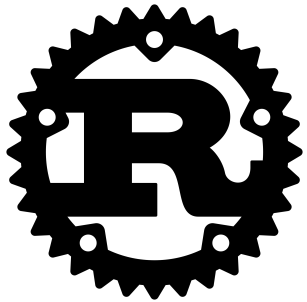
Introduction to Rust

Ian McCormack



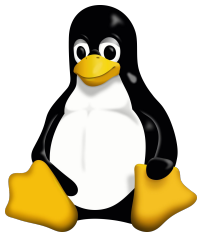
Why would you choose one programming language over another?

What is systems programming?

Rust is popular and widely used in production.

Chosen as **the “most loved” language** in StackOverflow’s annual developer survey for **the last eight years.**





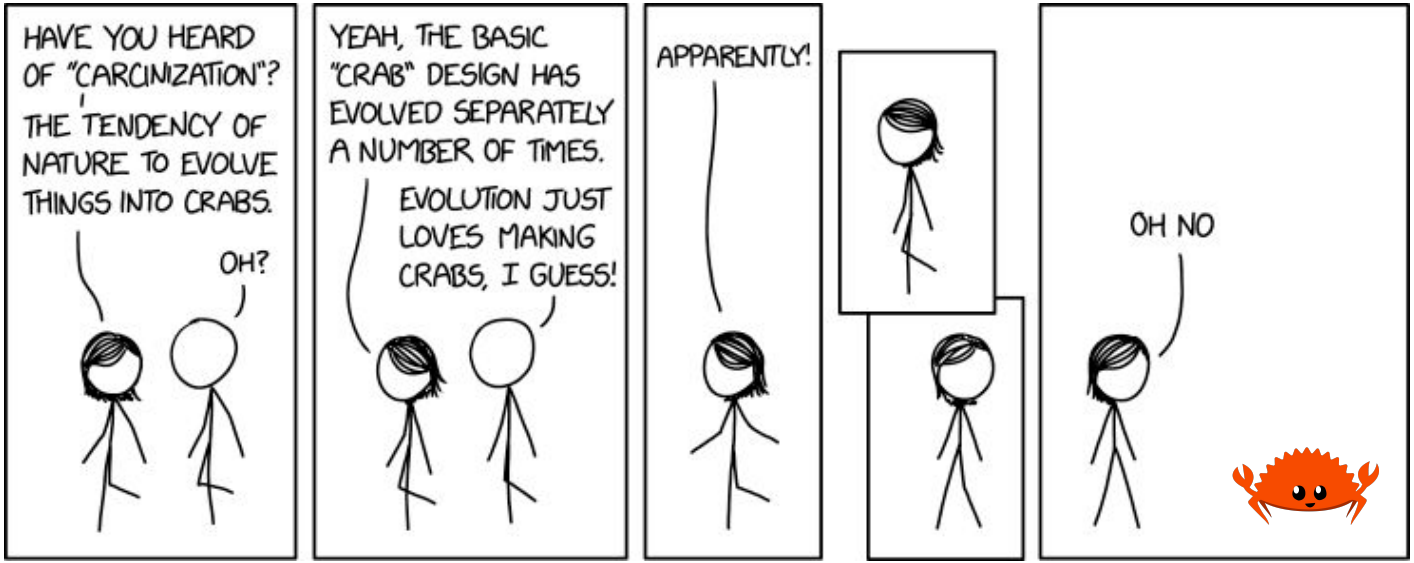

Total

	Energy (J)
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time (ms)
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) [illegible]	
(i) J[illegible]	

Pereira et al., 2021


HAVE YOU HEARD OF "CARCINIZATION"? THE TENDENCY OF NATURE TO EVOLVE THINGS INTO CRABS.
OH?
YEAH, THE BASIC "CRAB" DESIGN HAS EVOLVED SEPARATELY A NUMBER OF TIMES.
EVOLUTION JUST LOVES MAKING CRABS, I GUESS!
APPARENTLY!
OH NO

