301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

http://pages.di.unipi.it/corradini/

***AP-24**: RUST #2*

The RUST programming language

- *Brief history*
- *Memory safety*
- Avoiding Aliases + Mutable
- *Ownership* and borrowing
- Lifetimes
- Enums, Structs, Generics, Traits…
- Unsafe
- Smart Pointers
- Concurrency

RECAP

Ownership System

- Rust has an **ownership system**, which supports RAII in a strict way
- Based on the concepts of ownership and borrowing
- Ownership can be summarized by three rules:

[O1] Every value is owned by a variable, identified by a name (possiby a path);

[O2] Each value has at most one owner at a time;

[O3] When the owner goes out-of-scope, the value is reclaimed / destroyed.

Borrowing

- Ownership rules are too restrictive.
- A resource can be borrowed from its owner (via assignment or parameter passing).
- To guarantee memory safety, borrowing rules ensure that **ALIASING** and **MUTABILITY cannot coexist**
- Values can be passed
  - by immutable reference (with `x = &y`)
  - by mutable reference (with `x = &mut y`)
  - or by value (with `x = y`)

Borrowing Rules

[B1] At most one mutable reference to a resource can exist at any time

[B2] If there is a mutable reference, no immutable references can exist

[B3] If there is no mutable reference, several immutable references to the same resource can exist

- During borrowing, ownership is reduced or suspended:

[B4] Owner cannot free or mutate its resource while it is immutably borrowed

[B5] Owner cannot even read its resource while it is mutably borrowed

Borrowing: examples

[B1] At most one mutable reference to a resource can exist at any time

[B2] If there is a mutable reference, no immutable references can exist

[B3] If there is no mutable reference, several immutable references to the same resource can exist

```
let mut s = String::from("example");
let r1 = &mut s;
let r2 = &mut s;
println!("{} {}", r1, r2);     // does not compile by rule B1
```

```
let mut s = String::from("example");
let r1 = &s;
let r2 = &mut s;
println!("{} {}", r1, r2);     // does not compile by rule B2
```

```
let s = String::from("example");
let r1 = &s;
let r2 = &s;
println!("{} {}", r1, r2);                   // ok by rule B3
```

Strings in Rust

Two main types for strings:

- **String**: does not require to know the length at compilation time, thus allocated on heap
- **&str**: size must be known statically, allocated on the stack

Method **String::from()** allocates memory on the heap: it takes an argument of type **&str** and returns a **String**.

A **String** object has three components: a *reference* to the heap location containing the character sequence, a *capacity* and a *length* unsigned integer values.

String does not implement **Copy**, thus assignment has move semantics.

Assignment creates a copy of *length*, *capacity* and *reference*, but not of the char sequence in the heap.

Dangling pointers: not in Rust

Translation of C++ code does not compile by rule [B4]

```
fn main() { // Rust code
  let  s;
  {
    let s1 = String::from("scope 1");
    s = &s1;
  }
  {
    let _s2 = String::from("scope 2");
  }
  println!("s == {}", s);
}
```

```
string *s; // C++ code
{
   string s1 = "scope 1";
   s = &s1;
}
{
   string s2 = "scope 2";
}
cout << *s << endl;
```

iled with x86-64 clang 13.0.1, but it prints "scope 2" if compiled with x86-64 gcc 11.2 (see https://godbolt.org/)

```
error[E0597]: `s1` does not live long enough
  --> src\main.rs:7:13
   |
7  |          s = &s1;
   |              ^^^ borrowed value does not live long enough
8  |      }
   |      - `s1` dropped here while still borrowed
...
12 |     println!("s == {}", s);
   |                         - borrow later used here
```

Lifetimes

- A ***lifetime*** is a construct that the ***borrow checker*** uses to ensure the validity of the above rules
- Lifetimes are associated with each individual ownership and borrowing
- A lifetime begins when the ownership starts, and ends when it is moved / destroyed.
- For borrowings, it ends **where the borrowed value is accessed the last time**
- Lifetimes are mostly inferred. Sometimes must be made explicit using the same syntax of generics
- Using lifetimes, the compiler checks the validity of the rules of ownership and borrowing in the expected way
- In particular, it ensures that (the owner of) every borrowed variable/reference has a lifetime that is longer than the borrower [B4,B5]

Lifetime and borrowing: example

```
fn main() {
    let  mut s= String::from("ex-1");
    println!("s-0 == {}", s);
    let t = &mut s;
    *t = String::from("ex-2");
//    println!("s-1 == {}", s); // what happens if uncommented?
    println!("t == {}", t);
    println!("s-2 == {}", s);
    let z = &s;
    println!("s-3 == {}", s);
    let w = z;
    println!("{},{},{}",z,w,s);
}
```

```
s-0 == ex-1
t == ex-2
s-2 == ex-2
s-3 == ex-2
ex-2,ex-2,ex-2
```

