Programming Languages (4)
Rust Memory Management

Kenjiro Taura

Contents

Overview

Rust basics

Assignments move the value

`Box<T>` (owning pointer) type

`&T` (borrowing pointer) type

Borrow checking details

Concluding Remarks

Contents

Overview

Rust basics

Assignments move the value

`Box<T>` (owning pointer) type

`&T` (borrowing pointer) type

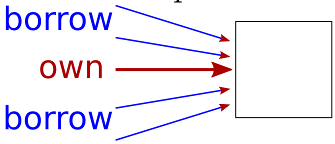
Borrow checking details

Concluding Remarks

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 outlive (live longer than) the data*
- note: 2nd rule is inaccurate and will be restated later
- ⇒ *it can safely reclaim the data when the owning pointer goes away*

"single-owner-multiple-borrowers rule"



The rule is enforced statically

- Rust maintains the rule *statically* (as opposed to *dynamically*)
- equivalently,
  - *compile-time* rather than at *runtime*
  - *before* execution not *during* execution

Ways outside the basics

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

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

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

Contents

Overview

Rust basics

Assignments move the value

Box<T> (owning pointer) type

&T (borrowing pointer) type

Borrow checking details

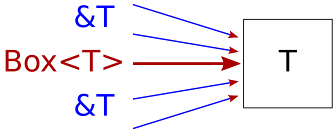
Concluding Remarks

Pointer-like data types in Rust

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

1. `&`T (pronounced *"ref T"*) : immutable borrowing pointer to data of T (through which you cannot modify it)
2. `&mut` T (*"ref mute T"*) : mutable borrowing pointer to data of T (through which you can modify it)
3. `Box<`T`>` (*box T*) : owning pointer to T
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 *and* `Box<`T`>`



[1] we 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 immutable borrowing pointer (through which you cannot modify the referent)
2. `&mut` e (of type `&mut` T) : a mutable borrowing pointer (through which you can modify the referent)
3. `Box::new(`e`)` (of type `Box<`T`>`) : an owning pointer

An example

```
{
  let mut a = S{x: ...}; // allocate memory for S
  let b: &S = &a; // make a borrowing pointer to a
  let c: &mut S = &mut a; // make a borrowing pointer to a
  let o: Box<S> = Box::new(a); // make an owning pointer to a
}

```

- 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

Assignments move the value

`Box<T>` (owning pointer) type

`&T` (borrowing pointer) type

Borrow checking details

Concluding Remarks

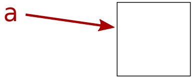
Assignments move the value

- to maintain only one "owner" pointer, an assignment in Rust *moves* the value out of righthand side, disallowing further use of it

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

- e.g.,

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



}
```



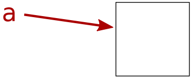
Assignments move the value

- to maintain only one "owner" pointer, an assignment in Rust *moves* the value out of 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



}
```



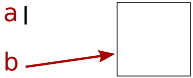
Assignments move the value

- to maintain only one "owner" pointer, an assignment in Rust *moves* the value out of 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 value moves away from a to b
  let b = a;


}
```



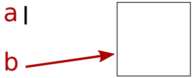
Assignments move the value

- to maintain only one "owner" pointer, an assignment in Rust *moves* the value out of righthand side, disallowing further use of it

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

- e.g.,

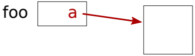
```
fn foo() {
  let a = S{x: ..., y: ...};
  ... a.x ...; // OK, as expected
  ... a.y ...; // OK, as expected
  // the value moves away from a to b
  let b = a;
  a.x; // NG, the value has moved out
  b.x; // OK
}
```



Argument-passing also moves the value

- passing a value to a function also moves the value out

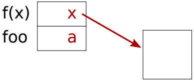
```
fn foo() {
  let a = S{x: ..., y: ...};
  ... a.x ...; // OK, as expected
  ... a.y ...; // OK, as expected

}
```



Argument-passing also moves the value

- passing a value to a function also moves the value out

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



Note: exceptions to “assignment moves the value”

- the value-moving assignment

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

  contradicts what you have seen

- does it apply to a primitive type, say `f64`?

