COMP 737011 - Memory Safety and Programming Language Design

# Lecture 7: Rust OBRM


徐 辉

xuh@fudan.edu.cn

# Outline

- 1. Ownership
- 2. RAII and Lifetime
- 3. Unsafe Code

# 1. Ownership

# Motivation of Design

- Dangling pointer is unacceptable
- Causal of dangling pointer?
  - Mannual reclaim
    - we should prevent such manual reclaim
  - Automatic reclaim
    - alias analysis is NP-hard
    - shared pointer or gc is inefficient
- Rust comes to rescue

# Overall Idea of Ownership

- Each object is owned by one variable.
  - Recall C++ uniptr<T>
- Ownership can be borrowed in two mode:
  - immutable: read only
  - mutable: read/write
- Exclusive mutability principle
  - two variables should not share mutable access to the same object at the same program point.
- Restrict the complexity of the alias analysis problem
  - we only need to trace the mutable pointer for each object
    - only one mutable pointer at each program point
  - only mutable pointers can lead to dangling pointers

# Ownership & Borrowing

- Borrowed ownership will be returned automatically if no longer used.

```
fn main(){
   let mut alice = Box::new(1);
   let bob = alice;
   println!("bob:{}", bob);
   println!("alice:{}", alice);
}
```

alice owns the object

transfer the vector to bob; alice loses the ownership

```
fn main(){
   let mut alice = Box::new(1);
   let bob = &alice;
   println!("bob:{}", bob);
   println!("alice:{}", alice);
}
```

bob borrows the ownership

bob returns the ownership to alice automatically

# Move Operator (=)

- If a type is not Copy (trait), move transfers the ownership.
  - e.g., Box<T> is not copy
- If a type is Copy, move does not transfer the ownership and only copies the value

```
fn main(){
  let mut alice = 1;
  let bob = alice;
  println!("bob:{}", bob);
  println!("alice:{}", alice);
}
```

alice owns the object

copy the object to bob

# Which Type Can be Copy?

- Primitive types on stack
- Composite types with all fields implementing copy
- How to (deep) copy objects of other non-Copy types:
  - implement Clone (trait)
  - each Copy type is also Clone.

```
fn main(){
  let mut alice = Box::new(1);
  let bob = alice.clone();
  println!("bob:{}", bob);
  println!("alice:{}", alice);
}
```

alice owns the object

clone the object for bob

# Mutability

mutable object

```
let mut alice = 1;
alice+=1;
```

```
let alice = 1;
alice+=1;
```

mutable borrow

```
let mut alice = 1;
let bob = &mut alice;
*bob+=1;
```

```
let mut alice = 1;
let bob = &alice;
*bob+=1;
```

```
let alice = 1;
let bob = &mut alice;
*bob+=1;
```

# Mutability cont'd

```
let mut alice = 1;
let mut carol = 1;
let mut bob = &mut alice;
*bob+=1;
bob = &mut carol;
*bob+=1;
```

mutable object bob + mutable borrow

```
let mut alice = 1;
let mut carol = 1;
let bob = &mut alice;
*bob+=1;
bob = &mut carol;
*bob+=1;
```

immutable object bob + mutable borrow

bob cannot be modified

# Debug Ownership Conflict

```
fn main(){
  let mut alice = 1;
  let bob = &mut alice;
  println!("bob:{}", bob);
  println!("alice:{}", alice);
}
```

mutable borrow

bob returns the ownership

```
fn main(){
  let mut alice = 1;
  let bob = &mut alice;
  println!("alice:{}", alice);
  println!("bob:{}", bob);
}
```

exclusive mutability

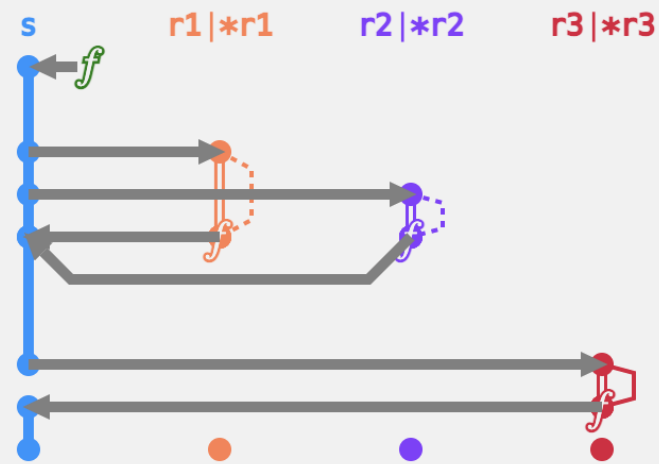
# Rustviz Project: Visualize The Lifetime

*Hover over timeline events (dots), states (vertical lines), and actions (arrows) for extra information.*

```rust
fn main(){
    let mut s = String::from("hello");

    let r1 = &s;
    let r2 = &s;
    assert!(compare_strings(r1, r2));


    let r3 = &mut s;
    clear_string(r3);
}
```




Marcelo Almeida, Cyrus Omar, *et. al.*, "RustViz: Interactively Visualizing Ownership and Borrowing." In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1-10. 2022.

