CSC 447 - Concepts of Programming Languages

Rust and Memory Safety

Instructor: Eric J. Fredericks

Learning Objectives

How can a systems language be memory-safe *without* garbage collection?

- Introduce the Rust programming language and its design goals
- Describe ownership as a compile-time memory management strategy
- Apply move semantics and understand when copies occur
- Describe borrowing rules and their relationship to aliasing and mutation
- Identify how Rust's ownership model prevents common memory bugs at compile time

The Problem

Recall our earlier discussion of safety in C:

```c
int main() {
    int *p = (int*) malloc (sizeof(int));
    *p = 42;
    free(p);
    printf("%d\n", *p);  // use-after-free!
}
```

- C gives you total control over memory.
- C also gives you total responsibility for memory.
- Java and Scala solve this with garbage collection — but what if you can't afford a GC?

The Design Space

How do languages manage memory?

Strategy	Examples	Trade-off
Manual	C, C++	Fast, but unsafe
Garbage Collection	Java, Scala, Go	Safe, but runtime cost
Ownership	**Rust**	**Safe *and* fast — but restrictive**

- Rust's key insight: **enforce memory safety rules at compile time**.
- No garbage collector. No manual free. The compiler does the work.

Rust at a Glance

A quick taste of Rust syntax

```rust
fn main() {
    let x: i32 = 42;          // immutable by default!
    let mut y: i32 = 10;      // explicit mutability
    y += x;
    println!("y = {y}");      // macro, not a function
}
```

```
$ rustc hello.rs && ./hello
y = 52
```

- Variables are **immutable by default** — opposite of C, Java, Scala.
- Type inference works, but we'll be explicit here for clarity.

Rust: Functions and Types

```rust
fn add(a: i32, b: i32) -> i32 {
    a + b    // no semicolon = expression (return value)
}

fn greet(name: &str) {
    println!("Hello, {name}!");
}

fn main() {
    let sum = add(3, 4);
    greet("CSC 447");
    println!("sum = {sum}");
}
```

```
Hello, CSC 447!
sum = 7
```

- The last expression in a block is its value — similar to Scala!
- `&str` is a *reference* to a string — we'll come back to `&`.

Rust: Structs and Enums

Structs

```rust
struct Point {
    x: f64,
    y: f64,
}

fn distance(p: &Point) -> f64 {
    (p.x * p.x + p.y * p.y).sqrt()
}
```

Enums (algebraic data types!)

```rust
enum Shape {
    Circle(f64),
    Rectangle(f64, f64),
}

fn area(s: &Shape) -> f64 {
    match s {
        Shape::Circle(r) =>
            std::f64::consts::PI * r * r,
        Shape::Rectangle(w, h) => w * h,
    }
}
```

- Rust enums define ADTs — like Scala's `enum` / `sealed trait`.
- `match` must be exhaustive, just like Scala!

Ownership: The Big Idea

What is ownership?

Rust's ownership rules (enforced at compile time):

1. Every value has exactly **one owner** (a variable)
2. When the owner goes **out of scope**, the value is **dropped** (freed)
3. Ownership can be **transferred** (moved), but not implicitly shared

```
1 fn main() {
2     let s = String::from("hello");  // s owns the String
3     println!("{s}");
4 }   // s goes out of scope — String is dropped (freed) here
```

- No `free`. No garbage collector. The compiler inserts the cleanup.
- This is **deterministic** — you know exactly when memory is freed.

Move Semantics

What happens when you assign one variable to another?

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;          // ownership MOVES from s1 to s2

    println!("{s2}");      // fine
    println!("{s1}");      // compile error!
}
```

```
error[E0382]: borrow of moved value: `s1`
 --> move.rs:5:16
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`
3 |     let s2 = s1;
  |              -- value moved here
5 |     println!("{s1}");
  |               ^^ value borrowed here after move
```

- After the move, `s1` is **no longer valid**. The compiler rejects this!
- This prevents **double-free** — only one owner can drop the value.

Why Move?

Why not just copy, like C?

C: Shallow copy = disaster

```c
char *s1 = malloc(6);
strcpy(s1, "hello");
char *s2 = s1;   // copies pointer
free(s1);
printf("%s\n", s2); // use-after-free!
```

Rust: Move = safe

```rust
let s1 = String::from("hello");
let s2 = s1;    // ownership moves
// s1 is gone — can't use it
drop(s2);       // only s2 is freed
```

- In C, two pointers to the same memory = aliasing = bugs.
- Rust's move ensures **one owner at a time**. No aliasing by default.

Move into Functions

Passing a value to a function also moves it!

```
fn take_ownership(s: String) {
    println!("I own: {s}");
}   // s is dropped here

fn main() {
    let s = String::from("hello");
    take_ownership(s);
    println!("{s}");   // compile error!
}
```

```
error[E0382]: borrow of moved value: `s`
```

- The function took ownership — the caller can no longer use `s`.
- This seems annoying. What if I want to *use* a value without *consuming* it?

Copy Types

Not everything moves — simple types are **copied**.

