

Programming Languages (8)

Rust Memory Management

Kenjiro Taura

Contents

Overview

Rust basics

Owning pointers

- Assignments of owning pointers

- `Box<T>` type

Borrowing pointers (`&T`)

Borrow checking details

Summary

Contents

Overview

Rust basics

Owning pointers

- Assignments of owning pointers

- `Box<T>` type

Borrowing pointers (`&T`)

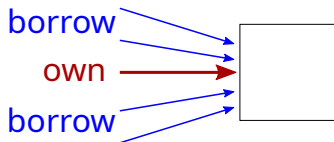
Borrow checking details

Summary

Rust's basic idea to memory management

- ▶ Rust maintains that, for any live object,
 1. there is one and only one pointer that “owns” it (*the owner pointer*)
 2. “*multiple borrowers*” : there are arbitrary number of non-owning pointers (*borrowing pointers*) pointing to it, but *they cannot be dereferenced after the owning pointer goes away*
- ▶ \Rightarrow *it can safely reclaim the data when the owning pointer goes away*

“*single-owner-multiple-borrowers* rule”



The rules are enforced statically

- ▶ Rust enforces the rules (or, detect violations thereof) *statically* (as opposed to *dynamically*)
 - ▶ *compile-time* rather than at *runtime*
 - ▶ *before* execution not *during* execution

“borrow checker”

Ways outside the basic rules

to be sure, there are some ways to get around the rules

1. **reference counting pointers** (\approx multiple owning pointers)
 - ▶ counts the number of owners *at runtime*, and reclaim the data when all these pointers are gone
2. **unsafe/raw pointers** (\approx totally up to you)

they are not specific to Rust, and we'll not cover them in the rest of this slide deck

Contents

Overview

Rust basics

Owning pointers

Assignments of owning pointers

`Box<T>` type

Borrowing pointers (`&T`)

Borrow checking details

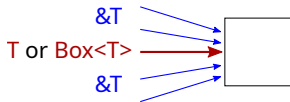
Summary

Pointer-like data types in Rust

given a type T (`i32`, `struct`, `enum`, ...), below are types representing “references (pointers) to T ”¹

1. T : **owning** pointer to T
2. `Box<T>` (`box T`) : **owning** pointer to T
3. `&T` (pronounced “*ref T*”) : **borrowing** pointer to data of T (through which you cannot modify it)
4. `Rc<T>` and `Arc<T>` : shared (reference-counting) owning pointer to T
5. `*T` : unsafe pointer to T

following discussions are focused on T , `Box<T>` and `&T`.



¹we use pointers and references interchangeably

Pointer-making expressions

given an expression e of type T , below are expressions that make pointers to the value of e

1. e (of type T) : an owning pointer
2. $\text{Box}::\text{new}(e)$ (of type $\text{Box}<T>$) : an owning pointer
3. $\&e$ (of type $\&T$) : a borrowing pointer

An example

```
1 {  
2   let a: S      = S{x: ...};    // allocate memory for S  
3                                   // and make a owning pointer to it  
4   let b: S      = a;            // an owning pointer  
5   let c: Box<S> = Box::<S>::new(a); // an owning pointer  
6   let d: &S     = &a;           // a borrowing pointer  
7 }
```

- ▶ note: type of variables can be omitted (spelled out for clarity)
- ▶ note: the above program violates several rules so it does not compile

Contents

Overview

Rust basics

Owning pointers

- Assignments of owning pointers

- `Box<T>` type

Borrowing pointers (`&T`)

Borrow checking details

Summary

Assignments of owning pointers

- ▶ to maintain the “single-owner” rule, an assignment of owning pointers in Rust *does not copy, but moves it* out of the righthand side, disallowing further use of it

```
x = y;  
// y can no longer be used
```

- ▶ e.g.,

```
fn foo() {  
  let a = S{x: ..., y: ...};
```



```
}
```

Assignments of owning pointers

- ▶ to maintain the “single-owner” rule, an assignment of owning pointers in Rust *does not copy, but moves it* out of the righthand side, disallowing further use of it

```
x = y;  
// y can no longer be used
```

- ▶ e.g.,

```
fn foo() {  
  let a = S{x: ..., y: ...};  
  ... a.x ...; // OK, as expected  
  ... a.y ...; // OK, as expected
```



```
}
```

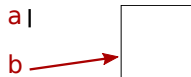
Assignments of owning pointers

- ▶ to maintain the “single-owner” rule, an assignment of owning pointers in Rust *does not copy, but moves it* out of the righthand side, disallowing further use of it

```
x = y;  
// y can no longer be used
```

- ▶ e.g.,

```
fn foo() {  
  let a = S{x: ..., y: ...};  
  ... a.x ...; // OK, as expected  
  ... a.y ...; // OK, as expected  
  // the reference moves out from a  
  let b = a;  
  
}
```



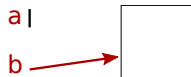
Assignments of owning pointers

- ▶ to maintain the “single-owner” rule, an assignment of owning pointers in Rust *does not copy, but moves it* out of the righthand side, disallowing further use of it

```
x = y;  
// y can no longer be used
```

