Object System, Slices and Modules

-

If it’s a modular sliced duck and it looks like slices of a modular duck then it’s

Home exercises

- Too hard?
- Too easy?

```
> # Updating YOUR fork
> git remote add upstream https://github.com/cohenarthur/rust-gistre-workshop
> git fetch upstream # Fetch OUR repository's master branch
> git checkout master
> git rebase upstream/master # Rebase YOUR work on OUR updates
> # Tada!
```

Slices


```c
int main() {
    char *line = NULL;
    size_t len = 0;
    getline(&line, &len, stdin);
    char *last_word = get_last_word(line);
    printf("The last word is: %s", last_word);

    free(line);
    return 0;
}
```

```c
char *get_last_word(char *str) {
    size_t word_start = 0;
    for (size_t i = 0; str[i]; i++) {
        if (str[i] == ' ') {
            word_start = i + 1;
        }
    }
    return str + word_start;
}
```

USE AFTER FREE

```c
int main() {
    char *line = NULL;
    size_t len = 0;
    getline(&line, &len, stdin);
    char *last_word = get_last_word(line);
    printf("The last word is: %s", last_word);
    free(line);
    // Some more work done on last_word
    return 0;
}
```

UNEXPECTED BEHAVIOUR

```c
int main() {
    char *line = NULL;
    size_t len = 0;
    getline(&line, &len, stdin);
    char *last_word = get_last_word(line);
    printf("The last word is: %s", last_word);
    getline(&line, &len, stdin);
    // Some more work done on last_word
    free(line);
    return 0;
}
```

```c
C SOLUTION: MORE ALLOC
int main() {
    char *line = NULL;
    size_t len = 0;
    getline(&line, &len, stdin);

    char *last_word = strdup(get_last_word(line));
    printf("The last word is: %s", last_word);
    free(line);
    return 0;
}
```


```rust
fn main() {
    let mut line = String::new();
    io::stdin().read_line(&mut line).expect("Could not read from stdin");

    let last_word = get_last_word(&line);
    println!("The last word is: {}", last_word);
}

fn get_last_word(line: &str) -> &str {
    line.split_whitespace().last().unwrap_or("")
}
```

USE AFTER FREE ?

```
fn main() {
    let mut line = String::new();
    io::stdin().read_line(&mut line).expect("Could not read from stdin");

    let last_word = get_last_word(&line);
    drop(line);
    println!("The last word is: {}", last_word);
}
```

USE AFTER FREE ?

```
error[E0505]: cannot move out of `line` because it is borrowed
  --> src/main.rs:10:10
   |
9  |     let last_word = get_last_word(&line);
   |                                   ----- borrow of `line` occurs here
10 |     drop(line);
   |          ^^^^ move out of `line` occurs here
11 |     println!("The last word is: {}", last_word);
   |                                      --------- borrow later used here
```

UNEXPECTED BEHAVIOUR ?

```
fn main() {
    let mut line = String::new();
    io::stdin().read_line(&mut line).expect("Could not read from stdin");

    let last_word = get_last_word(&line);
    line.clear();
    println!("The last word is: {}", last_word);
}
```

UNEXPECTED BEHAVIOUR ?

```
> cargo build
error[E0502]: cannot borrow `line` as mutable because it is also borrowed as
immutable
  --> src/main.rs:10:5
   |
9  |     let last_word = get_last_word(&line);
   |                                   ----- immutable borrow occurs here
10 |     line.clear();
   |     ^^^^^^^^^^^^ mutable borrow occurs here
11 |     println!("The last word is: {}", last_word);
   |                                      --------- immutable borrow later used here
```

```
Str is a slice
fn main() {
    let mut line = String::new();
    io::stdin().read_line(&mut line).expect("Could not read line");

    let last_word = get_last_word(&line);
}

fn get_last_word(line: &str) -> &str {
    line.split_whitespace().last().unwrap_or("")
}
```


```rust
fn main() {
    let values = vec!(1, 2, 3, 4, 5);
    let slice = &values[1..4]; // slice is of type [i32]
    println!("The slice contains {:?}", slice);
}

// The slice contains [2, 3, 4]
```


```rust
fn main() {
    let i_am_a_str = "Hello !";
}
```

Modules

- mod
  - *Declare* the existence of a module
  - Similar to C/C++ #include <...> and adding the <...>.c to the compiler line
    - The compiler now registers that there is another file/directory to compile!

