Lesson 6 Outline

- Recap Lesson 5
- Collections
- HashMap
- Vectors
- Practice & Examples

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

Recap lesson 5

Rust standard library

- Two different library provided by the language
- **Core** library provide the basic of the language
- **Std** library provide the minimal language ecosistem

Rust core library

- Designed to be used in **embedded** and **system** programming (#[no_std])
- No dependecies
- No libc or System libraries

Rust Std library

- Re-export the core library
- Portable
- Minimal and battle-tested functionalities
- Core types like Vec<T> and String
- Multithreading support
- I/O funcionalities
- Standard macros (println, vec, assert, panic ecc..)

Rust Std library

```rust
use std::fs;

let contents: String = fs::read_to_string("/path/to/file")
    .expect("failed to read file content");
```

Rust Std library

```rust
use std::process::Command;

let output = Command::new("echo")
        .arg("Hello world")
        .output()
        .expect("Failed to execute command");
```

```rust
use std::thread;

thread::spawn(move || {
    println!("I'm in a child thread");
});
```

Rust code management

- **Workspaces** (organize multiple packages)
- **Packages** for easily build, test and share crates
- **Crates**, a numbers of modules that produces library or executables
- **Modules** and **use** keyword
- Fine control for public and private code via **pub** keyword

```rust
// MyStruct is private
// can be used just onthe same mod
struct MyStruct;

// MyCratePubStruct is public
// can be used by differents mod in my crate
// cannot be exported in other crates
pub (crate) MyCratePubStruct;

// MyPubStruct is public
// can bu used by different mods and other crates
pub MyPubStruct;
```

```rust
// lib.rs
mod inner;

pub trait MyPubTrait {}

pub mod prelude {
  pub use inner::*;
  pub use MyPubTrait;
}

// other crate
use crate_io::prelude::*;
```

2015 edition

Cargo.toml
src/main.rs
    /inner/mod.rs
    /inner/something.rs
    /inner/other_mod/mod.rs
    /inner/other_mod/other.rs

2018 edition

```
Cargo.toml
src/main.rs
    /inner.rs
    /inner/something.rs
    /inner/other_mod.rs
    /inner/other_mod/other.rs
```

crates

```
$ cargo new my_lib --lib
$ tree
.
├── Cargo.toml
└── src
    └── lib.rs
```

crates

```
$ cargo new my_app
$ tree
.
├── Cargo.toml
└── src
    └── main.rs
```

dependencies

```
[package]
name = "my_app"
version = "0.1.0"
authors = ["user.name <user.email>"]
edition = "2018"

[dependencies]
log = "0.4.8"
```

Login with cargo and publish

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
$ cargo publish
```

Appendix: **workspace**

```
# Cargo.toml

[workspace]
members = [
    "my_crate",
    "my_other_crate",
]
```

collections

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**

Sequences

BinaryHeap

- A priority queue implemented with a binary heap
- push O(1)~
- pop O(log n)
- peek O(1)

BinaryHeap

```rust
pub struct BinaryHeap<T> {
    data: Vec<T>,
}
// T must implement Ord Trait
```

BinaryHeap

```rust
use std::collections::BinaryHeap;
let mut heap = BinaryHeap::new();
heap.push(1);
heap.push(5);
heap.push(2);
```

BinaryHeap

```
// Borrow of most important value
assert_eq!(heap.peek(), Some(&5));
// check the lenght
assert_eq!(heap.len(), 3);
// get the most important value
assert_eq!(heap.pop(), Some(5));
assert_eq!(heap.pop(), Some(2));
```

BinaryHeap

```
// Borrow of most important value
assert_eq!(heap.peek(), Some(&5));
// check the lenght
assert_eq!(heap.len(), 3);
// get the most important value
assert_eq!(heap.pop(), Some(5));
assert_eq!(heap.pop(), Some(2));
```

BinaryHeap

```
#[derive(Eq)]
struct Page {
  url: String,
  downloads: i32,
}
```

BinaryHeap

