TD-2

Ownership, borrowing, mutability, heap and stack in Rust

Pierre Cochard and Tanguy Risset

5TC-RUST 2025

Summary

- Move semantics and Copy semantics
- References and borrowing
- Mutable Reference
- Heap and Stack: the String example
- Smart Pointers
- Rust Lifetimes

Ownership

- Set of rules that govern how a Rust program **manages memory while running**.
- Some languages have **garbage collection** (*Java*) that regularly looks for no-longer-used memory as the program runs.
- In other languages, the programmer must **explicitly allocate and free the memory** (C).

- Rust uses a **third approach**: memory is managed through *ownership* with a set of rules that are verified during compilation.
- If any of the rules are violated, the program **won't compile**.
- Ownership in Rust **has zero cost**: it has no computational impact on a program at runtime.

Ownership Rules:

- Each value in Rust has an owner (which is, typically, a variable). An *owner* is to be understood as an *L-value* in C, i.e. an object which is stored in a memory location.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

The *scope* notion is (for the moment) the same as in traditional languages such as C.

First Example of Ownership: The Cube

Q1: Write a Rust program that defines a structure `Cube` and prints its size.

```rust
struct Cube {
     c : f32,
}

fn main() {
    let c = Cube { c: 1.0 };
    // Print the size of the cube here
}
```

Q2: Can you print the cube itself (`println!("My cube: {}", c);`)? If not, how can you make it printable?

Derive `Debug` Trait to print a Cube

```
#[derive(Debug)]
struct Cube {
     c : f32,
}

fn main() {
    let c = Cube { c: 1.0 };
    // The 'Debug' trait is used with the '{:?}' format:
    println!("Cube! {c:?}");
}
```

You can derive the Debug trait (#[derive(Debug)]) and use the {:?} format, or implement the `std::fmt::Display` trait yourself. `Display` (enabling the `{}` format) cannot be derived.

Q3: Define a variable `x` assigned to a given cube, print it and then define a second variable `y` defined by `let y = x;`. Then print `x` again, what is the problem?

Move semantics

In Rust, *move semantics* refers to the *ownership* transfer of data from one variable to another. Rust enforces a strict ownership model where **each piece of data has a single owner at a time**, and ownership can be transferred (or "*moved*") when a value is assigned to another variable or passed as an argument to a function.

By default, an assignment such as `let y = x` implies a transfer of ownership of the content of `x` to `y` (hence the term *Move semantics*). This limits side effects: modifying `y` does not modify `x`.

Default semantics of assignment for the `Cube` type (a structure) is *move*.

The alternative to *move semantics* is *copy semantics*: the traditional assignment semantics.

Copy semantics and Clone

In order to duplicate the cube (as it would be done in any language),
one has two possibilities: 1) to clone it explicitly, using the `clone()` method, or 2) to implement the `Copy` trait for the type of the variable.

Q4 Modify your program by cloning `x` into `y` and/or deriving the `Copy` Trait.

```
#[derive(Debug, Clone, Copy)]
struct Cube {
    c: f32,
}

fn main() {
    let x = [Cube{c:0.5}, Cube{c:0.75}, Cube{c:1.0}];
    let y = x;
    println!("x is: {:?}", x);
    println!("y is: {:?}", y);
}
```

Copy semantics for Integer Tuples

Q5: Try the following program:

```
let x = [(10,20),(30,40),(50,60)];
let y = x;
println!("x is: {:?}", x);
println!("y is: {:?}", y);
```

- Why does it work?

Types with `copy` semantics (`Copy` Trait)

Category	Types
Integers	`i8`, `i16`, `i32`, `i64`, `i128`, `isize`, `u8` … `u128`, `usize`
Floating point	`f32`, `f64`
Boolean Character	`bool`, `char`
Shared references	`&T` (hence also `&str`, `&[T]`)
Raw pointers	`*const T`, `*mut T`
Functions	function pointers `fn(...)`, function items
Compound types	tuples `(A, B, …)` and arrays `[T; N]` if all elements are `Copy`
Option	`Option<T>` if `T` is `Copy`
Closures	if all their captures are `Copy`

Types with `move` semantics (not `Copy`)

Category	Types
Owned strings	`String`
Collections	`Vec<T>`, `HashMap`, `HashSet`, `VecDeque`, etc.
Smart pointers	`Box<T>`, `Rc<T>`, `Arc<T>` (they are `Clone`, not `Copy`)
Mutable references	`&mut T`, to guarantee borrow uniqueness (in practice the compiler often performs an implicit *reborrow*)
Compound types	tuples and arrays containing at least one non-`Copy` element
User-defined types	`struct` and `enum` by default, unless `#[derive(Clone, Copy)]`

`Move/Copy` general rule

In Rust, a type has copy semantics if it implements the `Copy` trait. All other types are *moved* on assignment or when passed as an argument.

