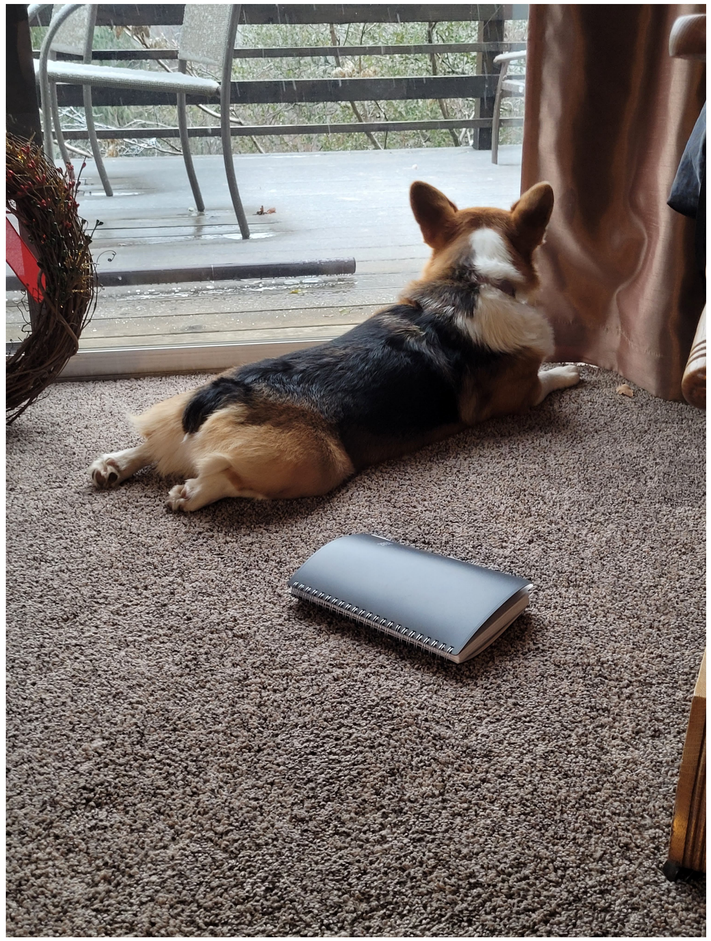
CMPT 383

Lecture 15: Lifetimes and Pointers

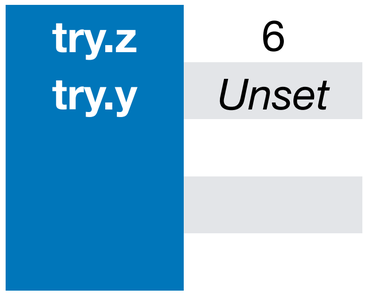


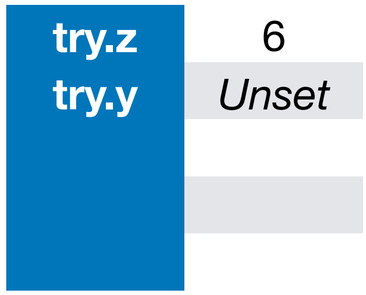
Anders Miltner

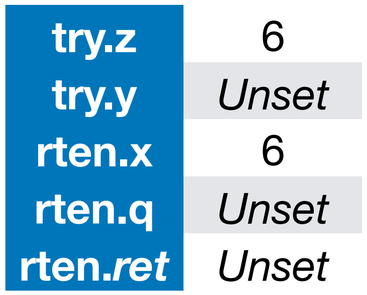
Tricky Examples, Stack Allocated Ref

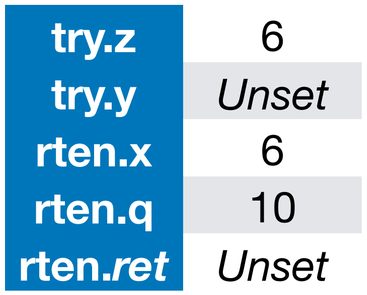
```
fn rten(int x) -> &int {
    q = 10;
    return &10;
}
```

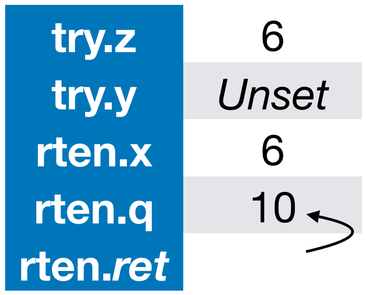
```
fn tryit() {
   let z = 6;
   let y = reference_ten(15);
   other();
   return;
}
```

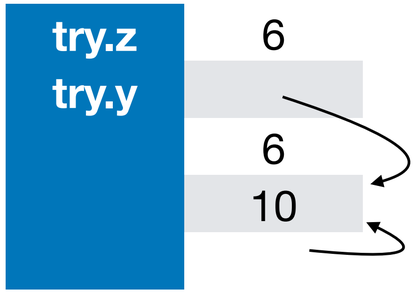


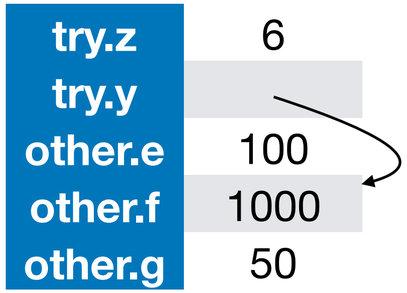












Tricky Examples, Conflicting Lifetimes

```
fn find_longer(x:&Vec<u32>,y:&Vec<u32>) -> &Vec<u32> {
    if x.len() > y.len() {
        return x;
    } else {
        return y;
    }
}
```

```
fn tryit() {
    let a = vec![1,2,3];
    let c;
    {
      let b = vec![4,5,6];
      c = find_longer(a,b)
    }
    … more code
}
```

When can c no longer be used?

