PROGRAMMING CONCEPTS AND LANGUAGES, 2024W

RUST Programming Language: Part II

CONTENTS

Part I: Short Recap

- Basic data types, stack and heap, variables, pointers, references, ownership and borrowing, …

Next

- Tuples, Arrays, Structs
- Traits
- Enums
- Control Flow (Conditionals, Loops)
- Common Collections

Literature

- https://doc.rust-lang.org, https://github.com/rust-lang/rustlings/, https://doc.rust-lang.org/rust-by-example/
- https://play.rust-lang.org/

RUST :: TUPLES

Compound types can group multiple values into one type

- Rust has two primitive compound types: **tuples** and **arrays**

Tuple

- Grouping together values with a variety of types into one compound type
- Fixed length - once declared they cannot grow or shrink
- Individual values are called elements
- Specified as a comma-separated list enclosed in parentheses (`<value>, <value>, ...`).
- The tuple without any values has a special name, **unit**
  - **Unit is** written as **()** and represent an empty value or a return type

```
fn main() {
  // let tup: (i32, &str, bool) = (42, "answer", true);
  let tup: (i32, &str, bool) = (42, "answer", true); // variable tup bound to the whole tuple

  println!("{} is the {}! ({})", tup.0, tup.1, tup.2);
}
```

RUST :: TUPLES :: EXAMPLES

Tuples can hold different types

```
let x = (1, "hello");
// or: let x: (i32, &str) = (1, "hello");
```

Accessing fields in a tuple through *indexing* or a *destructuring let*

```
let tuple = (1, 2, 3);
let x = tuple.0;
let mut y = tuple.1;
let z = tuple.2;
println!("x: {}, y: {}, z: {}", x, y, z);
```

Disambiguate a single-element tuple from a value in parentheses with a comma

```
(0,); // A single-element tuple.
(0); // A zero in parentheses.
```

RUST :: ARRAYS

Fixed-size list of elements of the same type

- Immutable by default
- A list type of `[T; N]`
  - T a substitute of some type T (generics)
  - N is a compile time constant representing the length of the array → can neither grow or shrink (fixed length)!
- Values as a comma-separated list inside square brackets
- Values accessed using indexing (square brackets)
- Contiguous in memory on the stack of known size
- Accesses checked at the compile time and at runtime

```
fn main() {
  let a = [1, 2, 3];      // a: [i32; 3]
  let mut m = [1, 2, 3]; // m: [i32; 3]

  let a = [0; 20];        // a: [i32; 20]

  let a = [1, 2, 3];      // shadowing
  println!("a has {} elements", a.len());

  // names: [&str; 3]
  let names = ["Marvin", "Arthur", "Zaphod"];
  println!("The second name is: {}", names[1]);
}
```

Output:

```
a has 3 elements
The second name is: Arthur
```

see: https://doc.rust-lang.org/book/ch03-02-data-types.html

RUST :: ARRAYS :: EXAMPLE, RUNTIME CHECKS

```rust
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("Please enter an array index.");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {index} is: {element}");
}
```

```
Please enter an array index.
7
thread 'main' panicked at 'index out of bounds: the len is 5 but the index is 7',
.\main.rs:19:19
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

see: https://doc.rust-lang.org/book/ch03-02-data-types.html

RUST :: STRUCTURES

A type that is composed of other types

- Defined with the **`struct`** keyword
- Elements in a struct are called ***fields***
- The fields in a struct can have different data types
- Fields can be named, so it's clear what the value means

```
struct User {
  active: bool,
  username: String,
  email: String,
  sign_in_count: u64,
}
```

`struct` definition

Three types

- Classic C-like structs, accessed by using the syntax `<struct>.<field>`
- Tuple structs, where fields have no names, e.g., `struct Color(u8, u8, u8);`
- Unit structs that are used as markers
  - Similar to `()`, e.g., `struct MyStruct;`
  - Useful for implementing traits on a type

RUST :: STRUCTURES :: EXAMPLE