- ▶ e.g.,

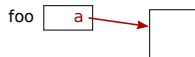
```
fn foo() {  
  let a = S{x: ..., y: ...};  
  ... a.x ...; // OK, as expected  
  ... a.y ...; // OK, as expected  
  // the reference moves out from a  
  let b = a;  
  a.x; // NG, the value has moved out  
  b.x; // OK  
}
```



Argument-passing also moves the reference

- ▶ passing a value to a function also moves the reference out of the source

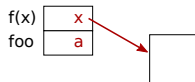
```
fn foo() {  
  let a = S{x: ..., y: ...};  
  ... a.x ...; // OK, as expected  
  ... a.y ...; // OK, as expected  
}
```



Argument-passing also moves the reference

- ▶ passing a value to a function also moves the reference out of the source

```
fn foo() {  
  let a = S{x: ..., y: ...};  
  ... a.x ...; // OK, as expected  
  ... a.y ...; // OK, as expected  
  // moves the reference out of a  
  f(a);  
  a.x; // NG, the reference has moved  
}
```



Exceptions to “assignment moves the reference”

- ▶ you may think the moving assignment

```
x = y;  
// y can no longer be used
```

contradicts what you have seen

- ▶ if it applies everywhere, does the following program violate it?

```
fn foo() -> f64 {  
    let a = 123.456;  
    // does the reference to 123.456 move out from a!?  
    let b = a;  
    a + 0.789 // if so, is this invalid!?  
}
```

- ▶ answer: no, it does *not* apply to primitive types like `i32`, `f64`, etc.
- ▶ a more general answer: it does not apply to data types that implement `Copy` trait

Copy trait

- ▶ define your struct with `#[derive(Copy, Clone)]` like

```
1 #[derive(Copy, Clone)]  
2 struct S { ... }
```

- ▶ and assignment or argument-passing of `S` makes a copy of the righthand side

```
fn foo() {  
  let a = S{x: ..., y: ...};  
  a.x; // OK, as expected  
  a.y; // OK, as expected
```



```
}
```

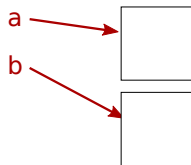
Copy trait

- ▶ define your struct with `#[derive(Copy, Clone)]` like

```
1 #[derive(Copy, Clone)]  
2 struct S { ... }
```

- ▶ and assignment or argument-passing of `S` makes a copy of the righthand side

```
fn foo() {  
    let a = S{x: ..., y: ...};  
    a.x; // OK, as expected  
    a.y; // OK, as expected  
    // the value is copied  
    let b = a;  
}
```



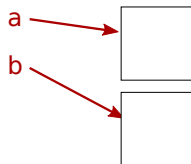
Copy trait

- ▶ define your struct with `#[derive(Copy, Clone)]` like

```
1 #[derive(Copy, Clone)]  
2 struct S { ... }
```

- ▶ and assignment or argument-passing of `S` makes a copy of the righthand side

```
fn foo() {  
    let a = S{x: ..., y: ...};  
    a.x; // OK, as expected  
    a.y; // OK, as expected  
    // the value is copied  
    let b = a;  
    a.x; // OK  
    b.x; // OK, too  
}
```



Copy types and the single-owner rule

- ▶ when a copy is made on every assignment or argument passing, the single-owner rule is trivially maintained
- ▶ below, we will only discuss types not implementing Copy trait (*non-Copy types*)

Box<T> makes an owning pointer

- ▶ making a pointer by `Box::new(v)` moves the reference out of v , too, and `Box::new(v)` becomes the owning pointer

```
fn foo() {  
    let a = S{x: ..., y: ...};  
    a.x; // OK, as expected  
    a.y; // OK, as expected  
}
```



Box<T> makes an owning pointer

- ▶ making a pointer by `Box::new(v)` moves the reference out of `v`, too, and `Box::new(v)` becomes the owning pointer

```
fn foo() {  
  let a = S{x: ..., y: ...};  
  a.x; // OK, as expected  
  a.y; // OK, as expected  
  // OK, now o is the owning pointer  
  let b = Box::new(a)  
  
}
```



Box<T> makes an owning pointer

- ▶ making a pointer by `Box::new(v)` moves the reference out of `v`, too, and `Box::new(v)` becomes the owning pointer

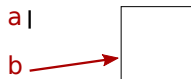
```
fn foo() {  
    let a = S{x: ..., y: ...};  
    a.x; // OK, as expected  
    a.y; // OK, as expected  
    // OK, now o is the owning pointer  
    let b = Box::new(a)  
    a.x; // NG, the value has moved out  
}
```



Box<T> makes an owning pointer

- ▶ making a pointer by `Box::new(v)` moves the reference out of `v`, too, and `Box::new(v)` becomes the owning pointer

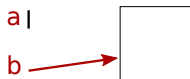
```
fn foo() {  
    let a = S{x: ..., y: ...};  
    a.x; // OK, as expected  
    a.y; // OK, as expected  
    // OK, now o is the owning pointer  
    let b = Box::new(a)  
    a.x; // NG, the value has moved out  
    (*b).x; // OK  
}
```



Box<T> makes an owning pointer

- ▶ making a pointer by `Box::new(v)` moves the reference out of `v`, too, and `Box::new(v)` becomes the owning pointer

```
fn foo() {  
  let a = S{x: ..., y: ...};  
  a.x; // OK, as expected  
  a.y; // OK, as expected  
  // OK, now o is the owning pointer  
  let b = Box::new(a)  
  a.x; // NG, the value has moved out  
  (*b).x; // OK  
  b.x; // OK. abbreviation of (*b).x  
}
```



Difference between T and `Box<T>`?

