Module 02 - Structs n' Co

Introduction

Using primitive types will only get so far. Composing those types into larger types is the way to go! Rust makes good use of encapsulation, product and sum types, and more generally, algebraic data types to enpower developpers into constructing powerful abstractions without losing any bit of performance.

This module will show you how to create new types to better represent the data you manipulate. By the end, you should have a general understanding of how types are created in Rust, and how to use them.

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: A Point In Space

```
turn-in directory:
    ex00/

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

allowed dependencies:
```

Let's create a simple datastructure. Create a `struct` representing a 2D point, which itself describes the position of an object in 2D space. The created type should be named `Point`. The fields of `Point` must be accessible using the `.x` / `.y` syntax.

Your type has to implement the following associated functions:

- `new`, which creates a new `Point` instance with specific coordinates.
- `zero`, which creates a new `Point` at coordinates `(0, 0)`.
- `distance`, which computes the distance between two existing points.
- `translate`, which adds the vector `(dx, dy)` to the coordinates of the point.

```
impl Point {
    pub fn new(x: f32, y: f32) -> Self;
    pub fn zero() -> Self;
    pub fn distance(&self, other: &Self) -> f32;
    pub fn translate(&mut self, dx: f32, dy: f32);
}
```

Add tests to your crate to prove that each function is working as expected.

Note: you might need to add the `pub` modifier to your `struct` declaration in order to silence some warnings.

Exercise 01: Matching Color Names

```
turn-in directory:
    ex01/

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

allowed dependencies:
```

Here is the definition a type named `Color`. It is used to describe color using a red, a green and a blue component.

```
pub struct Color(u8, u8, u8);
```

You assignment is to create a `name` function that computes a string representation of any instance of the `Color` type. That function cannot be implemented using a forest of `if` statements. The prototype of that function must be:

```
impl Color {
    pub fn name(&self) -> &str;
}
```

The name of a color is determined using the following rules, applied in order. The first rule that `match`es the input color should be selected.

- The color `Color(0, 0, 0)` is "pure black".
- The color `Color(255, 255, 255)` is "pure white".
- The color `Color(255, 0, 0)` is "pure red".
- The color `Color(0, 255, 0)` is "pure green".
- The color `Color(0, 0, 255)` is "pure blue".
- The color `Color(128, 128, 128)` is "perfect grey".
- Any color whose components are all bellow 31 is "almost black".
- Any color whose red component is above 128, whose green and blue components are between 0 and 127 is "redish".
- Any color whose green component is above 128, whose red and blue components are between 0 and 127 is "greenish".
- Any color whose blue component is above 128, whose red and green components are between 0 and 127 is "blueish".
- Any other color is named "unknown".

You must include unit tests to prove the `name` function works as expected.

Exercise 02: Signing Numbers

```
turn-in directory:
    ex02/

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

allowed dependencies:
```

Create an `enum` that represents the sign of a number. That type should be named `Sign` and should be able to represent three separate states: one for when the value is zero, one for when the value is negative, and one for when the value is positive.

The sign type must implement a method to create itself from an existing integer.

```
impl Sign {
    pub fn of(i: i32) -> Self;
}
```

Add tests to show that the `Sign::of` function works as expected. Note that you may need to add the `#[derive(Debug, PartialEq)]` attribute to your type in order to use `assert_eq!` properly.

Exercise 03: Sharing Is Caring

```
turn-in directory:
    ex03/

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

allowed dependencies:
```

During the previous module, you should've understood that Rust wants to know how long references will live before compiling the code. It does that to make sure the code you are trying to write is indeed valid and cannot create dangling references. But what can you do if the type you create itself contains references?

Create a type named `Warior`. Wariors are very strong, but they need weapons to unlock their full potential. Create a `Weapon` type.

The `Weapon` type must be an `enum`. Each variant of that `enum` must be a weapon type. Add at least two weapon types. This type can have an associated `print` method to display its name.

```
impl Weapon {
    fn print(&self);
}
```

You can add the `pub` modifier on the `Weapon` type to silence some warnings about unused variants.

The `Warior` type must store a *reference* to a `Weapon`.

Create a `main` function that showcases two `Warior`s sharing the same `Weapon` instance.

Exercise 04: Literal Value

```
turn-in directory:
    ex04/

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

allowed dependencies:
```

C-like enumerations are nice, but we can do better!