```
struct User {
  active: bool,
  username: String,
  email: String,
  sign_in_count: u64,
}

fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");

    let user2 = User {
      email: String::from("another@example.com"),
      ..user1
    };
    struct Color(i32, i32, i32);
    struct MyStruct;
}
```

How about using `&str` instead of `String`?

- We want the struct to own its data
- Storing references in a struct is possible but requires the use of lifetime specifiers!

RUST :: SLICES

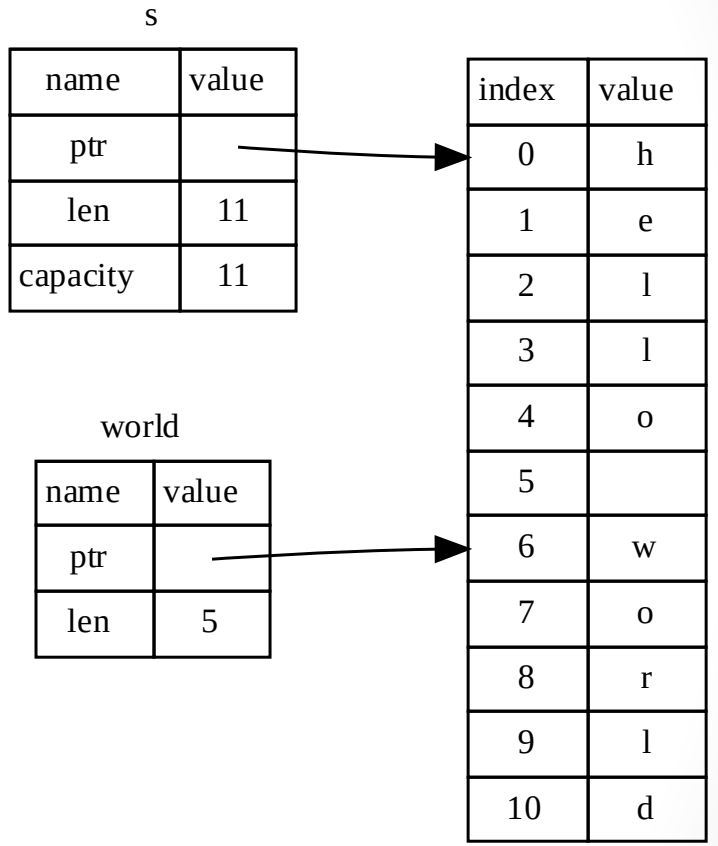
Allows referencing a contiguous sequence of elements in a collection

- A slice is a kind of **reference**, so it does not have ownership

```
fn main() {
    let s = String::from("hello world");
                         ^^^^^^^^^^^^^
                         string literal

    let hello = &s[0..5];
    let world = &s[6..11];

    println!("{} {}", hello, world);
}
```



Here we have a "string slice"

- an `&str` represents an immutable string literal (e.g., "hello world")
- "String" is heap-allocated string type that owns its contents

We can also take slices of an array of other types

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
```

RUST :: TRAITS

A trait defines functionality a particular type has and can share with other types

- Defines a **shared behavior** in an abstract way
- Like interfaces are in other languages, but with some differences
- Set of methods that can be implemented for **multiple types** to provide **common functionality** and behavior between them
- Consists of method signatures, which then have to be implemented by the target type
- Can be passed as parameters

Example (on the right)

- The `Animal` trait defines only the signature of the sound method
- We implement this trait for every type we want

```
trait Animal {
  fn sound(&self) -> String;
}

struct Sheep;
struct Cow;

impl Animal for Sheep {
  fn sound(&self) -> String {
     String::from("Maah");
  }
}

impl Animal for Cow {
  fn sound(&self) -> String {
     String::from("Moo");
  }
}
```

See https://doc.rust-lang.org/book/ch10-02-traits.html

RUST :: DERIVABLE TRAITS

Traits that can be automatically implemented for a `struct` or an `enum` by the Rust compiler

- Called *derivable* because they can be derived automatically

Most common are:

- **Debug**: Allows output of content via "{:?}"
- **Clone**: Enables type to be duplicated with clone() method
- **Copy**: Enables type to be copied implicitly, without requiring explicit clone() method
- **PartialEq**: Enables comparison

Example

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

RUST :: DERIVABLE TRAITS :: EXAMPLE

```rust
use std::ops::Add; // operator overloading

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

impl Add for Point {
    type Output = Self;

    fn add(self, other: Point) -> Self {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    let p2 = Point { x: 3, y: 3 };
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 }, p2);

   println!("{:?}, {:?}", p2, Point { x: 1, y: 0 } + Point { x: 2, y: 3 });
}
```

Associated types that determines the type returned from the add method

See: https://doc.rust-lang.org/book/ch19-03-advanced-traits.html

RUST :: TRAIT AS A PARAMETER

Traits can be passed as parameters to functions

- Example: the function takes as an argument any type that has implemented the Summary trait

```
pub fn notify(item: &impl Summary) {
  println!("Breaking news: {}", item.summarize());
}
```

Traits Bounds

- Traits can be declared as generics
- Type T must implement the Summary trait
- Similar to above, but more verbose
- Can be useful when you have more parameters that implement the same type
- Can also be specified in different ways (see `where` clause)

```
pub fn notify<T: Summary> (item1: &T, item2: &T) {
  println!("Breaking news: {}, {}", item1.summarize(), item2.summarize());
}
```

See https://doc.rust-lang.org/book/ch10-02-traits.html

RUST :: DISPLAY

Used for customizing the output

- Cannot be automatically derived
- Placeholder for display is `{}` (as in `println!("{}", myvar);`
  - For Debug the placeholder is `{:?}` (as in `println!("{:?}", myvar);`

Example

```
use std::fmt;

struct Complex {
   a: i32,
   b: i32,
}

impl fmt::Display for Complex {
   fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
      write!(f, "({} + {}i)", self.a, self.b)
   }
}

fn main() {
   let c = Complex{a: 2, b: 3};

   println!("{}", c);
}
```

RUST :: RETURN TYPES THAT IMPLEMENT TRAITS

The `impl` Trait syntax can also be used in the return position to return a value of some type that implements a trait

Example

- This method returns some type that implements the Summary trait
- No concrete type has been named
- Note: you can only use `impl` Trait if you're returning a single type

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        // ...
    }
}
```

See https://doc.rust-lang.org/book/ch10-02-traits.html

RUST :: ENUMERATIONS

Enums (enumerations) allows defining a type by enumerating its possible variants

- We use the enum keyword to create an enumeration type
- Each enum variant can hold its own data
- enum variants can have named fields, but they can also have fields without names, or no fields at all.
- enum types are capitalized
- variants of the enum are *namespaced* under its *identifier*

```
enum IpAddr {
    V4,
    V6,
}
```

Examples

```
enum IpAddr {
    V4(String), // we could also use V4(u8, u8, u8, u8),
    V6(String),
}

fn main() {
    let home = IpAddr::V4(String::from("127.0.0.1"));
    let loopback = IpAddr::V6(String::from("::1"));
}
```

```
struct Ipv4Addr {
    // ...
}

struct Ipv6Addr {
    // ...
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

See https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

RUST :: OPTION ENUM

The `Option` type encodes the very common scenario in which a value could be something or it could be nothing

- Represents a value that may or may not be present
- Used for preventing common bugs, e.g., where returning a value may fail
- Also Rust has no `null` feature (unlike C++ for example)
- The Option enum is available by default
- Also its variants are available by default (even without the "`Option::`" prefix)
- The Option enum looks like this

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

See https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

RUST :: OPTION ENUM :: EXAMPLE

```rust
fn main() {
    let x: i8 = 5;
    let y: Option<i8> = Some(5);

    let sum = x + y;

    println!("sum is: {}", sum);
}
```

Does this work?

