Advanced Rust - Lab 5: Unsafe Rust

Lukáš Hozda

2025

Setup

1. Create a new project: `cargo new unsafe_lab`
2. Each exercise should be implemented in separate modules in your project

Exercise 1: Raw Pointers Basics (20 minutes)

Implement the following functions that demonstrate basic operations with raw pointers:

1. Create a function that converts a reference to a raw pointer
2. Create a function that safely dereferences a raw pointer
3. Create a function that safely performs pointer arithmetic

Requirements

```rust
// Create these functions:

/// Converts a reference to a raw pointer
fn to_raw_ptr<T>(value: &T) -> *const T {
    // TODO: Implement
}

/// Safely dereferences a raw pointer
/// Returns None if the pointer is null
fn safe_deref<T>(ptr: *const T) -> Option<&'static T> {
    // TODO: Implement - CAREFUL WITH THE LIFETIME!
}

/// Safely performs pointer arithmetic to get the next element
fn ptr_offset<T>(ptr: *const T, offset: isize) -> *const T {
    // TODO: Implement
}

fn main() {
    // Test reference to raw pointer
    let value = 42;
    let ptr = to_raw_ptr(&value);
    println!("Raw pointer: {:?}", ptr);

    // Create an array and a pointer to its first element
    let array = [1, 2, 3, 4, 5];
    let first_elem_ptr = to_raw_ptr(&array[0]);
```

```rust
    // Perform pointer arithmetic to access array elements
    for i in 0..array.len() {
        let ptr = ptr_offset(first_elem_ptr, i as isize);
        // CAREFULLY dereference the pointer
        unsafe {
            println!("Element at offset {}: {}", i, *ptr);
        }
    }

    // Demonstrate why safe_deref is problematic and should NOT be used in real code
    // This comment is a hint: think about lifetime issues!
}
```

Questions to Consider

1. Why is raw pointer dereferencing an unsafe operation in Rust?
2. What guarantees is the compiler unable to make about raw pointers?
3. How can we minimize the scope of unsafe code when working with raw pointers?

Exercise 2: Building a Thread-Safe Reference-Counted Pointer (60 minutes)

Objective

Implement a thread-safe reference-counted pointer type with exclusive mutable access capability.

Instructions

Create a new smart pointer type called ‘Safe<T>‘ that:

1. Implements reference counting (object is deallocated only when the last copy is deallocated)
2. Provides interior mutability via a ‘.get$_{\text{mut}}$()‘ method that allows safe read-write access (only when no other thread is accessing the data)
3. Is thread-safe without using any types from ‘std::sync‘ (like ‘Arc‘, ‘Mutex‘, or ‘RwLock‘)
4. Behaves similarly to an ‘Arc<RwLock<T»‘ but implemented from scratch

Requirements

```rust
use std::ops::{Deref, DerefMut};
use std::ptr::NonNull;
use std::marker::PhantomData;
use std::cell::UnsafeCell;
use std::fmt;
use std::sync::atomic::{AtomicUsize, Ordering};

/// A thread-safe reference-counted pointer with interior mutability
pub struct Safe<T> {
    // TODO: Implement the necessary fields
    // Hint: You'll need a pointer to inner data and atomic values for synchronization
}

// Inner structure that holds the value and synchronization state
struct SafeInner<T> {
```

```rust
    // TODO: Implement the necessary fields
    // Hint: You'll need the value, reference count, and write lock state
}

// Guard type that releases the write lock when dropped
pub struct SafeMutGuard<'a, T> {
    // TODO: Implement the necessary fields
}

impl<T> Safe<T> {
    /// Create a new Safe<T> with a reference count of 1
    pub fn new(value: T) -> Self {
        // TODO: Implement
    }

    /// Get a shared reference to the inner value
    pub fn get(&self) -> &T {
        // TODO: Implement
    }

    /// Try to get exclusive mutable access to the inner value
    /// Returns None if another thread has write access
    pub fn get_mut(&self) -> Option<SafeMutGuard<T>> {
        // TODO: Implement
        // This should attempt to acquire a write lock
    }

    // Helper function to access the inner data
    fn inner(&self) -> &SafeInner<T> {
        // TODO: Implement
    }
}

// TODO: Implement Deref and DerefMut for SafeMutGuard

// TODO: Implement Drop for SafeMutGuard (to release the lock)

// TODO: Implement Clone for Safe<T>

// TODO: Implement Drop for Safe<T>

// TODO: Implement Debug for Safe<T> where T: Debug

// TODO: Implement Send and Sync for Safe<T> where T: Send + Sync

fn main() {
    // Basic usage
    let safe = Safe::new(42);
    println!("Value: {}", *safe.get());

    // Cloning and reference counting
    let safe2 = safe.clone();
    println!("After clone: {} {}", *safe.get(), *safe2.get());
```

```rust
// Interior mutability through get_mut
{
    if let Some(mut guard) = safe.get_mut() {
        *guard += 1;
        println!("Modified to: {}", *guard);
    } else {
        println!("Couldn't get mutable access");
    }
}

// Other threads can now access again
println!("Value after modification: {}", *safe.get());

// Trying to get simultaneous mutable access
let handle = {
    let safe_clone = safe.clone();
    std::thread::spawn(move || {
        // This will only succeed if the main thread doesn't have a mutable guard
        if let Some(mut guard) = safe_clone.get_mut() {
            *guard += 100;
            println!("Background thread modified value to: {}", *guard);
            true
        } else {
            println!("Background thread couldn't get mutable access");
            false
        }
    })
};

// Try to get mutable access in main thread
{
    if let Some(mut guard) = safe.get_mut() {
        // If we get here, the background thread should fail to get access
        *guard += 10;
        // Sleep to ensure the background thread tries to get access during this time
        std::thread::sleep(std::time::Duration::from_millis(100));
        println!("Main thread modified value to: {}", *guard);
    } else {
        println!("Main thread couldn't get mutable access");
    }
}

// Wait for background thread
let bg_success = handle.join().unwrap();

// Final value depends on which thread(s) got access
println!("Final value: {}", *safe.get());
println!("Background thread got access: {}", bg_success);

// Test with multiple threads contending for access
let shared = Safe::new(Vec::<usize>::new());

let handles: Vec<_> = (0..5)
    .map(|i| {
```

```
            let shared_clone = shared.clone();
            std::thread::spawn(move || {
                for j in 0..10 {
                    // Try to get exclusive access
                    if let Some(mut guard) = shared_clone.get_mut() {
                        guard.push(i * 100 + j);
                        println!("Thread {} added value {}", i, i * 100 + j);
                        // Hold the lock briefly
                        std::thread::sleep(std::time::Duration::from_millis(5));
                    } else {
                        // Couldn't get the lock, wait and retry
                        std::thread::sleep(std::time::Duration::from_millis(2));
                        j -= 1; // Retry this iteration
                    }
                }
            })
        })
        .collect();

    // Wait for all threads to complete
    for handle in handles {
        handle.join().unwrap();
    }

    // Print the final vector to see what was added
    println!("Final vector: {:?}", *shared.get());
    println!("Vector length: {}", shared.get().len());
}
```

Questions to Consider

1. What invariants must you maintain to ensure soundness of your unsafe code?
2. How do atomic operations ensure thread safety without traditional locks?
3. Why is a guard pattern useful for releasing locks automatically?
4. How does your implementation compare to Rust's standard library ‘Arc<RwLock<T»‘?