Module 01: References And Slices

Introduction

Rust is basically operating on the same hardware abstraction as C. As such, it does have a way to create pointers to any existing value. In Rust, however, it is *impossible* to create invalid pointers. When using that language, the compiler *ensures* that every pointer you create won't ever be invalidated while you are using it. To provide this guarentee, Rust uses a system known as the *Borrow Checker*.

General Rules

Any program you turn in should compile using the `cargo` package manager, either with `cargo run` if the subject requires a *program*, or with `cargo test` otherwise. Only dependencies specified in the `allowed dependencies` section are allowed.

Any program you turn in should compile *without warnings* using the `rustc` compiler available on the school's machines without additional options. You are allowed to use attributes to modify lint levels, but you must be able to explain why you did so. You are *not* allowed to use `unsafe` code anywere in your code.

Exercise 00: Creating References

```
turn-in directory:
    ex00/

files to turn-in:
    src/lib.rs  Cargo.toml

allowed dependencies:
```

Creating a reference isn't exactly an involved process. Using properly those references can be quite a bit harder however.

Create a *function* that adds two integers. It should be prototyped as follows:

```rust
fn add(a: &i32, b: i32) -> i32;
```

Notice that `a` is a *reference* to an `i32`.

Now, create another functions, but this time, it should store the result of the operation in the first number.

```rust
fn add_assign(a: &mut i32, b: i32);
```

How does using a `&mut` or a `&` change the semantics of the function? Would it be possible to create an `add_assign` function using a regular `&i32` reference (without de `mut`). You should be able to answer this question during the defense.

As always provide some tests to prove every function behaves as expected.

Exercise 01: Dangling References

```
turn-in directory:
    ex01/

files to turn-in:
    src/main.rs  Cargo.toml
```

Rust won't ever allow you to create a dangling reference (a reference whose referenced value has been lost).

```rust
fn main() {
    let b;

    {
        let a: i32 = 0;
        b = &a;
    }

    println!("{}", b); // error!
}
```

This exercise simply requires you to understand why above code does not compile (and why it *shouldn't* compile). Don't try to fix it, just be prepared to being asked why happened here during defense.

Copy this flawed `main` into your project and move on to the next exercise.

Exercise 02: Point Of No Return (v2)

```
turn-in directory:
    ex02/

files to turn in:
    src/lib.rs  Cargo.toml

allowed dependencies:
```

Do you remember the point of the exercise 01 from the first module? You had to create a function prototyped as so:

```rust
fn min(a: i32, b: i32) -> i32;
```

The assignment of this exercise is to write the same exact function, but this time, the inputs of this function are references.

```rust
fn min(a: &i32, b: &i32) -> &i32;
```

The above function returns the reference to the smallest integer among `a` and `b`. Note that you may have to add some *lifetime annotations* to the function in order to make it compile.

Can you create a `spike` test that showcases how *not* having those annotations would be an issue? You can comment that non-compiling test out before pushing and explain that fairly difficult concept to your evaluators!

You must write tests!

Exericse 03: Array Addition

```
turn-in directory:
    ex03/

files to turn in:
    src/lib.rs  Cargo.toml

allowed dependencies:
```

What about arrays? Contiguous arrays are a core component of most languages out there, and Rust has them in two flavors.

The first flavor is *arrays*. Those are compile-time sized and can be created on the stack. Your assignment is to create a *function* that adds two of those vectors index-wise.

```rust
fn add_vectors(a: [i32; 3], b: [i32; 3]) -> [i32; 3];
```

Example:

```rust
let a = [1, 2, 3];
let b = [2, 3, 4];
assert_eq!(add_vectors(a, b), [3, 5, 7]);
```

You must write tests to prove your function is behaving as expected.

Exercise 04: Largest Subslice

```
turn-in directory:
    ex04/

files to turn in:
    src/lib.rs  Cargo.toml

allowed dependencies:
```

The second array flavor in Rust is *slices*. Slices are just like regular arrays, except their size is not known at compile time. Instead, each reference to a *slice* stores the number of elements they point to *with* their pointer, allowing the developper to easily create sub-slices.

Create a function that computes the smallest sub-slice whose sum is above a given treshold. You are only allowed to use the indexing operator `slice[...]`, you'll se it can be quite powerful. When multiple slices of the same length are above the treshold, the first one is returned. If no such slice is found, the empty slice is returned.

```rust
fn smallest_subslice(slice: &[u32], treshold: &u32) -> &[u32];
```

Once again, you may need to specify some *lifetime annotations* for the function. To check whether your annotations are correct for that case, you can use this pre-defined `test_lifetimes` function. It should compile.

```rust
#[test]
fn test_lifetimes() {
    let array = [3, 4, 1, 2, 12];
    let result;

    {
        let treshold = 1000;
        result = smallest_subslice(&array, &treshold);
    }

    assert_eq!(result, &[]);
}
```

Exercise 05: Sorting A Slice

```
turn-in directory:
    ex05/

files to turn in:
    src/lib.rs  Cargo.toml

allowed dependencies:
```

Iterating over an array is fine, but doing that while modifying it is better!

Create a function that sorts a slice of `i32`s. You cannot use anything other than the indexing operator `slice[...]`.

```rust
fn sort_slice(slice: &mut [i32]);
```

You must provide tests!

Exercise 06: The Size Of Slices

```
turn-in directory:
    ex06/

files to turn in:
    src/main.rs  Cargo.toml

allowed dependencies:
```

Copy this bit of code inside of the `main` function.

```rust
dbg!(std::mem::size_of::<i32>());
dbg!(std::mem::size_of::<&i32>());
dbg!(std::mem::size_of::<&[i32; 16]>());
dbg!(std::mem::size_of::<&[i32]>());
```

Do you understand why it prints those values? You will be asked during defense.