- No → it will fail to compile

```
error[E0277]: cannot add `Option<i8>` to `i8`
 --> enums2.rs:6:17
  |
6 |     let sum = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `Add<Option<i8>>` is not implemented for `i8`
  = help: the following other types implement trait `Add<Rhs>`:
            <&'a i8 as Add<i8>>
            <&i8 as Add<&i8>>
            <i8 as Add<&i8>>
            <i8 as Add>

error: aborting due to previous error
```

See https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

RUST :: STATEMENTS AND EXPRESSIONS

Statements

- Instructions that perform some action but do not produce a value
- E.g., code that ends with ';', function definitions, …

Expressions

- Evaluate to a resultant value
- All control flow tools are expressions
  - In C/C++ these are mainly statements (e.g., if, switch)
- Also (expr1 ? expr2 : expr3) does not exist in Rust

```
fn main() {
    let (a, b) = (42, 43);
    let res = if a < b {
        true
    } else {
        false
    };

    println!("res is {}", res);
}
```

BLOCKS AND SEMICOLONS

Block Expression

- Produces a value and can be used whenever a value is needed
- Note that there is no semicolon ";"
- If there is a semicolon at the end of an expression, return value is dropped.
- Examples:
  - `let x = 42; // semicolon is always required`

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
}
```

Simple expression

Block expression

RUST :: CONTROL FLOW

Controling the order in which statements or instructions are executed in a program

Examples (Conditionals):

- `if / else`
- `match`

Examples (Loops):

- `for / while / loop`
- `continue / break`

RUST :: CONTROL FLOW :: IF / ELSE

Each condition must be an expression of type bool

- Rust does not implicitly convert numbers or pointers to Boolean values
- No parentheses are required around conditions
- Curly braces "{" and "}" are required
- `else` and `else if` blocks are optional

```
if condition {
    block1
} else if condition {
    block2
} else {
    block3
}
```

Examples:

```
if a < b {
    true
} else {
    false
};
```

```
let is_valid = true;
let number = if is_valid { 5 } else { 6 };
```

*All blocks of an if expression must produce values of the same type!

RUST :: CONTROL FLOW :: MATCH

A powerful control flow construct

- Compare a *value* against a series of *patterns* and then execute code based on which pattern matches
- Can be made up of literals, variables, wildcards, etc.
- It does not need to evaluate to boolean
- Matches are exhaustive → patterns must cover all possibilities

```
match value {
    pattern => expr,
    ...
}
```

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

RUST :: CONTROL FLOW :: MATCH WITH OPTION

```
fn main() {
    let x: i8 = 5;
    let y: Option<i8> = Some(6);

    let sum = match y {
        None => x,
        Some(i) => x + i
    };
    println!("sum is: {}", sum);
}
```

Does this work? → yes

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

A function that returns an Option enum

- If there is a value add 1, otherwise return None

RUST :: CONTROL FLOW :: MATCH WITH OPTION

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

Does this work?

- No - we did not handle the None case (matches are exhaustive)
  - Will return "^ pattern `None` not covered" error
- We could use "_" Placeholder (catch-all pattern) to handle all other possibilities

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
        _ => None
    }
}
```

RUST :: LOOPS

Loops are expressions in rust

- Can be used with the `continue` and `break` keywords

`loop`

- Executing a code block until explicitly stopped (e.g., with the `break` with keyword)
- Can also be used in combination with the `continue` keyword

```
loop {
   block
}
```

`while` (a conditional loop)

- Loop while a condition is true

```
while condition {
   block
}

while let pattern = expr {
   block
}
```

`for` (looping through a collection)

```
for pattern in iterable {
   block
}
```

RUST :: LOOPS :: EXAMPLES

```
let mut counter = 0;
let result = loop {
   counter += 1;
   if counter == 10 {
      break counter * 2;
   }
};
println!("The result is {result}");
```

`loop` example

- Counter is incremented until it is 10
- Break stops the loop and returns the value

```
let mut number = 3;

while number != 0 {
   println!("{number}!");

   number -= 1;
}
```

`while` example

- The number is printed and decremented until the condition evaluates to false

RUST :: LOOPS :: FOR-LOOP EXAMPLES

Iterating over an array of integers

- Here the elements in **a** are copied (no move here)

```
let a = [10, 20, 30, 40, 50];

for element in a {
   println!("the value is: {element}");
}
```