Tricky Examples, Borrows in Structs

```
struct Person {
  name: &String,
  age: i32,
}
```

```
fn DefaultAge(name: &String) -> Person {
    let age = 50;
    return Person { name, age }
}
```

What is the lifetime of DefaultAge(“John Doe”)?

Lifetimes!

- You can describe the lifetime of a certain borrowed variable
- You can ensure that the lifetimes agree with each other

```
fn find_longer<'a>(x:&'a Vec<u32>,y:&'a Vec<u32>) -> &’a Vec<u32> {
    if x.len() > y.len() {
        return x;
    } else {
        return y;
    }
}
```

What lifetimes are there

- Ones you can write
- Ones that are inferred from the code structure
- Ones that are built from other lifetimes

Lifetimes you can write

- Static data
- That’s it!

```
const &’static str johnDoe = “John Doe”
```

```
fn find_longer_description(x:&str,y:&str) -> &’static str {
    if x.len() > y.len() {
        return “x is longer”;
    } else {
        return “x is not longer”;
    }
}
```

Lifetimes inferred from code structure

- Everything we've seen before

```
fn normal_code(mut x:& Vec<i32>) {
    let y:& mut Vec<i32> = x;
    y.push(1);
    let z:&Vec<i32> = x;
    println!("{:?}",z);
}
```

```
fn normal_code(mut x:& Vec<i32>) {
    let y:& mut Vec<i32> = x;
    let z:&Vec<i32> = x;
    println!("{:?}",y);
    println!("{:?}",z);
}
```

Lifetimes built from other lifetimes

- Lifetime polymorphism

```
fn find_longer<‘a>(x:&’a Vec<u32>,y:&’a Vec<u32>) -> &’a Vec<u32> {
    if x.len() > y.len() {
        return x;
    } else {
        return y;
    }
}
```

What does ‘a mean here?

- Does it mean the lifetimes need to be the *exact same*
  - No!
- Think about it like trait requirements in generics

```
fn find_longer<‘a>(x:&’a str,y:&’a str) -> &’a str {
    if x.len() > y.len() {
        return x;
    } else if y.len() > x.len() {
        return y;
    } else {
        return “Equal Lengths”;
    }
}
```

Lifetimes Built from Others 1

```
fn find_longer<‘a>(x:&’a Vec<u32>,y:&’a Vec<u32>) -> &’a Vec<u32> {
    if x.len() > y.len() {
        return x;
    } else {
        return y;
    }
}
```

```
fn tryit() {
    let x = vec![1,2,3];
    {
      let y = vec![1,2,3];
      let z = find_longer(x,y);
    }
}
```

Lifetimes Built from Others 2

```
fn find_longer<‘a>(x:&’a str,y:&’a str) -> &’a str {
    if x.len() > y.len() {
        return x;
    } else {
        return y;
    }
}
```

```
fn tryit() {
    {
      let y = GetUserInput();
      let z = find_longer(“hi”,y);
    }
}
```

Lifetimes inferred from code structure and built from others

Lifetime elision in functions

In order to make common patterns more ergonomic, lifetime arguments can be *elided* in function item, function pointer, and closure trait signatures. The following rules are used to infer lifetime parameters for elided lifetimes. It is an error to elide lifetime parameters that cannot be inferred. The placeholder lifetime, `'_`, can also be used to have a lifetime inferred in the same way. For lifetimes in paths, using `'_` is preferred. Trait object lifetimes follow different rules discussed below.

- Each elided lifetime in the parameters becomes a distinct lifetime parameter.
- If there is exactly one lifetime used in the parameters (elided or not), that lifetime is assigned to *all* elided output lifetimes.

https://doc.rust-lang.org/reference/lifetime-elision.html

Pointers

- Borrow = Reference = &
  - All borrows are references and all references are borrows
  - And all of them are defined using &
- All references are pointers
- Not all pointers are references