# Pros and Cons

- Benefit
  - Compile-time prevention of shared mutable aliases
- When shared mutability is a Must…
  - Such as double linked lists?
  - Two options (we will discuss later):
    - shared pointer (reference counter)
    - unsafe code

# 2. RAII and Lifetime

RAII: Resource Acquisition is Initialization

# Idea of RAII

- Ties resources to object lifetime
- Object/resource is initialized/allocated once created
  - all pointers refer to particular objects
  - no raw or dangling pointers
  - no uninitialized memory
- Resource deallocation is done during object destruction by the destructor
  - no manual deallocation is needed
  - achieved through static lifetime inference

# Lifetime

- Each object has a lifetime constraint
- The object is reclaimed automatically after death
- A variable cannot borrow an object with a shorter lifetime

```
fn main(){
  let alice;
  {
    let bob = 5;
    alice = &bob;
  }
  println!("alice:{}", alice);
}
```

bob lives in this subscope

alice points to an expired bob

# Review Move for Non-Copy Types

- Extend the lifespan of the object on heap

```
fn test(){
    let alice;
    {
        let bob = Box::new(1);
        alice = bob;
    }
    println!("alice:{}", alice);
}
```

transfer the ownership to alice

```
fn testret() -> Box<u64>{
    Box::new(1)
}

let r = testret();
println!("return:{}", r);
```

move the ownership to the ret value

# Lifetime Declaration

- Lifetime should be declared during function declaration

```
fn stringcmp(){
    let str1 = String::from("alice");
    let str2 = String::from("bob111");
    let result = longer(&str1, &str2);
    println!("The longer string is {}", result);
}

fn longer<'a>(x:&'a String, y:&'a String)->&'a String{
    if x.len()>y.len(){
        x
    } else {
        y
    }
}
```

# Partial Order of Lifetime

- <'a: 'b, 'b> means a is relatively larger than b

```
fn stringcmp(){
    let str1 = String::from("alice");
    let result;
    //{
        let str2 = String::from("bob111");
        result = longer(&str1, &str2);
    //}
    println!("The longer string is {}", result);
}

fn longer<'a:'b,'b>(x:&'a String, y:&'b String)->&'b String{
    if x.len()>y.len(){
        x
    } else {
        y
    }
}
```

The return value cannot be 'a. Why?

# Non-lexical Lifetime

- The default mode is non-lexical unless necessary
- Rust compiler tries to minimize the lifespan

```
'a: { let str1 = "alice";
    'b: { let str2 = "bob";
        'c: { let result = longer(str1,str2);
              println!("The longer string is {}", result);
        }
    }
}
```

# Lifetime Elision to Be More Ergonomic

- Lifetime declaration can be elided if
  - there is only one input lifetime position
  - there are multiple positions, but one is &self or &mut self
    - assign the lifetime of self to elided output lifetimes

```
fn foo<'a>(s: &'a str, until: usize) -> &'a str;
```

```
fn foo(s: &str, until: usize) -> &str;
```

```
fn foo(s: &str, t: &str) -> &str; // ILLEGAL
```

```
fn foo<'a, 'b>(&'a mut self, t: &'b str) -> &str;
```

```
fn foo (&mut self, t: &str) -> &str;
```

# More About Lifetime

- A static object means it lives for the entire lifetime
  - use the reserved lifetime 'static
  - all strings are static by default
- The lifetime can be unbounded
  - more flexible than static during lifetime inference

```
let s: &str = "hello world";
```

