Advanced Rust - Lab 5: Unsafe Rust

Lukáš Hozda

2025

Introduction

This lab explores Rust's unsafe features. While Rust's safety guarantees are a core strength of the language, sometimes we need to go beyond what the type system can verify. Unsafe code allows direct manipulation of memory, calling unsafe functions, and implementing abstractions that the compiler cannot verify but that you, as the programmer, guarantee are sound.

Setup

1. Make sure you have Rust installed with `rustup`
2. Create a new project: `cargo new unsafe_lab`
3. 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

```
// 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() {
```

```rust
    // 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]);

    // 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?

Solution

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

/// DANGEROUS: Dereferences a raw pointer with a 'static lifetime
/// This function is inherently UNSOUND and provided ONLY for educational purposes!
/// The 'static lifetime is a lie - the actual lifetime depends on the pointed-to value.
fn safe_deref<T>(ptr: *const T) -> Option<&'static T> {
    if ptr.is_null() {
        None
    } else {
        // SAFETY: This is NOT actually safe! We're lying about the lifetime.
        // The pointer might point to freed memory or a stack value that will be dropped.
        // This is a demonstration of why raw pointers are dangerous.
        unsafe { Some(&*ptr) }
    }
}

/// A truly safe version would require passing the original reference's lifetime:
fn actually_safe_deref<'a, T>(ptr: *const T, _lifetime_proof: &'a ()) -> Option<&'a T> {
    if ptr.is_null() {
        None
    } else {
```

```
        // SAFETY: We're binding the lifetime to the provided proof, which helps
        // but still requires the caller to ensure the pointer is valid
        unsafe { Some(&*ptr) }
    }
}

/// Safely performs pointer arithmetic to get the next element
fn ptr_offset<T>(ptr: *const T, offset: isize) -> *const T {
    // This is safe because we're not dereferencing, just calculating a new address
    unsafe { ptr.offset(offset) }
}

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]);

    // 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
    let dangling_example = {
        let x = 42;
        let ptr = to_raw_ptr(&x);

        // DON'T DO THIS! This is unsound and will cause undefined behavior
        // because x will be dropped when this block ends
        // Uncommenting these lines might work (by chance) or crash:
        // let fake_static_ref = safe_deref(ptr).unwrap();
        // println!("Dangling reference: {}", *fake_static_ref);

        // A safer approach using actually_safe_deref
        let lifetime_anchor = ();
        let bounded_ref = actually_safe_deref(ptr, &lifetime_anchor);
        println!("Safe reference: {:?}", bounded_ref);

        ptr // Return the pointer, which will now be dangling
    };

    // This would be undefined behavior if uncommented:
    // let bad_ref = safe_deref(dangling_example).unwrap();
    // println!("Using dangling pointer: {}", *bad_ref);
```

```
    println!("Program completed safely");
}
```

Explanation

This exercise demonstrates several important concepts about raw pointers:

1. **Creating raw pointers is safe**: The ‘$\text{to}_{\text{rawptr}}$‘ function shows that you can create a raw pointer from a reference without using unsafe code. This is because creating a pointer doesn't allow any operations that could violate memory safety.
2. **Lifetime issues**: The ‘$\text{safe}_{\text{deref}}$‘ function highlights a fundamental issue with raw pointers: they don't carry lifetime information. Returning a reference with a ‘'static‘ lifetime is a lie, since the actual lifetime depends on the original value. This can lead to dangling references.
3. **Better approach**: The ‘$\text{actually}_{\text{safederef}}$‘ function shows a safer pattern by requiring a ”lifetime proof” parameter that ties the output reference's lifetime to something the caller provides.
4. **Pointer arithmetic**: The ‘$\text{ptr}_{\text{offset}}$‘ function demonstrates safe pointer arithmetic. While calculating a new pointer address is safe, dereferencing it still requires unsafe code.

The key insight is that raw pointers in Rust let you opt out of the borrow checker, which means you need to manually ensure memory safety. This is why dereferencing raw pointers requires unsafe blocks.