- ▶ as you have seen, the effect of

```
1 let b = a;
```

and

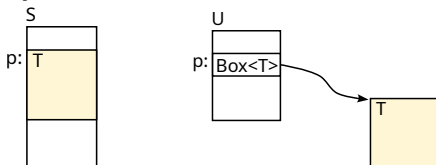
```
1 let b = Box::new(a);
```

look identical

- ▶ as far as data lifetime is concerned, it is in fact safe to say T and `Box<T>` are identical
- ▶ Rust have the distinction for
 - ▶ specifying data layout
 - ▶ specifying where data are allocated (stack vs. heap)

Data layout differences between T and $\text{Box}\langle T \rangle$

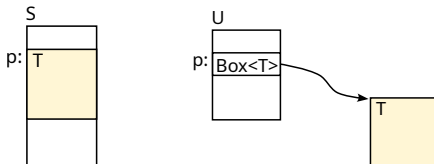
- ▶ S and U below have different data layouts
 - ▶ `struct S { ..., p: T , }` “embeds” a T into S
 - ▶ `struct U { ..., p: $\text{Box}\langle T \rangle$, }` has p point to a separately allocated T



- ▶ in particular, $\text{Box}\langle T \rangle$ is essential to define recursive data structures
 - ▶ `struct S { ..., p: S , }` is not allowed, whereas
 - ▶ `struct U { ..., p: $\text{Box}\langle U \rangle$, }` is
- ▶ note: U above can never be constructed; a recursive data structure typically looks like `struct U { ..., p: $\text{Option}\langle \text{Box}\langle U \rangle \rangle$, }`

Data layout differences between T and $\text{Box}\langle T \rangle$

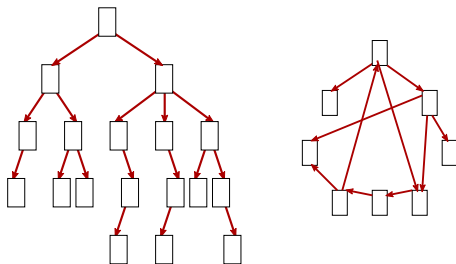
- ▶ the distinction is insignificant when discussing lifetimes



- ▶ in both cases, data of T (yellow box) is gone exactly when the enclosing structure is gone
- ▶ Rust spec also says it allocates T on stack and move it to heap when $\text{Box}\langle T \rangle$ is made
- ▶ again, it has nothing to do with lifetime (unlike C/C++)

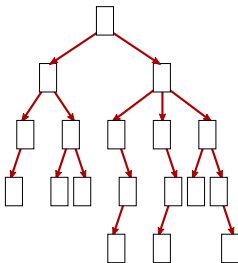
A (huge) implication of the single-owner rule

- ▶ with only owning pointers (T and $\text{Box}\langle T \rangle$),
 - ▶ you can make *a tree of T* ,
 - ▶ but you *cannot make a general graph of T* (acyclic or cyclic), where *a node may be pointed to by multiple nodes*
- ▶ if you want to make a graph of T , you use either
 - ▶ $\&T$ to represent edges, or
 - ▶ $\text{Vec}\langle T \rangle$ to represent nodes and $\text{Vec}\langle (\text{i32}, \text{i32}) \rangle$ to represent edges



The (huge) implication to memory management

- ▶ if there are only owning pointers (i.e., no borrowing pointers)
- ▶ *whenever an owning pointer is gone* (e.g.,
 - ▶ a variable goes out of scope or
 - ▶ a variable or field is overwritten),*the entire tree rooted from the pointer can be safely reclaimed*



Contents

Overview

Rust basics

Owning pointers

Assignments of owning pointers

`Box<T>` type

Borrowing pointers (`&T`)

Borrow checking details

Summary

- ▶ you can make any number of borrowing pointers to T ($\&T$) from T or $\text{Box}\langle T \rangle$
- ▶ both the owning pointer and borrowing pointers can be used at the same time

```
1 let a = S{x: .., y: ..};  
2 let b = &a;  
3 ... a.x + b.x ... // OK
```

- ▶ the issue is how to prevent a program from *dereferencing borrowing pointers after its owning pointer is gone*