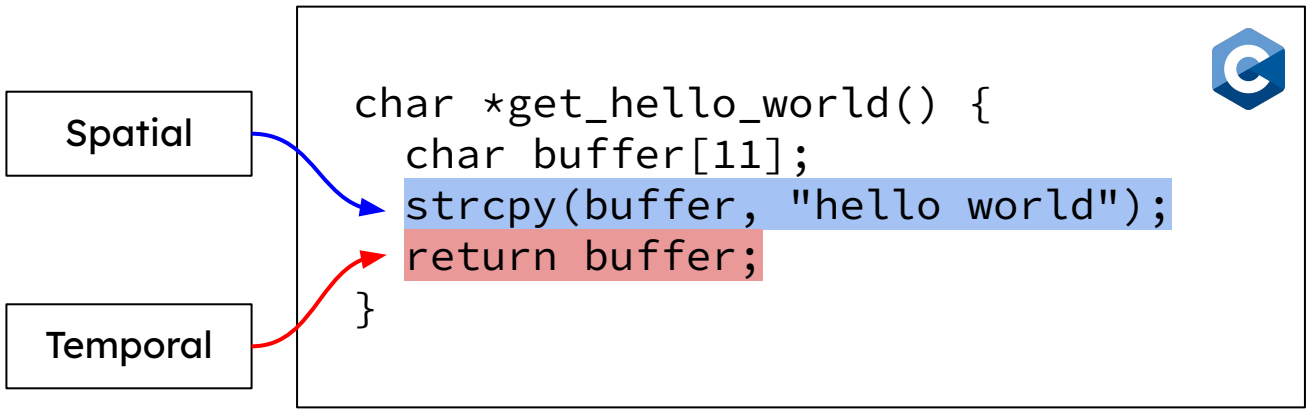
What is a memory error?

“...an object accessed using a pointer expression is different from the one intended.”

— van der Veen et al., 2012

Nearly 70% of security bugs found by Google (2015 - 2020) and Microsoft (2006 - 2018) were caused by memory errors.

```
char *get_hello_world() {
  char buffer[11];
  strcpy(buffer, "hello world");
  return buffer;
}
```


Spatial
Temporal
char *get_hello_world() {
char buffer[11];
strcpy(buffer, "hello world");
return buffer;
}

Garbage collection supports **dynamic memory safety**.

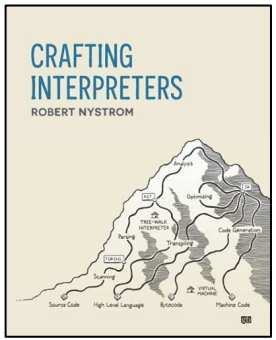
Tracing garbage collection treats memory as a reachability graph, and periodically eliminates nodes that are unreachable.

Reference counting frees memory when the count of references to an allocation in scope reaches zero.

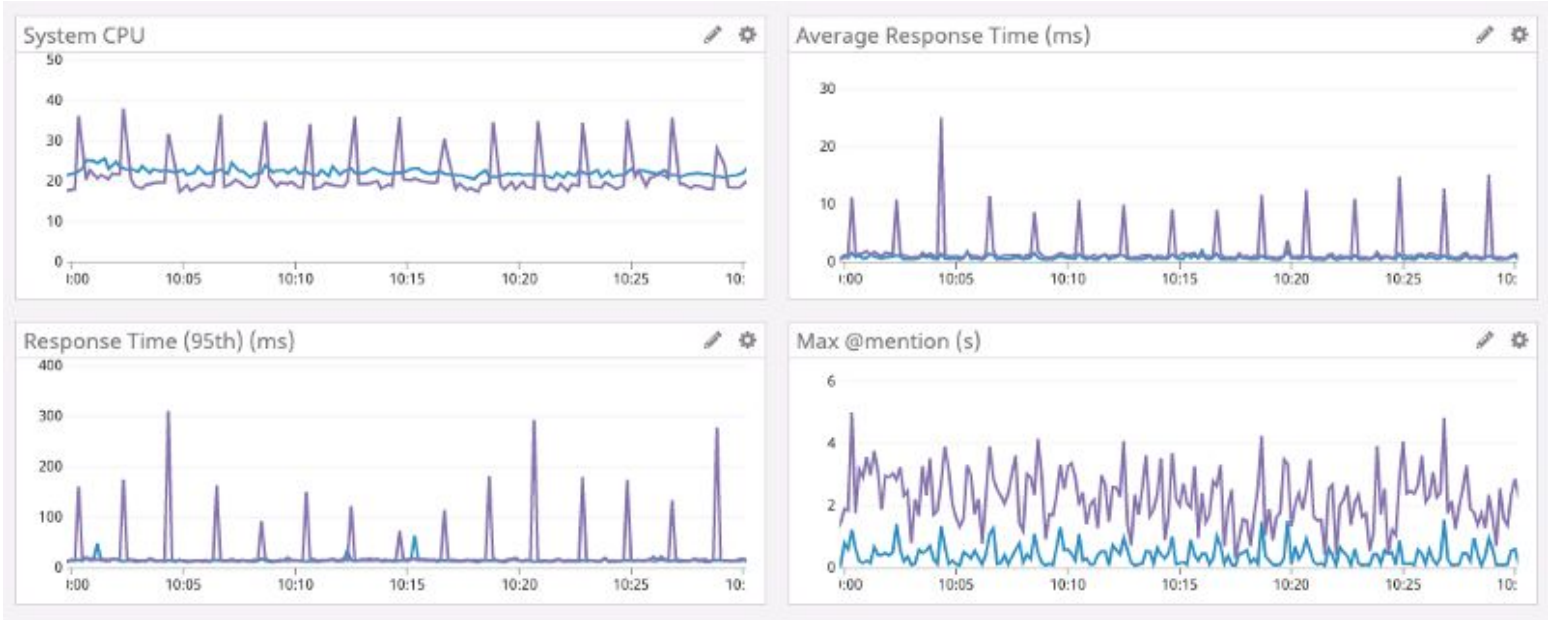
No use-after free!

