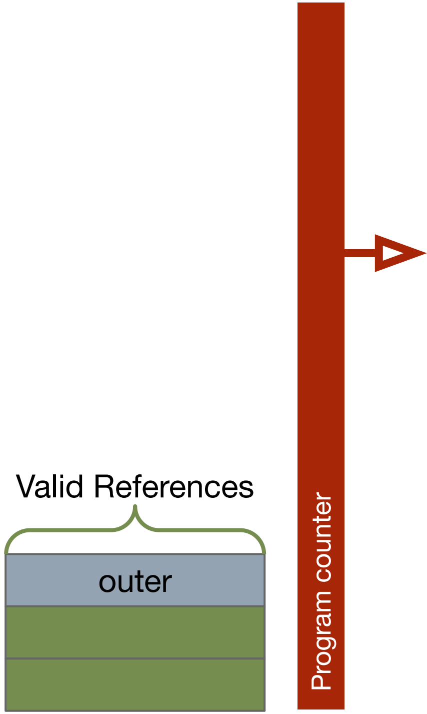
```
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```



REFERENCES: DANGLES



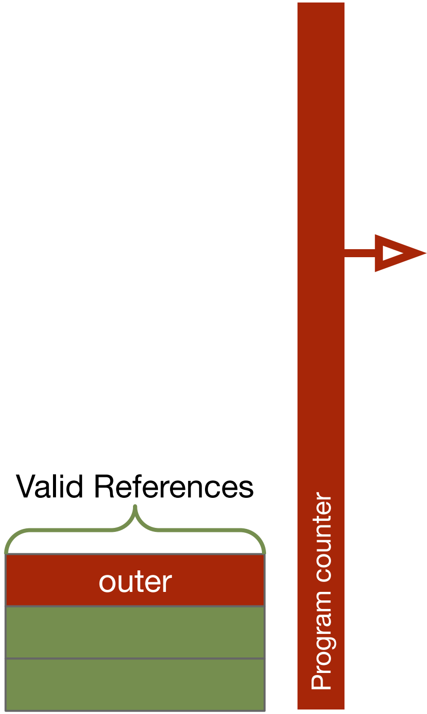
Example

```
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```

REFERENCES: DANGLES



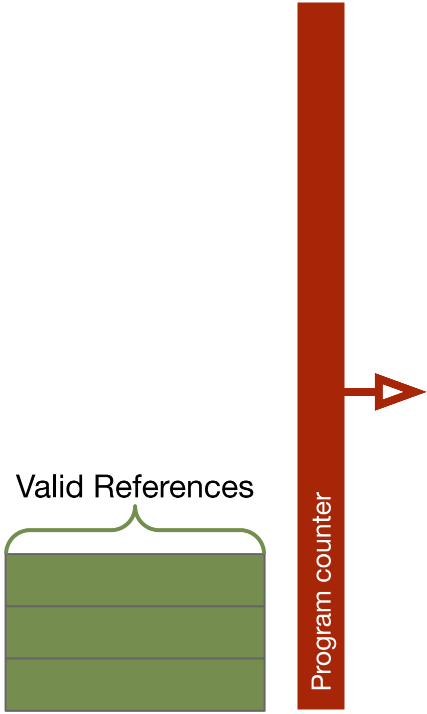
Example

```
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```

REFERENCES: DANGLES



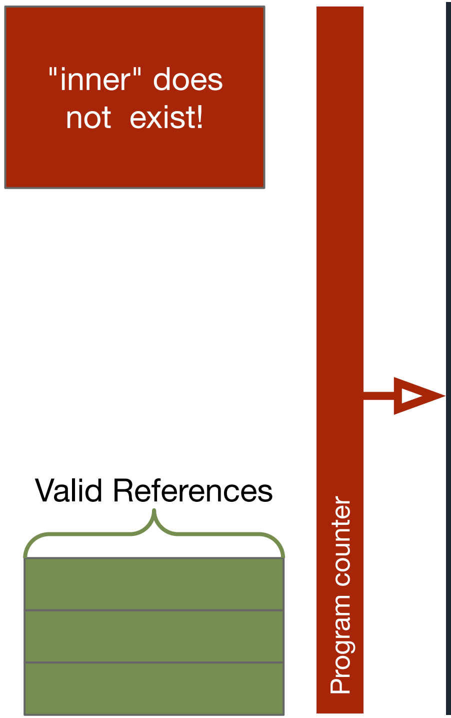
Example

```
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```

REFERENCES: DANGLES



"inner" does not exist!

