301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

***AP-23**: RUST*

The RUST programming language

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

Brief History

- Development started in 2006 by Graydon Hoare at Mozilla.
- Mozilla sponsored RUST since 2009, and announced it in 2010.
- In 2010 shift from the initial compiler in **OCaml** to a self-hosting compiler written in **Rust**, **`rustc`**: it successfully compiled itself in 2011.
- **`rustc`** uses **LLVM** as its back end.
- Most loved programming language in the Stack Overflow annual surveys from 2016 to 2022.
- February 8, 2021: The development of the language passes to the Rust Foundation (non-profit independent) funded by da Mozilla, Microsoft, Google, AWS e Huawei.

On RUST goals and syntax

- **Rust** is a general purpose, system programming language with a focus on safety, especially safe concurrency, supporting both functional and imperative paradigms
- Main goal: ensuring safety without penalizing efficiency
- Concrete syntax similar to C and C++ (blocks, `if-else`, `while`, `for`), `match` for pattern matching
- *Despite the superficial resemblance to C and C++, the syntax of Rust in a deeper sense is closer to that of the ML family of languages as well as the Haskell language.*
- Nearly every part of a function body is an expression (including `if-else`).
- No Runtime required (GC, Dynamic typing/binding,…)
- More control (over memory allocation/destruction…)

More than that …

C/C++

Haskell/Python

more control,
less safety

less control,
more safety

Rust

more control,
more safety

Rust overview

Performance, as with C
  - Rust compilation to object code for bare-metal performance

But, supports memory safety
  - Programs dereference only previously allocated pointers that have not been freed
  - Out-of-bound array accesses not allowed

With low overhead
  - Compiler checks to make sure rules for memory safety are followed
  - Zero-cost abstraction in managing memory (i.e. no garbage collection)

Via
  - Advanced type system
  - **Ownership**, **borrowing**, and **lifetime** concepts to prevent memory corruption issues

But at a cost
  - Cognitive cost to programmers who must think more about rules for using memory and references as they program

Memory safety

- Rust is designed to be **memory safe**, even in the presence of concurrency:
  - No null pointers
  - No dangling pointers
  - No double frees
  - No data races
  - No iterator invalidation
- These properties are guaranteed **statically**: if the program compiles it will never manifest those problems.
- Memory safety is obtained with a careful combination of several techniques: linguistic design choices, memory management policies, and powerful static (data-flow) analysis

Null pointers

- **Problem: accessing a variable which does not hold a value**
- Two approaches to guarantee that a **variable** holds a **value** when accessed:
  1. Check that it has been assigned, via data flow analysis
  2. Use **default values**
- Pros and cons…
- In Java, solution 1. for local vars of methods, solution 2. for instance and static variables.

Why???

- Sol. 2 is much simpler, sol. 1 hardly applicable to “global variables”
- Numeric variables typically have 0 as default value
- **Tony Hoare** introduced **Null** references in ALGOL W.
  - “The billion dollar mistake”…
- **NullPointerException** most thrown exception in Java

Avoiding null pointers in Rust

- A **Null** value does not exist in Rust
- Data values can only be initialized through a fixed set of forms, requiring their inputs to be already initialized.
- Compile time error if any branch of code fails to assign a value to the variable.
- Static/global variables must be initialized at declaration time
- For ***nullable types***, a generic **Option<T>** type exist, playing the role of Haskell's Maybe or Java's Optional

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

Digression: Primitive types in Rust

- Numeric types:
  - `i8 / i16 / i32 / i64 / isize`
  - `u8 / u16 / u32 / u64 / usize`
  - `f32 / f64`
- `bool`
- `char` (4-byte unicode)
- Type inference for variables declarations with **let**
- No overloading for literals: type annotations to disambiguate
- Tuples: like in Haskell
- Arrays: with fixed length. **Runtime check for out-of-bound!**

```
fn main() {
    let k = 3; // 3u8, 3.0, 3.2f32, ...
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;
    println!("The value of y is: {}",y);
    println!("The value of tup.1 is: {}",tup.1);
    let a: [i32;5] = [1,2,3,4,5];
    let b: [i32;5] = [3;5];
}
```

