# Programming Languages (8)
# Rust Memory Management

Kenjiro Taura

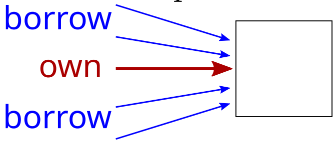
# Contents

# Contents

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

called *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

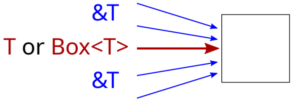
# Contents

# 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 `&`$T$.




[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 owning pointer
2. `Box::new(`$e$`)` (of type `Box<`$T$`>`) : an owning pointer
3. `&`$e$ (of type `&`$T$) : a borrowing pointer

# An example

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

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

# Contents

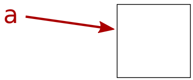
# Assignments do *not copy*, but *move*, the reference

- to maintain "single-owner" rule, an assignment in Rust *moves* the value 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: ...};



}
```



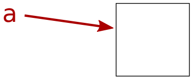
# Assignments do *not copy*, but *move*, the reference

- to maintain "single-owner" rule, an assignment in Rust *moves* the value 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



}
```



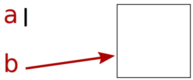
# Assignments do *not copy*, but *move*, the reference

- to maintain "single-owner" rule, an assignment in Rust *moves* the value 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;


}
```



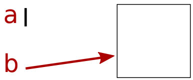
# Assignments do *not copy*, but *move*, the reference

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

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

- e.g.,

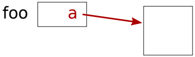
```
fn foo() {
  let a = S{x: ..., y: ...};
  ... a.x ...; // OK, as expected
  ... a.y ...; // OK, as expected
  // the reference moves out from a
  let b = a;
  a.x; // NG, the value has moved out
  b.x; // OK
}
```



# Argument-passing also moves the reference

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

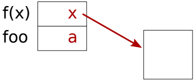
```
fn foo() {
  let a = S{x: ..., y: ...};
  ... a.x ...; // OK, as expected
  ... a.y ...; // OK, as expected


}
```



# 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
}
```



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

```
fn foo() {
  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

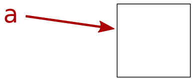
# Copy trait

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

```
#[derive(Copy, Clone)]
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
}
```



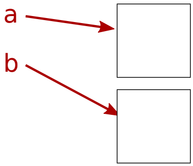
# Copy trait

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

```
#[derive(Copy, Clone)]
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;


}
```



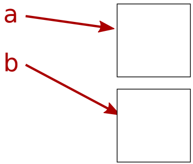
# Copy trait

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

```
#[derive(Copy, Clone)]
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
}
```



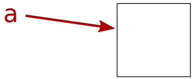
# Copy types and the single-owner rule

- 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 it becomes the owning pointer

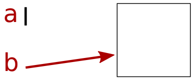
```
fn foo() {
  let a = S{x: ..., y: ...};
  a.x; // OK, as expected
  a.y; // OK, as expected



}
```



# Box<$T$> makes an owning pointer

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

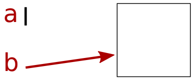
```
fn foo() {
  let a = S{x: ..., y: ...};
  a.x; // OK, as expected
  a.y; // OK, as expected
  // OK, now o is the owning pointer
  let b = Box::new(a)



}
```



# Box<$T$> makes an owning pointer

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

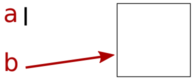
```
fn foo() {
  let a = S{x: ..., y: ...};
  a.x; // OK, as expected
  a.y; // OK, as expected
  // OK, now o is the owning pointer
  let b = Box::new(a)
  a.x; // NG, the value has moved out


}
```



# Box<$T$> makes an owning pointer

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

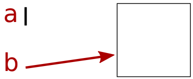
```
fn foo() {
  let a = S{x: ..., y: ...};
  a.x; // OK, as expected
  a.y; // OK, as expected
  // OK, now o is the owning pointer
  let b = Box::new(a)
  a.x; // NG, the value has moved out
  (*b).x; // OK

}
```



# Box<$T$> makes an owning pointer

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



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

- as you have seen, the effect of