```
fn main() {
    let x: i32 = 42;
    let y = x;        // copy, not move!
    println!("x = {x}, y = {y}");  // both valid!
}
```

```
x = 42, y = 42
```

- Types like `i32`, `f64`, `bool`, `char` implement the `Copy` trait.
- Copying is cheap — they live on the stack, fixed size, no heap allocation.
- `String` **does not** implement `Copy` — it owns heap-allocated data.
- Rule of thumb: if it's on the stack and small, it copies. If it owns heap data, it moves.

Borrowing: References

What if I want to *look at* a value without taking ownership?

```
fn print_length(s: &String) {   // s is a reference — borrows, doesn't own
    println!("length = {}", s.len());
}   // s goes out of scope, but it doesn't own the String — nothing is dropped

fn main() {
    let s = String::from("hello");
    print_length(&s);          // lend s to the function
    println!("{s}");           // still valid!
}
```

```
length = 5
hello
```

- `&` creates a **reference** — a borrow that does not take ownership.
- The original owner retains ownership and responsibility for cleanup.

Borrowing Rules

What are the rules for references?

Rust's borrowing rules (enforced at compile time):

1. You may have **any number** of immutable references (`&T`)
2. **OR** exactly **one** mutable reference (`&mut T`)
3. **Never both at the same time**

```
┌──────────────────────────────────────────────────────┐
│  &T, &T, &T, ...      ← many readers: OK             │
│  &mut T                ← one writer: OK              │
│  &T + &mut T           ← reader + writer: COMPILE ERROR │
└──────────────────────────────────────────────────────┘
```

- This is the **aliasing XOR mutability** principle.
- You can alias, or you can mutate, but not both.

Immutable References

Multiple immutable borrows are fine — no one is changing the data.

```
fn main() {
    let s = String::from("hello");

    let r1 = &s;
    let r2 = &s;
    let r3 = &s;

    println!("{r1}, {r2}, {r3}");  // all fine!
}
```

```
hello, hello, hello
```

- Multiple readers, no writers — safe to share.

Mutable References

You can mutate through a mutable reference — but only one at a time!

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    r1.push_str(", world");
    println!("{r1}");
}
```

```
hello, world
```

- The variable must be declared `mut`, *and* the reference must be `&mut`.

Mutable + Immutable = Error

What if I try to mix mutable and immutable references?

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s;           // immutable borrow
    let r2 = &mut s;       // mutable borrow — conflict!

    println!("{r1}");
}
```

```
error[E0502]: cannot borrow `s` as mutable because it is
              also borrowed as immutable
 --> borrow.rs:5:14
  |
4 |     let r1 = &s;
  |              -- immutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ mutable borrow occurs here
6 |     println!("{r1}");
  |               -- immutable borrow later used here
```

- The compiler prevents data races **at compile time**.

Why This Rule?

Why can't we have a reader and a writer at the same time?

C++: Iterator invalidation

```cpp
std::vector<int> v = {1, 2, 3};
for (auto &x : v) {
    if (x == 2) {
        v.push_back(4); // reallocates!
    }
    // x is now a dangling reference
}
```

Rust: Compiler says no

```rust
let mut v = vec![1, 2, 3];
for x in &v {        // immutable borrow
    if *x == 2 {
        v.push(4);  // compile error!
    }
}
```

- C++ lets you shoot yourself. Rust's borrow checker catches this at compile time.
- The `push` would need `&mut v`, but the loop already holds `&v`.

Dangling References: Prevented

Can I return a reference to a local variable?

```
fn dangle() -> &String {
    let s = String::from("hello");
    &s    // return a reference to s
}         // s is dropped here — reference would be dangling!
```

```
error[E0106]: missing lifetime specifier
 --> dangle.rs:1:16
  |
1 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  |
help: this function's return type contains a borrowed value,
      but there is no value for it to be borrowed from

```

- The compiler refuses to let you create a dangling reference.
- Compare with C: returning `&x` from a function is undefined behavior. Rust makes it a compile error.

Lifetimes

How does the compiler know how long a reference is valid?

- Every reference has a **lifetime —** the scope for which it is valid.
- Most of the time, lifetimes are **inferred** (just like types).
- Sometimes you need to annotate them explicitly:

```
1 fn longer<'a>(s1: &'a str, s2: &'a str) -> &'a str {
2     if s1.len() > s2.len() { s1 } else { s2 }
3 }
```

- `'a` is a **lifetime parameter —** it says: "the returned reference lives as long as *both* inputs remain alive." When when one of their lifetimes end, the returned reference's lifetime ends.
- The compiler uses lifetimes to prove that no reference outlives its data.
- Think of it as a *scope contract* between caller and callee.

Lifetimes: Compiler Enforcement

What happens if I violate the lifetime contract?

