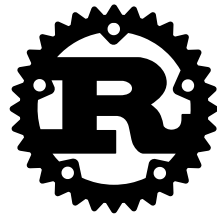


Rust for Safety



General coding tips

Overview

This chapter will introduce you to useful things you need to get started. It is not exhaustive.

Structs

```
#[derive(Eq, PartialEq, Debug)]
```

1

```
pub struct Point {
```

2

```
    x: i32,
```

```
    y: i32,
```

```
}
```

- 1 Derives allow to generate some standard functionality
- 2 Any type can carry a visibility modifier to export them

Useful Derives: Debug

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 1, y: 2 },
    println!("{:?}", p); 1
    println!("{:#?}", p); 2
}
```

- 1 Debug makes the Debug formatting string work
- 2 There's also a more structured version, also enabled through it

Useful Derives: Eq, PartialEq

```
#[derive(Eq, PartialEq, Debug)]
```

① ②

```
struct Point {  
    x: i32, ③  
    y: i32,  
}
```

```
fn main() {  
    let p1 = Point { x: 1, x: 2  
},  
    let p2 = Point { x: 1, x: 2  
},  
    if p1 == p2 { ④  
        println!("The same!");  
    }  
    assert_eq!(p1, p2); ⑤  
}
```

- ① Eq describes total equality: for every pair of values, equality is defined
- ② PartialEq is enough for getting ==
- ③ Both can only be derived if all inner fields are both
- ④ Equality in action!
- ⑤ The `assert_eq!` compares to values and panics if they don't match!

Enums

```
#[derive(Eq,PartialEq,Debug)]
```

2

```
enum Direction {  
    North, 1  
    South, 1  
    West, 1  
    East, 1  
}
```

```
fn main() {  
    let dir = Direction::North;  
    3  
    println!("Direction: {:?}",  
dir);  
    let dir2 =  
Direction::North;  
    assert_eq!(dir, dir2);  
}
```

- 1 Enums carry multiple *variants*.
- 2 Derives work as expected.
- 3 Variants are referred to through their main type.

Enums with data

```
#[derive(Eq, PartialEq, Debug)]
```

```
enum Movement {  
    North(u32), ①  
    South(u32),  
    West(u32),  
    East(u32),  
    StickAround, ②  
}
```

```
fn main() {  
    let mov =  
    Movement::North(1); ③  
    println!("Movement: {:?}",  
    dir);  
    let mov2 =  
    Movement::StickAround;  
    assert_eq!(mov, mov2);  
}
```

- ① Variants can carry data.
- ② Variants carrying data and not can be mixed.
- ③ Construction works similar to structs.

Usage: Option

```
fn main() {  
    let vec = vec![1,2];  
    let mut iter = vec.iter();  
    check(iter.next()); ①  
    check(iter.next()); ②  
    check(iter.next()); ③  
}
```

- ① prints "1"
- ② prints "2"
- ③ prints "Iterator exhausted"

```
fn check(item: Option<u32>) {  
    match item {  
        Some(value) =>  
            println!("{}",  
value),  
        None =>  
            println!("Iterator  
exhausted"),  
    }  
}
```

Conclusion: Option

`Option` encodes the *potential, but expected* absence of a value.



Due to optimisations, **`Option<u32>`** is as large as **`u32`**.

Useful enums: Result

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

- 1 Results are generic over *two* types.
- 2 One is the value indicating success.
- 3 The other is the value type indicating error.

Result usage

```
fn main() -> Result<(),  
io::Error> {  
    let file_res: Result<File,  
io::Error> = ❷  
        File::open("test"); ❶  
    match file_res { ❸  
        Ok(file) => {  
            //...  
            Ok( ( ) ) ❹  
        },  
        Err(e) => { ❺  
            println!("Error  
opening: {}", path);  
            Err(e) ❻  
        }  
    }  
}
```

- ❶ open returns a `Result` indicating success or failure
- ❷ Type annotation for clarity
- ❸ `Result`'s are also handled with match`
- ❹ If success holds no value, `Ok` with `()` is used. Usually written `Ok(())`.
- ❺ Errors are handled the same
- ❻ If errors are passed on, the must be wrapped again

Special behaviour: must be used

```
fn main() -> Result<(), io::Error> {  
    File::open("test"); ❶  
}
```

warning: unused `std::result::Result` that must be used

```
--> scratch.rs:2:5  
  |  
2 |     std::fs::File::open("test");  
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
  |
```

= note: `#[warn(unused_must_use)]` on by default

= note: this `Result` may be an `Err` variant, which should be handled

Conclusion: Result

`Result` encodes the potential for error. It forces the user to inspect the `Result` and check if an error occurred or denies access to the inner value otherwise.



Similar optimisations as to **`Option`** apply to **`Result`**

Slices and Vectors

```
fn main() {  
    let bytes: &[u8] = &  
    [1,2,3]; ❶  
    let vec: Vec<u8> =  
    Vec::from(bytes); ❷  
    vec.push(4);  
    let vec_slice: &[u8] =  
    vec.as_slice(); ❸  
}
```

- ❶ The (reference) slice `&[..]` is a reference to a region of memory. It stores the length of the data and bounds checked.
- ❷ The vector `Vec<..>` is an owned region of memory. It is growable and shrinkable.
- ❸ Slices can be taken to the memory a vector owns, binding the slices to the vector.

Vector reallocation

```
fn main {  
    let mut v = vec![1,2]; ①  
    let slice = &v[..]; ②  
  
    vec.push(4); ③  
    //^^^^^^ Error here  
  
    println!("{:?}", slice);  
}
```

- ① This is a shorthand vector initialization macro.
- ② Taking a slice borrows the memory region the vector owns. `&v[..]` is syntax for "borrow from beginning to end".
- ③ Pushing on vector requires a mutable reference to it, violating borrow rules.

Strings and their slices

Strings and string slices work much the same.

```
fn main() {  
    let slice: &str = "Hello world!";  
    let string: String = String::from(slice);  
}
```

Strings and string slices are `Vec<u8>` and `&[u8]` internally, with the added invariant that they are UTF-8.

Use owned strings



Spending the memory to make a heap copy and owning data makes your life a lot easier.

```
struct Owned {  
    string_data: String  
}
```

```
struct Borrowed<'a> {  
    slice: &'a str  
}
```

Warning



Never bypass the UTF-8 invariant on **String** or **&str**, this might lead to memory unsafety.



This is also not possible in safe Rust.

Owned vs. borrowed types

Borrowed types need you to make sure that the pointee is always alive. Owned types are easier. Liberally allocate `String` and `Vec` if you run into problems.



Rust makes it easy to safely refactor towards more efficient code. Get something working first, before you avoid each and every copy.

Testing

```
use my_library::my_function; ❶
```

```
#[test]
```

```
fn my_test() {  
    assert_eq!(1, 1);  
}
```

```
#[test]
```

```
#[should_fail]
```

```
fn failing_test() {  
    assert_eq!(1, 2);  
}
```

Rust and Cargo allows you to easily provide test for your code.

These can be put either directly in the source file or in any file in tests.

- ❶ Only needed when putting files in tests.