```
let b = a;
```

and

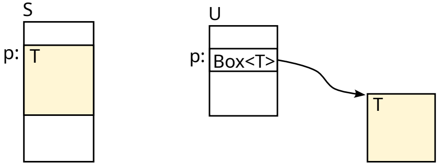
```
let b = Box::<T>::new(a);
```

look identical

- as far as data lifetime is concerned, it is in fact safe to think $T$ 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 `Box::<T>`

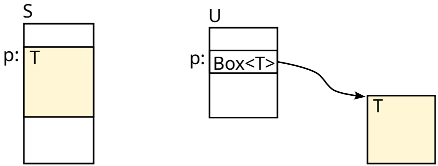
- `S` and `U` below have different data layouts
  - `struct S { ..., p:T, }` "embeds" a $T$ into `S`
  - `struct U { ..., p:Box::<T>, }` has `p` point to a separately allocated $T$




- in particular, `Box::<T>` is essential to define recursive data structures
  - `struct S { ..., p:S, }` is not allowed, whereas
  - `struct U { ..., p:Box::<U>, }` is

# Data layout differences between $T$ and `Box::<`$T$`>`

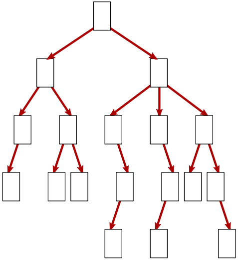
- the distinction is insignificant when discussing lifetimes

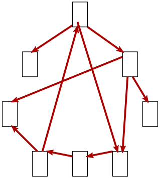



- in both cases, `p` (yellow box) is gone exactly when the enclosing structure is gone
- Rust spec also says it allocates $T$ on stack and move it to heap when `Box<`$T$`>` is made
- again, it has nothing to do with lifetime (unlike C/C++)

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

- with only owning pointers ($T$ and `Box<`$T$`>`),
  - 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
  - `Vec<`$T$`>` to represent nodes and `Vec<(i32,i32)>` to represent edges

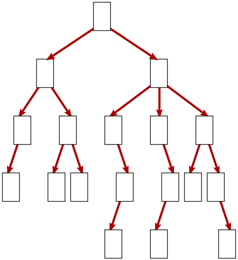




# The (huge) implication to memory management

- assume there are only owning pointers for now (forget about borrowing pointers)
- when
  - a variable of owning pointer type goes out of scope or
  - a variable or field of owning pointer type is overwritten,

  the entire tree rooted from the pointer can be safely reclaimed



# Contents

# Basics

- you can make a borrowing pointer to $T$ (`&`$T$) from $T$ or `Box<`$T$`>`
- both the owning pointer and borrowing pointers can be used

```
let a = S{x: .., y: ..};
let b = &a;
... 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

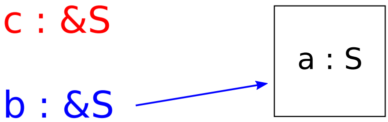
- a borrowing pointer cannot be dereferenced after its owning pointer is gone

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



  }

}
```

# Borrowers rule in action

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

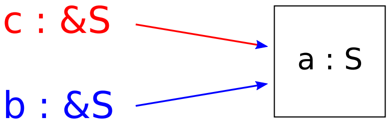
```
fn foo() -> i32 {
  let c: &S; // a reference to S
  { // an inner block
    let b: &S; // another reference
    let a = S{x: ...}; // allocate S
    // OK (both a and b live only until the end of the inner block)
    b = &a;

  }

}
```



# Borrowers rule in action

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

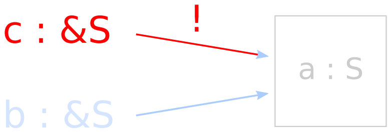
```
fn foo() -> i32 {
  let c: &S; // a reference to S
  { // an inner block
    let b: &S; // another reference
    let a = S{x: ...}; // allocate S
    // OK (both a and b live only until the end of the inner block)
    b = &a;
    c = b; // dangerous (c outlives a)
  }

}
```



# Borrowers rule in action

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

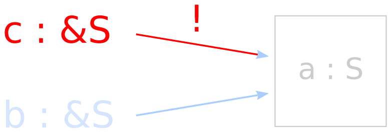
```
fn foo() -> i32 {
  let c: &S; // a reference to S
  { // an inner block
    let b: &S; // another reference
    let a = S{x: ...}; // allocate S
    // OK (both a and b live only until the end of the inner block)
    b = &a;
    c = b; // dangerous (c outlives a)
  } // a dies here, making c a dangling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

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

  - they are *immutable* references

- you can modify data through a mutable reference `&mut` $T$

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

# Additional restrictions on `&mut` $T$

- 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
}
```

# Contents

# 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 `&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: &mut P, q: &Q) {
  p.x = q; // OK?
}
```

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

```
let c = ...;
{
  let a = Q{...};
  let b = &a;
  f(c, b);
}
... 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

```
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

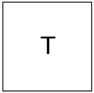
- problem: how to check the validity of dereferencing a pointer obtained from a data structure

```
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

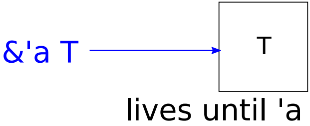



lives until 'a

- *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 `&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);
    }
  }
  *r
}
```

