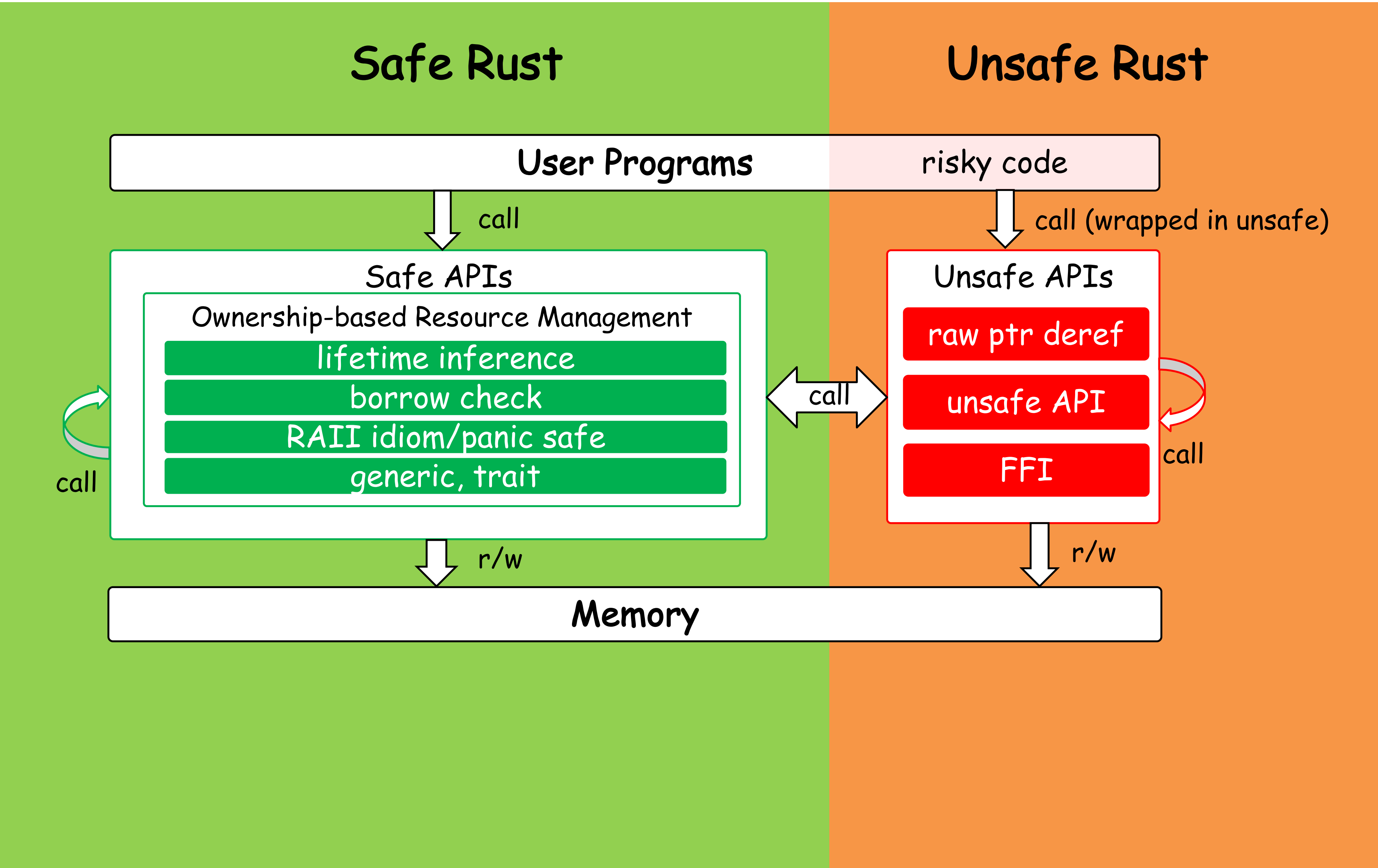
COMP 737011 - Memory Safety and Programming Language Design

