Rust : Structures and Enumerations

David Bouchet

david.bouchet.epita@gmail.com

Declaring Structures (1)

Use the **struct** keyword.

```
struct Person
{
    first_name: String,
    last_name: String,
    age: u8,
}
```

Person is now a new type.

Declaring Structures (2)

Camel case should be used.

```
struct person
{
    first_name: String,
    last_name: String,
    age: u8,
}
```

```
warning: type `person` should have a camel case name such as `Person`
 --> declaring_lowercase.rs:1:1
  |
1 | / struct person
2 | | {
3 | |     first_name: String,
4 | |     last_name: String,
5 | |     age: u8,
6 | | }
  | |_^
```

Assigning Values

```
let p1 = Person
{
    first_name: String::from("René"),
    last_name: String::from("Descartes"),
    age: 53,
};

let p2 = Person
{
    first_name: String::from("Emmanuel"),
    last_name: String::from("Kant"),
    age: 79,
};
```

Accessing Fields

Use dot-notation.

```
let p = Person
{
    first_name: String::from("René"),
    last_name: String::from("Descartes"),
    age: 53,
};

dbg!(p.first_name);
dbg!(p.last_name);
dbg!(p.age);
```

```
p.first_name = "René"
p.last_name = "Descartes"
p.age = 53
```

Modifying Fields

Variables must be mutable.

```
let mut p = Person
{
    first_name: String::from("René"),
    last_name: String::from("Descartes"),
    age: 53,
};

p.age = 10;

dbg!(p.first_name);
dbg!(p.last_name);
dbg!(p.age);
```

```
p.first_name = "René"
p.last_name = "Descartes"
p.age = 10
```

Printing Structures (1)

```
let p = Person
{
    first_name: String::from("René"),
    last_name: String::from("Descartes"),
    age: 53,
};

println!("p = {}", p);
```

```
error[E0277]: `Person` doesn't implement `std::fmt::Display`
  --> printing_structures.rs:17:24
   |
17 |     println!("p = {}", p);
   |                        ^ `Person` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Person`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
   = note: required by `std::fmt::Display::fmt`
```

Printing Structures (2)

```
let p = Person
{
    first_name: String::from("René"),
    last_name: String::from("Descartes"),
    age: 53,
};

println!("p = {:?}", p);
```

```
error[E0277]: `Person` doesn't implement `std::fmt::Debug`
  --> printing_structures.rs:17:26
   |
17 |     println!("p = {:?}", p);
   |                          ^ `Person` cannot be formatted using `{:?}`
   |
   = help: the trait `std::fmt::Debug` is not implemented for `Person`
   = note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug`
   = note: required by `std::fmt::Debug::fmt`
```

Printing Structures (3)

Use the derive attribute.

```
#[derive(Debug)]
struct Person
{
    first_name: String,
    last_name: String,
    age: u8,
}
```

- Some code is automatically generated to implement the Debug formatter.
- The same goes for the dbg!() macro.

Printing Structures (4)

```
let p = Person
{
    first_name: String::from("René"),
    last_name: String::from("Descartes"),
    age: 53,
};

println!("p = {:?}", p);
```

```
#[derive(Debug)]
struct Person
{
    first_name: String,
    last_name: String,
    age: u8,
}
```

```
p = Person { first_name: "René", last_name: "Descartes", age: 53 }
```

Declaring Enumerations

Use the **enum** keyword.

```
#[derive(Debug)]
enum Color
{
    Black,
    Red,
    Green,
    Blue,
    Grey,
    White,
}
```

For display purposes only.

Color is now a new type.

Variants

Use camel case for both type names and variants.

Assigning Values

```
let c1 = Color::Black;
let c2 = Color::Green;

dbg!(c1);
dbg!(c2);
```

```
c1 = Black
c2 = Green
```

Variants with Data

```
#[derive(Debug)]
enum Shape
{
    Point,                      // No dimension
    Rectangle(u8, u8),          // Width, height
    Square(u8),                 // Side length
}
```

```
let p = Shape::Point;
let r = Shape::Rectangle(6, 8);
let s = Shape::Square(8);

println!("p = {:?}", p);
println!("r = {:?}", r);
println!("s = {:?}", s);
```

```
p = Point
r = Rectangle(6, 8)
s = Square(8)
```

Anonymous Structures

```
#[derive(Debug)]
enum Shape
{
    Point,
    Rectangle { w: u8, h: u8 },
    Square(u8)
}
```

Anonymous Structure

```
let p = Shape::Point;
let r = Shape::Rectangle { w: 6, h: 8 };
let s = Shape::Square(8);

println!("p = {:?}", p);
println!("r = {:?}", r);
println!("s = {:?}", s);
```

```
p = Point
r = Rectangle { w: 6, h: 8 }
s = Square(8)
```

The `Option<T>` Enumeration (1)

Option<T> is defined by the standard library.
Used when no value may be returned.

```
enum Option<T>
{
    Some(T),
    None,
}
```

T can be any type.

The Option<T> Enumeration (2)

```
let i1: Option<i32> = Some(42);      // Option<i32>
let i2: Option<i32> = None;          // Option<i32>
let i3 = Some(24);                   // Option<i32>
let s1 = Some("Hello World");        // Option<&str>
let s2: Option<&str> = None;         // Option<&str>

// s2 = i2;     ERROR even if s2 is mutable
```

```
println!("i1 = {:?}", i1);
println!("i2 = {:?}", i2);
println!("i3 = {:?}", i3);
println!("s1 = {:?}", s1);
println!("s2 = {:?}", s2);
```

```
i1 = Some(42)
i2 = None
i3 = Some(24)
s1 = Some("Hello World")
s2 = None
```

The `Option<T>` Enumeration (3)

```
let i1 = Some(42);
let i2 = Some("Hello World");
let i3: Option<i32> = None;

println!("i1 = {:?}", i1);
println!("i2 = {:?}", i2);
println!("i3 = {:?}", i3);

println!("i1.unwrap() = {}", i1.unwrap());
println!("i2.unwrap() = {}", i2.unwrap());
println!("i3.unwrap() = {}", i3.unwrap());  // Panics
```

The `Option<T>` Enumeration (4)

```
i1 = Some(42)
i2 = Some("Hello World")
i3 = None
i1.unwrap() = 42
i2.unwrap() = Hello World
thread 'main' panicked at 'called
`Option::unwrap()` on a `None` value'...
```

The `Result<T, E>` Enumeration (1)

Result<T, E> is defined by the standard library.
Used when an error may be returned.

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

T and E can be any type.

The `Result<T, E>` Enumeration (2)

```
fn div(a: i32, b: i32) -> Result<i32, String>
{
    if b == 0
    {
        Err(String::from("Division by zero."))
    }

    else
    {
        Ok(a / b)
    }
}
```

The `Result<T, E>` Enumeration (3)

```
let q = div(10, 2);
println!("q = {:?}", q);

let q = div(10, 0);
println!("q = {:?}", q);
```

```
q = Ok(5)
q = Err("Division by zero.")
```

The `Result<T, E>` Enumeration (4)

```
let q = div(10, 2);
println!("q.unwrap() = {:?}", q.unwrap());

let q = div(10, 0);
println!("q.unwrap() = {:?}", q.unwrap());  // Panics
```

```
q.unwrap() = 5
thread 'main' panicked at 'called `Result::unwrap()`
on an `Err` value: "Division by zero."'...
```