Valid References

Example

```rust
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```

REFERENCES: LIFETIMES

Rust tracks the **lifetime** of references to make sure that each of them points to valid data at any point in the program where they are used.

The lifetime is related to the *scope* where the reference and the data it points to is available.

REFERENCES: LIFETIMES

The `inner` variable has a smaller lifetime than the `outer` variable.

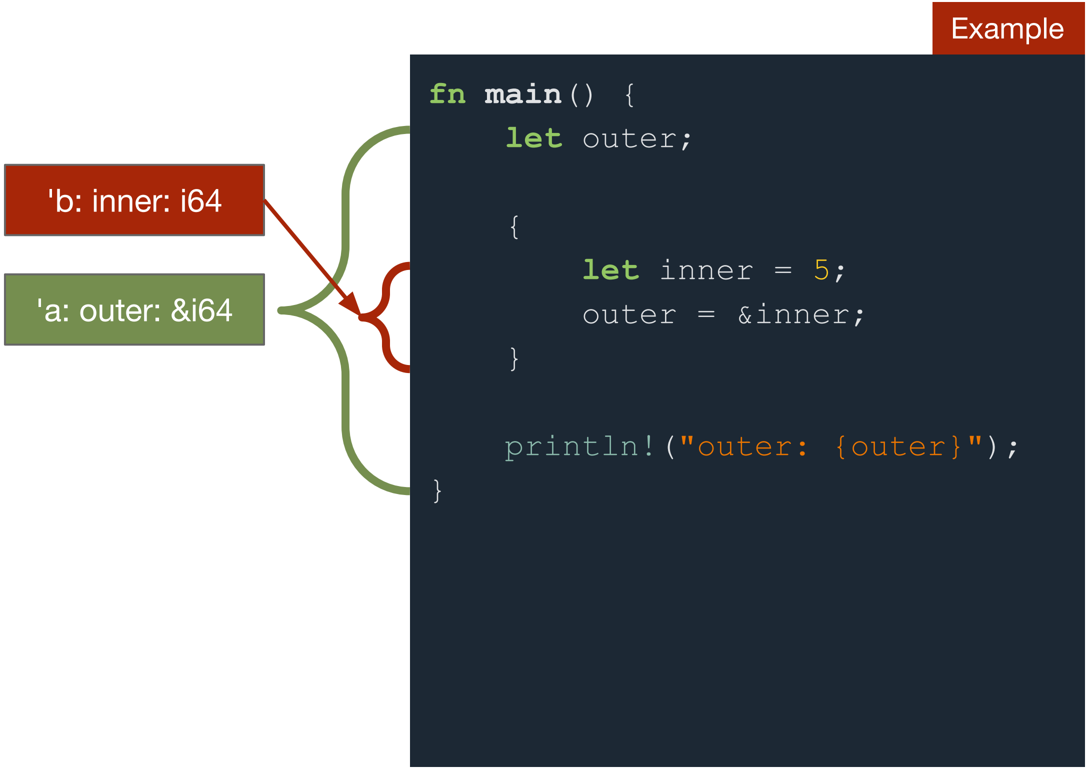
Example

