Introduction to Lifetimes

Nathan Ringo (`remexre`)

November 8, 2023

Recap and outline

- `search`
  - "Imperative way"
  - "Functional way"
- References and Lifetimes
- Interior Mutability

search

- Base: https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=367a4490e732f842e45800ba60249795

search

- Base: https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=367a4490e732f842e45800ba60249795

- `pub fn retain<F>(&mut self, f: F) where F: FnMut(&T) -> bool`
  https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=79cd0ae6e3dcc22c37242e6b0bf0bdff

search

- Base: https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=367a4490e732f842e45800ba60249795

- `pub fn retain<F>(&mut self, f: F) where F: FnMut(&T) -> bool`
  https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=79cd0ae6e3dcc22c37242e6b0bf0bdff

- Iterators and `filter`!
  https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=fd52e11181bfb94d7f685ef568b72a62

How do I access values?

Type
`&String`
`&mut String`
`String`

How do I access values?

Type	Read
`&String`	✓
`&mut String`	✓
`String`	✓

`&`: “shared reference” or “immutable reference” (slight misnomer)

- “kind of like C’s `const*`” but...

```
pub fn len(&self) -> usize { ... }
```

How do I access values?

Type	Read	Write
`&String`	✓	✗
`&mut String`	✓	✓
`String`	✓	✓

`&`: "shared reference" or "immutable reference" (slight misnomer)

- "kind of like C's `const*`" but...

`&mut`: "unique reference" or "mutable reference" (slight misnomer)

- "kind of like C's `*`" but...

```
pub fn len(&self) -> usize { ... }
pub fn push_str(&mut self, string: &str) { ... }
```

How do I access values?

Type	Read	Write	Free
`&String`	✓	✗	✗
`&mut String`	✓	✓	✗
`String`	✓	✓	✓

`&`: "shared reference" or "immutable reference" (slight misnomer)

- "kind of like C's `const*`" but...

`&mut`: "unique reference" or "mutable reference" (slight misnomer)

- "kind of like C's `*`" but...

```
pub fn len(&self) -> usize { ... }
pub fn push_str(&mut self, string: &str) { ... }
pub fn into_bytes(self) -> Vec<u8> { ... }
```

How do I access values?

Type	Read	Write	Free	How many?
`&String`	✓	✗	✗	∞ (but no `&mut`)
`&mut String`	✓	✓	✗	1
`String`	✓	✓	✓	...

`&`: "shared reference" or "immutable reference" (slight misnomer)

- "kind of like C's `const*`" but...

`&mut`: "unique reference" or "mutable reference" (slight misnomer)

- "kind of like C's `*`" but...

```
pub fn len(&self) -> usize { ... }
pub fn push_str(&mut self, string: &str) { ... }
pub fn into_bytes(self) -> Vec<u8> { ... }
```

References have lifetimes

- `pub fn len(&self) -> usize` is implicitly

  `pub fn len<'a>(&'a self) -> usize`

- `pub fn push_str(&mut self, s:&str)` is implicitly

  `pub fn push_str<'a>(&'a mut self, s:&'a str)`

- `fn f(&mut self, x:&T) -> &U` is implicitly

  `fn f<'a>(&'a mut self, x:&'a T) -> &'a U`

What are lifetimes?

- The range of the program where a reference is valid
  - If you want to make a &mut for a value, all its &s must be dead
  - If you want to own a value, all its &s and &muts must be dead

What are lifetimes?

- The range of the program where a reference is valid
  - If you want to make a &mut for a value, all its &s must be dead
  - If you want to own a value, all its &s and &muts must be dead

- Since you need to own something to free it, if you have a reference (that's alive) to it, it can't have been freed!
  - No use-after-free bugs!
  - `https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=37e5d4df6251aac80838b2753b295337`

What are lifetimes?

- The range of the program where a reference is valid
  - If you want to make a &mut for a value, all its &s must be dead
  - If you want to own a value, all its &s and &muts must be dead

- Since you need to own something to free it, if you have a reference (that's alive) to it, it can't have been freed!
  - No use-after-free bugs!
  - `https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=37e5d4df6251aac80838b2753b295337`

- If you have a reference to a value, nobody else can have a &mut to it. For *most* types, this means nobody can mutate it out from under you!
  - Counterexamples: `Mutex`, `AtomicU32`, `Cell`, ...

Interior Mutability

- Some types let you mutate them with only a &
  - This is why &mut = “mutable reference” is a slight misnomer
- Simple example: `Cell`
- Example you’ll use: `Mutex`
- *Not* bad, but “don’t do it unless you need to”
  - You don’t know “nobody can mutate it out from under you”