Introductory Rust (2026): Ownership and Borrowing

Lecture 1

Lukáš Hozda

Winter 2026/27

MFF CUNI

Ownership Core

Why Rust Ownership Exists

- Prevent memory corruption at compile time.
- Remove class of lifetime and aliasing bugs common in C/C++.
- Force explicit resource management.

Move Semantics

```
let a = String::from("hello");
let b = a;
// println!("{}", a); // move occurred
println!("{}", b);
```

- Assignment of non-`Copy` types moves ownership.
- Exactly one owner for heap allocation.

Copy Types

```
let x: i64 = 10;
let y = x;
println!("{x} {y}");
```

- Small plain-data types implement `Copy`.
- Copy duplicates bits; move invalidates source.

Borrowing

```
fn len_of(s: &String) -> usize { s.len() }

let s = String::from("abc");
println!("{}", len_of(&s));
println!("{}", s);
```

- Shared borrow `&T` allows reading.
- Owner remains valid.

Mutable Borrowing

```rust
fn push_bang(s: &mut String) {
    s.push('!');
}

let mut s = String::from("rust");
push_bang(&mut s);
println!("{s}");
```

- `&mut T` requires unique access.
- Alias + mutation is forbidden.

Borrow Rules

- Any number of immutable borrows OR one mutable borrow.
- References must never outlive owner.
- Rules are checked statically.

Slices

```
fn first_word(s: &str) -> &str {
    s.split_whitespace().next().unwrap_or("")
}
```

- `&str` and `&[T]` are borrowed views.
- Great for APIs that avoid allocation.

Ownership In APIs

- Input by value: takes ownership.
- Input by `&T`: read-only borrow.
- Input by `&mut T`: mutable borrow.
- Return `T` for ownership transfer.

Typical Beginner Mistakes

- Overusing `clone` to silence borrow checker.
- Holding mutable borrows for too long.
- Returning references to temporary values.

Practical Rule

- Start with borrows in helper functions.
- Own data in long-lived structs.
- Clone only with explicit reason.