```
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```

REFERENCES: LIFETIMES



'b: inner: i64

'a: outer: &i64

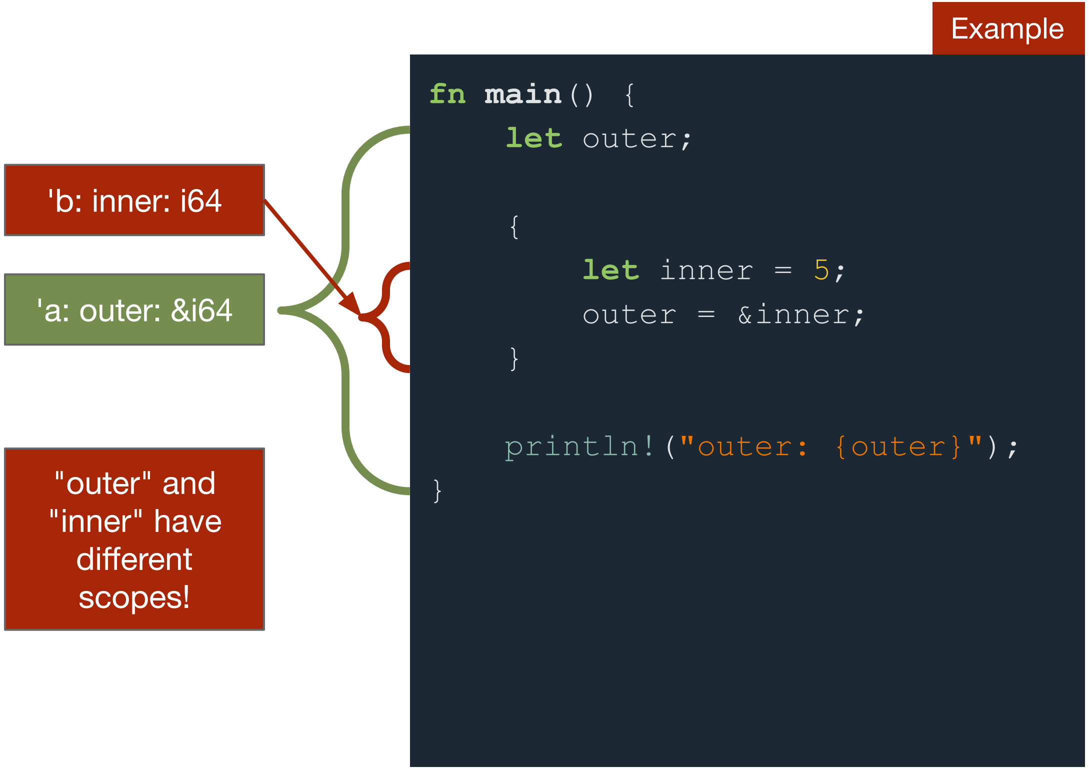
Example

```
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```

REFERENCES: LIFETIMES



'b: inner: i64

'a: outer: &i64

"outer" and "inner" have different scopes!

Example

```
fn main() {
    let outer;

    {
        let inner = 5;
        outer = &inner;
    }

    println!("outer: {outer}");
}
```

REFERENCES: LIFETIMES

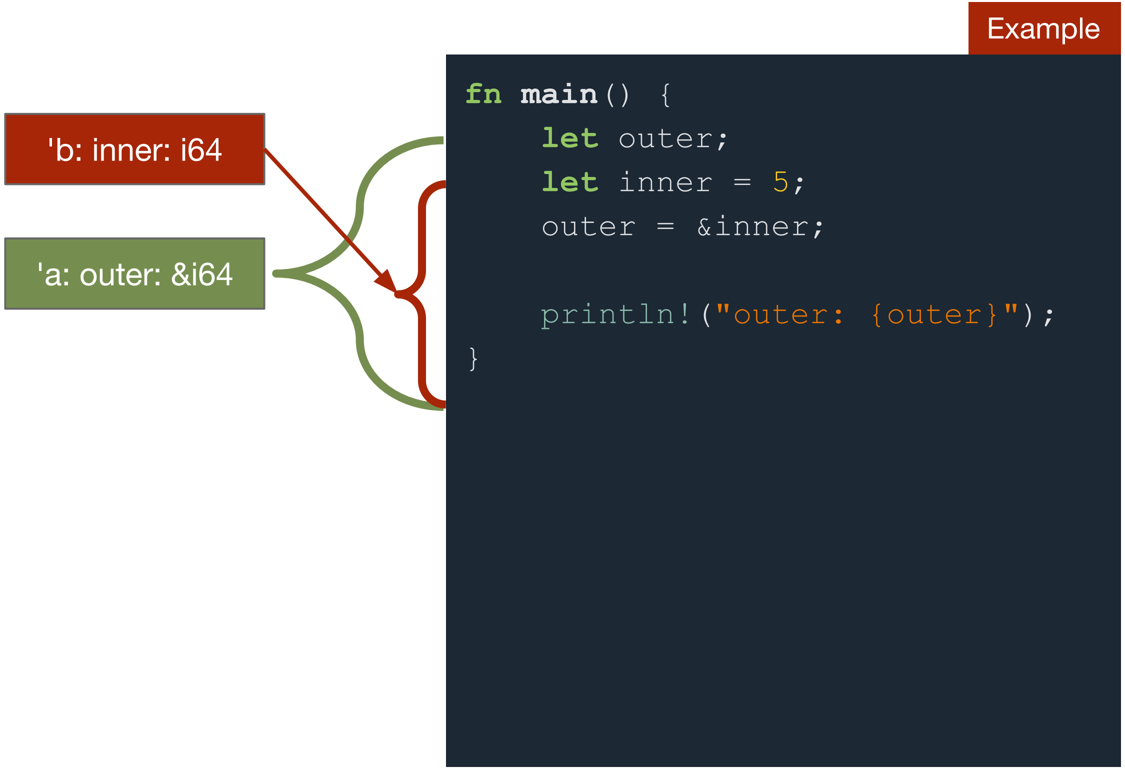
The `outer` and `inner` variables now have overlapping scopes.

Example