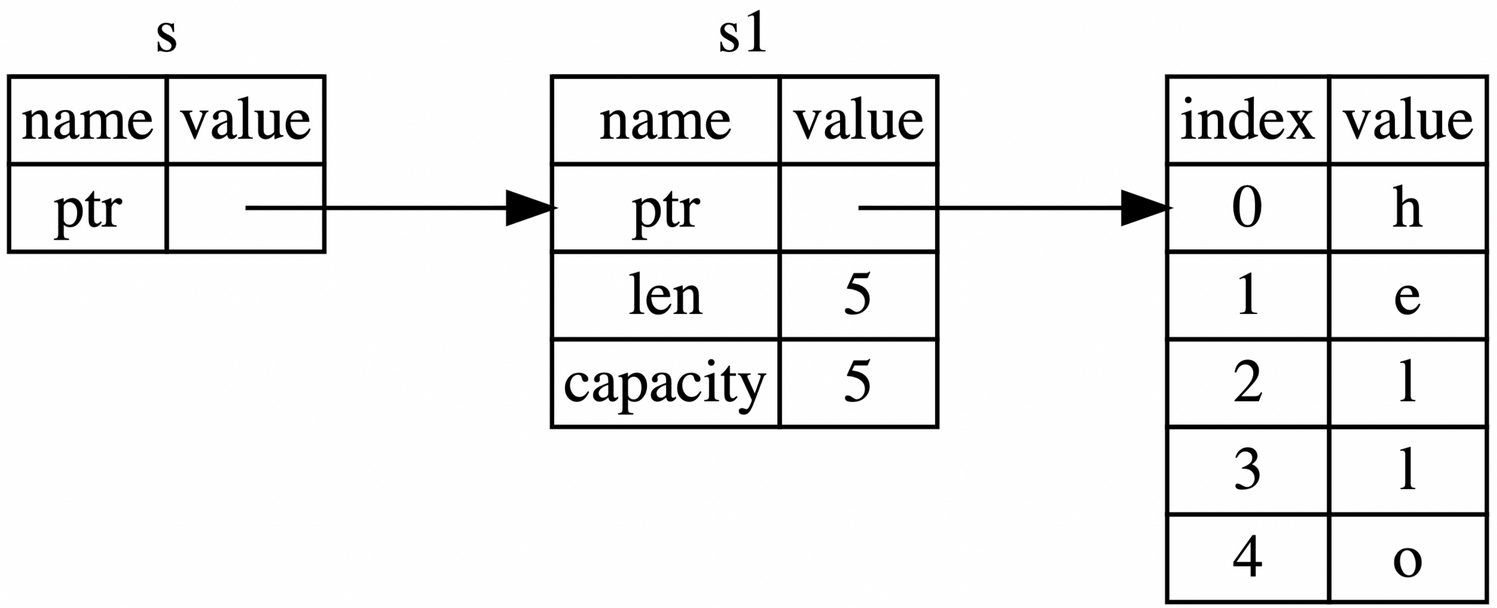
Example: Borrowing an Array

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```



https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Let’s focus on references/borrows

- That’s what we’ve been doing so far
- Also dereferencing exists
- But, like referencing, there’s trickiness to it
- Dereferencing only can happen with cloning or when implementing the Copy trait

Dereferencing Examples

```
fn ident_fn<a>(x:a) -> a {
    return x;
}
```

```
fn ident_fn<a>(x:a) -> a {
    return *&x;
}
```

```
fn ident_fn<a:Copy>(x:a) -> a {
    return *&x;
}
```

Other types of references

- Arrays
  - This is just built-in. We'll go into arrays more later.
- Box
  - Allocating on the heap
- Rc
  - Reference counting + multiple ownership
- Ref<T> and RefMut<T>
  - Same as borrowing but enforced at runtime
- ???

Box<T>

- Most basic type of reference
- Points to data on the heap
- Box::new<a>(x:a)
  - Box::new<i32>(5)
  - Box::new<&str>(“Hello, Box!”)

Box for recursive types

```
Enum Tree<a> {
    Leaf,
    Node(Tree<a>,a,Tree<a>)
}
```

```
error[E0072]: recursive type `Tree` has infinite size
  --> exercises/strings/strings1.rs:10:1
   |
10 | enum Tree<a> {
   | ^^^^^^^^^^^^ recursive type has infinite size
11 |     Leaf,
12 |     Node(Tree<a>,a,Tree<a>)
   |          -------   ------- recursive without indirection
   |          |
   |          recursive without indirection
   |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `Tree` representable
```

We must allocate memory, how much?

```
Enum Tree<a> {
    Leaf,
    Node(Tree<a>,a,Tree<a>)
}
```

Depends on the size of the tree

Instead, let’s use pointers!

```
Enum Tree {
    Leaf,
    Node(Box<Tree>,i32,Box<Tree>)
}
```

```
let x = Node(Box::new(Leaf),2,Box::new(Leaf))
```

How do you escape a Box?

```
let x = Box::new(10);
let y = *x;
```

*** really means follow a pointer, not the opposite of “reference”**

Rc

- Reference Counting Pointer
- Permits multiple owners of the same data
- Looks like a clone: isn’t actually a clone, actually just refers to the same data

Instead, let’s use pointers!

```
fn tester() {
    rc_str = “RC String”.to_string();
    let rc_a::Rc<String> = Rc::new(rc_examples);
    {
        let rc_b::Rc<String> = Rc::clone(&rc_a);
        println!(“{}”,Rc::strong_count(&rc_a);//prints 2
    }
    println!(“{}”,Rc::strong_count(&rc_a);//prints 1
}
```