Iterating over an array of strings

```
let a = [String::from("a"), String::from("b"), String::from("c")];
for element in a {
    println!("the value is: {element}");
}

println!("the array is: {:?}", a); // Does not compile → violation of the ownership rules
```

Instead, we could do

```
let a = [String::from("a"), String::from("b"), String::from("c")];
for element in &a {
    println!("the value is: {element}");
}

println!("the array is: {:?}", a);
```

RUST :: LOOPS :: FOR-LOOP EXAMPLES

Using a range

- With provided range (1..4)
- .rev() reverses the order

```
for element in (1..4).rev() {
   println!("the value is: {element}");
}
```

Iterating the indexing and values in an array

- Here we use `enumerate()`, which is implemented in the iterator trait

```
let a = [1, 2, 3, 4, 5, 6];
for (k, v) in a.iter().enumerate() {
   println!("the element at position {} has this value: {}", k, v);
}
```

RUST :: COMMON COLLECTIONS

Rust's standard library includes a number of data structures called *collections*

- Can contain multiple values
- Stored on the heap
- Able to grow/shrink at runtime
- Collections have different capabilities and costs
  - It is important to choose the right one for the problem at hand
- There are 4 groups of collections in std*:
  - Sequences (Vec, VecDeque, LinkedList),
  - Maps (HashMap, BTreeMap),
  - Sets (HashSet, BTreeSet,
  - Misc (BinaryHeap)

Most common collections are:

- **Strings** – collections of characters
- **Vectors** – for storing a number of values next to each other
- **HashMap** – associating a value with a particular key

*https://doc.rust-lang.org/std/collections/

RUST :: VECTORS

Storing more than one value in a single data structure

- Created with `Vec<T>`
- Contiguous in memory
- Stored on the heap
  - Vec is a (pointer, capacity, length) triplet
- Can dynamically grow/shrink
- Dropping a vector drops its elements

Example

```
fn main() {

   let mut vec1: Vec<i32> = Vec::new(); // create an empty vector to hold values of type i32
   vec1.push(1);

   let vec2 = vec![1, 2, 3]; // usage of vec! macro to create that hold the given values

   println!("vec1[0]: {}, vec2[0]: {}", vec1[0], vec2[1]);
}
```

RUST :: VECTORS :: READING

Two ways of referencing a value in vector

- Via indexing (with square brackets [])
  - Returns a reference to the element at the index value (panics if you use an index which is out of range)
- Via get method
  - Returns an Option<&T> that we can use with match
  - Allows you to define what happens if an index outside of the vector is used, returning None instead of panicking at runtime
- *Note that you can also use slicing

Example

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("The third element is {third}");

let third: Option<&i32> = v.get(2);
match third {
    Some(third) => println!("The third element is {third}"),
    None => println!("There is no third element."),
}
```

RUST :: VECTORS :: MODIFYING

Ownership rules are enforced on valid references to elements of a vector

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];

    let first = &v[0];

    v.push(6);

    println!("The first element is: {first}");
}
```

Does this work?

- No, because of the rules of ownership and borrowing
  - An immutable reference to the first element is held, but the push methods try to change the vector, which may cause the date to relocate, error message:

```
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
```

RUST :: VECTORS :: ITERATING

We can use loops to iterate over a vector

- Here we use a for-loop to get immutable references of each element of the vector

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{i}");
}
```

- We can also modify values by using mutable references

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50; // we have to use the dereference operator to change the value
}
```

*We can also use iterators with collections, but we will come back to that

RUST :: CLOSURES

Anonymous functions you can save in a variable or pass as arguments to other functions

- Can capture values from the scope in which they're defined
- Allow for code reuse and behavior customization
- Don't usually require you to annotate the types of the parameters or the return value like fn functions do
- Typically, short and relevant in a narrow context
- Captures variables in the least restrictive manner possible (see example below)
  - Can be done in three ways borrowing immutably, borrowing mutably and takin ownership
  - The compiler will decide based on the body of the function