Lifetimes and function calls

- Borrowed (reference) formal parameters of a function have a lifetime.
- If borrowed values are returned, each must have a lifetime. The compiled tries to infer lifetimes according to some rules:

[R1] The lifetimes of the borrowed paramers are, by default, all distinct

[R2] If there is exactly one input lifetime, it will be assigned to each output lifetime

[R3] If a method has more than one input lifetime, but one of them is **&self** or **&mut self**, then this lifetime is assigned to all output lifetimes

- Otherwise explicit lifetimes are necessary

```
fn longest(s1: &str, s2: &str) -> &str { //does not compile
        if s1.len() > s2.len() { s1 }
        else { s2 }
        }
```

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

Explicit Lifetimes in function calls

```
// `print_refs` takes two references to `i32` which have different
// lifetimes `'a` and `'b` (passed as generic parameters).
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}
```

```
// A function whith no arguments but with a lifetime parameter `'a`.
fn failed_borrow<'a>() {
    let _x = 12;
    // ERROR: `_x` does not live long enough
    // let y: &'a i32 = &_x;  // uncomment this!
    // The lifetime of `&_x` is shorter than that of `y`.
    // A short lifetime cannot be coerced into a longer one.
}
```

```
fn main() {
    let (four, nine) = (4, 9);  // Create variables to be borrowed
    print_refs(&four, &nine); //Borrows of both variables are passed
    // The lifetime of `four` and `nine` must
    // be longer than that of `print_refs`.
    failed_borrow();
}
```

Enums: algebraic data types

- Like in Haskell
- Replace unions in C/C++

```
enum RetInt {
    Fail(u32),
    Succ(u32)
}

fn foo_may_fail(arg: u32) -> RetInt {
    let fail = false;
    let errno: u32;
    let result: u32;
    ...
    if fail {
        RetInt::Fail(errno)
    } else {
        RetInt::Succ(result)
    }
}
```

```
enum std::option::Option<T> {
    None,
    Some(T)
}
```

Enums: Trees as ADT, generic

```
#[derive(Debug)] // needed to print
enum Tree<T> {
    Empty,
    Node(T, Box<Tree<T>>, Box<Tree<T>>)
}

fn main() {
    let tree = Tree::Node(
        42,
        Box::new(Tree::Node(
            0,
            Box::new(Tree::Empty),
            Box::new(Tree::Empty)
        )),
        Box::new(Tree::Empty));

    println!("{:?}", tree);
  // prints Node(42, Node(0, Empty, Empty), Empty)
}
```

Pattern matching

- Compiler enforces that matching is complete
- Useful for Enums, but also for integral types

```
fn main() {
    let x = 5; // try others…

    match x {
        1             => println!("one"),
        2             => println!("two"),
        3|4           => println!("three or four"),
        5..=10        => println!("five to ten"),
        e @ 11..=20   => println!("{}", e),
        i32::MIN..=0  => println!("less than zero"),
        21..          => println!("large"),
        _             => println!("???"),
    }
}
```

Classes: Struct + Impl

```
#[derive(Debug)]
struct Rectangle {      // class
    width: u32,         // instance variable
    height: u32,
}

impl Rectangle {         // methods
    fn area(&self) -> u32 {      // first argument is this
        self.width * self.height  // try to change width...
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!(
       "The area of the rectangle is {} square pixels.", rect1.area()
    );
}
```

No inheritance in RUST! ➔ Pushing **composition over inheritance**

Traits

- Equivalent to Type Classes in Haskell and to Concepts in C++20, similar to Interfaces in Java
- A trait can include abstract and concrete (default) methods. It cannot contain fields / variables.
- A struct can *implement* a trait providing an implementation for at least its abstract methods

```
impl <TraitName> for <StructName>{ … }
```

- The **#[derive]** clause can be used to derive automatically an implementation of a trait, if possible
- Support for **bounded universal explicit polymorphism** with **generics**, as in Java, where bounds are one or more traits.