Borrowers rule in action

- ▶ a borrowing pointer cannot be dereferenced after its owning pointer is gone

```
fn foo() -> i32 {  
    let c: &S; // a reference to S  
    { // an inner block  
  
    }  
}
```

c : &S

Borrowers rule in action

- ▶ a borrowing pointer cannot be dereferenced after its owning pointer is gone

```
fn foo() -> i32 {  
    let c: &S; // a reference to S  
    { // an inner block  
        let b: &S; // another reference  
  
    }  
}
```

c : &S

b : &S

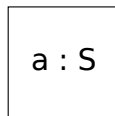
Borrowers rule in action

- ▶ a borrowing pointer cannot be dereferenced after its owning pointer is gone

```
fn foo() -> i32 {  
  let c: &S; // a reference to S  
  { // an inner block  
    let b: &S; // another reference  
    let a = S{x: ...}; // allocate S  
  
  }  
}
```

c : &S

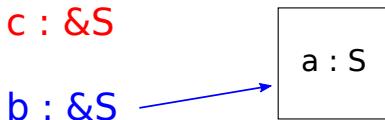
b : &S



Borrowers rule in action

- ▶ a borrowing pointer cannot be dereferenced after its owning pointer is gone

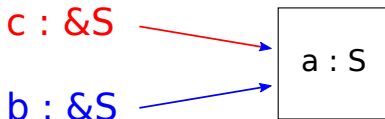
```
fn foo() -> i32 {  
  let c: &S; // a reference to S  
  { // an inner block  
    let b: &S; // another reference  
    let a = S{x: ...}; // allocate S  
    // OK (both a and b live only until the end of the inner block)  
    b = &a;  
  }  
}
```



Borrowers rule in action

- ▶ a borrowing pointer cannot be dereferenced after its owning pointer is gone

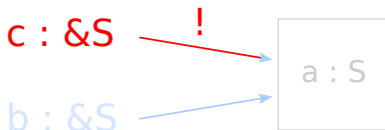
```
fn foo() -> i32 {  
  let c: &S; // a reference to S  
  { // an inner block  
    let b: &S; // another reference  
    let a = S{x: ...}; // allocate S  
    // OK (both a and b live only until the end of the inner block)  
    b = &a;  
    c = b; // dangerous (c outlives a)  
  }  
}
```



Borrowers rule in action

- ▶ a borrowing pointer cannot be dereferenced after its owning pointer is gone

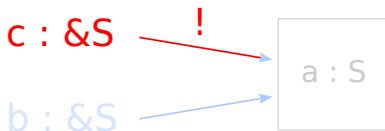
```
fn foo() -> i32 {  
    let c: &S; // a reference to S  
    { // an inner block  
        let b: &S; // another reference  
        let a = S{x: ...}; // allocate S  
        // OK (both a and b live only until the end of the inner block)  
        b = &a;  
        c = b; // dangerous (c outlives a)  
    } // a dies here, making c a dangling pointer  
}
```



Borrowers rule in action

- ▶ a borrowing pointer cannot be dereferenced after its owning pointer is gone

```
fn foo() -> i32 {  
  let c: &S; // a reference to S  
  { // an inner block  
    let b: &S; // another reference  
    let a = S{x: ...}; // allocate S  
    // OK (both a and b live only until the end of the inner block)  
    b = &a;  
    c = b; // dangerous (c outlives a)  
  } // a dies here, making c a dangling pointer  
  c.x // NG (deref a dangling pointer)  
}
```



A *mutable* borrowing reference (`&mut T`)

- ▶ you cannot modify data of type T through ordinary borrowing references `&T`

```
1 let a : S = S{x: 10, y: 20};  
2 let b : &S = &a;  
3 b.x = 100; // NG
```

- ▶ they are *immutable* references
- ▶ you can modify data only through a *mutable reference* (`&mut T`)

```
1 let mut a : S = S{x: 10, y: 20};  
2 let b : &mut S = &mut a;  
3 b.x = 100; // OK
```

- ▶ the difference is largely orthogonal to memory management

Contents

Overview

Rust basics

Owning pointers

- Assignments of owning pointers

- `Box<T>` type

Borrowing pointers (`&T`)

Borrow checking details

Summary

A technical remark about borrowers rule

- ▶ it's *not a creation* of a dangling pointer, *per se*, that is not allowed, but *dereferencing* of it
- ▶ *a slightly modified code below compiles without an error*, despite that `c` becomes a dangling pointer to `a` (as it is not dereferenced past `a`'s lifetime)

```
fn foo() -> i32 {  
    let c: &S; // a reference to S  
    { // an inner block  
        let b: &S; // another reference  
        let a = S{x: ...}; // allocate S  
        // OK (both a and b live only until the end of the inner block)  
        b = &a;  
        c = b; // dangerous (c outlives a)  
    } // a dies here, making c a dangling pointer  
    // c.x don't deref c  
}
```

A more precise statement of borrowers rule

1. for each borrowing reference ($\&T$ or $\&\text{mut } T$ type), Rust compiler determines *the lifetime of data it points to* (*referent lifetime*) as part of its static type

```
fn foo() -> i32 {  
    let c: &S; // → ??  
    {  
        let b: &S; // → ??  
        let a = S{x: ...};  
        b = &a;  
        c = b;  
    } // a dies here ( $\alpha$ )  
    c.x  
}
```

