rust-memory-safety-examples src/use_after_free_prevention.rs: Code Companion

Reference code for the Ownership and Lifetimes Mastery lecture. Sections correspond to the lecture document.

Section 1: The Vulnerability We're Preventing

```
// C code demonstrating the use-after-free vulnerability
// This is the pattern Rust's ownership system prevents

int* vulnerable_use_after_free() {
    int* ptr = malloc(sizeof(int));  // Allocate heap memory
    *ptr = 42;                        // Store a value
    free(ptr);                        // Memory freed - ptr is now dangling
    return ptr;                       // DANGER: returning invalid pointer!
}

void exploit() {
    int* p = vulnerable_use_after_free();
    printf("%d\n", *p);  // USE-AFTER-FREE: undefined behavior
    // Memory might still contain 42... or anything else
    // Attacker could control what's now in that memory location
}
```

This C code compiles without warnings but contains a critical security vulnerability. The pointer `ptr` becomes invalid after `free()`, yet nothing stops us from returning and dereferencing it. Rust makes this pattern impossible to express.

Section 2: Ownership as the First Line of Defense

```rust
pub fn ownership_prevents_uaf() {
    let data = Box::new(42);  // Box<i32>: heap-allocated integer
    let value = *data;        // Copy the i32 value (i32 implements Copy)

    drop(data);  // Explicitly free the heap memory

    // This would NOT compile:
    // println!("{}", *data);  // Error: use of moved value `data`

    println!("Copied value: {}", value);  // Safe: value is independent copy
}
```

The `drop()` function takes ownership of `data`, consuming it. After this point, the compiler marks `data` as invalid. The `*data` dereference copies the integer because `i32: Copy`—primitive types are bitwise copied rather than moved.

Section 3: Lifetimes Preventing Dangling References

```rust
pub fn lifetime_prevents_dangling_ref() {
    let _reference: Option<&String> = None;  // Reference lives in outer scope

    {
        let _data = String::from("temporary");  // String lives in inner scope

        // This would NOT compile:
        // reference = Some(&_data);
        // Error: `_data` does not live long enough
        //        borrowed value does not live long enough
    }
    // _data is dropped here - its memory freed

    // If the assignment above compiled, _reference would be dangling here
}
```

The lifetime of `_data` is limited to the inner block (the curly braces). The compiler performs lifetime analysis and sees that `_reference` would outlive what it references. This is a compile-time check with zero runtime cost.

Section 4: Borrowing Rules and Aliasing Control

```rust
pub fn borrowing_prevents_uaf() {
    let mut data = vec![1, 2, 3];  // Vec<i32> with heap-allocated storage

    let reference = &data[0];  // Immutable borrow of first element

    // This would NOT compile while `reference` exists:
    // data.clear();
    // Error: cannot borrow `data` as mutable because it is also
    //        borrowed as immutable

    println!("First element: {}", reference);  // Last use of `reference`
    // Non-Lexical Lifetimes: borrow ends here, not at scope end

    data.clear();  // Now mutation is allowed
    println!("Vector cleared");
}
```

The `clear()` method requires `&mut self`, but an immutable borrow (`reference`) is active. Rust's rule: you can have either one mutable reference OR any number of immutable references, never both. The borrow of `reference` ends after its last use, enabling the subsequent mutation.

Section 5: Reference Counting for Shared Ownership

```rust
use std::rc::Rc;

pub fn shared_ownership_safe() {
    let data = Rc::new(vec![1, 2, 3]);  // Rc<Vec<i32>>: reference-counted
    let clone1 = Rc::clone(&data);       // Increments ref count, doesn't clone
data
    let clone2 = Rc::clone(&data);       // Ref count now 3

    println!("Data from clone1: {:?}", clone1);
    println!("Data from clone2: {:?}", clone2);

    // Memory freed only when ALL Rc handles are dropped
    drop(clone1);  // Ref count: 2
    drop(clone2);  // Ref count: 1
    drop(data);    // Ref count: 0 → memory freed
}
```

`Rc::clone()` is cheap—it increments an integer counter, not a deep copy. The underlying `Vec` is deallocated only when `strong_count` reaches zero. Note: `Rc` is not thread-safe; use `Arc` for multi-threaded scenarios.

Section 6: Consuming Self for Ownership Transfer

```rust
pub struct SafeObject {
    data: Vec<i32>,
}

impl SafeObject {
    pub fn new(data: Vec<i32>) -> Self {
        Self { data }
    }

    // Borrows self immutably - object still usable after call
    pub fn get_data(&self) -> &[i32] {
        &self.data
    }

    // Takes ownership of self - object consumed, cannot be used again
    pub fn consume(self) -> Vec<i32> {
        self.data  // Moves data out of the struct
    }
}

pub fn safe_object_usage() {
    let obj = SafeObject::new(vec![1, 2, 3]);
    let data_ref = obj.get_data();  // Borrows obj
    println!("Data: {:?}", data_ref);

    let owned_data = obj.consume();  // Moves obj - takes ownership

    // This would NOT compile:
    // obj.get_data();  // Error: value used after move

    println!("Owned data: {:?}", owned_data);
}
```

The method signature tells the whole story: `&self` borrows, `self` consumes. When `consume(self)` takes ownership, the caller loses access to `obj`. This pattern is common for builder types and resource cleanup.

Section 7: Drop Trait and Deterministic Cleanup

```rust
pub fn compile_time_safety_demo() {
    struct Resource {
        id: i32,
    }

    impl Drop for Resource {
        fn drop(&mut self) {
            println!("Resource {} freed", self.id);
            // Custom cleanup logic runs here
        }
    }

    let res = Resource { id: 1 };
    println!("Resource created: {}", res.id);

    // `res` automatically dropped at scope end
    // Drop::drop() called deterministically - not garbage collected
}
// Output:
// Resource created: 1
// Resource {} freed: 1
```

The `Drop` trait provides a destructor that runs when a value goes out of scope. Unlike garbage collection, this is deterministic—you know exactly when cleanup occurs. The compiler inserts the `drop` call automatically; you cannot call `drop()` method directly (use `std::mem::drop()` to drop early).

Quick Reference

Ownership Pattern	Signature	Effect
Take ownership	`fn foo(x: T)`	Caller loses access to `x`
Borrow immutably	`fn foo(x: &T)`	Caller retains access, cannot mutate
Borrow mutably	`fn foo(x: &mut T)`	Caller retains access, exclusive mutation
Return owned	`fn foo() -> T`	Caller receives ownership
Return reference	`fn foo(&self) -> &T`	Lifetime tied to `self`

Smart Pointer Comparison

Type	Ownership	Thread-Safe	Use Case
`Box<T>`	Single owner	N/A (moved)	Heap allocation, recursive types
`Rc<T>`	Shared, counted	No	Single-threaded shared ownership
`Arc<T>`	Shared, counted	Yes	Multi-threaded shared ownership

Key Compiler Errors

```
// "use of moved value"
let a = Box::new(1);
let b = a;
println!("{}", a);  // Error: a was moved to b

// "does not live long enough"
let r;
{ let x = 5; r = &x; }  // Error: x dropped while r borrows it

// "cannot borrow as mutable because also borrowed as immutable"
let mut v = vec![1];
let first = &v[0];
v.push(2);  // Error: push needs &mut, but first holds &
```