+ No pointers
+ Bounds checks on array accesses

Check out *Crafting Interpreters*!



Performance of Discord's "Read States" Service



Go Rust

(February 2020)

Rust is nearly as *performant* and *efficient* as C,
but with **memory safety guarantees.**

Rust

- Basic language features
- Ownership & the borrow checker
- Generic types

Rust Resources

The Rust Programming Language — https://doc.rust-lang.org/book/

A tutorial on every aspect of Rust; a great starting point.

The Rust Reference — https://doc.rust-lang.org/stable/reference/

Comprehensive description of Rust's language features.

Rust By Example — https://doc.rust-lang.org/rust-by-example/index.html

Explanations of Rust concepts alongside runnable examples.

Mutability

Variables are immutable by default.

Can be declared as mutable with the ‘mut’ keyword.

```
let x = 5;
x = 7;
```


```
let mut x = 5;
x = 7;
```

Why do we have immutability by default?

Shadowing

```
let x:u16 = 0xffff;
{
    let x:u32 = 0xffffffff;
    {
        let x:u64 = 0xffffffffffffffff;
    }
}
```

```
let x:u16 = 0xffff;
let x:u32 = 0xffffffff;
let x:u64 = 0xffffffffffffffff;
```

Numerical Types

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

Tuples & Arrays

```
let x:(u8, i32, char) = (1, -1000000, '@');

let (byte, int, at) = x;

let y = x.2;
```

```
let x:[i32;3] = [1, 1, 1];

let x = [1;3];
```

Functions & Closures

```
fn max(a: u32, b: u32) -> bool {

    let min = || -> bool {
        if a > b {
            b
        } else {
            a
        }
    }

    !min()
}
```

Structs & Traits

```
struct GradStudent {
    coffee_allowance: u8,
}

trait Customer {
    fn pay(& mut self, price: u8)
}

impl Customer for GradStudent {
    fn pay(& mut self, price: u8) {
        if self.coffee_allowance < price {
            panic!(“..”)
        }else{
            self.coffee_allowance -= price
        }
    }
}
```

The Borrow Checker

The Borrow Checker

"…a system for statically building a proof that data in **memory is either uniquely owned** (and thus able to allow unguarded mutation) **or collectively shared, but not both**."

Check out *Oxide: the Essence of Rust*!

Oxide: The Essence of Rust

AARON WEISS, Northeastern University, USA
OLEK GIERCZAK, Northeastern University, USA
DANIEL PATTERSON, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

arXiv:1903.00982v4 [cs.PL] 19 Oct 2021

The Borrow Checker

1. All values have exactly one owner.

   ```
   let x = String::from(“Hello world!”);
   ```

2. A reference to a value cannot outlive the owner.

   ```
   let y = & mut x;
   ```

3. A value can have ***one mutable*** reference **or *many immutable*** references

— Crichton 2021

You spend 3-6 months in a cave,

breathing, eating and *sleeping* Rust.