- 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,...>(p_0 : T_0, p_1 : T_1, ...) -> T_r { ... }
```

  $T_0, T_1, \cdots$ 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

- ```
  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`

```
{
  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) -> R {
  R{p: a}
}
```

or taking `R` (or reference to it)

```
fn take_r(r: &mut R, a: &i32, b: &i32, c: &i32) {
  r.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 }
```

  and a function

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

  so that the following compiles

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

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

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

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

# A more complex example Rust cannot verify

- say we now 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
```

- 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`

```
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}
}
```

# The compiler complains

```
[E0597] Error: ‘b‘ does not live long enough
    [command_36:1:1]
 16 │    s = make_s(&a, &b);
    .                  ---
    .                   +--- borrowed value does not live long enough
 17 │  }
    . _
    . +--- ‘b‘ dropped here while still borrowed
 18 │  s.p.x
    . -----
    .   +----- borrow later used here

```

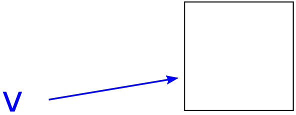
- I don't know what is the exact spec of Rust that rejects this program, but I hypothesize that to dereference `s` for any field (`p`), all fields must be alive

# Contents

# Why memory management is difficult

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

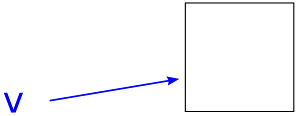
```
{
  let v = T{x:  ...};
  ...

}
```



# 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

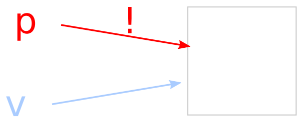
- *every* language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- the problem would have been trivial if *you could reclaim v's referent as soon as v goes out of scope*
- this is not the case, as *v's referent may still be reachable from other variables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

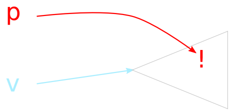
```
let p : &T;
{
  let v = T{x: ...};
  ...
  p = &v;
} // v never used below, but its referent is
... p.x ...
```



# C vs. GC vs. Rust

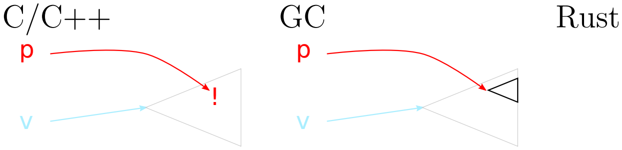
- C/C++ : it's up to you

C/C++ GC Rust



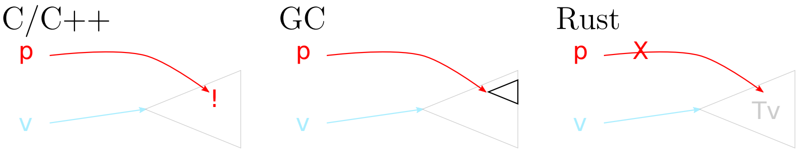
# C vs. GC vs. Rust

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



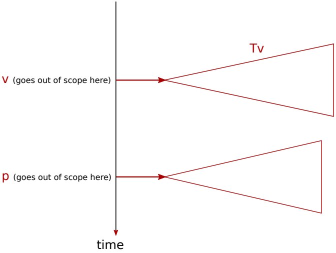
# C vs. GC vs. Rust

- 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



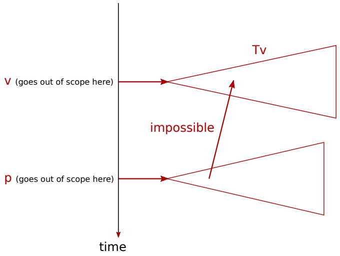
# How Rust achieved it?

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



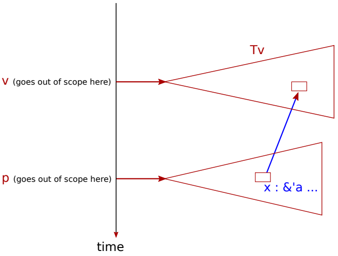
# How Rust achieved it?

- 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 $T_v$



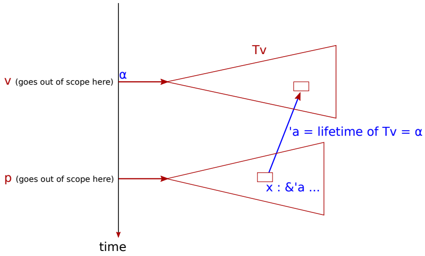
# How Rust achieved it?

- 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
- $\Rightarrow$ 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
- as a result, a pointer that can reach $T_v$ cannot be dereferenced after $v$ goes out of scope