```
fn foo() {
  let a = 123.456;
  // does the value 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

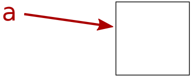
Copy trait

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

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

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

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



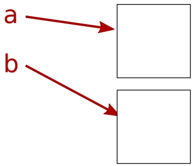
Copy trait

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

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

- and assignment or argument-passing of `S` makes a copy of 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;


}
```



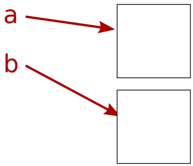
Copy trait

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

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

- and assignment or argument-passing of `S` makes a copy of 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*)

Contents

Overview

Rust basics

Assignments move the value

`Box<T>` (owning pointer) type

`&T` (borrowing pointer) type

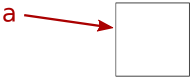
Borrow checking details

Concluding Remarks

Box<T> makes an owning pointer

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

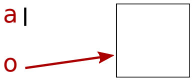
```
fn foo() {
  let a = S{x: ..., y: ...};
  a.x; // OK, as expected
  a.y; // OK, as expected
}
```



Box<T> makes an owning pointer

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

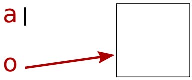
```
fn foo() {
  let a = S{x: ..., y: ...};
  a.x; // OK, as expected
  a.y; // OK, as expected
  // OK, now o becomes the owning pointer
  let o = Box::new(a)


}
```



Box<T> makes an owning pointer

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

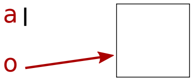
```
fn foo() {
  let a = S{x: ..., y: ...};
  a.x; // OK, as expected
  a.y; // OK, as expected
  // OK, now o becomes the owning pointer
  let o = Box::new(a)
  a.x; // NG, the value has moved out


}
```



Box<T> makes an owning pointer

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

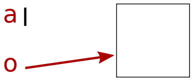
```
fn foo() {
  let a = S{x: ..., y: ...};
  a.x; // OK, as expected
  a.y; // OK, as expected
  // OK, now o becomes the owning pointer
  let o = Box::new(a)
  a.x; // NG, the value has moved out
  (*o).x; // OK

}
```



Box<T> makes an owning pointer

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

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



Make no mistake: making `Box::new(v)` does not affect lifetime

- `a = Box::new(v)` has no effect of making v live longer
- when `a` goes out of scope, v will be gone

```
fn foo() {
  ...
  {
    let a = S{...};
    let p = Box::new(a);
  } // --- S{...} will die here, too
}
```

just like

```
fn foo() {
  ...
  {
    let a = S{...};
    let p = a;
  } // --- S{...} will die here
}
```

Note: difference between T and `Box::<T>`?

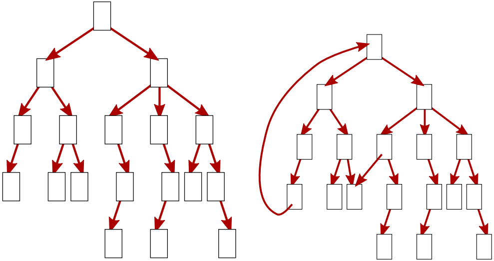
- for any value v of type T, you can only have one and only one (still usable) variable that refers to v, which is *either of type* T *or* `Box::<T>`
- in this sense, you can think of T as just another kind of pointer to T just like `Box::<T>`
- so is there any reason for Rust to have both T and `Box::<T>`?

T and `Box::<T>`

- the distinction becomes important when you reason about data layout
  - `struct S { p: T, ... }` "embeds" a `T` into `S`
  - `struct U { p: Box<T>, ... }` has `p` point to a separately allocated T
- in particular,
  - `struct S { p: S, ... }` is not allowed, whereas
  - `struct U { p: Box<U>, ... }` is
- the distinction is not important when discussing lifetimes (you can consider T a pointer without being confused)

A (huge) implication of the single-owner rule (1)

- with only owning pointers (T and `Box<`T`>`),
  - you can make *trees of T*,
  - but you *cannot make general graphs of T* (acyclic or cyclic), where *a node may be pointed to by multiple nodes*
- if you want to make graphs of T, you use either
  - `&`T to represent edges, or
  - `Vec<`T`>` to represent nodes and `Vec<(i32,i32)>` to represent edges



A (huge) implication of the single-owner rule (2)

- with only owning pointers, *no two names in scope ever refer to the same object* (*no aliasing*)
- `a` and `b` *never* refer to the same object

```
fn take_two(a : Box<T>, b : Box<T>) {
  ...
}
```

- a boon for the compiler
- a useful property to avoid mistakes, too

Contents

Overview

Rust basics

Assignments move the value

`Box<T>` (owning pointer) type

`&T` (borrowing pointer) type

Borrow checking details

Concluding Remarks

Borrowers rule in action

- a borrowing pointer should not outlive the data it points to (note: inaccurate; we will restate it later)

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



  }

}
```

c : &S

Borrowers rule in action

- a borrowing pointer should not outlive the data it points to (note: inaccurate; we will restate it later)

```
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

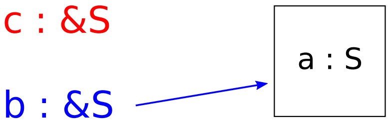
- a borrowing pointer should not outlive the data it points to (note: inaccurate; we will restate it later)

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



  }

}
```