```
fn main() {
    let outer;
    let inner = 5;
    outer = &inner;

    println!("outer: {outer}");
}
```

REFERENCES: LIFETIMES



'b: inner: i64

'a: outer: &i64

Example

```rust
fn main() {
    let outer;
    let inner = 5;
    outer = &inner;

    println!("outer: {outer}");
}
```

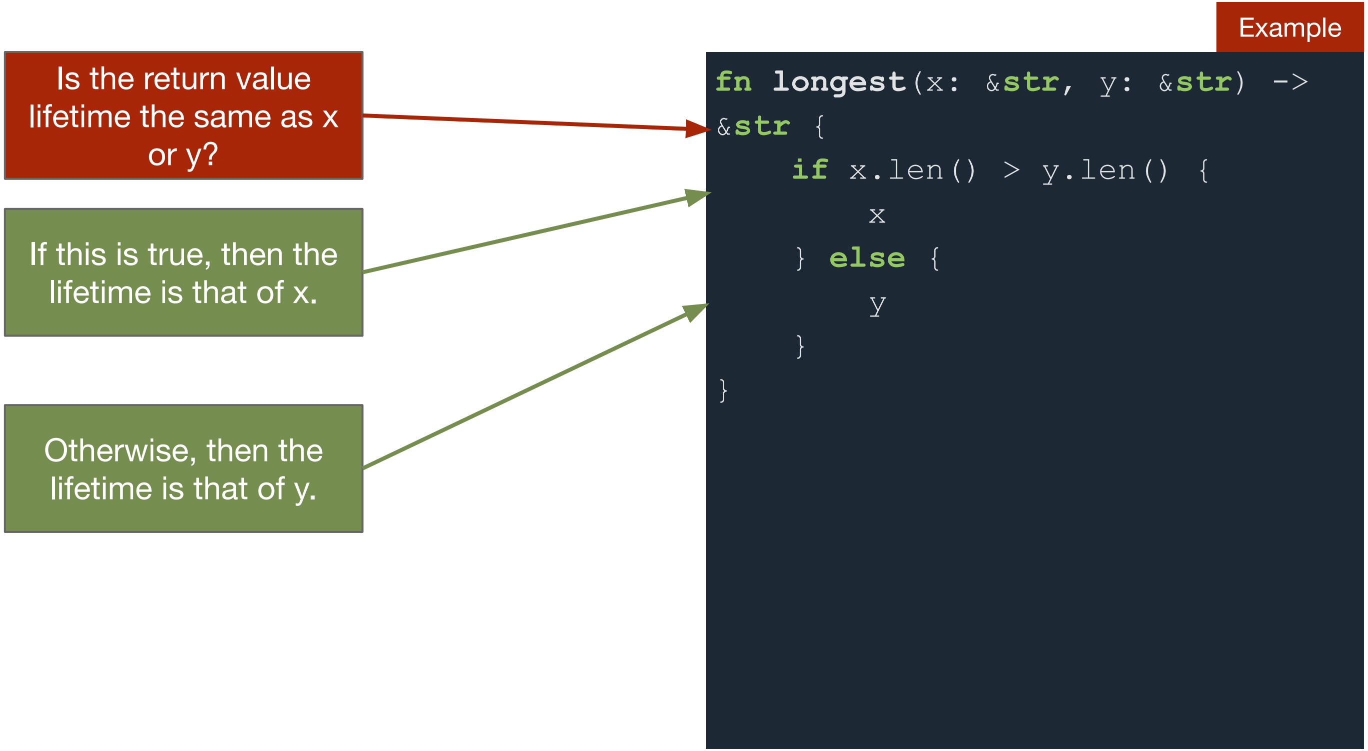
REFERENCES: FUNCTION LIFETIMES

Rust usually is able to determine the lifetimes of each reference in a program. However, there are cases when lifetimes of multiple references used in functions are ambiguous.

Example

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

REFERENCES: FUNCTION LIFETIMES



Is the return value lifetime the same as x or y?

If this is true, then the lifetime is that of x.

Otherwise, then the lifetime is that of y.

Example

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

LIFETIME ANNOTATIONS

Rust supports **lifetime annotations**, which are generic "data types" placed as part of the generic data type list in functions.

Lifetime annotations are named using lowercase letters and are prefixed by an apostrophe. These are placed immediately after the ampersand.

Example