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

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 ‘$.\text{get}_{\text{mut}}()$‘ 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

```
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»‘?

Solution

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

/// A thread-safe reference-counted pointer with interior mutability
pub struct Safe<T> {
    // Pointer to the inner data
    inner: NonNull<SafeInner<T>>,

    // PhantomData to indicate that Safe<T> logically owns a T
    _phantom: PhantomData<SafeInner<T>>,
}
```

```rust
// Inner structure that holds the value and synchronization state
struct SafeInner<T> {
    // The value wrapped in UnsafeCell for interior mutability
    value: UnsafeCell<T>,

    // Strong reference count (number of Safe<T> instances)
    ref_count: AtomicUsize,

    // Write lock state (0 = unlocked, 1 = write locked)
    write_lock: AtomicUsize,
}

// Guard type that releases the write lock when dropped
pub struct SafeMutGuard<'a, T> {
    // Reference to the Safe instance that created this guard
    safe: &'a Safe<T>,

    // Reference to the value (we know it's safe because we have the lock)
    value: &'a mut T,
}

// Safe to send between threads if T can be sent
unsafe impl<T: Send + Sync> Send for Safe<T> {}
unsafe impl<T: Send + Sync> Sync for Safe<T> {}

impl<T> Safe<T> {
    /// Create a new Safe<T> with a reference count of 1
    pub fn new(value: T) -> Self {
        // Allocate a new SafeInner on the heap
        let inner = Box::new(SafeInner {
            value: UnsafeCell::new(value),
            ref_count: AtomicUsize::new(1),
            write_lock: AtomicUsize::new(0),
        });

        // Convert the Box to a NonNull pointer
        // SAFETY: Box does not create null pointers
        let inner = unsafe { NonNull::new_unchecked(Box::into_raw(inner)) };

        Safe {
            inner,
            _phantom: PhantomData,
        }
    }

    /// Get a shared reference to the inner value
    pub fn get(&self) -> &T {
        // SAFETY: We can safely provide a shared reference because:
        // 1. The UnsafeCell allows interior mutability
        // 2. We only create &T, not &mut T
        // 3. get_mut() ensures exclusive access for mutable references using the write_lock
        unsafe { &*self.inner().value.get() }
    }
```

```rust
    /// Try to get exclusive mutable access to the inner value
    /// Returns None if another thread already has write access
    pub fn get_mut(&self) -> Option<SafeMutGuard<T>> {
        // Try to acquire the write lock (0 -> 1)
        if self
            .inner()
            .write_lock
            .compare_exchange(0, 1, Ordering::Acquire, Ordering::Relaxed)
            .is_ok()
        {
            // Successfully acquired write lock
            // SAFETY: We have the write lock, so no other thread can access the value mutably
            let value = unsafe { &mut *self.inner().value.get() };

            Some(SafeMutGuard { safe: self, value })
        } else {
            // Someone else has the write lock
            None
        }
    }

    /// Helper function to get a reference to the inner struct
    fn inner(&self) -> &SafeInner<T> {
        // SAFETY: The pointer is valid as long as self is alive
        unsafe { self.inner.as_ref() }
    }
}

impl<'a, T> Deref for SafeMutGuard<'a, T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        self.value
    }
}

impl<'a, T> DerefMut for SafeMutGuard<'a, T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        self.value
    }
}

impl<'a, T> Drop for SafeMutGuard<'a, T> {
    fn drop(&mut self) {
        // Release the write lock
        self.safe.inner().write_lock.store(0, Ordering::Release);
    }
}

impl<T> Clone for Safe<T> {
    fn clone(&self) -> Self {
        // Increment the reference count
        // We use Acquire ordering for the fetch_add to synchronize with the Release in Drop
        let old_count = self.inner().ref_count.fetch_add(1, Ordering::Acquire);
```

```
        // Check for potential overflow - this is extremely unlikely but safe to check
        if old_count > usize::MAX / 2 {
            std::process::abort(); // Abort if we're at risk of overflow
        }

        // Create a new Safe pointing to the same inner data
        Safe {
            inner: self.inner,
            _phantom: PhantomData,
        }
    }
}

impl<T> Drop for Safe<T> {
    fn drop(&mut self) {
        // Decrement the reference count
        // We use Release ordering to ensure all previous accesses are visible
        // before another thread can take ownership after the count drops to 0
        let old_count = self.inner().ref_count.fetch_sub(1, Ordering::Release);

        // If this was the last reference
        if old_count == 1 {
            // We need an Acquire fence to synchronize with the Release operation
            // This ensures we see all changes other threads made before dropping their references
            std::sync::atomic::fence(Ordering::Acquire);

            // SAFETY: We know we're the last reference, so it's safe to deallocate
            unsafe {
                // Convert the NonNull back to a Box and drop it
                let _ = Box::from_raw(self.inner.as_ptr());
            }
        }
    }
}

impl<T: fmt::Debug> fmt::Debug for Safe<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_struct("Safe")
            .field("value", self.get())
            .field("ref_count", &self.inner().ref_count.load(Ordering::Relaxed))
            .field(
                "write_lock",
                &self.inner().write_lock.load(Ordering::Relaxed),
            )
            .finish()
    }
}

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

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

// 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 {
                    loop {
                        // 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));
                            break;
                        } else {
                            // Couldn't get the lock, wait and retry
                            std::thread::sleep(std::time::Duration::from_millis(2));
                        }
                    }
                }
            })
        })
        .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());
}
```

Explanation

This implementation provides a thread-safe reference-counted pointer with interior mutability:

1. **Structure Design**:
   - ‘Safe<T>‘: The main smart pointer type with a pointer to ‘SafeInner<T>‘
   - ‘SafeInner<T>‘: Contains the actual value and synchronization primitives
   - ‘SafeMutGuard<'a, T>‘: A guard type that provides mutable access and releases the lock when dropped
2. **Reference Counting**:
   - Uses ‘AtomicUsize‘ for the reference count to ensure thread safety
   - ‘Clone‘ increments the count, ‘Drop‘ decrements it
   - When the count reaches zero, the inner data is deallocated
3. **Interior Mutability**:
   - ‘get()‘ provides shared read-only access to the inner value
   - ‘$\text{get}_{\text{mut}}()$‘ tries to acquire exclusive write access
   - Uses atomic compare-and-exchange to implement a simple mutex-like lock
4. **Thread Safety**:
   - All shared state is protected by atomic operations with appropriate memory ordering

- The ‘write$_{\text{lock}}$‘ ensures only one thread can modify the data at a time
- Proper memory barriers ensure visibility of changes across threads

5. **RAII Lock Pattern**:
   - ‘SafeMutGuard‘ automatically releases the write lock when dropped
   - This ensures locks are always released, even in the presence of panics
6. **Memory Safety**:
   - ‘UnsafeCell‘ allows internal mutability
   - ‘NonNull‘ ensures we never have null pointers
   - Carefully managed lifetime relationships prevent use-after-free

This implementation differs from ‘Arc<RwLock<T»‘ in that it only provides exclusive write access (like a mutex) rather than the read-write lock's ability to have multiple readers. However, it successfully demonstrates the core principles of building thread-safe shared mutable state from scratch.

The key insight is using atomic operations to implement both the reference counting and the mutual exclusion lock, without relying on the standard library's synchronization primitives.