```
fn longer<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() { s1 } else { s2 }
}

fn main() {
    let result;
    let s1 = String::from("long string");
    {
        let s2 = String::from("hi");
        result = longer(&s1, &s2);  // s2 doesn't live long enough!
    }
    println!("{result}");
}
```

```
error[E0597]: `s2` does not live long enough
  --> lifetime.rs:10:31
   |
10 |         result = longer(&s1, &s2);
   |                               ^^ borrowed value does not live long enough
11 |     }
   |     - `s2` dropped here while still borrowed
```

- The compiler proves that `result` could refer to `s2`, which is about to be dropped.

Ownership in Action: A Linked List

Let's see ownership work in a real data structure.

```
enum List {
    Cons(i32, Box<List>),   // Box = heap-allocated, owned pointer
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));

    // When list goes out of scope:
    // - Cons(1, ...) is dropped, which drops Box containing Cons(2, ...)
    // - Cons(2, ...) is dropped, which drops Box containing Cons(3, ...)
    // - Cons(3, ...) is dropped, which drops Box containing Nil
    // All memory freed. No GC. No manual free.
}
```

- `Box<T>` is Rust's owned heap pointer — like `unique_ptr` in C++.
- Ownership chains through the data structure. Dropping the head frees everything.

Ownership + Pattern Matching

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

fn sum(list: &List) -> i32 {
    match list {
        List::Nil => 0,
        List::Cons(val, rest) => val + sum(rest),
    }
}
```

- We take `&List` — we *borrow* the list, we don't consume it.
- Pattern matching works with references — `val` is `&i32`, `rest` is `&Box<List>`.
- The caller keeps ownership. We just read.

What Does Ownership Prevent?

Memory bugs eliminated at compile time:

Bug	How Rust prevents it
Use-after-free	Moved values can't be used
Double-free	Only one owner can drop
Dangling pointers	Lifetimes tracked by compiler
Data races	Aliasing XOR mutability

Bug	How Rust prevents it
Buffer overflows	Bounds checking + safe slices
Null pointer deref	No null — `Option<T>` instead
Iterator invalidation	Borrow checker

- All of these are caught **before the program runs**.

No Null: Option Types (Again!)

Rust has no `null`. Instead, it uses `Option<T>` — just like Scala!

```rust
fn find(haystack: &[i32], needle: i32) -> Option<usize> {
    for (i, &val) in haystack.iter().enumerate() {
        if val == needle {
            return Some(i);
        }
    }
    None
}

fn main() {
    let nums = vec![10, 20, 30, 40];
    match find(&nums, 30) {
        Some(idx) => println!("Found at index {idx}"),
        None => println!("Not found"),
    }
}
```

- You *must* handle the `None` case — the compiler enforces it via exhaustive matching.

The Borrow Checker in Practice

How does this feel in practice?

Rejected (initially frustrating)

```
let mut data = vec![1, 2, 3];
let first = &data[0]; // immutable borrow
data.push(4);         // needs &mut
println!("{first}");  // error!
```

Fixed (restructure your logic)

```
let mut data = vec![1, 2, 3];
let first = data[0];  // copy the value
data.push(4);         // fine now
println!("{first}");  // fine!
```

- The borrow checker forces you to think about *who owns what, when*.
- This is annoying at first. Then you realize it catches real bugs.
- Rustaceans call the learning curve **"fighting the borrow checker."**

Unsafe Rust

What if I really need to break the rules?

```
unsafe {
    let ptr = 0x012345 as *const i32;
    println!("{}", *ptr);   // dereferencing a raw pointer — unsafe!
}
```

- `unsafe` blocks let you opt out of some compiler checks.
- Used for: raw pointers, calling C code (FFI), implementing low-level data structures.
- The key design: unsafe code is **explicitly marked and isolated**.
- Compare with our earlier discussion of safety — Rust *isolates the unsafe bits*.

Comparing Approaches

How does Rust compare to what we've seen?

	C	Java/Scala	Rust
Memory management	Manual (`malloc` / `free`)	GC (automatic)	Ownership (compile-time)
Null	`NULL` pointer	`null` reference	`Option<T>`
Aliasing control	None	None	Borrow checker
Data race prevention	None	Runtime (locks)	Compile-time
Unsafe operations	Everywhere	Restricted	Explicit `unsafe` blocks
Runtime overhead	None	GC pauses	None

- Rust occupies a unique point in the design space: systems-level performance with high-level safety guarantees.

Summary

- **Ownership**: every value has exactly one owner; when the owner goes out of scope, the value is freed.
- **Move semantics**: assignment transfers ownership; prevents double-free and use-after-free.
- **Copy types**: simple stack-allocated types are copied instead of moved.
- **Borrowing**: references let you use values without taking ownership.
- **Aliasing XOR mutability**: you can have many readers or one writer, never both.
- **Lifetimes**: the compiler tracks how long references are valid — no dangling pointers.
- **No null**: `Option<T>` forces explicit handling of absent values.
- Rust achieves memory safety **at compile time** with **zero runtime cost** — but demands more from the programmer.