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
- Recalls on Heap and Stack

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.
- There can only be **one owner at a time**.
- When the owner goes out of scope, its value will be *dropped*.

Move and Copy semantics

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

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);
```

1. Default semantics of assignment for the `Cube` type is *move*.
2. The derivation of the `Copy` trait turns it into a *copy* semantics, hence `x` and `y` represent two different values.

Question 1)

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?

References and borrowing

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

Question 2)

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

- 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

- Equivalent to references in any language: a pointer to the same content.
- In Rust, a reference is *always guaranteed to point to a valid value of a particular type during the lifetime 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: `*`.

- In practice, the `*` operator can in many cases (not all) be omitted; this is called *deref coercion* or *autoderef*.
- But, for instance, assigning a value to a dereferenced mutable reference has to be done **explicitly**:

```
let mut x = 10;
let y = &mut x;
*y = 20; // explicit dereferencing is required here
```

Reference in Rust

- 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 *want* to have a function call that modifies an object.
- For this, a *mutable reference* can be used: `let y = &mut x`.
- They **do not change ownership**. They only provide **exclusive access to a value** for mutation, while ensuring that the ownership of the value remains unchanged.

Question 3)

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

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

Mutable References

- *Ownership* in Rust means having **full control over a value** (*stored in memory*).
- 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.
- **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 of 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 evaluate very precisely the *scope* (lifetime) of every variable.

Question 4)

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);
```

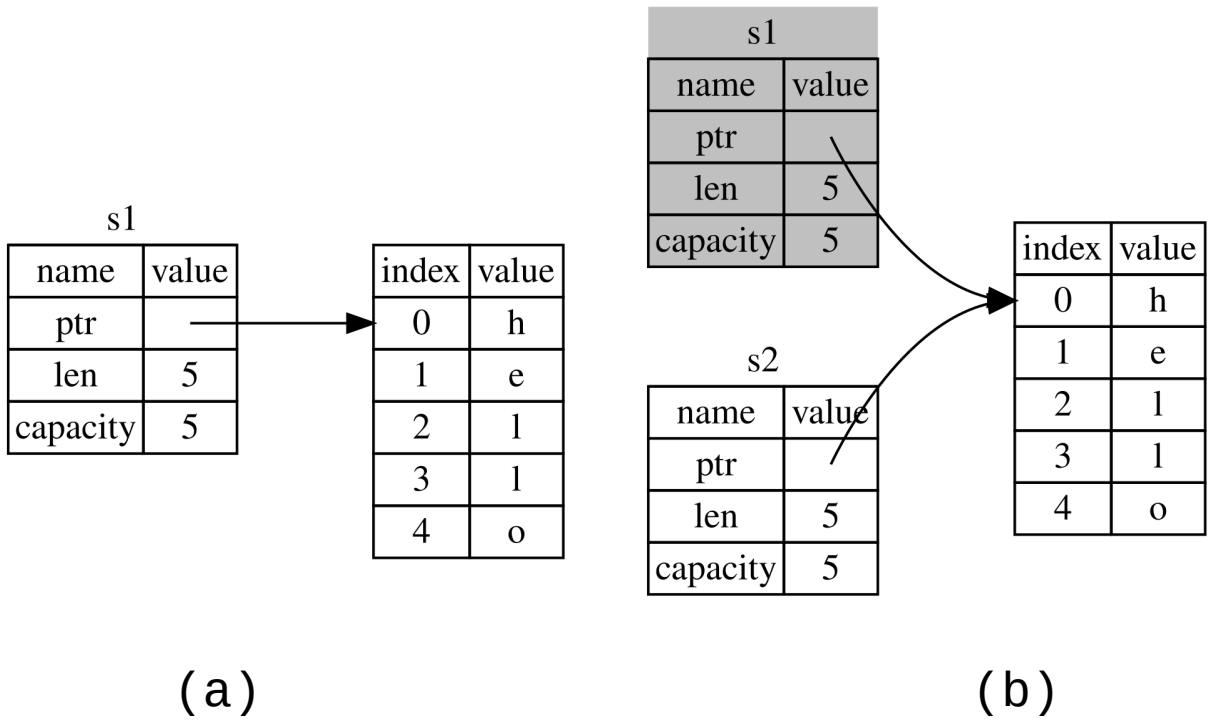
Heap and Stack: the String example

- 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) 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`

```
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.

Question 5)

- 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.

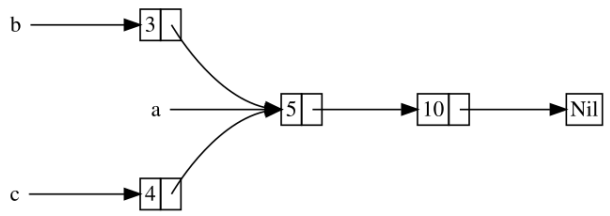
Question 6) List definition

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?

Note: If you use the symbols "Nil" and "Cons", you will have to declare the use of the created symbols after the definition of List by writing: `use crate::List::{Cons,Nil};`

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):



Question 7) 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 GC, 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.

```
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;
}
```

Question 8)

- 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
    }
}
```

End of TD2