- use
  - *Import* a module or a function into the current namespace
  - Similar to C++ using namespace <...>
    - Not necessary, but makes it easier to call functions from other modules

Modules

- mod
  - Can be used for local modules (mod <name> { … })
  - Can be used for files in the same directory (mod <file_name_without_the_rs>)
  - Can be used for directories (mod <directory_name>) but you need a mod.rs file inside
  - The root module is called crate
  - You can access a module higher in the project structure by using super
- use
  - Used for external crates (use clap::Arg)
  - Can be used to import multiple modules at once (use clap::{Arg, App})

```cpp
int main(void) {
    std::vector<std::string> v { "hello", "from", "c++", "\n" };

    std::for_each(v.begin(), v.end(),
    [](const std::string& n) { std::cout << " " << n; });

    return 0;
}
```


```cpp
using namespace std; // “Uses” everything from std!

int main(void) {
    vector<string> v { "hello", "from", "c++", "\n" };

    for_each(v.begin(), v.end(),
    [](const string& n) { cout << " " << n; });

    return 0;
}
```

```
mod standard;



fn main() {
    let v: standard::vector<standard::string> =
        vec!["hello", "from", "rust", "\n"];

    standard::for_each(v, |n| print!(" {}", n));
}
```


```rust
mod standard;
use standard::{vector, string, for_each};
// `use standard` would do nothing, it doesn't "use" anything

fn main() {
    let v: vector<string> =
        vec!["hello", "from", "rust", "\n"];

    for_each(v, |n| print!(" {}", n));
}
```


```rust
pub type vector<T> = Vec<T>; // Ewwww snake_case types!
pub type string<'s> = &'s str; // EWWWWWW LIFETIMES!!!!

// What the fuck is `impl`??? What's a FnMut????
pub fn for_each(v: Vec<&str>, mut lambda: impl FnMut(&str)) {
    v.iter().for_each(|n| lambda(n))
}
```


```
src
├── boat.rs
└── main.rs
```


```rust
pub fn sail() {
    println!("I'm sailing, wouhouuuuu");
}
```

```
mod /* ... */;
// We want to let the compiler know there is something else to
// compile. In that case, a simple file

fn main() {
    boat::sail();
}
```


```
mod /* ... */;

fn main() {
    boat::sail();
}
```

- No ‘mod’ necessary
- `mod boat;`
- `mod boat::sail;`
- `mod boat; use boat::sail;`


```rust
mod boat;

fn main() {
    boat::sail();
}
```

❌ No ‘mod’ necessary

✅ `mod boat;`

❌ `mod boat::sail;`

❌ `mod boat; use boat::sail;`


```
mod /* ... */;

fn main() {
    sail();
}
```

- No ‘mod’ necessary
- `mod boat;`
- `mod boat::sail;`
- `mod boat; use boat::sail;`


```rust
mod boat;
use boat::sail;

fn main() {
    sail();
}
```

❌ No ‘mod’ necessary

❌ `mod boat;`

❌ `mod boat::sail;`

✅ `mod boat; use boat::sail;`


```
src
├── boat.rs
└── main.rs
```

```rust
pub fn sail() {
    println!("I'm sailing, wouhouuuuu");
}

pub mod titanic {
    pub fn sink() {
        println!("Oh no gloub gloub gloub");
    }
}
```


```rust
mod boat;

fn main() {
    boat::sail();

    /* Call `sink()` */
}
```

- `boat::sink();`
- `boat::titanic::sink();`
- `titanic::sink();`
- `sink();`


```rust
mod boat;

fn main() {
    boat::sail();

    boat::titanic::sink();
}
```

❌ `boat::sink();`

✅ `boat::titanic::sink();`

❌ `titanic::sink();`

❌ `sink();`


```rust
mod boat;
use boat::sail;

fn main() {
    sail();

    /* Call `sink()` */
}
```

- `boat::sink();`
- `sail::sink();`
- `boat::titanic::sink();`
- `sink();`
- `titanic::sink();`


```rust
mod boat;
use boat::sail;

fn main() {
    sail();

    boat::titanic::sink();
}
```

❌ `boat::sink();`
❌ `sail::sink();`
✅ `boat::titanic::sink();`
❌ `sink();`
❌ `titanic::sink();`


```
/* ... */

fn main() {
    sail();

    titanic::sink();
}
```

```
-  mod boat; use boat;
-  use boat::sail;
   use boat::titanic;
-  mod boat;
   use boat::sail;
   use boat::titanic;
-  mod boat; use boat::sail;
   use boat::titanic::sink;
```


```rust
mod boat;
use boat::sail;
use boat::titanic;

fn main() {
    sail();

    titanic::sink();
}
```