A type can be `Copy` only if a bitwise copy is sufficient. This rules out any type that owns a resource, such as heap memory or a file descriptor. It is also why `Copy` and `Drop` are mutually exclusive: a type that implements `Drop` cannot be `Copy`.

Back to Cube

```
#[derive(Debug, Clone, Copy)]
struct Cube {
    c: f32,
}
```

The default semantics of assignment for type `Cube` is *move*.

The derivation of the `Copy` trait turns it into a *copy* semantics, hence `x` and `y` represent two different values.

References and borrowing

- Alternative to *moving* or duplicating (i.e. *cloning*) a value: you can *borrow* it.
- ***Borrowing*** in Rust is done with the *reference* operator: `&`.

Q6 In the original Cube program **which has not derived the `Copy` trait**, create a reference to `x` by using `let y = & x`. Can you print `x` after that?

Reference in Rust

References in Rust are equivalent to references in any language: a **pointer** to the same content, except that, because of the strong static verifications performed by the compiler, a reference is *always* guaranteed to point to a valid value of a particular type for the life of that reference.

References are indicated by the `'&'` operator. As in C, the opposite of referencing is dereferencing, which is accomplished with the dereference operator: `'*'`.

Autoderef

in practice, the `'*'` operator can be omitted; this is called *deref coercion* or *autoderef* (it is implemented in a trait `Deref` that is implemented for all references).

This *autoderef* is implemented in almost all cases, except when you assign a value to a dereferenced mutable reference (explained in two slides):

```
let mut x = 10;
let y = &mut x;

*y = 20; // explicit dereferencing is required here
```

Reference used in Function Calls

- Borrowing is extremely useful in **function calls**.
- Without references, the ownership of the object passed as a parameter is **transferred (or moved) to the function**.
- If instead you pass a reference to the object, **ownership does not change**.

Mutable Reference

Sometimes, you *wish* to have a function call that modifies an object. For that, you can use a *mutable reference* with the syntax: `let y = &mut x`. Mutable references in Rust do not change ownership. They only provide exclusive access to a value for mutation while ensuring that the ownership of the value remains unchanged.

Q7 By using a mutable reference to x (`let y = &mut x`), write a function called `double` that double the size of your cube `x`.

Mutable Reference and Borrowing

Ownership in Rust means having full control over a value (here a value is to be understood as an *L-value*, i.e. a value which is stored in a memory box). The owner is responsible for managing the value's lifetime and cleaning up its resources when it goes out of scope. Ownership can be transferred (moved) but is unique at any given time (except in very special cases that we will see).

Borrowing (via references, either `&T` or `&mut T`) allows you to access a value without transferring ownership. Immutable borrow (&T): Grants read-only access to a value. Mutable borrow (& mut T): Grants exclusive, write-access to a value.

Rules for Mutable References

- You can only have one mutable reference to a value at a time.
- While a mutable reference exists, no other references (mutable or immutable) to the same value are allowed.

It is important to understand that the Rust compiler evaluates very precisely the *scope* of variables. (see next slide)

Q8

In the two codes below, only one of them is correct, which one and why?

```
let mut x = Cube { c: 0.75 };
let y = &mut x;
double(y);
println!("My cube is: {:?}", x);
println!("My cube is: {:?}", y);
```

```
let mut x = Cube { c: 0.75 };
let y = &mut x;
double(y);
println!("My cube is: {:?}", y);
println!("My cube is: {:?}", x);
```

