Lesson 7 Outline

- Recap Lesson 6
- Gestione Errori
- Smart Pointers
- Testing
- Practice & Examples

https://github.com/RustRome/corso-rust

Recap lesson 6

Rust std library's collections

- Standard implementations of data structures
- Allows communications between libraries
- Sequences: **Vec**, **VecDeque**, **LinkedList**
- Maps: **HashMap**, **BTreeMap**
- Sets: **HashSet**, **BTreeSet**
- **BinaryHeap**

Vec

- A contiguous growable array type
- pronounced vector
- Can hold every custom types
- vec![] macro for easily initialization
- Could be used as a efficient stack

VecDeque

- A double-ended queue implemented with a growable ring buffer.
- **Vec** **D**ouble-**E**nded **Que**ue
- Most notable use is as efficient Queue
- push_back push element on the tail
- pop_front pop element from the head

LinkedList

- A doubly-linked list with **owned** nodes
- allows pushing and popping elements at either end in constant time
- Not Efficient
- (In general) **Vec** or **VecDeque** are more memory efficient and make better use of CPU cache

HashMap

- A hash map implemented with quadratic probing and SIMD lookup
- default hashing algorithm: **SipHash 1-3**
- Hash custom algorithm selectable
- Multiple hash algorithms available on crates.io
- Default algorithm provides resistance against HashDoS attacks

BTreeMap, BTreeSet

- A map based on a binary search tree (BST or B-Tree)
- every element is stored in its own individual heap-allocated node
- In theory O(log n) search
- naive linear search
- Implemented as a contiguous array

Error Handling

Rust doesn’t have exceptions

🎉

Two kind of Errors

1. Unrecoverable
2. Recoverable

Unrecoverable

Unrecoverable

```rust
fn main() {
    panic!("crash and burn");
}
```

Unrecoverable

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Unrecoverable

```rust
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

Unrecoverable

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
libcore/slice/mod.rs:2448:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Unrecoverable Debug

```
$ RUST_BACKTRACE=1 cargo run
```

Recoverable

Parsing!

```rust
fn parse_number(input : &str) -> i32 {
    input.parse::<i32>()
}
```

Parsing!

```
   Compiling playground v0.0.1 (/playground)
error[E0308]: mismatched types
  --> src/main.rs:11:5
   |
10 | fn parse_number(input : &str) -> i32 {
   |                                  --- expected `i32` because of return type
11 |     input.parse::<i32>()
   |     ^^^^^^^^^^^^^^^^^^^^ expected i32, found enum `std::result::Result`
   |
   = note: expected type `i32`
              found type `std::result::Result<i32, std::num::ParseIntError>`
```

Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Parsing Fixed

```
fn parse_number(input : &str) -> Result<i32,std::num::ParseIntError> {
    input.parse::<i32>()
}
```

Parsing usage

```rust
fn main() {

    let number = parse_number("3");

    let number = number + 3;

    println!("{}",number);

}
```

Parsing compilation error

```
Compiling playground v0.0.1 (/playground)
error[E0369]: binary operation `+` cannot be applied to type `std::result::Result<i32,
std::num::ParseIntError>`
 --> src/main.rs:8:25
  |
8 |     let number = number + 3;
  |                  ------ ^ - {integer}
  |                  |
  |                  std::result::Result<i32, std::num::ParseIntError>
```

Parsing usage fix

```
fn main() {

    let number = match parse_number("3") {
        Ok(n) => n,
        Err(e)=> {
            panic!("It cannot fail")
        }
    };

    let number = number + 3;

    println!("{}",number);

}
```

Error Propagation

```
fn parse_and_add(input: &str, n : i32) -> Result<i32,std::num::ParseIntError> {

    match parse_number(input) {
        Ok(number) => Ok(number + n),
        Err(e) => Err(e)
    }
}
```

Error Propagation short version

```
fn parse_and_add(input: &str, n : i32) -> Result<i32,std::num::ParseIntError> {
    let number = parse_number(input)?;
    Ok( number + n)
}
```

Parse_and_add usage

```
fn main() {

    let number = parse_and_add("3",3);

    println!("{:?}",number);

}
```

Ok(6)

I know what i’m doing

```
fn main() {

    let number = parse_and_add("3",3).unwrap();

    println!("{:?}",number);

}
```

I know what i’m doing

```rust
fn main() {

    let number = parse_and_add("3",3).expect("Oh noo!");

    println!("{:?}",number);

}
```

Custom Error

```rust
#[derive(Debug)]
enum MyError {
    Parse(std::num::ParseIntError)
}
```

Custom Error

```
fn parse_and_add(input: &str, n : i32) -> Result<i32,MyError> {
    let number = parse_number(input)?;
    Ok( number + n)
}
```

Custom Error