Using Option

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

```
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        None
    } else {
        Some(dividend / divisor)
    }
}

fn try_division(dividend: i32, divisor: i32) {
    // `Option` values can be pattern matched, just like other enums
    match checked_division(dividend, divisor) {
        None => println!("{} / {} failed!", dividend, divisor),
        Some(quotient) => {
            println!("{} / {} = {}", dividend, divisor, quotient)
        } } }
    let opt_float = Some(0f32);
    // Unwrapping a `Some` variant will extract the value wrapped.
    println!("{:?} unwraps to {:?}", opt_float, opt_float.unwrap());
```

Dangling pointers: example in C++

- **Problem: Pointers to invalid memory location**
  - Pointers to explicitly deallocated objects;
  - Pointers to locations beyond the end of an arrays;
  - Pointers to objects allocated on the stack; …
- Unpredictable effects
  - Random results
  - Segmentation fault
  - Corruption of memory manager

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

Prints "scope 1" if compiled with x86-64 clang 13.0.1, but it prints "scope 2" if compiled with x86-64 gcc 11.2 (see https://godbolt.org/)

Double free: example in C++

- **Problem: A memory location in the heap is reclaimed twice**
- This can happen in languages with explicit deallocation of memory (like C, C++)
- A double-free error could corrupt the state of the memory manager, causing a program to crash or modification of execution flow
- It could be exploited for software attacks

```
// C++ code
auto *s1 = new string("example");
auto *s2 = s1;
// ...
delete s1;
delete s2;
```

Race Condition: example in C++

- **Problem: unpredictable results in concurrent computations**
- The following multithreaded code typically prints values smaller than 20000, because of race conditions

```
// C++ code
int main() {
   int counter = 0;
   const auto task = [&] {
      for (int i = 0; i < 100000; ++i) {
      counter++;
      }
   };
   thread thread1(task);
   thread thread2(task);
   thread1.join();
   thread2.join();
   cout << counter << endl;
   return 0;
}
```

Memory management

- As usual, Rust uses a STACK of activation records, and a HEAP for dynamically allocated data structures.
- Rust favors stack allocation (default).
- The user is forced to be aware of where the data are stored: No implicit **boxing**.

```
fn main() {
    let x = 3;   // ‘let’ allocates a variable on the stack
    let y = Box::new(3); // y is a reference to 3 on the heap
    println!("x == y is {}", x == *y); // "x == y is true"
}
```

- Modern languages either use Garbage Collection, or leave the programmer the responsibility of managing the heap
- Pros and cons:
  - GC slows down or interrupts the execution; could be unfeasable for real-time systems
  - Memory management by programmer can introduce subtle errors
- Rust avoids both, providing deterministic management of resources, with very low overhead, using **RAII**

Immutability by default

- By default, Rust variables are immutable
  - Usage checked by the compiler

mut is used to declare a resource as mutable.

```
fn main() {
    let a: i32 = 0;
    a = a + 1;
    println!("a == {}", a);
}
```

```
fn main() {
    let mut a: i32 = 0;
    a = a + 1;
    println!("a == {}", a);
}
```

```
rustc 1.14.0 (e8a012324 2016-12-16)
error[E0384]: re-assignment of immutable variable `a`
 --> <anon>:3:5
  |
2 |     let a: i32 = 0;
  |         - first assignment to `a`
3 |     a = a + 1;
  |     ^^^^^^^^^ re-assignment of immutable variable

error: aborting due to previous error
```

```
rustc 1.14.0 (e8a012324 2016-12-16)
A = 1
Program ended.
```

Resource Acquisition Is Initialization

- The Resource Acquisition Is Initialization (**RAII**) programming idiom states that Resource allocation is done during **object initialization**, by the constructor, while resource deallocation (release) is done during **object destruction** (specifically **finalization**), by the destructor.
- Popular in modern C++. Small objects better allocated on stack. Large resources are on the heap (or elsewhere) and are *owned* by an object on the stack. The object is then responsible for releasing the resource in its destructor.
- The object is bound to the scope (function, block) where it is declared; when the scope closes it is reclaimed, together with any owned resource.
- Each resource has a unique owner.

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.