Borrowers rule in action

- a borrowing pointer should not outlive the data it points to (note: inaccurate; we will restate it later)

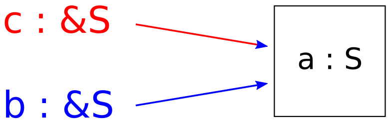
```
fn foo() -> i32 {
  let c: &S; // a reference to S
  { // an inner block
    let b: &S; // another reference
    let a = S{x: ...}; // allocate S
    // OK (both a and b live only until the end of the inner block)
    b = &a;

  }

}
```



Borrowers rule in action

- a borrowing pointer should not outlive the data it points to (note: inaccurate; we will restate it later)

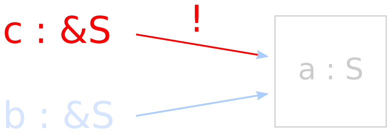
```
fn foo() -> i32 {
  let c: &S; // a reference to S
  { // an inner block
    let b: &S; // another reference
    let a = S{x: ...}; // allocate S
    // OK (both a and b live only until the end of the inner block)
    b = &a;
    c = b; // dangerous (c outlives a)
  }

}
```



Borrowers rule in action

- a borrowing pointer should not outlive the data it points to (note: inaccurate; we will restate it later)

```
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 should not outlive the data it points to (note: inaccurate; we will restate it later)

```
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) has an additional restriction

- a stronger restriction is imposed on &mut T
  - you cannot use the originating (owning) pointer (T or Box<T>) or
  - derive other borrowing pointers (mutable or not) from a mutable borrowing reference (&mut T)

  where a mutable borrowing reference is *active* in scope
- *active* ≈ may be used in future (omitting details)

```
fn mut_ref() {
  let mut a = S{x: ...};
  let m = &mut a; // make a mutable ref to a
  ... a.x ...;    // NG: cannot use a (the originating pointer)
  let d = &a;     // NG: cannot borrow from a either
  let c = m;      // NG: cannot derive another reference
  m.x             // --- m is active up to this point
  ... a.x ...;    // OK: as m no longer active here
}
```

A mutable borrowing reference enjoys no aliasing, too (even stronger one)

- like an owning pointer, a mutable reference also enjoys the no aliasing property
- even more strongly, it cannot alias with other borrowing references (mutable or not)
- `p` below cannot be an alias of any of others
- `q` and `r` may be an alias of each other

```
fn take_many(p: &mut T, q: &T, r: &T, a: T, b: Box<T>) {
  ...
}
```

- discussions below are focused on memory management, and apply both to immutable and mutable references

Working with `Box<T>` or `&T`

- `Box<T>` and `&T` are both pointers
- you might naturally wonder which one to use when
- generally, use `Box<T>` to *link data structures together*
- use `&T` to *work on existing data structures without any allocation or deallocation*
- for this reason, many functions that take data structures as input take `&T`

Contents

Overview

Rust basics

Assignments move the value

`Box<T>` (owning pointer) type

`&T` (borrowing pointer) type

Borrow checking details

Concluding Remarks

A subtle technical point about borrowers rule

- it's actually *not a creation* of a dangling pointer, *per se*, that is not allowed, but *dereferencing* of it
- in fact, *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 `&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 (α)
  c.x
}
```

A more precise statement of borrowers rule

1. for each borrowing reference (`&`T or `&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 (α)
  c.x
}
```

A more precise statement of borrowers rule

1. for each borrowing reference (`&`T or `&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; // → α
    let a = S{x: ...}; // lives until α
    b = &a; // b's referent lifetime = a's lifetime
    c = b;
  } // a dies here (α)
  c.x
}
```

A more precise statement of borrowers rule

1. for each borrowing reference (`&`T or `&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; // → α
  {
    let b: &S; // → α
    let a = S{x: ...}; // lives until α
    b = &a; // b's referent lifetime = a's lifetime
    c = b; // c's referent lifetime = b's referent lifetime
  } // a dies here (α)
  c.x
}
```