Then find like-minded advocates

who are prepared to sacrifice their first born

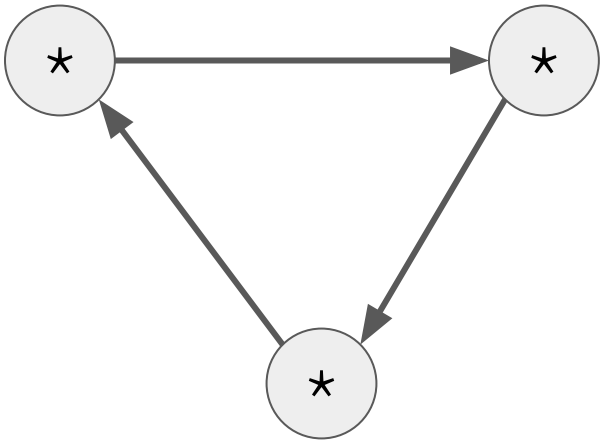
to perpetuate the unfortunate sentiment that Rust is the future.

— Interviewee from [Fulton et al. 2021]

Learning Rust Ownership is

the **best**

The Borrow Checker rejects “valid” programs.



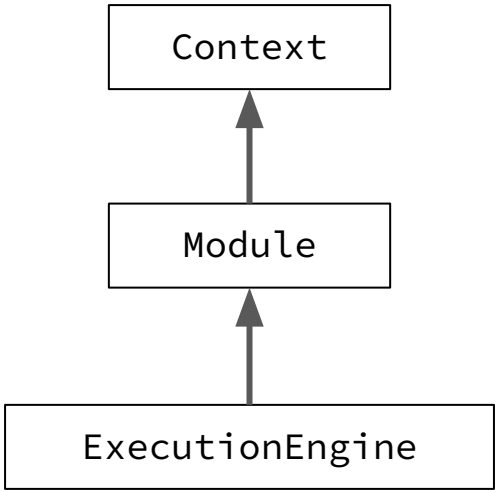
❌ Doubly-linked lists

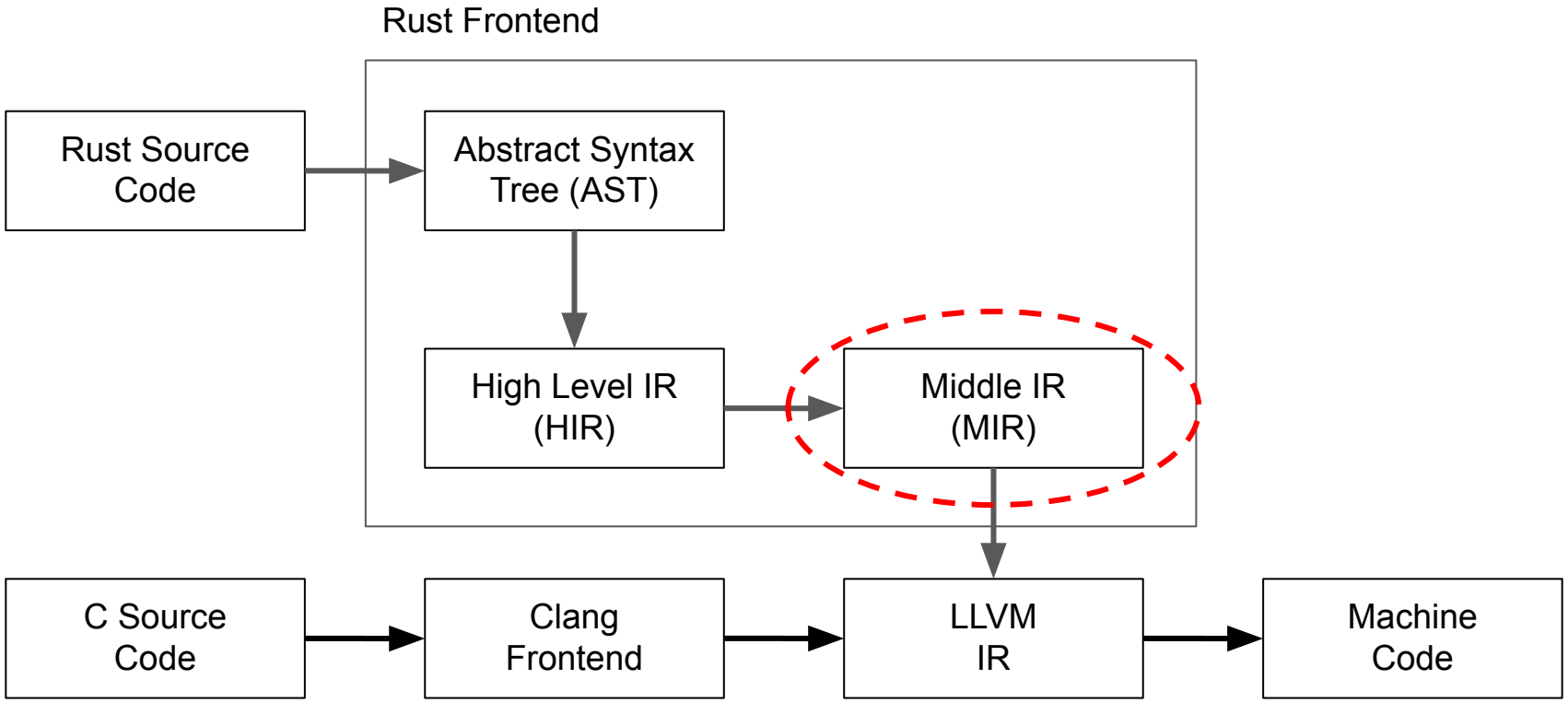