Example

- This closure captures the value of x and modifies it
- A mutable reference of X is taken automatically, since taking ownership will be more restrictive

```
fn main() {
  let x = 41;
  let closure = |val| val + x; // bind a variable to a closure definition

  println!("x: {}, closure: {}", 41, closure(1) );
}
```

sources: rust by practice, the rust book

RUST :: CLOSURES :: EXAMPLES

```
fn main() {
  let x = 41;

  fn add_one_v1 (x: u32) -> u32 { x + 1 }
  let add_one_v2 = |x: u32| -> u32 { x + 1 };
  let add_one_v3 = |x|             { x + 1 };   // has to be evaluated to compile
  let add_one_v4 = |x|               x + 1 ;    // has to be evaluated to compile

  println!("{}, {}, {}, {}", add_one_v1(x), add_one_v2(x), add_one_v3(x), add_one_v4(x));
}
```

Difference between

- A function definition
- A fully annotated closure
- A closure without the optional type annotations
- A closure without the optional type annotations and brackets

sources: rust by practice, the rust book

RUST :: CLOSURES :: EXAMPLES

```
fn main() {
  let example_closure = |x| x;

  let s = example_closure(String::from("hello"));
  let n = example_closure(5);
}
```

Does this work?

- No, since the first time we call the closure with the string type, the compiler will infer the type of x and the return type

This would work:

```
fn main() {
  let example_closure = |x| x;

  let s = example_closure(String::from("hello"));
}
```

sources: rust by practice, the rust book

RUST :: CLOSURES :: BORROWING IMMUTABLY

```
fn main() {
   let list = vec![1, 2, 3];
   println!("Before defining closure: {:?}", list);

   let only_borrows = || println!("From closure: {:?}", list);

   println!("Before calling closure: {:?}", list);

   only_borrows();

   println!("After calling closure: {:?}", list);
}
```

Does this work?

- Yes

RUST :: CLOSURES :: BORROWING MUTABLY

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    println!("Before calling closure: {:?}", list);

    borrows_mutably();

    println!("After calling closure: {:?}", list);
}
```

Does this work?

- No
- First borrow occurs in when defining the closure, afterwards we use immutable borrow

sources: rust by practice, the rust book

RUST :: CLOSURES :: BORROWING MUTABLY

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    println!("Before calling closure: {:?}", list); // Does not compile

    borrows_mutably();

    println!("After calling closure: {:?}", list);
}
```

Does this work?

- No
- First borrow occurs in when defining the closure, afterwards we use immutable borrow

sources: rust by practice, the rust book

RUST :: CLOSURES :: TAKING OWNERSHIP

Closures can take ownership of the values they use

- Can be done explicitly with the `move` keyword before the parameter list

Example

- Move data so it is owned by the new thread
- Immutable reference to list is passed since it is the least amount of access needed to print it
- We need to move the ownership since the main thread could finish fist and drop the list

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

sources: rust by practice, the rust book

RUST :: RESULT TYPE

Unlike Option, the `Result` type describes possible error instead of possible absence

Result<T, E> can have two outcomes:

- `Ok(T)`: An element T was found
- `Err(E)`: An error was found with element E

By convention, the expected outcome is `Ok` while the unexpected outcome is `Err`.

Since Result is an enum, the possible variants can be matched using the match pattern

Many associated methods with it

- unwrap()
  - In case of success, it takes out the value wrapped by Result
  - In case of error, it panics
  - Used only if certain that returned variant will be Ok
- ? Operator
  - Shorthand way to propagate errors or unwrap results
  - Similar to unwrap but instead of panic it returns an error
  - Replaces an entire match statement

RUST :: RESULT TYPE

Divide Function

- Check if **y** is zero and if so return an Error
- Otherwise return **x/y**

Main function

- Use match pattern to check if the result is valid
- Otherwise print the returned error

```
fn divide(x: f32, y: f32) -> Result<f32, &'static str> {
    if y == 0.0 {
        return Err("Division by zero");
    }
    Ok(x/y)
}

fn main() {
    let result = divide(84.0, 2.0);

    match result {
        Ok(val) => println!("Result is: {}", val),
        Err(msg) => println!("Error: {}", msg),
    }
}
```