# Lecture 6: Rust OBRM

徐 辉
xuh@fudan.edu.cn

# Rust Overview



# Outline

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

# 1. Ownership

# Motivation of Design

- Dangling pointer is unacceptable
- Causal of dangling pointer?
  - memory reclaim or object destruction
    - manual reclaim of heap data
      - we should prevent such manual reclaim
      - automatic reclaim => garbage collection or else?
    - stack object destruction
      - there could be other aliases to part of the object
      - alias analysis is NP-hard
      - shared pointer? inefficient
- Rust comes to rescue

# Idea

- Ownership: each object is owned by one variable.
- Borrow: the ownership can be borrowed by other variables in two mode:
  - immutable: read only
  - mutable: read/write
- Key rules: exclusive mutability
  - two variables cannot share mutable access to the object at the same time.
- Benefit: make the alias analysis problem much easier, why?
  - only mutable pointers can lead to dangling pointers
  - we only need to trace the mutable pointer for each object

# 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

alice is mutable

```
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;
```

bob is mutable

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

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

# What if…

```
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

# Pros and Cons

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

# 2. RAII and Lifetime

Resource Acquisition is Initialization

# Idea of RAII

- Ties resources to object lifetime
- Resource allocation is done during object creation by the constructor
  - all pointers refer to specific objects
  - no raw or dangling pointers
  - even 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);
}
```

# 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);
}
```

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

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

# Lifetime Declaration

- Lifetime constraint can be lexical 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 lifetime a>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
    }
}
```

# 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

- Sometimes lifetime declaration can be elided
  - exactly one input lifetime position
  - multiple positions, but one is &self or &mut self
- Assign the lifetime to elided output lifetimes.

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

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

# More About Lifetime

- Static, e.g., all str
- Unbounded

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

is equivalent to the following one

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

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

# Constructor

- Use struct/enum to define a new type
- Create an instance of a user-defined type by explicitly passing values to each field

```
struct MyType1 {
    a:u8
    b:u64,
}
let v = MyType1 {a:1, b:2};
```

```
enum MyType3{
    a(u8),
    b(u64),
}
let v = MyType1::a(1);
```

```
struct MyType2 {
    a:u8
    b:Box<u64>,
}
let v = MyType2 {a:1, Box::new(2)};
```

# Automatic Reclaim

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

```
struct Droppable{}

impl Drop for Droppable {
    fn drop(&mut self){
        println!("dropping...");
    }
}
fn testdrop(){
    let mut alice = Droppable{};
}
```

# Recursive Drop

- Recursively call the destructor of each field
- Rust prevent manually calling Drop::drop()
- mem::drop() is another different function
  - can invoke Drop::drop() by consumes the ownership

```
struct Droppable{ }
struct ParentDrop{ a:Droppable, b:Droppable,}

impl Drop for Dropable {
    fn drop(&mut self){
        println!("dropping...");
    }
}

impl Drop for ParentDrop {
    fn drop(&mut self){
        println!("parent dropping...");
    }
}
let mut alice = ParentDrop{a:Droppable{},b:Droppable{}};
```

# Some Limitations of So Far…

- RAII prevents uninitialized data, but sometimes uninitialized objects are needed
  - e.g., linked list
  - we can append the list dynamically
  - how to represent the next pointer?
- Shared mutable aliases are needed
  - e.g., double-linked list

