301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

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

***AP-25**: RUST #3*

The RUST programming language

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

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 is 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
```

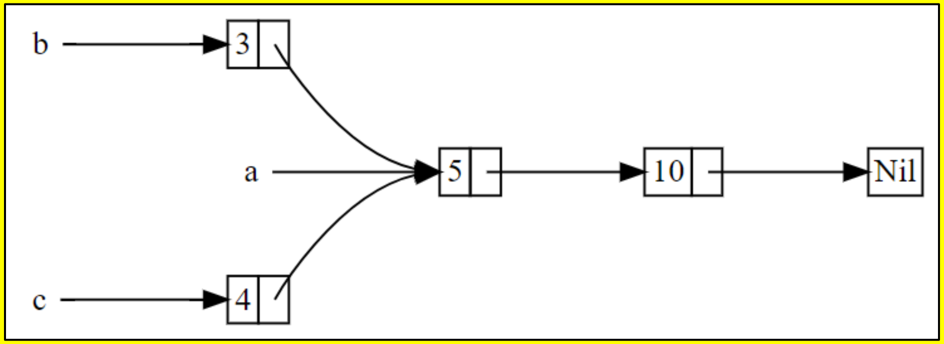
```
enum Tree<T> { //OK
    Empty,
    Node(T, Box<Tree<T>>, Box<Tree<T>>)
}
```

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

Rc<T>: reference counting

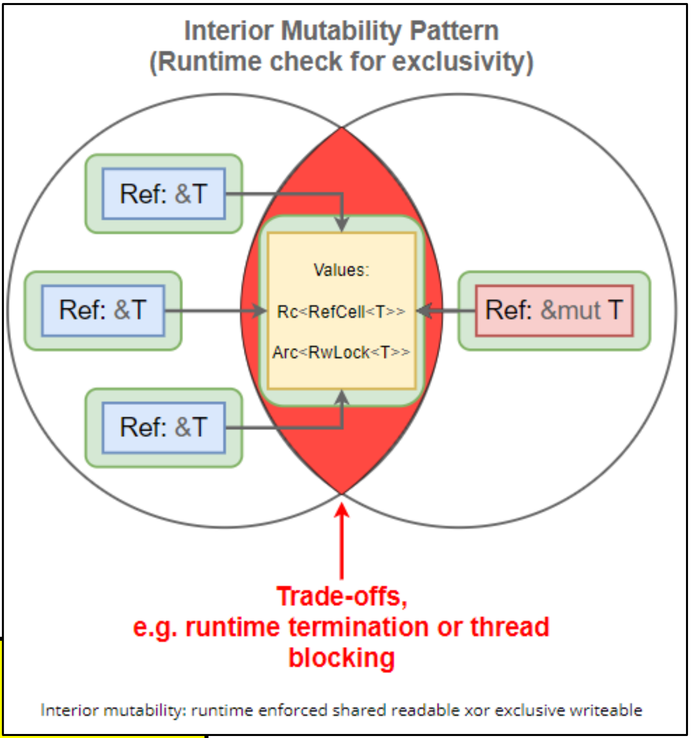
- **Rc<T>**: supports **immutable** access to resource with reference counting
- Method **Rc::clone()** doesn't clone! It returns a new reference, incrementing the counter
- **Rc::strong_count** returns the value of the counter
- When the counter is 0 the resource is reclaimed

```
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 Pattern
(Runtime check for exclusivity)

Trade-offs,
e.g. runtime termination or thread blocking

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

```cpp
// 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
// 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>`
```

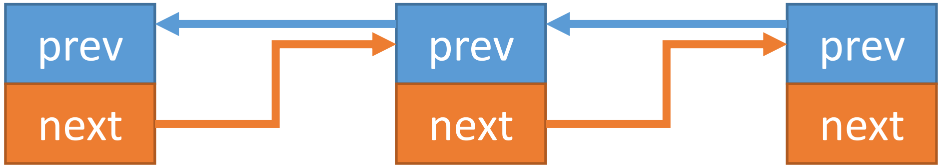
The only solution is to use a Mutex wrapped into an Arc, but with Mutex race conditions cannot happen

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**

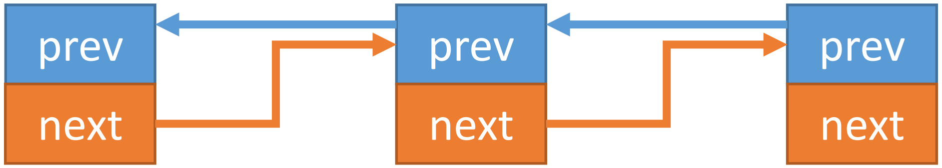
And if Mutably Sharing is necessary?

- 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
  - raw pointers can be initialised in safe Rust, but they cannot be dereferenced because it is not guaranteed that the memory they point to is actually allocated
- Call an unsafe function or method
  - using unsafe functions gives one access to the Rust allocator which is inherently unsafe as it has to deal with the OS
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of unions

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

Correctness of Rust: RustBelt

The RustBelt project provides a formalization of Rust and of its typing rules. These are used to formally prove its correctness as “absence of undefined behaviour”.

The proof is divided into three steps:

(1) Verifying that the typing rules are **semantically sound**, i.e. that the semantic interpretation of the conclusion follows from the semantic interpretation of the premises.

(2) Verifying that if a program is **semantically well-typed**, then its execution will not have problems such as undefined behaviours.

(3) Verifying that libraries using *unsafe* are **semantically safe when used through their interface**.

- Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. “RustBelt: Securing the Foundations of the Rust Programming Language”. In: *Proc. ACM Program. Lang.* 2.POPL (2017)