The result can also be “unwrapped”

- `unwrap()` method takes out the value wrapper inside `Ok(T)` in case of success, otherwise it panics
- Can be used when you are certain that the return value will be `Ok`
- There is also `expect()` method that can be used to provide a custom error message
- Both unwrap and expect work on Result<T, E> and on Option<T> types

RUST :: RESULT TYPE :: ? OPERATOR

Shorthand way to propagate error or unwrap Ok results

- Similar to `unwrap()` but instead of panicking returns an error
- Replaces entire match statement and can be used in the main function
- unwraps valid values or returns erroneous values, propagating them to the calling function

Example

```rust
use std::num::ParseIntError;

fn main() -> Result<(), ParseIntError> {
    let number_str = "10";

    let number = match number_str.parse::<i32>() {
        Ok(number) => number,
        Err(e) => return Err(e),
    };

    println!("{}", number);
    Ok(())
}
```

```rust
use std::num::ParseIntError;

fn main() -> Result<(), ParseIntError> {
    let number_str = "10";

    let number = number_str.parse::<i32>?;

    println!("{}", number);
    Ok(())
}
```

RUST :: ITERATOR PATTERN

The iterator pattern allows you to perform some task on a sequence of items in turn

An iterator is responsible for the logic of iterating over each item and determining when the sequence has finished

```
let a = vec![1, 2, 3, 4, 5, 6];

for x in a {
   println!("{}", x);
}
```

```
let a = vec![1, 2, 3, 4, 5, 6];

for x in a.into_iter() {
   println!("{}", x);
}
```

By default, `into_iter()` is applied to the collection (left and right are the same)

- Converts a collection into an iterator
- Iterator becomes the owner of the data
- There are three variants: `iter()`, `into_iter()`, and `iter_mut()`
- Range notation (a..b) is also an iterator, e.g., 0..100

https://doc.rust-lang.org/book/ch13-02-iterators.html, https://doc.rust-lang.org/std/iter/trait.Iterator.html

RUST :: ITERATOR PATTERN :: NEXT METHOD

Iterators implement a Trait called Iterator, defined in the standard library

- "Type Item" and "Self::Item" are defining an associated type with this trait
- Relates to the return type of the next method

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // ...
}
```

The next() method is the only required method for iterators

- It is wrapped in an Some, and
  when iteration is over returns None
- Can be called directly

```
let v1 = vec![1, 2, 3];

let mut v1_iter = v1.iter();

assert_eq!(v1_iter.next(), Some(&1));
assert_eq!(v1_iter.next(), Some(&2));
assert_eq!(v1_iter.next(), Some(&3));
assert_eq!(v1_iter.next(), None);
```

Note that we needed to make v1_iter mutable

https://doc.rust-lang.org/book/ch13-02-iterators.html

RUST :: ITERATOR PATTERN

into_iter()

- Consumes the collection, afterwards it is no longer available for reuse because of its ownership has been moved within the loop

```
let a = vec![1, 2, 3, 4];

for x in a.into_iter() {
   println!("{}", x);
}
```

iter()

- Borrows each element of the collection through each iteration, thus leaving the collection untouched and available for reuse after the loop

```
let a = vec![1, 2, 3, 4];

for x in a.iter() {
   println!("{}", x);
}
```

iter_mut()

- Mutably borrows each element of the collection, allowing the collection to be modified in place

```
let mut a = vec![1, 2, 3, 4];

for x in a.iter_mut() {
   *x=10;
   println!("{}", x);
}
println!("{}", a[0]);
```

https://doc.rust-lang.org/book/ch13-02-iterators.html

RUST :: ITERATORS :: METHODS

There are other iterator-specific methods

- `last()` – returns the last element
- `advance_by(..., n: usize)` – advances the iterator by n elements
- `nth(..., n: usize)` – returns the nth element of the iterator
- `step_by(..., step: usize)` – allows stepping by the given amount at each iteration
- `chain(...)` – a new iterator from two iterators, iterating first over the first, then over the second one