# Use Options for Uninitialized Objects?

- Options: an enumeration type
  - Some(T): the object type
  - None: if the object is uninitialized

```
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 standard(){
    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-threaded 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

# Unsafe

- Dereference raw pointers
- Call unsafe functions
- Call functions of foreign language (FFI)

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

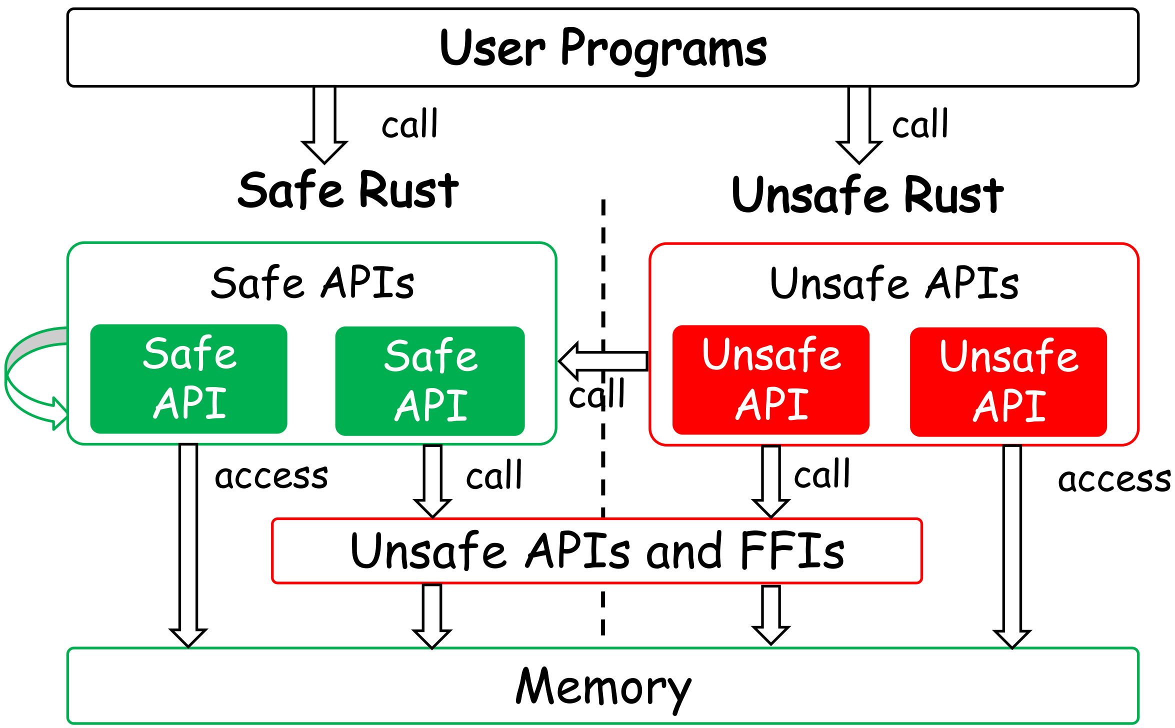
Dereference raw pointers

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

Call unsafe functions

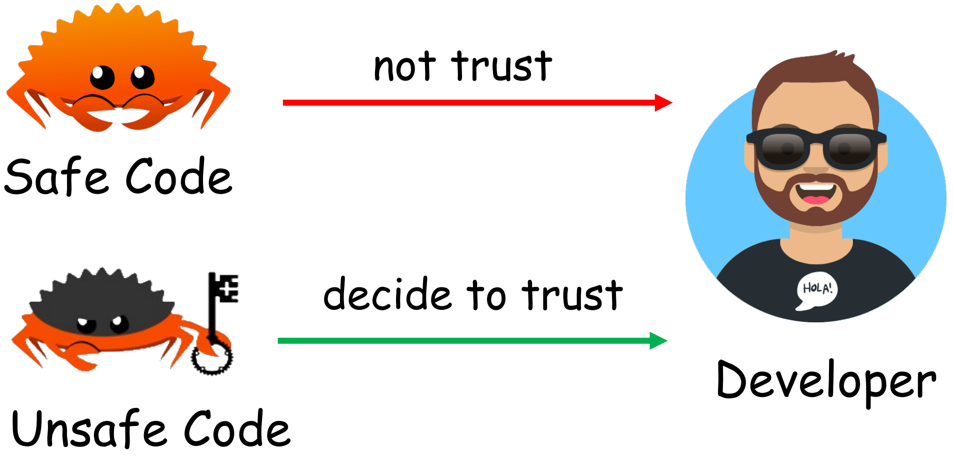
# Principle of Rust

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



# Trust Model

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



# Problem for Double-Linked List

- Both prev and next owns a node within the list
- Violate exclusive mutability