```
use std::cmp::{Ord, PartialOrd, Ordering, PartialEq};

impl PartialEq for Page {
    fn eq(&self, other: &Self) -> bool {
        self.downloads == other.downloads
    }
}
```

BinaryHeap

```
impl PartialOrd for Page {
    fn partial_cmp(&self, other: &Self) → Option<Ordering> {
        Some(self.cmp(other))
    }
}
impl Ord for Page {
    fn cmp(&self, other: &Self) → Ordering {
        self.downloads.cmp(&other.downloads)
    }
}
```

BinaryHeap

```
let mut bin_heap = BinaryHeap::new();
bin_heap.push(Page{url: "url".to_owned(), downloads: 1000});
bin_heap.push(Page{url: "url2".to_owned(), downloads: 3000});
```

BinaryHeap

```
if let Some(most_downloaded) = bin_heap.pop() {
    println!("most downloaded page this mount");
    println!("url: {}", most_downloaded.url);
    println!("downloads: {}", most_downloaded.downloads);
}
```

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

LinkedList

```
use std::collections::LinkedList;

let list: LinkedList<u32> = LinkedList::new();
```

LinkedList

```
list.push_back(0);
list.push_back(1);
list.push_front(2);
for i in list {
  println!("{}", i); // 0 1 2
}
```

LinkedList

```
let mut list1 = LinkedList::new();
list1.push_back('a');
let mut list2 = LinkedList::new();
list2.push_back('b');
list2.push_back('c');
list1.append(&mut list2);
```

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

BTreeMap

```rust
pub struct BTreeMap<K, V> {
    root: node::Root<K, V>,
    length: usize,
}
// K must implement Ord
```

BTreeMap

```
use std::collections::BTreeMap;
let mut movie_reviews = BTreeMap::new();
```

BTreeMap

```
movie_reviews.insert("Office Space",
  "Deals with real issues in the workplace.");
movie_reviews.insert("Pulp Fiction",
  "Masterpiece.");
```

BTreeMap

```
// check for key
if !movie_reviews.contains_key("Les Misérables") {
    println!("We've got {} reviews, but Les Misérables ain't one.",
             movie_reviews.len());
}
```

BTreeMap

```
println!("Movie review: {}", movie_reviews["Office Space"]);

for (movie, review) in &movie_reviews {
    println!("{}: \"{}\"", movie, review);
}
```

BTreeMap

```
// remove
movie_reviews.remove("The Blues Brothers");
```

BTreeMap

```
player_stats.entry("health").or_insert(100);
fn func() → u8 {
    42
}
player_stats.entry("defence").or_insert_with(func);
```

BTreeMap

```rust
use std::collections::BTreeMap;
let mut count: BTreeMap<&str, usize> = BTreeMap::new();
for x in vec!["a","b","a","c","a","b"] {
    *count.entry(x).or_insert(0) += 1;
}
assert_eq!(count["a"], 3);
```

BTreeSet

```rust
pub struct BTreeSet<T> {
    map: BTreeMap<T, ()>,
}
// T must implements Ord
```

BTreeSet

```
let mut books = BTreeSet::new();
books.insert("A Dance With Dragons");
books.insert("To Kill a Mockingbird");
books.remove("The Odyssey");
```

BTreeSet

```
if !books.contains("The Winds of Winter") {
    println!("We have {} books, but The Winds of Winter ain't one.",
             books.len());
}
for book in &books {
    println!("{}", book);
}
```

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

Vec

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}
```

Vec

```
let mut vec = Vec::new();
vec.push(1);
vec.push(2);
assert_eq!(vec.len(), 2);
```

Vec

```
let mut vec = Vec::new();
vec.push(1);
vec.push(2);
assert_eq!(vec.pop(), Some(2));
assert_eq!(vec.len(), 1);
```

Vec

```
let mut vec = Vec::new();
vec.push(1);
assert_eq!(vec[0], 1);
vec[0] = 7;
assert_eq!(vec[0], 7);
```