❌ Trees with parent and child pointers

❌ ***Any*** self-referential struct

Self-Referential Structs

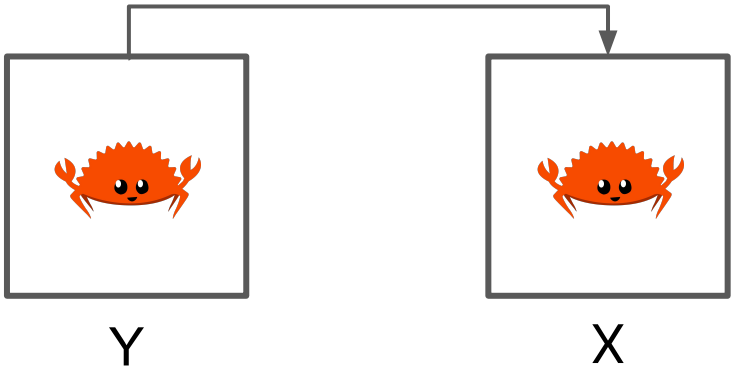
```
struct LLI {
    context: Context,
    module: Module,
    engine: ExecutionEngine,
}
```




Rust Frontend
Rust Source Code
Abstract Syntax Tree (AST)
High Level IR (HIR)
Middle IR (MIR)
C Source Code
Clang Frontend
LLVM IR
Machine Code