A more precise statement of borrowers rule

1. for each borrowing reference ($\&T$ or $\&\text{mut } T$ type), Rust compiler determines *the lifetime of data it points to* (*referent lifetime*) as part of its static type
2. assignment between borrowing pointers ($p = q$) equate their referent lifetimes

```
fn foo() -> i32 {  
    let c: &S; // → ??  
    {  
        let b: &S; // → ??  
        let a = S{x: ...};  
        b = &a;  
        c = b;  
    } // a dies here ( $\alpha$ )  
    c.x  
}
```

A more precise statement of borrowers rule

1. for each borrowing reference ($\&T$ or $\&\text{mut } T$ type), Rust compiler determines *the lifetime of data it points to (referent lifetime)* as part of its static type
2. assignment between borrowing pointers ($p = q$) equate their referent lifetimes

```
fn foo() -> i32 {  
    let c: &S; //  $\rightarrow ??$   
    {  
        let b: &S; //  $\rightarrow \alpha$   
        let a = S{x: ...}; // lives until  $\alpha$   
        b = &a; // b's referent lifetime = a's lifetime  
        c = b;  
    } // a dies here ( $\alpha$ )  
    c.x  
}
```

A more precise statement of borrowers rule

1. for each borrowing reference ($\&T$ or $\&\text{mut } T$ type), Rust compiler determines *the lifetime of data it points to (referent lifetime)* as part of its static type
2. assignment between borrowing pointers ($p = q$) equate their referent lifetimes

```
fn foo() -> i32 {  
    let c: &S; //  $\rightarrow \alpha$   
    {  
        let b: &S; //  $\rightarrow \alpha$   
        let a = S{x: ...}; // lives until  $\alpha$   
        b = &a; // b's referent lifetime = a's lifetime  
        c = b; // c's referent lifetime = b's referent lifetime  
    } // a dies here ( $\alpha$ )  
    c.x  
}
```


A more precise statement of borrowers rule

1. for each borrowing reference ($\&T$ or $\&\text{mut } T$ type), Rust compiler determines *the lifetime of data it points to (referent lifetime)* as part of its static type
2. assignment between borrowing pointers ($p = q$) equate their referent lifetimes
3. dereferencing a borrowing pointer p (e.g., $p.x$) is allowed only within the p 's referent lifetime

```
fn foo() -> i32 {  
    let c: &S; //  $\rightarrow \alpha$   
    {  
        let b: &S; //  $\rightarrow \alpha$   
        let a = S{x: ...}; // lives until  $\alpha$   
        b = &a; // b's referent lifetime = a's lifetime  
        c = b; // c's referent lifetime = b's referent lifetime  
    } // a dies here ( $\alpha$ )  
    c.x  
}
```

A more precise statement of borrowers rule

1. for each borrowing reference ($\&T$ or $\&\text{mut } T$ type), Rust compiler determines *the lifetime of data it points to (referent lifetime)* as part of its static type
2. assignment between borrowing pointers ($p = q$) equate their referent lifetimes
3. dereferencing a borrowing pointer p (e.g., $p.x$) is allowed only within the p 's referent lifetime

```
fn foo() -> i32 {  
    let c: &S; //  $\rightarrow \alpha$   
    {  
        let b: &S; //  $\rightarrow \alpha$   
        let a = S{x: ...}; // lives until  $\alpha$   
        b = &a; // b's referent lifetime = a's lifetime  
        c = b; // c's referent lifetime = b's referent lifetime  
    } // a dies here ( $\alpha$ )  
    c.x // NG (deref outside c's referent lifetime =  $\alpha$ )  
}
```

Programming with borrowing references

- ▶ programs using borrowing references must help compilers track their referent lifetimes
- ▶ this must be done for functions called from unknown places, function calls to unknown functions and data structures
- ▶ to this end, the programmer sometimes must annotate *reference types with their referent lifetimes*

References in function parameters

- ▶ problem: how to check the validity of functions taking references

```
1 fn p_points_q(p: &mut P, q: &Q) {  
2   p.x = q; // OK?  
3 }
```

*without knowing all its callers, and function calls
passing references*

```
1 let c = ...;  
2 {  
3   let a = Q{...};  
4   let b = &a;  
5   f(c, b);  
6 }  
7 ... c.x.y ... // OK?
```

without knowing the definition of f?

References in function return values

- ▶ problem: how to check the validity of functions returning references

```
1 fn return_ref(...) -> &P {  
2   ...  
3   let p: &P = ...  
4   ...  
5   p // OK?  
6 }
```

without knowing its all callers, and function calls receiving references from function calls

```
1 fn receive_ref() {  
2   ...  
3   let p: &P = return_ref(...);  
4   ...  
5   p.x // OK?  
6 }
```

References in data structures

- ▶ problem: how to check the validity of dereferencing a pointer obtained from a data structure

```
1 fn ref_from_struct() {  
2     ...  
3     let p: &P = a.p;  
4     ...  
5     p.x // OK?  
6 }
```

- ▶ what about functions taking data structures containing references and returning another containing references, etc.?

