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

Reference code for the Ownership in Action lecture. Sections correspond to the lecture document.

## Section 1: The Structure of a Demonstration Runner

```rust
//! Use-after-free prevention example

use rust_memory_safety_examples::use_after_free;

fn main() {
    println!("=== Use-After-Free Prevention in Rust ===\n");

    // Each call delegates to library module functions
    println!("1. Ownership Prevents Use-After-Free");
    use_after_free::ownership_prevents_uaf();

    println!("\n2. Borrowing Prevents Dangling References");
    use_after_free::borrowing_prevents_dangling();

    println!("\n3. C vs Rust Comparison");
    use_after_free::compare_c_vs_rust();

    // Summary output reinforces educational goals
    println!("\n=== Key Takeaways ===");
    println!("✓ Rust's ownership system makes use-after-free impossible");
    println!("✓ Compile-time prevention - errors caught before execution");
    println!("✓ Lifetimes ensure references are always valid");
    println!("✓ No runtime overhead - zero-cost abstraction");
}
```

The entire `main()` function is an orchestrator—no memory safety logic lives here. All demonstrations are imported from `use_after_free` module in the library crate.

## Section 2: Understanding Use-After-Free

```
// What use-after-free looks like in C (CONCEPTUAL - this is unsafe code)
//
// char* ptr = malloc(100);    // Allocate memory
// strcpy(ptr, "hello");       // Use memory
// free(ptr);                  // Free memory
// printf("%s", ptr);          // BUG: Use after free!
//                             // ptr still holds the address, but memory is gone

// The freed memory might:
// - Still contain "hello" (appears to work)
// - Contain garbage (crashes or wrong output)
// - Be reallocated for something else (security vulnerability)
```

This conceptual C code shows why use-after-free is dangerous: the pointer `ptr` remains valid syntactically even after `free()`, but the memory it points to is no longer owned by the program.

## Section 3: Ownership as Temporal Safety

```
// How Rust's ownership prevents use-after-free
fn ownership_prevents_uaf() {
    let data = String::from("hello");  // data owns the String

    let moved_data = data;  // Ownership transfers to moved_data
                            // data is now invalid - compiler enforces this

    // println!("{}", data);  // COMPILE ERROR: value borrowed after move
    //                        // ^^^^^^ This line would not compile

    println!("{}", moved_data);  // Only the current owner can access
}   // moved_data goes out of scope, memory is freed
    // No dangling pointers possible - the only handle is gone
```

The `move` semantic is Rust's key insight: when ownership transfers, the previous variable becomes unusable. The compiler tracks this statically—there's no runtime check.

## Section 4: Borrowing and the Lifetime Connection

```rust
// Borrowing with lifetime enforcement
fn borrowing_prevents_dangling() {
    let reference: &String;  // Declare a reference (no value yet)

    {
        let data = String::from("temporary");
        // reference = &data;  // COMPILE ERROR: `data` does not live long enough
        //                     // The borrow would outlive the owner
    }   // data dropped here - if reference pointed to it, it would dangle

    // Correct approach: reference must not outlive referent
    let data = String::from("lives long enough");
    reference = &data;  // OK: data lives as long as reference needs
    println!("{}", reference);
}   // Both data and reference go out of scope together - safe

// The compiler error would look like:
// error[E0597]: `data` does not live long enough
//    |     reference = &data;
//    |                 ^^^^^ borrowed value does not live long enough
//    | }
//    | - `data` dropped here while still borrowed
```

Lifetimes are the compiler's way of tracking "how long does this reference need to be valid?" When a reference would outlive its referent, compilation fails.

---

## Section 5: The Compile-Time Advantage

```rust
// These print statements emphasize the compile-time nature
println!("✓ Compile-time prevention - errors caught before execution");
println!("✓ No runtime overhead - zero-cost abstraction");

// Comparison of safety approaches:
//
// | Approach           | When Detected | Runtime Cost | Coverage    |
// |--------------------|---------------|--------------|-------------|
// | AddressSanitizer   | Runtime       | 2-3x slower  | Tested paths|
// | Garbage Collection| Runtime       | GC pauses    | Complete    |
// | Rust Ownership     | Compile-time  | Zero         | Complete    |
```

The key distinction: Rust's safety checks happen entirely during compilation. The resulting binary contains no ownership checks, no reference counting, no garbage collector—just the logic you wrote.

## Section 6: C vs Rust Mental Models

```rust
// Mental model comparison

// C pointer: "I believe valid data lives here"
// - The compiler trusts you
// - No verification of belief
// - Undefined behavior if wrong

// Rust reference: "The compiler verified valid data lives here"
// - The compiler proves correctness
// - Static analysis of all code paths
// - Won't compile if unverifiable

fn compare_c_vs_rust() {
    // In Rust, you cannot create an invalid reference in safe code

    let valid_ref: &i32;
    let owner = 42;
    valid_ref = &owner;  // Compiler verifies: owner lives long enough

    // There is no Rust equivalent to:
    //   int* ptr = (int*)0xDEADBEEF;  // C: trust me, this is valid
    //   *ptr = 42;                    // Undefined behavior

    println!("Reference always valid: {}", valid_ref);
}
```

Safe Rust makes it impossible to construct a reference to invalid memory. Every reference creation is verified at compile time.

---

## Quick Reference: Ownership vs Borrowing

| Concept | Symbol | Meaning | Example |
|---|---|---|---|
| Ownership | `T` | Owns the value, controls its lifetime | `let s = String::new()` |
| Shared borrow | `&T` | Read-only access, owner retains control | `fn len(s: &String)` |

| | | | |
|---|---|---|---|
| Mutable borrow | `&mut T` | Exclusive read-write access | `fn push(s: &mut String)` |
| Move | (implicit) | Ownership transfers, source invalidated | `let s2 = s1` |

## Key Compile Errors

| Error | Meaning |
|---|---|
| "value borrowed after move" | Tried to use a value after ownership transferred |
| "does not live long enough" | Reference would outlive the data it points to |
| "cannot borrow as mutable" | Tried to mutate through a shared reference |

## The Ownership Rules

1. Each value has exactly one owner
2. When the owner goes out of scope, the value is dropped
3. Ownership can be transferred (moved) or temporarily lent (borrowed)
4. References must never outlive their referent