Copy Semantics

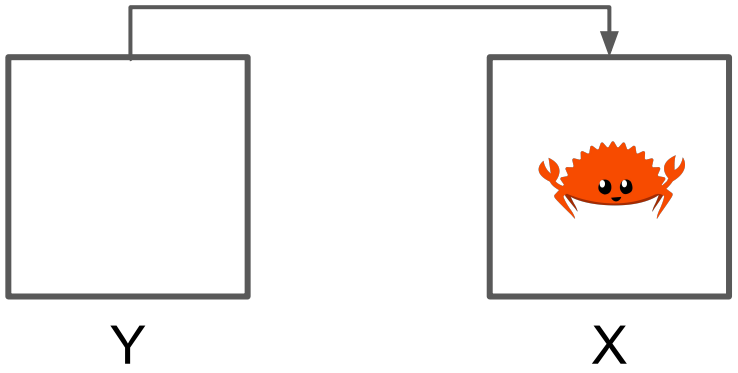
x = y



Y X

Move Semantics

x = y



Y X

Strings

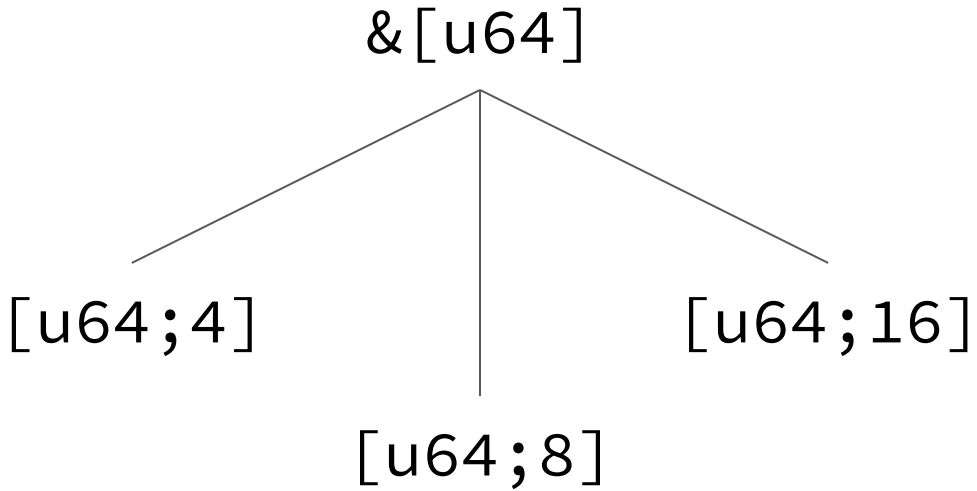
```
let greeting = String::from("Hello world!");

// create a response
let response = greeting;

// remove the ‘!’
response.pop();

// add a ‘?’ to the end
response.insert(response.len(), '?');

println!("{}", greeting);

println!("{}", response);
```

Borrowing

```
let greeting: String = String::from("Hello world!");
let response: &str = &greeting[..11];
println!("{}", greeting);
println!("{}?", response);
```

✅

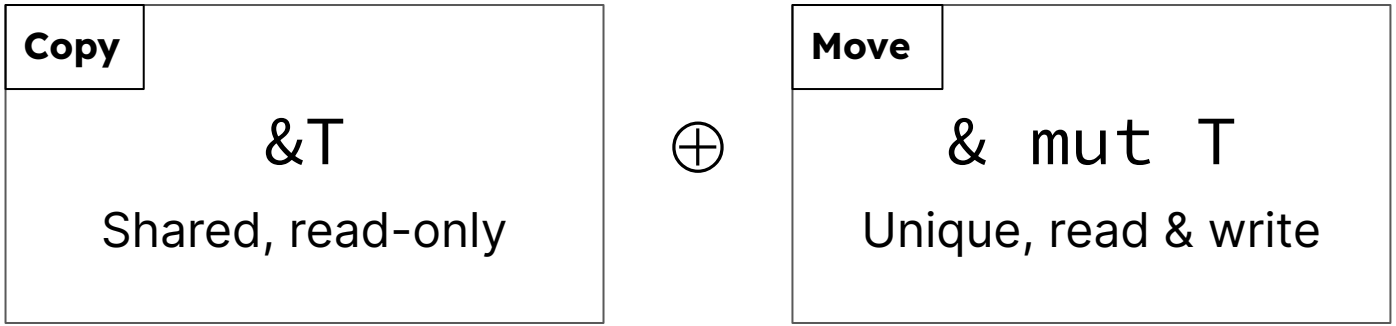
Slices



Borrowing

```
let greeting: String = String::from("Hello world!");
let response: &str = &greeting[..11];   // &[u8]
println!("{}", greeting);
println!("{}?", response);
```

✅

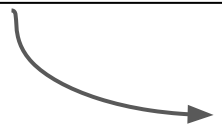

Copy
&T
Shared, read-only
⊕
Move
& mut T
Unique, read & write

We can avoid moves with borrowing.

```
{
    let mut greeting = String::from("Hello world!");

    let response = &greeting[..11];

    println!("{}", greeting);

    greeting.clear();                    // fn clear (& mut self)

    println!("{}?", response);
}
```

We can avoid moves with borrowing.

```
error[E0502]: cannot borrow `greeting` as mutable because it is also
borrowed as immutable
  --> src/lib.rs:8:2
   |
4  |     let response = &greeting[..11];
   |                    -------- immutable borrow occurs here
...
8  |     greeting.clear();
   |     ^^^^^^^^^^^^^^^^ mutable borrow occurs here
9  |
10 |     println!("{}?", response);
   |                    -------- immutable borrow later used here
```



Rust has awesome error messages!

Rust's borrow checker reasons using **lifetimes**.