```
 Compiling playground v0.0.1 (/playground)
error[E0277]: `?` couldn't convert the error to `MyError`
  --> src/main.rs:19:37
   |
19 |     let number = parse_number(input)?;
   |                                     ^ the trait
`std::convert::From<std::num::ParseIntError>` is not implemented for `MyError`
   |
   = note: the question mark operation (`?`) implicitly performs a conversion on the error
value using the `From` trait
   = note: required by `std::convert::From::from`
```

Custom Error Fix

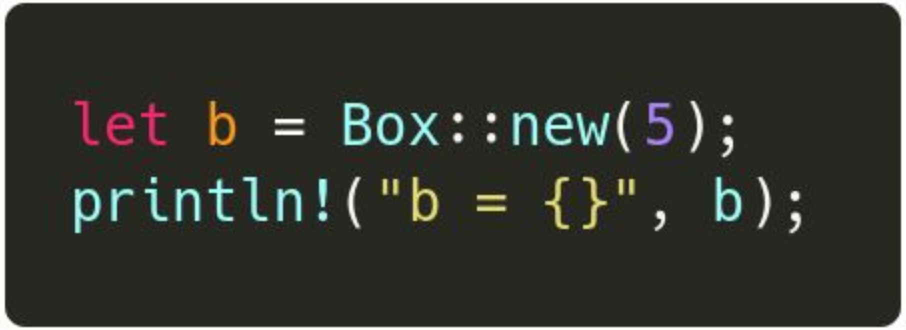
```rust
impl From<std::num::ParseIntError> for MyError {

    fn from(err : std::num::ParseIntError) -> MyError {
        MyError::Parse(err)
    }
}
```

Smart Pointers

Smart pointers

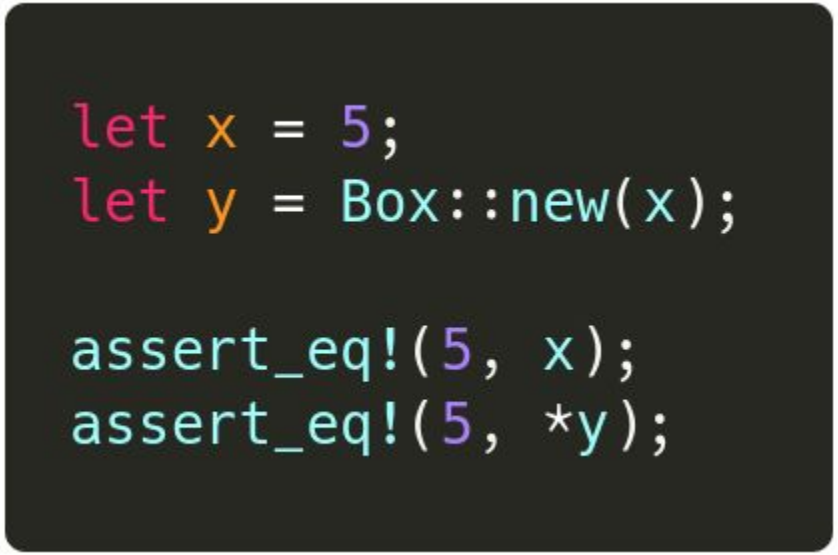
1. **Box<T>** for values allocated (heap)
2. **Rc<T>** **r**eference **c**ounter (multiple ownership)
3. **Ref<T>**, **RefMut<T>** mutability at runtime (**RefCell<T>**)

Box<T>



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

Box<T>



```
let x = 5;
let y = Box::new(x);

assert_eq!(5, x);
assert_eq!(5, *y);
```

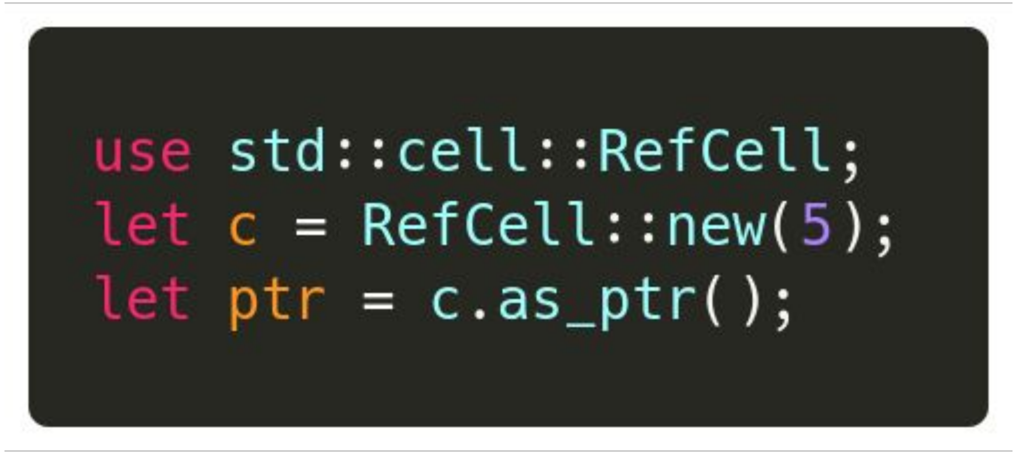
Rc<T>

