Advanced Rust: Ownership, Borrowing, and Encapsulation

Introduction

Rust is often defined as "a language empowering everyone to build reliable and efficient software." While memory safety is frequently highlighted as Rust's main feature, this talk explores the broader implications of Rust's ownership system and how it enables powerful encapsulation patterns.

The core challenge Rust addresses comes from Mozilla's experience with Firefox (21 million lines of code, billions of deployments): most bugs aren't local to individual functions, but arise from how components interact with each other. This "action at a distance" problem - where changes in one component break functionality elsewhere - was an explicit design goal for Rust to solve.

Fundamental Concepts

Data and Functions

Rust fundamentally deals with two main constructs: - **Data structures**: structs (multiple fields) and enums (alternatives)
- **Functions**: the primary way programs work - everything is a function call

Unlike object-oriented languages, Rust has no classes, inheritance, or automatic virtual dispatch. When components communicate (like passing data between threads), it's done through function calls.

Ownership

Every value in Rust has exactly one unique owner. This owner can: - Mutate the value - Destroy it - Pass ownership to other parts of the system

When ownership is lost or given up, the value is automatically dropped and cleaned up.

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Drop for Point {
    fn drop(&mut self) {
        println!("Dropping point!");
    }
}

fn main() {
    let point = Point { x: 1, y: 2 };
    println!("{:?}", point);
    // Point is automatically dropped here when it goes out of scope
}
```

What Can You Own?

Ownership in Rust models different types of resources:

1. **Plain data**: Numbers, simple structures in memory
2. **Heap allocations**: Vectors that allocate and manage memory
3. **Resources with lifecycle requirements**: File handles, mutex guards

```rust
use std::fs::File;

// Owning plain data
let point = Point { x: 1, y: 2 };

// Owning a resource (file handle)
let file = File::open("example.txt")?;
// When `file` goes out of scope, it's automatically closed
```

The key insight: Rust's ownership is about **resource safety**, not just memory safety. A `File` in Rust doesn't even have a `close()` method - closing happens automatically when you drop ownership.

Borrowing

Values can be referenced through borrowing, where the owner promises not to touch the data while it's borrowed.

Immutable References

```rust
fn takes_borrow(point: &Point) {
    println!("Point: ({}, {})", point.x, point.y);
}
```

Immutable references (`&T`): - Can be shared (multiple references can exist) - Guarantee no mutation is observable - Prevent the owner from removing the data from memory

Mutable References

```rust
fn modify_point(point: &mut Point) {
    point.x += 1;
}
```

Mutable references (`&mut T`): - Are unique (only one can exist at a time) - Allow mutation of the referenced data - Cannot alias with any other references

Borrow Checking

The borrow checker uses region-based memory management, drawing regions where data lives and where it's borrowed:

```rust
let data = Point { x: 1, y: 2 };
let reference = &data;        // Borrow starts here
drop(data);                   // ❌ Error: data borrowed until...
println!("{:?}", reference);  // ...here
```

The checker ensures borrowing regions are within the data's lifetime regions.

Advanced Patterns: Building a Mutex

Let's explore how Rust's concepts work together by implementing a mutex - a runtime expression of Rust's compile-time "unique access" guarantee.

Using a Mutex

```rust
use std::sync::Mutex;

struct Counter {
    value: i32,
}

fn main() {
    let counter = Counter { value: 0 };
    let mutex = Mutex::new(counter); // Takes ownership

    manipulate_counter(&mutex);
}

fn manipulate_counter(mutex: &Mutex<Counter>) {
    let mut guard = mutex.lock().unwrap();
    guard.value += 1;
}
```

Notice the ownership transfer: the `Counter` has a very short lifetime (line 1-2), then the `Mutex` takes ownership and lives for the rest of the program.

Interior Mutability

The mutex example reveals something interesting: we get mutable access (`&mut Counter`) from an immutable reference (`&Mutex<Counter>`). This is **interior mutability** - producing mutable access to immutable memory locations.

This isn't a hack - it's a core language feature that makes sense: 1. We borrow the mutex immutably (can't destroy it) 2. We call `lock()` to register our interest 3. We get a `MutexGuard` token representing unique access 4. The guard can produce a mutable reference to the inner data

Implementing a Simple Mutex

```rust
use std::cell::UnsafeCell;

pub struct Mutex<T> {
    data: UnsafeCell<T>,
    // ... locking machinery
}

pub struct MutexGuard<'a, T> {
    mutex: &'a Mutex<T>,
}

impl<T> Mutex<T> {
    pub fn new(data: T) -> Self {
        Mutex {
            data: UnsafeCell::new(data),
        }
    }

    pub fn lock(&self) -> MutexGuard<T> {
        // Acquire lock from OS (e.g., pthread_mutex_lock)
        MutexGuard { mutex: self }
    }
}

impl<T> MutexGuard<'_, T> {
    pub fn borrow_mut(&self) -> &mut T {
        unsafe {
            let ptr = self.mutex.data.get();
            &mut *ptr
        }
    }
}

impl<T> Drop for MutexGuard<'_, T> {
    fn drop(&mut self) {
        // Unlock mutex (e.g., pthread_mutex_unlock)
    }
}
```

The Role of Unsafe

The `unsafe` block doesn't disable language features - it enables six additional operations that could break language guarantees if misused. Here we use raw pointer dereferencing.

The key insight: the number of `unsafe` lines isn't important. What matters is the **argument** for why the unsafe code is valid. In our mutex:

1. We own the inner data through the `Mutex`
2. We have an immutable reference to the mutex (it can't be destroyed)

3. `lock()` returns a guard only when we've successfully acquired the lock
4. The guard is unique due to the locking mechanism
5. Therefore, the mutable reference we return is guaranteed to be unique

Software Architecture Implications

Ownership vs Borrowing as Contracts

Ownership is a contract forcing the caller to give up interest in data, putting the callee in charge. This is a tool for **decoupling**.

Borrowing is a contract where the caller gives access under conditions, but makes promises about keeping data available. This creates **strong coupling**.

```
// Three different contracts for writing to a file:

// 1. Takes ownership - maximum decoupling
fn submit_write_op(file: File, data: String) { /* ... */ }

// 2. Borrows file, owns data
fn write_to_file(file: &File, data: String) { /* ... */ }

// 3. Borrows both - tightest coupling
fn write_to_file_borrowed(file: &File, data: &str) { /* ... */ }
```

Fighting the Borrow Checker

When developers "fight" the borrow checker, they're often fighting their software architecture. Borrowing works well for short-term access, but becomes problematic when multiple components express opinions about memory buffer management.

The solution is often to use ownership for communication between components, keeping borrowing within component boundaries.

Key Takeaways

When coding in Rust, ask yourself:

1. **What resources do I manage?** Much more than just memory - files, network connections, locks, etc.

2. **What are their lifecycles?** Rust provides basic lifecycle management through ownership
3. **How can I encode invariants?** Use the type system to make invalid states unrepresentable
4. **How do I encapsulate complexity?** Hide internal management behind safe interfaces
5. **If unsafety is involved, what's my safety argument?** Document why unsafe code upholds Rust's guarantees

The Power of Encapsulation

Rust's encapsulation allows hiding complex resource management behind simple interfaces. The distinction between "immutable parts you can't touch" (safety machinery) and "inner parts you can access" (actual data) is a powerful pattern for building safe abstractions.

Interior mutability and unsafe code aren't escape hatches - they're tools for building safe abstractions that hide complexity from users while maintaining Rust's safety guarantees.