```
let a1 = [1, 2, 3];
let a2 = [4, 5, 6];

let mut iter = a1.iter().chain(a2.iter());
```

- `map(...)` – transforms one iterator into another, by means of its argument (e.g., a closure – see next slide)
- `filter(...)` – creates an iterator which uses a closure to determine if an element should be yielded
- `filter_map(...)` – creates an iterator that both filters and maps
- `enumerate(…)` – creates an iterator that gives the current iteration count as well as the next value
- `take()`, `take_while()`, `map_while()`, `skip()`, `scan()`, `flat_map()`, `rev()`,
- `reduce()`, `count()`, `fold()`, `find()`, `min()`, `max()`, `sum()`, `...`

https://doc.rust-lang.org/std/iter/trait.Iterator.html

RUST :: COLLECTIONS, ITERTORS, EXAMPLES

Using closures with vectors and iterators

- Counting occurrences using “count()”
  - Note that count() is an "adapter" tat calls the next() method to use up the iterator

```
fn main() {
   let a = vec![1, 2, 2, 4];

   let count_twos = a.iter().filter(|&x| *x == 2).count();

   println!("{}", count_twos);
}
```

- Using iterator adapters to change one iterator into another

```
fn main() {
   let v1 = vec![1, 2, 3];

   let v2: Vec<i32> = v1.iter().map( |x| x + 1).collect();

   println!("{:?}", v2);
}
```

RUST :: PARSING COMMAND-LINE ARGUMENTS

To read the command-line arguments we can use the standard library

`std::env::args()`

- Returns an iterator of the given arguments

We can use all iterator-specific methods on it, examples:

```
fn main() {
   let args: Vec<String> = std::env::args().collect();

   println!("The name of the binary is: {}", args[0]);

   println!("The first argument is: {}", args[1]); // will panic at runtime if there is no argument
}
```

You can also get nth argument directly

- We can use "expect" to provide a custom error message if error occurs at runtime

```
fn main() {
    let second_arg = std::env::args().nth(2).expect("no argument given");
    println!("The second argument is: {}", second_arg);
}
```

RUST :: SMART POINTERS

A pointer is a general concept for a variable that contains an address in memory

- Most common pointer in Rust is a reference

Smart pointers act as pointers but have additional metadata and capabilities/guarantees

- They own the data they point to
- They implement Drop trait for customizing what happens when the smart pointer goes out of scope
- They implement Deref trait allowing an instance of the smart pointer to behave like a reference
- Note that String, which has capacity as metadata and uses UTF-8 and Vec<T> are smart pointers

Common smart pointers in Rust

- `Box<T>` - for allocating values on the heap
- `Rc<T>` - a reference counting type that enables multiple ownership
- `Ref<T>` and `RefMut<T>`, accessed through `RefCell<T>` - a type that enforces the borrowing rules at runtime instead of compile time

RUST :: UNSAFE RUST

Drops some safety guarantees for performance and flexibility

Useful when you want to go arround some of the restrictions imposed by the compiler

You can mix unsafe code, with safe code with the `unsafe` keyword

Unsafe Rust allows

- Dereferencing a raw pointer
- Calling an unsafe function or method
- Accessing or modifying a mutable static variable
- Implementing an unsafe trait
- Accessing fields of unions

Borrow checker still works, but the memory safety is not guaranteed and safety is programmers concern

RESOURCES, REFERENCES AND LINKS

1. **Resources at rust-lang.org**
    1. The Book: https://doc.rust-lang.org/book/
    2. Rust by example: https://doc.rust-lang.org/rust-by-example/
    3. Course: https://github.com/rust-lang/rustlings/
2. **The Rust Reference: https://doc.rust-lang.org/beta/reference/**
3. **https://learn.microsoft.com/en-us/training/paths/rust-first-steps/**
4. **https://dhghomon.github.io/easy_rust/Chapter_12.html**
5. **https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/primitive-types.html**
6. **https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/unsized-types.html**