Create an `enum` named `Literal`. That type should be able to represent the following literal values:

- A string (example: `"Hello, World!"`)
- An integer (example: `-1543`)
- A floating-point number (example: `14.2`)
- A boolean (example: `true`)

The `Literal` type should implement a method to print its content as if it were written in a code file. It should also have functions to determine the type currently represented.

```
impl Literal {
    pub fn display(&self);

    pub fn is_string(&self) -> bool;
    pub fn is_int(&self) -> bool;
    pub fn is_float(&self) -> bool;
    pub fn is_bool(&self) -> bool;
}
```

You have to include tests to prove the functions you wrote are indeed correct. For the `display` function, you have to include a `main`.

```
>_ cargo run
12.3
"Hello, World!"
false
true
1
-14
```

Exercise 05: Type Aliases

```
turn-in directory:
    ex05/

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

allowed dependencies:
```

Rust allows you to rename types. This functionality can be used to provide more precise type-documentation to existing functions.

Copy/Past the following code and make it compile by adding type alias definitions.

```
fn seconds_to_minutes(seconds: Seconds) -> Minutes {
    seconds / 60.0
}

fn main() {
    let s: Seconds = 120.0;
    let m: Minutes = seconds_to_minutes(s);

    println!("{s} seconds is {m} minutes");
}
```

```
>_ cargo run
120 seconds is 2 minutes
```

Exercise 06: Update Syntax

```
turn-in directory:
    ex06/

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

allowed dependencies:

```

Here is a `struct` definition.

```
pub struct User {
    pub id: u64,
    pub first_name: &'static str,
    pub last_name: &'static str,
    pub email: &'static str,
    pub phone_number: &'static str,
    pub password: &'static str,
}
```

Copy this definition in your `lib.rs` file, and create a function to create the following functions.

```
impl User {
    pub fn empty() -> Self;
    pub fn new(id: u64, first_name: &'static str) -> Self;
}
```

The `User::empty` method must create a new `User` with default parameters. The fields of the returned instance must be initialized to the following values.

```
id              : 0
first_name      : "Jean"
last_name       : "Dupont"
email           : "jean.dupont@example.net"
phone_number    : "00 00 00 00 00"
password        : "11223344"
```

The `User::new` method must create a new `User` with the exact same values as the value returned by `User::basic`, but with `id` and `first_name` set to specific values. You must implement this function using Rust's *update syntax*.

You have to write tests to prove the functions you wrote are working as expected.

Exercise 07: MicroComputer

```
turn-in directory:
    ex07/

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

allowed dependencies:

```

Let's simulate a simple computer using the Rust programming language. A computer is basically made of two things: a CPU (Central Processing Unit), and some kind of memory storage.

Create a `Computer` type, which stores the state of a the simulated computer's CPU as well as its internal RAM (it can be implemented as an array of 256 bytes). The CPU supports four 8-bit registers, **A**, **B**, **C** and **D**. A register a value stored directly baked into the CPU; registers do not have a memory address, but they can be used to reference memory stored in the RAM.

It is able to execute the following operations.

- **Copy** the content of a register into another register.
- **Read** the memory pointed by a register into a register (potentially the same).
- **Write** the content of a register at the memory pointed by a register (potentially the same).
- **Set** a specific register to a given value.
- **Increment** the value of a register by 1 (the operation should wrap on overflow).
- **Decrement** the value of a register by 1 (the operation should wrap on overflow).
- **Jump** to a specific instruction index.
- **JumpIfZero** if the register **A** has the value `0`, then jump to a specific instruction index.

The operations supported by the CPU must be implemented as an `enum` named `Instruction`.

A program is simply an ordered list of instructions. It executes those instructions in order, starting from the first one in the list. The `Computer` type have to implement an associated method to execute a specific program. The function ends as soon as the current instruction index gets out of bounds. You should also add a `new` function to easily create new instances of the type.

```
impl Computer {
    pub fn new() -> Self; // optional
    pub fn execute(&mut self, program: &[Instruction]);
}
```

As always, you must include tests to show that the different operations do behave as they should.

Here is a program that you should be able to execute on the created computer, as long as a null-terminated string exists at address 0x00 in the RAM of the computer.

```
 00  SET B to 0
01  READ B into A
02  JUMP_IF_ZERO to 05
03  INCREMENT B
04  JUMP to 01
05  Set C to 255
06  WRITE B at C
```