Reference type with a lifetime parameter

- ▶ to address this problem, Rust's borrowing reference types ($\&T$ or $\&\text{mut } T$) carry *lifetime parameter* representing their referent lifetimes
- ▶ syntax:
 - ▶ $\&'a T$: reference to “ T whose lifetime is $'a$ ”
 - ▶ $\&'a \text{ mut } T$: ditto; except you can modify data through it

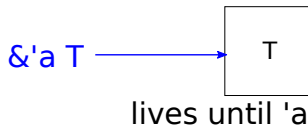


lives until 'a

- ▶ *every* reference carries a lifetime parameter, though there are places you can omit them
- ▶ roughly, you must write them explicitly in function parameters, return types, and struct/enum fields; and can omit them for local variables

Reference type with a lifetime parameter

- ▶ to address this problem, Rust's borrowing reference types ($\&T$ or $\&\text{mut } T$) carry *lifetime parameter* representing their referent lifetimes
- ▶ syntax:
 - ▶ $\&'a T$: reference to “ T whose lifetime is $'a$ ”
 - ▶ $\&'a \text{ mut } T$: ditto; except you can modify data through it



- ▶ *every* reference carries a lifetime parameter, though there are places you can omit them
- ▶ roughly, you must write them explicitly in function parameters, return types, and struct/enum fields; and can omit them for local variables

Attaching lifetime parameters to functions

- ▶ the following does not compile

```
1 fn foo(ra: &i32, rb: &i32, rc: &i32) -> &i32 {  
2     ra  
3 }
```

- ▶ with errors like

```
1 |  
2 | fn foo(ra: &i32, rb: &i32, rc: &i32) -> &i32 {  
3 |     ----      ----      ----      ^ expected named lifetime parameter  
4 |  
5 | = help: this function's return type contains a borrowed value, but the signature does not  
6 |         say whether it is borrowed from 'ra', 'rb', or 'rc'  
7 | help: consider introducing a named lifetime parameter  
8 |  
9 | fn foo<'a>(ra: &'a i32, rb: &'a i32, rc: &'a i32) -> &'a i32 {  
   |         ++++      ++          ++          ++          ++
```

Why do we need an annotation, *fundamentally*?

- ▶ without any annotation, how to know whether this is safe, *without knowing the definition of foo?*

```
1 {  
2   let r : &i32;  
3   let a = 123;  
4   {  
5     let b = 456;  
6     {  
7       let c = 789;  
8       r = foo(&a, &b, &c);  
9     }  
10  }  
11  *r  
12 }
```

- ▶ essentially, the compiler complains “tell me what kind of lifetime `foo(&a, &b, &c)` has”

Attaching lifetime parameters to functions

► syntax:

```
1 fn f<'a,'b,'c,...>(p0 : T0, p1 : T1, ...) -> Tr { ... }
```

T_0, T_1, \dots and T_r may use `'a`, `'b`, `'c`, ... as lifetime parameters (e.g., `&'a i32`)

► `f<'a,'b,'c,...>` is a function that takes parameters of respective lifetimes

One way to attach lifetime parameters

```
1 fn foo<'a>(ra: &'a i32, rb: &'a i32, rc: &'a i32) -> &'a i32
```

- ▶ effect: the return value is assumed to point to the shortest of the three
- ▶ why? generally, when Rust compiler finds `foo(x, y, z)`, it tries to determine `'a` so that it is contained in the lifetime of all $(x, y$ and $z)$
- ▶ as a result, our program does not compile, even if `foo(&a, &b, &c)` in fact returns `&a`

```
1 {  
2   let r: &i32;  
3   let a = 123;  
4   {  
5     let b = 456;  
6     {  
7       let c = 789;  
8       r = foo(&a, &b, &c); // 'a ← shortest of { $\alpha, \beta, \gamma$ } =  $\gamma$   
9       // and r's type becomes  $\&\gamma$  i32  
10      } // c's lifetime (=  $\gamma$ ) ends here  
11      } // b's lifetime (=  $\beta$ ) ends here  
12      *r // NG, as we are outside  $\gamma$   
13    } // a's lifetime (=  $\alpha$ ) ends here
```

An alternative

```
1 fn foo<'a, 'b, 'c>(ra: &'a i32, rb: &'b i32, rc: &'c i32) -> &'a i32
```

- ▶ signifies that the return value points to data whose lifetime is `ra`'s referent lifetime (and has nothing to do with `rb`'s or `rc`'s)
- ▶ for `foo(x, y, z)`, Rust compiler tries to determine `'a` so it is contained in the lifetime of `x`'s referent (therefore `'a = α`)
- ▶ as a result, the program we are discussing compiles

```
1 {  
2   let r: &i32;  
3   let a = 123;  
4   {  
5     let b = 456;  
6     {  
7       let c = 789;  
8       r = foo(&a, &b, &c); // 'a → shortest of { $\alpha$ } =  $\alpha$   
9       // and r's type becomes & $\alpha$  i32  
10    } // c's lifetime (=  $\gamma$ ) ends here  
11    } // b's lifetime (=  $\beta$ ) ends here  
12    *r // OK, as here is within  $\alpha$   
13  } // a's lifetime (=  $\alpha$ ) ends here
```

Types with lifetime parameters capture/constrain the function's behavior

- ▶ what if you try to fool the compiler by

```
1 fn foo<'a, 'b, 'c>(ra: &'a i32, rb: &'b i32, rc: &'c i32) -> &'a i32
2     rb
3 }
```