A more precise statement of borrowers rule

1. for each borrowing reference (`&`T or `&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; // → α
  {
    let b: &S; // → α
    let a = S{x: ...}; // lives until α
    b = &a; // b's referent lifetime = a's lifetime
    c = b; // c's referent lifetime = b's referent lifetime
  } // a dies here (α)
  c.x
}
```

A more precise statement of borrowers rule

1. for each borrowing reference (`&`T or `&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; // → α
  {
    let b: &S; // → α
    let a = S{x: ...}; // lives until α
    b = &a; // b's referent lifetime = a's lifetime
    c = b; // c's referent lifetime = b's referent lifetime
  } // a dies here (α)
  c.x // NG (deref outside c's referent lifetime = α)
}
```

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

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

without knowing all its callers, and function calls passing references

```
f(c, b);
... c.x ... // OK?
```

without knowing the definition of `f`*?*

References in function return values

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

```
fn return_ref(...) -> &P {
  ...
  let p: &P = ...
  ...
  p // OK?
}
```

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

```
fn receive_ref() {
  ...
  let p: &P = return_ref(...);
  ...
  p.x // OK?
}
```

References in data structures

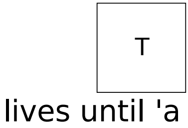
- problem: how to check the validity of dereferencing a pointer obtained from a data structure

```
fn ref_from_struct() {
  ...
  let p: &P = a.p;
  ...
  p.x // OK?
}
```

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

Reference type with a lifetime parameter

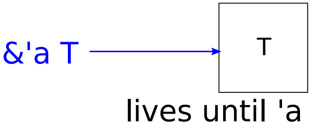
- to address this problem, Rust's borrowing reference types (`&T` or `&mut T`) carry *lifetime parameter* representing their referent lifetimes
- syntax:
  - `&'a T` : reference to "T whose lifetime is `'a`"
  - `&'a mut T` : ditto; except you can modify data through it



- *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

Reference type with a lifetime parameter

- to address this problem, Rust's borrowing reference types (`&`T or `&mut` T) carry *lifetime parameter* representing their referent lifetimes
- syntax:
  - `&'`a T : reference to "T whose lifetime is `'`a"
  - `&'`a `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

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

- with errors like

```
|
| fn foo(ra: &i32, rb: &i32, rc: &i32) -> &i32 {
|            ----      ----      ----     ^ expected named lifetime parameter
|
= help: this function's return type contains a borrowed value, but the signature does not
             say whether it is borrowed from `ra`, `rb`, or `rc`
help: consider introducing a named lifetime parameter
|
| 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`*?*

```
{
  let r : &i32;
  let a = 123;
  {
    let b = 456;
    {
      let c = 789;
      r = foo(&a, &b, &c);
    }
  }
}
p.x
```

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

Attaching lifetime parameters to functions

- syntax:

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

  $T_0, T_1, \cdots$ 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

- ```rust
  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`

```rust
{
  let r: &i32;
  let a = 123;
  {
    let b = 456;
    {
      let c = 789;
      r = foo(&a, &b, &c); // 'a ← shortest of {α, β, γ} = γ
      // and r's type becomes &γ i32
    } // c's lifetime (= γ) ends here
  } // b's lifetime (= β) ends here
  *r // NG, as we are outside γ
} // a's lifetime (= α) ends here
```
```

An alternative

- ```
  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` $= \alpha$)
- as a result, the program we are discussing compiles

```
{
  let r: &i32;
  let a = 123;
  {
    let b = 456;
    {
      let c = 789;
      r = foo(&a, &b, &c); // 'a → shortest of {α} = α
      // and r's type becomes &α i32
    } // c's lifetime (= γ) ends here
  } // b's lifetime (= β) ends here
  *r // OK, as here is within α
} // a's lifetime (= α) ends here
```
```

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

- what if you try to fool the compiler by

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

- 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

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

into

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

with `bar` something like

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

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?