The **scope of a value** is the duration for which it is **allocated.**

The **lifetime of a reference** is the *duration* for which it is **used.**

What are the scopes and lifetimes involved in this example?

```
{
        let greeting = String::from("Hello world!");

        let response = &greeting[..11];

        println!("{}", greeting);

        greeting.clear();     // implicit, & mut greeting

        println!("{}", response);

}
```

What are the scopes and lifetimes involved in this example?

```
{
        let greeting = String::from("Hello world!");

        let response = &greeting[..11];

        println!("{}", greeting);

        greeting.clear();     // implicit, & mut greeting

        println!("{}", response);
}
```

Resource Allocation is Initialization (RAII)

```
#include <vector>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    for (auto item : v) {
        v.push_back(item);
    }
}
```

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];
    for item in v.iter() {
        v.push(*item)
    }
}
```

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:9
  |
3 |     for item in v.iter() {
  |                 --------
  |                 |
  |                 immutable borrow occurs here
  |                 immutable borrow later used here
4 |         v.push(*item)
  |         ^^^^^^^^^^^^ mutable borrow occurs here
```

Generics

```
fn cashier<T:Customer>(customers: Vec<T>, items: Vec<u8> ) {

    for (customer, item) in customers.iter().zip(items) {
        customer.pay(item);
    }
}
```

Enums

```
match something_error_prone() {
    Ok(success_value) => { ... }
    Err(err_value) => { ... }
}
```

Result<T,E>

```
match optional_result() {
    Some(value) => { ... }
    None => { ... }
}
```

Option<T>

Shadowing

```
let value: Option<u64> = find_value();

let value: u64 = match value {
    Some(v) => v,
    None => ...
}
```

```
let value: u64 = find_value()
    .unwrap_or(|| ...);
```

Lifetimes

```
fn assign(mut a: & i32, b: & i32)
{
    a = b;
}
```


```
error: lifetime may not live long enough
 --> src/lib.rs:3:5
  |
1 | fn assign(mut a: & i32, b: & i32)
  |                  -         - let's call the lifetime of this reference `'1`
  |                  |
  |                  let's call the lifetime of this reference `'2`
2 | {
3 |     a = b;
  |     ^^^^^ assignment requires that `'1` must outlive `'2`
  |
```

Lifetimes

```
fn assign<'a, 'b>(mut a: & 'a i32, b: & 'b i32)
{
    a = b;
}
```

`'a :> 'b`

The region of code represented by ‘**b** must fully contain the region ‘**a**.

The region ‘**b** is a subtype of the region ‘**a.**

Lifetimes are types.

```
class Animal { ... }

class Crab extends Animal { ... }
```


```
Animal a = (Animal) new Crab();
```

1

```
Animal some_animal;
...
Crab a = (Crab) some_animal;
```

2

Lifetimes

```
fn assign<'a, 'b>(mut a: & 'a i32, b: & 'b i32)
where 'b:'a
{
    a = b;
}
```

`'a :> 'b`

The region of code represented by **‘b** must fully contain the region **‘a**.

The region **‘b** is a subtype of the region **‘a.**

Lifetimes

```
fn assign<'a, 'b>(mut a: & 'a i32, b: & 'b i32)
where 'b:'a
{
    a = b;
}
```

```
fn assign<'a>(mut a: & 'a i32, b: & 'a i32)
{
    a = b;
}
```

```
help: consider introducing a named lifetime parameter
  |
1 | fn assign<'a>(mut a: &'a  i32, b: &'a  i32)
  |           ++++         ++            ++
```

Exercise:

Complete the signature and implementation for the following versions of the ‘`swap`’ function without using any other dependencies. Some signatures are incomplete.

1. `fn swap(x: & i32, y: & i32)`
2. `fn swap(x: & & i32, y: & & i32)`
3. `fn swap<T>(x: & T, y: & T)`

Each should swap the values of the two parameters such that, after the function returns, we have:

```
*y = old(*x)  &&  *x = old(*y)
```

Answer #1

```
fn swap<T>(x: & mut i32, y: & mut i32)
{
     let temp = *x;
     *x = *y;
     *y = temp;
}
```

Answer #2

```
fn swap<T, ‘a>(x: & mut ‘a & T, y: & mut ‘a & T)
{
     let temp = *x;
     *x = *y;
     *y = temp;
}
```

Answer #3

std::mem::swap

```
fn swap<T>(x: & mut T, y: & mut T)
{
    unsafe {
        let a = ptr::read(x);
        let b = ptr::read(y);
        ptr::write(x, b);
        ptr::write(y, a);
    }
}
```