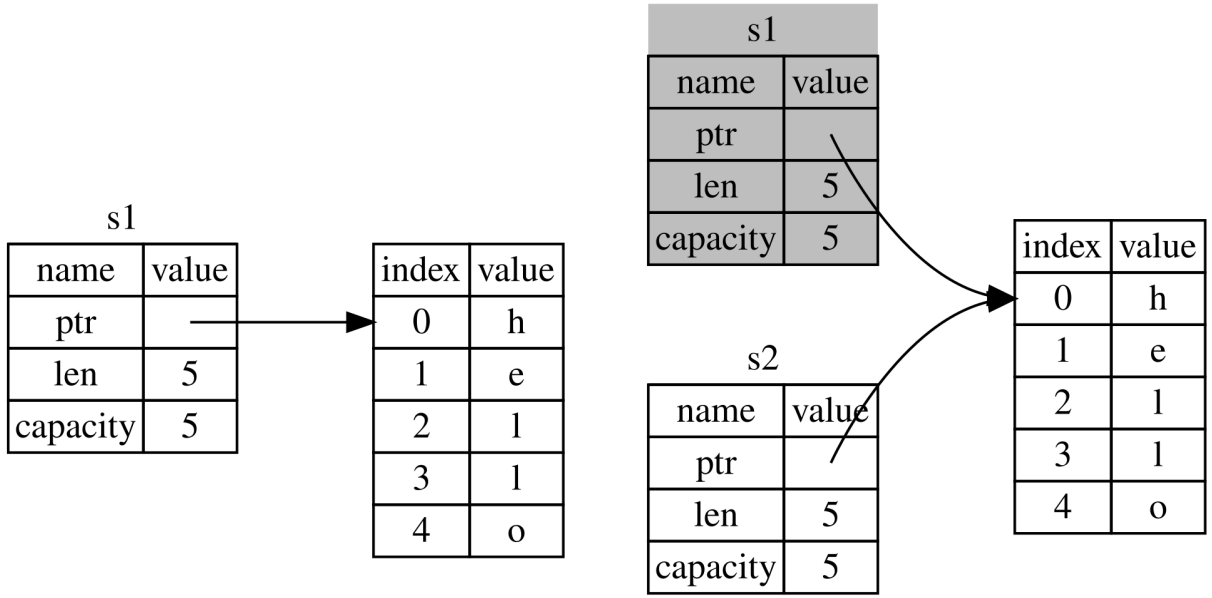
Heap and Stack: the String example

- Many programming languages don't require you to think about the **stack** and the **heap** very often.
- In a *system programming language* like Rust, it affects how the language behaves and why you have to make certain decisions.
- See Section 6 if you're not familiar with these concepts.

Take a look at the following piece of code:

```
let s1 = String::from("hello");
let s2 = s1;
```

- If `s1` and `s2` had integer types, then `s2` would have been set to a *copy* of `s1`, because all integer types have *copy semantics* by default.
- `String` *does not*, because while some of its information is *stack-allocated* (its *size* & *capacity*), it's underlying data (the actual *string* it represents) is stored on the *heap*.



(a) (b)

(a) Representation in memory of a String holding the value "hello" bound to `s1`. (b) Representation in memory of the variable `s2` that has a copy of the pointer, length, and capacity of `s1`

```rust
let s1 = String::from("hello");
let s2 = s1;
```

- When we assign `s1` to `s2`, the String data is *moved*.
- `s1` pointer is moved into `s2`, as is its `len` and `capacity`.
- The contents of the string is **not copied or re-allocated elsewhere on the heap**.
- `ptr` points to the *exact same address in memory*.
- `s1` is not accessible anymore.

Q9

- Write a function called `fn append_world(s: & mut String)` which appends " world" to string s.
- Call it with, as parameter, a **mutable reference** to `s2`.
- You can use `pub fn push_str(&mut self, string: &str)` inside your function to append a word of your choice to the `String`.

Smart pointers

- Smart pointers are inherited from other languages, such as C++.
- They are data structures that act like a pointer but also have additional *metadata* and *capabilities*.
- Rust has several smart pointer types in its *standard library*, such as `Box` (equivalent to `unique_ptr` in C++), or `Rc`, equivalent to `shared_ptr`.

The `Box<T>` type

- Boxes allow you to store data on the heap rather than the stack with a *unique* pointer system (exclusive ownership).
- The heap-allocated contents of a `Box` can be *moved*, or *cloned*, but not *copied*.
- They are particularly useful to create *recursive types*, and *trait-dynamic* types.