⇩ equivalent to

```
let s: &'static str = "hello world";
```

```
fn foo<'a>() -> &'a str;
```

unbounded lifetime

# Automatic Reclaim/Drop

- Objects of Copy type (stack) can be reclaimed automatically
- For other objects with heap data?
  - Drop (trait) unused objects by calling the destructor
  - Recursively call the destructor of each field
- Drop and Copy are exclusive in Rust
  - Box<T> is Drop trait

```
struct MyType {a:u8, b:Box<u64>}

impl Drop for MyType {
    fn drop(&mut self){
        println!("dropping MyType object...");
    }
}
fn testdrop(){
    //create an object via "struct literal"
    let v = MyType {a:1, b:Box::new(2)};
}
```

automatically executed when the lifetime ends

# Option<T> for Uninitialized Objects

- Option: an enumerate type
  - Some(T): the object type
  - None: if the object is uninitialized (null pointer)

```
pub enum Option<T> {
    None,
    Some(T),
}

let v = Some(…)
match v.next {
    Some(n) => …,
    None => panic!(),
}
```

# Example with a Singly-linked List

```
struct List{
    val: u64,
    next: Option<Box<List>>,
}

fn foo(){
    let mut l = List{val:1, next:None};
    l.next = Some(Box::new(List{val:2, next:None}));
    match l.next {
        Some(ref mut n) =>
            n.next = Some(Box::new(List{val:3, next:None})),
        None => panic!(),
    }
    let mut h = &l;
    loop {
        println!{"{}", h.val};
        match h.next {
            Some(ref n) => h = n,
            None => break,
        }
    }
}
```

# RAII for Thread Panic

- In a multi-threaded application, what happens when one thread exit exceptionally ?
  - abort: directly terminate the thread
  - panic: perform stack unwinding before exit
- Importance of RAII during stack unwinding
  - release locks (mutex)
  - release opened file descriptors
  - release allocated memories on heap

# Sample Multi-thread Program

- When a spawned thread panics, unwind its stack
- The main thread continues execution
- Ineffective for fatal errors: e.g., stack overflow

```
fn main() {
    let handle = thread::spawn(|| {
        for i in 0..5 {
            println!("new thread print {}", i);
            thread::sleep(Duration::from_millis(10));
        }
        panic!();
        //recursive();
    });
    for i in 0..10 {
        println!("main thread print {}", i);
        thread::sleep(Duration::from_millis(10));
    }
    handle.join();
}
```

# 3. Unsafe Rust

# Problem for Double-Linked List

- A node is owned by both its prev and next node
- Violate exclusive mutability


```
struct List{
    val: u64,
    prev: Option<Box<List>>,
    next: Option<Box<List>>,
}
```

# Unsafe

- Operations that may lead to undefined behaviors are unsafe
  - Dereference raw pointers
  - Call unsafe functions
  - Call functions of foreign language (FFI)
- Using unsafe code within the unsafe scoop

```
let mut num = 5;
let r1 = &num as *const i32;
println!("r1 is: {}", unsafe { *r1 });
```

r1 is a raw pointer

raw pointer dereference

Dereference raw pointers

```
unsafe fn foo() {
    let addr = 0x012345usize;
    let r = addr as *const i32;
}
unsafe { foo(); }
```