Move semantics of assignment

- By default, an assignment between variables has a **move semantics**: the ownership is moved from the rhs to the lhs (by [O2])

```
fn main() {
    let x = Box::new(3);
    let _y = x;  // underscore to avoid ‘unused’ warning
    println!("x = {}", x); // error!
}
```

- For primitive types and types implementing the **Copy trait**, assignment has a **copy semantics**
- [O2] is satisfied because a new value is created

```
fn main() {
    let x = 3;
    let _y = x;
    println!("x = {:?}", x); // OK
}
```

```
fn main() {
    let x = Option::Some(3);
    let _y = x;
    println!("x = {:?}", x); // OK
}
```

Move semantics of parameter passing

- The same with parameter passing and function return

```
fn foo<T>(z: T) -> T { // polymorphic identity function
    z
}
fn main(){
    let x = Box::new(3);
    let _y = foo(x);
    println!("x == {}", x);  // error
}
```

```
fn main(){
    let x = 3;
    let _y = foo(x);
    println!("x == {}", x); // OK
}
```

- Any value passed to the function will be reclaimed when it returns, as the formal parameters gets out of scope
- Only the returned value can survive (tuples to return more)

```
fn main(){
    let mut x = Box::new(3);
    x = foo(x);
    println!("x == {}", x); // OK
}
```

Ownership: Unique Owner

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {
                    a: 0,
                    b: 0
                 });
    take(res);
    println!(“res.a = {}”, res.a);
}
```

Compiling Error!

Ownership is moved from res to arg

```
fn take(arg: Box<Dummy>) {
}
```

arg is out of scope and the resource is freed automatically

Double free: not in Rust

- Remember the C++ code

```
// Codice C++
auto *s1 = new string("esempio");
auto *s2 = s1;
// ...
delete s1;
delete s2;
```

- Rust does not allow for explicit memory deallocation.
- Because of RAII, memory is freed automatically when the owner goes out of scope
- By rule [O2], each value has only one owner.
- The move semantics of assignment guarantees that s2 only owns the string, thus when s1 goes out of scope nothing is reclaimed.

```
// Rust code
let s1 = String::new("esempio");
let s2 = s1;
```

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 using **`&T`**, by mutable reference using **`&mut T`** (or by value using **`T`**)

[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
```

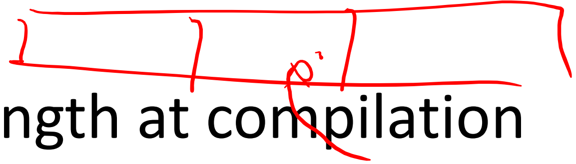
```
let mut s = String::from("example");
let r1 = &s;
let r2 = &mut s;
println!("{} {}", r1, r2);     // does not compile by rule B2
```

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

Strings in Rust

Slice Type

s[1..5] ←



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 hast 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

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

struct Slot<T> {
    data: Box<T>,
    prev: 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

```
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)

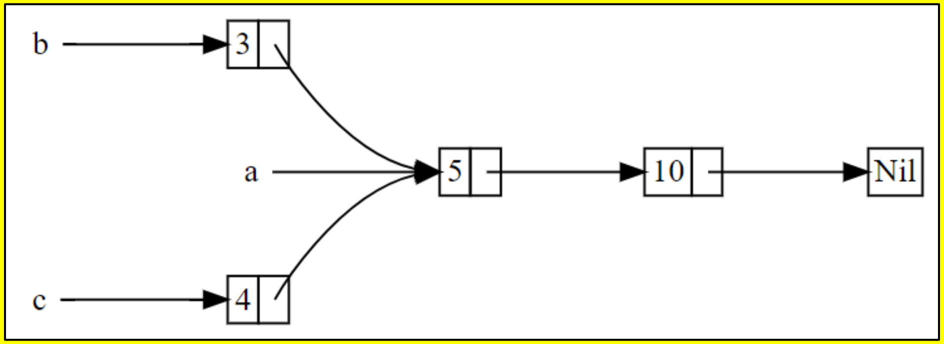
```
enum Tree<T> { // error
    Empty,
    Node(T, Tree<T>, Tree<T>)
} // type has infinite size
```