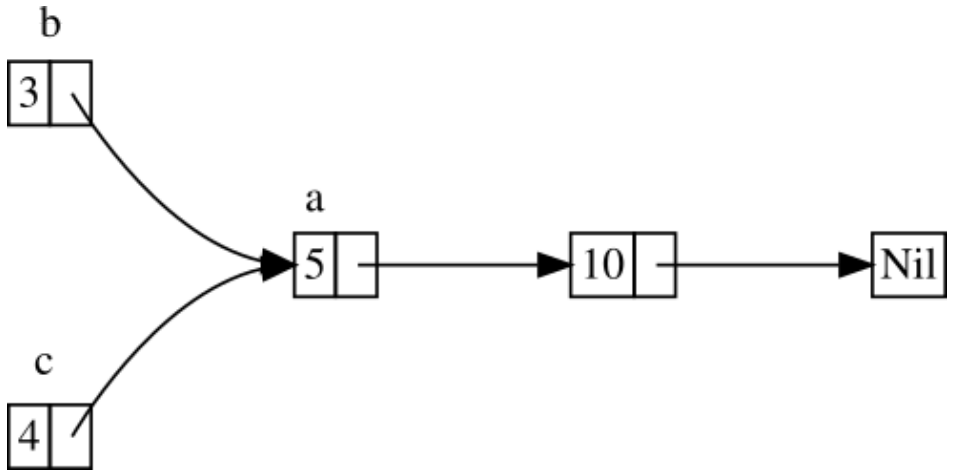
Q 10: List definition

- Create a recursive type `List` based on the following structure: a list is either (the "either" correspond to an **enum**) the constant `Nil` or the concatenation of an integer and a List: `Cons(i32, List)`. Can you do that without using Box?
- You can define an enum as follows:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
```

Multiple ownership

- In the majority of cases, ownership is clear: you know exactly which variable owns a given value.
- However, multiple ownership can be required in some cases, when, for instance, data is to be shared between multiple modules or operators.
- Example of "shared" List structure (graph):



The `Rc<T>` type

Question 11: graph structure

consider the graph on previous slide where a list (`a`) is shared by two other lists (`b` and `c`). Write a program that creates this object by using `Rc<T>` instead of `Box<T>` . For that you will have to:

- `use std::rc::Rc;`
- Provide 2 clones (for `b` and `c`) of references to `a` : `Rc::clone(&a)`

Rust Lifetimes

- In order to guarantee memory safety without using garbage collector, Rust relies on:
  - *Ownership*
  - *Borrowing*
  - ***Lifetimes***
- Lifetimes allow the compiler to check the *validity of references*.
- It's a **compile-time** concept only, **lifetimes don't exist during runtime**.
- They are often inferred by the compiler, but it is sometimes necessary to write them explicitly.

- In practice, a lifetime indicates *how long a reference must remain valid*. It's syntax is typically `'a`, `'b` etc.:

```
fn demo<'a>(x: &'a i32) { /* ... */ }
```

- Here, the `x` reference must stay valid for the entire duration of lifetime `'a`.
- Lifetime `'a` is the source object's lifetime in the context of the program.
- Here <[...]> means "generic parameter of function", they can be types (see TD4), lifetimes of constants.

```rust
fn main() {
    { // <-- Let's call this scope 's'.
        // Here: the 'foo' variable
        // is only valid within the scope of 's'
        // Therefore, foo lifetime = 's
        let foo: i32 = 0;
    }
    // Here: foo lifetime = 'main
    let foo: i32 = 0;
    // Here, FOO lifetime = 'static
    // it is valid for the whole duration of the program.
    static FOO: i32 = 0;
}
```

Q12

- Why is the following code invalid?
- Try to correct it.

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}
```

Lifetimes in Structs

- A `struct` containing references must almost always define lifetimes:

```
struct Holder<'a> {
    value: &'a str,
}
```

- In this example, a `Holder` instance cannot outlive the value it references.
- Its lifetime is tied to its `&str` value.

- In method implementations, lifetimes apply to `self`:

```
impl<'a> Holder<'a> {
    fn get(&self) -> &str {
        self.value
    }
}
```

- Thanks to the elision rules, no annotation is needed: the returned reference has the same lifetime as the `&self` borrow (which can be shorter than `'a`).
- To return a reference that lives as long as the data itself: `fn get(&self) -> &'a str`.

End of TD2