```
{
  let a = 123;
  let mut r = &0;
  {
    let b = 456;
    {
      let c = 789;
      bar(&mut r, &a, &b, &c); // r → ???
    } // c's lifetime (= γ) ends here
  } // b's lifetime (= β) ends here
  *r // OK???
} // a's lifetime (= α) 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

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

References in data structures

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

```
struct R {
  p: &i32
  ...
}
```

and functions returning R

```
fn ret_r(a: &i32, b: &i32, c: &i32) -> S {
  S{p: a}
}
```

or taking R (or reference to it)

```
fn take_r(s: &mut S, a: &i32, b: &i32, c: &i32) {
  s.p = a;
}
```

References in data structures

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

```
struct R {
  p: &i32
  ...
}
```

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

```
struct R<'a> {
  p: &'a i32
  ...
}
```

- `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

```
struct T { x: i32 }
struct S {
  p: &T,
  q: &T
}
```

  and a function

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

  so that the following compiles

```
  let s;
  let a = T{...};
  {
    let b = T{...};
    s = make_s(&a, &b);
  }
  s.p.x
```

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

Answer

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

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

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

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

Note

- if it was

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

then a valid lifetime specification becomes

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

Contents

Overview

Rust basics

Assignments move the value

`Box<T>` (owning pointer) type

`&T` (borrowing pointer) type

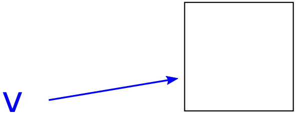
Borrow checking details

Concluding Remarks

Why memory management is difficult

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

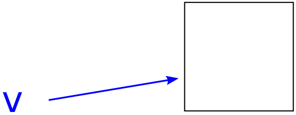
```
{
  let v = T{x:  ...};
  ...

}
```



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

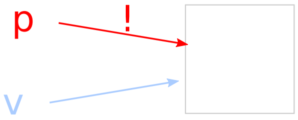
- *every* language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- the problem would have been trivial if *you could reclaim v's referent as soon as v goes out of scope*
- this is not the case, as *v's referent may still be reachable from other variables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

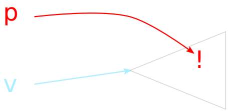
```
let p : &T;
{
  let v = T{x: ...};
  ...
  p = &v;
} // v never used below, but its referent is
... p.x ...
```



C vs. GC vs. Rust

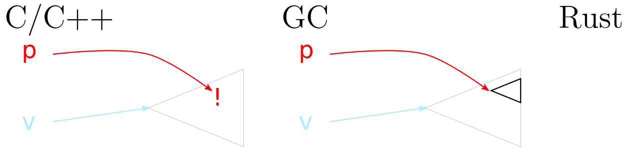
- C/C++ : it's up to you

C/C++ GC Rust



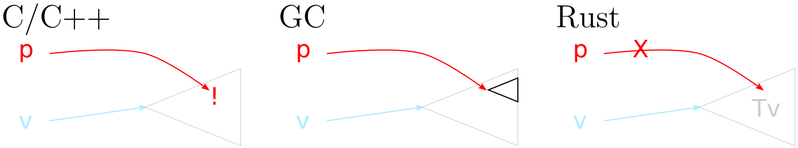
C vs. GC vs. Rust

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



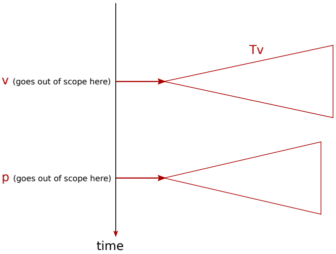
C vs. GC vs. Rust

- 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



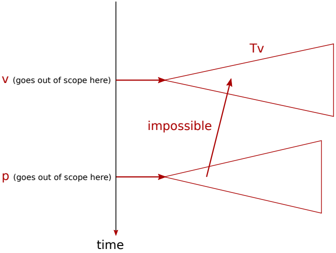
How Rust achieved it?

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



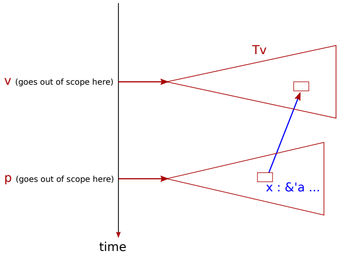
How Rust achieved it?

- 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



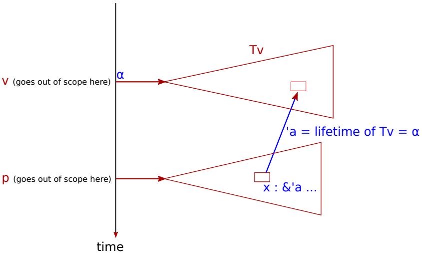
How Rust achieved it?

- 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



How Rust achieved it?

- recall the “single-owner rule,” which guarantees there is only one owning pointer to any node
- ⇒ there can be no *owning* pointers from outside T_v to inside T_v
- ⇒ any such pointer must be a borrowing pointer
- crucially, such a borrowing pointer must have a lifetime parameter of the referent
- as a result, a pointer that can reach T_v cannot be dereferenced after v goes out of scope