```
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



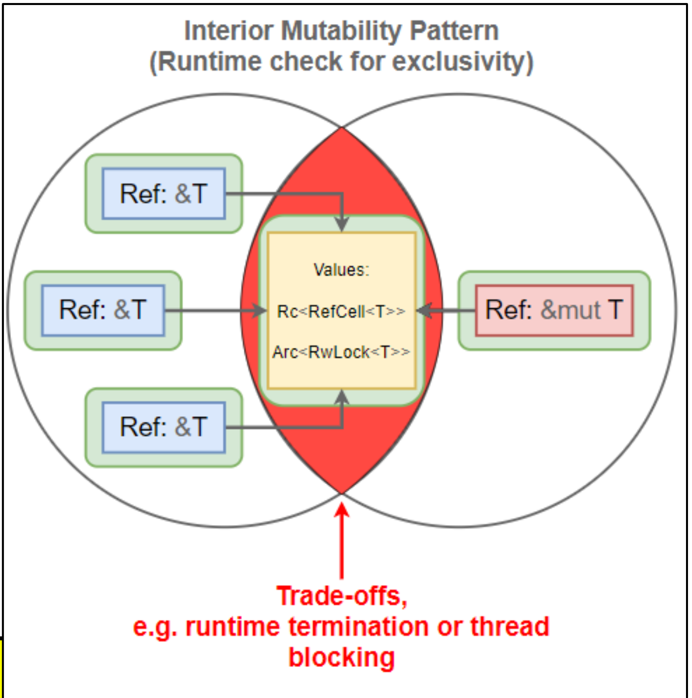
```
use crate::List::{Cons, Nil};
use std::rc::Rc;

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));
}
```

RefCell<T>: interior mutability

- **RefCell<T>**: supports shared access to a mutable resource through the **interior mutability** pattern
- It has methods **borrow()** and **borrow_mut()** which return a smart pointer (**Ref<T>** or **RefMut<T>**)
- **RefCell<T>** keeps track of how many **Ref<T>** and **RefMut<T>** are active, and panics if the ownership/borrowing rules are invalidated.
- Single-threaded, typically used with **Rc<T>** to allow multiple accesses.



Interior mutability: runtime enforced shared readable xor exclusive writeable

```
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}
...
fn main() {
    let value = Rc::new(RefCell::new(5));
    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));
    *value.borrow_mut() += 10;
    println!(...);
}
```

Comparing smart pointers

Type	Sharable?	Mutable?	Thread Safe?
&	yes *	no	no
&mut	no *	yes	no
Box	no	yes	no
Rc	yes	no	no
Arc	yes	no	yes
RefCell	yes **	yes	no
Mutex	yes, in Arc	yes	yes

* but doesn't own contents, so lifetime restrictions.

** while there is no mutable borrow

Closures, iterators, functional

```
fn main(){
    let x = 5;
    let greater_than_x = |y| y > x; // Parameters within ||
    println!("{}",greater_than_x(3));  // prints "false"
}
```

- Closures can capture non-local variables in three ways, corresponding to ownership, mutable and immutable borrowing.
- This is reflected in the trait they implement: **FnOnce**, **FnMut** and **Fn**.
- This is inferred. With **move** before || **FnOnce** is enforced.

```
let vector = vec![1, 2, 3, 4, 5]; // stream-like
vector.iter()
   .map(|x| x + 1)
   .filter(|x| x % 2 == 0)
   .for_each(|x| println!("{}", x));
```

Race Conditions: How Rust avoids them

```
// C++ code
int main() {
int counter = 0;
const auto task = [&] {
  for (int i = 0; i < 100000; ++i) {
  counter++;
                      }
};
thread thread1(task);
thread thread2(task);
thread1.join();
thread2.join();
cout << counter << endl;
return 0;
}
```

```
// Rust: does not compile
fn main() {
   let mut counter = 0;
   let task = || { // closure
      for _ in 0..100000 {
          counter += 1;
      }
};
let thread1 = thread::spawn(task);
let thread2 = thread::spawn(task);
thread1.join().unwrap();
thread2.join().unwrap();
println!("{}", counter);
}
```

```
error[E0373]: closure may outlive the current function, but it borrows
`counter`, which is owned by the current function
--> src\main.rs:57:16
let task = || {
^^ may outlive borrowed value `counter`
for _ in 0..100000 {
counter += 1;
------- `counter` is borrowed here
help: to force the closure to take ownership of `counter` (and any other
referenced variables), use the `move` keyword
let task = move || {    // would it work?
++++
```