❌
```rust
mod boat; use boat;
```
❌
```rust
use boat::sail;
use boat::titanic;
```
✔
```rust
mod boat;
use boat::sail;
use boat::titanic;
```
❌
```rust
mod boat; use boat::sail;
use boat::titanic::sink;
```

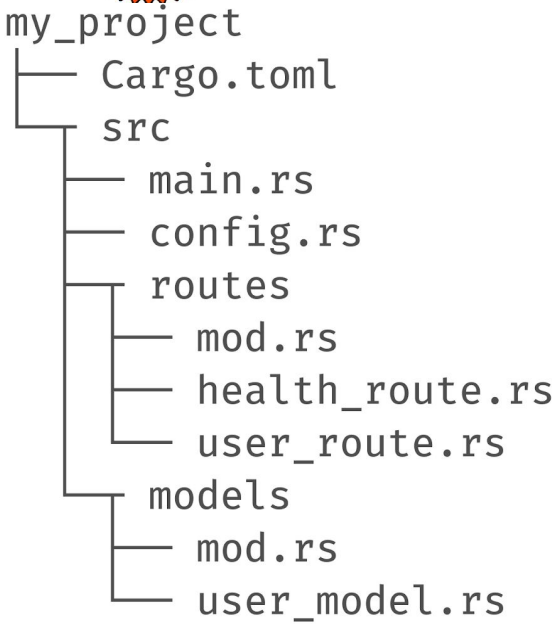

```
src
├── animals
│   └── gistre.rs
├── boat.rs
└── main.rs

# All good?
```


```
src
├── animals
│   └── gistre.rs
├── boat.rs
└── main.rs

# All good?
```

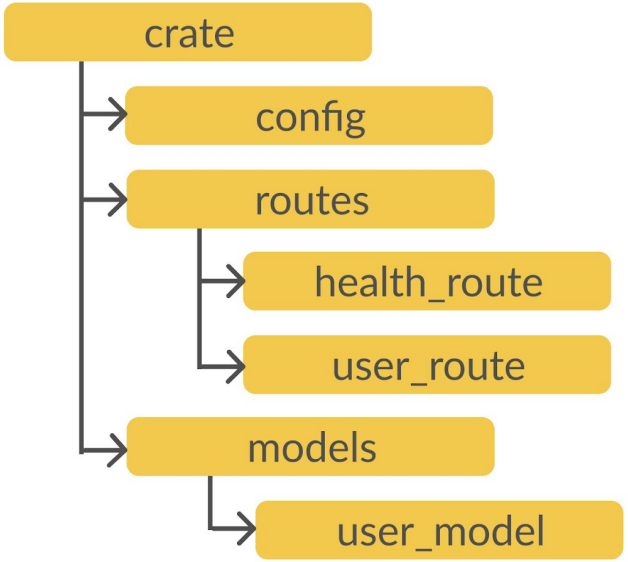
- We don’t need anything
- We need a mod.rs
- We need an animals.rs
- We need a use.rs


```
src
├── animals
│   ├── gistre.rs
│   └── mod.rs
├── boat.rs
└── main.rs

# All good?
```

❌ We don't need anything
✅ We need a mod.rs
❌ We need an animals.rs
❌ We need a use.rs



File System Tree



Module System Tree

source: http://www.sheshbabu.com/posts/rust-module-system/

Rust’s object paradigm

Structs or Classes?

- struct keyword
  - But closer to C++ class keyword
    - Fields are private by default
    - You can add methods to a struct
  - A bit like C structs
    - But with methods
    - And encapsulation

- impl keyword to add methods to a type

Structs or Classes?

- self
  - Equivalent to C++/Java ‘s this
  - Explicitly given as parameter to the function, no hidden magic variable
  - Therefore, a static function in C++/Java is simply a method without self as its first argument in Rust


```
struct Vector3 {
    x: u32, y: u32, z: u32,
}

impl Vector3 {
    pub fn add(&self, other: &Vector3) -> Vector3 {
        Vector3 {
            x: self.x + other.x,
            y: self.y + other.y,
            z: self.z + other.z,
        }
    }
}
```

Inheritance?