- ▶ the compiler rejects returning `rb` (of type `&'b`) when the function's return type is `&'a`
- ▶ in general, *the compiler allows assignments only between references having the same lifetime parameter*

Another example (make a reference between inputs)

► what if we rewrite

```
1 r = foo(&a, &b, &c);
```

into

```
1 bar(&mut r, &a, &b, &c);
```

with bar something like

```
1 fn bar(r: &mut &i32, a: &i32, b: &i32, c: &i32) {  
2   *r = a;  
3 }
```

Make a reference between inputs

- how to specify lifetime parameters so that
 1. `*r = a;` in `bar`'s definition is allowed, and
 2. we can dereference `*r` at the end of the caller?

```
1  {
2    let a = 123;
3    let mut r = &0;
4    {
5      let b = 456;
6      {
7        let c = 789;
8        bar(&mut r, &a, &b, &c); // r → ???
9      } // c's lifetime (=  $\gamma$ ) ends here
10   } // b's lifetime (=  $\beta$ ) ends here
11   *r // OK???
12 } // a's lifetime (=  $\alpha$ ) ends here
```


Answer

- ▶ again, we need to signify `r` points to `a` (and not `b` or `c` after `bar(&r, &a, &b, &c)`)
- ▶ a working lifetime parameter is the following

```
1 fn bar<'a, 'b, 'c>(r: &mut &'a i32, a: &'a i32,  
2                       b: &'b i32, c: &'c i32) {  
3     *r = a;  
4 }
```

References in data structures

- ▶ problem: how to check the validity of programs using data structure containing a borrowing reference

```
1 struct R {  
2     p: &i32  
3     ...  
4 }
```

and functions returning R

```
1 fn ret_r(a: &i32, b: &i32, c: &i32) -> R {  
2     R{p: a}  
3 }
```

or taking R (or reference to it)

```
1 fn take_r(r: &mut R, a: &i32, b: &i32, c: &i32) {  
2     r.p = a;  
3 }
```

References in data structures

- ▶ you cannot simply have a field of type $\&T$ in struct/enum like this

```
1 struct R {  
2     p: &i32  
3     ...  
4 }
```

- ▶ you need to specify the lifetime parameter of p , and signifies that R takes a lifetime parameter

```
1 struct R<'a> {  
2     p: &'a i32  
3     ...  
4 }
```

- ▶ $R<'a>$ represents R whose p field points $i32$ whose lifetime is $'a$

Attaching lifetime parameters to data structure

- ▶ say we like to have data structures

```
1 struct T { x: i32 }  
2 struct S { p: &T }
```

and a function

```
1 fn make_s(a: &T, b: &T) -> S { S{p: a} }
```

so that the following compiles

```
1 let s;  
2 let a = T{...};  
3 {  
4     let b = T{...};  
5     s = make_s(&a, &b);  
6 }  
7 s.p.x
```

- ▶ the compiler needs to verify `s.p` points to `a`, not `b`
- ▶ we have to signify that by appropriate lifetime parameters

Answer

- ▶ define `S<'a>` so
 - ▶ its `p`'s referent lifetime is `'a`

```
1 struct S<'a> { p: &'a T }
```

- ▶ define `make_s` so it returns `S<'a>` where `'a` is the referent lifetime of its *first* parameter

```
1 fn make_s(a: &'a T, b: &'b T) -> S<'a> {  
2     S{p: a}  
3 }
```

A more complex example Rust cannot verify

- ▶ say we now have data structures

```
1 struct T { x: i32 }  
2 struct S {  
3     p: &T,  
4     q: &T  
5 }
```

and a function

```
1 fn make_s(a: &T, b: &T) -> S { S{p: a, q: b} }
```

so that the following compiles

```
1 let s;  
2 let a = T{...};  
3 {  
4     let b = T{...};  
5     s = make_s(&a, &b);  
6 }  
7 s.p.x
```

- ▶ again, the compiler needs to verify `s.p` points to `a`, not `b`

Answer that I thought should work but didn't

- ▶ define `S` so
 - ▶ its `p` points to `T` of lifetime `'a` and
 - ▶ its `q` points to `T` of lifetime `'b`

```
1 struct S<'a, 'b> {  
2   p: &'a T,  
3   q: &'b T  
4 }
```

- ▶ define `make_s` so it returns `S<'a, 'b>` where `'a` is the lifetime of its first parameter, like

```
1 fn make_s(a: &'a T, b: &'b T) -> S<'a, 'b> {  
2   S{p: a, q: b}  
3 }
```

The compiler complains

```
1 [E0597] Error: 'b' does not live long enough
2   [command_36:1:1]
3   16 |     s = make_s(&a, &b);
4       |           ---
5       |           +---- borrowed value does not live long enough
6   17 | }
7       | -
8       | +---- 'b' dropped here while still borrowed
9   18 | s.p.x
10      | -----
11      | +----- borrow later used here
12
```

- ▶ I don't know what is the exact spec of Rust that rejects this program, but I hypothesize that to dereference `s` for any field (`p`), all fields must be alive