```rust
use std::rc::Rc;
let my_rc = Rc::new(());

Rc::downgrade(&my_rc);
```

Rc<T>

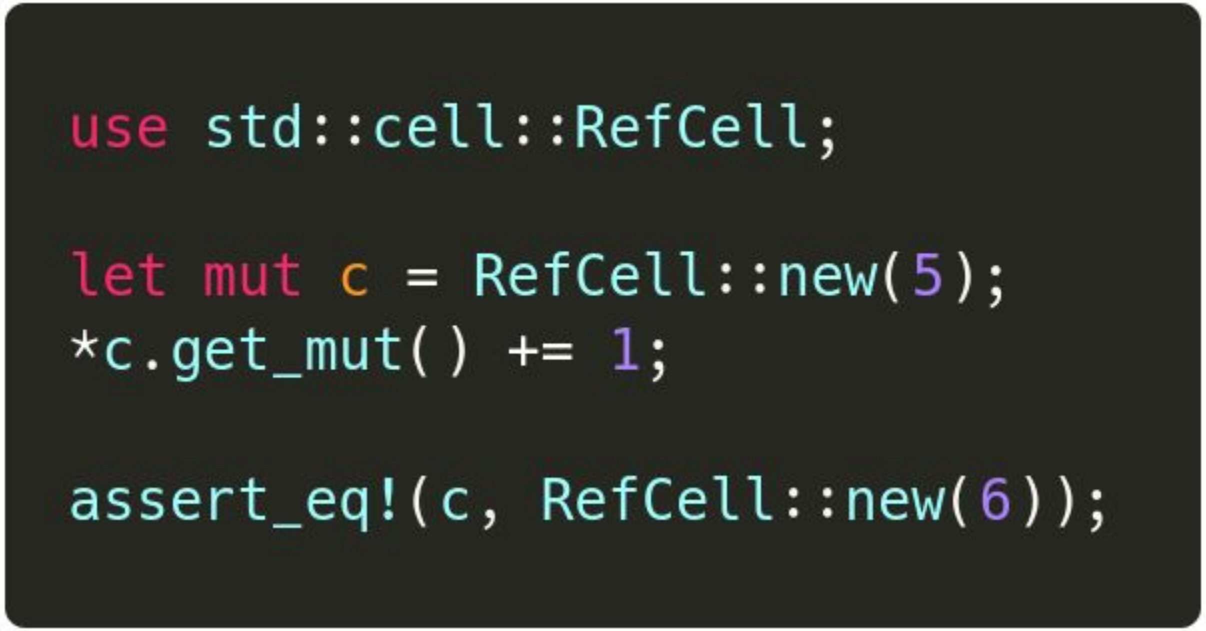
```rust
use std::rc::Rc;
let foo = Rc::new(vec![1.0, 2.0, 3.0]);
// The two syntaxes below are equivalent.
let a = foo.clone();
let b = Rc::clone(&foo);
// a and b both point to the same memory location as foo.
```

Rc<T>



```rust
use std::cell::RefCell;
let c = RefCell::new(5);
let ptr = c.as_ptr();
```

Rc<T>



```rust
use std::cell::RefCell;

let mut c = RefCell::new(5);
*c.get_mut() += 1;

assert_eq!(c, RefCell::new(6));
```

Testing

Testing

1. Integrated and accessible via cargo
2. tests mod

Test mod

```rust
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    // Note this useful idiom: importing names from outer (for mod tests) scope.
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }
}
```

Cargo test

```
$ cargo test

running 1 tests
test tests::test_add ... ok


test result: PASSED. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

```
$ cargo test

running 1 tests
test tests::test_bad_add ... FAILED

failures:

---- tests::test_bad_add stdout ----
        thread 'tests::test_bad_add' panicked at 'assertion failed: `(left == right)`
  left: `-1`,
 right: `3`', src/lib.rs:21:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    tests::test_bad_add

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

```rust
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[ignore]
    fn test_add() {
        assert_eq!(add(2, 2), 4);
    }

}
```

Next lesson

Giovedì 12 Dicembre orario 18-20

Per info e domande:

alex179ohm@gmail.com

enrico.risa@gmail.com

Oggetto: **corso rust sapienza**

Slack: http://rust-italia.herokuapp.com channel: #rust-roma