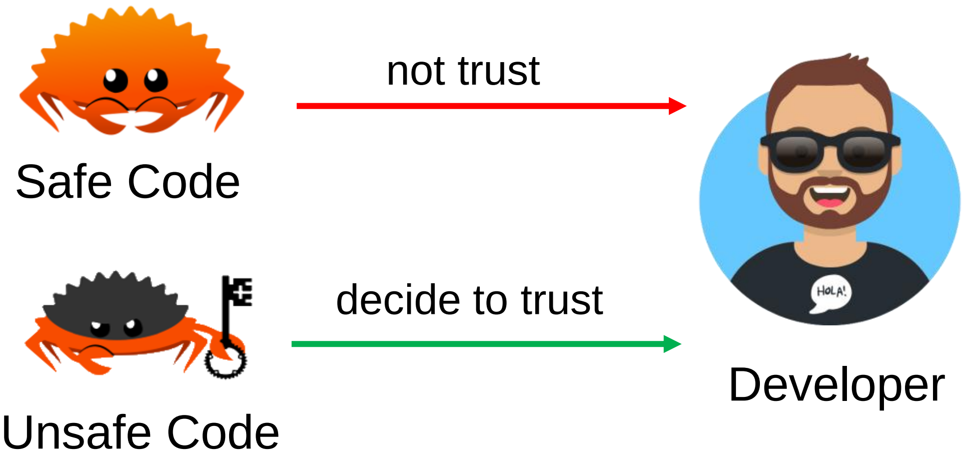
define an unsafe function

call the unsafe function

Call unsafe functions

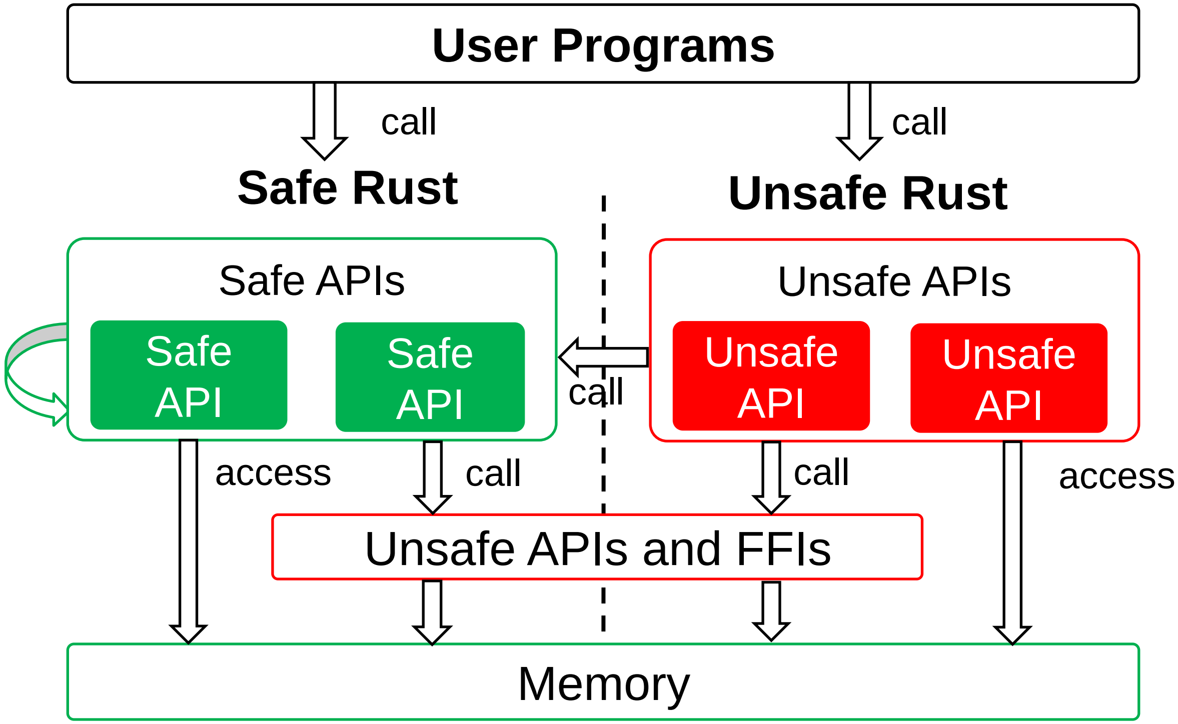
# Trust Model

- Rust does not trust developers, so only safe code is allowed
- If a developer declares he knows the risk, Rust will trust him



# Principle of Rust

- Safe API should not incur undefined behaviours
- Interior unsafe: wrap unsafe code into safe APIs
- Avoid using unsafe code unless necessary



# Unsafe Version with Raw Pointers

- The objects pointed by raw pointers are not owned
- The resource may not be dropped automatically
- Prone to dangling pointers (Not RAII)

```
struct List{
    val: u64,
    next: *mut List,
    prev: *mut List,
}

fn foo(){
    let mut l = List{val:1, next:null_mut(), prev:null_mut()};
    l.next = &mut List{val:2, next:null_mut(), prev:null_mut()};
    unsafe {
        let mut cur = &mut *(l.next);
        cur.prev = &mut l;
        cur.next = &mut List{val:3, next:null_mut(), prev:null_mut()};
        (*(cur.next)).prev = cur;
    }
}
```

# Solution Hint with RC<T> and RefCell<T>

- RC<T>: single-thread reference-counting pointer
  - enables shared immutable aliases
  - can mutate only if counter = 1 (cannot solve the list problem)
- RefCell<T>: a mutable memory location
  - convert immutable to mutable

```
struct List{
    val: u64,
    prev: Option<Rc<RefCell<List>>>,
    next: Option<Rc<RefCell<List>>>,
}
```