Contents

Overview

Rust basics

Owning pointers

- Assignments of owning pointers

- `Box<T>` type

Borrowing pointers (`&T`)

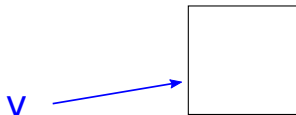
Borrow checking details

Summary

Why memory management is difficult

- ▶ every language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*

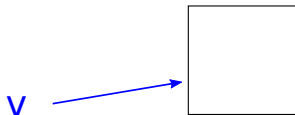
```
{  
  let v = T{x: ...};  
  ...  
}
```



Why memory management is difficult

- ▶ every language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- ▶ the problem would have been trivial if *you could reclaim v 's referent as soon as v goes out of scope*

```
{  
  let v = T{x: ...};  
  ...  
}
```



Why memory management is difficult

- ▶ every language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- ▶ the problem would have been trivial if *you could reclaim v 's referent as soon as v goes out of scope*

```
{  
  let v = T{x: ...};  
  ...  
} // OK to drop v's referent here?
```



Why memory management is difficult

- ▶ every language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- ▶ the problem would have been trivial if *you could reclaim v 's referent as soon as v goes out of scope*
- ▶ this is not the case, as *v 's referent may still be reachable from other variables when v goes out of scope*

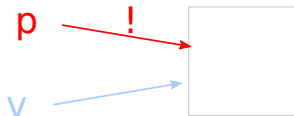
```
{  
  let v = T{x: ...};  
  ...  
}
```



Why memory management is difficult

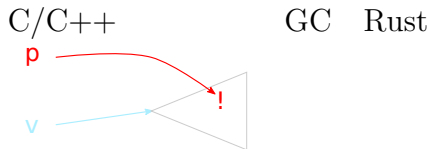
- ▶ every language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- ▶ the problem would have been trivial if *you could reclaim v 's referent as soon as v goes out of scope*
- ▶ this is not the case, as *v 's referent may still be reachable from other variables when v goes out of scope*

```
let p : &T;  
{  
  let v = T{x: ...};  
  ...  
  p = &v;  
} // v never used below, but its referent is  
... p.x ...
```



C vs. GC vs. Rust

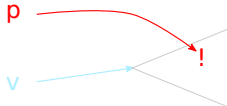
- ▶ C/C++ : it's up to you



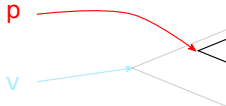
C vs. GC vs. Rust

- ▶ C/C++ : it's up to you
- ▶ GC : if it is reachable from other variables, I retain it for you

C/C++



GC

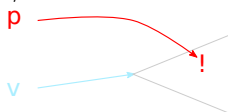


Rust

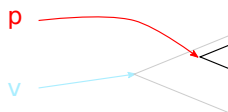
C vs. GC vs. Rust

- ▶ C/C++ : it's up to you
- ▶ GC : if it is reachable from other variables, I retain it for you
- ▶ Rust : when v goes out of scope,
 1. I reclaim T_v , all data *reachable from v through owning pointers*
 2. T_v may be reachable from other variables via borrowing references, but I nevertheless guarantees a reclaimed memory block is never accessed

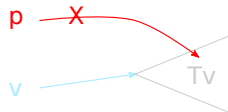
C/C++



GC

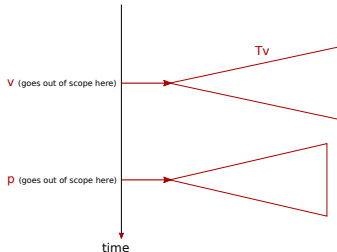


Rust



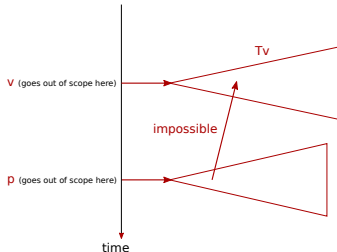
How Rust achieved it?

- recall the “single-owner rule,” which guarantees there is only one owning pointer to any node



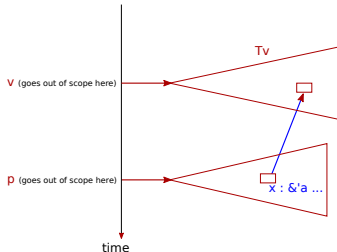
How Rust achieved it?

- ▶ recall the “single-owner rule,” which guarantees there is only one owning pointer to any node
- ▶ \Rightarrow there can be no *owning* pointers from outside T_v to inside T_v



How Rust achieved it?

- ▶ recall the “single-owner rule,” which guarantees there is only one owning pointer to any node
- ▶ \Rightarrow there can be no *owning* pointers from outside T_v to inside T_v
- ▶ \Rightarrow any such pointer must be a borrowing pointer



How Rust achieved it?

- ▶ recall the “single-owner rule,” which guarantees there is only one owning pointer to any node
- ▶ \Rightarrow there can be no *owning* pointers from outside T_v to inside T_v
- ▶ \Rightarrow any such pointer must be a borrowing pointer
- ▶ crucially, such a borrowing pointer must have a lifetime parameter of the referent