```
// &str
// &'a str
// &'a mut str

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

LIFETIME ANNOTATIONS

Lifetime annotations inform the compiler that certain parameters and return values in a function will have the same or different lifetimes.

For example, parameters and return values annotated with the same lifetime will have the same lifetime, and hence, should be valid throughout the whole function.

Example

```
// &str
// &'a str
// &'a mut str

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

LIFETIME ANNOTATIONS

x, y, and the return value are all annotated to have the same lifetime!

Example

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

LIFETIME ELISION RULES

Technically, all parameters and return values in functions should be explicitly annotated with lifetimes. However, the compiler follows empirically-derived rules to be able to automatically infer lifetimes.

If the compiler is able to infer the lifetimes of all of the parameters and return values, then there is no need to explicitly annotate them.

LIFETIME ELISION RULES

Rust follows some basic rules regarding automatic annotation of lifetimes in functions and methods.

1. Each reference parameter is assigned separate lifetimes.
2. If there is only one reference input, the return value will have the same lifetime as it.
3. If it is a method, the return value will have the same lifetime as `self`.

LIFETIME ELISION

```
fn first_word(s: &str) -> &str
```

Rule 1:
Label with new
lifetime 'a

LIFETIME ELISION

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

Rule 2:
Function with single parameter - label with lifetime 'a

LIFETIME ELISION

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

All references have resolved lifetimes, so no explicit annotation is needed!

LIFETIME ELISION

```
fn longest(x: &str, y: &str) -> &str
```

Rule 1:
Label with new
lifetime 'a

LIFETIME ELISION

```
fn longest(x: &'a str, y: &str) -> &str
```

Rule 1:
Label with new
lifetime 'b

LIFETIME ELISION

```rust
fn longest(x: &'a str, y: &'b str) -> &str
```

All references do not have resolved lifetimes, so explicit annotation is needed!

LIFETIME ANNOTATIONS: STRUCTS

If structs contain a reference as one of its fields, those fields *require* lifetime annotations.

Since lifetime annotations are "generic types", it should also be declared in the generic type list.

Example

```
struct UserAcct <'a> {
    active: bool,
    alias: &'a String,
    username: String,
    sign_in_count: u64,
}
```

LIFETIME ANNOTATIONS: METHODS

If a method is to be written for a struct or enum that contains a lifetime annotation, the `impl` block should also have that lifetime annotation. Lifetime elision rules still apply.

```
impl <'a> UserAcct <'a> {
    fn get_alias(&self) -> &String {
        self.alias
    }
}
```

Example

LIFETIME ANNOTATIONS: STATIC

A data can be made to live for the entire duration of the program by making it *static*. The data can be annotated using the special `'static` keyword.

This annotation should be sparingly used since it skirts the lifetime checks and memory optimizations of Rust.

```
// A str has an implied 'static lifetime
let s: &'static str prog_name = "Rust";
```

Example

RESOURCES

- The Rust Book

CoE 164

Computing Platforms

03c: Lifetimes