Trait example: Stack of Slots of <T>

```
struct Slot<T> {
    data: Box<T>,
    prev: Option<Box<Slot<T>>>
}

struct SLStack<T> {
    top: Option<Box<Slot<T>>>
}
```

```
trait Stack<T> {
    fn new() -> Self;
    fn is_empty(&self) -> bool;
    fn push(&mut self, data: Box<T>);
    fn pop(&mut self) -> Option<Box<T>>;
}

impl<T> Stack<T> for SLStack<T> {
    fn new() -> SLStack<T> {
        SLStack{ top: None }
    }
    ...
    fn is_empty(&self) -> bool {
        match self.top {
            None     => true,
            Some(..) => false,
        }
    }
}
```

Generic functions: Bounded polymorphism

- Generic functions may have the generic type of parameter bound by one or more traits. Within such a function, the generic value can only be used through those traits.
- Therefore a generic function can be type-checked when defined (as in Java, unlike C++ templates).
- However, *implementation* of Rust generics similar to typical implementation of C++ templates: a separate copy of the code is generated for each instantiation.
- Thus Rust uses **monomorphization** and contrasts with the type erasure scheme of Java.
  - Pros: optimized code for each specific use case
  - Cons: increased compile time and size of the resulting binaries.

Using Traits for Bounded Polymorphism

```
trait Stack<T> {
    fn new() -> Self;
    fn is_empty(&self) -> bool;
    fn push(&mut self, data: Box<T>);
    fn pop(&mut self) -> Option<Box<T>>;
}

fn generic_push<T, S: Stack<T>>(stk: &mut S,
                                data: Box<T>) {
    stk.push(data);
}

fn main() {
    let mut stk = SLStack::<u32>::new();
    let data = Box::new(2048);
    generic_push(&mut stk, data);
}
```

Multiple Traits as bounds

```rust
trait Clone {
    fn clone(&self) -> Self;
}

impl<T> Clone for SLStack<T> {
    ...
}

fn immut_push<T, S: Stack<T>+Clone>(stk: &S, data: Box<T>) -> S {
    let mut dup = stk.clone();
    dup.push(data);
    dup
}

fn main() {
    let stk = SLStack::<u32>::new();
    let data = Box::new(2048);
    let stk = immut_push(&stk, data);
}
```

System Traits

- Traits are widely used as predicates/annotations on data types, useful for the compiler
- **Clone**: allows to create a deep copy of a value using the method **clone()**. The duplication process might involve running arbitrary code
- **Copy**: allows to duplicate a value by only copying bits stored on the stack; no arbitrary code is necessary. **Marker trait**
- **Debug**: support default conversion to text, for printing (marker)
- **Display**: programmable conversion to text, **fmt()**
- **Deref** and **Drop**: implemented by *Smart Pointers*
- **Synch** and **Send**: declare if a data type can be moved to another thread (marker)

Smart Pointers

- Originate in C++. Generalize references (*borrowing* in Rust, **&s**)
- Smart pointers: act as a pointer but with additional metadata and capabilities
- Examples: **String** (encapsulate **&str**), **Vect<T>**,…
- Typically structs, implementing **Deref** (*) and **Drop** (reclaiming when out of scope)
- "**Deref Coercion**"…

Box<T>

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

- Allow to store a data of type T on the heap
- No performance overhead
- **Deref** (*) returns the value. Optional by coercion.
- Useful when
  - Size of data not known statically (eg recursive types)

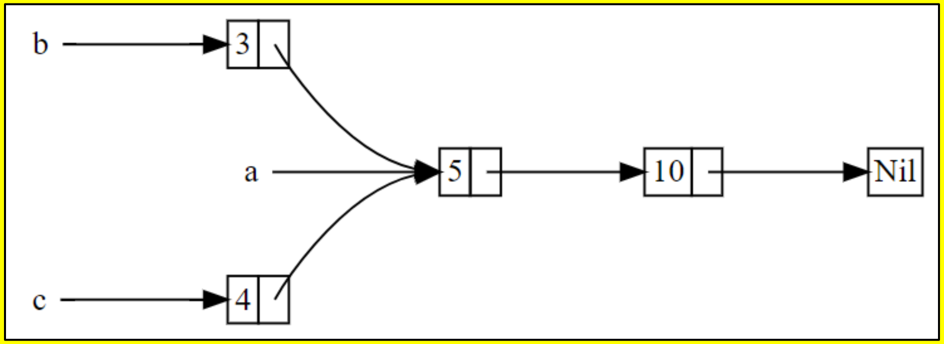
```
enum Tree<T> { // error
    Empty,
    Node(T, Tree<T>, Tree<T>)
} // type has infinite size
```

```
enum Tree<T> { //OK
    Empty,
    Node(T, Box<Tree<T>>, Box<Tree<T>>)
}
```

  - Big data, and you want to transfer ownership without copying it

Rc<T>: reference counting

- **Rc<T>**: supports **immutable** access to resource with reference counting
- Method **Rc::clone()** doesn't clone! It returns a new reference, incrementing the counter
- **Rc::strong_count** returns the value of the counter
- When the counter is 0 the resource is reclaimed



```
use crate::List::{Cons, Nil};
use std::rc::Rc;

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

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```