# RC<T>

- Reference counter for shared aliases
- Mutate via get_mut()
  - mutual exclusion during compile time
  - if cloned, get_mut() returns None during run time

```
fn main(){
    let mut x = Rc::new(1);
    //let _y = Rc::clone(&x);
    let t1 = Rc::get_mut(&mut x).unwrap();
    //let t2 = Rc::get_mut(&mut x).unwrap();
    *t1 = 2;
    assert_eq!(*x, 2);

    let _y = Rc::clone(&x);
    assert!(Rc::get_mut(&mut x).is_none());
}
```

if cloned, get_mut() returns None

compile error

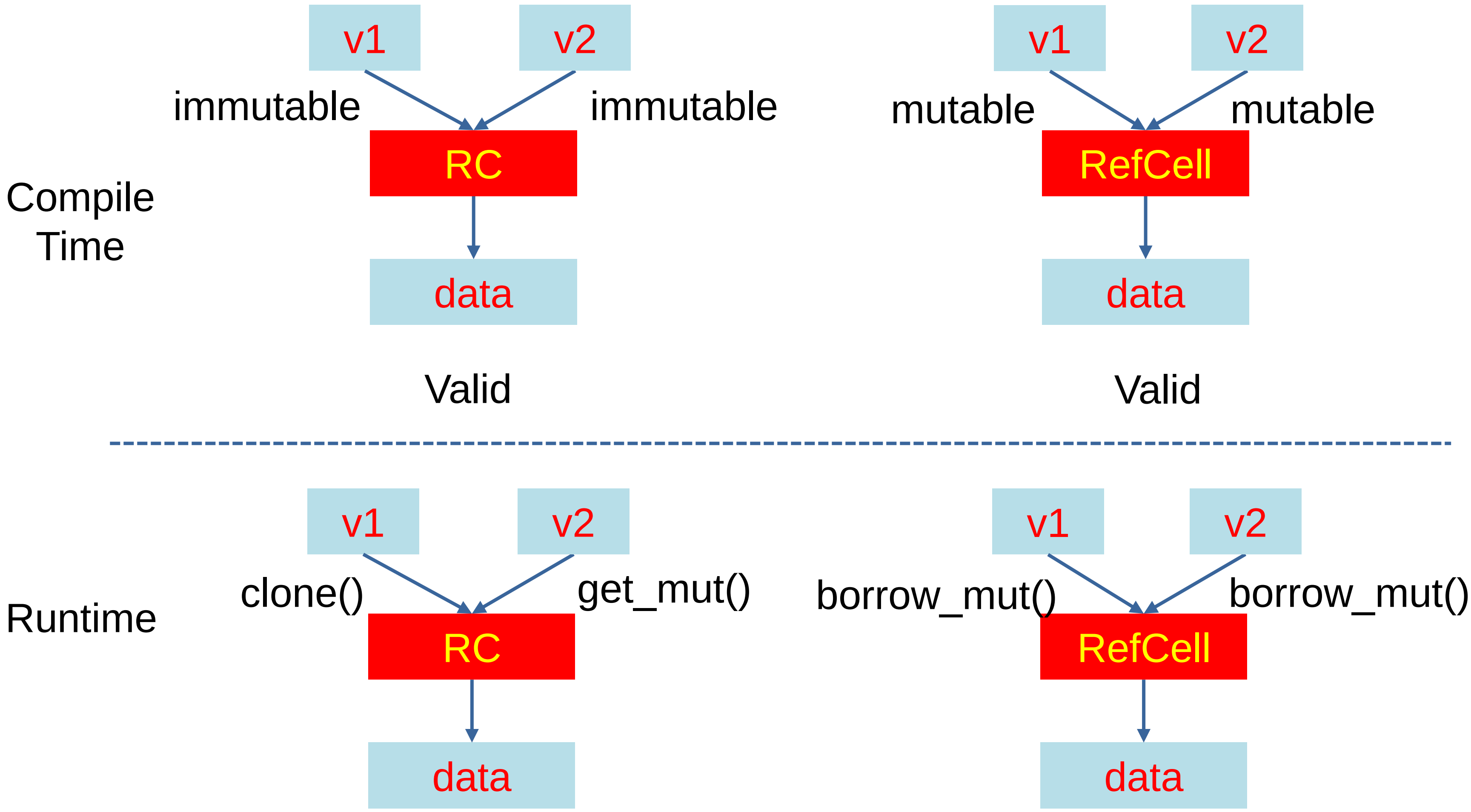
# RefCell<T>

- A mutable memory location
- Perform borrow check during runtime

```
fn testrefcell(){
    let x = RefCell::new(Box::new(1));
    {
        let mut y = x.borrow_mut();
        //let z = x.borrow_mut();
        *(*y) = 2;
    }
    assert_eq!(2, *(*x.borrow()));
}
```

panic during runtime

# RC vs RefCell




get_mut() may return None if cloned

the second borrow_mut() triggers panic

# In-class Practice

1) Implement a double linked list with Safe Rust
   - insert
   - delete
   - search
2) Implement a binary search tree with Safe Rust
   - insert
   - search