Race Conditions: How Rust avoids them

```
// Rust code: Doesn't compile
fn main() {
   let mut counter = 0;
   let task = || {
       for _ in 0..100000 {
           counter += 1;

   }
};
let thread1 = thread::spawn(task);
let thread2 = thread::spawn(task);
thread1.join().unwrap();
thread2.join().unwrap();
println!("{}", counter);
}
```

```
// Rust code with Arc<T>: Doesn't compile
fn main() {
let mut counter = Arc::new(0);
let c1 = Arc::clone(&counter);
let c2 = Arc::clone(&counter);
let thread1 = thread::spawn(move || {
     for _ in 0..100000 {
     *c1 += 1; // Increment c1
}
});
let thread2 = thread::spawn(move || {
for _ in 0..100000 {
*c2 += 1; // Increment c2
}
});
thread1.join().unwrap();
thread2.join().unwrap();
println!("{}", counter);
}
```

```
error[E0594]: cannot assign to data in an `Arc`
--> src\main.rs:52:13
*c1 += 1;
^^^^^^^^ cannot assign
help: trait `DerefMut` is required to modify
   through a dereference, but it is not
   implemented for `Arc<i32>`
```

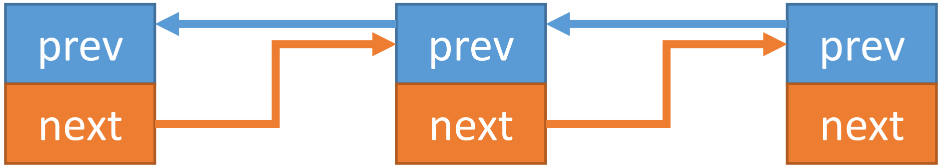
The only solution is to use a Mutex wrapped into an Arc, but with Mutex race conditions cannot happen

Traits Sync and Send (markers)

- **Send** : an error is signaled by the compiler if the ownership of a value not implementing **Send** is passed to another thread.
- For a value to be referenced by more threads,it has to implement **Sync**
- A type **T** implements **Send** if and only if **&T** implements **Sync**
- Examples: **Rc<T>** is neither **Send** nor **Sync**: operations on the internal counter are not thread safe
- **Arc<T>** is the thread-safe version of **Rc<T>**: it is **Send** and **Sync**
- **Mutex<T>** supports mutual exclusive access to a value via a lock. It is both **Send** and **Sync**, and typically wrapped in **Arc**

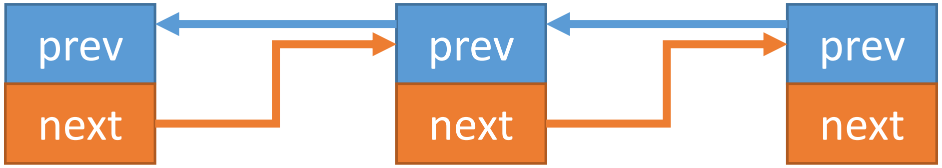
And if Mutably Sharing is necessary?

- Mutably sharing is *inevitable* in the real world.
- Example: mutable doubly linked list



```
struct Node {
    prev: option<Box<Node>>,
    next: option<Box<Node>>
}
```

Rust’s Solution: Raw Pointers



```
struct Node {
   prev: option<Box<Node>>,
   next: *mut Node
}
```

Raw pointer

- Compiler does *NOT* check the memory safety of most operations *wrt.* raw pointers.
- Most operations *wrt.* raw pointers should be encapsulated in a *unsafe* {} syntactic structure.

Rust’s Solution: Raw Pointers

```
let a = 3;

unsafe {
    let b = &a as *const i32 as *mut i32;
    *b = 4;
}

println!(“a = {}”, a);
```

I know what I’m doing

Print “a = 4”

Foreign Function Interface (FFI)

- All foreign functions are unsafe.

```
extern {
   fn write(fd: i32, data: *const u8, len: u32) -> i32;
}

fn main() {
   let msg = ”Hello, world!\n”;
   unsafe {
      write(1, &msg[0], msg.len());
   }
}
```

Unsafe superpowers

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of unions

Note: *unsafe{}* does not switch off the borrow checker