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

# Raw Pointer Version (Not Recommend)

- The resource may not be dropped automatically
- Prone to dangling pointers

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

fn rawptr(){
    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 and RefCell

- RC: single-threaded reference-counting pointer
  - RC enables shared immutable aliases
- RefCell: a mutable memory location with dynamically checked borrow rules

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

# RC

- 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

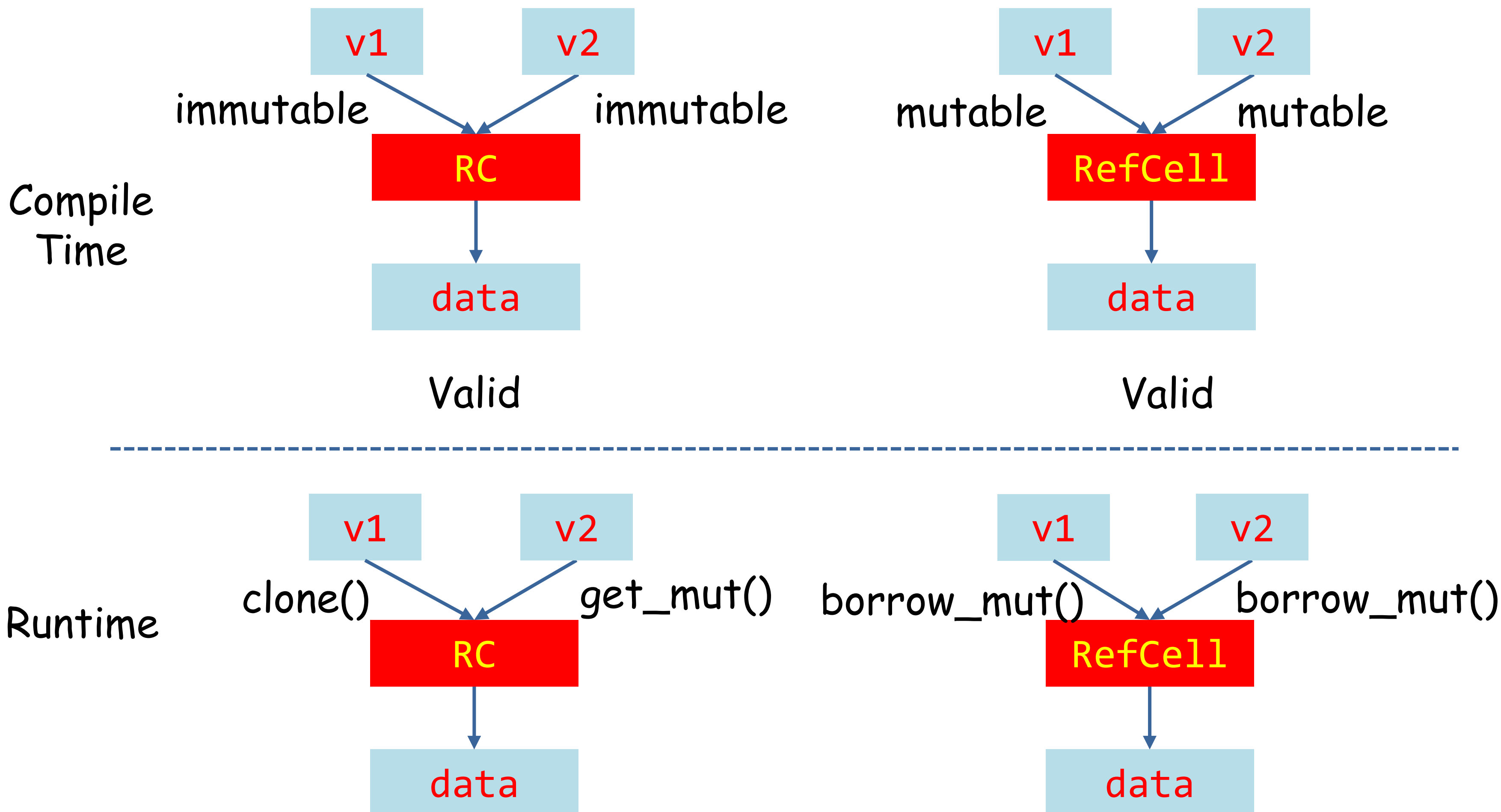
# RefCell

- 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

- Implement a binary search tree with
  - insert function
  - search function
- Implement a double linked list with Safe Rust

# Reference

- https://doc.rust-lang.org/book/
- https://doc.rust-lang.org/stable/nomicon/