Vec

```rust
let mut vec = Vec::new();
vec.push(7);
vec.extend([1, 2, 3].iter().cloned());
assert_eq!(vec, [7, 1, 2, 3]);
```

Vec

```
let v = vec![0, 2, 4, 6];
assert_eq!(vec[2], 4);
assert_eq!(vec[6], 7); // panic
```

Vec

```rust
#[derive(Debug)]
struct User {
    age: i32,
}
let users: Vec<User> = vec![User{age: 32}, User{age: 12}, User{age: 18}];
println!("{:?}", users);
println!("{}", users[0].age);
```

Vec

```
struct WebPage {
  nodes: Vec<Nodes>,
}
struct Web<T> {
  elements: Vec<T>,
}
```

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

VecDeque

```
pub struct VecDeque<T> {
    tail: usize,
    head: usize,
    buf: RawVec<T>,
}
```

VecDeque

```rust
use std::collections::VecDeque;

let vector: VecDeque<u32> = VecDeque::new();
```

VecDeque

```rust
use std::collections::VecDeque;
let mut buf = VecDeque::new();
buf.push_back(3);
buf.push_back(4);
buf.push_back(5);
assert_eq!(buf, [3, 4, 5]);
```

VecDeque

```rust
use std::collections::VecDeque;
let mut buf = VecDeque::new();
buf.push_back(5);
buf.push_back(4);
assert_eq!(buf, [4, 5]);
assert_eq!(buf.pop_front(), Some(4));
```

VecDeque

```rust
use std::collections::VecDeque;
let mut buf = VecDeque::new();
buf.push_back(5);
buf.push_back(4);
assert_eq!(buf, [4, 5]);
assert_eq!(buf.pop_back(), Some(5));
```

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

HashMap

```rust
pub struct HashMap<K, V, S> {
    base: base::HashMap<K, V, S>,
}
// K must implement Hash and Eq
// S needed to implements Hash algoritms
```

HashMap

```
use std::collections::HashMap;
let mut book_reviews = HashMap::new();
```

HashMap

```
book_reviews.insert(
    "Adventures of Huckleberry Finn".to_string(),
    "My favorite book.".to_string(),
);
book_reviews.insert(
    "Grimms' Fairy Tales".to_string(),
    "Masterpiece.".to_string(),
);
book_reviews.remove("Adventures of Huckleberry Finn");
```

HashMap

```
if !book_reviews.contains_key("Les Misérables") {
    println!("We've got {} reviews, but Les Misérables ain't one.",
             book_reviews.len());
}
println!("Review for Jane: {}", book_reviews["Pride and Prejudice"]);
for (book, review) in &book_reviews {
    println!("{}: \"{}\"", book, review);
}
```

HashMap

```rust
use std::collections::HashMap;
let mut player_stats = HashMap::new();
fn func() -> u8 {
    42
}
player_stats.entry("health").or_insert(100);
player_stats.entry("defence").or_insert_with(func);
```

HashMap

```
#[derive(Hash, Eq, PartialEq, Debug)]
struct Person {
    id: u32,
    name: String,
    phone: u64,
}
let mut rank = HashMap::new();
rank.insert(Person{id: 1, name: String::new(), phone: 0}, 3);
rank.insert(Person{id: 1, name: String::new(), phone: 0}, 10);
println!("{:?}", users);
```

HashMap

```rust
use std::collections::HashMap;
use std::hash::{Hash, Hasher};
#[derive(Eq, Debug)]
struct Person {
    id: u32,
    name: String,
    phone: u64,
}
```

HashMap

```
impl PartialEq for Person {
    fn eq(&self, other: &Person) -> bool {
        self.id == other.id
    }
}
impl Hash for Person {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.id.hash(state);
    }
}
```

HashMap

```
impl Person {
    fn new(id: u32) → Self {
        Person { id, ..Default::default() }
    }
}
let mut rank = HashMap::new();
rank.insert(Person::new(1), 3);
rank.insert(Person::new(34), 10);
println!("{:?}", rank);
```

Next lesson

Martedì 10 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