“If a language must have inheritance to be an object-oriented language, then Rust is not one”

- No inheritance in Rust, but Trait Inheritance
- Traits?
  - Similar to Interfaces in Java / Abstract Classes in C++
  - Define behavior that can be shared between types
  - Examples: Display, Debug, Clone, Add...


```rust
struct Vector3 {
    x: u32, y: u32, z: u32,
}

impl std::fmt::Display for Vector3 {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "<{}, {}, {}>", self.x, self.y, self.z)
    }
    // Now if we do `println!("{}", some_vector3)` it will display
    // correctly on stdout using the following format: <x, y, z>
}
```

```rust
pub trait Animal {
    // Default implementation for trait methods
    // Can be overriden
    fn eat() {
        println!("Miom miom miom");
    }

    fn scream();
}
```


```rust
pub trait Cat: Animal { // Inherit methods from trait Animal
    fn be_cute();
    fn meow();
}

// If you are a Cat, then you are also an Animal
// If you WANT to be a Cat, you NEED to be an Animal first
```


```rust
struct Panther {
    nb_dots: u32,
    eye_color: String,
}
```

```rust
impl Cat for Panther {
    fn be_cute() {
        println!("I'm not cute");
    }

    fn meow() {
        println!("RAAAAAAAWR");
    }
}
```

```
> cargo run
error[E0277]: the trait bound `Panther: Animal` is not satisfied
  --> src/main.rs:50:6
   |
14 | pub trait Cat: Animal {
   |                --- required by this bound in `Cat`
...
50 | impl Cat for Panther {
   |      ^^^^^^ the trait `Animal` is not implemented for `Panther`

error: aborting due to previous error
```

```rust
struct Panther {
    nb_dots: u32,
    eye_color: String,
}

impl Animal for Panther {
    // Default implem for eat()
    fn scream() {
        Panther::meow();
    }
}
```

```rust
impl Cat for Panther {
    fn be_cute() {
        println!("I'm not cute");
    }

    fn meow() {
        println!("RAAAAAAAWR");
    }
}
```

Encapsulation

- Useful in OOP languages
- Hides implementation details
- The API doesn’t change, but your internal implementation is allowed to

```rust
pub enum Category {
    Suv, Limousine, SportsCar, Coupe,
}

pub struct Car {
    pub nb_wheels: u8,
    pub hybrid: bool,
    pub kind: Category,
}
```

```rust
fn main() {
    let prius = Car {
        nb_wheels: 4,
        hybrid: true,
        kind: Category::SportsCar, // Vroom vroom
    };

    println!("My prius has {} wheels", prius.nb_wheels);
}
```

```rust
pub enum Category {
    Suv, Limousine, SportsCar, Coupe,
}

pub struct Car {
    nb_wheels: u8,
    color: String,
    brand: String,
    hybrid: bool,
    kind: Category,
} // Fields are now private
```


```
> cargo check # Remember?
error[E0451]: field `color` of struct `Car` is private
 --> src/main.rs:7:9
  |
7 |         color: String::from("Grey"),
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^ private field
```


```rust
impl Car { // Let's create "constructors"
    pub fn new(nb_wheels: u8, hybrid: bool, kind: Category) -> Car {
        Car { nb_wheels, hybrid, kind }
    }

    pub fn basic() -> Car {
        Car::new(4, false, Category::Suv)
    }

    pub fn basic_hybrid() -> Self {
        let mut hybrid_car = Car::basic();
        hybrid_car.hybrid = true;
        hybrid_car
    }
}
```


```rust
impl Car { // Let's add accessors
    // Getters need a non-mutable reference on self...
    pub fn is_hybrid(&self) -> bool {
        self.hybrid
    }
    pub fn nb_wheels(&self) -> u8 {
        self.nb_wheels
    }

    // ...While setters need to modify the instance, thus taking a mutable reference
    pub fn set_hybrid(&mut self, hybrid: bool) {
        self.hybrid = hybrid;
    }
}
```

```rust
fn main() {
    let honda = Car::basic_hybrid();
    let prius = Car::new(4, true, Category::SportsCar);


    println!("My Honda has {} wheels", honda.nb_wheels());
}
```

```
fn main() {
    // We could also implement the Builder pattern
    let prius = Car::new()
        .with_wheels(4)
        .with_hybrid(true)
        .with_kind(Category::SportsCar);
        // Remember clap?
}
```

Encapsulation

- Useful in OOP languages
- Hides implementation details
- The API doesn’t change, but your internal implementation is allowed to
- Where’s protected?

Questions?